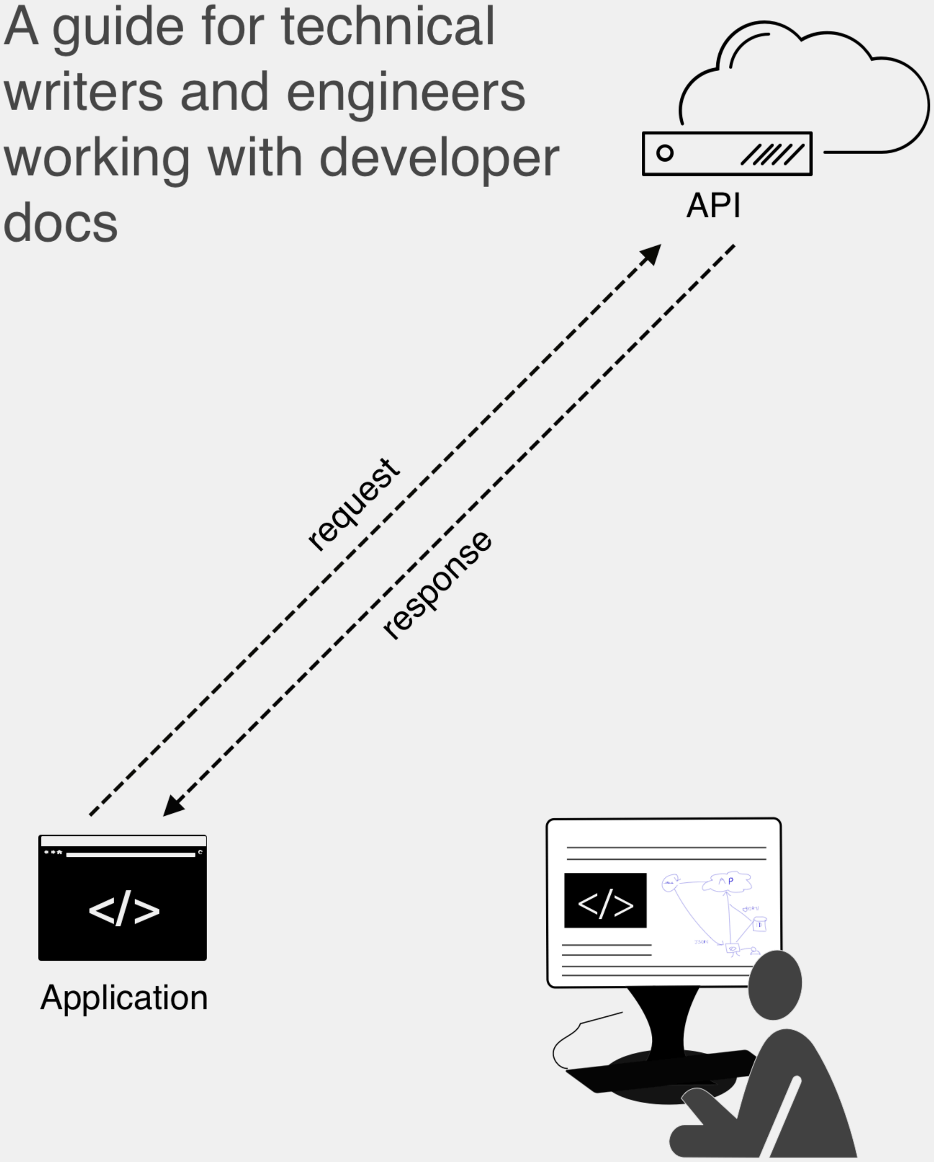
Documenting APIs

A guide for technical writers and engineers working with developer docs



By Tom Johnson

idratherbewriting.com

Copyright 2023. All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, contact Tom Johnson at tomjoht@gmail.com.

Table of Contents

Chapter 1: Introduction to REST APIs 1

Course Overview.......... 2

What's new 9

Workshop video recordings.......... 11

Download PDFs 12

Workshop agenda, slides, activities 13

Why I developed this course 18

About the author 20

Introduction to REST API documentation.......... 22

What is a REST API? 36

Activity: Identify your goals.......... 43

Developer Documentation Trends: Survey Results 44

Glossary 55

Chapter 2: Using an API like a developer 64

Scenario for using a weather API 65

Get authorization keys 69

Submit requests through Postman.......... 71

curl intro and installation.......... 78

Make a curl call.......... 80

Understand curl more 83

Activity: Use methods with curl 88

Analyze the JSON response 94

Inspect the JSON from the response payload 98

Access and print a specific JSON value.......... 103

Dive into dot notation.......... 109

Chapter 3: Documenting API endpoints 114

A new endpoint to document 115

API reference tutorial overview 118

Step 1: Resource description 120

Step 2: Endpoints and methods 126

Step 3: Parameters 132

Step 4: Request example 141

Step 5: Response example and schema 154
Putting it all together 169
Activity: What's wrong with this API reference topic 173
Activity: Evaluate API reference docs for core elements 177

Chapter 4: OpenAPI spec and generated reference docs ..179

Overview of REST API specification formats 180
Introduction to the OpenAPI specification 181
Stoplight: Visual modeling tools for creating your spec 195
Getting started tutorial: Using Stoplight Studio to create an OpenAPI specification document 201
Swagger UI tutorial 218
SwaggerHub introduction and tutorial 230
Swagger UI Demo 240
Integrating Swagger UI with the rest of your docs 241
Redocly tutorial -- authoring and publishing API docs with Redocly's command-line tools 249
Redoc Community Edition Basic Demo 268

Chapter 5: Step-by-step OpenAPI code tutorial 271

OpenAPI tutorial using Swagger Editor and Swagger UI: Overview 272
Working in YAML 276
Step 1: The openapi object 282
Step 2: The info object 285
Step 3: The servers object 288
Step 4: The paths object 291
Step 5: The components object 302
Step 6: security object 327
Step 7: The tags object 334
Step 8: The externalDocs object 337
Activity: Create an OpenAPI specification document 340

Chapter 6: Testing API docs 342

Overview of testing your docs 343
Set up a test environment 345
Test all instructions yourself 348
Test your assumptions 353
Activity: Test your project's documentation 357

Chapter 7: Conceptual topics in API docs 358
API conceptual topics overview 359
API product overviews 360
API getting started tutorials 379
API authentication and authorization 393
API status and error codes 402
API rate limiting and thresholds 408
API quick reference 412
API glossary 418
API best practices 426
Activity: Complete the SendGrid Getting Started tutorial 429
Activity: Assess the conceptual content in your project 430

Chapter 8: Code tutorials 432
Common characteristics of code tutorials 433
Why documenting code is so difficult 434
What research tells us about documenting code 438
Five strategies for documenting code 451
Code samples 464
Sample apps 476
SDKs (software development kits) 480
API design and usability 488
Developer experience (DevX) usability 497

Chapter 9: The writing process 501
Overview of the writing process 502
1. Planning 504
2. Information gathering 507
3. Writing 513
4. Reviewing 519
5. Publishing 527

Chapter 10: Publishing API docs 530
Overview for publishing API docs 531
Survey of API doc sites 536
Design patterns with API doc sites 540
Docs-as-code tools 551

More about Markdown 556
Version control systems (e.g., Git) 562
Activity: Manage content in a GitHub wiki 569
Activity: Use the GitHub Desktop Client 577
Activity: Pull request workflows through GitHub 586
Static site generators 591
Hosting and deployment options 602
Hybrid documentation systems 607
Using Oxygen XML with docs-as-code workflows 612
Blobr: An API portal that arranges your API's use cases as individual products 626
Which tool to choose for API docs — my recommendations 633
Jekyll and CloudCannon continuous deployment tutorial 637
Case study: Switching tools to docs-as-code 645
Tools FAQ 657

Chapter 11: Thriving in the API doc space 659

The job market for API technical writers 660
How much code do you need to know? 664
Best locations for API documentation jobs 674
Activity: Find an Open-Source Project 686
Activity: Create or fix an API reference documentation topic 690

Chapter 12: Native library APIs 691

Overview of native library APIs 692
Get the sample Java project 695
Java crash course 701
Activity: Generate a Javadoc from a sample project 707
Javadoc tags 712
Explore the Javadoc output 720
Make edits to Javadoc tags 722
Doxygen, a document generator mainly for C++ 723
Create non-ref docs with native library APIs 726

Chapter 13: Processes and methodology 727

DX content strategy with developer portals 728
Following agile scrum with documentation projects 733
Managing large documentation projects 737

Managing small documentation requests.......... 750
Managing SDK releases.......... 755
Documentation kickoff meetings and product demos 760
Processes for reviewing documentation 764
Maintaining existing documentation.......... 770
Collecting feedback post-release.......... 775
Managing content from external contributors 782
Changing internal doc culture.......... 785
Sending doc status reports -- a tool for visibility and relationship building 789
Broadcasting your meeting notes to influence a wider audience.......... 794
Ensuring documentation coverage with each software release 798

Chapter 14: Metrics and measurement 800

Measuring documentation quality through user feedback 801
Different approaches for assessing information quality 804
Quality checklist for API documentation.......... 814
Quantifying your progress 825

Chapter 15: AI and API documentation 826

First look at the Oxygen XML AI Positron Assistant 827
AI document engineering with pattern-based prompts.......... 842
Using AI tools to build, stage, and publish API reference docs 851
Using AI to learn coding 856
Using AI for language advice.......... 860
Using AI to create doc updates based on bugs 864
Using AI For thematic analysis 870
Using AI for comparison tasks with API responses.......... 879

Chapter 16: Additional resources..........892

Documenting GraphQL APIs 893
More REST API activities.......... 900
Activity: Get event information using the Eventbrite API 901
Activity: Retrieve a gallery using the Flickr API.......... 908
Activity: Get wind speed using the Aeris Weather API 918
RAML tutorial.......... 924
API Blueprint tutorial.......... 938
API jeopardy answer key 951

What's wrong with this topic answer key 952
Menlo Park API workshop video recording 953
Denver API workshop video recording 954
API doc presentation video recordings 955

Chapter 1: Introduction to REST APIs

REST APIs are flourishing in the marketplace, and the web is becoming a mashup of interconnected APIs. REST APIs consist of requests to and responses from a web server. Job prospects are hot for technical writers who can write developer documentation. This course will help you break into API documentation, especially if you complete the many portfolio-building activities.

Chapter sections:

Course Overview 2
What's new 9
Workshop video recordings 11
Download PDFs 12
Workshop agenda, slides, activities 13
Why I developed this course 18
About the author 20
Introduction to REST API documentation 22
What is a REST API? 36
Activity: Identify your goals 43
Developer Documentation Trends: Survey Results 44
Glossary 55

Documenting APIs: A guide for technical writers and engineers

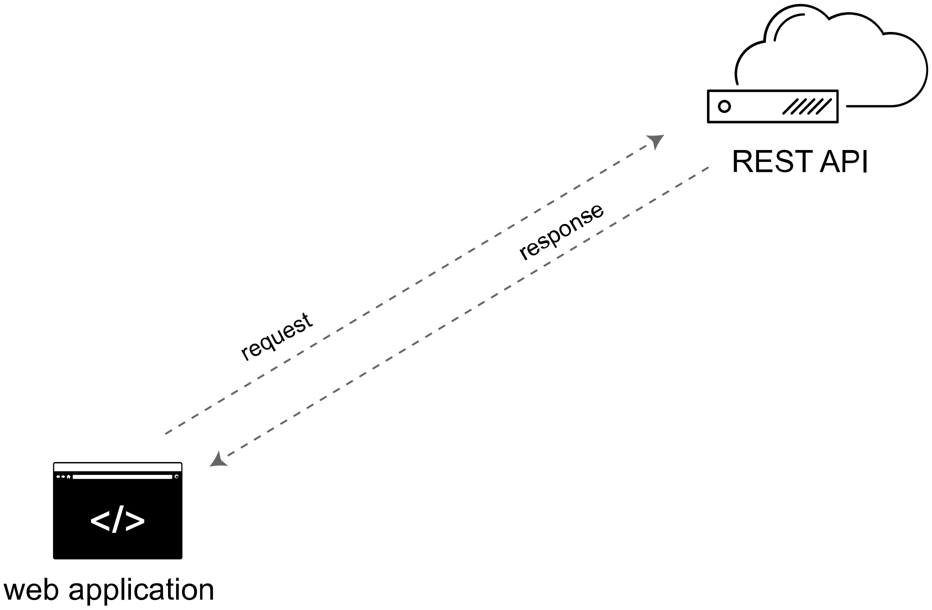
In this course on writing documentation for APIs, instead of just talking about abstract concepts, I contextualize APIs with a direct, hands-on approach. You'll first learn about API documentation by using a simple weather API to put a weather forecast on your site.

As you use the API, you'll learn about endpoints, parameters, data types, authentication, curl, JSON, the command line, Chrome's Developer Console, JavaScript, and more. The idea is that rather than learning about these concepts independent of any context, you learn them by immersing yourself in a real scenario while using an API. Immersion in real scenarios makes these tools and technologies more meaningful.

We'll then transition into standards, tools, and specifications for REST APIs. You'll learn about the required sections in API documentation, analyze examples of REST API documentation from various companies, learn how to join an open-source project to get experience, and more.

About REST APIs

In a nutshell, REST APIs (which are a type of web API) involve requests and responses, not too unlike visiting a web page. You make a request to a resource stored on a server, and the server responds with the requested information. The protocol used to transport the data is HTTP. "REST" stands for Representational State Transfer.



REST APIs involve requests and responses over HTTP protocol

I dive more into the principles of REST in What is a REST API? (p. 36) In REST API documentation, you describe the various endpoints available, their methods, parameters, and other details, and you also document sample responses from the endpoints.

From practice to documentation

In this course, after you practice using an API like a developer (p. 64), you'll then shift perspectives and "become a technical writer" tasked with documenting a new endpoint (p. 114) that engineers added to an API. As a technical writer, you'll tackle each element of a reference topic in REST API documentation:

1. Resource descriptions (p. 120)
2. Endpoints and methods (p. 126)
3. Parameters (p. 132)
4. Request example (p. 141)
5. Response example (p. 154)

Exploring each of these sections will give you a solid understanding of how to document REST APIs. You'll also learn how to document the conceptual sections for an API (p. 358), such as the getting started tutorial (p. 379), product overview (p. 360), status and error codes (p. 402), request authorization (p. 393), and more.

You'll also dive into different ways to publish REST API documentation (p. 530), exploring tools and specifications such as GitHub (p. 569), static site generators like Jekyll (p. 637), and other docs-as-code approaches (p. 551). You'll learn how to leverage templates, build interactive API consoles so users can try out requests and see responses, and learn how to manage your content through version control (p. 562).

We'll also dive into specifications such as the OpenAPI specification (p. 272) and Swagger UI (p. 181) (which provides tooling for the OpenAPI specification). Additionally, you'll learn how to document native library APIs (p. 691) and generate Javadoc (p. 707).

Tech writing is about more than tools and technology, though. As such, I even explore processes and methodologies (p. 727) in depth, explaining how to manage both large (p. 737) and small (p. 750) documentation projects, how to collect documentation feedback (p. 775), how to keep stakeholders happy and updated (p. 789), and more.

I've also included a section on metrics and measurement (p. 801), which lists a comprehensive quality checklist (p. 814). The criteria described can help you evaluate your own developer portal in myriad ways against industry best practices.

Throughout this course, I put these concepts in real, applicable contexts with hands-on activities and demos.

Who the course is for

The course primarily serves the following audiences:

- Professional technical writers looking to transition from traditional documentation into more API-focused documentation for developers.
- Students learning how to prepare themselves technically to succeed in the tech comm field, which is becoming more focused on developer documentation.
- Developers who are documenting their own APIs and want to know best practices for structure, terminology, and style with tech docs.
- Educators who are training technical writing students about API documentation practices and technologies.

Course organization

The course consists of the following sections:

- I: Introduction to REST APIs (p. 1)
- II: Using an API like a developer (p. 64)
- III: Documenting API endpoints (p. 114)
- IV: OpenAPI spec and generated reference docs (p. 179)
- V: Step-by-step OpenAPI code tutorial (p. 271)
- VI: Testing API docs (p. 342)
- VII: Conceptual topics in API docs (p. 358)
- VIII: Code tutorials (p. 432)
- IX: The writing process (p. 501)
- X: Publishing API docs (p. 530)
- XI: Thriving in the API doc space (p. 659)
- XII: Native library APIs (p. 691)
- XIII: Processes and methodology (p. 727)
- XIV: Metrics and measurement (p. 800)
- XV: Additional resources (p. 892)

Sequence and activities

You don't have to read the sections in order — feel free to skip around as you prefer. Some of the earlier sections (such as the section on Using a REST API like a developer (p. 64) and Documenting endpoints (p. 114) follow a somewhat sequential order with the same weather API scenario (p. 65), but by and large you can jump around as desired.

Because the purpose of the course is to help you learn, there are many activities that require hands-on coding and other exercises. Along with the learning activities, there are also conceptual deep dives, but the focus is always on *learning by doing*. Where there are hands-on activities, I typically include this icon in the section title: . Other topics have the word "Activity" in the title. The activities are integrated into various sections, but you can also see many of the activities in the Workshop Activities (p. 13). These are the activities we do during live workshops.

I refer to the content here as a "course" instead of a book or a website, primarily because I include a lot of exercises throughout in each section, and I find that people who want to learn API documentation prefer a more hands-on "course" experience. Below each topic is a progress indicator that reminds you to keep moving through the course.

How long will it take to finish the course?

Don't be daunted if you never make it entirely through the course. If printed to PDF, the content is about 900 pages. And I keep adding to the content, refining it, rearranging and adjusting it as I see fit and as I grow my experience and awareness. It's a living document. Almost no one really finishes the course from beginning to end because by the time they do, I'll have added new content.

Also, you should only focus on those sections that you need to learn. See this Reddit post, Has anyone here completed the idratherbewriting API writing course by Tom Johnson? ↗. For someone already familiar with many concepts, they can breeze through the course in less than a week. For someone new to API documentation, they might spend many months making their way through the course. If you keep finding value in the content, take as long as you want. If you don't find yourself learning anything, skip forward.

Will this course help you get a job in API documentation?

The most common reason people take this course is to transition into API documentation. This course will help you make that transition, but you can't just passively read through the content. You need to do the activities outlined in each section, especially those topics that involve working with content from an open-source project (p. 686) (or something similar). These activities are crucial to building experience and credibility with a portfolio. I provide more details in Getting an API documentation job and thriving (p. 659). Without a strong portfolio to demonstrate your writing expertise, almost no manager will hire you, even if you have an extensive tech background.

No programming skills required

As for the needed technical background for the course, you don't need any programming background or other prerequisites, but it will help to know some basic HTML, CSS, and JavaScript.

If you do have some familiarity with programming concepts, you might speed through some of the sections and jump ahead to the topics you want to learn more about. This course assumes you're a beginner, though.

Some of the code samples in this course use JavaScript. JavaScript may or may not be a language that you actually use when you document REST APIs, but most likely there will be some programming language or platform that becomes important to know.

JavaScript is one of the most useful and easy languages to become familiar with, so it works well in code samples for this introduction to REST API documentation. JavaScript allows you to test code by merely opening it in your browser (rather than compiling it in an IDE). (I have a quick crash-course in JavaScript here ↗ if you need it.)

What you'll need

Here are a few tools you'll need to do the activities in this course:

- **Computer.** You need a computer (if attending the live workshop, a laptop and charging cord), as there are many activities to work through.
- **Text editor**. If you don't already have a favorite text editor, download Sublime Text ↗, as it works well on both Mac and Windows and is free. If you have another text editor you prefer (e.g., Visual Studio Code ↗, Atom ↗, or even Notepad++ ↗), that will work too. Just make sure you can write code in plain text.
- **Chrome browser**. Chrome ↗ provides a Javascript Console that works well for inspecting JSON, so we'll be using Chrome. Also, in order to read JSON responses more easily in the browser, install the JSON Formatter ↗ Chrome extension.
- **Postman**. Postman ↗ is an app that allows you to make requests and see responses through a visual client. *Make sure you download the app and not the Chrome extension.*
- **curl**. curl ↗ is essential for making requests to endpoints from the command line. Mac already has curl built-in, but it might not be available by default on Windows. (Some Windows 10 builds already have it in Powershell.) On Windows, open a Command Prompt and type `curl -V`. If it's not installed, go to confusedbycode.com/curl ↗ and install a version (usually "With Administrator Privileges (free), 64-bit"). Close and re-open your Command Prompt and try typing `curl -V` again.
- **Git**. Git ↗ is a version control tool developers often use to collaborate on code. For Windows, see https://gitforwindows.org/ ↗ to set up Git and the Git BASH terminal emulator. For Mac, see Downloading Git ↗.
- **GitHub account**. GitHub ↗ will be used for various activities, sometimes to demonstrate the Git workflow and other times as an authentication service for developer tools. If you don't already

have a GitHub account, sign up for one.
- **Stoplight Studio Editor**. When working with the OpenAPI specification, we'll use the Stoplight Studio Editor. Stoplight Studio provides visual modeling tools for working with the OpenAPI specification. Stoplight offers both a web browser and standalone app versions of the editor. We'll be using the web browser version because it provides more complete functionality (such as trying out requests). Go to https://stoplight.io/studio and log in with GitHub.
- **OpenWeatherMap API key**. We'll be using the OpenWeatherMap API for some exercises. It takes a couple of hours for the OpenWeatherMap API key to become active, so it's best if you get the API key ahead of time — then when you get to the OpenWeatherMap API activities, you'll be all set. To get your (free) OpenWeatherMap API key, go to https://openweathermap.org/. Click Sign Up in the top nav bar and create an account. After you sign up, OpenWeatherMap sends you an API key to your email. you can also find it when you log in and click the API Keys tab from the dashboard. Copy the key into a place you can easily find it.

Testing Your Setup

In the past, people have asked for some tests to check whether their laptops are correctly set up.

- If you want to test whether Postman works, open up the Postman app and paste this into the GET box: `https://api.openweathermap.org/data/2.5/weather?zip=95050&units=imperial&appid=126cac1a482f51de0f1287b45ae2bf9a`. Then click **Send**. If you get a response, it's working correctly. (In rare cases, sometimes people have security restrictions on their computers that block all network access.)
- If you want to test whether curl is installed, open Terminal (on Mac) or Command Prompt (on Windows) and paste in `curl --get "https://api.openweathermap.org/data/2.5/weather?zip=95050&units=imperial&appid=126cac1a482f51de0f1287b45ae2bf9a"`. If you get a JSON response, you're good.
- To check whether Git is installed, open up Terminal (on Mac) or Command Prompt (on Windows) and type `git --version`. If it's installed, you'll see the version.

Video recordings

For video recordings of this course, see the Video recordings of API doc workshops (p. 11). The page lists the most recent video recordings available. The video doesn't go into the same level of detail as the written material, but it would be a good start.

See my Upcoming Presentations on my blog for details about future workshops and presentations.

Course slides

For the live workshops, I have various slides that cover different sections of this course. See Course Slides (p. 13) for the links. The slides use RevealJS, which is an HTML/CSS/JS framework for slides. The images are single-sourced between the site and the slides, so they'll more likely stay in sync.

Copyright and re-use of materials

Some people have asked whether they can use materials from this course to give their own API documentation workshops. I only allow the material to be used for non-profit workshops where the workshop leader isn't charging participants for the instruction. Of course, many of the ideas and concepts in this course aren't specific or unique to me, and you're entitled to fair use. However, please don't just hijack my site's materials, activities, and other information for your own commercial endeavors.

If you're a teacher at a college or university (or high school) and are trying to teach APIs and developer docs to your students, feel free to use this site as part of your instructional materials. I appreciate links back to the site and attribution as needed (links pointing to my site boost search engine rankings). My larger goal for this site is to help educate and empower tech writers with developer documentation strategies, workflows, and best practices. If you'd like to share your API documentation course syllabus, contact me and I'll add a link here.

Is this course really free?

Yes, this course is entirely free. Advertising helps make this content free, so you will see little text snippets and other ads on the course pages promoting links to API service providers. Sometimes I also include sponsored pages as well to highlight a vendor's particular offerings. Whenever content is sponsored, I note that on the page. I also wouldn't include any content in the course that didn't meet my standards or which I wouldn't recommend.

If you'd like to give back, feel free to buy me coffee by clicking the Buy me a coffee button on the bottom of each page or by buying the PDF of the course. I'm grateful to see many people who find enough value in the course to contribute back. Seeing contributions helps me feel in a more immediate way how much the content helps people.

PDF and Kindle/eBook formats

To read the same content in PDF or Kindle/eBook formats, see PDF and eBook formats (p. 12). You can print out the section you want, or load it into a PDF reader to highlight/annotate content. Or if you prefer Kindle or another eReader, download the content in ebook format and load it onto your device.

Let me know if any content is out of date

One of the challenges in any technical course is ensuring the content stays up to date. Technology changes rapidly, and given the many hands-on activities in the course, it's easy for some steps to become out of date as time passes. I've tried to maintain a healthy balance between general and specific details in the content here. If you find something is out of date, either add a comment on that page or let me know .

Stay updated

If you're following this course, you most likely want to learn more about APIs. I publish regular articles that talk about APIs and strategies for documenting them. You can stay updated about these posts by subscribing to my free newsletter through the form below. My newsletter, branded as the I'd Rather Be Writing newsletter, helps you stay updated about everything going on in the techcomm world, with a special emphasis on API documentation, docs-as-code tools, and lately AI.

Other resources

If you've looking for additional resources to learn API documentation as well as tech writing best practices, check out the following:

- Cherryleaf's API documentation training course, by Ellis Pratt. (See a preview here.)
- Peter Gruenbaum's API doc courses on Udemy, by Peter Gruenbaum of SDK Bridge. (He has multiple Udemy courses on API docs, Swagger, Git, and more.)
- Docs Like Code, by Anne Gentle

- The Design of Web APIs, by Arnaud Lauret
- Docs for Developers: An Engineer's Field Guide to Technical Writing, Authors: Bhatti, J., Corleissen, Z.S., Lambourne, J., Nunez, D., Waterhouse, H.
- Standards and Guidelines for API Documentation: For Technical Writers, Software Developers, Information and Software Architects, by Anne Tarnoruder

These resources aren't API-specific but are still relevant:

- Modern Technical Writing: An Introduction to Software Documentation, by Andrew Etter
- Documentation Guide – Write the Docs Community
- Every Page Is Page One: Topic-Based Writing for Technical Communication and the Web, by Mark Baker
- The Product is Docs: Writing technical documentation in a product development group, by Christopher Gales and Splunk Documentation Team
- The Good Docs Project: Best practice templates and writing instructions for documenting open source software
- Google's technical writing courses for engineers

For even more books, see Relevant Books from the Write the Docs site.

What's new

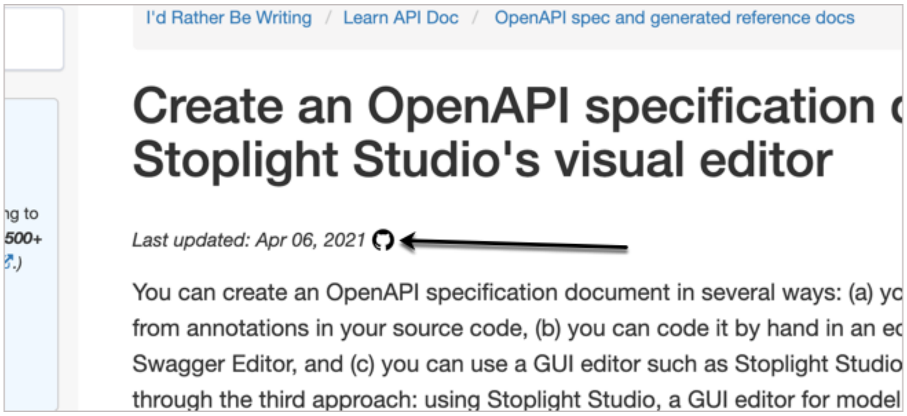
If you're looking to see what's new in the API doc site/course, you can browse new content in several ways.

Most recent updates

See the api-doc-site-updates category on my blog. I describe the updates I'm making here, often with commentary about why I made the update.

Seeing what content has been updated

To see the history of page, click the GitHub button next to the timestamp on the page:



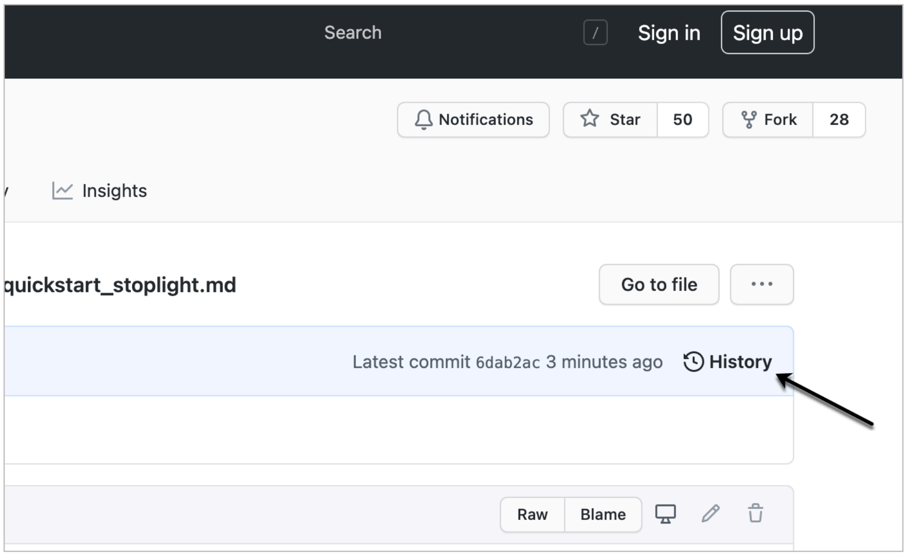
I'd Rather Be Writing / Learn API Doc / OpenAPI spec and generated reference docs

Create an OpenAPI specification
Stoplight Studio's visual editor

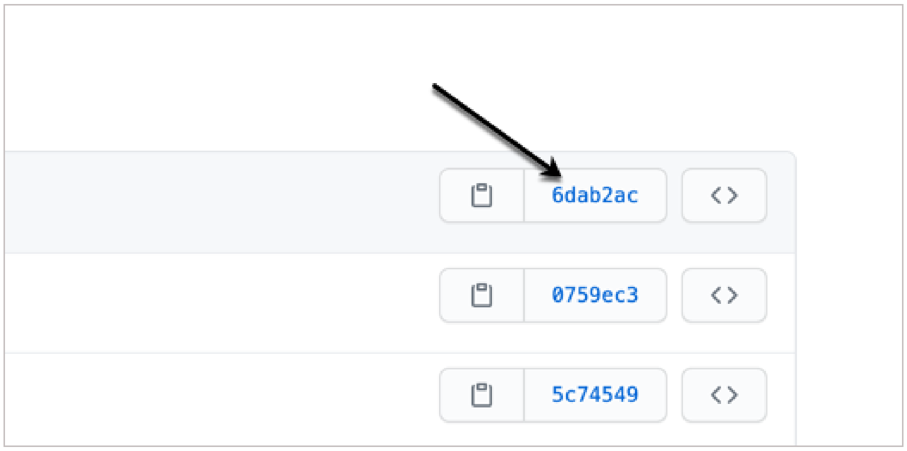
Last updated: Apr 06, 2021

You can create an OpenAPI specification document in several ways: (a) y
from annotations in your source code, (b) you can code it by hand in an e
Swagger Editor, and (c) you can use a GUI editor such as Stoplight Studio
through the third approach: using Stoplight Studio, a GUI editor for model

This takes you to the file in GitHub. In GitHub, click the History link:



Then click the commit IDs to browse file diffs for the commits that include the file:



You can see the stream of commits in GitHub here.

Video recordings of API doc workshops

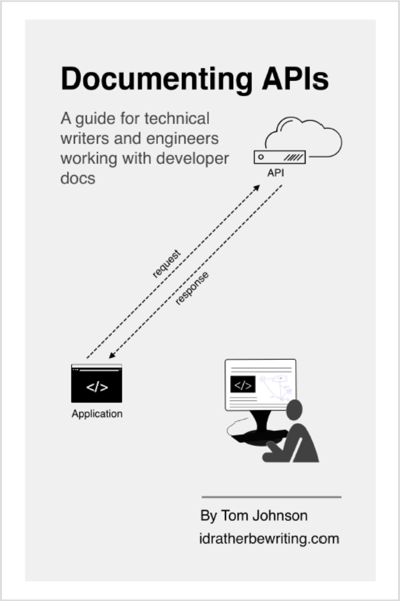
I’ve given numerous presentations and workshops on API documentation, and I’ve recorded some of them. The most recent recordings are available below. I divided this all-day workshop into multiple separate videos. Keep in mind that my API content evolves, so some older presentations might no longer match the course content.

Note: This content doesn’t embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_course_videos.html] to view the content.

Download PDFs

In addition to reading on the web, you can also can read this content in PDF format. The PDF version (total pages are 900+) is current with the web version.

Documenting APIs: A guide for technical writers and engineers writing API documentation



Documenting APIs

A guide for technical writers and engineers working with developer docs

By Tom Johnson
idratherbewriting.com

The date the pages were last generated appears in the header of the PDF. Payment is handled through the buymeacoffee.com platform.

Workshop -- agenda, slides, activities

The workshop agenda, slides, and activities correspond to full-day API workshop. The slides and activities mirror similar sections in the course.

Note that for API workshops, it helps to consolidate activities into a single page with brief instructions. The content for the activities below is the same content that appears in other parts of the course — it's just pulled in here (single-sourced) for convenience. Workshops require a healthy amount of hands-on activities to be engaging. If you have feedback about how to improve the activities, or places where you keep getting stuck, let me know.

Not all activities in this course are consolidated here, since participants can only do so much during a workshop. The following activities are those I've selected for workshops.

8:00 - 9:00am: Registration + breakfast

Doors open at 8:00am. Check your name off at the registration table and get a name tag. Light breakfast (coffee and pastries) will be available. Find a table and get situated and acquainted with others. If you didn't finish all the pre-workshop tasks as described in What You'll Need (p. 5), do that now.

9:00 - 9:30am: Intro to API documentation

Section: Introduction to API documentation (p. 1)

INTRO TO API DOCUMENTATION

By Tom Johnson / @tomjohnson
idratherbewriting.com

Slides available at
idratherbewriting.com/learnapidoc/slides/intro-api-documentation/

1

Activity 1: Identify your goals (p. 43)

9:30 - 10:15am: Using an API like a Developer

Section: Using an API like a developer (p. 64)

USING AN API LIKE A DEVELOPER

By Tom Johnson / @tomjohnson
idratherbewriting.com

Slides available at
idratherbewriting.com/learnapidoc/slides/using-api-like-developer/

Activity: Explore OpenWeatherMap API (p. 66)

Activity: Get an OpenWeatherMap API key (p. 69)

Activity: Make requests with Postman (p. 71)

Activity: Make the OpenWeatherAPI request using curl (p. 80)

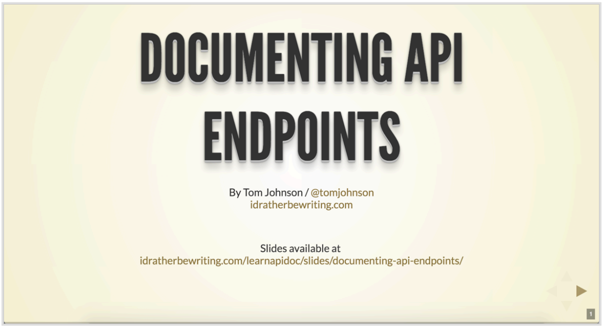
Activity: Make an API request on a web page (p. 98)

10:15 - 10:30am: Break

Break time

10:30 - 11:30pm: API endpoints

Section: Documenting API endpoints (p. 114)



DOCUMENTING API ENDPOINTS

By Tom Johnson / @tomjohnson
idratherbewriting.com

Slides available at
idratherbewriting.com/learnapidoc/slides/documenting-api-endpoints/

Activity: What’s wrong with this API reference topic (p. 173)

Activity: Evaluate API ref docs to identify core elements (p. 177)

11:30 - 12:30: OpenAPI and Swagger

Section: OpenAPI and Swagger (p. 179)

OPENAPI AND SWAGGER

By Tom Johnson / @tomjohnson
idratherbewriting.com

Slides available at
idratherbewriting.com/openapi-and-swagger

Activity: Explore Swagger UI through the Petstore Demo (p. 188)

Activity: Create an OpenAPI specification using Stoplight Studio (p. 201)

12:30 - 1:30pm: Lunch

Lunch provided through catering.

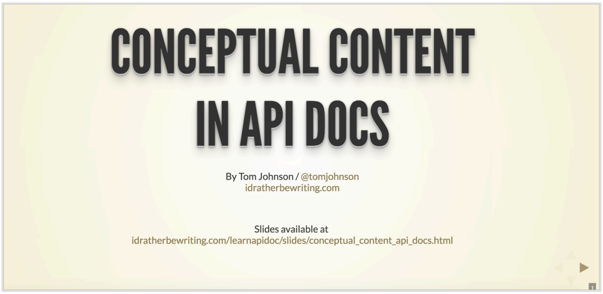
1:30 - 2:00pm: OpenAPI and Swagger (continued)

Activity: Use Redoc Community Edition to render OpenAPI spec (p. 268)

Activity: Create a Swagger UI display with an OpenAPI spec document (p. 222)

2:00 - 2:30pm: Conceptual topics

Section: Conceptual topics (p. 358)



CONCEPTUAL CONTENT IN API DOCS

By Tom Johnson / @tomjohnson
idratherbewriting.com

Slides available at
idratherbewriting.com/learnapidoc/slides/conceptual_content_api_docs.html

Activity: Complete the SendGrid Getting Started tutorial (p. 429)

Activity: Judge conceptual content and decide which is best (p. 430)

2:30 - 2:45pm: Break

Break. Snacks provided.

2:45 - 3:30pm: Code tutorials

Section: Code tutorials (p. 432)

CODE TUTORIALS

By Tom Johnson / @tomjohnson
idratherbewriting.com

Slides available at
idratherbewriting.com/learnapidoc/slides/code_tutorials.html

Activity: Analyze two code tutorials (p. 433)

3:30 - 4:15pm: Publishing API docs

Section: Publishing API Documentation (p. 530)

PUBLISHING API DOCUMENTATION

By Tom Johnson / @tomjohnson
idratherbewriting.com

Slides available at
idratherbewriting.com/publishing-api-docs

Activity: Create a GitHub wiki and publish content on a sample page (p. 571)

Activity: Clone your GitHub repo locally (p. 572)

Activity: Push local changes to the remote (p. 574)

4:00 - 4:30: Participant’s challenges surfaced and discussed

During this time, I’d like to have participants surface specific challenges that they are facing and address them as a whole.

4:30 - 5:00pm: Thriving in the API doc space

Section: Thriving in the API doc space (p. 659)

THRIVING IN THE API DOC SPACE

By Tom Johnson / @tomjohnson
idratherbewriting.com

Slides available at
idratherbewriting.com/learnapidoc/slides/thriving_in_api_docs

Conclusion

Note: This content doesn't embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/workshop.html] to view the content.

5:00 - 5:30pm: Individual consulting

The general workshop ends and we transition into any individual consulting as desired. If you have specific questions not addressed during the workshop, let's chat specifically about them. For all those interested, I'll write your name on then board and then just meet with you individually for about 5 minutes each until everyone's questions are answered.

Why I developed this course

I initially compiled this material to teach a series of workshops to a local tech writing firm in the San Francisco Bay area. They felt they either needed to train their existing technical writers on how to document APIs, or they would need to let some of their writers go. I taught a series of three workshops delivered in the evenings, spread over several weeks.

These workshops were fast-paced and introduced the writers to a host of new tools, technologies, and workflows. Even for writers who had been working in the tech comm field for 20 years, API documentation presented new challenges and concepts. The tech landscape is so vast, even for writers who had detailed knowledge of one technology, their tech background didn’t always carry over into REST API documentation.

After the workshops, I put the material on my site, idratherbewriting.com, and opened it up to the broader world of technical writers. I did this for several reasons. First, I felt the information would be useful to the tech writing community. There are very few books or courses that dive into API documentation strategies for technical writers.

Second, I knew that through feedback, I could refine the information and make it better. Almost no content hits the mark on its first release. Instead, content needs to iterate a while through user testing and feedback. Just as this iterative review helps refine user documentation, the same principle applies to course material as well. I’ve given dozens of presentations and workshops on API documentation for several years now, and each time I’ve used the feedback to improve this content.

Finally, the content would help drive traffic to my site. In fact, visits to the API documentation course pages outperform visits to my blog. I reflected on this traffic source in a blog post — see If writing is no longer a marketable skill, what is? How would people discover the material if they couldn’t find it online? If the material were only trapped in a print book or behind a firewall, it would be difficult to discover. Content is a rich information asset that draws traffic to any site. It’s what people primarily search for online.

After putting the API doc on my site for some months, the feedback was positive. One person said:

> Tom, this course is great. I’m only part way through it, but it already helped me get a job by appearing fluent in APIs during an interview. Thanks for doing this. I can’t imagine how many volunteer hours you’ve put into helping the technical communication community here.

Another person commented:

> Hi Tom, I went through the whole course. Its highly valuable and I learned a bunch of things that I am already applying to real world documentation projects. … I think for sure the most valuable thing about your course is the clear step by step procedural stuff that gives the reader hands-on examples to follow (its so great to follow a course by an actual tech. writer!)

And another:

> I love this course (I may have already posted that)—it's the best resource I have come across, explained in terms I understand. I've used it as a basis for my style guide and my API documentation….

These comments inspired me to continue adding to the course, building out more tutorials, sections, and refinements. What began as a simple three-session course transformed into a larger endeavor, and I aspired to convert the content into a full-fledged book and multi-week course. I continue to receive emails from technical writers, many of whom are trying to transition into developer documentation. The other week someone wrote to me:

> Just an email to thank you for the wonderful API course on your site. I am a long-time tech writer for online help that was recently assigned a task to document a public API. I had no experience in the subject, but had to complete a plan within a single sprint. Luckily I remembered from your blog posts over the years that you had posted material about this.
>
> Your course on YouTube gave me enough information and understanding to be able to speak intelligently on the subject with developers in a short timeframe, and to dive into tools and publishing solutions.

And another:

> I am nearly in tears after finding this site! I think I stumbled upon it some time ago, but I must not have been ready for what you have to say. NOW I am ready! As a former technical writer now knowledge manager, I stumbled upon API writing and have learned a lot simply by being curious and observant…

Of course, not all comments or emails are praiseworthy. Some people note problems on pages, such as broken links or broken code, unclear areas or missing information. As much as possible, after receiving this feedback, I go back and clarify or strengthen those areas.

One question I faced in preparing the content is whether I should stick with text, or combine the text with video. While video can be helpful at times, it's too cumbersome to update. Given the fast-paced, rapidly evolving nature of the technical content, videos get out of date quickly.

Additionally, videos force the user to go at the pace of the narrator. If your skill level matches the narrator, that's great. But in my experience, videos often go too slow or too fast. In contrast, text lets you more easily skip ahead when you already know the material, or slow down when you need more time to absorb concepts.

Despite the constant changes in the technology landscape, I want to keep this course current and up to date. As such, I'll continue to add and edit and refine it as needed. I want this content to become a vital learning resource for all technical writers, both now and in the years to come as technologies evolve. If you have general feedback about this course, feel free to drop me a line ↗.

About the author

In case you'd like to know a little bit about me, I'm currently based in Seattle, Washington, working for Google. (Previously, I was at Amazon and a couple of startups in the Bay area.)

Most people know me through my blog, I'd Rather Be Writing, which has been an active online blog for tech comm for the past decade.

Like most technical writers, I stumbled into technical writing after working in other fields. I first earned a BA in English and an MFA in Literary Nonfiction, and then started my career as a writing teacher. After a stint in teaching, I transitioned into marketing copywriting and then turned to technical writing (mainly for financial reasons).

Despite my initial resistance to the idea of technical writing (I thought it would be boring), I found that I actually liked technical writing — a lot more than copywriting. Technical writing combines my love for writing and my fascination with technology. I get to play with tools and handle all aspects of content production, from design to styles to publishing workflows.

I worked as a traditional technical writer for some years, mostly documenting applications with user interfaces. One day, my organization decided to lay off the tech writing team. After that, and based on my proclivity for tinkering with tools, I decided to steer my career into a tech writing market that was more in demand: developer documentation, particularly API documentation. I also moved to Silicon Valley to be at the center of tech.

I started documenting my first API at a gamification startup and then transitioned to another semi-startup to continue with more API documentation. I was no longer working with applications that had user interfaces, and the audiences for my docs were primarily developers. Developer doc was a new landscape to navigate, with different tools, expectations, goals, and deliverables.

If you want to read more personal details, see My life story, or reflections on what shaped my life's career trajectory.

Although I didn't have a programming background, I've always been somewhat technical. As a teacher, I created my own interactive website. As a traditional technical writer, I often set up or hacked the authoring tools and outputs. I like learning and experimenting with new technologies. The developer documentation landscape suits me well, and I enjoy it.

Still, I'm by no means a programmer. As a technical writer, in-depth technical knowledge is helpful but not always essential, as it tends to be too specialized and comes at the expense of other skills and knowledge. What matters most is the ability to learn something new, across a lot of different domains and products, even if it's challenging at first. And then to articulate the knowledge in easy-to-consume ways. The writing process (p. 502) is still just as relevant when writing API docs as other forms of docs.

You’re probably taking this course because you want to develop your skills and knowledge to increase your capabilities at work, to enhance your skillset’s marketability, or maybe figure out how to document the new API your company is rolling out.

You’re in the right place. By the time you finish this course, you’ll have a solid understanding of how to document APIs. You’ll be familiar with the right tools, approaches, and other techniques you need to be successful with developer documentation projects.

By the way, I keep adding to this course in a Winchester Mystery House ↗ way, which means I keep adding rooms and extra hallways and doors, etc. If you were to print it out, the course would be more than 500 pages long. Few people get through the whole of it, and by the time they do, I’ve usually added a new section. So jump in and read through the topics you find most relevant and interesting. Don’t feel compelled to get through it all.

If you have a question for me, or just want to drop me a line, you can contact me through my Contact page ↗. However, for most questions, you’ll get a much better response by asking them in the Write the Docs Slack ↗. I’m also on Write the Docs Slack @tomjohnson ↗, so feel free to ask me questions there.

Introduction to REST API documentation

Before we dive into the technical aspects of APIs, let's explore the market, general landscape, and trends with API documentation.

Different types of APIs

The API landscape is diverse, with many different types of APIs. Although this course focuses on REST APIs, there are many other types of APIs as well. Often when people start browsing GitHub looking for API projects to join, or when they peruse the various APIs in their own company, they are surprised that the APIs look unfamiliar from the APIs covered in this course. There are many types of APIs you will likely encounter.

One way to sort the different types of APIs is to categorize them into two general buckets: web service APIs versus native library APIs. Web service APIs send and receive messages across the web using HTTP to transport the request and response; web service APIs are language agnostic. Native library APIs, on the other hand, involve incorporating code directly into your project for the desired functionality; native library APIs are language-specific.

The following list describes the most common types of APIs you will encounter:

- **Native library APIs**: Native library APIs, also called "library-based APIs," refer to code libraries (for example, JAR files) that developers add directly to their projects to provide additional functionality through classes or other functions that can be called locally. These APIs are specific to a programming language — e.g., Java, C++, Python, Ruby, .NET, and so on. With native library APIs, the functions are incorporated locally within the code to expand the operations you can perform within your project, usually without requiring you to access resources in the cloud. Native library APIs require you to be familiar with the programming language and tend to be the most challenging type of API to document for technical writers. See Native Library APIs (p. 692) in this course for more information into Java APIs.
- **SOAP APIs**: SOAP (Simple Object Access Protocol) APIs are web services that rely on a strict XML protocol to define the message exchange format for requests and responses. SOAP is common with financial APIs and regulated industries, though it has largely been replaced by REST in popularity. As a standardized protocol, SOAP's XML message format is usually defined through a WSDL (Web Services Description Language) file that specifies the allowed elements and attributes in the message exchange. The WSDL file is machine-readable and used by the servers interacting with each other to facilitate the communication. See SOAP ↗ for introductory details. More detail about SOAP is also provided in What is a REST API? (p. 36)
- **RPC-based APIs**: RPC stands for Remote Procedure Call. RPC-based APIs are web services that call a method on a remote server by delivering an encoded message through HTTP. The encoded message format might be XML for XML-RPC APIs ↗ or JSON for JSON-RPC APIs ↗, but in both cases, the message travels to the remote server via HTTP like other web services. The methods on the remote servers can be in any language. For example, an XML-RPC API can call a Java or Python or C++ method.
- **gRPC APIs**: gRPC APIs are web services similar to RPC-based APIs in that the web service calls a function or runs a procedure on a remote server; however, gRPC uses protocol buffers ↗ (specified in .proto files) rather than XML or JSON as the message exchange format. The protocol buffer lets you define the structure for your data and the way to convert (serialize) the data to be consumed by the receiving server. Protocol buffers are lighter and more efficient than XML.

gRPC APIs were developed by Google and published as an open-source platform. See gRPC APIs for details.

- **REST APIs**: REST (Representational State Transfer) are web services that let you make requests for resources through URL paths. You also supply the operation to be performed with the path (e.g., GET, CREATE, DELETE). As with other web service APIs, the requests and responses travel via HTTP across the web, and the servers receiving the requests are language agnostic about the request (not required to be a specific programming language). Responses are typically returned in JSON or XML formats. REST APIs have many different paths (endpoints) with various parameters you can configure to determine the results you want. This course mainly focuses on REST APIs. See What is a REST API? (p. 36) for details.
- **GraphQL APIs**: GraphQL APIs are web services developed by Facebook that let users dynamically query for results they need through a single path (endpoint). GraphQL eliminates the need for multiple request URLs or other post-filtering on the returned results to get what you need. Your query retrieves only the needed data, allowing the request and response to be fast and specific. See graphql.org for more detail. Also see If I am learning to write developer documentation, should GraphQL be on my radar?.
- **Voice Assistant APIs**: Voice Assistant APIs are used with voice assistants such as Alexa. These APIs originate from the cloud and call an endpoint based on natural language processing of voice commands spoken by users. This is a case where APIs operate behind the scenes in the cloud, and developers create code, such as in a Lambda function (cloud computing), that handles incoming requests sent from the voice assistant API.
- **Internet of Things (IoT) APIs**: IoT APIs are used by physical devices (such as sensors or wearables) that transmit or receive data to connect the device to an online network. For example, a thermostat sensor in a room might transmit the temperature to a central controller (such as with Nest) via an IoT API. For more detail, see App nirvana: When the Internet of Things meets the API economy. See also APIs in the world of IoT.

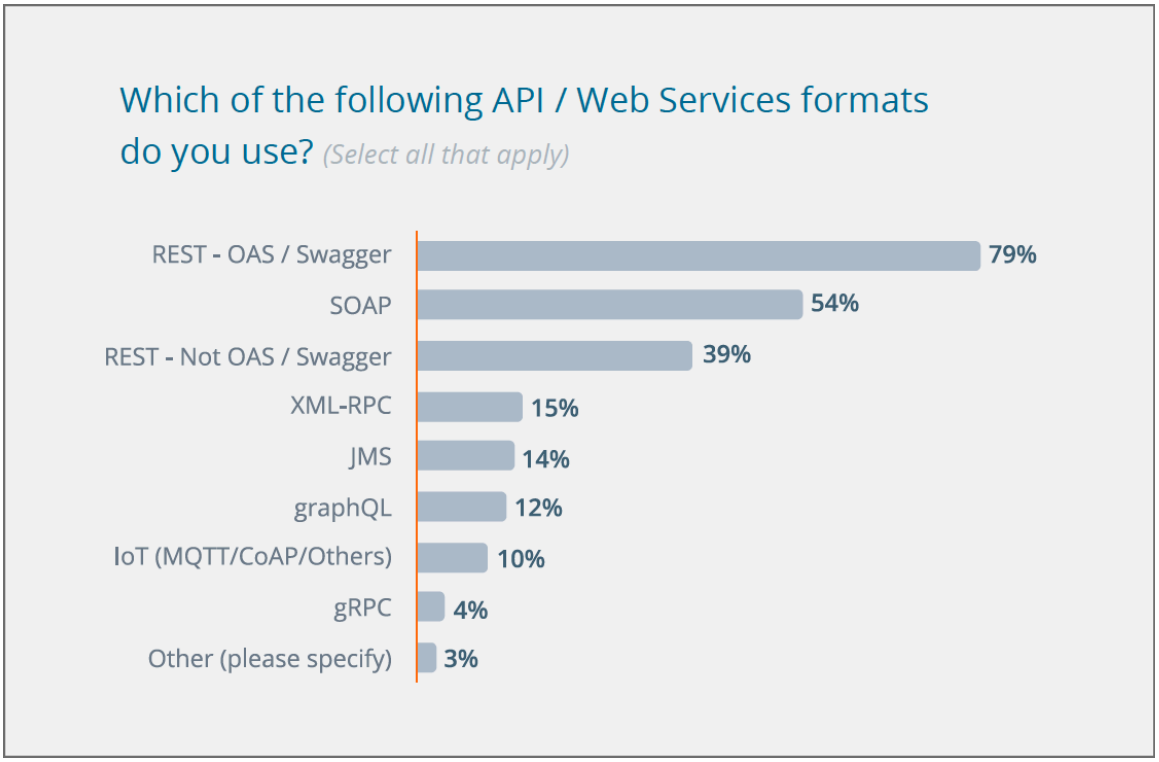
For more reading, see API types by Sarah Maddox. Sarah notes that there are also hardware APIs, object-remoting APIs, web socket APIs, OS functions and routines, and more.

Despite the variety of APIs, the defining characteristic of nearly all developer documentation is that it involves documenting *some type of API*. This is why “API documentation” and “developer documentation” are used somewhat synonymously. APIs make life easier for developers (who are consuming the API) because the APIs perform functions or other tasks in more efficient ways.

Most companies make their information and services available through APIs to help third-parties adopt and implement the company’s information/services. (This is the whole idea of the information economy.) Additionally, many APIs are available only internally to help developers within the same company implement various services. For example, a team handling payment controller operations might provide an API that another team developing the company’s app can implement to handle payment transactions.

Which type of API is most common?

As you browse APIs, you might be wondering which type of API is most common? Which types of APIs should you focus on? Among the web service APIs, *The State of API 2019* report from Smartbear surveyed more than 3,000 technology professionals and found that REST-OAS / Swagger was the most common web service used:



Popularity of web service APIs (The State of API 2019, p.20)

OAS stands for OpenAPI Specification, which is something I explore in-depth in Introduction to the OpenAPI specification (p. 181). The report even mentions some APIs not covered above — JMS APIs, which are used with Java to send messages.

As you can see, when it comes to APIs, one size/type does not fit all. Developers will implement the type of API that best aligns with their scenario and requirements, just as there are different types of cars (sports cars, trucks, semi-trucks, sedans, hearses, etc.) for different trips, drivers, passengers, and roads.

In this course, we'll be diving into REST APIs in depth. Remember that with REST APIs, you don't deliver a library of files to users. Instead, the users make requests for the resources on a web server, and the server returns responses containing the information. Both the system initiating the request and the system providing the response can be in any programming language, so long as they transmit the message via HTTP.

REST APIs follow the same protocol as the web. When you open a browser and type a website URL (such as `https://idratherbewriting.com`), you're actually making a GET request for a resource on a server. The server responds with the content and the browser makes the content visible.

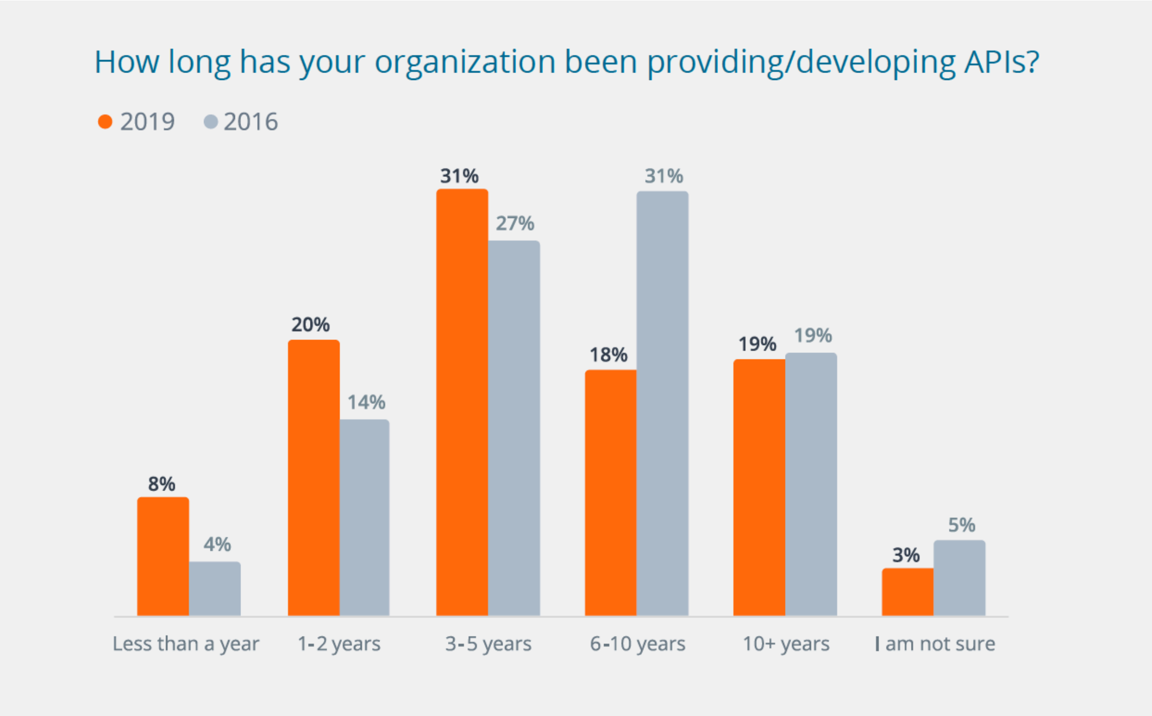
This course not only focuses on REST APIs because they are more popular and in-demand but because they're also more accessible to technical writers. You don't need to know a specific programming language to document REST APIs. And REST is becoming the most common type of API anyway.

Many companies are new to API development

According to *The State of API 2019* report, more and more companies are starting to develop APIs:

> While modern APIs have been used in software development for more than two decades, the last 10 years have been marked by a growth in API adoption. Our survey found that 59% of organizations began developing APIs in the last five years. Furthermore, 28% only began developing APIs in the last two years. The 2019 State of API Report saw a higher percentage of early adopters than the 2016 State of API Report, with twice as many respondents saying that they only began developing APIs in the last year.

The accompanying graph is as follows:



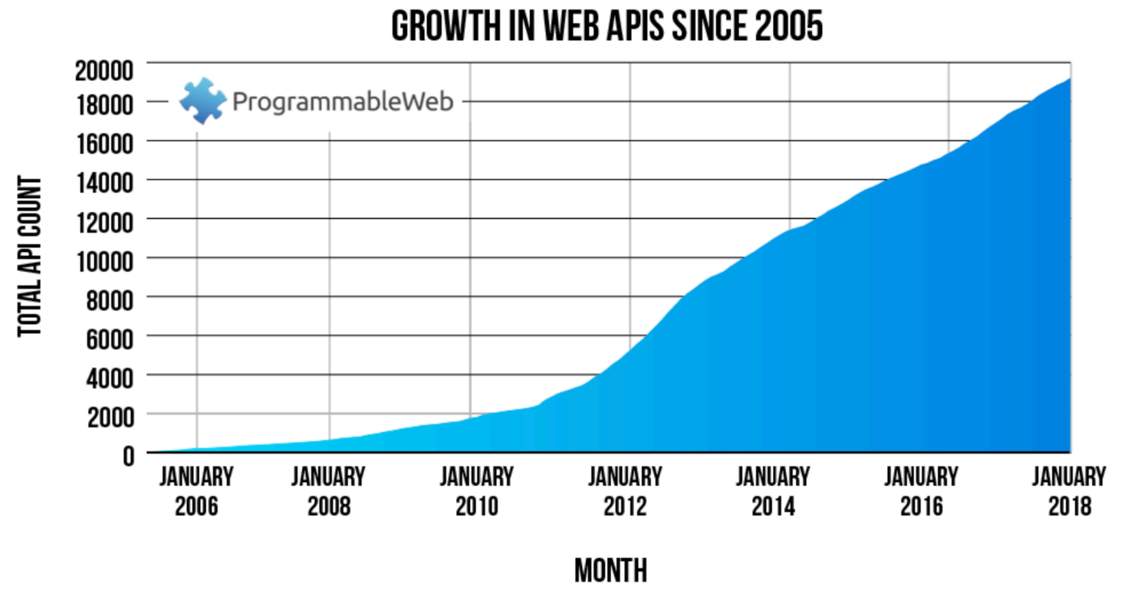
Most companies are new to API development

It’s astonishing that API development is so nascent in companies — “59% of organizations began developing APIs in the last five years.” If you look at Smartbear’s *The State of API 2016* report, which surveyed 2,300 professionals, you find similar growth rates:

> 42.1% of API providers have been providing/developing APIs for six years or more, while 51.5% began developing APIs in the last five years.

API development is clearly an area that is somewhat new for many companies, and the directions, methods, and other paths through this new territory aren’t clear. In fact, *The State of API 2019* report notes that companies are resoundingly asking for more standardization in this space.

Programmableweb.com was a site that charted and tracked the number of web APIs added to their directory. Programmableweb said, “Since January of 2014, an average of more than 2,000 APIs have been added per year” (Note: Programmableweb.com has since shut down.)



The growth over time of the ProgrammableWeb API directory to more than 19,000 entries

The phenomenal growth in web APIs

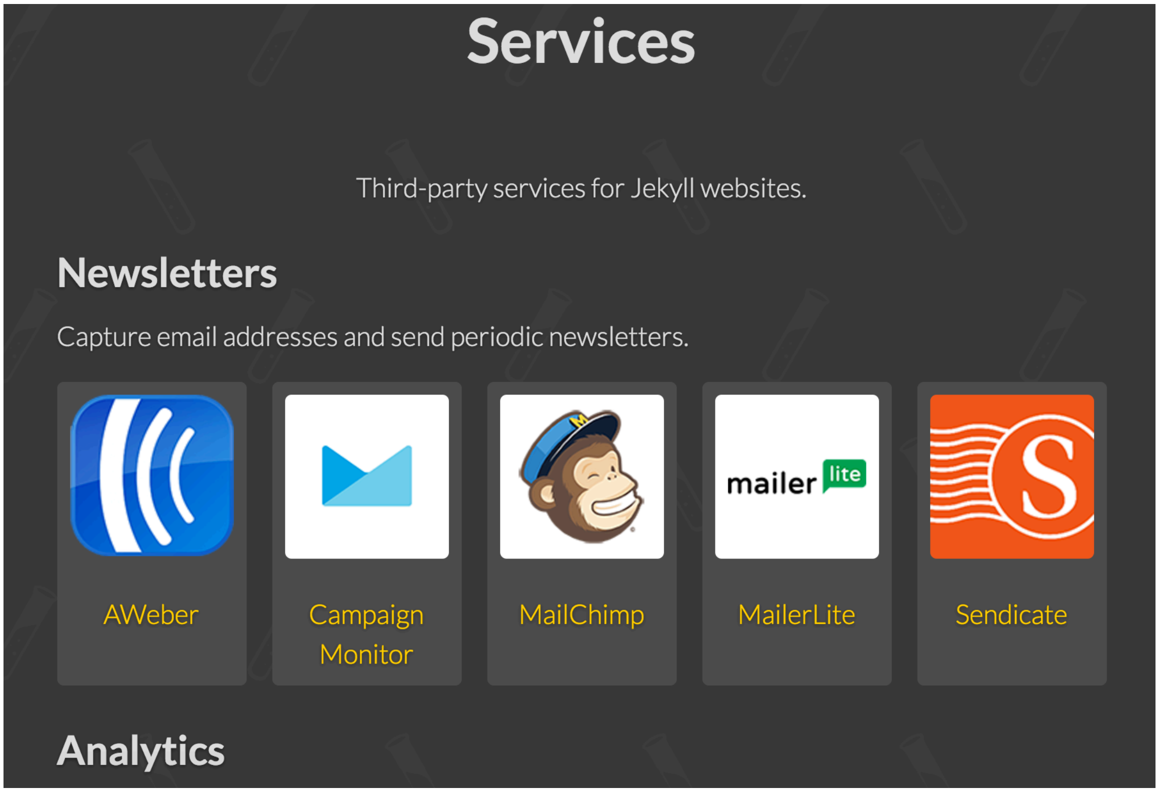
eBay’s API in 2005 was one of the first web APIs — the API allowed sellers to manage their products in their eBay stores. Since then, there has been tremendous growth in web APIs. Given the importance of clear and accurate API documentation, this presents a perfect market opportunity for technical writers. Technical writers can apply their communication skills to fill a gap in a market that is rapidly expanding.

Reasons for API growth?

Why are APIs growing in popularity, so much that you can pretty much search for any company name followed by “API” and land on developer docs for that company? One reason is that the web itself is evolving into a conglomeration of APIs. Instead of massive, do-it-all systems, websites are pulling in the services they need through APIs.

For example, rather than building your own search to power your website, you might use Algolia instead and leverage their service through the Algolia Search API ↗. Rather than building your own payment gateway, you might integrate the Stripe API ↗. Rather than building your own login system, you might use the UserApp API ↗. Rather than building your own e-commerce system, you might use the Snipcart API ↗. And so on.

Practically every service provides its information and tools through an API that you use. Jekyll, a popular static site generator, doesn’t have all the components you need to run a site. There’s no newsletter integration, analytics, search, commenting systems, forms, chat e-commerce, surveys, or other systems. Instead, you leverage the services you need into your static Jekyll site. (CloudCannon has put together a long list of services ↗ that you can integrate into your static site.)



Services

Third-party services for Jekyll websites.

Newsletters

Capture email addresses and send periodic newsletters.

Analytics

Many sites pull in all the services they need through external APIs

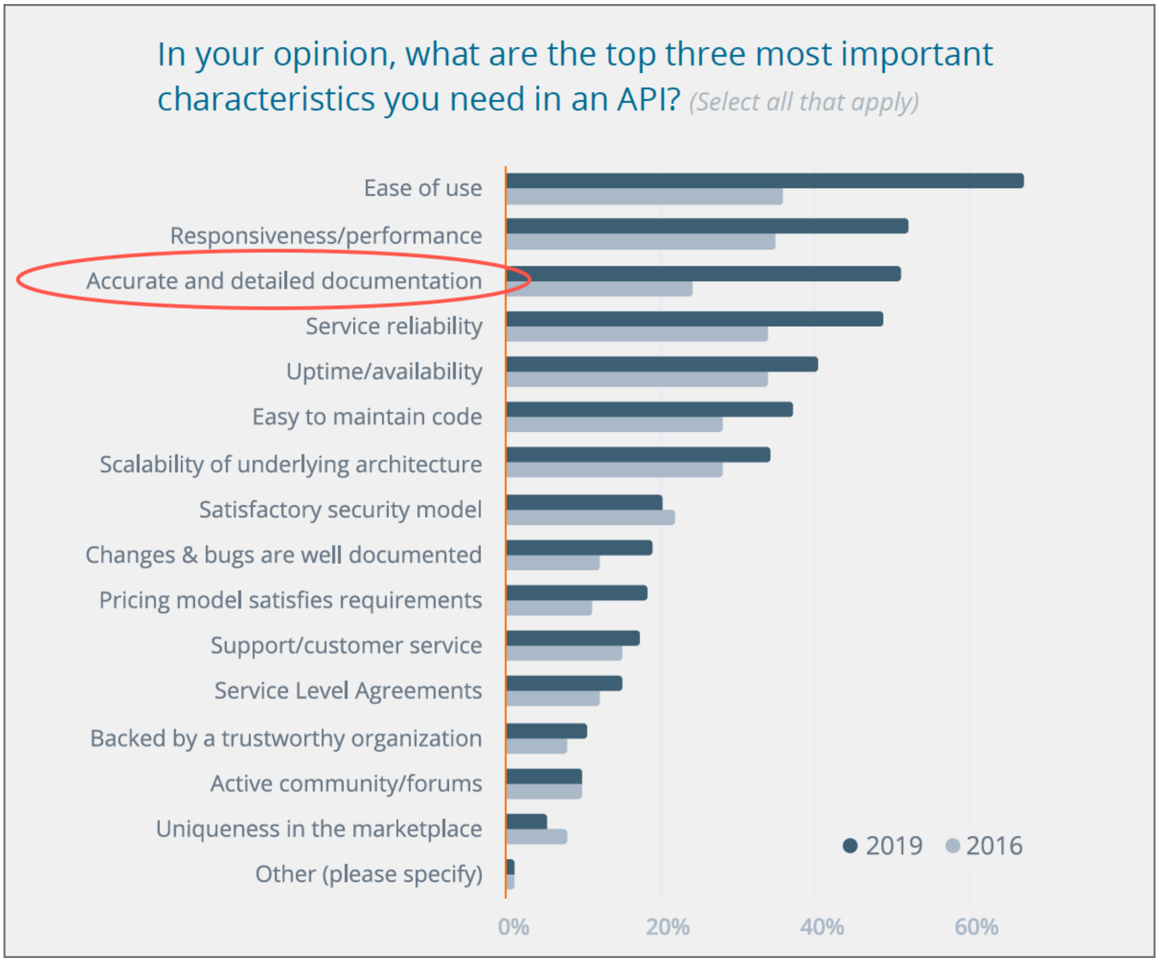
This cafeteria-style model is replacing the massive, swiss-army-site model that tries to do anything and everything. It's better to rely on specialized companies to create powerful, robust tools (such as search) and leverage their service rather than trying to build all of these services yourself.

The way each site leverages its service is usually through a REST API of some kind. Overall, the web is becoming an interwoven mashup of many different services and APIs interacting with each other.

The need for API documentation

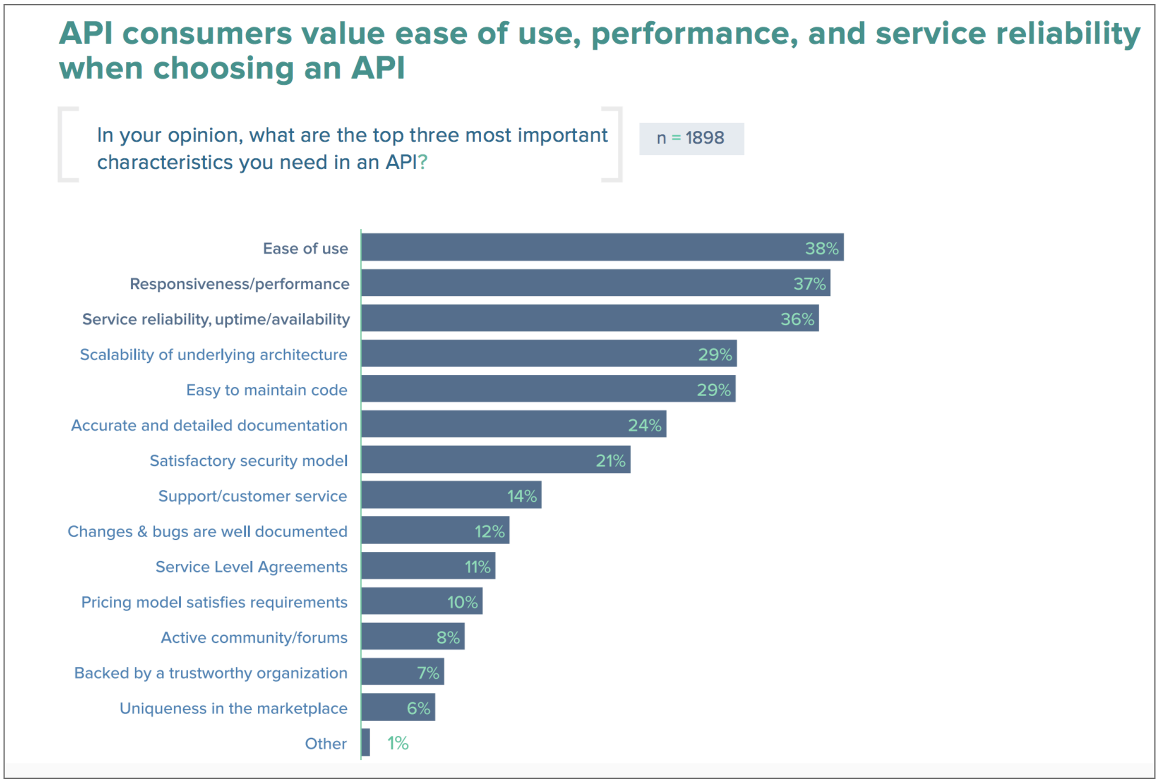
We've established that APIs are on the rise, following the model of the web, and that REST APIs are leading the pack as the most common type of API. But what about documentation for these APIs?

When asked "What are the top three most important characteristics you need in an API?", *The State of API 2019 report* found that "Accurate and detailed documentation" ranks third in importance:



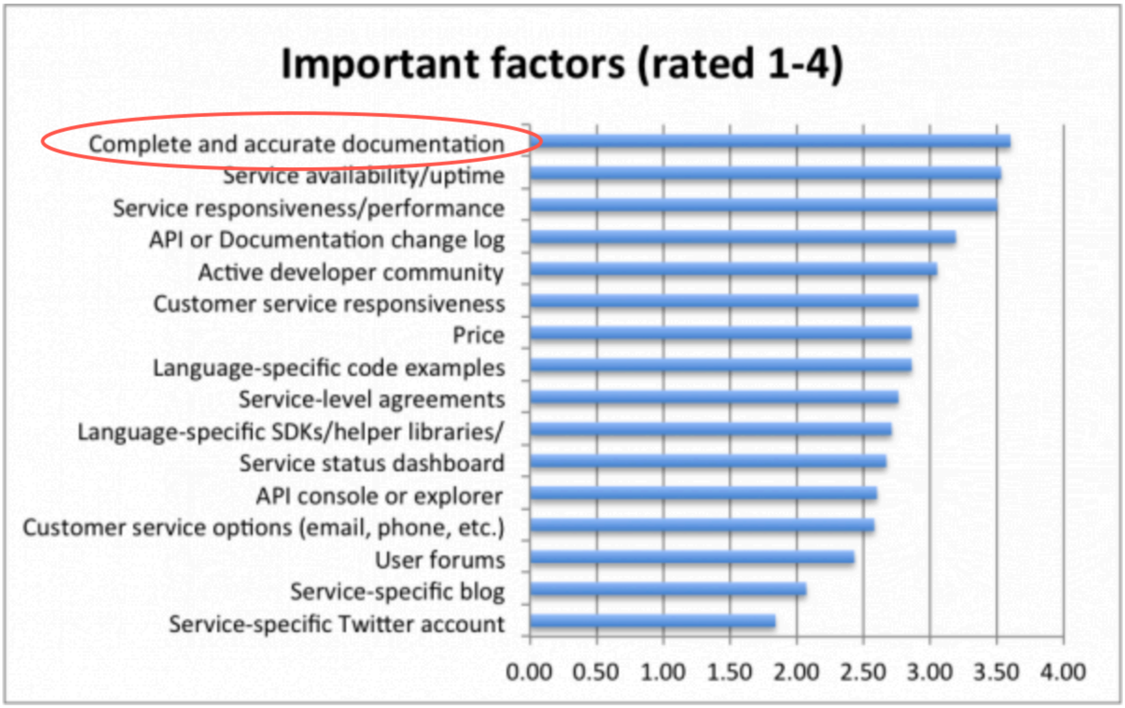
Top characteristics needed in an API (The State of API 2019, p.25)

This rank is three higher than noted in the *The State of API 2016* report, which put "Accurate and detailed documentation" sixth in importance:



In The State of API 2016 report, documentation ranked sixth in importance, p.19

An even earlier survey conducted by Programmableweb in 2013 (which included 250 respondents) found that “Complete and accurate documentation” actually ranked as the most important factor in an API (see API Consumers Want Reliability, Documentation and Community).



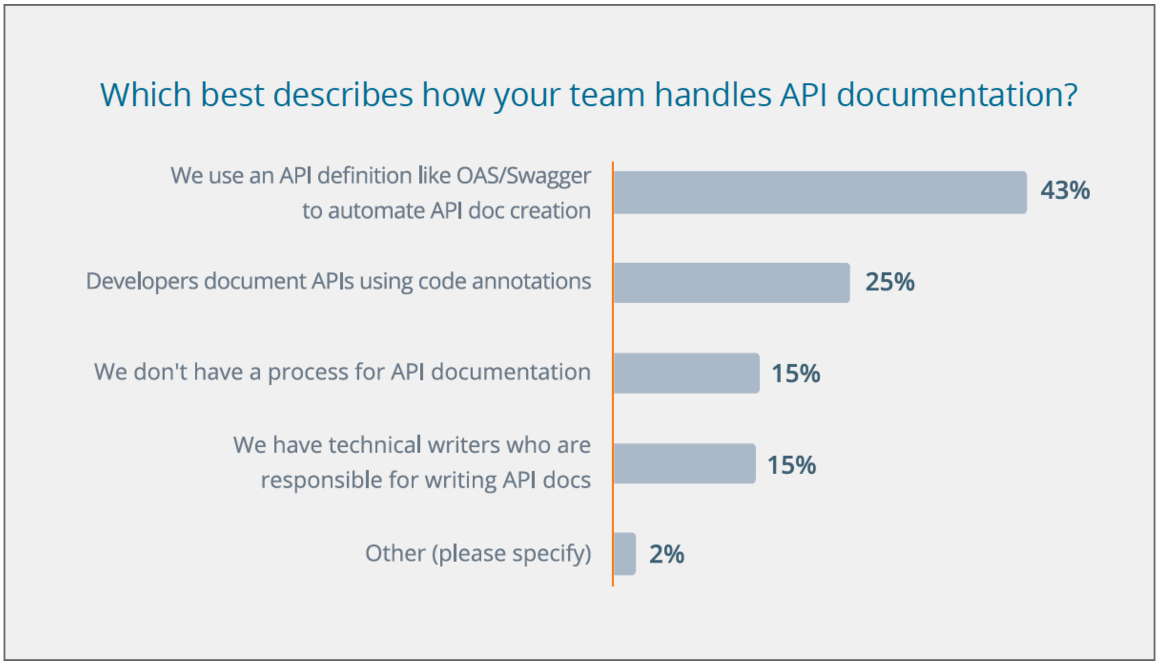
Programmableweb survey showing that complete and accurate documentation is the most important factor for developers

John Musser, one of the founders of Programmableweb.com (a site that's now retired), emphasizes the importance of documentation in his presentations. In 10 reasons why developers hate your API, he says the number one reason developers hate your API is because "Your documentation sucks."

APIs often fail because the doc fails the developers

Despite what might seem like a clear mandate for excellent API docs, technical writers aren’t always leveraged for these doc efforts. *The 2019 State of API* report found the following:

> Automation is playing a major role in how organizations document their APIs. 43% of respondents said they leverage an API standard, like the OpenAPI Specification, to generate API docs. One quarter of respondents are using a “code-first” approach, where developers are responsible for adding annotations to the API code to generate docs. Only 15% of participants are investing in technical writers to help with documenting APIs.



How teams are handling docs, The State of API 2019, p.42

Of course, the finding that "Only 15% of participants are investing in technical writers…" will upset technical writers. Fortunately, this question is poorly worded and might account for the disheartening answers. The wording suggests that employing technical writers to create docs is an *alternative* to generating docs through the OpenAPI specification (OAS). In reality, technical writers should be collaborating with engineers to generate reference documentation through OAS. Reference docs account for only part of the needed documentation (maybe half, if that). In this course, I heavily recommend that technical writers generate reference documentation through the OAS. This is covered in the OpenAPI spec and Swagger section (p. 180).

Thus, the connotation with this survey question misunderstands and distorts the role that technical writers play. Technical writers aren't old-school grammarians using feather and quill to etch out documentation in a tediously manual way (rather than auto-generating it via the OAS). Instead, many technical writers promote and champion OAS as a standard for creating the reference docs.

The *State of API 2016* report words the question a bit differently — "Does your organization have a formal API developer documentation process?" The 2016 report found that docs are a priority for about half of the respondents:

> Nearly 75% of API providers have a formal API developer documentation process, but only 45.6% say that it is a priority for their team. A quarter of API providers (25.3%) say they either do not have, or are unaware of their team's API developer documentation process.

Again, the question here is a bit vague. Exactly what is a "formal documentation process"? Given that one of Smartbear's key products is SwaggerHub (p. 230), which auto-generates reference documentation from the OpenAPI spec, a "formal documentation process" might mean generating docs from the OpenAPI spec.

Why the increased focus on documentation?

Why does documentation for REST APIs merit so much attention? Why is it ranking so high in these surveys? In short, documentation for REST APIs is important because REST follows an *architectural style* rather than an exact protocol standard.

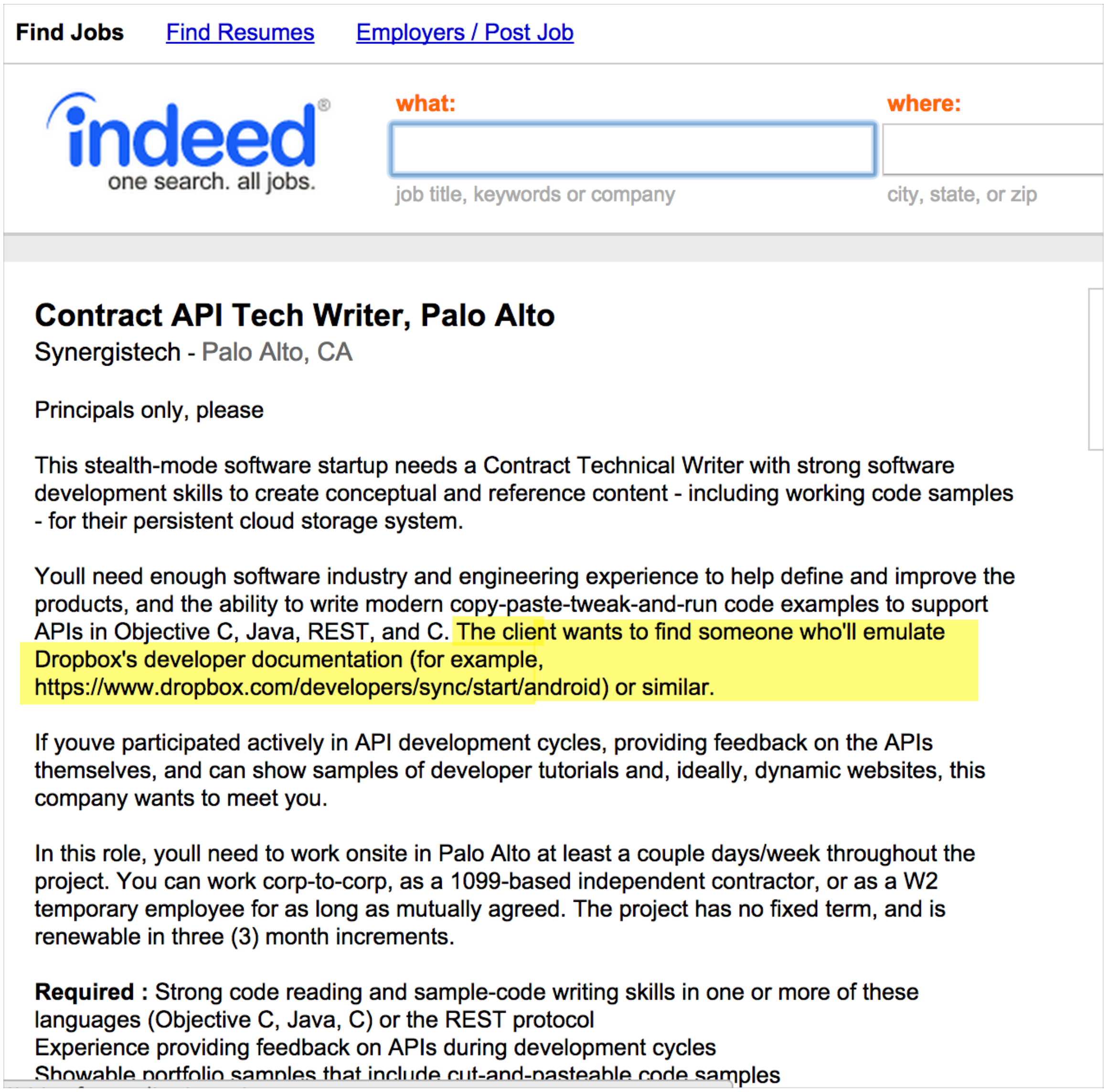
To understand the importance of documentation for REST APIs, it helps to compare REST with SOAP. REST APIs are a bit different from the SOAP APIs that were popular some years ago. SOAP APIs enforce a specific message format for sending requests and returning responses. As an XML message format, SOAP is very specific and has a WSDL (Web Service Description Language) file that describes how to interact with the API.

REST APIs, however, do not follow a standard message format. Instead, REST is an architectural *style*, a set of recommended practices for submitting requests and returning responses. To understand the request and response format for REST APIs, you don't consult the SOAP message specification or look at the WSDL file. Instead, you have to consult the REST API's *documentation*.

Each REST API functions a bit differently. There isn't a single way of doing things, and this flexibility and variety fuel the need for accurate and clear documentation. (I'll explain more about REST APIs in the What is a REST API? (p. 36)) As long as REST APIs vary from one to another, there will be a strong need for technical writers to provide documentation.

Job market is hot for API technical writers

Many employers are looking to hire technical writers who can create not only complete and accurate documentation but who can also create stylish outputs for their documentation. Here's a job posting from a recruiter looking for someone who can emulate Dropbox's documentation:



Contract API Tech Writer, Palo Alto

Synergistech - Palo Alto, CA

Principals only, please

This stealth-mode software startup needs a Contract Technical Writer with strong software development skills to create conceptual and reference content - including working code samples - for their persistent cloud storage system.

Youll need enough software industry and engineering experience to help define and improve the products, and the ability to write modern copy-paste-tweak-and-run code examples to support APIs in Objective C, Java, REST, and C. The client wants to find someone who'll emulate Dropbox's developer documentation (for example, https://www.dropbox.com/developers/sync/start/android) or similar.

If youve participated actively in API development cycles, providing feedback on the APIs themselves, and can show samples of developer tutorials and, ideally, dynamic websites, this company wants to meet you.

In this role, youll need to work onsite in Palo Alto at least a couple days/week throughout the project. You can work corp-to-corp, as a 1099-based independent contractor, or as a W2 temporary employee for as long as mutually agreed. The project has no fixed term, and is renewable in three (3) month increments.

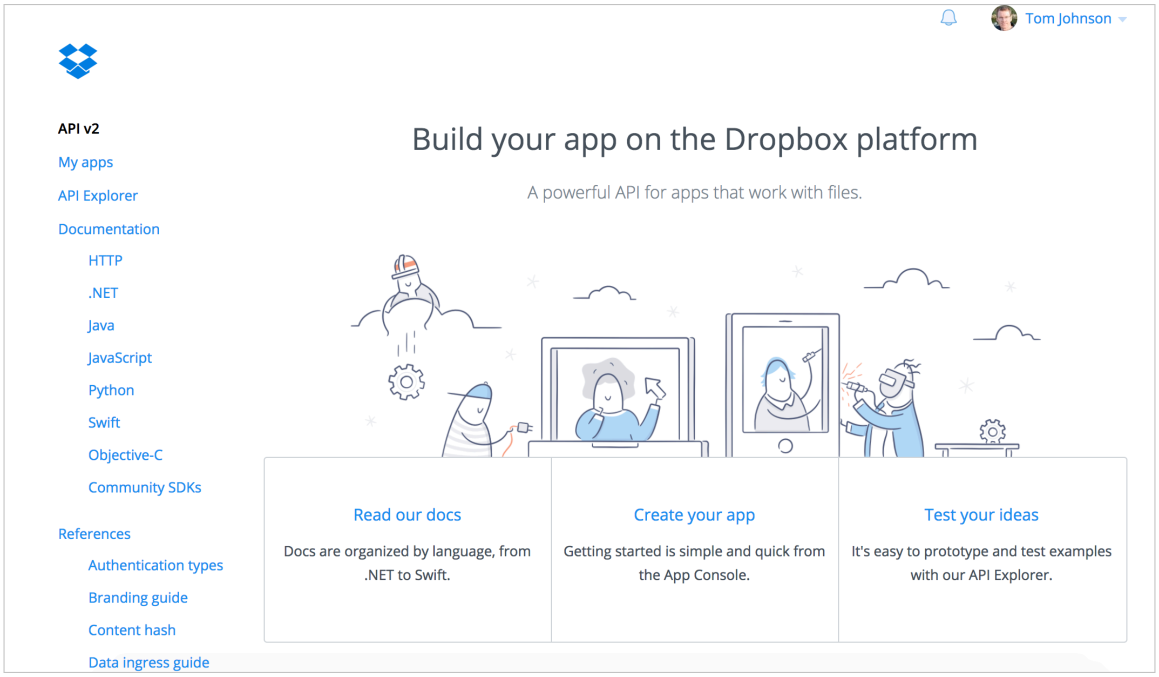
Required : Strong code reading and sample-code writing skills in one or more of these languages (Objective C, Java, C) or the REST protocol
Experience providing feedback on APIs during development cycles
Showable portfolio samples that include cut-and-pasteable code samples

Job description asking for someone with skills to create doc site like Dropbox

As you can see, the client wants to find "someone who'll emulate Dropbox's documentation."

Why does the look and feel of the documentation matter so much? With API documentation, there is no GUI interface for users to browse. Instead, the documentation *is* the interface. Employers know this, so they want to make sure they have the right resources to make their API docs stand out as much as possible.

Here's what the Dropbox API doc site looks like:



Build your app on the Dropbox platform

A powerful API for apps that work with files.

Read our docs

Docs are organized by language, from .NET to Swift.

Create your app

Getting started is simple and quick from the App Console.

Test your ideas

It's easy to prototype and test examples with our API Explorer.

Dropbox API's developer site has a simple but clean UI

It’s not a sophisticated design. But its simplicity and brevity are what make it appealing. When you consider that the API documentation is more or less the product interface, building a sharp, modern-looking doc site is paramount for credibility and traction in the market. (I dive into the job market for API documentation later (p. 660).) Basically, if you have strong technical skills and experience writing for developers, you can have nearly any job you want in Silicon Valley and command a base salary of $100-$150k or more.

API doc is a new world for most tech writers

API documentation is often a new world for technical writers. Many of the components may differ from traditional GUI documentation. For example, all of these aspects with developer documentation differ from traditional documentation:

- Authoring tools
- Audience
- Programming languages
- Reference topics
- User tasks

When you try to navigate the world of API documentation, you may be initially overwhelmed by the differences and intimidated by the tools and code. Additionally, the documentation content itself is often complex and requires familiarity with development concepts and processes.

Overall, technical writers are hungry to learn more about APIs. This course will help you build the foundation of what you need to know to get a job in API documentation and excel in this field. As a skilled API technical writer, you will be in high demand and will fulfill a critical role in companies that distribute their services through APIs.

What is a REST API?

This course is all about learning by doing, but while *doing* various activities, I'll periodically pause and dive into some more abstract concepts to fill in more detail. This topic is one of those deep-dive moments into concepts. Here we'll explore what a REST API is, comparing it to other types of APIs like SOAP. REST APIs have common characteristics but no definitive protocols like its SOAP predecessor.

What is an API?

In general, an API (or Application Programming Interface) provides an interface between two systems. It's like a cog that allows two systems to interact with each other. In this case, the two systems are computers that interact programmatically through the API.

An API is like a cog that allows two different systems to interact. (Image from Brent 2.0, spinning gears, CC BY-ND 2.0.)

APIs are often pulling and pushing data underneath user interfaces. Consider the example of a flight booking site like kayak.com or orbitz.com. When you search for flights, the site reaches out to many other airline services to retrieve information about flights matching your query. kayak.com itself doesn't have all of this information. Instead, it gathers this data through APIs.

Or consider other similar services. When you see a YouTube video embedded on a site, the site itself doesn't host the video. Instead, the embed code contains API requests that get the video from YouTube's hosting centers. When you embed a Twitter widget on your site, the widget's APIs pull in tweets from a Twitter server onto your site.

Consider another example from Fire TV. Suppose you want to view a pay-per-view event that requires payment. A pop-up dialog might appear letting you enter your credit card details. Your payment gets processed and you receive confirmation about the purchase's success. This payment processing doesn't occur within the app. Instead, the app makes API calls out to payment servers with the needed information, and all that payment processing happens in the cloud. When the processing finishes, the API returns a confirmation response.

The next time you push a button in some interface, think about what's going on underneath. Most likely, the button is internally wired to make calls to an external service for the information needed. That call is made through an API that performs a request and receives a response.

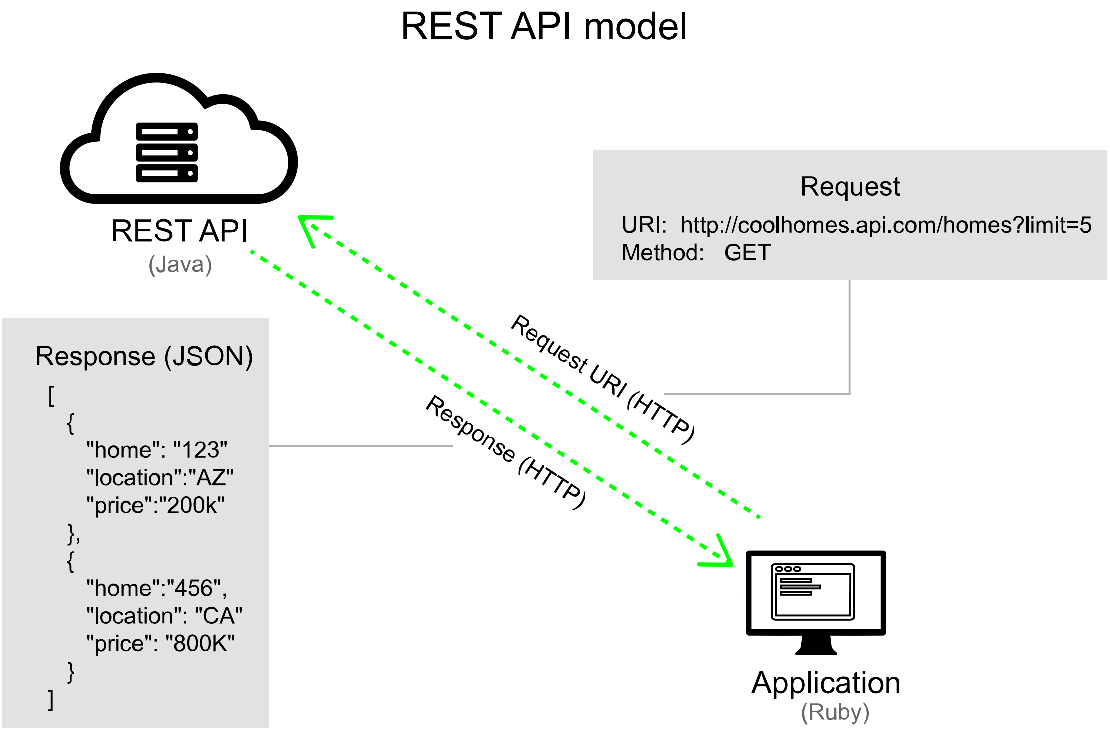
APIs that use HTTP protocol are "web services"

A "web service" is a web-based application that provides resources in a format consumable by other computers. Web services include various types of APIs, including both REST and SOAP APIs. Web services are basically request-and-response interactions between clients and servers (a computer requests a resource, and the web service responds to the request).

All APIs that use HTTP protocol as the transport format for requests and responses are considered "web services." With web services, the client making the request for the resource and the API server providing the response can use any programming language or platform — it doesn't matter because the message request and response are made through a common HTTP web protocol.

The web protocol is part of the beauty of web services: they are language agnostic and therefore interoperable across different platforms and systems. When documenting a REST API, it doesn't matter whether engineers build the API with Java, Ruby, Python, or some other language. The requests are made over HTTP, and the responses are returned through HTTP.

The following diagram shows the general model of a REST API:



Request
URI: http://coolhomes.api.com/homes?limit=5
Method: GET

Response (JSON)

```
[
  {
    "home": "123"
    "location":"AZ"
    "price":"200k"
  },
  {
    "home":"456",
    "location": "CA"
    "price": "800K"
  }
]
```

General model of a REST API

As you can see, there's a request and a response between a client to the API server. The client and server can be based in any language, but HTTP is the protocol used to transport the message. This request-and-response pattern is fundamentally how REST APIs work.

Each programming language that makes the request will have a different way of submitting a web request and parsing the response in its language. These language-specific functions for making requests and parsing responses aren't part of the REST API (though they might be provided in an accompanying SDK (p. 480)). The REST API is language agnostic and handles incoming and outgoing information across HTTP, just like the web.

SOAP APIs are the predecessor to REST APIs

Before REST became the most popular web service, SOAP (Simple Object Access Protocol) was much more common. To understand REST a little better, it helps to have some context with SOAP. This way you can see what makes REST different.

SOAP is a standardized protocol that requires XML as the message format for requests and responses. As a standardized protocol, the message format is usually defined through something called a WSDL (Web Services Description Language) file.

The WSDL file defines the allowed elements and attributes in the message exchanges. The WSDL file is machine readable and used by the servers interacting with each other to facilitate the communication.

SOAP messages are enclosed in an "envelope" that includes a header and body, using a specific XML schema and namespace. For an example of a SOAP request and response format, see SOAP vs REST 101: Understand The Differences ↗.

The main problem with SOAP is that the XML message format is too verbose and heavy. It is particularly problematic with mobile scenarios where file size and bandwidth are critical. The verbose message format slows processing times, which makes SOAP interactions lethargic.

SOAP is still used in enterprise application scenarios (especially with financial institutions) with server-to-server communication, but in the past five years, SOAP has mostly been replaced by REST, especially for APIs on the open web.

REST is a style, not a standard

Like SOAP, REST (Representational State Transfer) uses HTTP as the transport protocol for the message requests and responses. However, unlike SOAP, REST is an architectural style, not a standard protocol. This is why REST APIs are sometimes called *RESTful* APIs — REST is a general style that the API follows.

A RESTful API might not follow all of the official characteristics of REST as outlined by Dr. Roy Fielding, who first described the model. Hence these APIs are “RESTful” or “REST-like.” (Some developers insist on using the term “RESTful” when the API doesn’t fulfill all the characteristics of REST, but most people just refer to them as “REST APIs” regardless.)

As an architectural style, you aren’t limited to XML as the message format. REST APIs can use any message format the API developers want to use, including XML, JSON, Atom, RSS, CSV, HTML, and more.

Despite the variety of message format options, most REST APIs use JSON (JavaScript Object Notation) as the default message format. They use JSON because it provides a lightweight, simple, and more flexible message format that increases the speed of communication. The lightweight nature of JSON also allows for mobile processing scenarios and is easy to parse on the web using JavaScript.

In contrast, with XML, XSLT is used more for presenting or rather “transforming” (the “T” in XSLT) the content stored in an XML language. XSLT enables the human readability (rather than processing data stored in an XML format).

REST focuses on resources accessed through URLs

Another unique aspect of REST is that REST APIs focus on *resources* (that is, *things*, rather than actions) and ways to access the resources. Resources are typically different types of information. You access the resources through URLs (Uniform Resource Locators), just like going to a URL in your browser retrieves an information resource. The URLs are accompanied by a method that specifies how you want to interact with the resource.

Common methods include GET (read), POST (create), PUT (update), and DELETE (remove). The endpoint usually includes query parameters that specify more details about the representation of the resource you want to see. For example, you might specify (in a query parameter) that you want to limit the display to 5 instances of the resource.

Here’s what a sample endpoint might look like:

```
http://apiserver.com/homes?limit=5&format=json
```

The endpoint shows the whole path to the resource. However, in documentation, you usually separate this URL into more specific parts:

- The **base path** (or base URL or host) refers to the common path for the API. In the example above, the base path is `http://apiserver.com`.
- The **endpoint** refers to the end path of the endpoint. In the example above, `/homes`.
- The `?limit=5&format=json` part of the endpoint contains query string parameters for the

endpoint.

In the example above, this endpoint would get the “homes” resource and limit the result to 5 homes. It would return the response in JSON format.

You can have multiple endpoints that refer to the same resource. Here’s one variation:

```
http://apiserver.com/homes/{home id}
```

The above URL might be an endpoint that retrieves a home resource that contains a particular ID. What is transferred back from the server to the client is the “representation” of the resource. The resource may have many different representations (showing all homes, homes that match certain criteria, homes in a specific format, and so on), but here we want to see a home with a particular ID.

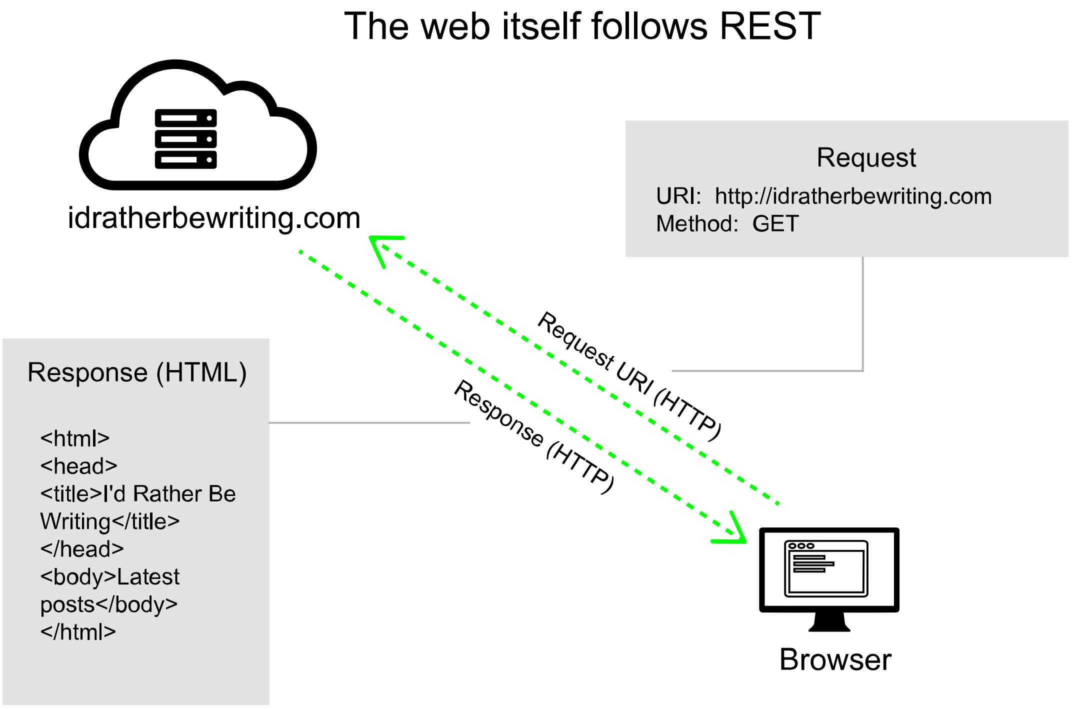
The relationship between resources and methods is often described in terms of “nouns” and “verbs.” The resource is the noun because it is an object or thing. The verb is what you’re doing with that noun. Combining nouns with verbs is how you form the language in REST.

We’ll explore endpoints in much more depth in the sections to come (for example, in the API reference tutorial (p. 118) we go through each property in a resource). But I wanted to provide a brief overview here.

The web itself follows REST

The terminology of “GET requests” and “message responses” transported over “HTTP protocol” might seem unfamiliar, but this is just the official REST terminology to describe what’s happening. Because you’ve used the web, you’re already familiar with how REST APIs work — the web itself essentially follows a RESTful style.

If you open a browser and go to https://idratherbewriting.com, you’re really using HTTP protocol (`https://`) to submit a GET request to the resource available on a web server. The response from the server sends the content at this resource back to you using HTTP. Your browser is just a client that makes the message response look pretty.


The web itself follows REST
idratherbewriting.com
Request
URI: http://idratherbewriting.com
Method: GET
Request URI (HTTP)
Response (HTTP)
Response (HTML)
<html>
<head>
<title>I'd Rather Be Writing</title>
</head>
<body>Latest posts</body>
</html>
Browser

Web as REST API

You can see this response in curl (p. 78) if you open a terminal prompt and type `curl https://idratherbewriting.com`. (This assumes you have curl installed (p. 78).)

Because the web itself is an example of RESTful style architecture, the way REST APIs work will likely become second nature to you.

REST APIs are stateless and cacheable

REST APIs are also stateless and cacheable. Stateless means that each time you access a resource through an endpoint, the API provides the same response. It doesn't remember your last request and take that into account when providing the new response. In other words, there aren't any previously remembered states that the API takes into account with each request.

The responses can also be cached to increase the performance. If the browser's cache already contains the information asked for in the request, the browser can just return the information from the cache instead of getting the resource from the server again.

Caching with REST APIs is similar to caching of web pages. The browser uses the last-modified-time value in the HTTP headers to determine if it needs to get the resource again. If the content hasn't been modified since the last time it was retrieved, the cached copy can be used instead. Caching increases the speed of the response.

REST APIs have other characteristics, which you can dive more deeply into on this REST API Tutorial. One of these characteristics includes links in the responses to allow users to page through to additional items. This feature is called HATEOAS, or Hypermedia As The Engine of Application State.

Understanding REST at a higher, more theoretical level isn't my goal here, nor is this knowledge necessary to document a REST API. However, there are many technical books, courses, and websites that explore REST API concepts, constraints, and architecture in more depth that you can consult if you want to. For example, check out Foundations of Programming: Web Services by David Gassner ↗ on lynda.com.

REST APIs don't use WSDL files, but some specs exist

An important aspect of REST APIs, especially in the context of documentation, is that they don't use a WSDL file to describe the elements and parameters allowed in the requests and responses.

Although there is a possible WADL (Web Application Description Language) file that can be used to describe REST APIs, WADL files are rarely used because they don't adequately describe all the resources, parameters, message formats, and other attributes of the REST API. (Remember that the REST API is an architectural style, not a standardized protocol.)

To understand how to interact with a REST API, you have to *read the documentation* for the API. The need to read the docs makes the technical writer's role extremely important with REST APIs.

Some formal specifications — for example, OpenAPI (p. 181) and RAML (p. 924) — have been developed to describe REST APIs. When you describe your API using the OpenAPI or RAML specification, tools that can read those specifications (such as Swagger UI (p. 218) or the RAML API Console (p. 936)) will generate an interactive documentation output.

The OpenAPI specification document can take the place of the WSDL file that was more common with SOAP. Tools like Swagger UI (p. 218) that read the specification documents usually produce interactive documentation (featuring API Consoles or API Explorers) and allow you to try out REST calls and see responses directly in the browser.

But don't expect the Swagger UI or RAML API Console documentation outputs to include all the details users would need to work with your API. For example, these outputs won't include info about authorization keys (p. 393), details about workflows and interdependencies between endpoints, and so on. The Swagger or RAML output usually contains reference documentation only, which typically accounts for only a third or half of the total needed documentation (depending on the API).

Overall, REST APIs are more varied and flexible than SOAP APIs, and you almost always need to read the documentation to understand how to interact with a REST API. As you explore REST APIs, you will find that they differ significantly from one to another (especially the format and display of their documentation sites, which we'll look at in Survey of API doc sites (p. 536)), but they all share the common patterns outlined here. At the core of any REST API is a request and response transmitted over the web.

Additional reading

- REST: a FAQ ↗, by Diogo Lucas
- Learn REST: A RESTful Tutorial ↗, by Todd Fredrich
- Understanding RPC Vs REST For HTTP APIs ↗, by Phil Sturgeon

Activity 1a: Identify your goals

Ramping up on API documentation, developing a portfolio of API documentation writing samples, and completing all the activities in this course will require a lot of time and effort. In this activity, you'll identify your goals and reasons for taking this course. Reflecting on your motivation will help you develop the right mindset and stamina to devote the necessary time for the course.

Activity: Identify your goals with API documentation

Identify your goals here and make sure they align with this course. Think about the following questions:

- Why are you taking this course?
- What are your career ambitions related to API documentation?
- Are you in a place where developer documentation jobs are plentiful?
- What would you consider to be a success metric for this course?
- Do you have the technical mindset needed to excel in developer documentation fields?

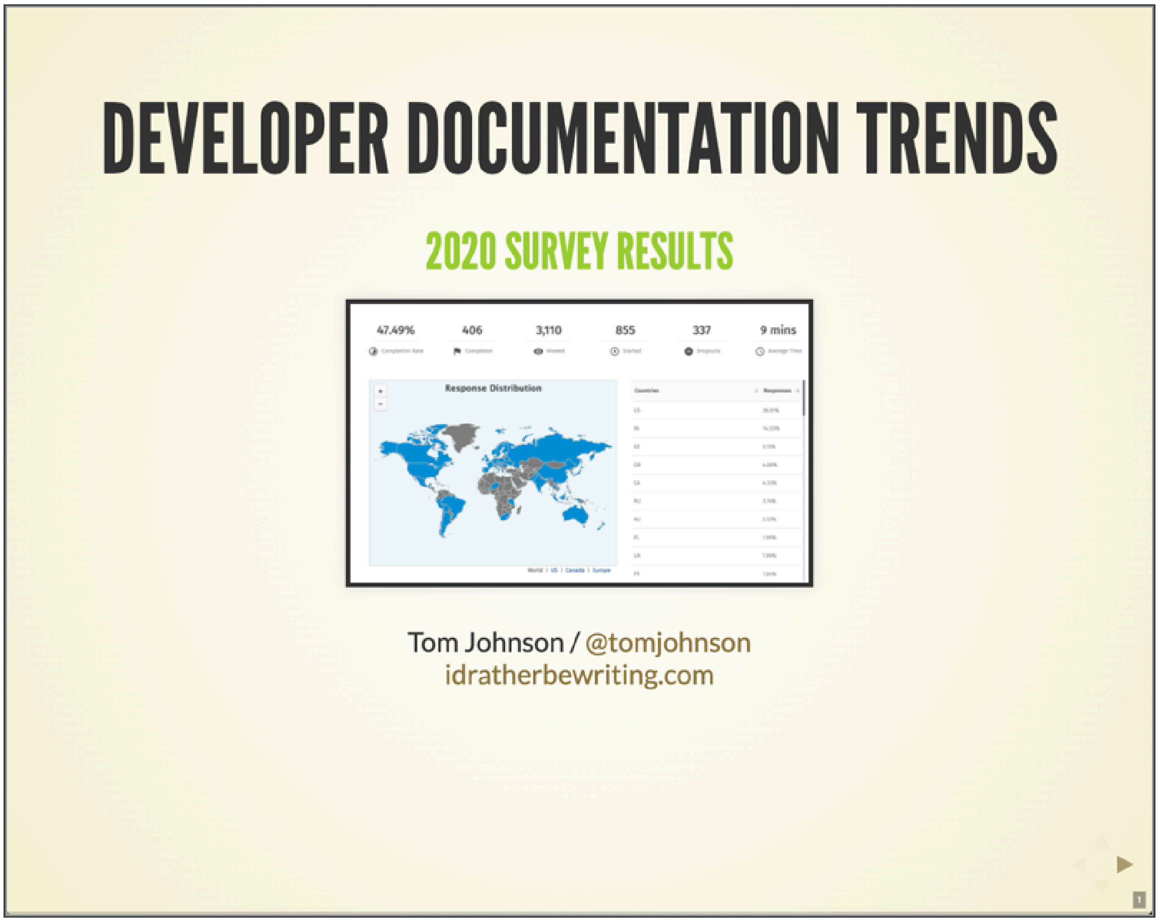
For live workshops, we typically share responses in a get-to-know-everyone format. But if you're taking this course online, consider jotting down some thoughts in a journal or blog entry.

Developer documentation trends — survey results

I recently conducted a survey about trends for those creating documentation for developers and engineers. You can view the content in several formats: slides, webinar, or article.

Slides

You can view the slides here:



DEVELOPER DOCUMENTATION TRENDS

2020 SURVEY RESULTS

Tom Johnson / @tomjohnson
idratherbewriting.com

47.49%
406
3,110
855
337
9 mins
Response Distribution
World | US | Canada | Europe
1

Webinar

You can also view a recorded webinar where I talk through the results here:

Note: This content doesn't embed well in print because it contains a YouTube video. Please go to [https://idratherbewriting.com/learnapidoc/docapis_course_videos.html] to view the content.

Article

I wrote an article for the *Institute of Scientific Technical Communicators* (ISTC) magazine (Autumn 2020).

Focus on eLearning and Training
Read multiple articles focusing on different aspects of eLearning and training
STC
Communicator
The Institute of Scientific and Technical Communicators
Autumn 2020
DO NOT DISTURB
Read about MadCap Flare 2020
Discover the latest developer documentation trends
Learn the benefits of HelpXplain from EC Software
Understand the importance of terminology management

The same content from the PDF is available in HTML below:

Developer documentation trends: How developer documentation trends differ from general technical communication trends

Introduction

Despite excellent research on trends in the technical communication space, so far no survey has focused exclusively on trends within developer documentation only. By developer docs, I mean documentation written primarily for developers and engineers. Two recent surveys on the general tech comm space include Saul Carliner's Tech Comm Census results (published in Dec 2018 *STC Intercom*) and Scott Abel's Benchmarking Survey (summarized in the same issue).

Reading the results of these surveys, one would assume that most technical writers use Microsoft Word, Adobe FrameMaker, help authoring tools, CCMSs, and DITA. However, these surveys miss out on an important and sweeping tool change, often referred to as "docs-as-code," that is taking place on the web. They also don't explore many other trends within the developer doc space.

Scott's survey does include some API-related information. He found that "Fifty-eight percent of technical communication teams surveyed say they currently document APIs; 10 percent plan to in the future." One challenge tech writers face in documenting APIs is "using software tools not optimized for ease-of-use or writing efficiency, and lack of experience." Scott's survey also found that 21% of technical communicators use Markdown to create docs.

These responses about APIs are more relevant to developer docs, but they don't go far enough. More developer-oriented topics are left out, such as how writers integrate with engineering Scrum teams, how writers interact with engineers on documentation, how writers handle the OpenAPI spec and other reference docs, and more.

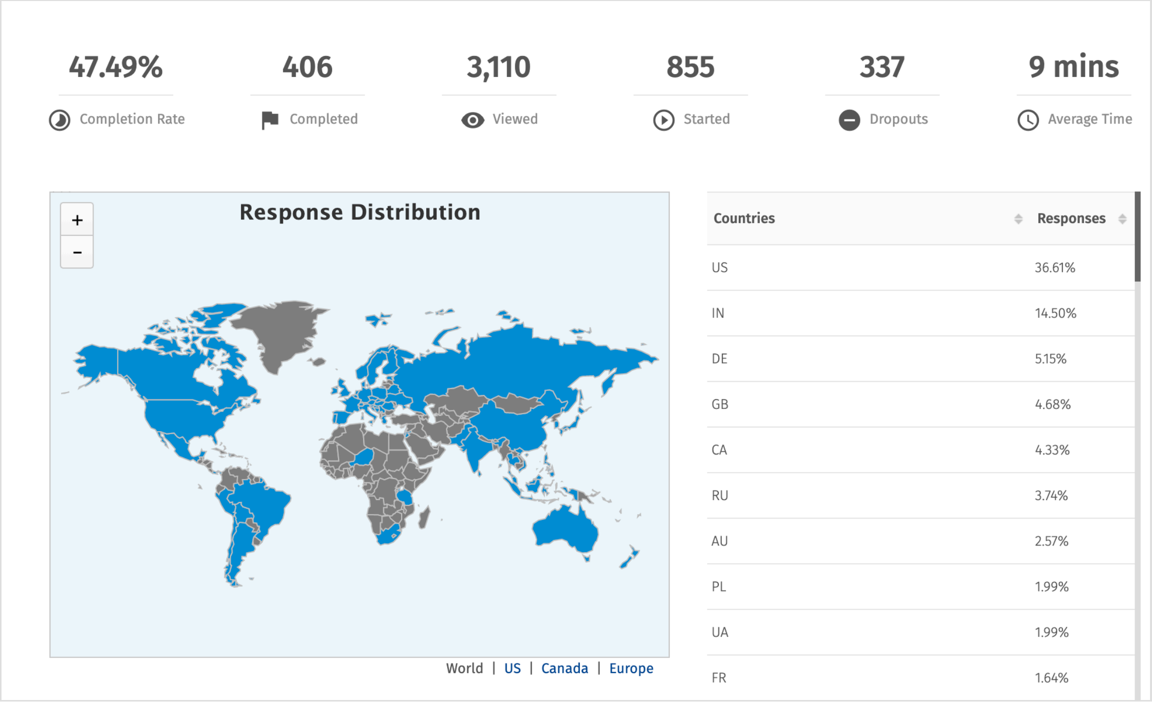
Don't get me wrong. I highly value these general surveys and the information they provide. But I was perplexed to see Adobe FrameMaker and Microsoft Word used so prominently. Admittedly, the tools usage reported by these surveys wasn't too far off from previous WritersUA Tools surveys. For example, in 2014, WritersUA found that 52% of writers (199 out of 382 respondents) used FrameMaker (2014 WritersUA Tools Survey ↗).

Reading these surveys made me wonder — is it really the case that so many tech writers are still using FrameMaker and Word? That didn't match what I was seeing around me in Silicon Valley. But was I living in a bubble, an anomaly to the rest of the tech comm world? Were trends toward docs-as-code tools much more widespread and common in developer docs? The general tech comm surveys left me with more questions than answers.

A survey focusing on developer docs

To gather data about trends in developer docs, I decided to create my own survey. In the first developer documentation survey of its kind, I created a list of 50 questions, mostly multiple choice. I limited the audience to people writing docs for developers/engineers only. I promoted the survey on my blog, LinkedIn, and Twitter, and left the survey open for about two months, from January to March 2020.

A total of 405 people completed the entire survey. Completing the survey means that after the 50th question, they clicked Submit. However, 855 started the survey, and 337 dropped out somewhere along the way. I allowed partial responses even if users dropped out along the way. So the actual number of respondents varied between 405 and 855, with some questions receiving more answers than others. About 37% of the respondents were in the US, about 15% in India, 5% in Germany, 5% in the UK, and smaller percentages from other countries.



47.49%	406	3,110	855	337	9 mins
Completion Rate	Completed	Viewed	Started	Dropouts	Average Time

Countries	Responses
US	36.61%
IN	14.50%
DE	5.15%
GB	4.68%
CA	4.33%
RU	3.74%
AU	2.57%
PL	1.99%
UA	1.99%
FR	1.64%

You can browse the results of the survey directly at https://www.questionpro.com/t/PGhS9ZgCFE.

Survey question categories

After the survey, to make better sense of the responses, I divided the 50 questions into five categories:

1. Tool responses (p. 47)
2. Formats and outputs responses (p. 49)
3. Process and workflow responses (p. 50)
4. API responses (p. 51)
5. Profile responses (p. 52)

In the sections that follow, I'll go through each section and provide summaries, highlights, and analyses. Percentages are rounded up or down. For more granular details, feel free to browse the survey results directly.

1. Tool responses

Survey summary:

- **Primary authoring tool**: 22% static site generators (such as Jekyll, Hugo, Gatsby, Sphinx), 14% wikis, 11% XML tools, 8% HATs, 3% FrameMaker
- **Text editor** 25% Visual Studio Code, 19% Notepad++, 14% Atom
- **Source format**: 37% Markdown, 15% HTML, 15% XML
- **Follow docs-as-code approach**: 56% yes, 22% somewhat, 20% no
- **Platform for publishing docs**: 31% company's own web servers or infrastructure, 15% GitHub Pages, 10% Gitlab
- **Computer type**: 54% Windows, 40% Mac
- **How you manage content**: 67% Git, 8% CMS, 5% CCMS

Summary and analysis:

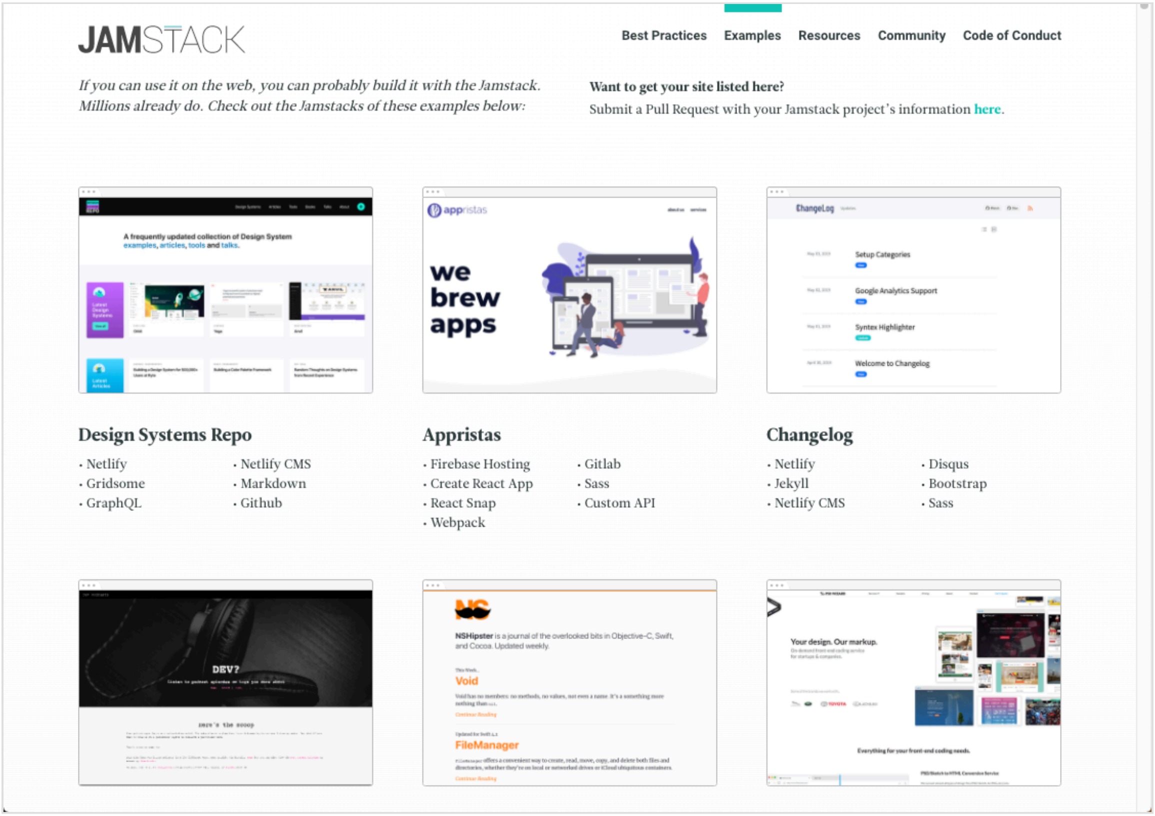
In the dev doc space, tech writers don't use a single tool for authoring, review, and publishing. Writers use different tools for different scenarios and purposes. For example, writers might use Confluence, Word, or Google Docs for early content development. When they transition the content into their authoring/publishing system, they work in Visual Studio Code or Atom as the text editor. Within these text editors, they usually write in Markdown formats with some YAML frontmatter.

Writers build the site output using a static site generator, such as Hugo, Jekyll, Sphinx, or MkDocs. To manage the content (for feature branches or to pull in work from others), they use Git. When it comes time to deploy the site build on a web server, they often have a continuous integration / continuous deployment (CI/CD) model that pushes the content onto GitHub, GitLab, or their company's own infrastructure through a few keystrokes on the command line. This workflow is known as a "docs-as-code" approach because it treats documentation similar to how software developers treat code (to an extent).

Given this workflow, which would you say is the author's "tool"? It's unclear. The days when writers used an all-in-one purpose tool (for example, a single help authoring tool) for authoring, review, collaboration, and publishing are gone.

Complicating the tool question even more are writers who don't have any tools outside their IDE, or integrated development environment (for example, IntelliJ IDEA). Some writers, usually engineers who are also writing docs, work only in code annotations and OpenAPI specifications. There is no "authoring tool." For these writers, Markdown is their tool, as they might format annotations with Markdown and use scripts to export the Markdown into different systems. Many systems can import or export Markdown, making it a somewhat standard source format in this space (despite the many variants of Markdown flavors).

For examples of how multiple tools are used together in different combinations and solutions, see Jamstack examples ↗. Jamstack refers to serverless websites built with JavaScript, APIs, and Markup and reflects modern web development trends. Jamstack excludes tools such as WordPress or other web apps that would require a heavy backend component on a server to run.



If you can use it on the web, you can probably build it with the Jamstack. Millions already do. Check out the Jamstacks of these examples below:

Want to get your site listed here?
Submit a Pull Request with your Jamstack project's information here.

Design Systems Repo

- Netlify
- Gridsome
- GraphQL
- Netlify CMS
- Markdown
- Github

Appristas

- Firebase Hosting
- Create React App
- React Snap
- Webpack
- Gitlab
- Sass
- Custom API

Changelog

- Netlify
- Jekyll
- Netlify CMS
- Disqus
- Bootstrap
- Sass

Overall, the survey results confirmed the predominance of the docs-as-code approach in the dev docs space. If you're working with developer docs, this approach is trending. However, there's also a decent amount of wikis, Oxygen XML, and MadCap Flare use, probably among those groups that have more robust localization and PDF requirements.

To read more thoughts about how source formats affect not just how we write but *what* we write, see my blog post, *How* you write influences *what* you write — interpreting trends through movements from PDF to web, DITA, wikis, CCMSs, and docs-as-code ↗.

2. Formats and output responses

Survey summary:

- **Primary output**: 72% HTML, 23% PDF
- **Create video tutorials**: 28% yes, 57% no, 14% plan to
- **Docs plug into dev portal**: 56% yes, 41% no
- **Localize your docs**: 73% no, 14% 1-3 languages, 10% 4+ languages
- **Generate PDFs & distribute to audience**: 57% no, 30% yes, 9% internal review only
- **Significant role in developing publishing solution**: 53% yes, 19% no

Summary and analysis:

Writers primarily create web content that fits into a larger developer portal. A developer portal is a centralized hub that serves as the home for many different sets of documentation. The developer portal might have a federated search, a login where developers can get API keys or make other configurations, and navigation to browse the different doc sets and products.

Writers often help shape and build the developer portal. They might help design the site, workflows, strategies for content re-use, stylesheets, etc. For example, a common task might be to brand the static-site-generated documentation to fit seamlessly into a React-based developer portal, as well as to define the Git workflows around content collaboration and publishing.

Localization, video tutorials, and PDF aren’t overwhelmingly produced in developer docs but do constitute about 25% of the output. The low amount localization eases up some pressure on the tools. If you don’t have to push your content into translation management systems, you aren’t as constrained with compatible format types and roundtrip workflows. (It’s still possible to localize with static site generators, just not as easy.)

I asked questions about video in the survey because I had heard negative comments about video formats from some developers. Additionally, a lot of developer docs consist of code, which might be tough to demonstrate in a video (you basically watch someone code in real time, which can be tedious and feel either too slow or fast for the audience). However, the survey found that most writers aren’t opposed to creating video content. The main reason for not creating video is due to lack of bandwidth, constantly changing technology, or because no one has asked for video — not necessarily because of an aversion for video.

Finally, the number of writers generating PDFs surprised me. It’s not so easy to generate PDFs from docs-as-code tools, especially for more long-form content with cross-references and other book-style formatting. However, PDF continues to be an important output, probably because there isn’t a good alternative for distributing content to beta partners prior to release. With docs-as-code tools, you don’t often have an authentication layer to gate the login. In these scenarios, sending partners a pre-release PDF is usually the easiest way to share content.

3. Process and workflow responses

Survey summary:

- **How do you interact with Scrum teams**: 33% participate in limited capacity, 27% participate as full-fledged member, 19% have their own documentation Scrum
- **How do you review docs**: 25% code review tools, 19% in-person meetings, 14% collaborative annotation tools
- **How do engineers contribute to docs**: 31% pull requests, 31% wikis or similar, 22% direct repo rights
- **Do you outsource docs offshore**: 93% yes, 4% no
- **Do you publish docs with CI/CD**: 48% yes, 33% no, 15% plan to
- **Do you have a style guide**: 77% yes, 20% no

Summary and analysis:

Most writers participate on Scrum teams, sometimes in limited capacity; other times they have their own documentation Scrum teams. Scrum is the standard operating approach for most engineering groups (for better or worse), and technical writers plug into this rhythm for documentation as well.

Writers review docs often using the same tools engineers use to review code (e.g, code review tools that show diffs between commits). They also review docs through in-person meetings or through collaborative annotation tools like Google Docs.

The review process for docs has always been multi-pathed, and what works at one company might not work with another. Engineers often prefer to review content through code tools because it fits into how they're reviewing code, so they're accustomed to this approach. However, I find these tools exclude non-engineers, which weakens the review process — see my extended thoughts on this in Treat code like code and prose like prose.

Engineers contribute content either through pull requests to the doc source or by putting the content on a wiki or equivalent (for example, Google Docs, Quip). Other times engineers have direct rights in the repo to work with the content.

Note that the survey did not filter out documentation-writing engineers from dedicated technical writers. Many companies don't have the luxury of technical writers, so engineers often play roles as documentarians. In these cases, it would be natural for engineers to have rights in documentation repos, or to store documentation in the same repos as the code. (See Integration documentation into engineering code and workflows for a summary of an engaging presentation about how Google's internal doc team transformed their documentation by moving Markdown files directly inside of code repos.)

Outsourcing developer docs with an offshore authoring agency is rare. I did not ask for reasons why outsourcing isn't more common, but there might not be many outsourcing agencies that can handle highly technical developer content. Or perhaps there are IP concerns about documenting the internal workings of APIs, or maybe the doc shops are so small that no one would manage an outsourced resource.

The publishing process for developer docs is streamlined through continuous integration and continuous deployment (CI/CD). This means writers can kick off builds and deployments with a few keystrokes on a command line. For example, if you set up a service on a specific branch, when you push changes to the branch, the service can start a build process on the server and then deploy the build onto a server. (For example, GitHub Pages offers automatic builds of Jekyll projects on any GitHub repository. You could also do this through Travis CI.) You can also run other verification scripts, such as link checkers, in an automated way.

Some hosting and deployment solutions like Netlify let you push out multiple builds, allowing you to create different environments for your content (alpha, beta, prod), with different conditions exposing different content in each environment. The automated publishing model is one of the biggest advantages of the docs-as-code approach. It allows you to constantly iterate on your content because the bandwidth for republishing requires such little effort.

Finally, most tech writers working with developer docs follow a style guide. It’s worth noting here that style guides for dev docs often take into consideration many elements of API design. Enforcing API styles (such as parameter casing or endpoint names) isn’t too different from doc style guides (where you enforce rules about title casing and verb forms). Arnaud Lauret’s *The Design of Web APIs* goes into this topic in detail — see API design and usability (p. 488) for a summary of key points.

4. API responses

Survey summary:

- **Documentation usually involves an API**: 81% yes, 14% no
- **Most common type of API**: 56% REST APIs, 17% native library APIs (for example, Java, C++), 7% GraphQL, 7% SOAP
- **Use OpenAPI docs for REST APIs** 47% yes, 17% no, 16% sometimes
- **Who generates the OpenAPI spec**: 36% engineers, 26% both engineers and tech writers, 6% tech writers
- **Who generates native library API docs**: 32% engineers, 27% both engineers and tech writers, 6% tech writers
- **Create OpenAPI spec manually or auto-generate it:** 23% auto-generated, 22% manually, 22% both
- **How to render OpenAPI spec into documentation**: 27% Swagger UI, 17% internally built tools, 8% ReDoc
- **Most common programming languages to know**: 24% JavaScript, 17% Java, 16% Python
- **Trending technologies you’re documenting**: 13% machine learning, 11% artificial intelligence, 11% big data, 9% Internet of Things (IoT)

Summary and analysis:

Although the survey focused on developer documentation, not specifically API documentation, most developer docs involve some kind of API. As such, it’s fair to use *developer documentation* and *API documentation* somewhat synonymously, even if the latter is a subset of the former.

What kind of APIs are writers mostly working with? REST APIs, but only about half the time. Other times, writers work with native library APIs (such as Java or C++ APIs), GraphQL, or SOAP.

When documenting REST APIs, most teams use the OpenAPI specification. This is a detailed description of the API that follows a highly structured YAML or JSON format. Usually, either engineers create this spec, or engineers collaborate with tech writers on it. The same goes with reference documentation for native library APIs.

Reference docs have traditionally been written by engineers, so I imagine the collaboration here is usually one where writers edit the material rather than provide the annotations in source code. While engineers will often lead the charge with reference documentation, they rarely expand beyond this scope to tackle other elements of documentation, such as conceptual overviews, getting started guides, tutorials, how-to content, glossaries, troubleshooting, and FAQs.

In terms of processes for creating the OpenAPI spec, there's a split between manually creating the spec and auto-generating it from annotations in the source code. The former approach embraces the spec as a blueprint or contract that engineers code against; the latter is used more by engineering documentarians who might be wary of documentation drift, or who find it more convenient to keep documentation together with code.

The OpenAPI spec alone isn't readable documentation, but many tools can generate out documentation from the OpenAPI spec. The most common tools for this are Swagger UI (p. 218), custom-built tools, or ReDoc (p. 249).

REST APIs are language agnostic, but there are usually accompanying software development kits (SDKs) that are language-specific (companies provide them to help developers implement the API). The most important languages to know (likely because of the SDKs that accompany APIs) are JavaScript, Java, and Python. Outside of programming languages, trending technologies include machine learning, artificial intelligence, big data, and Internet of Things (IoT).

5. Profile information

Survey summary:

- **Age range**: 17% ages 36-40, 16% ages 31-35, 14% ages 26-30, 12% ages 41-45, 11% ages 56-50, 11% ages 56-60, 8% ages 51-55, 4% 61-65 ages
- **Gender**: 52% male, 46% female
- **Company**: 200+ different companies listed
- **College degree**: 31% humanities, 28% engineering, 15% tech comm
- **Satisfied with job**: 38% agree, 37% strongly agree
- **Team size**: 34% lone writer, 31% team size 2-4 writers, 16% team size 8+ writers, 12% team size 5-7 writers
- **Organizational model for tech comm**: 40% centralized, 21% decentralized, 19% hybrid/distributed
- **Employment type**: 86% full-time, 10% contractor/vendor/temp
- **Community you have most affinity towards**: 39% WTD, 31% none, 14% STC
- **Time spent learning to keep up**: 28% 30 min/day, 27% 20 min, 14% 60 min
- **Biggest challenges**: technical know-how, time/bandwidth, getting reviews, addressing both novice and advanced users

Summary and analysis:

This final section covers profile and demographics data about the survey respondents. First to note is that the age range for writers in developer docs is fairly evenly distributed. This is reassuring given that ageism is a valid concern for senior workers in the technology industry. (Apparently, there are even resorts where aging tech workers in Silicon Valley go ↗, some still in their 30s, to cope with anxiety about their increasing age.)

It seems the tech writer's age is much less relevant, perhaps because this role is seen as supportive to engineers rather than a role where risk-taking is essential (as might be required for young tech entrepreneurs trying to disrupt larger companies). For an age comparison with developers, the 2020 Stack Overflow Developer Survey reports that the average age of developers (using Stack Overflow) is about 33.7 years.

The gender balance among dev doc writers is also reassuring. The Stack Overflow Developer survey found that 91.5% of their respondents were men, 8.0% women. Tech has a bad reputation for its "brogrammer culture." In contrast, my survey found that the ratio for tech writers is 52% male / 46% female, which is much more balanced.

Another reassuring finding is that not everyone in this space is a former engineer. Instead, 31% have humanities degrees, 15% have technical communication degrees, and only 28% have engineering degrees. There's often a presumption that to excel in developer docs, you need to be a former developer. Or if you are a former developer, you're can automatically step to the front of the line. That doesn't seem to be the case.

Job satisfaction is also strong, with 75% of people agreeing or strongly agreeing that they are satisfied with their jobs. Compare this with the 70% job satisfaction rate reported in Saul Carliner's Tech Comm Census. Developer docs can be an intimidating space, where you're frequently documenting code that's hard to understand, where doc tools operate similar to software development tools, and where engineers have little patience to explain concepts to less technical people. Perhaps the job satisfaction is high because the salaries tend to be higher, the job market more abundant, and you're in a space where you're constantly learning.

Team sizes for writers in dev docs are small. A third are lone writers, and another third are on teams of 2-4 writers. Large teams of 8+ writers are less common, accounting for only 16% of respondents. Despite the small team sizes, 40% are centralized on a tech comm team within their company, while others are either decentralized (embedded and reporting directly within a product team), and others are in a hybrid model somewhere between centralized and decentralized.

As far as professional groups, more writers in this space have an affinity for Write the Docs, but many don't have an affinity for any professional group.

Finally, the biggest challenges writers in dev docs face is having enough technical know-how to write docs and enough time/bandwidth to write it. Getting engineers to review docs is also challenging, as is creating content that addresses both novice and advanced groups.

Conclusion

The survey didn't present any major surprises to the trends that I've already observed in this space. However, the answers provided more definitive data that confirms how different and unique developer docs are from other types of documentation. Technical writers transitioning into this space face a whirlwind of different tools, practices, and challenges. With this data, we can identify trends and see what standard practices are emerging. These trends can serve as a guide and reference as writers make their way in this space.

But also note that this space changes quickly. As JavaScript frameworks come and go, static site generators tend to follow suit, and what's trending one year might fade the next. This is a plastic space where new technologies and experimentation can lead to overnight change.

About the author

Tom Johnson is a senior technical writer for Amazon in Sunnyvale, California. He is best known for his blog, I'd Rather Be Writing, where he posts regularly on technical communication topics. The blog has one of the largest followings of technical communicators online. Additionally, he has created an extensive web API documentation course at [https://idratherbewriting.com/learnapidoc/] that has helped hundreds of technical writers transition into API documentation.

Sources

- 2020 Developer Survey. *Stack Overflow.*
- Abel, Scott. Slides: The State of Technical Communication: 2019. *The Content Wrangler.*
- Abel, Scott. Survey Reveals Top Tools, Trends, and Technologies in Use in Technical Communication Teams. *STC Intercom*. Dec 2018.
- Abel, Scott. Webinar: The State of Technical Communication: 2019. *The Content Wrangler.* BrightTALK. Dec 13, 2018.
- Bowles, Nellie. A New Luxury Retreat Caters to Elderly Workers in Tech (Ages 30 and Up). *New York Times*. Mar 4, 2019.
- Carliner, Saul and Chen, Yuan. Job and Career Satisfaction Among Technical Communicators. *STC Intercom*. Dec 2018.
- Carliner, Saul and Chen, Yuan. Professional Development of Technical Communicators. *STC Intercom*. Dec 2018.
- Carliner, Saul and Chen, Yuan. What Technical Communicators Do. Carliner, Saul and Chen, Yuan. *STC Intercom*. Jan 2019. *STC Intercom*. Dec 2018.
- Carliner, Saul and Chen, Yuan. Who Technical Communicators Are: A Summary of Demographics, Backgrounds, and Employment. *STC Intercom*. Dec 2018.
- Johnson, Tom. API the Docs recording: How Trends in API Documentation Differ from other Tech Comm Trends
- Johnson, Tom. 2020 Developer documentation survey. Idratherbewriting.com. Dec 31, 2019.
- Johnson, Tom. Developer Documentation Trends — Survey Results (p. 44)
- Johnson, Tom. *How* you write influences *what* you write — interpreting trends through movements from PDF to web, DITA, wikis, CCMSs, and docs-as-code. Idratherbewriting.com. Feb 20, 2020.
- Johnson, Tom. Integrating documentation into engineering code and workflows. Idratherbewriting.com. May 26, 2015.
- Johnson, Tom. Treat code like code and prose like prose. Idratherbewriting.com. Jun 16, 2020.
- Johnson, Tom. API design and usability (p. 488). Idratherbewriting.com.
- Lauret, Arnaud. *The Design of Web APIs*. Manning Publications. 2019.
- Welinske, Joe. 2014 WritersUA Tools Survey. *WritersUA.* Aug 20, 2015.

Glossary

The API documentation landscape is full of jargon, acronyms, and many new terms. This glossary provides a list of terms and definitions.

A (p. 55) | B (p. 56) | C (p. 56) | E (p. 56) | G (p. 57) | H (p. 57) | J (p. 58) | M (p. 58) | O (p. 58) | P (p. 59) | Q (p. 60) | R (p. 60) | S (p. 61) | V (p. 62) | Y (p. 63) |

A

API

Application Programming Interface. Enables different systems to interact with each other programmatically. Two types of APIs are REST APIs (web APIs) and native-library APIs. See What is a REST API? (p. 36)

API Console

Renders an interactive display for the RAML spec. Similar to Swagger UI, but for RAML (p. 936). See github.com/mulesoft/api-console.

APIMATIC

Supports most REST API description formats (OpenAPI, RAML, API Blueprint, etc.) and provides SDK code generation, conversions from one spec format to another, and many more services. APIMATIC “lets you define APIs and generate SDKs for more than 10 languages.” For example, you can automatically convert Swagger 2.0 to 3.0 using the API Transformer service on this site. See https://apimatic.io/ and read their documentation.

API Transformer

A cross-platform service provided by APIMATIC that will automatically convert your specification document from one format or version to another. See apimatic.io/transformer.

Apiary

Platform that supports the full life-cycle of API design, development, and deployment. For interactive documentation, Apiary supports the API Blueprint specification, which similar to OpenAPI or RAML but includes more Markdown elements. It also supports the OpenAPI specification now too. See apiary.io.

API Blueprint

The API Blueprint spec is an alternative specification to OpenAPI or RAML. API Blueprint is written in a Markdown-flavored syntax. See API Blueprint (p. 938) in this course, or go to API Blueprint's homepage to learn more.

Apigee

Similar to Apiary, Apigee provides services for you to manage the whole lifecycle of your API. Specifically, Apigee lets you “manage API complexity and risk in a multi- and hybrid-cloud world by ensuring security, visibility, and performance across the entire API landscape.” Supports the OpenAPI spec. See apigee.com.

Asciidoc

A lightweight text format that provides more semantic features than Markdown. Used in some static site generators, such as Asciidoctor ↗ or Nanoc ↗. See http://asciidoc.org/ ↗.

B

branch

In Git, a branch is a copy of the repository that is often used for developing new features. Usually, you work in branches and then merge the branch into the master branch when you're ready to publish. If you're editing documentation in a code repository, developers will probably have you work in a branch to make your edits. The developers will then either merge your branch into the master when ready, or you might submit a pull request to merge your branch into the master. See git-branch ↗.

C

clone

In Git, clone is the command used to copy a repository in a way that keeps it linked to the original. The first step in working with any repository is to clone the repo locally. Git is a distributed version control system, so everyone working in it has a local copy (clone) on their machines. The central repository is referred to as the origin. Each user can pull updates from origin and push updates to origin. See git-clone ↗.

commit

In Git, a commit is when you take a snapshot of your changes to the repo. Git saves the commit as a snapshot in time that you can revert to later if needed. You commit your changes before pulling from origin or before merging your branch within another branch. See git-commit ↗.

CRUD

Create, Read, Update, Delete. These four programming operations are often compared to POST, GET, PUT, and DELETE with REST API operations.

curl

A command line utility often used to interact with REST API endpoints. Used in documentation for request code samples. curl is usually the default format used to display requests in API documentation. See curl ↗. Also written as cURL. See Make a curl call (p. 80) and Understand curl more (p. 83).

E

endpoints and methods

The endpoints indicate how you access the resource, while the method indicates the allowed interactions (such as GET, POST, or DELETE) with the resource.

The same resource usually has a variety of related endpoints, each with different paths and methods but returning different information about the same resource. Endpoints usually have brief descriptions similar to the overall resource description but shorter. Also, the endpoint shows the end path of a resource URL only, not the base path common to all endpoints. See Endpoints and methods (p. 126).

G

Git

Distributed version control system commonly used when interacting with code. GitHub uses Git, as does BitBucket and other version control platforms. Learning Git is essential for working with developer documentation, since this is the most common way developers share, review, collaborate, and distribute code. See https://git-scm.com/ ↗.

GitHub

A platform for managing Git repositories. Used for most open-source projects. You can also publish documentation using GitHub, either by simply uploading your non-binary text files to the repo, or by auto-building your Jekyll site with GitHub Pages, or by using the built-in GitHub wiki. See GitHub wikis (p. 569) in this course as well as on pages.github.com/ ↗.

Git repo

In Git, a repo (short for repository) stores your project’s code. Usually, you only store non-binary (human-readable) text files in a repo, because Git can run diffs on text files and show you what has changed.

H

HAT

Help Authoring Tool. Refers to the traditional help authoring tools (RoboHelp, Flare, Author-it, etc.) used by technical writers for documentation. Tooling for API docs tends to use docs-as-code tools (p. 551) more than HATs (p. 533).

HATEOS

Stands for Hypermedia as the Engine of Application State. Hypermedia is one of the characteristics of REST that is often overlooked or missing from REST APIs. In API responses, responses that span multiple pages should provide links for users to page to the other items. See HATEOS ↗.

Header parameters

Parameters that are included in the request header, usually related to authorization.

Hugo

A static site generator that uses the Go programming language as its base. Along with Jekyll, Hugo is among the top 5 most popular static site generators. Hugo is probably the fastest site generator available. Speed matters as you scale the number of documents in your project beyond several hundred. See https://gohugo.io/ ↗. For more about static site generators, see Static site generators (p. 591).

J

JSON

JavaScript Object Notation. A lightweight syntax containing objects and arrays, usually used (instead of XML) to return information from a REST API. See Analyze the JSON response (p. 94) in this course and http://www.json.org/ ↗.

M

Mercurial

A distributed revision control system, similar to Git but not as popular. See https://www.mercurial-scm.org/ ↗.

method

The allowed operation with a resource in terms of GET, POST, PUT, DELETE, and so on. These operations determine whether you’re reading information, creating new information, updating existing information, or deleting information. See Endpoints and methods (p. 126).

Mulesoft

Similar to Apiary or Apigee, Mulesoft provides an end-to-end platform for designing, developing, and distributing your APIs. For documentation, Mulesoft supports RAML (p. 924). See https://www.mulesoft.com/ ↗.

O

OAS

Abbreviation for OpenAPI specification.

OpenAPI

The official name for the OpenAPI specification. The OpenAPI specification provides a set of properties that can be used to describe your REST API. When valid, the specification document can be used to create interactive documentation, generate client SDKs, run unit tests, and more. You can read the specification details on GitHub at https://github.com/OAI/OpenAPI-Specification ↗. Under the Open API Initiative with the Linux Foundation, the OpenAPI specification aims to be vendor neutral (many companies steer its development ↗, not just one). See Introduction to the OpenAPI specification (p. 181).

OpenAPI contract

Synonym for OpenAPI specification document.

OpenAPI specification document

The file (either in YAML or JSON syntax) that describes your REST API. Follows the OpenAPI specification format. See https://www.openapis.org/ ↗. See also OpenAPI 3.0 tutorial (p. 272).

OpenAPI Initiative

The governing body that directs the OpenAPI specification. Backed by the Linux Foundation. See https://www.openapis.org/ ↗.

P

parameters

Parameters are options you can pass with the endpoint (such as specifying the response format or the amount returned) to influence the response. There are several types of parameters: header parameters, path parameters, and query string parameters. Request bodies are closely similar to parameters but are not technically a parameter.

The different types of parameters are often documented in separate groups on the same page. Not all endpoints contain each type of parameter. See Parameters (p. 132) for more details.

Path parameters

Parameters that appear within the path of the endpoint, before the query string (`?`). Path parameters are usually set off within curly braces. See Parameters (p. 132) for more details.

Pelican

A static site generator based on Python. See https://github.com/getpelican/pelican ↗. For more about static site generators, see Static site generators (p. 591).

Perforce

Revision control system often used before Git became popular. Often configured as a centralized repository instead of a distributed repository. See Perforce ↗.

pull

In Git, when you pull from origin (the master location where you cloned the repo), you get the latest updates from origin onto your local system. When you run `git pull`, Git tries to automatically merge the updates from origin into your copy. If the merge cannot happen automatically, you might see merge conflicts. See git-pull ↗.

Pull Request

A request from an outside contributor to merge a cloned branch back into the master branch. The pull request workflow is commonly used with open-source projects because developers outside the team will not usually have contributor rights to merge updates into the repository. GitHub provides a user-friendly interface for making and processing pull requests. See Pull Requests ↗.

push

In Git, when you want to update the origin with the latest updates from your local copy, you run `git push`. Your updates will bring origin back into sync with your local copy. See https://git-scm.com/docs/git-push ↗.

Q

Query string parameters

Parameters that appear in the query string of the endpoint, after the `?`. See Parameters (p. 132) for more details.

R

RAML

Stands for REST API Modeling Language and is similar to OpenAPI specifications. RAML is backed by Mulesoft, a commercial API company, and uses a more YAML-based syntax in the specification. See RAML tutorial (p. 924) in this course or RAML.

RAML Console

In Mulesoft, the RAML Console is where you design your RAML spec. Similar to the Swagger Editor for the OpenAPI spec.

Repo

A tool for consolidating and managing many smaller repos with one system. See git-repo. For the definition of a Git repository, see Git repo (p. 57).

request

The way information is returned from an API. In a request, the client provides a resource URL with the proper authorization to an API server. The API returns a response with the information requested. See Request example (p. 141) for more details.

request body

A block of information, usually JSON, included in the request body, often used with CREATE and PUT methods. See Parameters (p. 132) for more details.

response example and schema

The response example shows a sample response from the request example; the response schema defines all possible elements in the response. The response example is not comprehensive of all parameter configurations or operations, but it should correspond with the parameters passed in the request example. The response lets developers know if the resource contains the information they want, the format, and how that information is structured and labeled.

The description of the response is known as the response schema. The response schema documents the response in a more comprehensive, general way, listing each property that could possibly be returned, what each property contains, the data format of the values, the structure, and other details. See Response example and schema (p. 154) for details.

resource description

“Resources” refers to the information returned by an API. Most APIs have various categories of information, or various resources, that can be returned.

The resource description is brief (1-3 sentences) and usually starts with a verb. Resources usually have various endpoints to access the resource and multiple methods for each endpoint. On the same page, you usually have a general resource described along with a number of endpoints for accessing the resource, also described. See Resource description (p. 120) for more details.

response

The information returned by an API after a request is made. Responses are usually in either JSON or XML format. See Response example and schema (p. 154) for details.

response example and schema

The response example shows a sample response from the request example; the response schema defines all possible elements in the response. The response example is not comprehensive of all parameter configurations or operations, but it should correspond with the parameters passed in the request example. The response lets developers know if the resource contains the information they want, the format, and how that information is structured and labeled.

The description of the response is known as the response schema. The response schema documents the response in a more comprehensive, general way, listing each property that could possibly be returned, what each property contains, the data format of the values, the structure, and other details. See Response example and schema (p. 154) for details.

REST API

Stands for Representational State Transfer. Uses web protocols (HTTP) to make requests and provide responses in a language agnostic way, meaning that users can choose whatever programming language they want to make the calls. See What is a REST API? (p. 36) for more details.

S

SDK

Software development kit. Developers often create an SDK to accompany a REST API. The SDK helps developers implement the API using a specific language, such as Java or PHP. See SDKs (p. 480) for more details.

Smartbear

The company that maintains and develops the Swagger tooling — Swagger Editor, Swagger UI, Swagger Codegen, SwaggerHub, and others. See Smartbear.

Sphinx

A static site generator developed for managing documentation for Python. Sphinx is the most documentation-oriented static site generator available and includes many robust features – such as search, sidebar navigation, semantic markup, managed links – that other static site generators lack. Based on Python. See staticgen.com/sphinx for high-level details. For more about static site generators, see Static site generators (p. 596).

Static site generator

A breed of website compilers that package up a group of files (usually written in Markdown) and make them into a website. There are more than 350 different static site generators. See Jekyll (p. 637) in this course for a deep-dive into the most popular static site generator, or Staticgen for a list of all static site generators. See Static site generators (p. 591) for a deep-dive into this topic.

Stoplight

Provides a platform with visual modeling tools to create an OpenAPI document for your API — without requiring you to know the OpenAPI spec details or code the spec line by line. See http://stoplight.io/ for more information. See Stoplight — visual modeling tools for creating your OpenAPI spec (p. 195) for more information.

Swagger

Refers to API tooling related to the OpenAPI spec. Some of these tools include Swagger Editor, Swagger UI, Swagger Codegen, SwaggerHub, and others. These tools are managed by Smartbear. For more tools, see Swagger Tools. "Swagger" was the original name of the OpenAPI spec, but the name was later changed to OpenAPI to reinforce the open, non-proprietary nature of the standard. People sometimes refer to both names interchangeably (especially on older web pages), but "OpenAPI" is how the spec should be referred to. For more on naming conventions between OpenAPI and Swagger, see What Is the Difference Between Swagger and OpenAPI?.

Swagger Codegen

Generates client SDK code for a lot of different platforms (such as Java, JavaScript, Scala, Python, PHP, Ruby, Scala, and more). The client SDK code helps developers integrate your API on a specific platform and provides for more robust implementations that might include more scaling, threading, and other necessary code. In general, SDKs are toolkits for implementing the requests made with an API. Swagger Codegen generates the client SDKs in nearly every programming language. See Swagger Codegen for more information. See also SDKs (p. 480).

Swagger Editor

An online editor that validates your OpenAPI document against the rules of the OpenAPI specification. The Swagger Editor will flag errors and give you formatting tips. See Swagger Editor.

Swagger UI

An open-source web framework (on GitHub) that parses an OpenAPI specification document and generates an interactive documentation website. Swagger UI is the tool that transforms your spec into the Petstore-like site.

SwaggerHub

A site developed by Smartbear to help teams collaborate around the OpenAPI spec. In addition to generating interactive documentation from SwaggerHub, you can generate many client and server SDKs and other services. See Manage Swagger Projects with SwaggerHub (p. 230).

V

VCS

Stands for version control system. Git and Mercurial are examples.

version control system

A system for managing code that relies on snapshots that store content at specific states. Enables you to revert to previous states, branch the code into different versions, and more. See About Version Control ↗ in Git. See Version Control Systems (p. 562) for more information.

Y

YAML

Recursive acronym for “YAML Ain’t No Markup Language.” A human-readable, space-sensitive syntax used in the OpenAPI specification document. See More About YAML (p. 276).

Chapter 2: Using an API like a developer

Playing a brief role as a developer will help you understand developer needs better, as well as what developers typically look for in API documentation. Developers often use tools such as Postman or curl to make calls. They look at the structure of the response, and they dynamically integrate the needed information into web pages and other applications.

Chapter sections:

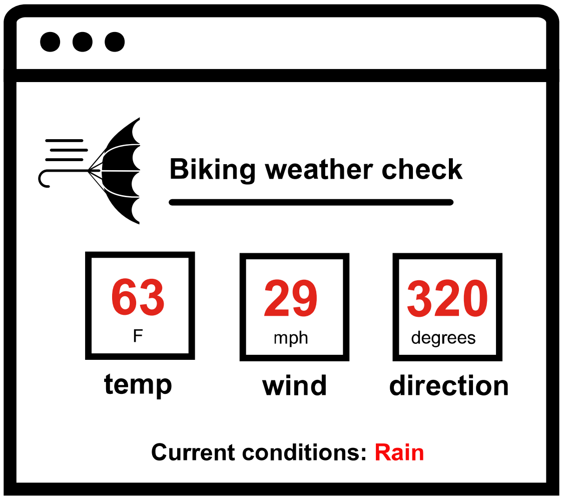
Scenario for using a weather API 65
Get authorization keys 69
Submit requests through Postman 71
curl intro and installation 78
Make a curl call 80
Understand curl more 83
Activity: Use methods with curl 88
Analyze the JSON response 94
Inspect the JSON from the response payload 98
Access and print a specific JSON value 103
Dive into dot notation 109

Scenario for using a weather API

Let's start using an actual REST API to get more familiar with how they work. In the upcoming sections, you'll explore some weather APIs in the context of a specific use case: retrieving a weather forecast. By first playing the role of a developer using an API, you'll gain a greater understanding of how your audience will use APIs, the type of information they'll need, and what they might do with the information.

Sample scenario: How windy is it?

Let's say that you're a web developer and you want to add weather information to your site. Your site is for cyclists. You want to allow users who come to your site to see the current wind and temperature conditions. You want something like this:



Wind meter conditions for website

You don't have your own meteorological service, so you'll need to make some calls out to a weather service to get this information. Then you will present that information to users.

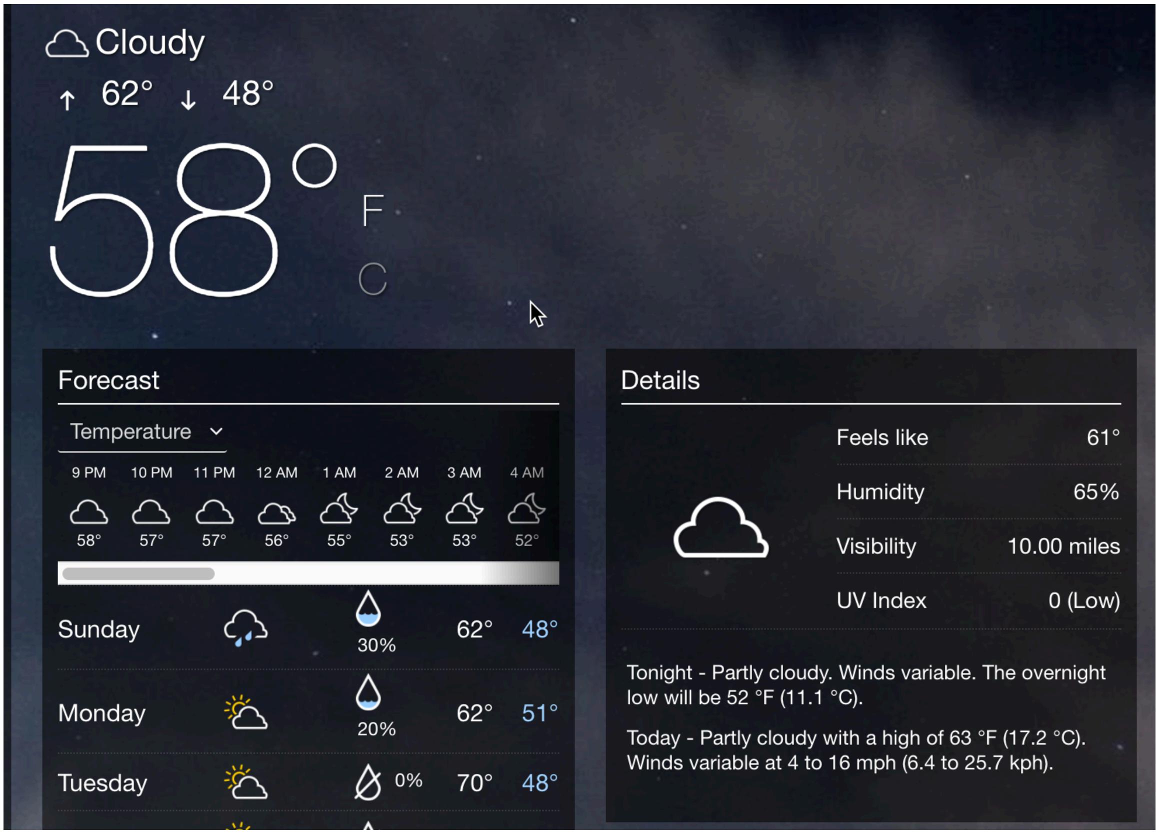
Get an idea of the end goal

To give you an idea of the end goal, here's a sample: idratherbewriting.com/assets/files/wind-openweathermap.html. It's not necessarily styled the same as the mockup, but it answers the question, "What's the current wind and temperature?"

Click the **Check wind conditions** button to see wind and temperature details. When you request this data, a call is made to the OpenWeatherMap API. The API retrieves the information, and some JavaScript code prints the information on the page.

Note: This content doesn't embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/docapis_scenario_for_using_weather_api.html] to view the content.

The above example is extremely simple. You could also build an attractive interface like this:



9 PM	10 PM	11 PM	12 AM	1 AM	2 AM	3 AM	4 AM
58°	57°	57°	56°	55°	53°	53°	52°

Sunday	30%	62°	48°
Monday	20%	62°	51°
Tuesday	0%	70°	48°

Feels like	61°
Humidity	65%
Visibility	10.00 miles
UV Index	0 (Low)

Tonight - Partly cloudy. Winds variable. The overnight low will be 52 °F (11.1 °C).

Today - Partly cloudy with a high of 63 °F (17.2 °C). Winds variable at 4 to 16 mph (6.4 to 25.7 kph).

The concept and general techniques are more or less the same. Behind the scenes, the code makes requests to a weather service API and then displays the returned information in a stylized user interface.

Activity: Explore the OpenWeatherMap API

Although there are many good weather API options for developers, I decided to use the OpenWeatherMap API in this course because the service is easy to use, free, and stable.

Let's explore the basic sections in the OpenWeatherMap API:

1. Go to the openweathermap.org.
2. Click **API** in the top navigation bar.
3. In the **Current weather data** section, click the **API doc** button.
4. Click some of the links in the **Examples of API calls** sections.

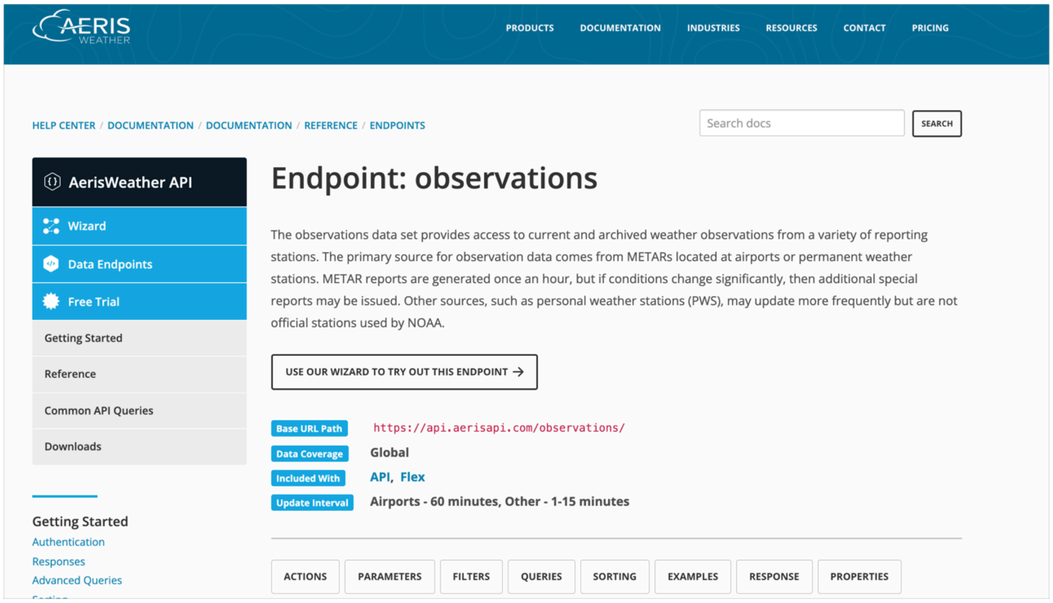
   Get a sense of the information this Current Weather Data API provides. The API calls provide developers with ways to pull information into their applications. In other words, the APIs will provide the data plumbing for the applications that developers build.

5. Answer the following questions about the Current Weather Data API endpoint:
   - Does the API provide the information we need about temperature, wind speed, wind direction, and current conditions?
   - How many different ways can you specify the location for the weather information?
   - What does a sample request look like?
   - How many endpoints does the API have?
   - What authorization credentials are required to get a response?

Explore the Aeris Weather API

Before diving too far down in the OpenWeatherMap API, let's look at another weather API for contrast. In contrast to the OpenWeatherMap API, the Aeris Weather API is a bit more robust and extensive. Explore the Aeris Weather API by doing the following:

1. Go to www.aerisweather.com.
2. Click **Documentation** on the top navigation bar. Then click **API Documentation**.
3. In the left sidebar, under Reference, click **Endpoints**. Select the **All** tab to show all endpoints, and then scroll down a bit and click **observations**.



Endpoint: observations

The observations data set provides access to current and archived weather observations from a variety of reporting stations. The primary source for observation data comes from METARs located at airports or permanent weather stations. METAR reports are generated once an hour, but if conditions change significantly, then additional special reports may be issued. Other sources, such as personal weather stations (PWS), may update more frequently but are not official stations used by NOAA.

Base URL Path	`https://api.aerisapi.com/observations/`
Data Coverage	Global
Included With	API, Flex
Update Interval	Airports - 60 minutes, Other - 1-15 minutes

4. Browse the type of information that is available through this endpoint. Does this endpoint provide information about wind and temperature that would work for our sample development scenario?

Here's the Aeris weather forecast API in action making mostly the same calls as I showed earlier with OpenWeatherMap: idratherbewriting.com/assets/files/wind-aeris.html. (Note: If this returns `[Object][Object]`, it's because my API key expired. You can view the page source to see how the call is constructed.)

For our development scenario (showing weather on a website), there are dozens of different weather APIs we could use. As you create your API documentation, think about how your users might have to make the same decisions in choosing APIs. Are there several APIs that provide similar information? What will make your API stand out more? Although you probably can't pick and choose what information your API returns, you might at least argue that the docs for your API will be superior!

More weather APIs

APIs differ considerably in their design, presentation, responses, and other detail. For more comparison, check out some of the following weather APIs:

- Accuweather API
- Weatherbit API
- ClimaCell's Weather API

Each weather API has an entirely different approach to documentation. As you’ll see going through this course, the variety and uniqueness of each API doc site (even when approaching the same topic — a weather forecast) presents a lot of challenges to tech writing teams. Not only do website styles vary but also API terminology and other vocabulary to describe similar concepts.

As I mentioned earlier, REST APIs are an architectural style that follows common characteristics and principles; they don’t all follow the same standard or specification. Users have to read the documentation to understand how to use the API. (See “REST is a style, not a standard.” (p. 39))

Spend a little time exploring the features and information that these weather APIs provide. Try to answer these basic questions:

- What does each API do?
- How many endpoints does each API have?
- What information do the endpoints provide?
- What kind of parameters does each endpoint take?
- What kind of response does the endpoint provide?

Sometimes people use the term "API" to refer to a whole collection of endpoints, functions, or classes. Other times they use API to refer to a single endpoint. For example, a developer might say, "We need you to document a new API." They mean they added a new endpoint or class to the API, not that they launched an entirely new API service.

Get authorization keys

Almost every API has a method in place to authenticate requests. You usually have to provide an API key in your requests to get a response. Although we'll dive into authentication and authorization (p. 393) later, we need to get some API keys now to make requests to our weather API.

Why requests need authorization

Requiring authorization allows API publishers to do the following:

- License access to the API
- Rate limit the number of requests
- Control availability of certain features within the API, and more

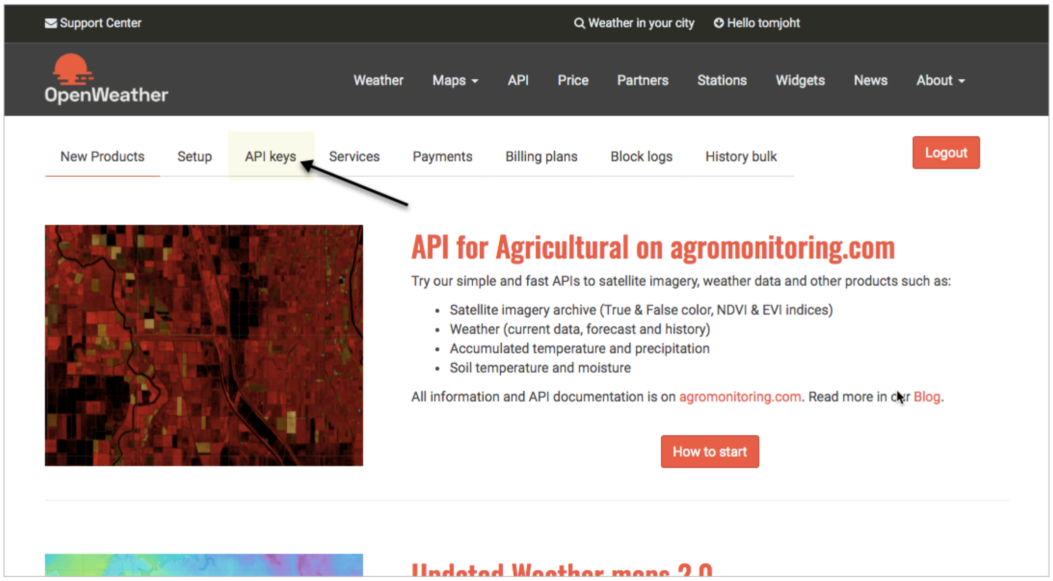
To run the code samples in this course, you will need to use your own API keys, since these keys are usually treated like passwords and not given out or published openly on a web page.

Activity: Get an OpenWeatherMap API key

To get an API key for the OpenWeatherMap API:

1. Go to openweathermap.org.
2. Click **Sign Up** in the top navigation bar and create an account.
3. After you sign up, your API key is sent to the email address you provide.

   You can also find your API key on the site's Developer Dashboard. (To find your API key on the site, return to the OpenWeatherMap homepage and click **Sign in**. After signing in, you'll see the developer dashboard. Click the **API Keys** tab (highlighted in the screenshot below).



API for Agricultural on agromonitoring.com

Try our simple and fast APIs to satellite imagery, weather data and other products such as:

- Satellite imagery archive (True & False color, NDVI & EVI indices)
- Weather (current data, forecast and history)
- Accumulated temperature and precipitation
- Soil temperature and moisture

All information and API documentation is on agromonitoring.com. Read more in our Blog.

API Keys tab on OpenWeatherMap Developer Dashboard

4. Copy the key to a place you can easily find it.

(Note: It can take an hour or so for a new OpenWeatherMap API key to activate.)

Get the Aeris Weather API secret and ID

Now for contrast, let's get the keys for the Aeris Weather API, using a free developer account. The Aeris Weather API requires both a secret and ID to make requests.

1. Go to https://www.aerisweather.com/develop/api/ and click **Free Trial**.
2. Click **Sign Up for Developer**. (The free developer version limits the number of requests you can make and expires in 30 days.)
3. Complete the required fields to sign in. Click the activation email as well.
4. After you sign up for an account, sign in and go to your Dashboard.
5. Click **Apps**, and then click **New Application**.
6. In the Add a New Application dialog box, enter the following:
   - **Application Name**: My biking app (or something)
   - **Application Namespace**: localhost

7. Click **Save App**.

After registering your app, you should see an ID, secret, and namespace for the app. Copy this information into a place you can easily access since you'll need it to make requests.

Keep in mind how users authorize calls with an API — this is something you usually cover in API documentation. Later in the course, we will dive into authorization methods (p. 393) in more detail.

Make sure you have a text editor

In the upcoming activities, you'll work with code in a text file. When you're working with code, you use a text editor (to work in plain text) instead of a rich text editor (which would provide a WYSIWYG interface). Here are a few choices for text editors:

- Visual Studio Code (Mac or PC). *(I recommend this editor the most.)*
- Sublime Text (Mac or PC)
- TextWrangler or BBEdit (Mac)
- WebStorm (Mac or PC)
- Notepad++ (PC)
- Atom (Mac or Windows)
- Komodo Edit (Mac or PC)
- Coda (Mac)

These editors provide features that let you better manage the text. Choose the one you want. (My preference is to use Sublime Text when I'm working with independent code samples, and Atom when I'm working with Jekyll projects.) Avoid using TextEdit since it adds some formatting behind the scenes that can corrupt your content.

Submit requests through Postman

When you're testing endpoints with different parameters, you can use one of the many GUI REST clients available to make the requests. (By "GUI," I mean there's a graphical user interface with boxes and buttons for you to click.) You can also use curl (p. 78) (which we'll cover soon), but GUI clients tend to simplify testing with REST APIs.

Why use a GUI client

With a GUI REST client, you can:

- Save your requests (and numerous variations) in a way that's easy to run again
- More easily enter information in the right format
- See the response in a prettified JSON view or a raw format
- Easily include header information

With a GUI REST client, you won't have to worry about getting curl syntax right and analyzing requests and responses from the command line.

Popular GUI clients

Some popular GUI clients include the following:

- Postman
- Insomnia
- Paw
- Advanced REST Client (Chrome browser extension)

Of the various GUI clients available, Postman is probably the best option, since it allows you to save both calls and responses, is free, works on both Mac and PC, and is easy to configure.

A lot of times, abstract concepts don't make sense until you can contextualize them with an action. In this course, I'm following more of an "experience-first" methodology. After you do an activity, we'll explore the concepts in more depth. So if it seems like I'm glossing over concepts now, such as what a GET method is or an endpoint, hang in there. When we deep dive into these points in Documenting API endpoints (p. 114), these concepts will be a lot clearer.

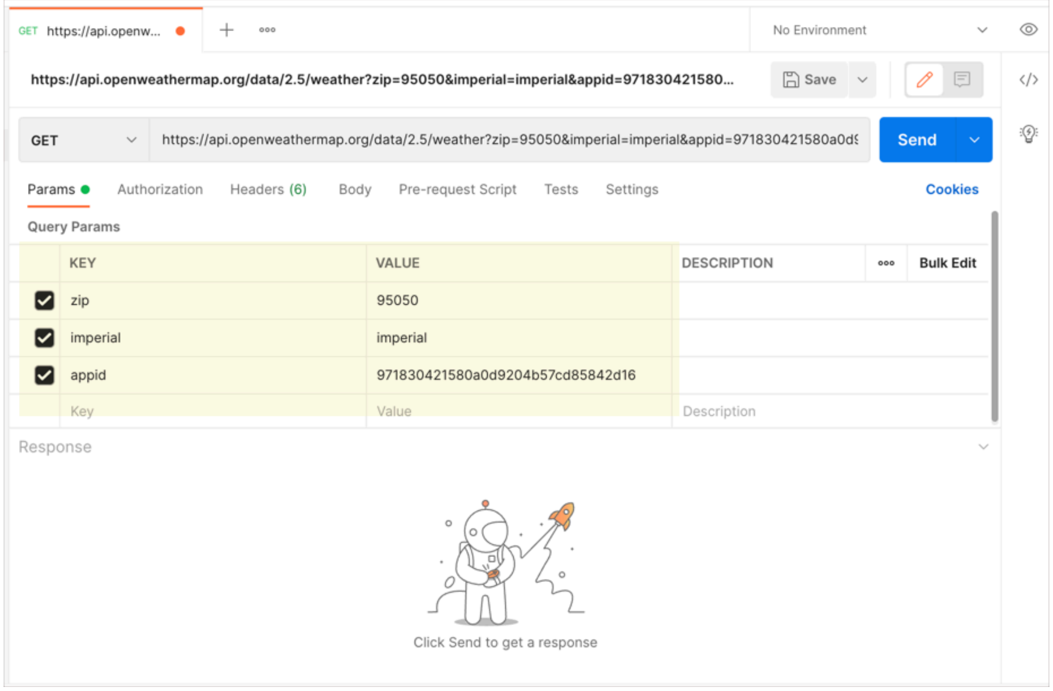
Activity: Make requests with Postman

In this exercise, you'll use Postman to make a request using OpenWeatherMap's current weather data API endpoint. To make the request:

1. If you haven't already done so, download and install the Postman app at https://www.getpostman.com/downloads/. (Make sure you download the app and not the deprecated Chrome extension.)
2. Start the Postman app and sign in when prompted. Close any welcome screens so you can make a request.
3. Insert the following endpoint into the box next to **GET**: `https://api.openweathermap.org/data/2.5/weather`
4. Click the **Params** tab (below the box where you inserted the endpoint) and then add the following three parameters in the **key** and **value** rows:
    - key: `zip` / value: `95050`

  - key: `units` / value: `imperial`
  - key: `appid` / value: <insert your own API key>

For the value for `appid`, use your own API key. Your Postman UI should look like this:



https://api.openweathermap.org/data/2.5/weather?zip=95050&imperial=imperial&appid=971830421580...

GET https://api.openweathermap.org/data/2.5/weather?zip=95050&imperial=imperial&appid=971830421580a0d

Query Params

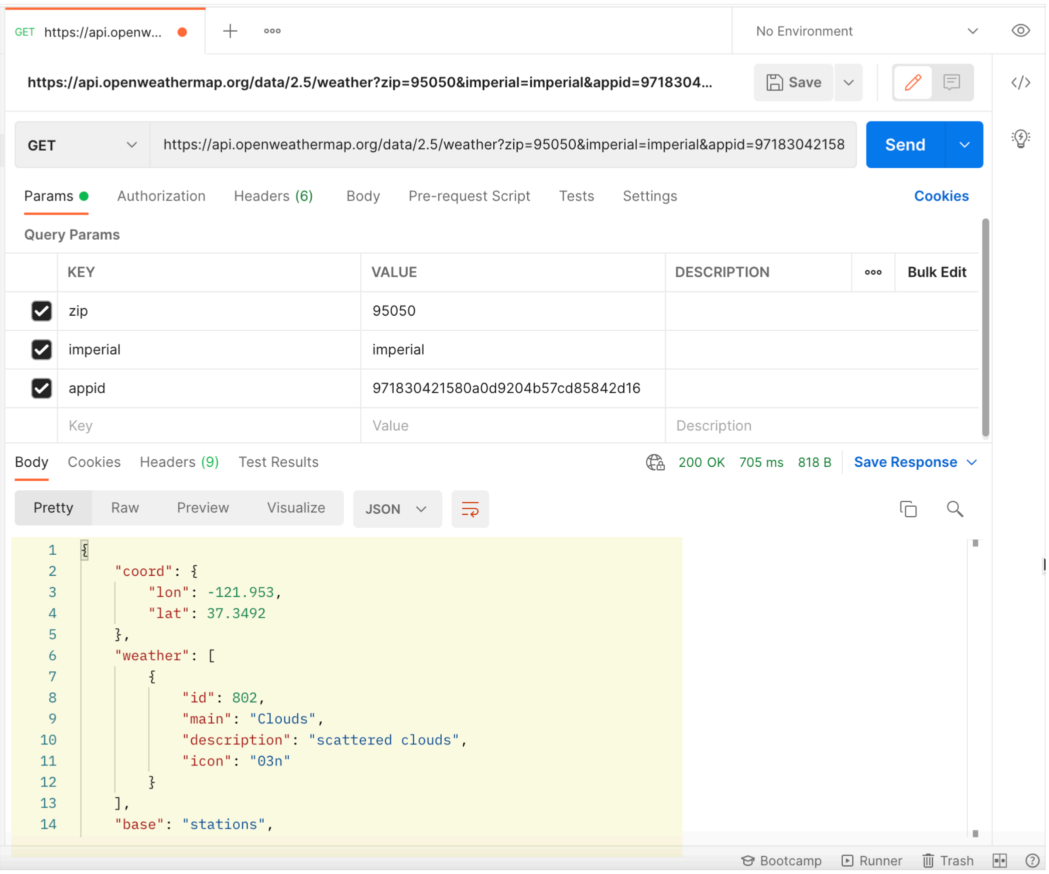
	KEY	VALUE	DESCRIPTION
☑	zip	95050	
☑	imperial	imperial	
☑	appid	971830421580a0d9204b57cd85842d16	
	Key	Value	Description

When you add these parameters, they appear as a query string to the endpoint URL in the GET box. For example, your endpoint will now look like this: `https://api.openweathermap.org/data/2.5/weather?zip=95050&units=imperial&appid=APIKEY` (but with different query string values and with your own API key instead of `APIKEY`). Query string parameters appear after the question mark `?` symbol and are separated by ampersands `&`. The order of query string parameters doesn't matter.

Note that many APIs pass the API key in the header rather than as a query string parameter in the request URL. (If that were the case, you would click the **Headers** tab and insert the required key-value pairs in the header. But OpenWeatherMap passes the API key as a query string parameter.)

5. Click **Send**.

   The response appears in the lower pane. For example:



https://api.openweathermap.org/data/2.5/weather?zip=95050&imperial=imperial&appid=9718304...

GET https://api.openweathermap.org/data/2.5/weather?zip=95050&imperial=imperial&appid=97183042158

Query Params

KEY	VALUE	DESCRIPTION
☑ zip	95050	
☑ imperial	imperial	
☑ appid	971830421580a0d9204b57cd85842d16	
Key	Value	Description

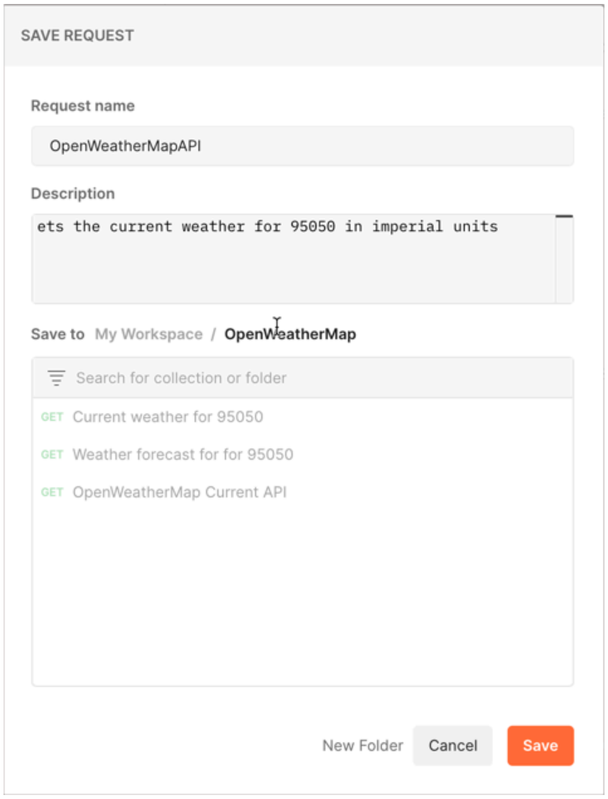
200 OK 705 ms 818 B

```
{
    "coord": {
        "lon": -121.953,
        "lat": 37.3492
    },
    "weather": [
        {
            "id": 802,
            "main": "Clouds",
            "description": "scattered clouds",
            "icon": "03n"
        }
    ],
    "base": "stations",
```

Save the request

1. In Postman, click the **Save** button (above Send). The Save Request dialog box appears.
2. In the **Request name** box, type a friendly name for the request, such as “OpenWeatherMap Current API.”
3. In the **Request description (Optional)** field, type a description such as “gets the current weather for 95050 in imperial units.”
4. Scroll down a bit and click **New Collection** to create a folder to save the request in. Name your new collection (e.g., “OpenWeatherMap”) and click the orange check mark. Then select the new collection you just created.

   After you create the collection, the Save button will be enabled. Your Postman collection should look something like this:


SAVE REQUEST

Request name: OpenWeatherMapAPI

Description: ets the current weather for 95050 in imperial units

Save to My Workspace / OpenWeatherMap

Search for collection or folder
GET Current weather for 95050
GET Weather forecast for for 95050
GET OpenWeatherMap Current API
New Folder
Cancel
Save

Collection dialog box

5. Click **Save**.

   Saved requests appear in the left side pane on the Collections tab. (If you don't see the Collections pane, click the **Show Sidebar** button in the lower-left corner to expand it.

(Optional) Make a request for the OpenWeatherMap 5 day forecast

Now instead of getting the current weather, let's use another OpenWeatherMap endpoint to get the forecast. Enter details into Postman for the 5 day forecast request. In Postman, you can click a new tab, or click the arrow next to Save and choose **Save As**. Then choose your collection and request name.

A sample endpoint for the 5 day forecast, which specifies location by zip code, looks like this:

```
https://api.openweathermap.org/data/2.5/forecast?zip=95050,us
```

Add in the query parameters for the API key and units:

```
https://api.openweathermap.org/data/2.5/forecast?zip=95050&appid=APIKEY&unit
s=imperial
```

(In the above code, replace `APIKEY` with your own API key.)

Observe how the response contains a `list` that provides the forecast details for five days.

(Optional) Make one more OpenWeatherMap API request

Make one more OpenWeatherMap API request, this time changing the way you specify the location. Instead of specifying the location by zip code, specify the location using `lat` and `lon` geocoordinates instead. For example:

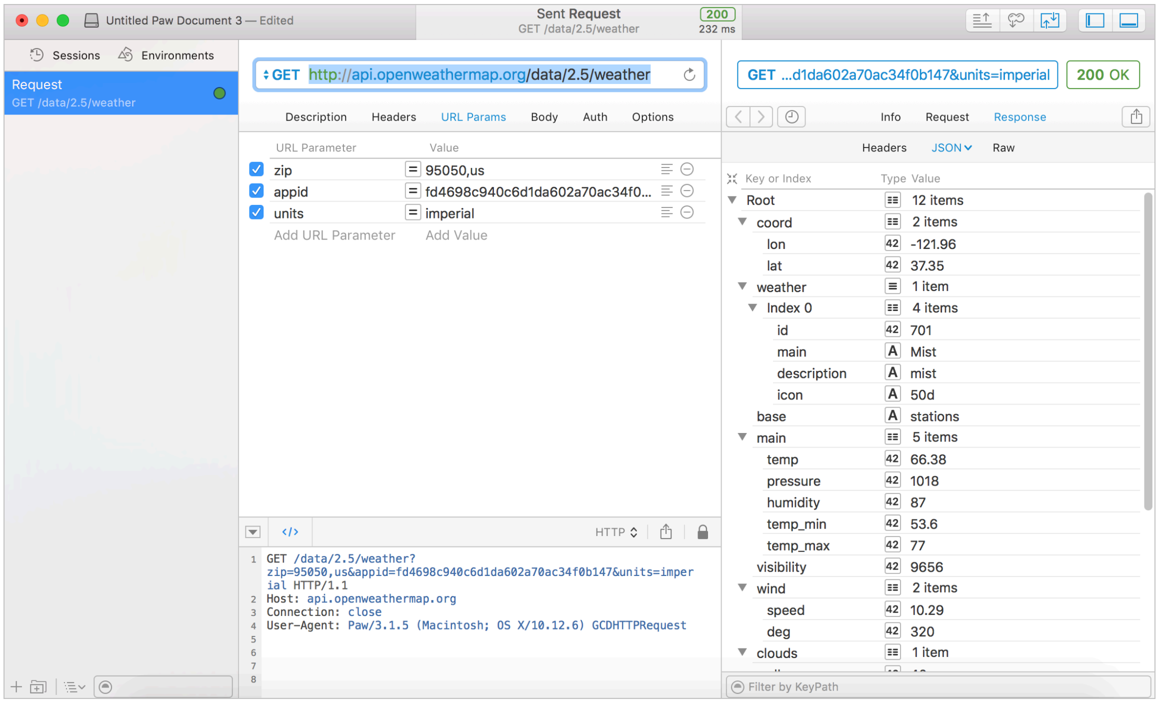
```
https://api.openweathermap.org/data/2.5/weather?lat=37.3565982&lon=-121.9689
848&units=imperial&appid=APIKEY
```

(In the above code, replace `APIKEY` with your actual API key.)

Postman has a lot of other functionality you can use. We'll revisit Postman later in the course for some other activities.

Same request but in Paw instead of Postman

Although Postman is a popular REST client, you can also use others, such as Paw. The following image shows the same current weather API request made in Paw (for Mac):



URL Parameter	Value
☑ zip	95050,us
☑ appid	fd4698c940c6d1da602a70ac34f0...
☑ units	imperial
Add URL Parameter	Add Value

```
GET /data/2.5/weather?
zip=95050,us&appid=fd4698c940c6d1da602a70ac34f0b147&units=imper
ial HTTP/1.1
Host: api.openweathermap.org
Connection: close
User-Agent: Paw/3.1.5 (Macintosh; OS X/10.12.6) GCDHTTPRequest
```

Key or Index	Type	Value
▼ Root	≡	12 items
▼ coord	≡	2 items
lon	42	-121.96
lat	42	37.35
▼ weather	≡	1 item
▼ Index 0	≡	4 items
id	42	701
main	A	Mist
description	A	mist
icon	A	50d
base	A	stations
▼ main	≡	5 items
temp	42	66.38
pressure	42	1018
humidity	42	87
temp_min	42	53.6
temp_max	42	77
visibility	42	9656
▼ wind	≡	2 items
speed	42	10.29
deg	42	320
▼ clouds	≡	1 item

Like Postman, Paw also allows you to easily see the request headers, response headers, URL parameters, and other data. I like that Paw shows the response in an expandable/collapsible way. The expand/collapse feature can make it easier to explore the response. Note that Paw is specific to Mac only, and like most products for Mac users, costs money.

Enter several requests for the Aeris API into Postman

Now let's switch APIs a bit and see some weather information from the Aeris Weather API, which you explored a bit in Scenarios for using a weather API (p. 67). Constructing the endpoints for the Aeris Weather API is a bit more complicated since there are many different queries, filters, and other parameters you can use to configure the endpoint.

Here are a few pre-configured requests to configure for Aeris. You can paste the requests directly into the URL request box in Postman (after customizing the `CLIENTID` AND `CLIENTSECRET` values), and the parameters will auto-populate in the parameter fields.

As with the OpenWeather Map API, the Aeris API doesn't use a Header field to pass the API keys — the key and secret are passed directly in the request URL as part of the query string.

When you make the following requests, insert your own values for the `CLIENTID` and `CLIENTSECRET` (assuming you retrieved them in Get the authorization keys (p. 70)).

Get the weather forecast for your area using the observations endpoint ↗:

```
http://api.aerisapi.com/observations/Santa+Clara,CA?client_id=CLIENTID&clien
t_secret=CLIENTSECRET&limit=1
```

Get the weather from a city on the equator — Chimborazo, Ecuador using the same observations endpoint ↗:

```
http://api.aerisapi.com/observations/Chimborazo,Ecuador?client_id=CLIENTID&c
lient_secret=CLIENTSECRET&limit=1
```

Find out if all the country music in Knoxville, Tennessee is giving people migraines using the indices endpoint ↗:

```
http://api.aerisapi.com/indices/migraine/Knoxville,TN?client_id=CLIENTID&cli
ent_secret=CLIENTSECRET
```

You're thinking of moving to Arizona, but you want to find a place that's cool. Use the normals endpoint ↗:

```
http://api.aerisapi.com/normals/flagstaff,az?client_id=CLIENTID&client_secre
t=CLIENTSECRET&limit=5&filter=hassnow
```

With both the OpenWeatherMap and Aeris Weather API, you can also make these requests by simply going to the URL in your address bar (because the API keys are passed in the query string rather than the header). If so, use the JSON Formatter extension for Chrome ↗ to automatically format the JSON response in the browser view.

By looking at these two different weather APIs, you can see some differences in the way the information is called and returned. However, fundamentally both APIs have endpoints that you can configure with parameters. When you make requests with the endpoints, you get responses that contain information, often in JSON format. This is the core of how REST APIs work — you send a request and get a response.

Automatically import the Postman collections

Postman has a nifty import feature that will automatically pull in the same requests you've been entering. You can click the Run in Postman buttons below to automatically import these two collections into your own instance of Postman.

To view these buttons, go to the web page for this content at https://idratherbewriting.com/learnapidoc/docapis_postman.html (p. 71).

Clicking the Run in Postman buttons should automatically prompt you to import the collections into Postman. If it doesn't work, copy the import link address and, in Postman, click **Import** in the upper-left corner. Then click the **Import From Link** tab, paste in the address and click **Import**.

If you'd like to learn more about Postman, listen to this interview with the Postman founder. We recorded this as part of the Write the Docs podcast and focused on the documentation features within Postman. For more information on creating the Run in Postman buttons, see the Run in Postman button (p. 382) section in the Getting started tutorial (p. 379).

curl intro and installation

While Postman (p. 71) is convenient, it's hard to use it to represent how to make calls with it in your documentation. Additionally, different users probably use different GUI clients, or none at all (preferring the command line instead).

Instead of describing how to make REST calls using a GUI client like Postman, the most conventional method for documenting request syntax is to use curl.

About curl

curl is a command-line utility that lets you execute HTTP requests with different parameters and methods. Instead of going to web resources in a browser's address bar, you can use the command line to get these same resources, retrieved as text.

Sometimes curl is written as cURL. It stands for Client URL. "curl" is the more common convention for its spelling, but both refer to the same thing.

Installing curl

curl is usually available by default on Macs but requires some installation on Windows. Follow these instructions for installing curl:

Install curl on Mac

If you have a Mac, by default, curl is probably already installed. To check:

1. Open Terminal (press **Cmd + spacebar** to open Finder, and then type "Terminal").
2. In Terminal type `curl -V`. The response should look something like this:

```
curl 7.54.0 (x86_64-apple-darwin16.0) libcurl/7.54.0 SecureTransport
zlib/1.2.8
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps
pop3 pop3s rtsp smb smbs smtp smtps telnet tftp Features: AsynchDNS I
Pv6 Largefile GSS-API Kerberos SPNEGO NTLM NTLM_WB SSL libz UnixSocke
ts
```

If you don't see this, you need to download and install curl.

Install curl on Windows

Installing curl on Windows involves a few more steps. First, determine whether you have 32-bit or 64-bit Windows by right-clicking **Computer** and selecting **Properties.** Then follow the instructions in this Confused by Code page. Most likely, you'll want to choose the **With Administrator Privileges (free)** installer.

After you install curl, test your version of curl by doing the following:

1. Open a command prompt by clicking the **Start** button and typing **cmd**.
2. Type `curl -V`.

The response should be as follows:

```
curl 7.54.0 (x86_64-apple-darwin14.0) libcurl/7.37.1 SecureTransport zlib/
1.2.5
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 p
op3s rtsp smtp smtps telnet tftp
Features: AsynchDNS GSS-Negotiate IPv6 Largefile NTLM NTLM_WB SSL libz
```

Make a test API call

After you have curl installed, make a test API call:

```
curl -X GET "https://api.openweathermap.org/data/2.5/weather?zip=95050&appi
d=APIKEY&units=imperial"
```

(In the above code, replace `APIKEY` with your actual API key.)

You should get minified JSON response back like this:

```
{"coord":{"lon":-121.96,"lat":37.35},"weather":[{"id":701,"main":"Mist","des
cription":"mist","icon":"50d"}],"base":"stations","main":{"temp":66.92,"pres
sure":1017,"humidity":50,"temp_min":53.6,"temp_max":75.2},"visibility":1609
3,"wind":{"speed":10.29,"deg":300},"clouds":{"all":75},"dt":1522526400,"sy
s":{"type":1,"id":479,"message":0.0051,"country":"US","sunrise":152250440
4,"sunset":1522549829},"id":420006397,"name":"Santa Clara","cod":200}
```

In Windows, Ctrl+ V doesn't work; instead, you right-click and then select **Paste**.

Notes about using curl with Windows

If you're using Windows, note the following formatting requirements when using curl:

- Use double quotes in the Windows command line. (Windows doesn't support single quotes.)
- Don't use backslashes (`\`) to separate lines. (This is for readability only and doesn't affect the call on Macs.)
- By adding `-k` in the curl command, you can bypass curl's security certificate, which may or may not be necessary.

Make a curl call

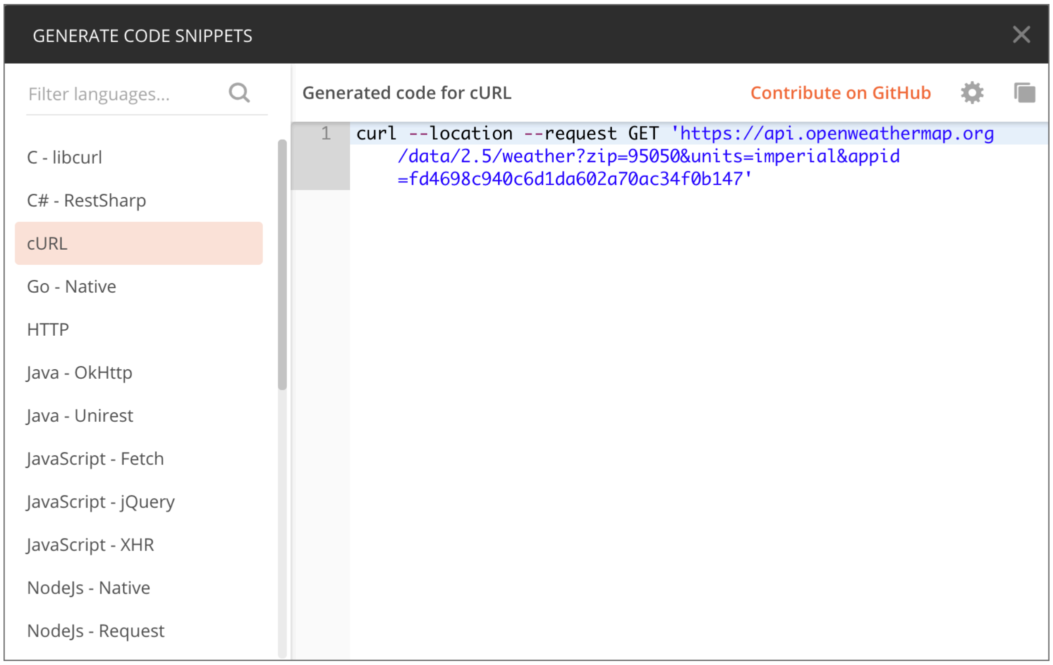
In this section, you'll use curl to make the same weather API requests you made previously with Postman. If you haven't installed curl, see curl intro and installation (p. 78) first.

Activity: Make the OpenWeatherAPI request using curl

This activity assumes you have curl installed. curl is available on Mac and some Windows 10 versions by default. If you're on an older Windows machine that doesn't have curl, see installing curl here for details. (Most likely, choose "With Administrator Privileges (free)" 64-bit version.) Close and restart your Command Prompt after installing curl.

To make a request with curl:

1. Assuming you completed the exercises in the Postman tutorial (p. 71), go back into Postman.
2. On any request you've configured, below the Save button in Postman, click the **Code** link. (If you don't see the link, scroll up.)
3. In the Generate Code Snippets dialog box, select **cURL** from the drop-down list, and then click **Copy to Clipboard**.



Generated code for cURL

```
curl --location --request GET 'https://api.openweathermap.org
    /data/2.5/weather?zip=95050&units=imperial&appid
    =fd4698c940c6d1da602a70ac34f0b147'
```

curl request in Postman

The Postman code for the OpenWeatherMap weather request in curl looks as follows:

```
curl --location --request GET 'https://api.openweathermap.org/data/
2.5/weather?zip=95050&units=imperial&appid=APIKEY'
```

(In the above code, replace `APIKEY` with your actual API key.)

The `--location` parameter will prompt curl to submit a new request if the URL is a redirect. The `--request` parameter specifies the operation for the request.

(Note that previously, Postman would include its own header information, designated with `-H`. If you see these parameters, delete them since they cause issues when submitted outside of Postman.)

In general, the code snippets can be copied and pasted directly into your terminal on a Mac. However, for Windows, you must change the single quotation marks to double quotation marks.

Also, on Windows, if your curl has any backslashes, (`\`) remove them and put all content onto the same line. (Backslashes are just added for readability). You can make these adjustments in a text editor before pasting the curl command into the Command Prompt.

4. Go to your Terminal (Mac) or Command Prompt (Windows).

   You can open your Terminal / Command Prompt by doing the following:

   - If you're on Windows, go to **Start** and search for **cmd** to open up the Command Prompt. Paste in the curl request and then press **Enter**. (If you can't paste it in, look for paste options on the right-click menu.)
   - If you're on a Mac, open Terminal by pressing **Cmd + spacebar** and typing **Terminal**. (Or go to **Applications > Utilities > Terminal**). (You could also use iTerm.) Paste in the curl request and then press **Enter**.

   The response from the OpenWeatherMap weather request should look as follows:

   ```
   {"coord":{"lon":-121.95,"lat":37.35},"weather":[{"id":802,"main":"Clouds","description":"scattered clouds","icon":"03d"}],"base":"stations","main":{"temp":68.34,"pressure":1014,"humidity":73,"temp_min":63,"temp_max":72},"visibility":16093,"wind":{"speed":3.36},"clouds":{"all":40},"dt":1566664878,"sys":{"type":1,"id":5122,"message":0.0106,"country":"US","sunrise":1566653501,"sunset":1566701346},"timezone":-25200,"id":0,"name":"Santa Clara","cod":200}
   ```

   This response is minified. You can un-minify it by going to a site such as JSON pretty print, or if you have Python installed, you can add `| python -m json.tool` at the end of your cURL request to un-minify the JSON in the response (see this Stack Overflow thread for details).

Note about single and double quotes with Windows curl requests

If you're using Windows to submit a lot of curl requests, and the curl requests require you to submit JSON in the request body (p. 137), you might run into issues with single versus double quotes. The problem is that request body content is often formatted in JSON, which requires double quotes.

Since you can't use double quotes inside of other double quotes, you'll run into issues in submitting curl requests in these scenarios.

Here's the workaround. If you have to submit body content in JSON, you can store the content in a JSON file. Then you reference the file with an `@` symbol, like this:

```
curl -H "Content-Type: application/json" -H "Authorization: 123" -X POST -d @mypostbody.json http://endpointurl.com/example
```

Here curl will look in the existing directory for the `mypostbody.json` file. (You can also reference the complete path to the JSON file on your machine.)

Understand curl more

Almost every API shows how to interact with the API using curl. So before moving on, let’s pause a bit and learn more about curl.

Why curl?

One of the advantages of REST APIs is that you can use almost any programming language to call the endpoint. The endpoint is simply a resource located on a web server at a specific path.

Each programming language has a different way of making web calls. Rather than exhausting your energies trying to show how to make web calls in Java, Python, C++, JavaScript, Ruby, and so on, you can just show the call using curl.

curl provides a generic, language-agnostic way to demonstrate HTTP requests and responses. Users can see the format of the request, including any headers and other parameters. Your users can translate this into the specific format for the language they’re using.

Try using curl to GET a web page

As mentioned earlier, one reason REST APIs are so familiar is that REST follows the same model as the web (see What is a REST API? (p. 40)). When you type an `http` address into a browser address bar, you’re telling the browser to make an HTTP request to a resource on a server. The server returns a response, and your browser converts the response to a more visual display. But you can also see the raw code.

To see an example of how curl retrieves a web resource, open a terminal and type the following:

```
curl http://example.com
```

Curl will retrieve the HTML code for the site example.com. The browser’s job is to make that code visually readable. curl shows you what you’re actually retrieving.

Requests and responses include headers too

When you type an address into a website, you see only the body of the response. But actually, there’s more going on behind the scenes. When you make the request, you’re sending a request header that contains information about the request. The response also contains a response header.

1. To see the response header in a curl request, include `-i` in the curl request:

   ```
   curl http://example.com -i
   ```

   The header will be included *above* the body in the response:

```
~/projects $ curl http://example.com -I
HTTP/1.1 200 OK
Content-Encoding: gzip
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Tue, 04 Dec 2018 04:35:43 GMT
Etag: "1541025663+gzip"
Expires: Tue, 11 Dec 2018 04:35:43 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (sjc/4F91)
X-Cache: HIT
Content-Length: 606
```

2. To limit the response to just the header, use `-I`:

   ```
   curl http://example.com -I
   ```

   The header contains the metadata about the response. All of this information is transferred to the browser when you make a request to a URL in your browser (for example, when you surf to a web page online), but the browser doesn't show you this information. You can see the header information using the Chrome Developer Tools console by looking on the Network tab.

3. Now let's specify the method. The GET method (read) is implied by default when no other method is specified, but we'll make it explicit here with the `-X` parameter:

   ```
   curl -X GET http://example.com -I
   ```

   When you go to a website, you submit the request using the GET HTTP method. There are other HTTP methods you can use when interacting with REST APIs. Here are the common methods used when working with REST endpoints:

HTTP Method	Description
POST	Create a resource
GET	Read a resource
PUT	Update a resource
DELETE	Delete a resource

   GET is used by default with curl requests. If you use curl to make HTTP requests other than GET, you need to specify the HTTP method.

Unpacking the weather API curl request

Let's look more closely at the request you submitted for the weather in the previous topic (Make a curl call (p. 80)):

```
curl -X GET -H "Cache-Control: no-cache" -H "Postman-Token: 930d08d6-7b2a-6ea2-0725-27324755c684" "https://api.openweathermap.org/data/2.5/weather?zip=95050&appid=APIKEY&units=imperial"
```

(In the above code, replace `APIKEY` with your actual API key.)

curl has shorthand names for the various options that you include with your request.

Here's what the commands mean:

- `-X GET`. The `-X` signifies the method used for the request. Common options are `GET`, `POST`, `DELETE`, `PUT`. (You might also see `--get` used instead. Most curl commands have a couple of different representations. `-X GET` can also be written as `--get`.)
- `-H`. Submits a custom header. Include an additional `-H` for each header key-value pair you're submitting.

Query strings and parameters

The zip code (`zip`), app ID (`appid`), and units (`units`) parameters were passed to the endpoint using "query strings." The `?` appended to the URL indicates the start of the query string. The query string parameters are the parameters that appear *after* the `?`:

```
?zip=95050&appid=APIKEY&units=imperial
```

(In the above code, replace `APIKEY` with your actual API key.)

After the query string, each parameter is separated from other parameters by the ampersand `&` symbol. The order of the query string parameters doesn't matter. The order only matters if the parameters are *on the left of the query string* (and thus part of the URL itself). Any configurable parts of the endpoint that appear before the query string are called path parameters (p. 136) (we'll dive into these later).

Common curl commands related to REST

curl has a lot of possible commands, but the following are the most common when working with REST APIs.

curl command	Description	Example
`-i` or `--include`	Includes the response headers in the response.	`curl -i http://www.example.com`
`-d` or `--data`	Includes data to post to the URL. The data needs to be url encoded. Data can also be passed in the request body.	`curl -d "data-to-post" http://www.example.com`

curl command	Description	Example
`-H` or `--header`	Submits the request header to the resource. Headers are common with REST API requests because the authorization is usually included in the header.	`curl -H "key:12345" http://www.example.com`
`-X POST`	Specifies the HTTP method to use with the request (in this example, `POST`). If you use `-d` in the request, curl automatically specifies a POST method. With GET requests, including the HTTP method is optional, because `GET` is the default method used.	`curl -X POST -d "resource-to-update" http://www.example.com`
`@filename`	Loads content from a file.	`curl -X POST -d @mypet.json http://www.example.com`

See the curl documentation for a comprehensive list of curl commands you can use.

Example curl command

Here's an example curl request that combines some of these commands:

```
curl -i -H "Accept: application/json" -X POST -d "{status:MIA}" http://perso
nsreport.com/status/person123
```

The request could also be formatted with line breaks to make it more readable:

```
curl -i \
     -H "Accept: application/json" \
     -X POST \
     -d "{status:MIA}" \
     http://personsreport.com/status/person123 \
```

(Line breaks are problematic on Windows, so I don't recommend formatting curl requests like this.)

The `Accept` header tells the server that the only format we will accept in the response is JSON.

Quiz yourself

Quiz yourself to see how much you remember. What do the following parameters mean?

- `-i`
- `-H`
- `-X POST`
- `-d`

When you use curl, the Terminal and iTerm on the Mac provide a much easier experience than using the command prompt in Windows. If you're going to get serious about API documentation, but you're still on a PC, consider switching. There are a lot of utilities that you install through a terminal that *just work* on a Mac. Also, if you're in Silicon Valley, using a PC instead of a Mac might make you look old-fashioned (see Why do most startups purchase MacBooks for their employees?). Alternatively, you can run Linux on Windows, and you'll get the same terminal experience (Bash shell). See this example tutorial on how to install Bash on Windows."

Activity: Use methods with curl

Our sample weather API (p. 65) doesn't allow you to use anything but a GET method, so for this exercise, to use other methods with curl, we'll use the petstore API from Swagger. However, we won't actually use the Swagger UI (which is something we'll explore later (p. 181)). For now, we just need an API with which we can use to create, update, and delete content.

In this example, using the Petstore API, you'll create a new pet, update the pet, get the pet's ID, delete the pet, and then try to get the deleted pet.

Create a new pet

To create a pet, you have to pass a JSON message in the request body. Rather than trying to encode the JSON and pass it in the URL, you'll store the JSON in a file and reference the file.

A lot of APIs require you to post requests containing JSON messages in the body. Request bodies are often how you configure a service. The list of JSON key-value pairs that the API accepts is called the "Model" in the Swagger UI display.

1. Insert the following into a text file. This information will be passed in the `-d` parameter of the curl request:

```
{
  "id": 123,
  "category": {
    "id": 123,
    "name": "test"
  },
  "name": "fluffy",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

2. Change the first `id` value to another integer (a whole number in this case). Also, change the pet's name of `fluffy` to something else.

   Use a unique ID and name that others aren't likely to also use. Also, don't begin your ID with the number 0.

3. Save the file in a directory that you can conveniently access from your terminal, such as your user directory (on a Mac, `Users/YOURUSERNAME` — replace `YOURUSERNAME` with your actual user name on your computer).
4. In your terminal, browse to the directory where you saved the **mypet.json** file. (Usually, the default directory is `Users/YOURUSERNAME` — hence the previous step.)

If you've never browsed directories using the command line, here's how you do it:

- On a Mac, find your present working directory by typing `pwd`. Then move up a level by typing change directory: `cd ../`. Move down a level by typing `cd pets`, where `pets` is the name of the directory you want to move into. Type `ls` to list the contents of the directory.
- On Windows, look at the prompt path to see your current directory. Then move up a level by typing `cd ../`. Move down a level by typing `cd pets`, where `pets` is the name of the directory you want to move into. Type `dir` to list the contents of the current directory.

5. After your terminal or command prompt is in the same directory as your JSON file, create the new pet with the following curl request:

```
curl -X POST --header "Content-Type: application/json" --header "Accept: application/json" -d @mypet.json "https://petstore.swagger.io/v2/pet"
```

The `Content-Type` indicates the type of content submitted in the request body. The `Accept` indicates the type of content we will accept in the response.

The response should look something like this:

```
{"id":51231236,"category":{"id":4,"name":"testexecution"},"name":"fluffernutter","photoUrls":["string"],"tags":[{"id":0,"name":"string"}],"status":"available"}
```

In the response, check to see that your pet's name was returned.

Update your pet

Guess what, your pet hates its name! Change your pet's name to something more formal using the update pet method.

1. In the **mypet.json** file, change the pet's name.
2. Use the `PUT` method instead of `POST` to update the pet's name (keep the same curl content otherwise):

```
curl -X PUT --header "Content-Type: application/json" --header "Accept: application/json" -d @mypet.json "https://petstore.swagger.io/v2/pet"
```

Get your pet's name by ID

Find your pet's name by passing the ID into the `/pet/{petID}` endpoint:

1. In your **mypet.json** file, copy the first `id` value.
2. Use this curl command to get information about that pet ID, replacing `51231236` with your pet ID.

```
curl -X GET --header "Accept: application/json" "https://petstore.swagger.io/v2/pet/51231236"
```

The response contains your pet's name and other information:

```
{"id":51231236,"category":{"id":4,"name":"test"},"name":"mr. fluffernutter","photoUrls":["string"],"tags":[{"id":0,"name":"string"}],"status":"available"}
```

You can format the JSON by pasting it into a JSON formatting tool:

```
{
  "id": 51231236,
  "category": {
    "id": 4,
    "name": "test"
  },
  "name": "mr. fluffernutter",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

Delete your pet

Unfortunately, your pet has died. It's time to delete your pet from the pet registry.

1. Use the DELETE method to remove your pet. Replace `5123123` with your pet ID:

```
curl -X DELETE --header "Accept: application/json" "https://petstore.swagger.io/v2/pet/5123123"
```

2. Now check to make sure your pet is removed. Use a GET request to look for your pet with that ID:

```
curl -X GET --header "Accept: application/json" "https://petstore.swagger.io/v2/pet/5123123"
```

You should see this error message:

```
{"code":1,"type":"error","message":"Pet not found"}
```

This example allowed you to see how you can work with curl to create, read, update, and delete resources. These four operations are referred to as CRUD and are common to almost every programming language.

Although Postman is probably easier to use, curl lends itself to power-level usage. Quality assurance teams often construct advanced test scenarios that iterate through a lot of curl requests.

Understanding idempotent methods

One concept important to understand with HTTP methods is “idempotency.” Roy Fielding defines idempotency as follows:

> A request method is considered “idempotent” if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. Of the request methods defined by this specification, PUT, DELETE, and safe request methods are idempotent” (RFC 7231, 4.2.2 ↗.

In other words, with idempotent methods, you can run them multiple times without multiplying the results. Idempotent methods include GET, PUT, and DELETE, while POST is not (see 8.1.3 ↗ for a more detailed list).

Todd Fredrich explains idempotency by comparing it to a pregnant cow ↗. Let’s say you bring over a bull to get a cow pregnant. Even if the bull and cow mate multiple times, the result will be just one pregnancy, not a pregnancy for each mating session.

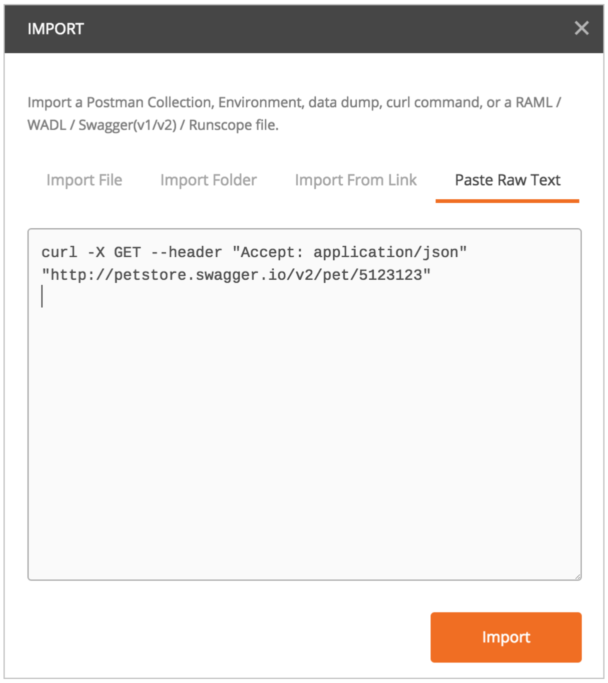
Import curl into Postman

You can import curl commands into Postman by doing the following:

1. Open a new tab in Postman and click the **Import** button in the upper-left corner.
2. Select **Paste Raw Text** and insert your curl command:

   ```
   curl -X GET --header "Accept: application/json" "https://petstore.swagger.io/v2/pet/5123123"
   ```

   Make sure you don’t have any extra spaces at the beginning.


IMPORT
Import a Postman Collection, Environment, data dump, curl command, or a RAML / WADL / Swagger(v1/v2) / Runscope file.
Import File
Import Folder
Import From Link
Paste Raw Text
Import

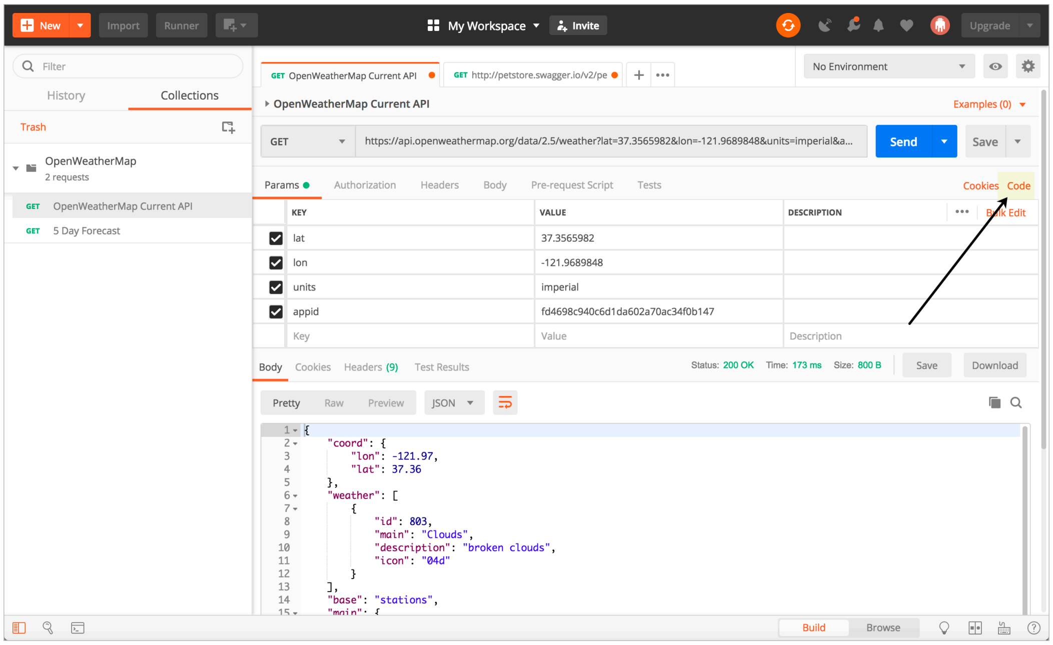
```
curl -X GET --header "Accept: application/json"
"http://petstore.swagger.io/v2/pet/5123123"
```

3. Click **Import**.
4. Close the dialog box.
5. Click **Send**. (If you deleted your pet, you will see the same "Pet not found" error message as before.)

Export Postman to curl

You can also export Postman to curl by doing the following:

1. If desired, select one of your OpenWeatherMap API requests in Postman.
2. Click the **Code** button (it's right below Save).



GET https://api.openweathermap.org/data/2.5/weather?lat=37.3565982&lon=-121.9689848&units=imperial&a...

	KEY	VALUE	DESCRIPTION
☑	lat	37.3565982	
☑	lon	-121.9689848	
☑	units	imperial	
☑	appid	fd4698c940c6d1da602a70ac34f0b147	
	Key	Value	Description

Status: 200 OK Time: 173 ms Size: 800 B

```
{
    "coord": {
        "lon": -121.97,
        "lat": 37.36
    },
    "weather": [
        {
            "id": 803,
            "main": "Clouds",
            "description": "broken clouds",
            "icon": "04d"
        }
    ],
    "base": "stations",
    "main": {
```

3. Select **curl** from the drop-down menu.
4. Copy the code snippet.

```
curl -X GET \
  'https://api.openweathermap.org/data/2.5/weather?lat=37.3565982&lon=-121.9689848&units=imperial&appid=APIKEY'
```

In place of `APIKEY` you should see your actual API key.

5. Remove the backslashes and line breaks. If you're on Windows, change the single quotes to double quotes.
6. Insert the curl command into your terminal and observe the result.

```
curl -X GET "https://api.openweathermap.org/data/2.5/weather?lat=37.3565982&lon=-121.9689848&units=imperial&appid=APIKEY"
```

Through Postman's Import and Code functionality, you can easily switch between Postman and curl.

Analyze the JSON response

JSON is the most common format for responses from REST APIs. Let's look at the JSON response for the OpenWeatherMap weather endpoint in more depth, distinguishing between arrays and objects in JSON.

JSON response from OpenWeatherMap weather endpoint

JSON stands for JavaScript Object Notation. It's the most common way REST APIs return information. Although some APIs return information in both JSON and XML, if you're trying to parse through the response and render it on a web page, JSON fits much better into the existing JavaScript + HTML + CSS technology that powers most web pages. With JavaScript, you can easily parse through the JSON and integrate it into your web content.

The unminified response from the OpenWeatherMap weather endpoint looks like this:

```
{
  "coord": {
    "lon": -121.96,
    "lat": 37.35
  },
  "weather": [
    {
      "id": 801,
      "main": "Clouds",
      "description": "few clouds",
      "icon": "02d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 70.14,
    "pressure": 1012,
    "humidity": 33,
    "temp_min": 62.6,
    "temp_max": 75.2
  },
  "visibility": 16093,
  "wind": {
    "speed": 14.99,
    "deg": 330
  },
  "clouds": {
    "all": 20
  },
  "dt": 1522619760,
  "sys": {
    "type": 1,
    "id": 479,
    "message": 0.0058,
    "country": "US",
    "sunrise": 1522590707,
    "sunset": 1522636288
  },
  "id": 420006397,
  "name": "Santa Clara",
  "cod": 200
}
```

We'll analyze the information structures within JSON responses in the following sections.

JSON objects are key-value pairs

JSON has two types of basic structures: objects and arrays. An object is a collection of key-value pairs, surrounded by curly braces:

```
{
  "key1": "value1",
  "key2": "value2"
}
```

The key-value pairs are each put into double quotation marks when both are strings. If the value is an integer (a whole number) or Boolean (true or false value), omit the quotation marks around the value. Each key-value pair is separated from the next by a comma.

JSON arrays are lists of items

An array is a list of items, surrounded by brackets:

```
["first", "second", "third"]
```

The list of items can contain strings, numbers, booleans, arrays, or other objects. With integers or booleans, you don't use quotation marks.

Integers:

```
[1, 2, 3]
```

Booleans:

```
[true, false, true]
```

Including objects in arrays, and arrays in objects

JSON can mix up objects and arrays inside each other. You can have an array of objects:

```
[
  object,
  object,
  object
]
```

Here's an example with values:

```
[
   {
      "name":"Tom",
      "age":39
   },
   {
      "name":"Shannon",
      "age":37
   }
]
```

And objects can contain arrays in the value part of the key-value pair:

```
{
"children": ["Avery","Callie","lucy","Molly"],
"hobbies": ["swimming","biking","drawing","horseplaying"]
}
```

Just remember, objects are enclosed by curly braces `{ }` and contain key-value pairs. Sometimes those values are arrays. Arrays are lists and are enclosed by square brackets `[ ]`. It’s common for arrays to contain lists of objects, and for objects to contain arrays.

It’s important to understand the difference between objects and arrays because it determines how you access and display the information. Later exercises with dot notation will require you to understand this difference.

Examine the weather response

Look at the response from the `weather` endpoint of the OpenWeatherMap weather API. Where are the objects? Where are the arrays? Which objects are nested? Which values are booleans versus strings?

More information

For more information on understanding the structure of JSON, see json.com.

Inspect the JSON from the response payload

Seeing the response from curl or Postman is cool, but how do you make use of the JSON data? With most API documentation, you don't need to show how to make use of JSON data. You assume that developers will use their front-end development skills to parse through the data and display it appropriately in their apps. However, to better understand how developers will access the data, we'll go through a brief tutorial to display the REST response on a web page.

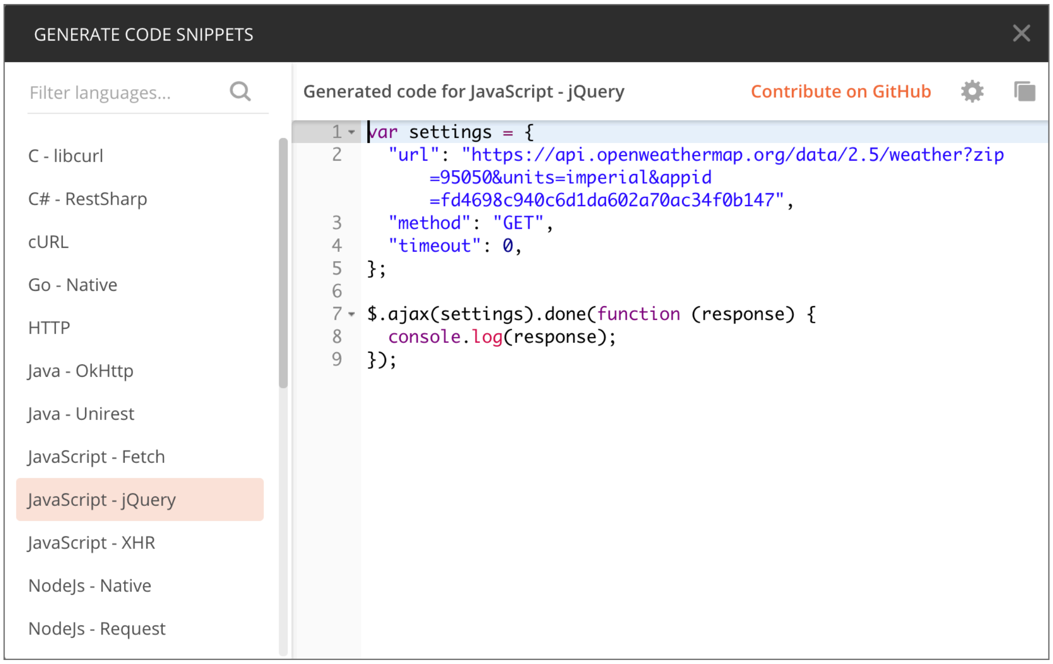
Activity: Make an API request on a web page

For this activity, you'll use JavaScript to display the API response on a web page. Specifically, you'll use some auto-generated jQuery code from Postman to create the AJAX request. You'll get the wind speed from the response and print it to the page.

1. In an editor such as Sublime Text, create a new HTML file called `weather.html` and insert the following boilerplate code:

```
<html>
   <meta charset="UTF-8">
   <head>
      <title>Sample page</title>
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.1
1.1/jquery.min.js"></script>
      <script>
      POSTMAN CODE GOES HERE
      </script>
   </head>
   <body>
      <h1>Sample Page</h1>
      wind speed: <span id="windSpeed"></span>
   </body>
</html>
```

2. Assuming you completed the exercises in the Postman tutorial (p. 71) to configure a request, go back into Postman.
3. In Postman, click the **Code** link (below the Save button) and go to **JavaScript - jQuery**:



```
var settings = {
  "url": "https://api.openweathermap.org/data/2.5/weather?zip
      =95050&units=imperial&appid
      =fd4698c940c6d1da602a70ac34f0b147",
  "method": "GET",
  "timeout": 0,
};

$.ajax(settings).done(function (response) {
  console.log(response);
});
```

Copying JavaScript code from Postman

4. Copy the Postman code above and insert it into the `POSTMAN CODE GOES HERE` place in your `weather.html` file.
5. Directly below `console.log(response);`, add these two lines:

```
var content = response.wind.speed;
$("#windSpeed").append(content);
```

6. Your final code should look as follows:

```
<html>
   <meta charset="UTF-8">
   <head>
      <title>Sample page</title>
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
      <script>
      var settings = {
        "url": "https://api.openweathermap.org/data/2.5/weather?zip=95050&units=imperial&appid=APIKEY",
        "method": "GET",
        "timeout": 0,
      };

      $.ajax(settings).done(function (response) {
        console.log(response);
        var content = response.wind.speed;
        $("#windSpeed").append(content);
      });

      </script>
   </head>
   <body>
      <h1>Sample Page</h1>
      wind speed: <span id="windSpeed"></span>
   </body>
</html>
```

(In the above code, replace `APIKEY` with your actual API key.)

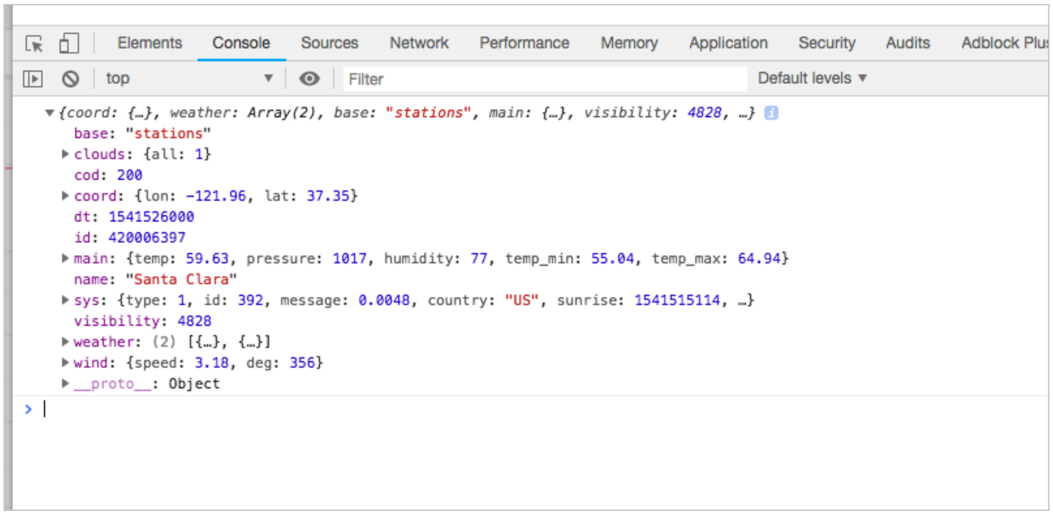
What is this code doing? In a nutshell, when `ajax` (a jQuery function) retrieves the response from the API, it assigns the response to `response`. A variable called `content` is created and set it equal to `response.wind.speed` (dot notation (p. 109) is used to access this value). jQuery's `append` method inserts `content` after an element called `#windSpeed` on the page. (I realize this is an extremely abbreviated explanation, but explaining JavaScript is beyond the scope of this course. In general, you can learn more by reading about the jQuery.ajax() function.)

7. Start **Chrome** and open the JavaScript Console.

   To open the JavaScript Console, on Chrome on a Mac, go to **View > Developer > Javascript Console**; on Windows, click the **menu** button (vertical ellipses) and go to **More tools > Developer tools**. Then click the **Console** tab.

8. In Chrome, press **Cmd+O** (Mac) or **Ctrl + O** (Windows) and select your `weather.html` file.

   The weather response should be logged to the JavaScript Console (due to the `console.log(response)` code in the request). If you expand the object returned to the console, it will look as follows:



```
▼{coord: {…}, weather: Array(2), base: "stations", main: {…}, visibility: 4828, …}
   base: "stations"
  ▶clouds: {all: 1}
   cod: 200
  ▶coord: {lon: -121.96, lat: 37.35}
   dt: 1541526000
   id: 420006397
  ▶main: {temp: 59.63, pressure: 1017, humidity: 77, temp_min: 55.04, temp_max: 64.94}
   name: "Santa Clara"
  ▶sys: {type: 1, id: 392, message: 0.0048, country: "US", sunrise: 1541515114, …}
   visibility: 4828
  ▶weather: (2) [{…}, {…}]
  ▶wind: {speed: 3.18, deg: 356}
  ▶__proto__: Object
>
```

You can view the file here: weather-plain.html.

The AJAX method from jQuery

In this section, I'll explain a bit more about the `ajax` function you used earlier. This information probably isn't essential for documenting REST APIs, but it's good to understand. In the code, here's the `ajax` script:

```
<script>
   var settings = {
     "async": true,
     "crossDomain": true,
     "url": "https://api.openweathermap.org/data/2.5/weather?zip=95050&appid=APIKEY&units=imperial",
     "method": "GET"
   }

   $.ajax(settings).done(function (response) {
     console.log(response);
   });
</script>
```

(In the above code, replace `APIKEY` with your actual API key.)

If you're working with JavaScript and APIs, the `ajax` method from jQuery can be helpful with code samples. This `ajax` method takes one argument: `settings`.

```
$.ajax(settings)
```

The `settings` argument is an object that contains a variety of key-value pairs.

```
var settings = {
}
```

Each of the allowed key-value pairs is defined in jQuery's ajax documentation.

Some important values are the `url`, which is the URI or endpoint you are submitting the request to. Another value is `headers`, which allows you to include custom headers in the request.

Look at the code sample you created. The `settings` variable is passed in as the argument to the `ajax` method. jQuery makes the request to the HTTP URL asynchronously, which means it won't hang up your computer while you wait for the response. You can continue using your application while the request executes.

You get the response by calling the method `done`.

```
$.ajax(settings).done(function (response) {
})
```

In the earlier code sample, `done` contains an anonymous function (a function without a name) that executes when `done` is called. The response object from the `ajax` call is assigned to the `done` method's argument, which in this case is `response`. (You can name the argument whatever you want.)

You can then access the values from the response object using object notation. In this example, the response is just logged to the console.

If you're new to JavaScript, this is likely a bit fuzzy right now. If so, don't worry — code becomes clearer the more you use it.

Notice how difficult it is to explain code? This is one of the challenges of developer documentation. Fortunately, you wouldn't need to explain much from standard programming languages like JavaScript. But you might need to explain how to work with your API in different languages. I cover this topic in more depth in Code samples and tutorials (p. 464).

Logging responses to the console

The line of code that logged the response to the console was simply this:

```
console.log(response);
```

Logging responses to the console can be a useful way to test whether an API response is working (it's also helpful for debugging or troubleshooting your code). The console collapses each object inside an expandable section. You can inspect the payload in the console to see if contains the values you expect (without printing values to the page).

Inspect the payload

Inspect the payload by expanding each of the sections returned in the JSON console object. Based on the information here, what's the forecast for today?

I realize the page is blank and unexciting. In the next section, Access and print a specific JSON value (p. 103), we'll pull out some values and print them to the page.

Access and print a specific JSON value

This tutorial continues from the previous topic, Inspect the JSON from the response payload (p. 98). In the sample page where you logged the `weather` response to the JS Console, the REST response information didn't appear on the page. It only appeared in the JS Console. You need to use dot notation and JavaScript to access the JSON values you want. In this tutorial, you'll use JavaScript to print some of the response to the page.

Note that this section will use a little bit of JavaScript. Depending on your role, you might not use this code much in your documentation, but it's important to know anyway.

Getting a specific property from a JSON response object

JSON wouldn't be very useful if you always had to print out the entire response. Instead, you select the exact property you want and pull that out through dot notation. The dot (`.`) after `response` (the name of the JSON payload, as defined arbitrarily in the jQuery AJAX function) is how you access the values you want from the JSON object.

As an example, this is the full response from the request made previously (p. 94):

```
{
  "coord": {
    "lon": -121.95,
    "lat": 37.35
  },
  "weather": [
    {
      "id": 802,
      "main": "Clouds",
      "description": "scattered clouds",
      "icon": "03d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 68.34,
    "pressure": 1014,
    "humidity": 73,
    "temp_min": 63,
    "temp_max": 72
  },
  "visibility": 16093,
  "wind": {
    "speed": 3.36
  },
  "clouds": {
    "all": 40
  },
  "dt": 1566664878,
  "sys": {
    "type": 1,
    "id": 5122,
    "message": 0.0106,
    "country": "US",
    "sunrise": 1566653501,
    "sunset": 1566701346
  },
  "timezone": -25200,
  "id": 0,
  "name": "Santa Clara",
  "cod": 200
}
```

In our scenario (p. 65) (creating a biking app), we want to pull out the wind speed part of the JSON response. Here's the dot notation you use:

```
response.wind.speed
```

To pull out the wind speed element from the JSON response and print it to the JavaScript Console, add this to your code sample (which you created in the previous tutorial (p. 94)), right below the `console.log(response)` line:

```
console.log("wind speed: " + response.wind.speed);
```

Your code should look like this:

```
$.ajax(settings).done(function (response) {
  console.log(response);
  console.log("wind speed: " + response.wind.speed);
});
```

Refresh your Chrome browser and see the information that appears in the console:

```
wind speed: 13.87
```

Printing a JSON value to the page

Let's say you wanted to print part of the JSON (the wind speed data) to the page, not just the console. (By "print," I mean make the value appear on the page, not send it to a printer.) Printing the value involves a little bit of JavaScript (or jQuery to make it easier).

I'm assuming you're starting with the same code from the previous tutorial (p. 98). That code looks like this:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquer
y.min.js"></script>
      <title>Sample Page</title>
      <script>
         var settings = {
           "async": true,
           "crossDomain": true,
           "url": "https://api.openweathermap.org/data/2.5/weather?zip=9505
0&appid=APIKEY&units=imperial",
           "method": "GET"
         }

         $.ajax(settings).done(function (response) {
           console.log(response);
         });
      </script>
   </head>
   <body>
      <h1>Sample Page</h1>
   </body>
</html>
```

(In the above code, replace `APIKEY` with your actual API key.)

To print a specific property from the response to the page,

1. Add the following inside the `ajax` function:

   ```
   $.ajax(settings).done(function (response) {
   console.log(response);

   var content = response.wind.speed;
   $("#windSpeed").append(content);

   });
   ```

   Your code should look as follows:

   ```
   <!DOCTYPE html>
   <html>
      <head>
         <meta charset="utf-8">
         <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.1
   1.1/jquery.min.js"></script>
         <title>Sample Page</title>
         <script>
            var settings = {
              "async": true,
              "crossDomain": true,
              "url": "https://api.openweathermap.org/data/2.5/weather?zi
   p=95050&appid=APIKEY&units=imperial",
              "method": "GET"
            }

            $.ajax(settings).done(function (response) {
              console.log(response);

              var content = response.wind.speed;
              $("#windSpeed").append(content);

            });
         </script>
      </head>
      <body>
         <h1>Sample Page</h1>
         <div id="windSpeed">Wind speed: </div>
      </body>
   </html>
   ```

   (In the above code, replace `APIKEY` with your actual API key.)

2. Refresh the page and you will see the wind speed printed to the page. Here's an example with both wind speed and weather conditions.

Here's what we changed:

Inside the tags of the AJAX `done` method, we pulled out the value we wanted into a variable, like this:

```
var content = response.wind.speed;
```

Then we added a named element to the body of the page, like this:

```
<div id="windSpeed">Wind speed: </div>
```

We used the jQuery `append` method to append the `content` variable to the element with the `windSpeed` ID on the page:

```
$("#windSpeed").append(content);
```

This code says to find the element with the ID `windSpeed` and add the `content` variable after it.

Get the value from an array

In the previous section, you retrieved a value from a JSON object. Now let's get a value from an array. Let's get the `main` property from the `weather` array in the response. Here's what the JSON array looks like:

```
{
  "weather": [
    {
      "id": 801,
      "main": "Clouds",
      "description": "few clouds",
      "icon": "02d"
    }
  ]
]
}
```

Remember that brackets signify an array. Inside the `weather` array is an unnamed object. To get the `main` element from this array, you would use the following dot notation:

```
response.weather[0].main
```

Then you would follow the same pattern as before to print it to the page. While objects allow you to get a specific property, arrays require you to select the position in the list that you want.

Here's the code from the sample page:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
<title>Sample Page</title>

<script>
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://api.openweathermap.org/data/2.5/weather?zip=95050&appid=APIKEY&units=imperial",
  "method": "GET"
}

$.ajax(settings).done(function (response) {
  console.log(response);

  var content = response.wind.speed;
  $("#windSpeed").append(content);

  var currentWeather = response.weather[0].main;
  $("#currentWeather").append(currentWeather);

});
</script>
</head>
<body>
<h1>Sample Page</h1>

<div id="windSpeed">Wind speed: </div>
<div id="currentWeather">Current weather conditions: </div>

</body>
</html>
```

(In the above code, replace `APIKEY` with your actual API key.)

More exercises

If you'd like to follow some more exercises that involve calling REST APIs, accessing specific values, and printing the values to the page, see the following topics in the Glossary and resources (p. 892) section:

- Get event information using the Eventbrite API (p. 901)
- Flickr example: Retrieve a Flickr gallery (p. 908)
- Get wind speed using the Aeris Weather API (p. 918)

Dive into dot notation

In the previous topic, Access and print a specific JSON value (p. 103), you accessed and printed a specific JSON value to the page. Let's dive into dot notation a little more since understanding how to access the right JSON value you want is necessary to make use of the response.

Dot notation

You use a dot after the object name to access its properties. For example, suppose you have an object called `data`:

```
"data": {
"name": "Tom"
}
```

To access `Tom`, you would use `data.name`.

Note the different levels of nesting so you can trace back the appropriate objects and access the information you want. You access each level down through the object name followed by a dot.

Use square brackets to access the values in an array

To access a value in an array, you use square brackets followed by the position number. For example, suppose you have the following array:

```
"data" : {
  "items": ["ball", "bat", "glove"]
}
```

To access glove, you would use `data.items[2]`.

`glove` is the third item in the array. You can't access an item directly in an array by the item's name — only by its position. Usually, programmers loop through an array and pull out values that match.

With most programming languages, you usually start counting at `0`, not `1`.

Exercise with dot notation

In this activity, you'll practice accessing different values through dot notation.

1. Create a new file in your text editor and insert the following into it:

```
<!DOCTYPE html>
 <html>
   <head>
      <script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
      <meta charset="utf-8">
      <title>JSON dot notation practice</title>
      <script>
         $( document ).ready(function() {

            var john = {
             "hair": "brown",
             "eyes": "green",
             "shoes": {
                 "brand": "nike",
                 "type": "basketball"
             },
             "favcolors": [
                 "azure",
                 "goldenrod"
             ],
             "children": [
                 {
                     "child1": "Sarah",
                     "age": 2
                 },
                 {
                     "child2": "Jimmy",
                     "age": 5
                 }
             ]
         }

         var sarahjson = john.children[0].child1;
         var greenjson = john.children[0].child1;
         var nikejson = john.children[0].child1;
         var goldenrodjson = john.children[0].child1;
         var jimmyjson = john.children[0].child1;

         $("#sarah").append(sarahjson);
         $("#green").append(greenjson);
         $("#nike").append(nikejson);
         $("#goldenrod").append(goldenrodjson);
         $("#jimmy").append(jimmyjson);
         });
      </script>
   </head>
   <body>
      <div id="sarah">Sarah: </div>
      <div id="green">green: </div>
      <div id="nike">nike: </div>
      <div id="goldenrod">goldenrod: </div>
      <div id="jimmy">Jimmy: </div>
    </body>
  </html>
```

Here we have a JSON object defined as a variable named `john`. (Usually, APIs retrieve the response through a URL request, but for practice here, we're just defining the object locally.)

If you view the page in your browser, you'll see the page says "Sarah" for each item because we're accessing this value: `john.children[0].child1` for each item.

```
var sarahjson = john.children[0].child1;
var greenjson = john.children[0].child1;
var nikejson = john.children[0].child1;
var goldenrodjson = john.children[0].child1;
var jimmyjson = john.children[0].child1;
```

2. Change `john.children[0].child1` to display the correct values for each item. For example, the word `green` should appear at the ID tag called `green`.

You can view the correct page here: https://idratherbewriting.com/assets/files/dot-notation-practice.html. This page also shows the answers printed.

Showing wind conditions on the page

At the beginning of the section on Using an API like a developer (p. 65), I showed an example of embedding the wind speed and other details on a website. Now let's revisit this code example and see how it's put together.

Copy the following code into a basic HTML file:

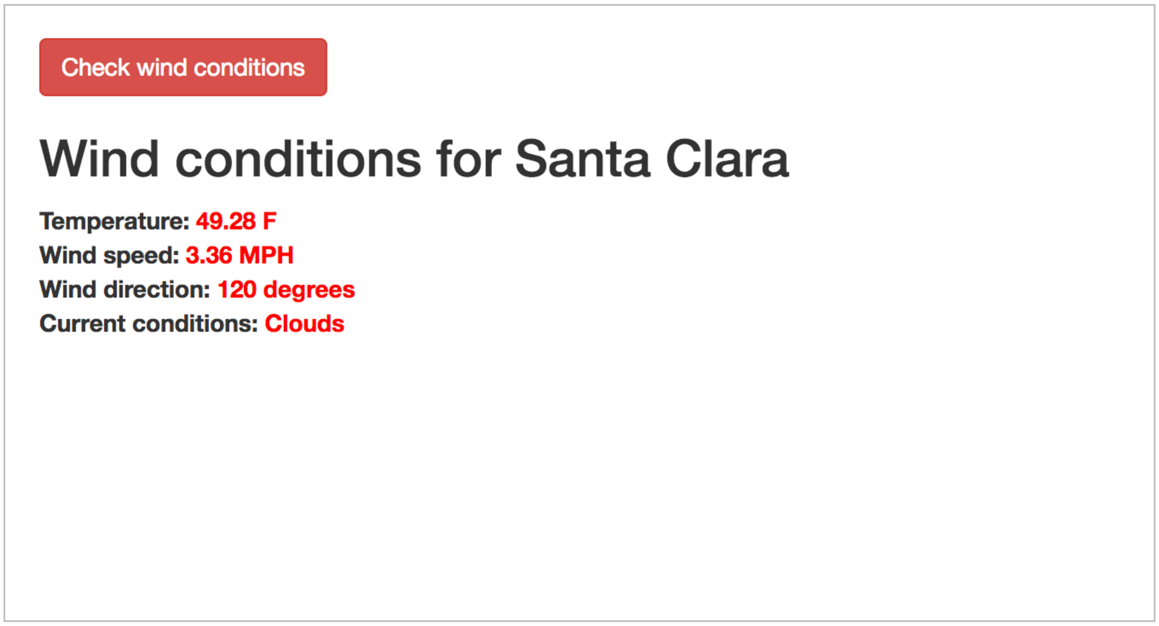
```
<!DOCTYPE html>
 <html>
   <head>
      <script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
      <meta charset="utf-8">
      <link rel="stylesheet"  href='https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css' rel='stylesheet' type='text/css'>
      <title>OpenWeatherMap Integration</title>
      <style>
         #wind_direction, #wind_speed, #wind_speed_unit, #wind_degree_unit, #weather_conditions, #main_temp_unit, #main_temp {color: red; font-weight: bold;}
         body {margin:20px;}
      </style>
   </head>
   <body>
      <script>
         function checkWind() {
           var settings = {
             "async": true,
             "crossDomain": true,
             "dataType": "json",
             "url": "https://api.openweathermap.org/data/2.5/weather?zip=95050,us&appid=APIKEY&units=imperial",
             "method": "GET"
         }

           $.ajax(settings)

           .done(function (response) {
             console.log(response);

         $("#wind_speed").append (response.wind.speed);
         $("#wind_direction").append (response.wind.deg);
         $("#main_temp").append (response.main.temp);
         $("#weather_conditions").append (response.weather[0].main);
         $("#wind_speed_unit").append (" MPH");
         $("#wind_degree_unit").append (" degrees");
         $("#main_temp_unit").append (" F");
         });
         }
      </script>
      <button type="button" onclick="checkWind()" class="btn btn-danger weatherbutton">Check wind conditions</button>
      <h2>Wind conditions for Santa Clara</h2>
      <span><b>Temperature: </b></span><span id="main_temp"></span><span id="main_temp_unit"></span><br/>
      <span><b>Wind speed: </b></span><span id="wind_speed"></span> <span id="wind_speed_unit"></span><br/>
      <span><b>Wind direction: </b></span><span id="wind_direction"></span><span id="wind_degree_unit"></span><br/>
      <span><b>Current conditions: </b></span><span id="weather_conditions"></span>
   </body>
</html>
```

(In the above code, replace `APIKEY` with your actual API key.)

A few things are different here, but it's essentially the same code as you created in the Access and print a specific JSON value (p. 103). Here's what's different:

- Rather than running the `ajax` method on page load, the `ajax` method is wrapped inside a function called `checkWind`. When the web page's button is clicked, the `onclick` method fires the `checkWind()` function.
- When the `checkWind` function runs, the values for temperature, wind speed, wind direction, and current conditions are written to several ID tags on the page.

When you load the page and click the button, the following should appear:



Wind conditions for Santa Clara

Temperature: 49.28 F
Wind speed: 3.36 MPH
Wind direction: 120 degrees
Current conditions: Clouds

You can view the file idratherbewriting.com/assets/files/wind-openweathermap.html.

Next section

As you've progressed through the exercises for using an API like a developer, you've gained a high-level understanding of how REST APIs work, what information developers need, how they might use an API, how they make requests, evaluate responses, and other details.

With this background, it's time to switch gears and put on your technical writing hat. In the next section, Documenting endpoints (p. 114), you'll assume the task of documenting a new endpoint (p. 115) that was added to a weather API. You'll learn the essential sections in endpoint reference documentation, the terminology to use, and formatting conventions for API reference information.

Chapter 3: Documenting API endpoints

Reference documentation for API endpoints consists of five general sections: resource descriptions, endpoints and methods, parameters, sample requests, and sample responses and schemas. To document the reference endpoints of an API, provide detailed information for each of these sections.

Chapter sections:

A new endpoint to document 115
API reference tutorial overview 118
Step 1: Resource description 120
Step 2: Endpoints and methods 126
Step 3: Parameters 132
Step 4: Request example 141
Step 5: Response example and schema 154
Putting it all together 169
Activity: What's wrong with this API reference topic 173
Activity: Evaluate API reference docs for core elements 177

A new endpoint to document

Until this point, you've been acting as a developer (p. 64) with the task of integrating the weather data into your site. The point was to help you understand the type of information developers need and how they use APIs.

Now let's shift perspectives. Now suppose you're a technical writer working with the OpenWeatherMap team. The team is asking you to document a new endpoint. What do you cover, and how do you approach it?

You have a new endpoint to document

The project manager calls you over and says the team has a new endpoint for you to document for the next release. (Sometimes teams will also refer to each endpoint as an "API" as well.)

"Here's the wiki page that contains info about the new API," the manager says. The information is scattered and random on the wiki page.

Most technical writers don't start from scratch with documentation projects. Engineers usually dump essential information onto an internal wiki page (or they communicate the info during meetings). However, the information on the wiki page will likely be incomplete and unnecessarily technical in places (like describing the database schema or high-level architectural workflows). The info might also include internal-only information (for example, test logins, access protocols, or code names) or have sections that are out-of-date.

Ultimately, the information will be oriented towards other engineers on the same knowledge level as the team's engineers. Your job as a technical writer will be to take this information and turn it into complete, accurate, usable information that communicates with your audience.

Wiki page with information about the new endpoint

It's now your task to sort through the information on this mock wiki page and create documentation from it. You can read through the mock wiki page below to get a sense of the information. In the upcoming topics, we will proceed section by section through an API reference topic.

Here's the mock internal wiki page:

The wiki page: "Surf Report API"

The new endpoint is `/surfreport/{beachId}`. This endpoint is for surfers who want to check things like tide and wave conditions to determine whether they should head out to the beach to surf. `{beachId}` is retrieved from a list of beaches on our site.

Optional parameters:

- Number of days: Max is 7. Default is 3. Optional.
- Units: imperial or metric. With imperial, you get feet and knots. With metric, you get centimeters and kilometers per hour. Optional.
- Time: time of the day corresponding to time zone of the beach you're inquiring about. Format is unix time, aka epoch. Unix time is the milliseconds since 1970. Time zone is GMT or UTC. Optional.

If you include the hour, then you only get back the surf condition for the hour you specified. Otherwise, you get back 3 days, with conditions listed out by hour for each day.

The response will include the surf height, the wind, temp, the tide, and overall recommendation.

Sample endpoint with parameters:

```
https://api.openweathermap.org/com/surfreport/123?&days=2&units=metrics&hour=1400
```

The response contains these elements:

surfreport:

- surfheight (units: feet)
- wind (units: kts)
- tide (units: feet)
- water temperature (units: F degrees)
- recommendation - string ("Go surfing!", "Surfing conditions okay, not great", "Not today -- try some other activity.")

The recommendation is based on an algorithm that takes optimal surfing conditions, scores them in a rubric, and includes one of three responses.

Sample format:

```
{
    "surfreport": [
        {
            "beach": "Santa Cruz",
            "monday": {
                "1pm": {
                    "tide": 5,
                    "wind": 15,
                    "watertemp": 60,
                    "surfheight": 5,
                    "recommendation": "Go surfing!"
                },
                "2pm": {
                    "tide": -1,
                    "wind": 1,
                    "watertemp": 50,
                    "surfheight": 3,
                    "recommendation": "Surfing conditions are okay, not great"
                }
                ...

            }
        }
    ]
}
```

Negative numbers in the tide represent incoming tide.

The report won't include any details about riptide conditions.

Although users can enter beach names, there are only certain beaches included in the report. Users can look to see which beaches are available from our website at `https://example.com/surfreport/beaches_available` (not a real URL). The beach names must be url encoded when passed in the endpoint as query strings.

To switch from feet to metrics, users can add a query string of `&units=metrics`. Default is `&units=imperial`.

Here's an example ↗ of how developers might integrate this information. This site shows the height of the surf coupled with a cam.

If the query is malformed, you get error code 400 and an indication of the error.

You can see that the information here is unstructured and hard to scan. By structuring the API reference information into five standard sections, the information will take more shape and be more readable.

Next steps

Let's jump into the API reference tutorial overview (p. 118) for an overview of the five steps we'll cover in creating the API reference topic for this new endpoint.

API reference tutorial overview

In this API reference tutorial, we'll work on creating five common sections in REST API reference documentation: resource description, endpoints and methods, parameters, request example, and response example and schema. To provide some context (and to continue with our sample documentation scenario), we'll structure the information from the new endpoint to document (p. 115) into these five sections.

Five common sections in REST API docs

Almost all API reference topics include these five sections:

1. Resource description (p. 120)

> "Resources" refers to the information returned by an API.

2. Endpoints and methods (p. 126)

> The endpoints indicate how you access the resource, while the method indicates the allowed interactions (such as GET, POST, or DELETE) with the resource.

3. Parameters (p. 132)

> Parameters are options you can pass with the endpoint (such as specifying the response format or the amount returned) to influence the response.

4. Request example (p. 141)

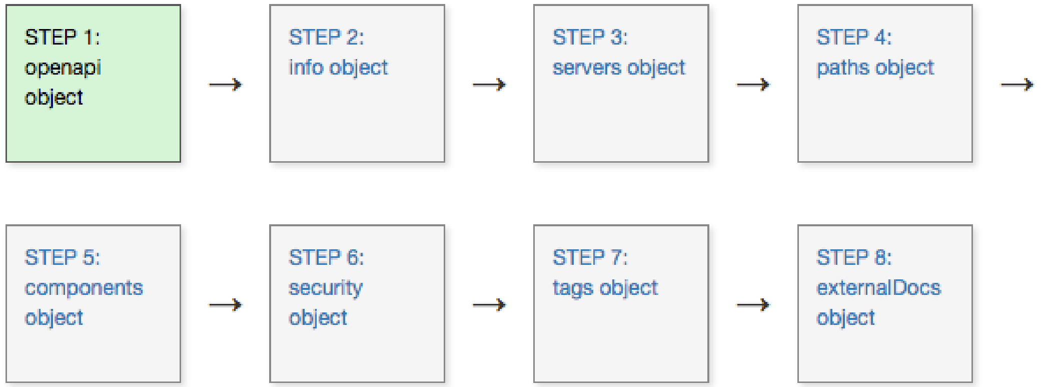
> The request example includes a sample request using the endpoint, showing some parameters configured.

5. Response example and schema (p. 154)

> The response example shows a sample response from the request example; the response schema defines all possible elements in the response.

Tutorial workflow map

The tutorial here includes a workflow map to help guide and orient you each step of the way.



After the tutorial

When we're finished, the end result will look like a real API help topic (see the finished result in Putting it all together (p. 169)). In the associated activities, you'll have an opportunity to edit or create an API reference topic (p. 177) with your own open-source API project (p. 686).

Although there are automated ways to publish API docs, we're focusing on content rather than tools in this section. In the next section, OpenAPI spec and generated reference docs (p. 179), we'll look at how to describe these same reference components using the OpenAPI specification. In the Publishing your API documentation section (p. 530), we'll look at ways to publish the information.

Next steps

Now that you have an idea of the tutorial, let's get started with the first section: Step 1: Resource description (p. 120).

Step 1: Resource description (API reference tutorial)

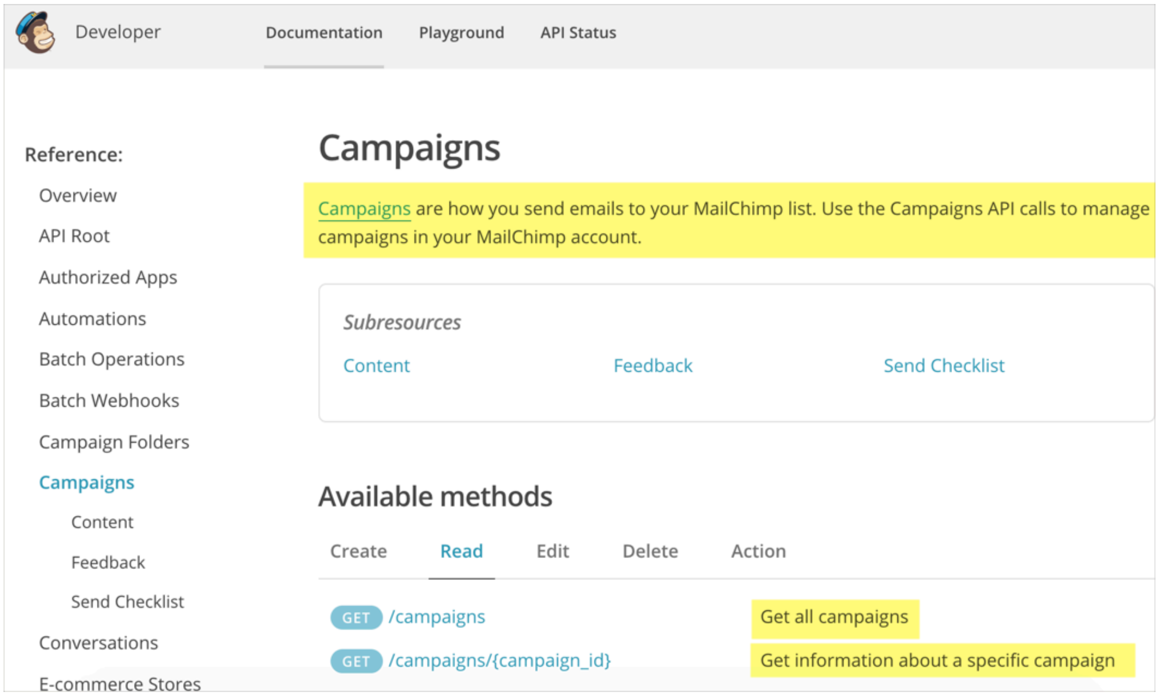
STEP 1: Resource description →
STEP 2: Resource URLs and methods →
STEP 3: Parameters →
STEP 4: Request example →
STEP 5: Response example and schema

“Resources” refers to the information returned by an API. Most APIs have various categories of information, or various resources, that can be returned.

The resource description is brief (1-3 sentences) and usually starts with a verb. Resources usually have various endpoints to access the resource and multiple methods for each endpoint. On the same page, you usually have a general resource described along with a number of endpoints for accessing the resource, also described.

Examples of resource descriptions

Here’s an example of a resource description from the Mailchimp API’s Campaigns resource:



Campaigns

Campaigns are how you send emails to your MailChimp list. Use the Campaigns API calls to manage campaigns in your MailChimp account.

Available methods

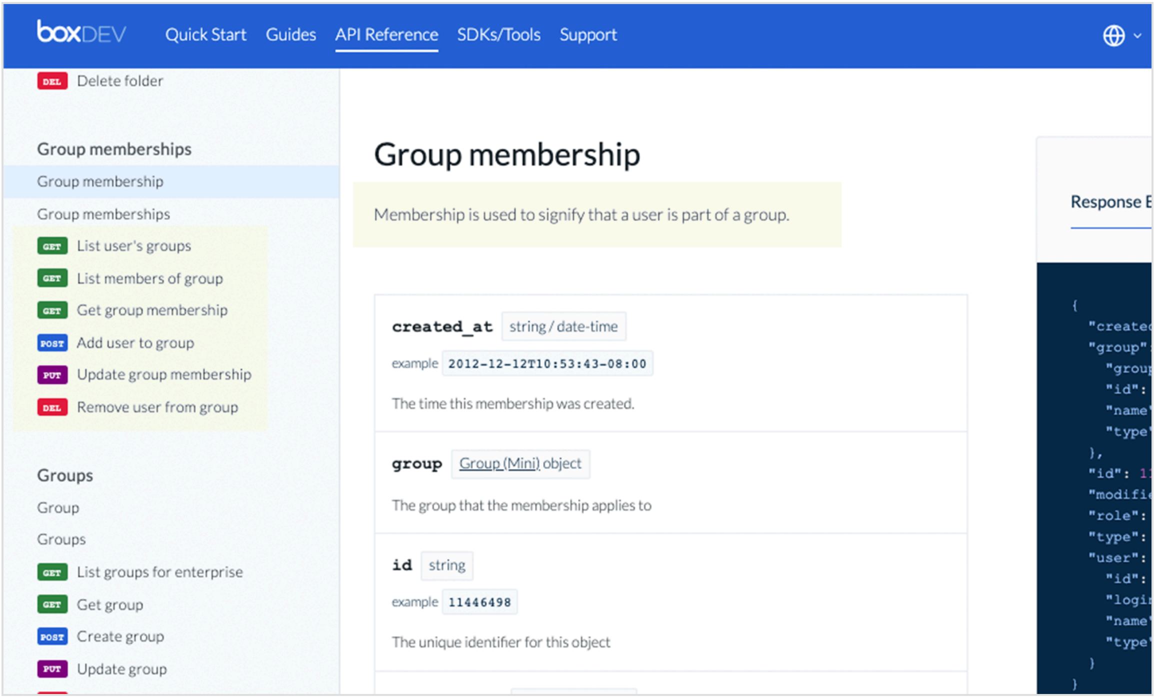
GET /campaigns Get all campaigns

GET /campaigns/{campaign_id} Get information about a specific campaign

Typically, an API will have a number of endpoints grouped under the same resource. In this case, you describe both the general resource and the individual endpoints. For example, the Campaigns resource has various endpoints that are also described:

- POST `/campaigns`
- GET `/campaigns`
- GET `/campaigns/{campaign_id}`
- PATCH `/campaigns/{campaign_id}`
- DELETE `/campaigns/{campaign_id}`
- POST `/campaigns/{campaign_id}/actions/cancel-send`
- POST `/campaigns/{campaign_id}/actions/pause`
- POST `/campaigns/{campaign_id}/actions/replicate`
- POST `/campaigns/{campaign_id}/actions/resume`
- POST `/campaigns/{campaign_id}/actions/schedule`
- POST `/campaigns/{campaign_id}/actions/send`
- POST `/campaigns/{campaign_id}/actions/test`
- POST `/campaigns/{campaign_id}/actions/unschedule`

Here’s a resource description for the Membership resource in the Box API:



Group membership

Membership is used to signify that a user is part of a group.

created_at string / date-time

example `2012-12-12T10:53:43-08:00`

The time this membership was created.

group Group (Mini) object

The group that the membership applies to

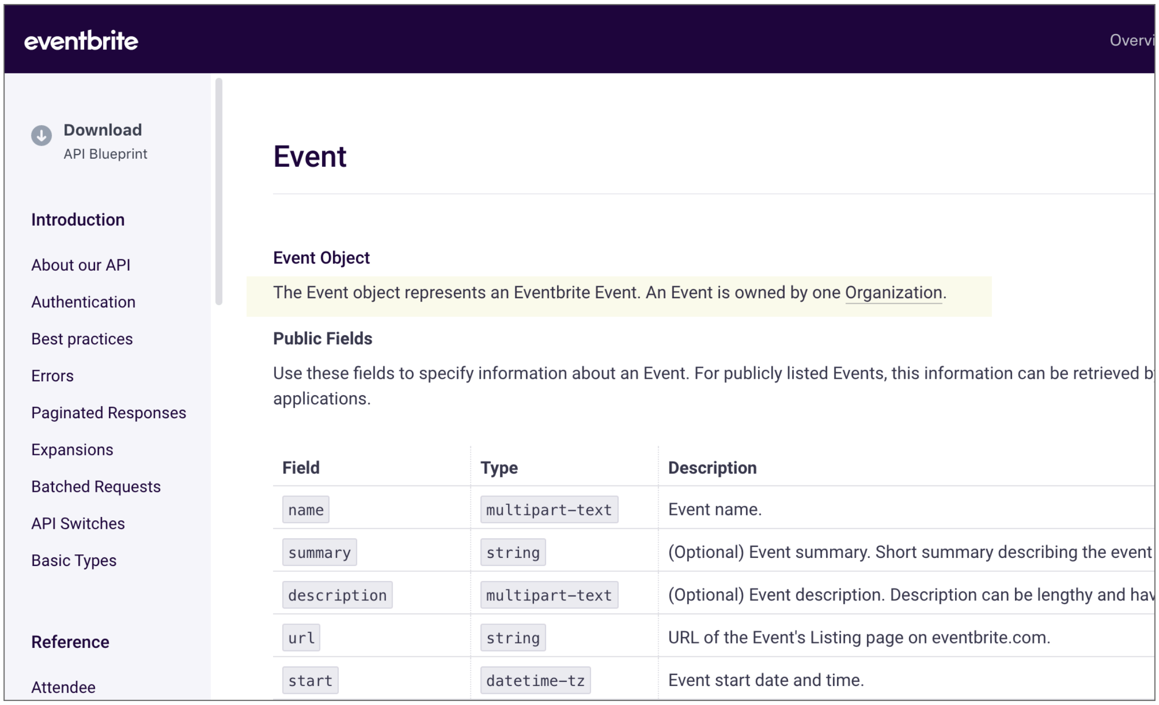
id string

example `11446498`

The unique identifier for this object

For the Membership resource (or “object,” as they call it), there are 7 different endpoints or methods you can call. The Box API describes the Membership resource and each of the endpoints that lets you access the resource.

Sometimes the general resource isn’t described; instead, it just groups the endpoints. The bulk of the description appears in each endpoint. For example, in the Eventbrite API, here’s the Events resource:



Event

Event Object

The Event object represents an Eventbrite Event. An Event is owned by one Organization.

Public Fields

Use these fields to specify information about an Event. For publicly listed Events, this information can be retrieved b
applications.

Field	Type	Description
`name`	`multipart-text`	Event name.
`summary`	`string`	(Optional) Event summary. Short summary describing the event
`description`	`multipart-text`	(Optional) Event description. Description can be lengthy and hav
`url`	`string`	URL of the Event's Listing page on eventbrite.com.
`start`	`datetime-tz`	Event start date and time.

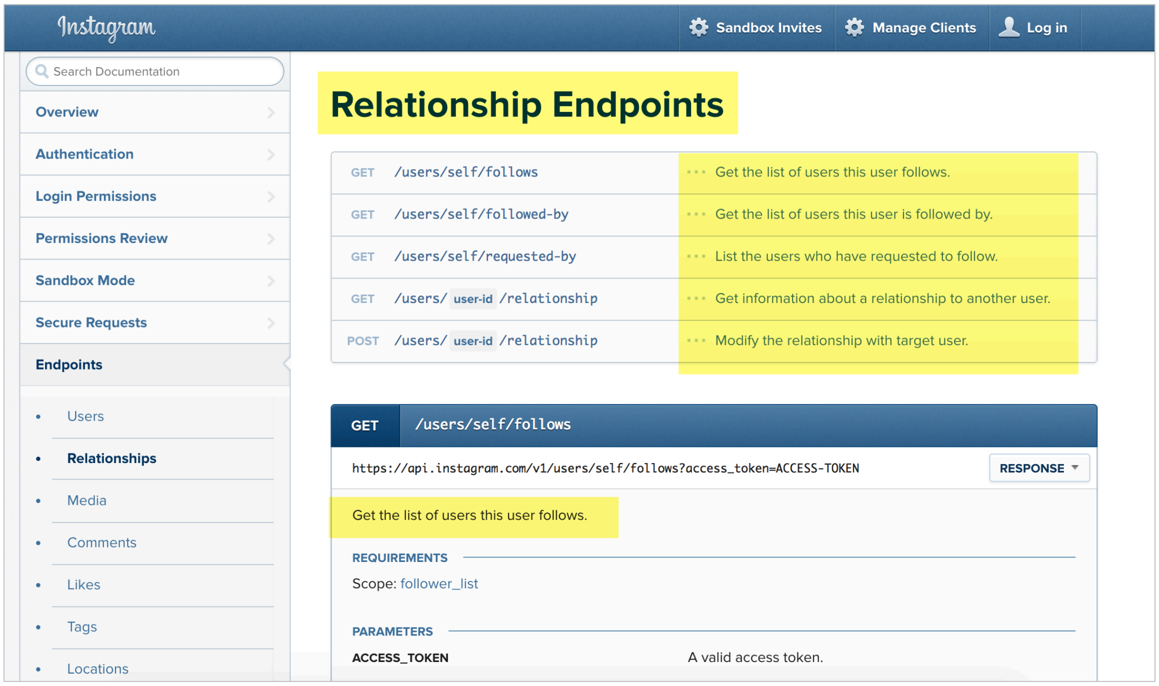
Although the Events resource isn't described here, descriptions are added for each of the Events endpoints. The Events resource contains all of these endpoints:

- `/events/search/`
- `/events/`
- `/events/:id/`
- `/events/:id/`
- `/events/:id/publish/`
- `/events/:id/cancel/`
- `/events/:id/`
- `/events/:id/display_settings/`
- `/events/:id/display_settings/`
- `/events/:id/ticket_classes/`
- `/events/:id/ticket_classes/:ticket_class_id/`
- `/events/:id/canned_questions/`
- `/events/:id/questions/`
- `/events/:id/attendees/`
- `/events/:id/discounts`

And so on.

When developers create APIs, they have a design question to consider: Use many variants of endpoints (as with Eventbrite's API), or provide lots of parameters to configure the same endpoint. Often there's a balance between the two. The trend seems to be toward providing separate endpoints rather than supplying a host of potentially confusing parameters within the same endpoint. On the other hand, GraphQL APIs (which I don't cover in this course) provide one endpoint with different ways to query the information the endpoint contains. See GraphQL And REST Differences Explained With Burgers for a good explanation about GraphQL versus REST.

As another example, the previous version of the Instagram API depicted a Relationships resource as follows:



Relationship Endpoints

GET	`/users/self/follows`	Get the list of users this user follows.
GET	`/users/self/followed-by`	Get the list of users this user is followed by.
GET	`/users/self/requested-by`	List the users who have requested to follow.
GET	`/users/` user-id `/relationship`	Get information about a relationship to another user.
POST	`/users/` user-id `/relationship`	Modify the relationship with target user.

GET `/users/self/follows`

`https://api.instagram.com/v1/users/self/follows?access_token=ACCESS-TOKEN`

Get the list of users this user follows.

REQUIREMENTS

Scope: follower_list

PARAMETERS

ACCESS_TOKEN	A valid access token.

The Relationships resource isn't described but rather acts as a container for relationship endpoints. Descriptions are added for each of the resources grouped within the Relationships resource:

- GET `/users/self/followsGet`
- GET `/users/self/followed-byGet`
- GET `/users/self/requested-byList`
- GET `/users/user-id/relationshipGet`
- POST `/users/user-id/relationshipModify`

(Instagram has since shifted to a Graph API.)

For another example of an API with resources and endpoints, check out the Trello API.

The description of the resource is likely something you'll re-use in different places — product overviews, tutorials, code samples, quick references, etc. As a result, put a lot of effort into crafting it. Consider storing the description in a re-usable snippet in your authoring tool so that you can list it without resorting to copy/paste methods in your quick start guide (p. 412).

Terminology for describing the resource

The exact terminology for referring to *resources* varies. The "things" that you access using a URL can be referred to in a variety of ways, but "resource" is the most common term because you access them through a URL, or uniform *resource* locator. Other than "resources," you might see terms such as *API calls*, *endpoints*, *API methods*, *calls*, *objects*, *services*, and *requests*. Some docs get around the situation by not calling them anything explicitly except "Reference."

Despite the variety with terminology, generally an API has various "resources" that you access through "endpoints." The endpoints give you access to the resource. (But terminology isn't standard, so expect variety.)

For more on how API terminology varies, see The difference between resources, endpoints, objects, and items in the bunq API documentation.

Recognize the difference between reference docs versus user guides

Resource descriptions (as well as endpoint descriptions) are typically short, usually 1-3 sentences. What if you have a lot more detail to add? In these situations, keep in mind the difference between reference documentation and user guides/tutorials:

- **Reference documentation**: Concise, bare-bones information that developers can quickly reference.
- **User guides/tutorials**: More elaborate detail about how to use the API, including step-by-step instructions, code samples, concepts, and procedures. I go into much more detail about this content in Documenting concepts sections (p. 358).

Although the description in an API reference topic provides a 1-3 sentence summary of the information that the resource contains, you might expand on this with much greater detail in the user guide. (You could link the reference description to the places in the guide where you provide more detail.)

Resource description for the surfreport endpoint

Let's review the surf report wiki page (p. 115) (which contains the information about the resource) and try to describe the resource in 1-3 sentences. Here's my approach:

Surfreport

Contains information about surfing conditions, including the surf height, water temperature, wind, and tide. Also provides an overall recommendation about whether to go surfing.

Next steps

Now it’s time to list out the endpoints and methods (p. 126) for the resource.

Step 2: Endpoints and methods (API reference tutorial)

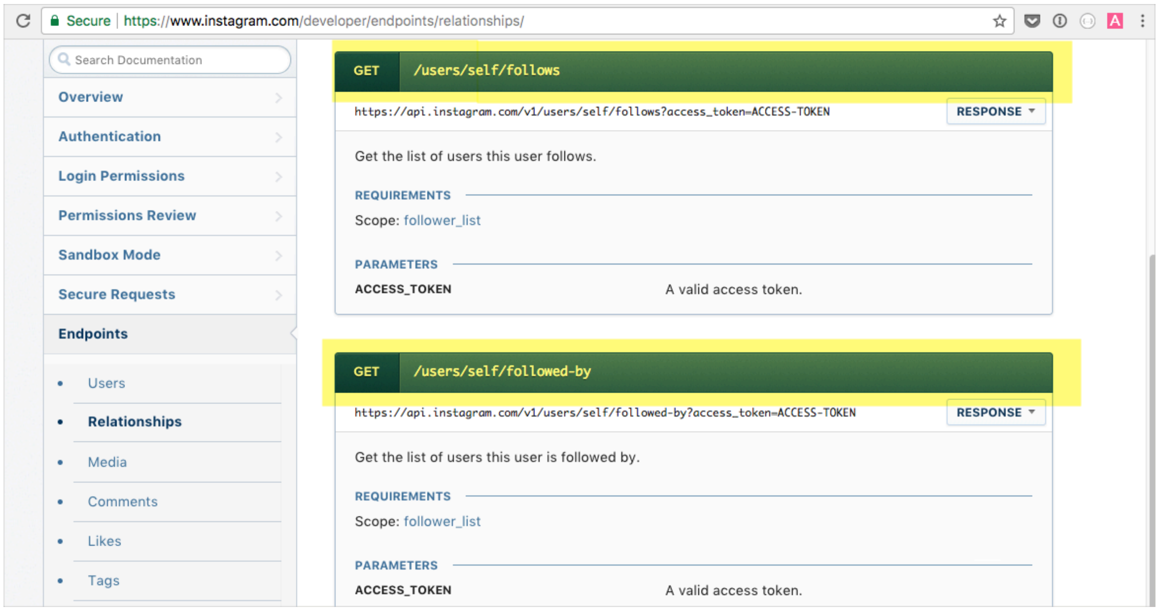
STEP 1: Resource description →
STEP 2: Resource URLs and methods →
STEP 3: Parameters →
STEP 4: Request example →
STEP 5: Response example and schema

The endpoints indicate how you access the resource, while the method indicates the allowed interactions (such as GET, POST, or DELETE) with the resource.

The same resource usually has a variety of related endpoints, each with different paths and methods but returning different information about the same resource. Endpoints usually have brief descriptions similar to the overall resource description but shorter. Also, the endpoint shows the end path of a resource URL only, not the base path common to all endpoints.

Examples of endpoints

Here’s an example of the endpoints for the Relationships resource in the Instagram API:



GET /users/self/follows

https://api.instagram.com/v1/users/self/follows?access_token=ACCESS-TOKEN

Get the list of users this user follows.

REQUIREMENTS

Scope: follower_list

PARAMETERS

ACCESS_TOKEN A valid access token.

GET /users/self/followed-by

https://api.instagram.com/v1/users/self/followed-by?access_token=ACCESS-TOKEN

Get the list of users this user is followed by.

REQUIREMENTS

Scope: follower_list

PARAMETERS

ACCESS_TOKEN A valid access token.

The endpoint is usually set off in a stylized way that gives it more visual attention. Much of the documentation is built around the endpoint, so it might make sense to give each endpoint more visual weight in your documentation.

The endpoint is arguably the most important aspect of API documentation because this is what developers will implement to make their requests.

Represent path parameters with curly braces

If you have path parameters (p. 136) in your endpoint, represent them through curly braces. For example, here's an example from Mailchimp's API:

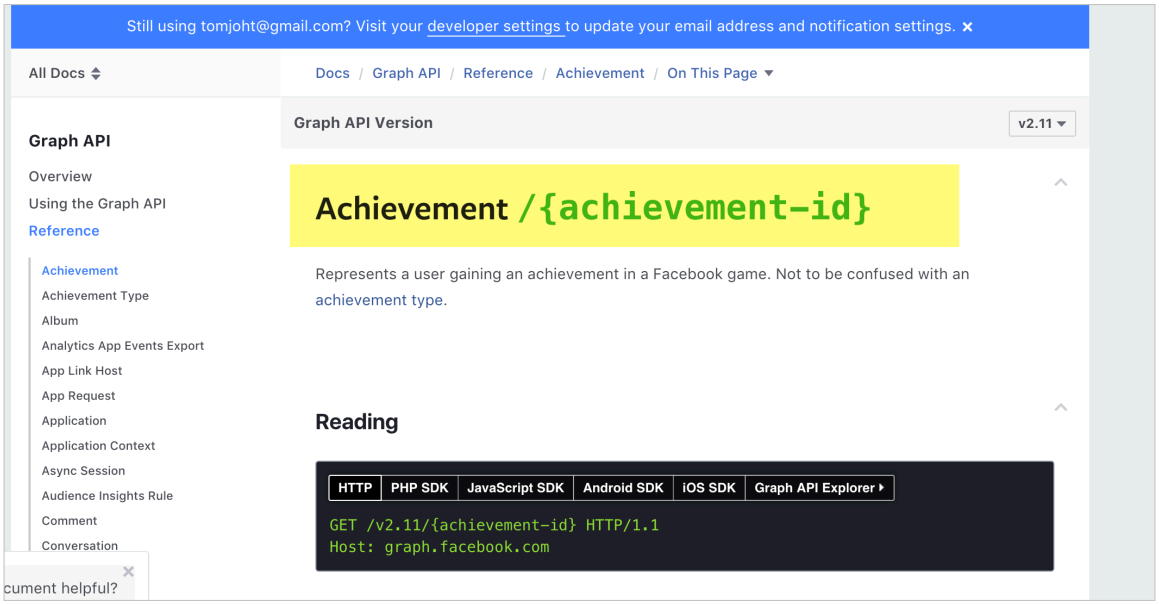
```
/campaigns/{campaign_id}/actions/send
```

If you can, put the path parameter in another color to set it off:

```
/campaigns/{campaign_id}/actions/send
```

Curly braces for path parameters are a convention that users will understand. In the above example, almost no endpoint uses curly braces in the actual path syntax, so the `{campaign_id}` is an obvious placeholder.

Here's an example from the Facebook API that colors the path parameter in an easily identifiable way:



Still using tomjoht@gmail.com? Visit your developer settings to update your email address and notification settings.

Achievement /{achievement-id}

Represents a user gaining an achievement in a Facebook game. Not to be confused with an achievement type.

Reading

```
GET /v2.11/{achievement-id} HTTP/1.1
Host: graph.facebook.com
```

When the parameters are described in Facebook's docs, the same green color is used to set off the parameters, which helps users recognize their meaning.

Path parameters aren't always set off with a unique color (for example, some precede it with a colon), but whatever the convention, make sure the path parameter is easily identifiable.

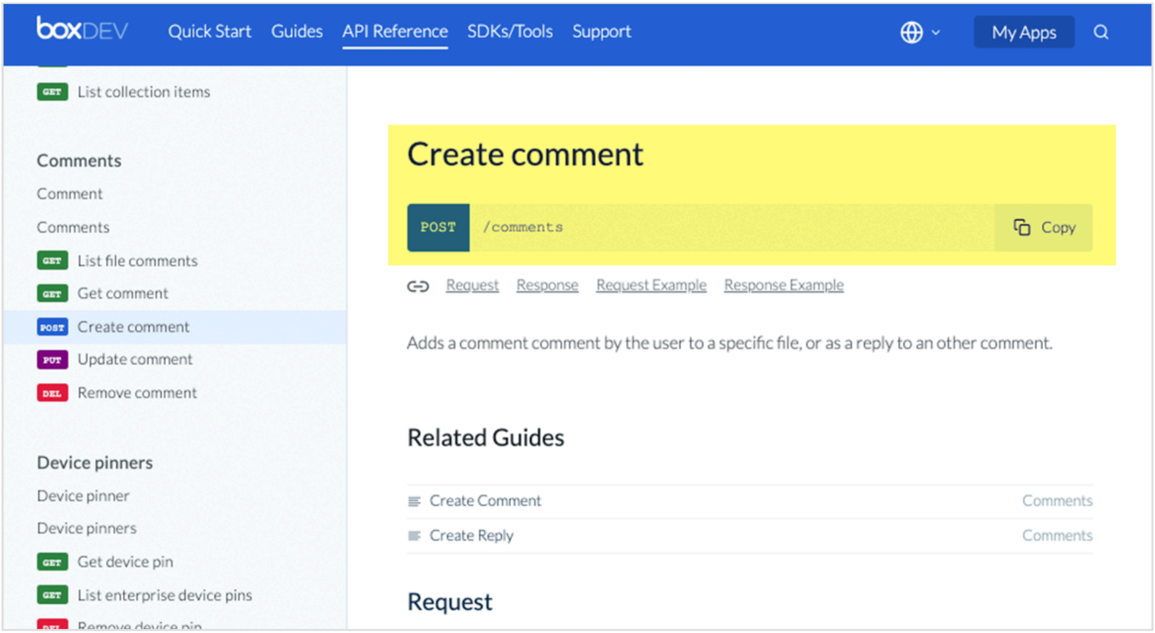
You can list the method next to the endpoint

It's common to list the method (GET, POST, and so on) next to the endpoint. The method defines the operation with the resource. Briefly, each method is as follows:

- GET: Retrieves a resource
- POST: Creates a resource
- PUT: Updates or creates within an existing resource
- PATCH: Partially modifies an existing resource
- DELETE: Removes the resource

See Request methods in Wikipedia's article on HTTP for more details. (There are some additional methods, but they're rarely used.)

Since there's not much to say about the method itself, it makes sense to group the method with the endpoint. Here's an example from the Box API:



Create comment

`POST /comments`

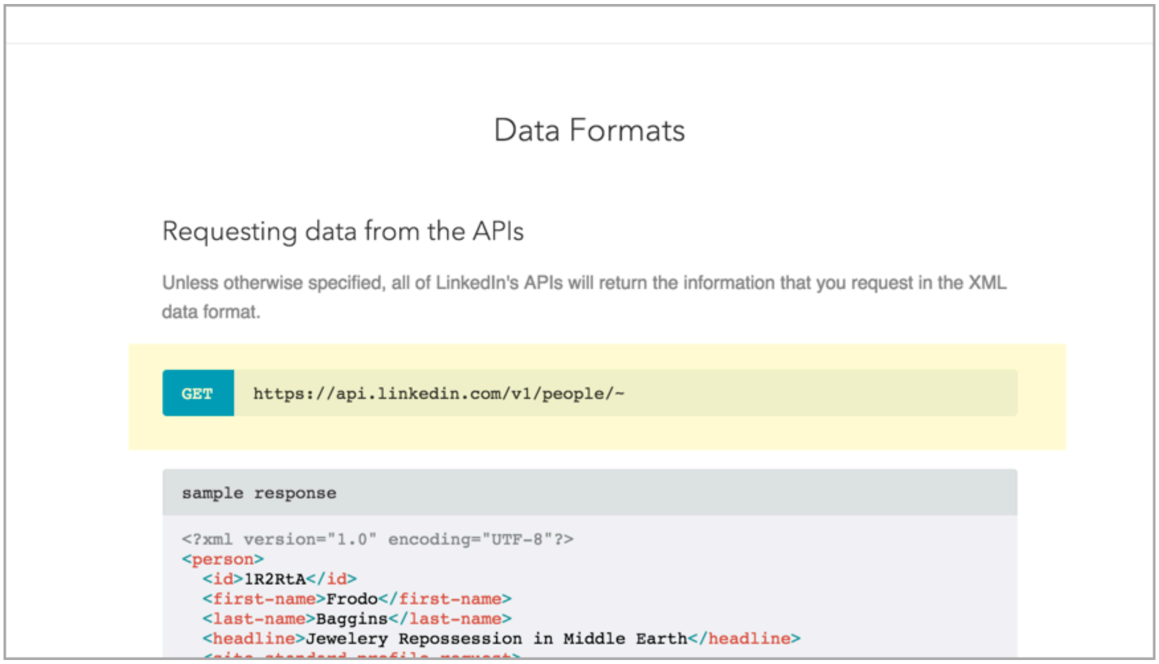
Adds a comment comment by the user to a specific file, or as a reply to an other comment.

Related Guides

Create Comment	Comments
Create Reply	Comments

Request

And here's an example from the Linkedin API:



Data Formats

Requesting data from the APIs

Unless otherwise specified, all of LinkedIn's APIs will return the information that you request in the XML data format.

```
GET https://api.linkedin.com/v1/people/~
```

sample response

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
  <id>1R2RtA</id>
  <first-name>Frodo</first-name>
  <last-name>Baggins</last-name>
  <headline>Jewelery Repossession in Middle Earth</headline>
```

Sometimes the method is referred to as the "verb." GET, PUT, POST, PATCH, and DELETE are all verbs or actions.

The endpoint shows the end path only

When you describe the endpoint, you list the end path only (hence the term "*end* point"). The full path that contains both the base path and the endpoint is often called a resource URL.

In our sample API scenario, the endpoint is just `/surfreport/{beachId}`. You don't have to list the full resource URL every time (which would be `https://api.openweathermap.org/surfreport/{beachId}`). Including the full resource URL would distract users from focusing on the path that matters. In your user guide, you usually explain the full resource URL, along with the required authorization (p. 393), in an introductory section (such as the Getting started tutorial (p. 379)).

How to group multiple endpoints for the same resource

Another consideration in documenting endpoints and methods is how to group and list the endpoints, particularly if you have a lot of endpoints for the same resource. In Examples of resource descriptions (p. 120), we looked at a variety of APIs. Many doc sites provide different designs for grouping or listing each endpoint for the resource, so I won't revisit all the same examples. Group the endpoints in some way that makes sense, such as by method or by the type of information returned.

For example, suppose you have three GET endpoints and one POST endpoint, all of which relate to the same resource. Some doc sites might list all the endpoints for the same resource on the same page. Others might break them into separate pages. Others might create one group for the GET endpoints and another for the POST endpoints. It depends how much you have to say about each endpoint.

If the endpoints are mostly the same, consolidating them on a single page could make sense. But if they're substantially unique (with different responses, parameters, and error messages), separating them onto different pages is probably better (and easier to manage). Then again, with a more sophisticated website design, you can make lengthy information navigable on the same page.

In a later section on design patterns (p. 540), I explain that long pages (p. 546) are common pattern with developer docs, in part because they make content easily findable for developers using Ctrl + F.

How to refer to endpoints in tutorials

In tutorials and other conceptual content, how do you refer to the endpoints within an API reference topic? Referring to the "`/aqi` endpoint" or to the "`/weatherdata`" endpoint doesn't make a huge difference. But with more complex APIs, using the endpoint to talk about the resource can be tricky.

At one company I worked at, our URLs for the Rewards endpoints looked like this:

```
/rewards

/rewards/{rewardId}

/users/{userId}/rewards

/users/{userId}/rewards/{rewardId}
```

And rewards in context of Missions looked like this:

```
/users/{userId}/rewards/{missionId}

/missions/{missionid}/rewards
```

To say that you could use the rewards resource wasn't always specific enough, because there were multiple rewards and missions endpoints.

It can get awkward referring to the endpoint. For example, you might have a sentence like this: “When you call `/users/{userId}/rewards/`, you get a list of all rewards. To get a specific reward for a specific mission for a specific user, the `/users/{userId}/rewards/{missionId}` endpoint takes several parameters…”

The longer the endpoint, the more cumbersome the reference becomes. These kinds of descriptions are more common in the conceptual sections (p. 358) of your documentation. Generally, there’s not a clear convention about how to refer to cumbersome endpoints. Adopt an approach that makes the most sense for your API.

Endpoint for surfreport API

Let’s create the Endpoints and methods section for our fictitious surfreport API (p. 115). Here’s my approach:

Endpoints

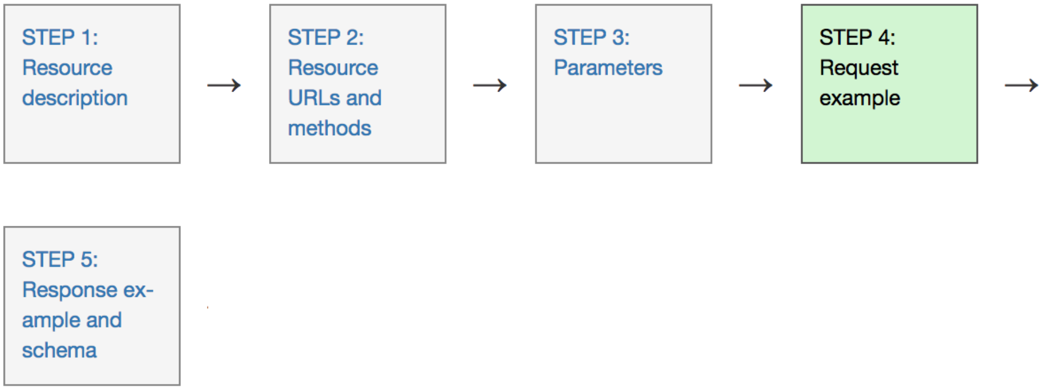
GET **surfreport/{beachId}**

Gets the surf conditions for a specific beach ID.

Next steps

Now that we’ve described the resource and listed the endpoints and methods, it’s time to tackle one of the most important parts of an API reference topic: the parameters section (p. 132).

Step 3: Parameters (API reference tutorial)

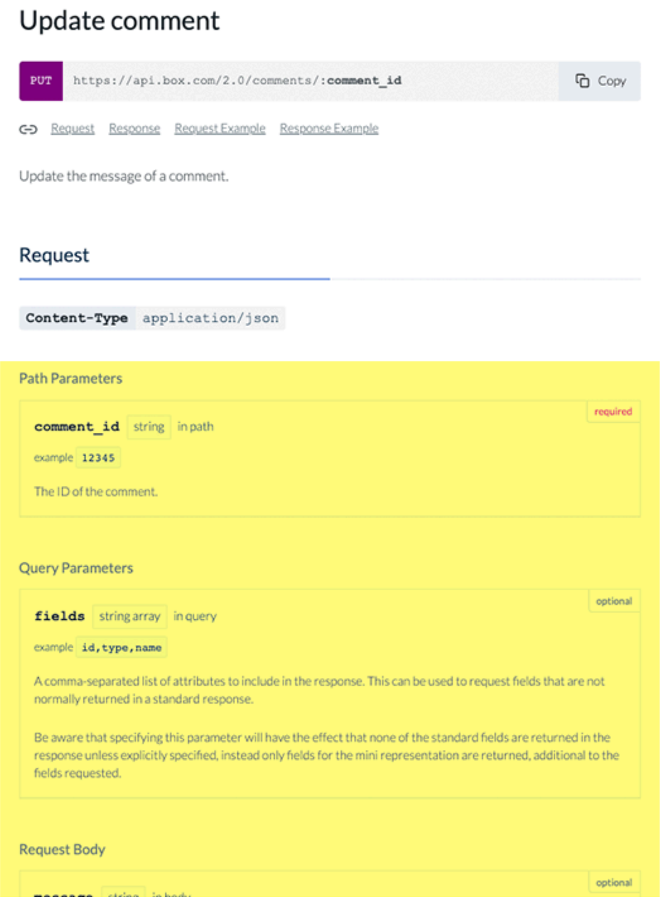

STEP 1: Resource description
→
STEP 2: Resource URLs and methods
→
STEP 3: Parameters
→
STEP 4: Request example
→
STEP 5: Response example and schema

Parameters are options you can pass with the endpoint (such as specifying the response format or the amount returned) to influence the response. There are several types of parameters: header parameters, path parameters, and query string parameters. Request bodies are closely similar to parameters but are not technically a parameter.

The different types of parameters are often documented in separate groups on the same page. Not all endpoints contain each type of parameter.

Examples of parameters

The following screenshot shows a sample parameters section with the Box API:



Update comment

`https://api.box.com/2.0/comments/:comment_id`

Update the message of a comment.

Request

Content-Type `application/json`

Path Parameters

comment_id string in path required

example **12345**

The ID of the comment.

Query Parameters

fields string array in query optional

example **id,type,name**

A comma-separated list of attributes to include in the response. This can be used to request fields that are not normally returned in a standard response.

Be aware that specifying this parameter will have the effect that none of the standard fields are returned in the response unless explicitly specified, instead only fields for the mini representation are returned, additional to the fields requested.

Request Body

optional

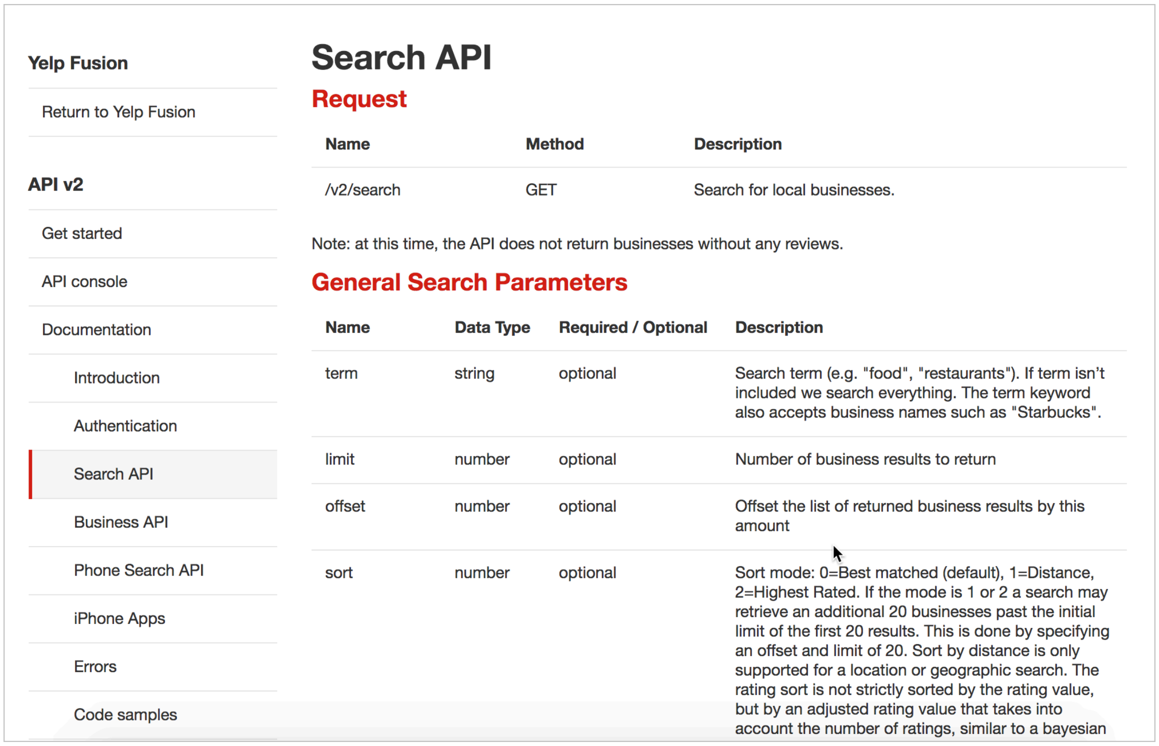
Sample parameters from Box API

In this example, the parameters are grouped by type: path parameters, query parameters, and body parameters. The endpoint also sets off the path parameter (`comment_id`) in a recognizable way in the endpoint definition.

Many times parameters are simply listed in a table or definition list like this:

Parameter	Required/Optional	Data Type
`format`	Optional	String

Here's an example from Yelp's documentation:



Search API

Request

Name	Method	Description
/v2/search	GET	Search for local businesses.

Note: at this time, the API does not return businesses without any reviews.

General Search Parameters

Name	Data Type	Required / Optional	Description
term	string	optional	Search term (e.g. "food", "restaurants"). If term isn't included we search everything. The term keyword also accepts business names such as "Starbucks".
limit	number	optional	Number of business results to return
offset	number	optional	Offset the list of returned business results by this amount
sort	number	optional	Sort mode: 0=Best matched (default), 1=Distance, 2=Highest Rated. If the mode is 1 or 2 a search may retrieve an additional 20 businesses past the initial limit of the first 20 results. This is done by specifying an offset and limit of 20. Sort by distance is only supported for a location or geographic search. The rating sort is not strictly sorted by the rating value, but by an adjusted rating value that takes into account the number of ratings, similar to a bayesian

You can format the values in a variety of ways (aside from a table). If you're using a definition list or other non-table format, be sure to develop styles that make the values easily readable.

Several types of parameters

REST APIs have several types of parameters:

- **Header parameters (p. 135)**: Parameters included in the request header, usually related to authorization.
- **Path parameters (p. 136)**: Parameters within the path of the endpoint, before the query string (`?`). These are usually set off within curly braces.
- **Query string parameters (p. 136)**: Parameters in the query string of the endpoint, after the `?`.

Another property closely related to parameters, and which used to be referred to as a parameter in OpenAPI v2.0, is the request body, or `requestBody` in OpenAPI code form. The request body is usually only used with CREATE or PUT methods and often includes a JSON object included in the body of the request. More details are provided in Request bodies (p. 137).

The terms for each of these parameter types comes from the OpenAPI specification (p. 272), which defines a formal specification that includes descriptions of each parameter type (see the Path object tutorial (p. 291)). Using industry standard terminology helps you develop a vocabulary to describe different elements of an API.

What to note in parameter documentation

Regardless of the parameter type, define the following with each parameter:

- Data type (p. 135)
- Max and min value (p. 135)

Data types for parameters

APIs may not process the parameter correctly if it's the wrong data type or wrong format. Listing the data type is usually a good idea with all parameter types but is especially true for request bodies, since these are typically formatted in JSON.

These data types are the most common with REST APIs:

- **string**: An alphanumeric sequence of letters and/or numbers
- **integer**: A whole number — can be positive or negative
- **boolean**: True or false value
- **object**: Key-value pairs in JSON format
- **array**: A list of values

There are more data types in programming, and if you have more specific data types that are important to note, be sure to document them. In Java, for example, it's important to note the data type allowed because Java allocates memory space based on the size of the data. As such, Java gets much more specific about the size of the numbers. You have a byte, short, int, double, long, float, char, boolean, and so on. However, you usually don't have to specify this level of detail with a REST API.

Max and min values for parameters

In addition to specifying the data type, the parameters should indicate the maximum, minimum, and allowed values if appropriate. For example, if the weather API allows only longitude and latitude coordinates of specific countries, these limits should be described in the parameters documentation. Omitting information about max/min values or other prohibited values (when applicable) is a common pitfall in docs.

Not every parameter needs max and min values, however. Note these exceptions:

- **Booleans**: With Booleans, the only options are true or false, so there's no need for max/min values.
- **Strings that use enums**: If a string restricts possible values to enums (an enumerated list), the max/min values wouldn't be appropriate. For example, a geo-related enum might allow only these values: north, south, east, west. There is no max/min value in this case.

In general, as you document parameters, if the parameters allow more freeform values (outside of Booleans and enums), consider ways that developers might break the API. For example, if the API provides an ID field, try entering an ID that is 300 characters long. If you can submit a file attachment, try submitting an 80 MB file.

Your developer audience needs to know the limits applicable to fields. Many times your product team might not even know what limitations exist. (Your QA team should know, though, since it's their job to try to push the system to its limits and break it.)

When you test an API, try running an endpoint without the required parameters, or with the wrong parameters, or with values that exceed the max or min amounts. See what kind of error response comes back. Include that response in your status and error codes section (p. 402). I talk more about the importance of testing in Testing your docs (p. 342).

Header parameters

Header parameters are included in the request header. Usually, the header just includes authorization parameters that are common across all endpoints; as a result, the header parameters aren't usually documented with each endpoint. Instead, the authorization details in header parameters are documented in the authorization requirements section (p. 393).

However, if your endpoint requires unique parameters to be passed in the header, you would document them in the parameters documentation within each endpoint.

Path parameters

Path parameters are part of the endpoint itself and are not optional. For example, in the following endpoint, `{user}` and `{bicycleId}` are required path parameters:

```
/service/myresource/user/{user}/bicycles/{bicycleId}
```

Path parameters are usually set off with curly braces, but some API doc styles precede the value with a colon or use a different syntax. When you document path parameters, indicate the default values, the allowed values, and other details.

Color coding the path parameters

When you list the path parameters in your endpoint, it can help to color code the parameters to make them more easily identifiable. Color coding the parameters makes it clear what's a path parameter and what's not. Through color, you create an immediate connection between the endpoint and the parameter definitions.

For example, you could color code your parameters like this:

```
/service/myresource/user/{user}/bicycles/{bicycleId}
```

You could then use the same color for these parameters in later descriptions:

URL Parameter	Description
`user`	Here's my description of the user parameter.
`bicycleId`	Here's my description of the bicycles parameter.

By color coding the parameters, it's easy to see the parameter being defined and how it relates to the endpoint definition.

Query string parameters

Query string parameters appear after a question mark (`?`) in the endpoint. The question mark followed by the parameters and their values is referred to as the "query string." In the query string, each parameter is listed one right after the other with an ampersand (`&`) separating them. The order of the query string parameters does not matter.

For example:

```
/surfreport/{beachId}?days=3&units=metric&time=1400
```

and

```
/surfreport/{beachId}?time=1400&units=metric&days=3
```

would return the same result.

However, with path parameters, the order *does* matter. If the parameter is part of the actual endpoint (not added after the query string), you usually describe this value in the description of the endpoint itself.

Request bodies

Frequently, with POST requests (where you're creating something), you submit a JSON object in the request body. This is known as a request body, and the format is usually JSON. This JSON object may be a lengthy list of key-value pairs with multiple levels of nesting.

For example, the endpoint may be something simple, such as `/surfreport/{beachId}`. But in the body of the request, you might include a JSON object with many key-value pairs, like this:

```
{
"days": 2,
"units": "imperial",
"time": 1433524597
}
```

In OpenAPI v2.0, request bodies were classified as a type of parameter, but in v3.0, they are not considered a parameter but rather a path property. Given that the request body functions like a parameter, I've decided to leave them classified as a parameter for now. However, note that in the OpenAPI spec, request bodies are technically not a parameter.

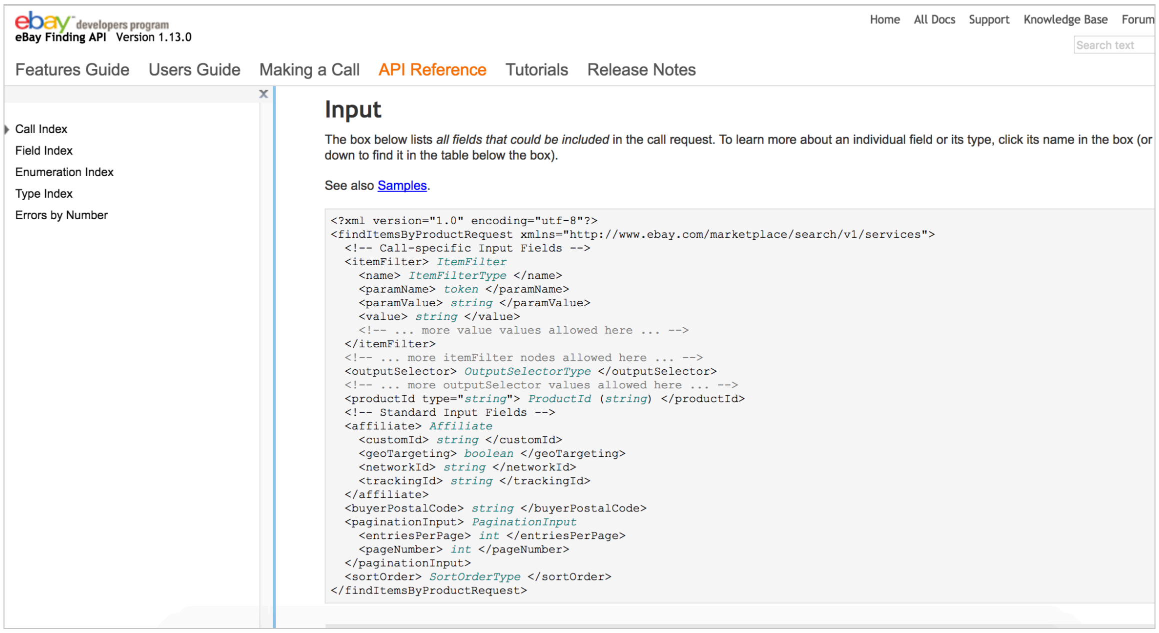
Documenting complex request bodies

Documenting JSON data (both in request bodies and responses) is one of the trickier parts of API documentation. Documenting a JSON object is easy if the object is simple, with just a few key-value pairs at the same level. But if you have a JSON object with multiple objects inside objects, numerous levels of nesting, and lengthy conditional data, it can be tricky. And if the JSON object spans more than 100 lines, or 1,000, you'll need to think carefully about how you present the information.

Tables work all right for documenting JSON, but in a table, it can be hard to distinguish between top-level and sub-level items. The object that contains an object that also contains an object, and another object, etc., can be confusing to represent.

By all means, if the JSON object is relatively small, a table is probably your best option. But there are other approaches that designers have taken as well.

Take a look at eBay's findItemsByProduct ↗ resource. Here's the request body (in this case, the format is XML):



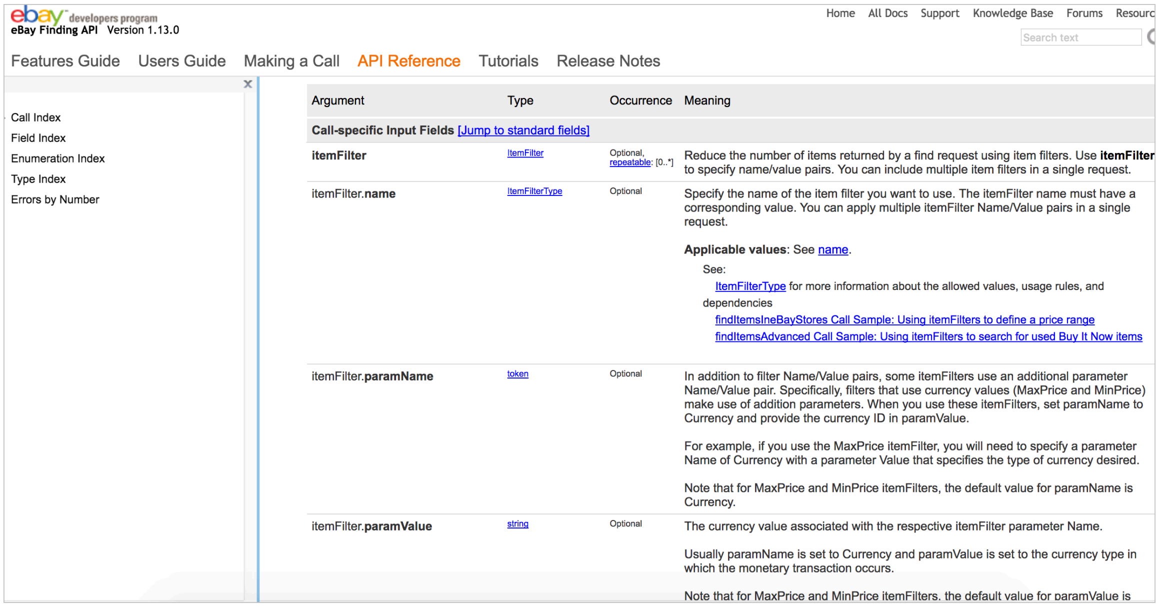
Input

The box below lists *all fields that could be included* in the call request. To learn more about an individual field or its type, click its name in the box (or down to find it in the table below the box).

See also Samples.

```
<?xml version="1.0" encoding="utf-8"?>
<findItemsByProductRequest xmlns="http://www.ebay.com/marketplace/search/v1/services">
  <!-- Call-specific Input Fields -->
  <itemFilter> ItemFilter
    <name> ItemFilterType </name>
    <paramName> token </paramName>
    <paramValue> string </paramValue>
    <value> string </value>
    <!-- ... more value values allowed here ... -->
  </itemFilter>
  <!-- ... more itemFilter nodes allowed here ... -->
  <outputSelector> OutputSelectorType </outputSelector>
  <!-- ... more outputSelector values allowed here ... -->
  <productId type="string"> ProductId (string) </productId>
  <!-- Standard Input Fields -->
  <affiliate> Affiliate
    <customId> string </customId>
    <geoTargeting> boolean </geoTargeting>
    <networkId> string </networkId>
    <trackingId> string </trackingId>
  </affiliate>
  <buyerPostalCode> string </buyerPostalCode>
  <paginationInput> PaginationInput
    <entriesPerPage> int </entriesPerPage>
    <pageNumber> int </pageNumber>
  </paginationInput>
  <sortOrder> SortOrderType </sortOrder>
</findItemsByProductRequest>
```

Below the request body is a table that describes each parameter:



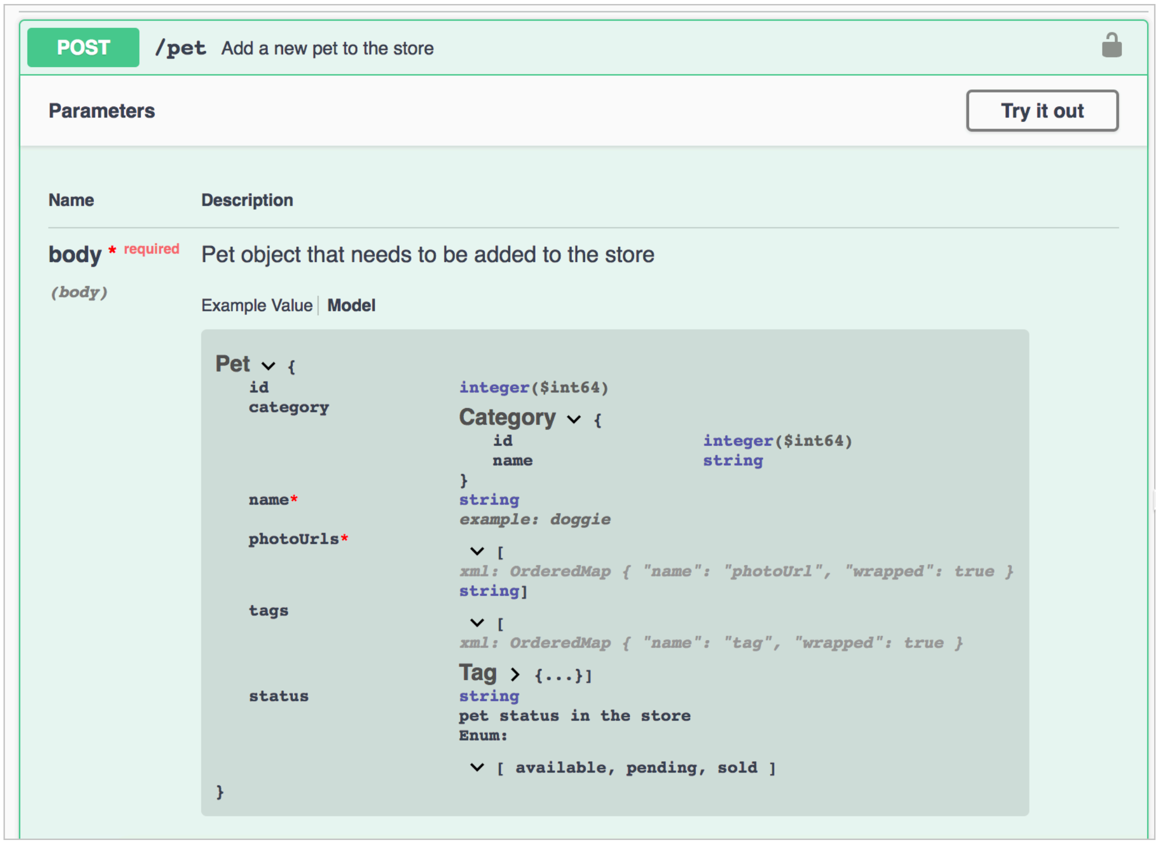
Argument	Type	Occurrence	Meaning
Call-specific Input Fields [Jump to standard fields]			
itemFilter	ItemFilter	Optional, repeatable: [0..*]	Reduce the number of items returned by a find request using item filters. Use **itemFilter** to specify name/value pairs. You can include multiple item filters in a single request.
itemFilter.**name**	ItemFilterType	Optional	Specify the name of the item filter you want to use. The itemFilter name must have a corresponding value. You can apply multiple itemFilter Name/Value pairs in a single request. **Applicable values**: See name. See: ItemFilterType for more information about the allowed values, usage rules, and dependencies findItemsIneBayStores Call Sample: Using itemFilters to define a price range findItemsAdvanced Call Sample: Using itemFilters to search for used Buy It Now items
itemFilter.**paramName**	token	Optional	In addition to filter Name/Value pairs, some itemFilters use an additional parameter Name/Value pair. Specifically, filters that use currency values (MaxPrice and MinPrice) make use of addition parameters. When you use these itemFilters, set paramName to Currency and provide the currency ID in paramValue. For example, if you use the MaxPrice itemFilter, you will need to specify a parameter Name of Currency with a parameter Value that specifies the type of currency desired. Note that for MaxPrice and MinPrice itemFilters, the default value for paramName is Currency.
itemFilter.**paramValue**	string	Optional	The currency value associated with the respective itemFilter parameter Name. Usually paramName is set to Currency and paramValue is set to the currency type in which the monetary transaction occurs. Note that for MaxPrice and MinPrice itemFilters, the default value for paramValue is

But the sample request also contains links to each of the parameters. When you click a parameter value in the sample request, you go to a page that provides more details about that parameter value, such as the `ItemFilter` ↗. The separate page with more detail is likely because the parameter values are more complex and require detailed explanation.

The same parameter values might be used in other requests as well, so eBay's approach likely facilitates re-use. Even so, I dislike jumping around to other pages for the information I need.

Swagger UI's approach to request bodies

Swagger UI (p. 218), which we explore later and also demo (p. 240), provides another approach to documenting the request bodies. Swagger UI shows the request bodies in the format that you see below. Swagger UI lets you toggle between an "Example Value" and a "Model" view for both responses and request bodies.



body * required *(body)* Pet object that needs to be added to the store

```
Pet ∨ {
    id                  integer($int64)
    category            Category ∨ {
                            id                  integer($int64)
                            name                string
                        }
    name*               string
                        example: doggie
    photoUrls*          ∨ [
                        xml: OrderedMap { "name": "photoUrl", "wrapped": true }
                        string]
    tags                ∨ [
                        xml: OrderedMap { "name": "tag", "wrapped": true }
                        Tag > {...}]
    status              string
                        pet status in the store
                        Enum:
                         ∨ [ available, pending, sold ]
}
```

See the Swagger Petstore to explore the demo here. The Example Value shows a sample of the syntax along with examples. When you click the **Model** link, you see a sample request body and any descriptions of each element.

The Model includes expand/collapse toggles with the values. (The Petstore demo doesn't include many parameter descriptions in the Model, but if you include descriptions, they would appear here in the Model rather than in the Example Value.)

We'll get into Swagger in much more detail in Introduction to the OpenAPI specification (p. 181). For now, focus on these core elements of API reference documentation. You will see these same sections appear in the OpenAPI specification.

You can see that there's a lot of variety in documenting JSON and XML in request bodies. There's no right way to document the information. As always, choose the method that depicts your API's parameters in the clearest, easiest-to-read way.

If you have relatively simple parameters, your choice won't matter that much. But if you have complex, unwieldy parameters, you may have to resort to custom styling and templates to present them more clearly. I explore this topic in more depth in the Response example and schema section (p. 154).

Parameters for the surfreport endpoint

For our new surfreport resource, let’s look through the parameters available and create a table describing the parameters. Here’s what my parameter information looks like:

Parameters

Path parameters

Path parameter	Description
`{beachId}`	The value for the beach you want to look up. Valid `beachId` values are available from our site at sampleurl.com.

Query string parameters

Query string parameter	Required / optional	Description	Type
`days`	Optional	The number of days to include in the response. Default is 3.	Integer
`time`	Optional	If you include the time, then only the current hour will be returned in the response.	Integer. Unix format (ms since 1970) in UTC.

Even if you use Markdown for docs, you might consider using HTML syntax with tables. You usually want the control over column widths to make some columns wider or narrower. Markdown doesn’t allow that granular level of control. With HTML, you can use a `colgroup` property to specify the `col width` for each column.

Next steps

Now that we’ve documented the parameters, it’s time to show a sample request (p. 141) for the resource.

Step 4: Request example (API reference tutorial)

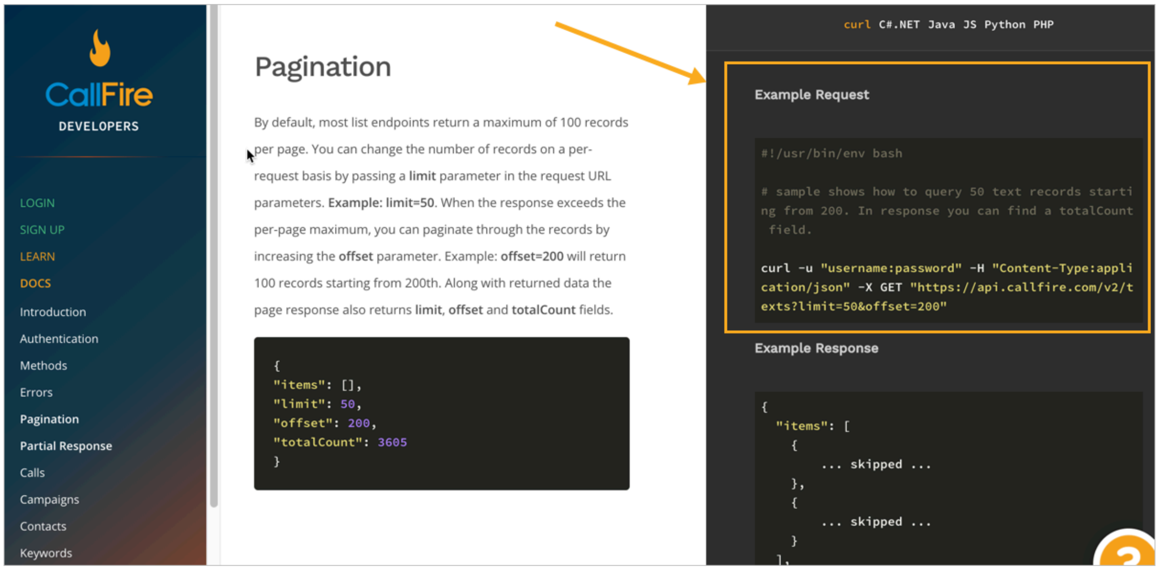
STEP 1: Resource description → STEP 2: Resource URLs and methods → STEP 3: Parameters → **STEP 4: Request example** → STEP 5: Response example and schema

The request example includes a sample request using the endpoint, showing some parameters configured. The request example usually doesn't show all possible parameter configurations, but it should be as rich as possible with parameters.

Sample requests sometimes include code snippets that show the same request in a variety of languages (besides curl). Requests shown in other programming languages are optional and not always included in the reference topics (but when available, users welcome them).

Examples of requests

The following example shows a sample request from the Callfire API ↗:



Pagination

By default, most list endpoints return a maximum of 100 records per page. You can change the number of records on a per-request basis by passing a **limit** parameter in the request URL parameters. **Example: limit=50**. When the response exceeds the per-page maximum, you can paginate through the records by increasing the **offset** parameter. Example: **offset=200** will return 100 records starting from 200th. Along with returned data the page response also returns **limit**, **offset** and **totalCount** fields.

```
{
"items": [],
"limit": 50,
"offset": 200,
"totalCount": 3605
}
```

Example Request

```
#!/usr/bin/env bash

# sample shows how to query 50 text records starti
ng from 200. In response you can find a totalCount
 field.

curl -u "username:password" -H "Content-Type:appli
cation/json" -X GET "https://api.callfire.com/v2/t
exts?limit=50&offset=200"
```

Example Response

```
{
  "items": [
    {
        ... skipped ...
    },
    {
        ... skipped ...
    }
  ],
```

Sample request from Callfire

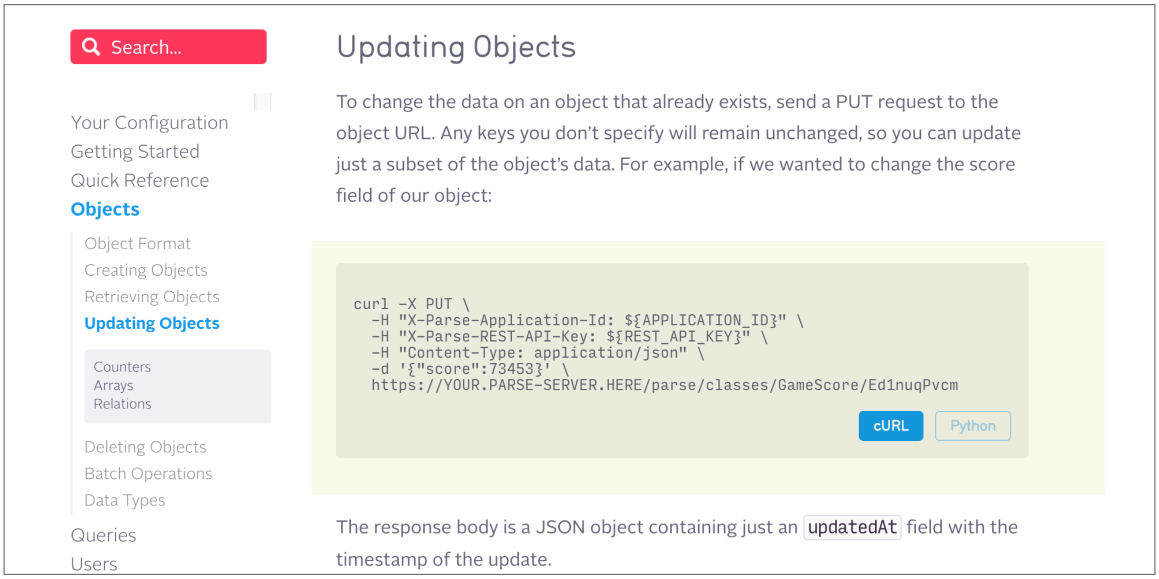
The design of this API doc site arranges the sample requests and responses in the right column of a three-column layout. The request is formatted in curl, which we explored earlier in Make a curl call (p. 80).

```
curl -u "username:password" -H "Content-Type:application/json" -X GET "http
s://api.callfire.com/v2/texts?limit=50&offset=200"
```

curl is a common format to show requests for several reasons:

- curl is language agnostic, so it’s not specific to one particular programming language.
- curl shows the header information required in the request.
- curl shows the method used with the request.

In general, use curl to show your sample request. Here’s another example of a curl request in the Parse API:



Updating Objects

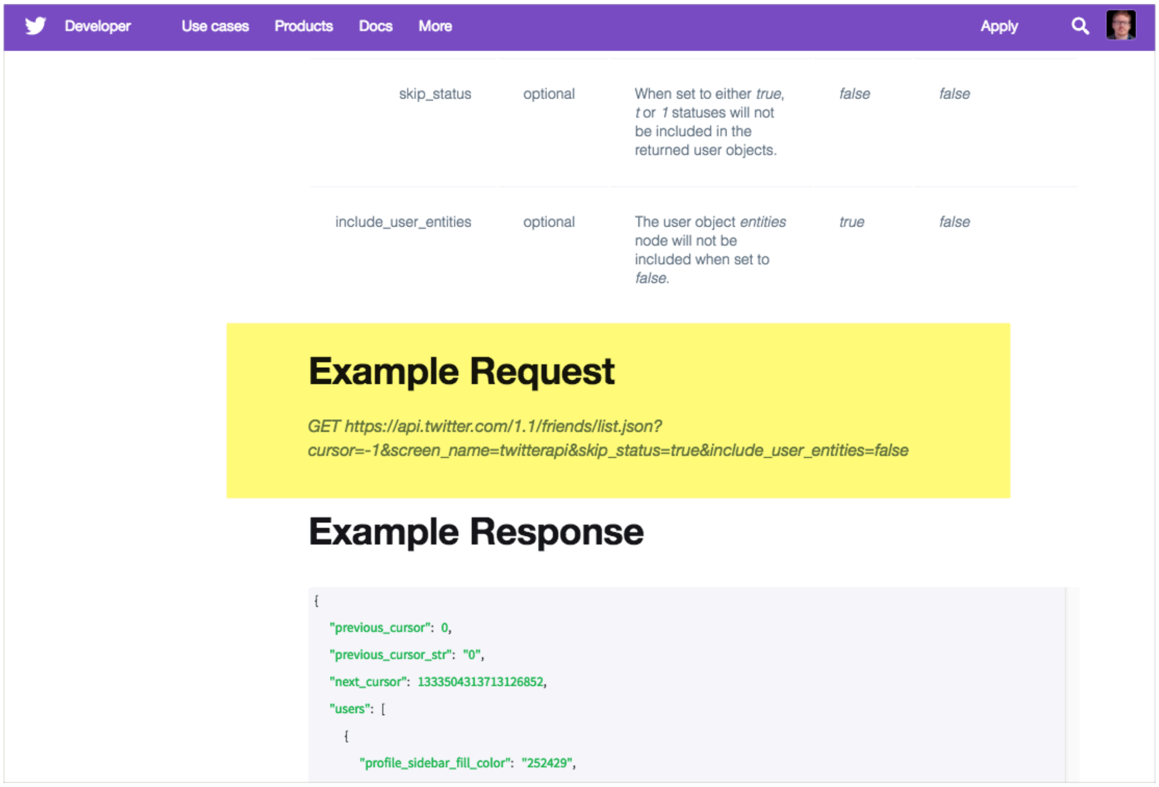
To change the data on an object that already exists, send a PUT request to the object URL. Any keys you don't specify will remain unchanged, so you can update just a subset of the object's data. For example, if we wanted to change the score field of our object:

```
curl -X PUT \
  -H "X-Parse-Application-Id: ${APPLICATION_ID}" \
  -H "X-Parse-REST-API-Key: ${REST_API_KEY}" \
  -H "Content-Type: application/json" \
  -d '{"score":73453}' \
  https://YOUR.PARSE-SERVER.HERE/parse/classes/GameScore/Ed1nuqPvcm
```

The response body is a JSON object containing just an `updatedAt` field with the timestamp of the update.

You can add backslashes in curl to separate each parameter onto its own line (though, as I pointed out in the curl tutorial (p. 81), Windows has trouble with backslashes).

Other API doc sites might use the full resource URL, such as this plain example from Twitter:



skip_status	optional	When set to either *true*, *t* or *1* statuses will not be included in the returned user objects.	*false*	*false*
include_user_entities	optional	The user object *entities* node will not be included when set to *false*.	*true*	*false*

Example Request

GET https://api.twitter.com/1.1/friends/list.json?cursor=-1&screen_name=twitterapi&skip_status=true&include_user_entities=false

Example Response

```
{
  "previous_cursor": 0,
  "previous_cursor_str": "0",
  "next_cursor": 1333504313713126852,
  "users": [
    {
      "profile_sidebar_fill_color": "252429",
```

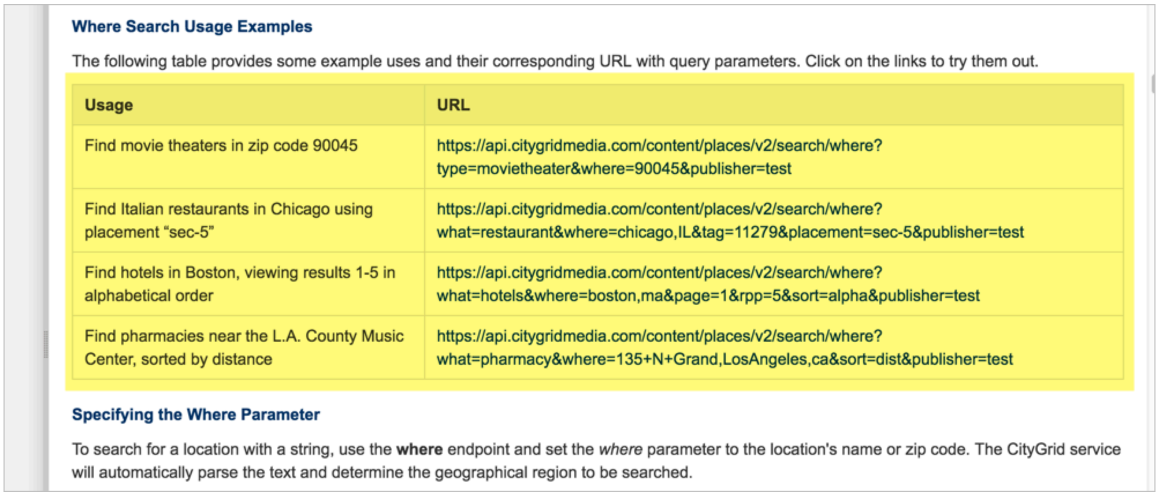
The resource URL includes both the base path and the endpoint. One problem with showing the full resource URL is that it doesn't indicate if any header information needs to be passed to authorize the request. (If your API consists of GET requests only and doesn't require authorization, great, but few APIs are set up this way.) curl requests can easily show any header parameters.

Multiple request examples

If you have a lot of parameters, consider including several request examples. In the CityGrid Places API, the `where` endpoint is as follows:

```
https://api.citygridmedia.com/content/places/v2/search/where
```

However, there are literally 17 possible query string parameters you can use with this endpoint. As a result, the documentation includes several sample requests that show various parameter combinations:



Where Search Usage Examples

The following table provides some example uses and their corresponding URL with query parameters. Click on the links to try them out.

Usage	URL
Find movie theaters in zip code 90045	https://api.citygridmedia.com/content/places/v2/search/where?type=movietheater&where=90045&publisher=test
Find Italian restaurants in Chicago using placement "sec-5"	https://api.citygridmedia.com/content/places/v2/search/where?what=restaurant&where=chicago,IL&tag=11279&placement=sec-5&publisher=test
Find hotels in Boston, viewing results 1-5 in alphabetical order	https://api.citygridmedia.com/content/places/v2/search/where?what=hotels&where=boston,ma&page=1&rpp=5&sort=alpha&publisher=test
Find pharmacies near the L.A. County Music Center, sorted by distance	https://api.citygridmedia.com/content/places/v2/search/where?what=pharmacy&where=135+N+Grand,LosAngeles,ca&sort=dist&publisher=test

Specifying the Where Parameter

To search for a location with a string, use the **where** endpoint and set the *where* parameter to the location's name or zip code. The CityGrid service will automatically parse the text and determine the geographical region to be searched.

Adding multiple request examples makes sense when the parameters wouldn't usually be used together. For example, there are few cases where you might actually include all 17 parameters in the same request, so any sample will be limited in what it can show.

This example shows how to "Find hotels in Boston, viewing results 1-5 in alphabetical order":

```
https://api.citygridmedia.com/content/places/v2/search/where?what=hotels&whe
re=boston,ma&page=1&rpp=5&sort=alpha&publisher=test&format=json
```

If you click the link, you can see the response directly. In the responses topic (p. 164), I get into more detail about dynamically showing the response when users click a request.

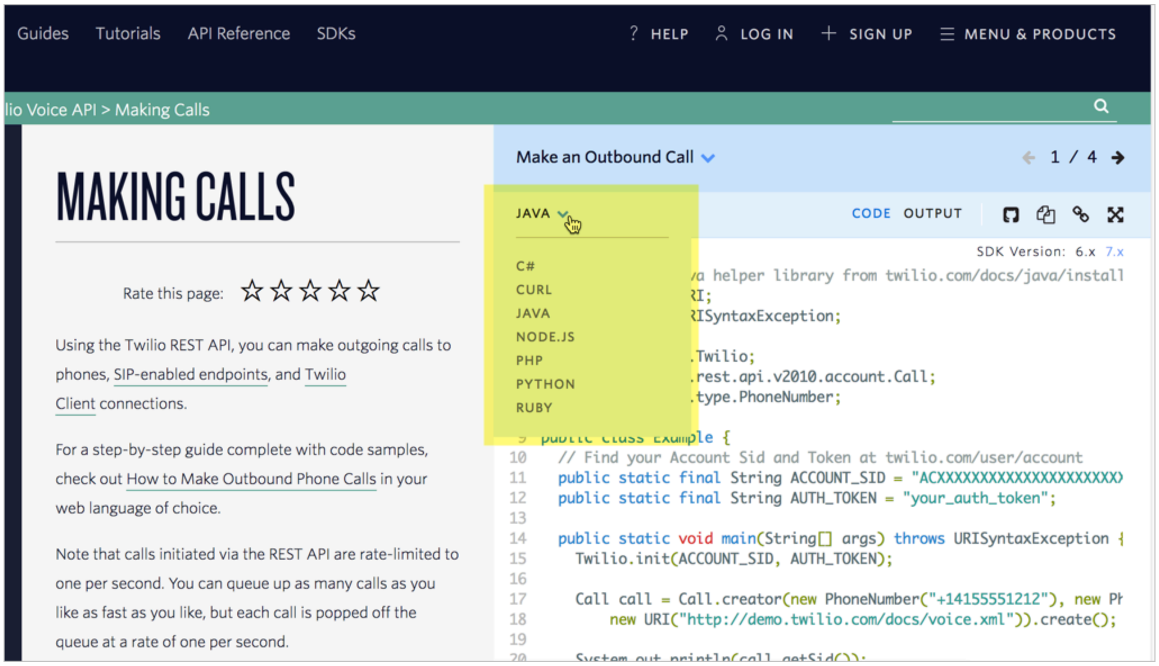
How many different requests and responses should you show? There's probably no easy answer, but probably no more than a few. You decide what makes sense for your API. Users will usually understand the pattern after a few examples.

Requests in various languages

As noted earlier, in What is a REST API? (p. 36), REST APIs are language agnostic. The universal protocol helps facilitate widespread adoption across programming languages. Developers can code their applications in any language, from Java to Ruby to JavaScript, Python, C#, Node JS, or something else. As long as developers can make an HTTP web request in that language, they can use the API. The response from the web request will contain the data in either JSON or XML.

Because you can't entirely know which language your end users will be developing in, it's kind of fruitless to try to provide code samples in every language. Many APIs just show the format for submitting requests and a sample response, and the authors will assume that developers will know how to submit HTTP requests in their particular programming language.

However, some APIs do show simple requests in a variety of languages. Here's an example from Twilio:



MAKING CALLS

Rate this page: ☆☆☆☆☆

Using the Twilio REST API, you can make outgoing calls to phones, SIP-enabled endpoints, and Twilio Client connections.

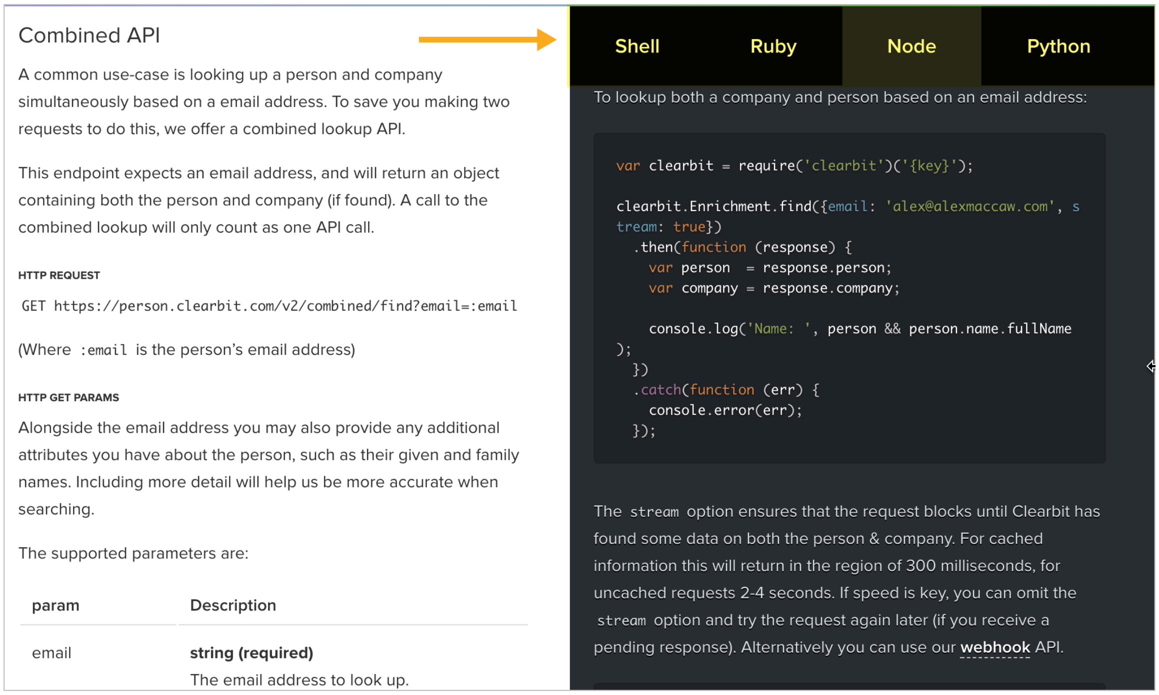
For a step-by-step guide complete with code samples, check out How to Make Outbound Phone Calls in your web language of choice.

Note that calls initiated via the REST API are rate-limited to one per second. You can queue up as many calls as you like as fast as you like, but each call is popped off the queue at a rate of one per second.

```
a helper library from twilio.com/docs/java/install
I;
ISyntaxException;

.Twilio;
.rest.api.v2010.account.Call;
.type.PhoneNumber;

public class Example {
  // Find your Account Sid and Token at twilio.com/user/account
  public static final String ACCOUNT_SID = "ACXXXXXXXXXXXXXXXXXXXXX
  public static final String AUTH_TOKEN = "your_auth_token";

  public static void main(String[] args) throws URISyntaxException {
    Twilio.init(ACCOUNT_SID, AUTH_TOKEN);

    Call call = Call.creator(new PhoneNumber("+14155551212"), new P
        new URI("http://demo.twilio.com/docs/voice.xml")).create();

    System.out.println(call.getSid());
```

You can select which language you want for the sample request: C#, curl, Java, Node.js, PHP, Python, or Ruby.

Here's another example from the Clearbit API:



Combined API

A common use-case is looking up a person and company simultaneously based on a email address. To save you making two requests to do this, we offer a combined lookup API.

This endpoint expects an email address, and will return an object containing both the person and company (if found). A call to the combined lookup will only count as one API call.

HTTP REQUEST

`GET https://person.clearbit.com/v2/combined/find?email=:email`

(Where `:email` is the person's email address)

HTTP GET PARAMS

Alongside the email address you may also provide any additional attributes you have about the person, such as their given and family names. Including more detail will help us be more accurate when searching.

The supported parameters are:

param	Description
email	**string (required)** The email address to look up.

To lookup both a company and person based on an email address:

```
var clearbit = require('clearbit')('{key}');

clearbit.Enrichment.find({email: 'alex@alexmaccaw.com', s
tream: true})
  .then(function (response) {
    var person  = response.person;
    var company = response.company;

    console.log('Name: ', person && person.name.fullName
);
  })
  .catch(function (err) {
    console.error(err);
  });
```

The `stream` option ensures that the request blocks until Clearbit has found some data on both the person & company. For cached information this will return in the region of 300 milliseconds, for uncached requests 2-4 seconds. If speed is key, you can omit the `stream` option and try the request again later (if you receive a pending response). Alternatively you can use our webhook API.

You can see the request in Shell (curl), Ruby, Node, or Python. Developers can easily copy the needed code into their applications, rather than figuring out how to translate the curl request into a particular programming language.

Providing a variety of requests like this, often displayed through tabs, helps make your API easier to implement. It's even better if you can automatically populate the API keys with the actual user's API keys based on their logged-in profile.

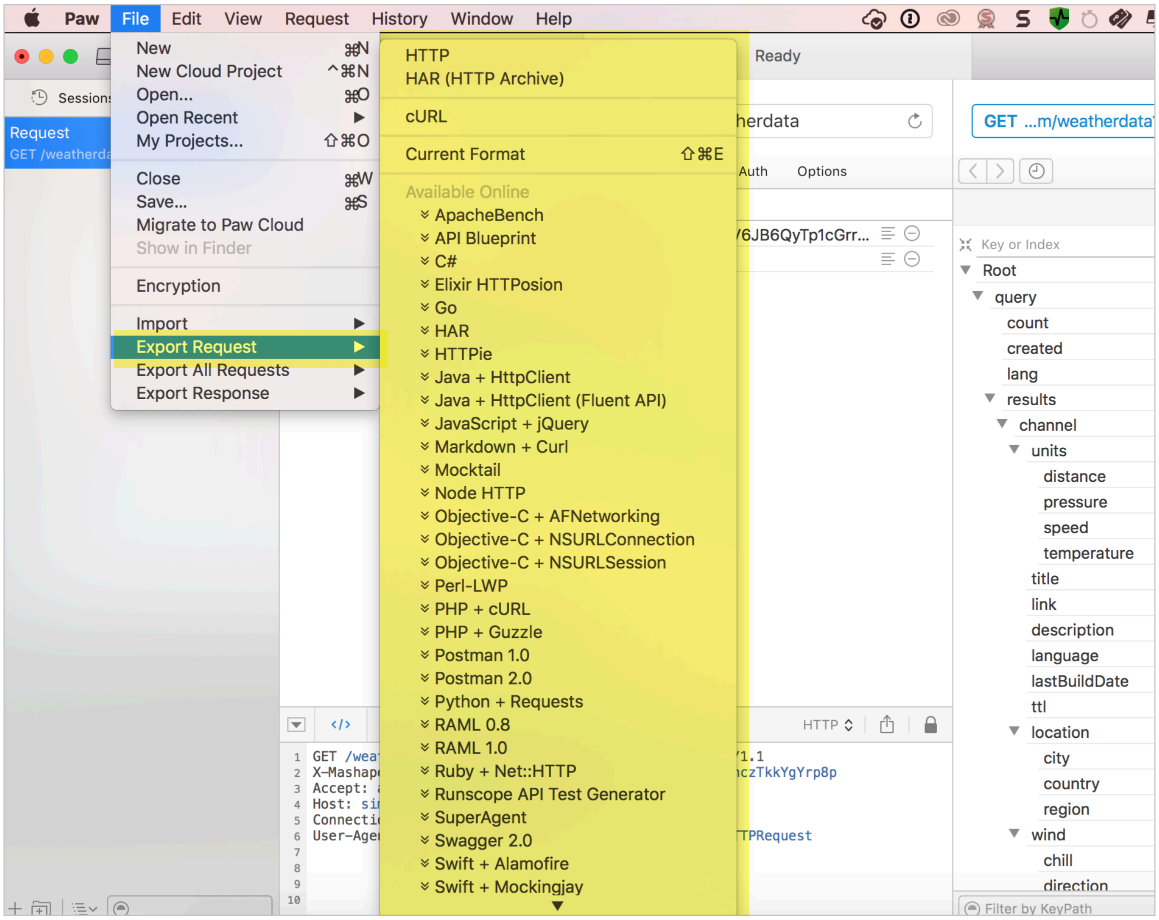
However, don't feel so intimidated by this smorgasbord of code samples. Some API doc tools (such as Readme.com or SwaggerHub (p. 230)) can automatically generate these code samples because the patterns for making REST requests in different programming languages follow a common template.

Many times, product managers know which programming languages the target users develop applications with. If you know the target audience's preferred programming language, you can focus your code samples on that language only.

Auto-generating code samples

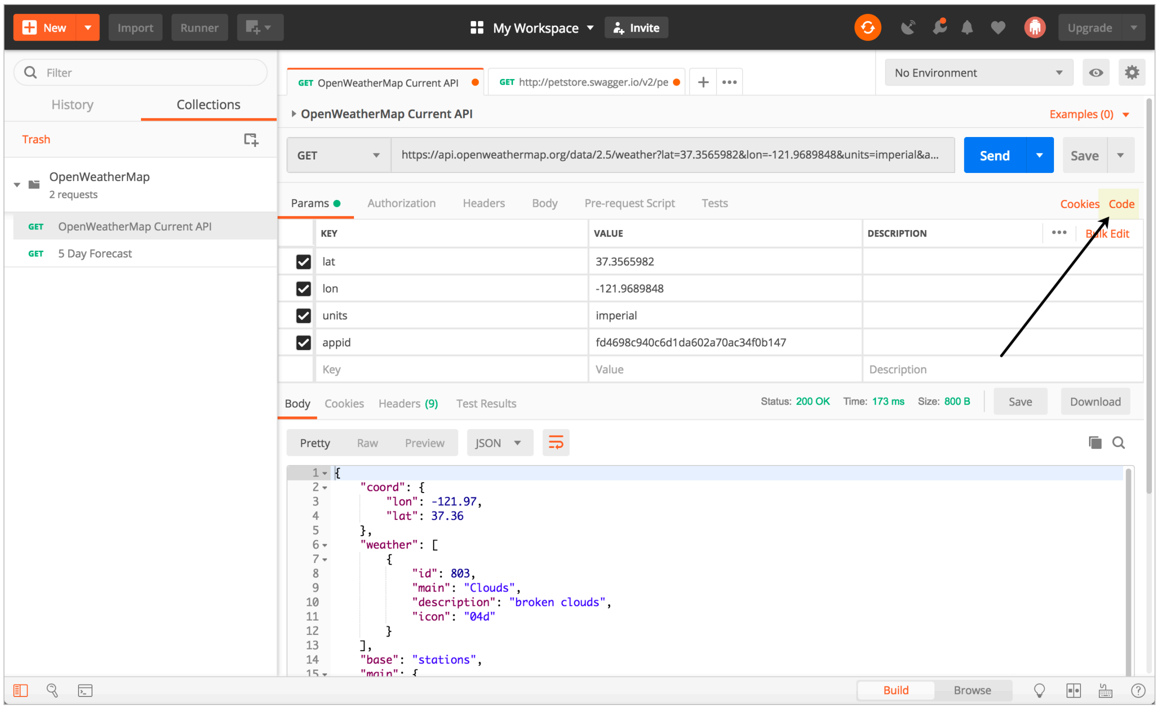
If you're not using an authoring tool that auto-generates code examples, and you want to provide these code snippets, you can auto-generate code samples from both Postman and Paw, if desired.

Paw (for Mac) lets you export your request into nearly every conceivable language:



After you have a request configured (a process similar to Postman (p. 71)), you can generate a code snippet by going to **File > Export Request**.

The Postman app can also generate code snippets in a similar way. I covered this process in an earlier tutorial on Inspect the JSON from the response payload (p. 98). In Postman, after you configure your request, click the **Code** link (which appears below the Save button in the upper-right area).



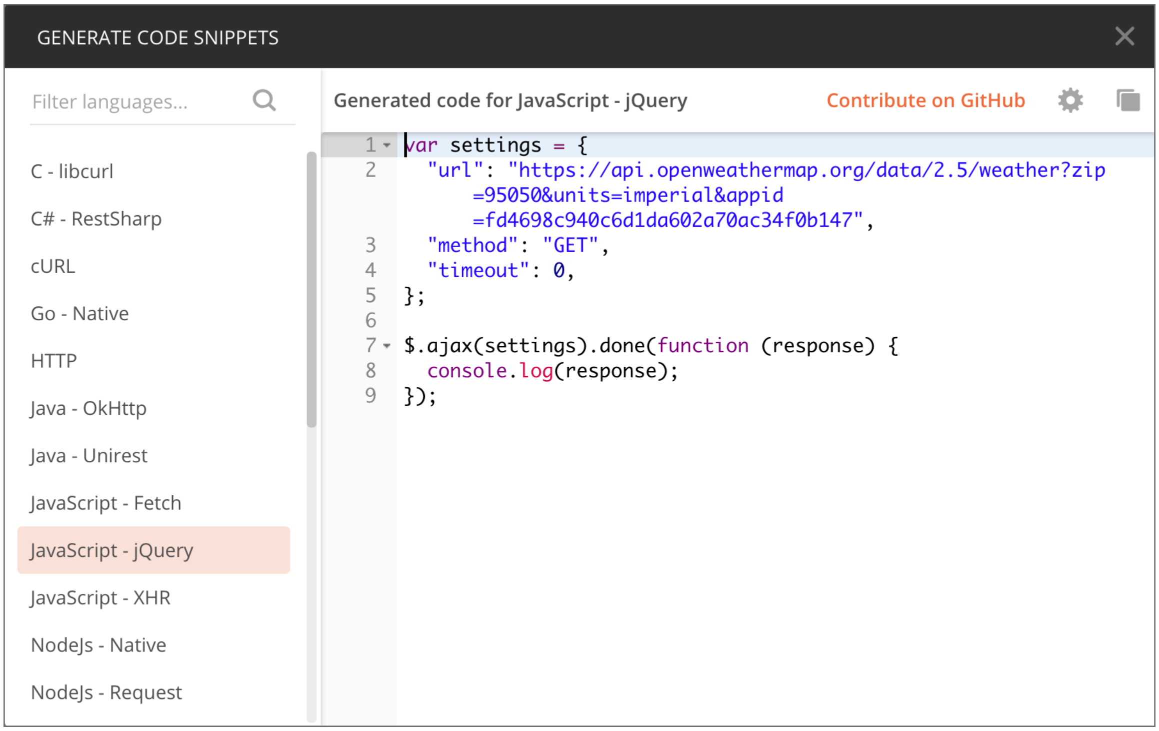
GET https://api.openweathermap.org/data/2.5/weather?lat=37.3565982&lon=-121.9689848&units=imperial&a...

	KEY	VALUE	DESCRIPTION
☑	lat	37.3565982	
☑	lon	-121.9689848	
☑	units	imperial	
☑	appid	fd4698c940c6d1da602a70ac34f0b147	
	Key	Value	Description

Status: 200 OK Time: 173 ms Size: 800 B

```
{
    "coord": {
        "lon": -121.97,
        "lat": 37.36
    },
    "weather": [
        {
            "id": 803,
            "main": "Clouds",
            "description": "broken clouds",
            "icon": "04d"
        }
    ],
    "base": "stations",
    "main": {
```

Then select the language you want, such as JavaScript > Jquery AJAX:



Generated code for JavaScript - jQuery

```
var settings = {
  "url": "https://api.openweathermap.org/data/2.5/weather?zip
      =95050&units=imperial&appid
      =fd4698c940c6d1da602a70ac34f0b147",
  "method": "GET",
  "timeout": 0,
};

$.ajax(settings).done(function (response) {
  console.log(response);
});
```

Although these code generators are probably helpful, they may or may not work for your API. Always review code samples with developers. In most cases, developers supply the code samples for the documentation, and technical writers briefly comment on the code samples.

(For an activity that involves using the generated jQuery code from Postman, see Inspect the JSON from the response payload (p. 98) and Access and print a specific JSON value (p. 103).)

SDKs provide tooling for APIs

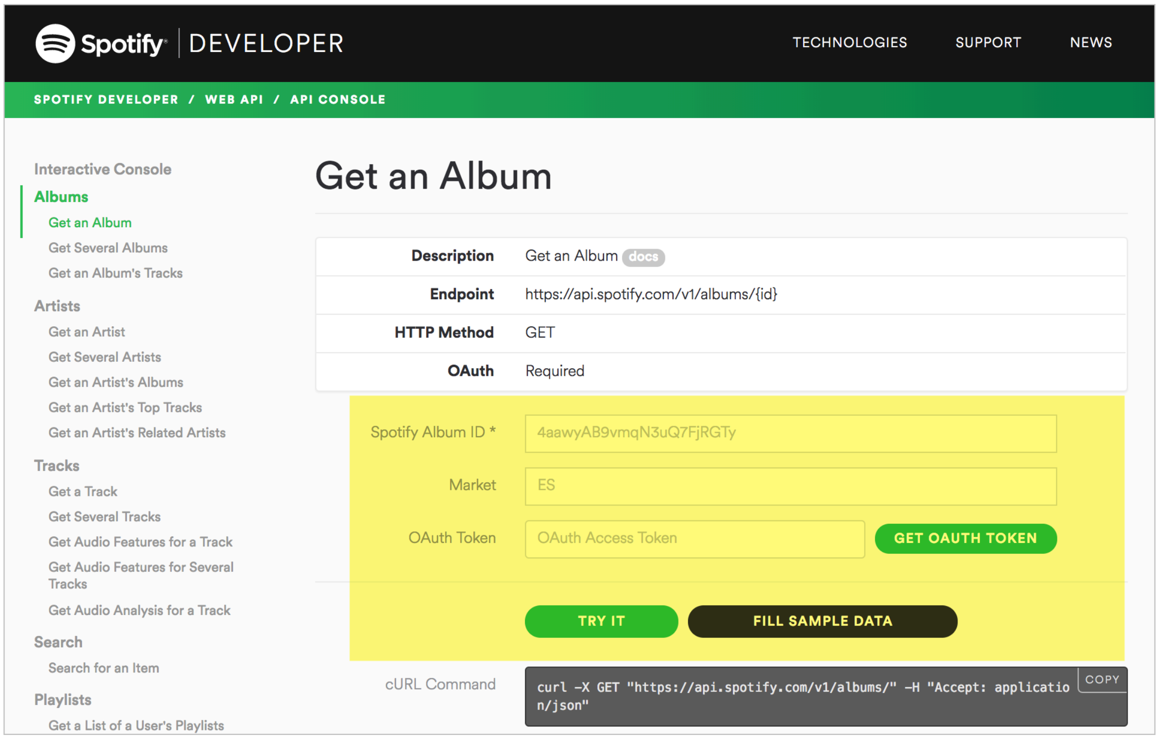
A lot of times, developers will create an SDK (software development kit) (p. 480) that accompanies a REST API. The SDK helps developers implement the API using specific tooling. While APIs are language agnostic, SDKs are language specific.

For example, at one company I worked at, we had both a REST API and a JavaScript SDK. Because JavaScript was the target language developers were working in, the company developed a JavaScript SDK to make it easier to work with REST using JavaScript. You could submit REST calls through the JavaScript SDK, passing a number of parameters relevant to web designers.

An SDK is any kind of tooling that makes it easier to work with your API. It's extremely common for a company to provide a language-agnostic REST API and then to develop an SDK that makes it easy to implement the API in the primary language they expect users to implement the API in. As such, peppering your sample requests with these small request snippets in other languages isn't that important, since the SDK provides an easier implementation. If you have an SDK, you'll want to make more detailed code samples (p. 464) showing how to use the SDK.

API explorers provide interactivity with your own data

Many APIs have an API explorer feature that lets users make actual requests directly from the documentation. For example, here's a typical reference page for Spotify's API docs:



Get an Album

Description	Get an Album docs
Endpoint	https://api.spotify.com/v1/albums/{id}
HTTP Method	GET
OAuth	Required

Spotify Album ID * 4aawyAB9vmqN3uQ7FjRGTy

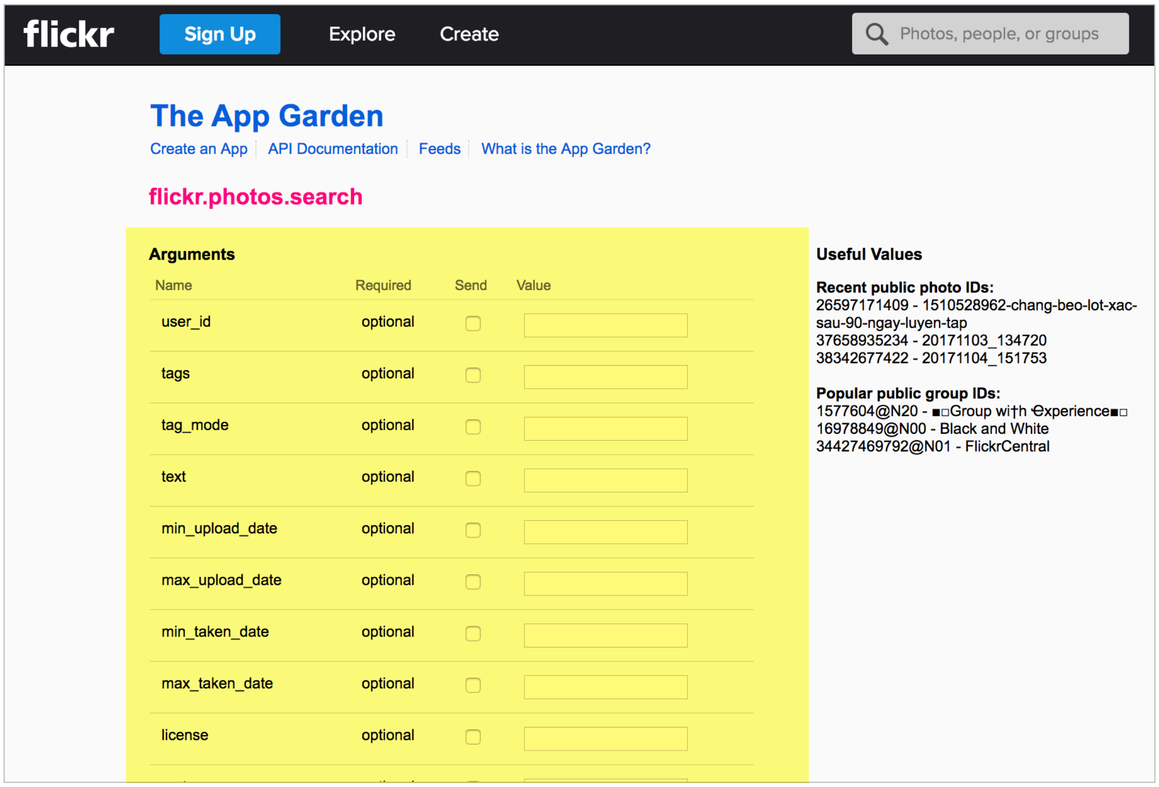
Market ES

OAuth Token OAuth Access Token

cURL Command

```
curl -X GET "https://api.spotify.com/v1/albums/" -H "Accept: applicatio
n/json"
```

Flickr's API docs also have a built-in API Explorer:



The App Garden

flickr.photos.search

Arguments

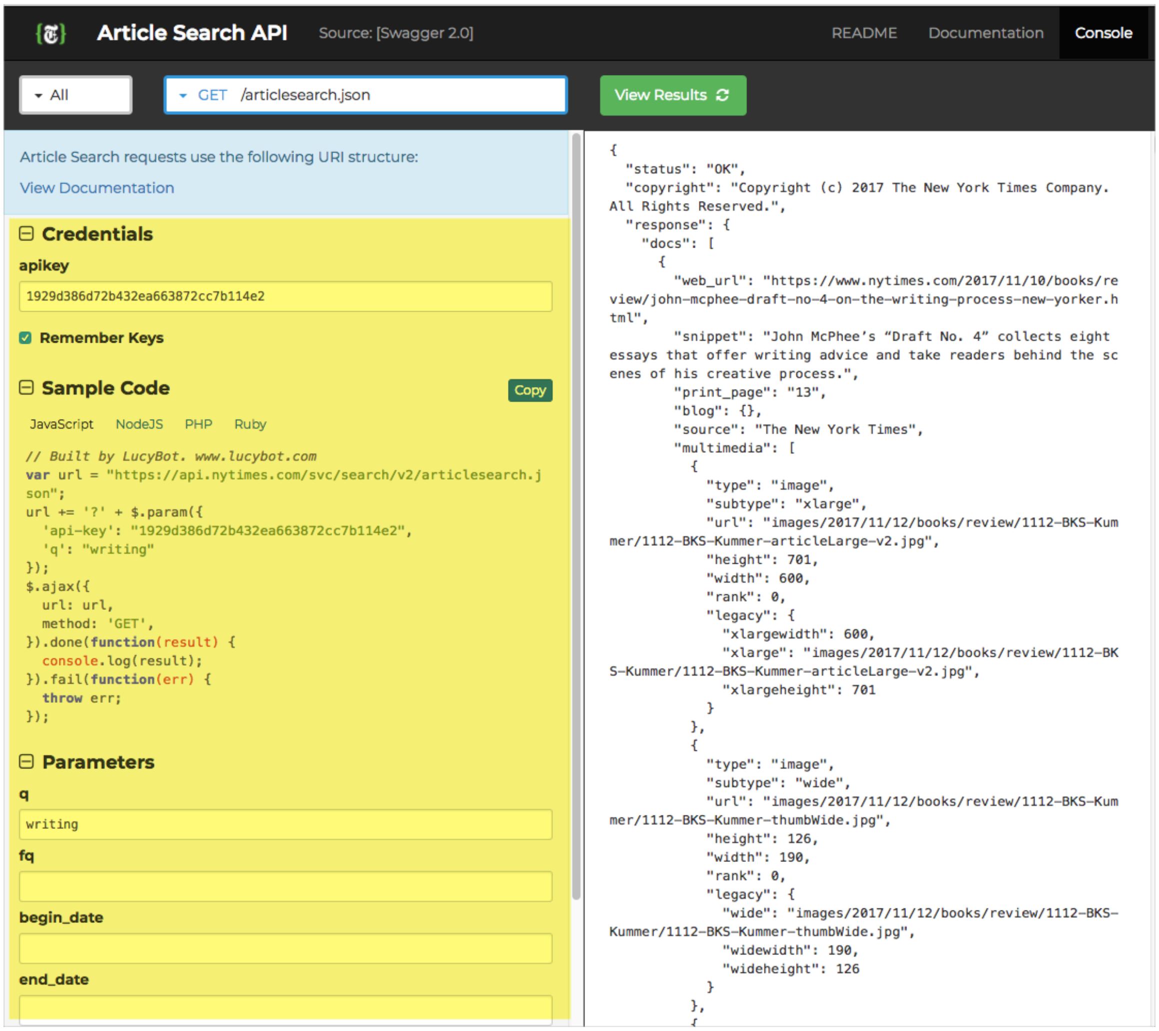
Name	Required	Send	Value
user_id	optional	☐	
tags	optional	☐	
tag_mode	optional	☐	
text	optional	☐	
min_upload_date	optional	☐	
max_upload_date	optional	☐	
min_taken_date	optional	☐	
max_taken_date	optional	☐	
license	optional	☐	

Useful Values

Recent public photo IDs:
26597171409 - 1510528962-chang-beo-lot-xac-sau-90-ngay-luyen-tap
37658935234 - 20171103_134720
38342677422 - 20171104_151753

Popular public group IDs:
1577604@N20 - ■□Group wi†h Ɛxperience■□
16978849@N00 - Black and White
34427469792@N01 - FlickrCentral

As does the New York Times API:



Article Search requests use the following URI structure:

apikey: 1929d386d72b432ea663872cc7b114e2

☑ Remember Keys

```
// Built by LucyBot. www.lucybot.com
var url = "https://api.nytimes.com/svc/search/v2/articlesearch.j
son";
url += '?' + $.param({
  'api-key': "1929d386d72b432ea663872cc7b114e2",
  'q': "writing"
});
$.ajax({
  url: url,
  method: 'GET',
}).done(function(result) {
  console.log(result);
}).fail(function(err) {
  throw err;
});
```

q: writing

fq:

begin_date:

end_date:

```
{
  "status": "OK",
  "copyright": "Copyright (c) 2017 The New York Times Company.
All Rights Reserved.",
  "response": {
    "docs": [
      {
        "web_url": "https://www.nytimes.com/2017/11/10/books/re
view/john-mcphee-draft-no-4-on-the-writing-process-new-yorker.h
tml",
        "snippet": "John McPhee’s “Draft No. 4” collects eight
essays that offer writing advice and take readers behind the sc
enes of his creative process.",
        "print_page": "13",
        "blog": {},
        "source": "The New York Times",
        "multimedia": [
          {
            "type": "image",
            "subtype": "xlarge",
            "url": "images/2017/11/12/books/review/1112-BKS-Kum
mer/1112-BKS-Kummer-articleLarge-v2.jpg",
            "height": 701,
            "width": 600,
            "rank": 0,
            "legacy": {
              "xlargewidth": 600,
              "xlarge": "images/2017/11/12/books/review/1112-BK
S-Kummer/1112-BKS-Kummer-articleLarge-v2.jpg",
              "xlargeheight": 701
            }
          },
          {
            "type": "image",
            "subtype": "wide",
            "url": "images/2017/11/12/books/review/1112-BKS-Kum
mer/1112-BKS-Kummer-thumbWide.jpg",
            "height": 126,
            "width": 190,
            "rank": 0,
            "legacy": {
              "wide": "images/2017/11/12/books/review/1112-BKS-
Kummer/1112-BKS-Kummer-thumbWide.jpg",
              "widewidth": 190,
              "wideheight": 126
            }
          },
          {
```

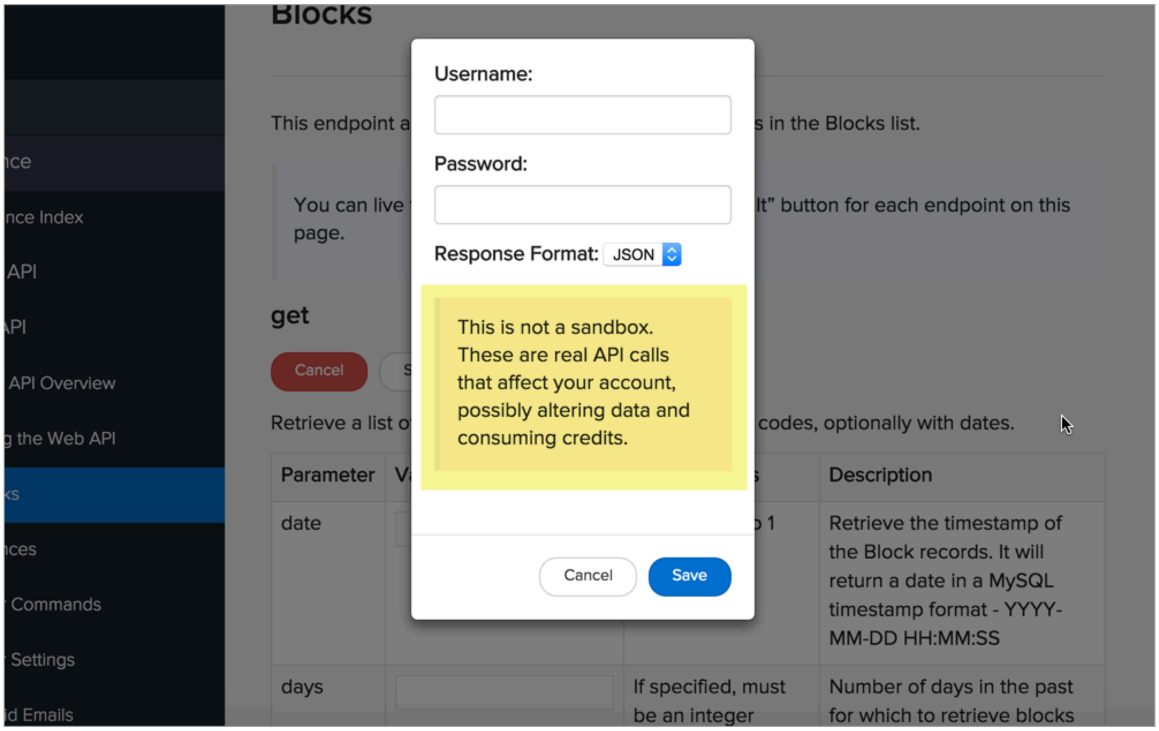
The API Explorer lets you insert your own values, your own API key, and other parameters into a request so you can see the responses directly in the API Explorer. Being able to see your own data makes the response more real and immediate.

However, if you don’t have the right data in your system, using your own API key may not show you the full response that’s possible. It works best when the resources involve public information and the requests are GET requests.

API Explorers can be dangerous in the hands of users

Although interactivity is powerful, API Explorers can be a dangerous addition to your site. What if a novice user who is trying out a DELETE method accidentally removes data? How do you later remove the test data added by POST or PUT methods?

It’s one thing to allow GET methods, but if you include other methods, users could inadvertently corrupt their data. In Sendgrid’s API, they include a warning message to users before testing out calls with their API Explorer:



This endpoint a … s in the Blocks list.

You can live … lt" button for each endpoint on this page.

Username:

Password:

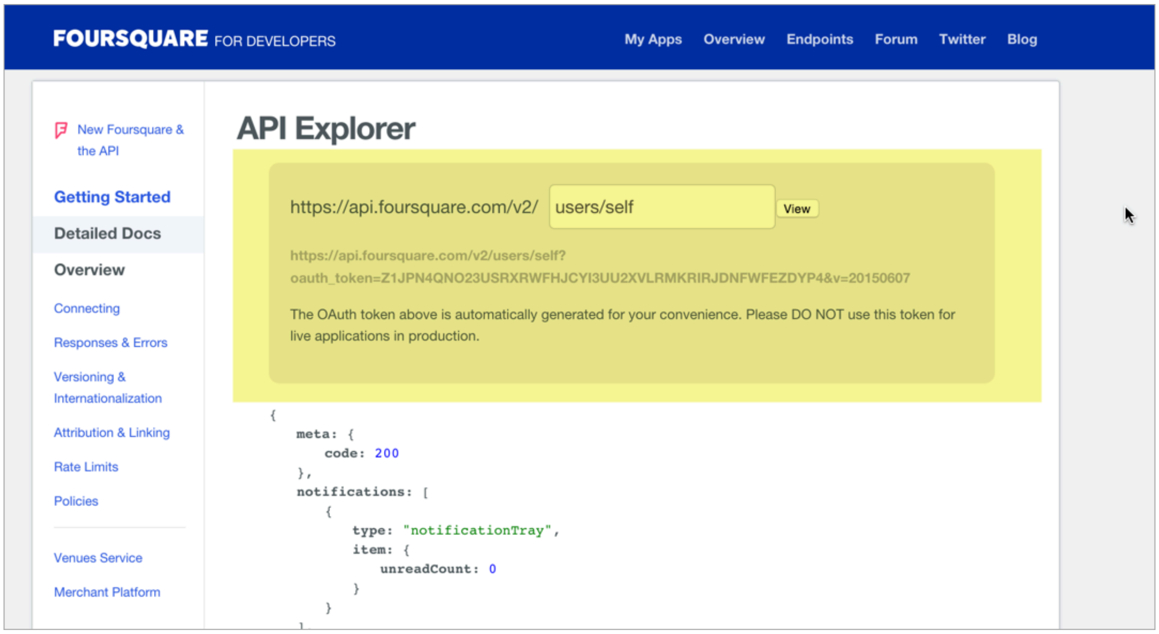
Response Format: JSON

This is not a sandbox. These are real API calls that affect your account, possibly altering data and consuming credits.

Retrieve a list o … codes, optionally with dates.

Parameter	V		Description
date		1	Retrieve the timestamp of the Block records. It will return a date in a MySQL timestamp format - YYYY-MM-DD HH:MM:SS
days		If specified, must be an integer	Number of days in the past for which to retrieve blocks

Foursquare's API docs used to have a built-in API explorer in the previous version of their docs (shown below), but they have since removed it. I'm not sure why — maybe they ran into some of these issues.



API Explorer

https://api.foursquare.com/v2/ users/self

https://api.foursquare.com/v2/users/self?oauth_token=Z1JPN4QNO23USRXRWFHJCYI3UU2XVLRMKRIRJDNFWFEZDYP4&v=20150607

The OAuth token above is automatically generated for your convenience. Please DO NOT use this token for live applications in production.

```
{
   meta: {
       code: 200
   },
   notifications: [
       {
           type: "notificationTray",
           item: {
               unreadCount: 0
           }
       }
```

As far as integrating custom API Explorer tooling, this is a task that should be relatively easy for developers. All the API Explorer does is map values from a field to an API call and return the response to the same interface. In other words, the API plumbing is all there — you just need a little JavaScript and front-end skills to make it happen.

However, you don't have to build your own tooling. Existing tools such as Swagger UI (which parses an OpenAPI specification document (p. 272)) and Readme.com (which allows you to enter the details manually or from an OpenAPI specification) can integrate API Explorer functionality directly into your documentation.

For a tutorial on how to create your own API explorer functionality, see the Swagger UI tutorial (p. 218).

Request example for the surfreport endpoint

Let's return to the `surfreport/{beachId}` endpoint in our sample scenario (p. 115) and create a request example for it. Here's my approach:

Sample request

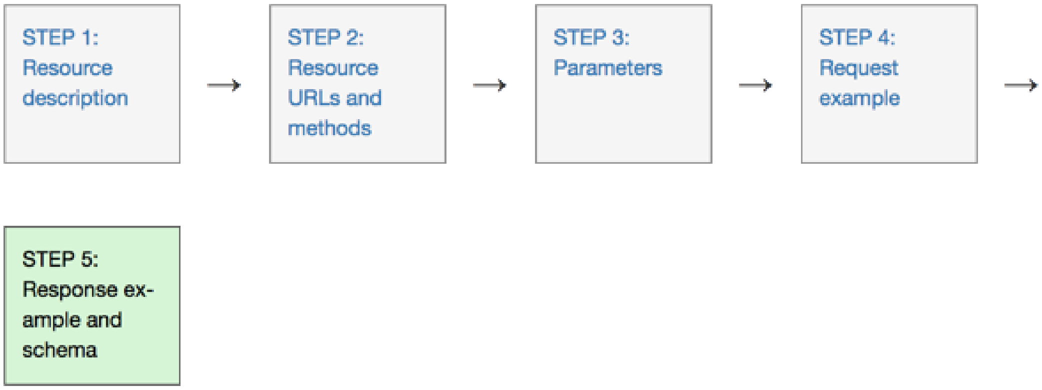
```
curl -I -X GET "https://api.openweathermap.org/data/2.5/surfreport?zip=9505
0&appid=APIKEY&units=imperial&days=2"
```

(In the above code, replace `APIKEY` with your actual API key.)

Next steps

Now that we've created a sample request, the next steps naturally follow — include a sample response that corresponds with the same request. We'll also document the model or schema of the response in general. Go to Step 5: Response example and schema (API reference tutorial) (p. 154).

Step 5: Response example and schema (API reference tutorial)

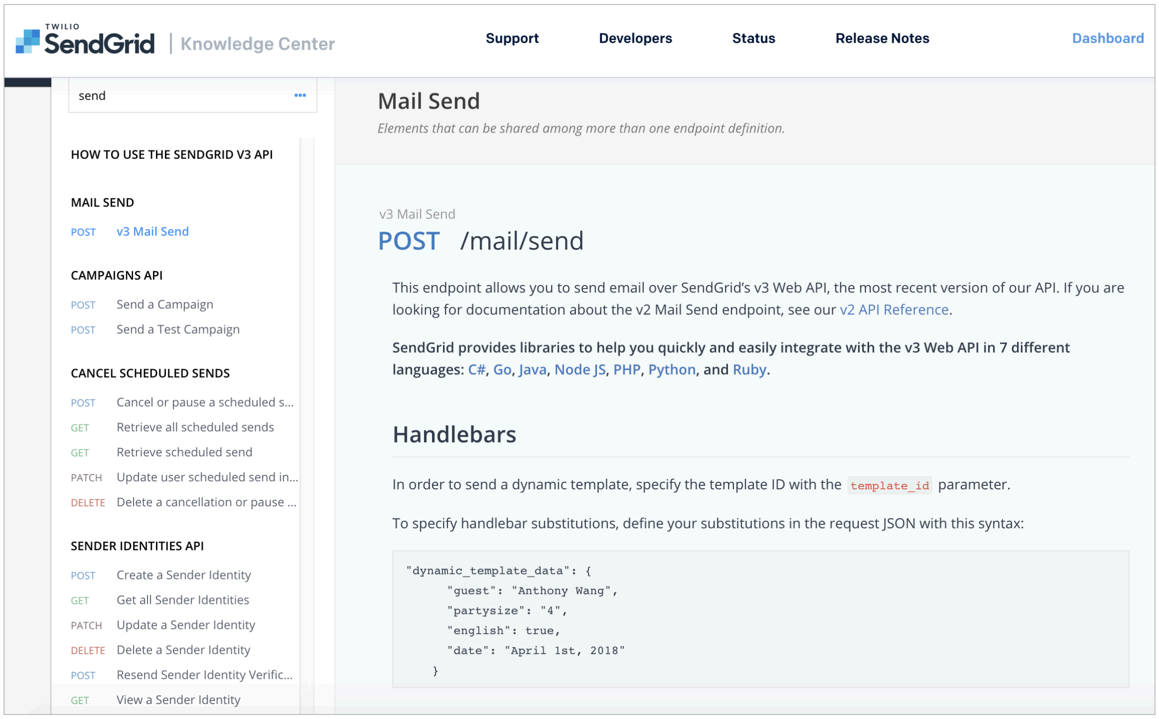

STEP 1: Resource description
→
STEP 2: Resource URLs and methods
→
STEP 3: Parameters
→
STEP 4: Request example
→
STEP 5: Response example and schema

The response example shows a sample response from the request example; the response schema defines all possible elements in the response. The response example is not comprehensive of all parameter configurations or operations, but it should correspond with the parameters passed in the request example. The response lets developers know if the resource contains the information they want, the format, and how that information is structured and labeled.

The description of the response is known as the response schema. The response schema documents the response in a more comprehensive, general way, listing each property that could possibly be returned, what each property contains, the data format of the values, the structure, and other details.

Examples of response examples and schemas

The following is a sample response from the SendGrid API. Their documentation provides a tabbed display with an **Example** on one tab:


TWILIO
SendGrid | Knowledge Center
Support
Developers
Status
Release Notes
Dashboard
send
HOW TO USE THE SENDGRID V3 API
MAIL SEND
POST v3 Mail Send
CAMPAIGNS API
POST Send a Campaign
POST Send a Test Campaign
CANCEL SCHEDULED SENDS
POST Cancel or pause a scheduled s...
GET Retrieve all scheduled sends
GET Retrieve scheduled send
PATCH Update user scheduled send in...
DELETE Delete a cancellation or pause ...
SENDER IDENTITIES API
POST Create a Sender Identity
GET Get all Sender Identities
PATCH Update a Sender Identity
DELETE Delete a Sender Identity
POST Resend Sender Identity Verific...
GET View a Sender Identity

Mail Send

Elements that can be shared among more than one endpoint definition.

v3 Mail Send

POST /mail/send

This endpoint allows you to send email over SendGrid's v3 Web API, the most recent version of our API. If you are looking for documentation about the v2 Mail Send endpoint, see our v2 API Reference.

SendGrid provides libraries to help you quickly and easily integrate with the v3 Web API in 7 different languages: C#, Go, Java, Node JS, PHP, Python, and Ruby.

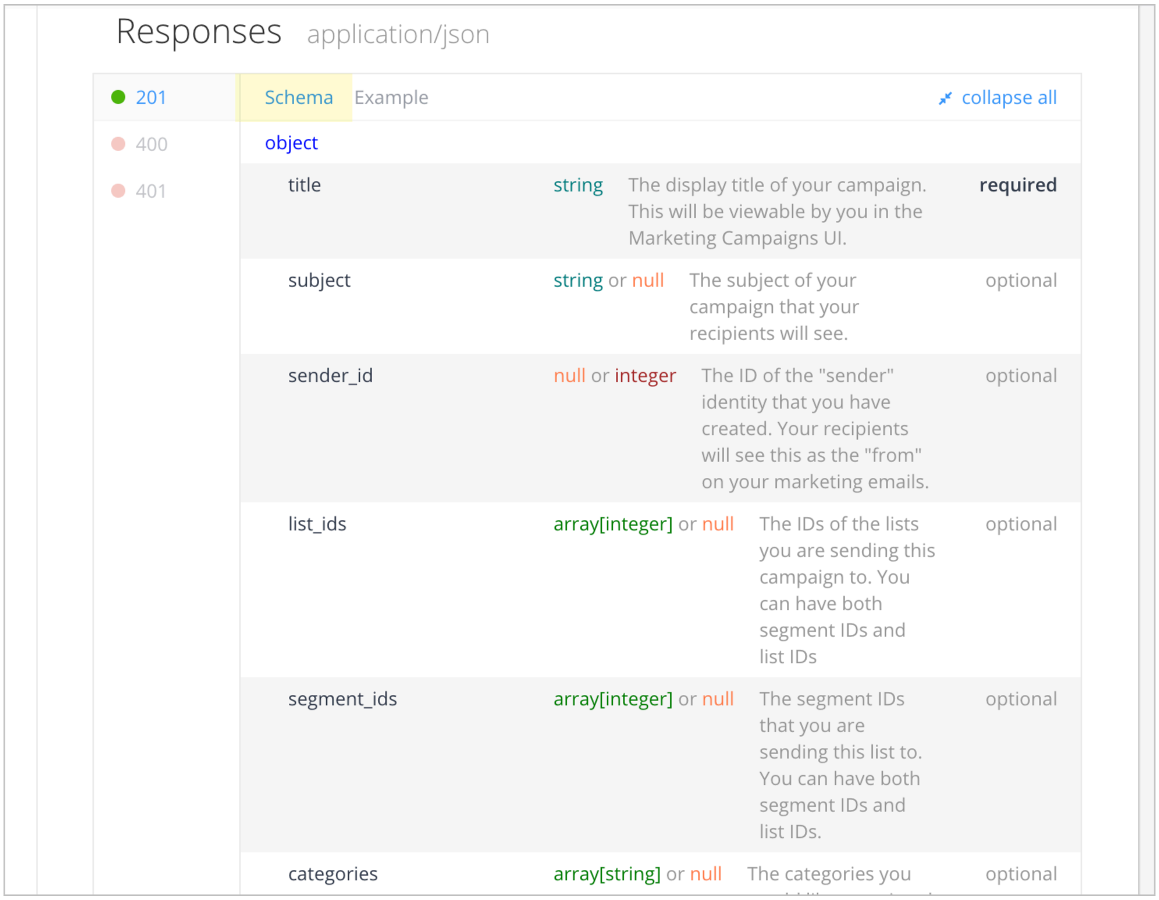
Handlebars

In order to send a dynamic template, specify the template ID with the `template_id` parameter.

To specify handlebar substitutions, define your substitutions in the request JSON with this syntax:

```
"dynamic_template_data": {
      "guest": "Anthony Wang",
      "partysize": "4",
      "english": true,
      "date": "April 1st, 2018"
    }
```

And the response **Schema** on another tab:

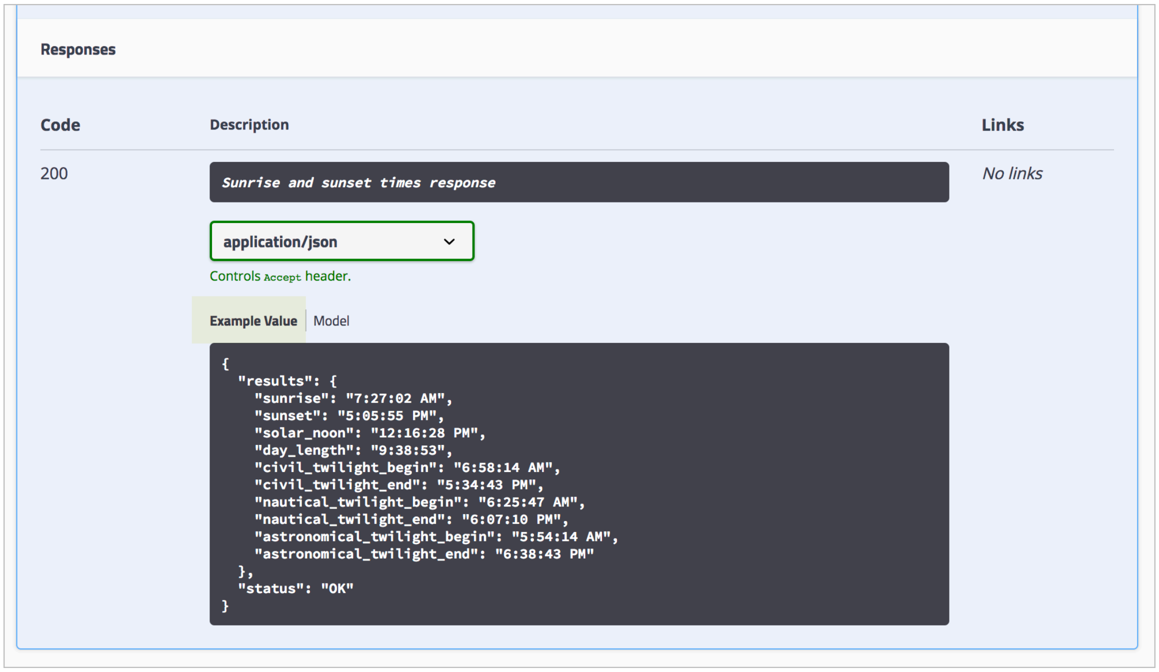


object

title	string	The display title of your campaign. This will be viewable by you in the Marketing Campaigns UI.	**required**
subject	string or null	The subject of your campaign that your recipients will see.	optional
sender_id	null or integer	The ID of the "sender" identity that you have created. Your recipients will see this as the "from" on your marketing emails.	optional
list_ids	array[integer] or null	The IDs of the lists you are sending this campaign to. You can have both segment IDs and list IDs	optional
segment_ids	array[integer] or null	The segment IDs that you are sending this list to. You can have both segment IDs and list IDs.	optional
categories	array[string] or null	The categories you	optional

The definition of the response is called the *schema* or *model* (the terms are used synonymously) and aligns with the JSON schema language and descriptions ↗. What works particularly well with the SendGrid example is the use of expand/collapse tags to mirror the same structure as the example, with objects at different levels.

Swagger UI also provides both an example value and a schema or model. For example, in the sample Sunrise and Sunset Times API doc that I used for the SwaggerUI activity ↗ (which comes later in the course), you can see a distinction between the response example and the response schema. Here's the **Example Value**:



Code	Description	Links
200	*Sunrise and sunset times response*	*No links*

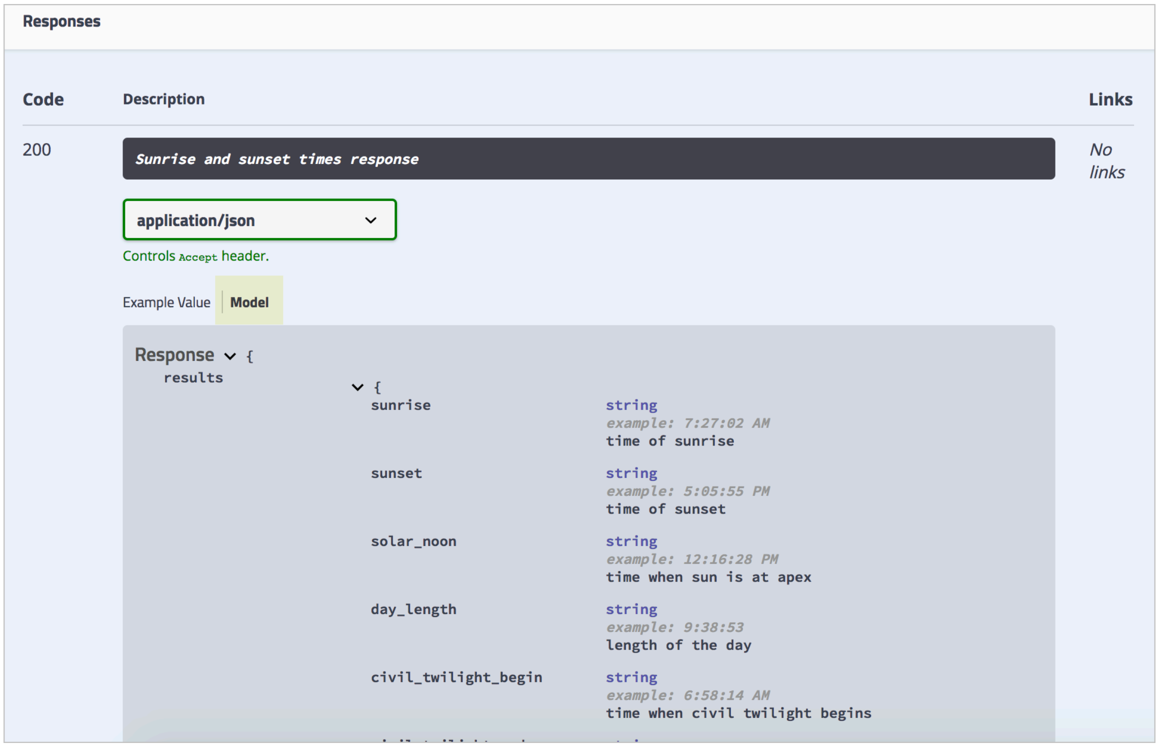
application/json

Controls `Accept` header.

```
{
  "results": {
    "sunrise": "7:27:02 AM",
    "sunset": "5:05:55 PM",
    "solar_noon": "12:16:28 PM",
    "day_length": "9:38:53",
    "civil_twilight_begin": "6:58:14 AM",
    "civil_twilight_end": "5:34:43 PM",
    "nautical_twilight_begin": "6:25:47 AM",
    "nautical_twilight_end": "6:07:10 PM",
    "astronomical_twilight_begin": "5:54:14 AM",
    "astronomical_twilight_end": "6:38:43 PM"
  },
  "status": "OK"
}
```

The example response should correspond with the example request. Just as the request example might only include a subset of all possible parameters, the response example might also be a subset of all possible returned information.

However, the *response schema* is comprehensive of all possible properties returned in the response. This is why you need both a response example and a response schema. Here's the response schema for the Sunrise and Sunset Times API:



```
Response ⌄ {
   results                ⌄ {
                            sunrise                  string
                                                     example: 7:27:02 AM
                                                     time of sunrise

                            sunset                   string
                                                     example: 5:05:55 PM
                                                     time of sunset

                            solar_noon               string
                                                     example: 12:16:28 PM
                                                     time when sun is at apex

                            day_length               string
                                                     example: 9:38:53
                                                     length of the day

                            civil_twilight_begin     string
                                                     example: 6:58:14 AM
                                                     time when civil twilight begins
```

The schema or model provides the following:

- Description of each property
- Definition of the data type for each property
- Whether each property is required or optional

If the header information is important to include in the response example (because it provides unique information other than standard status codes (p. 402)), you can include it as well.

Do you need to define the response?

Some API documentation omits the response schema because the responses might seem self-evident or intuitive. In Twitter's API, the responses aren't explained (you can see an example here).

However, most documentation would be better off with the response described, especially if the properties are abbreviated or cryptic. Developers sometimes abbreviate the responses to increase performance by reducing the amount of text sent. In one endpoint I documented, the response included about 20 different two-letter abbreviations. I spent days tracking down what each abbreviation meant, and found that many developers who worked on the API didn't even know what many of the responses meant.

Use realistic values in the example response

In the example response, the values should be realistic without being real. If developers give you a sample response, make sure the values are reasonable and not so fake they're distracting (such as users consisting of comic book character names).

Also, the sample response should not contain real customer data. If you get a sample response from an engineer, and the data looks real, make sure it's not just from a cloned production database, which is commonly done. Developers may not realize that the data needs to be fictitious but representative, and scraping a production database may be the easiest approach for them.

Format the JSON and use code syntax highlighting

Use proper JSON formatting for the response. A tool such as JSON Formatter and Validator can make sure the spacing is correct.

If you can add syntax highlighting as well, definitely do it. If you're using a static site generator such as Jekyll (p. 637) or markdown syntax with GitHub (p. 569), you can probably use the Rouge built-in syntax highlighter. Other static site generators might use Pygments or similar extensions.

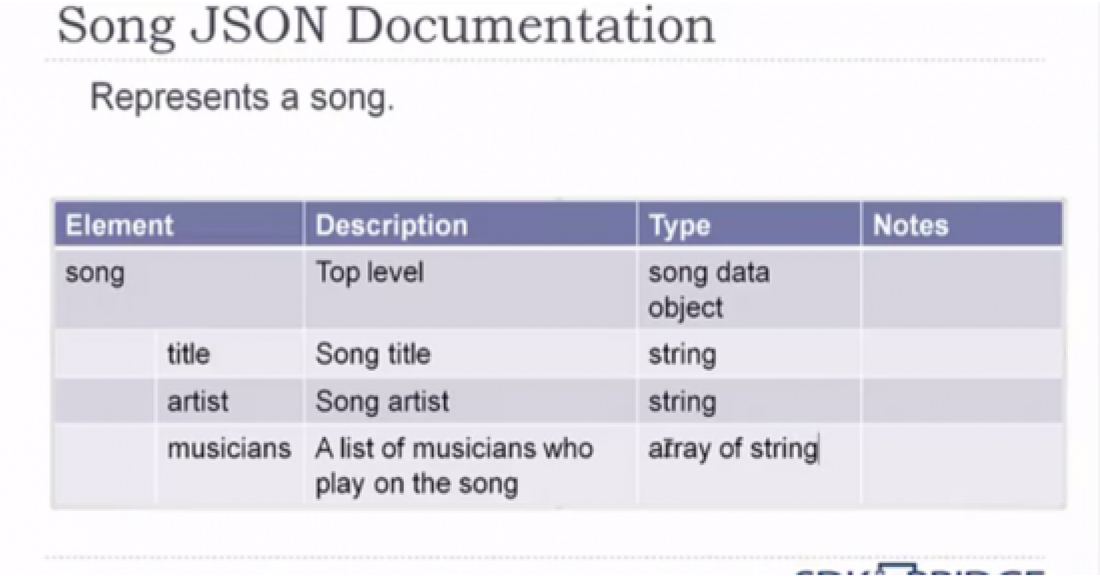
Rouge and Pygments rely on "lexers" to indicate how the code should be highlighted. For example, some common lexers are `java`, `json`, `html`, `xml`, `cpp`, `dotnet`, and `javascript`.

If you don't have any syntax highlighters to integrate directly into your authoring tool, you can use an online syntax highlighter such as tohtml.com/jScript/. However, manually pasting code into these editors will be tedious and probably unsustainable.

Strategies for documenting nested objects

Many times, the response contains nested objects (objects within objects) or has repeating elements. Formatting the documentation for the response schema is one of the more challenging aspects of API reference documentation.

Tables are most commonly used. In Peter Gruenbaum's API tech writing course on Udemy, Gruenbaum represents the nested objects using tables with various columns:



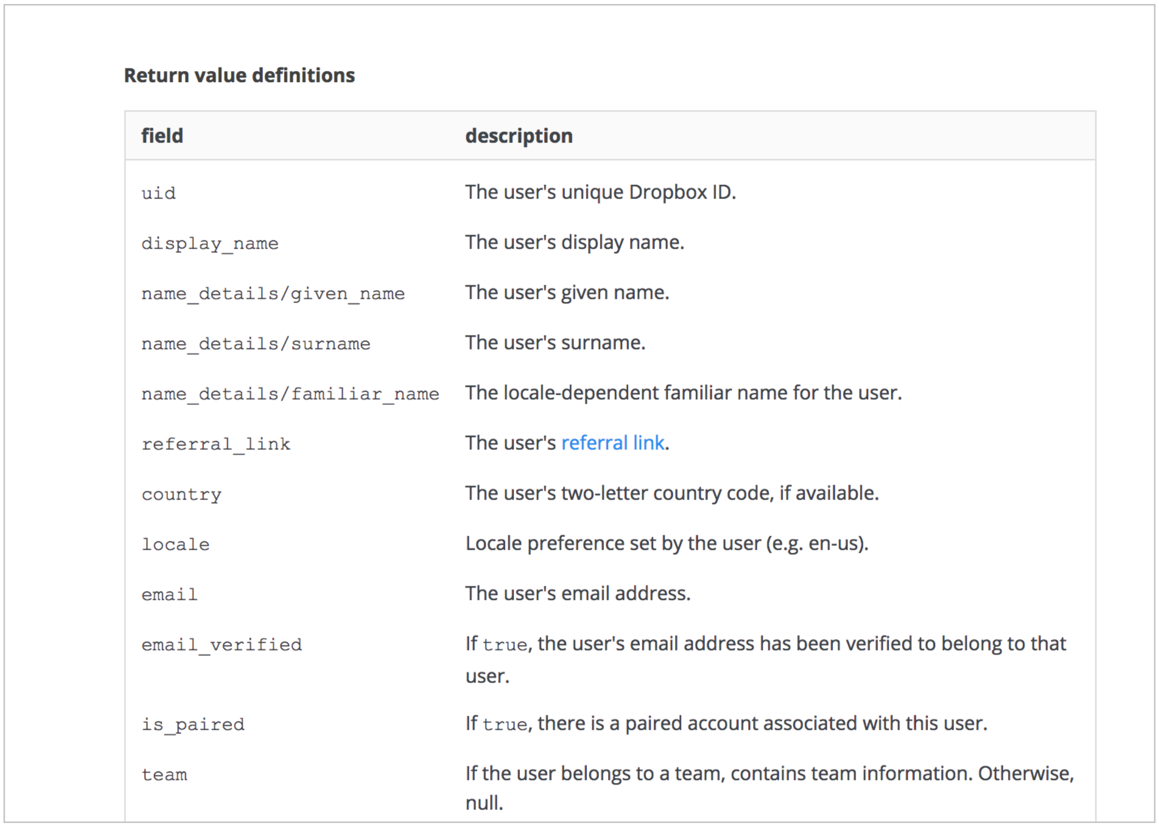
Song JSON Documentation

Represents a song.

Element		Description	Type	Notes
song		Top level	song data object	
	title	Song title	string	
	artist	Song artist	string	
	musicians	A list of musicians who play on the song	array of string	

Gruenbaum's use of tables is mostly to reduce the emphasis on tools and place it more on the content.

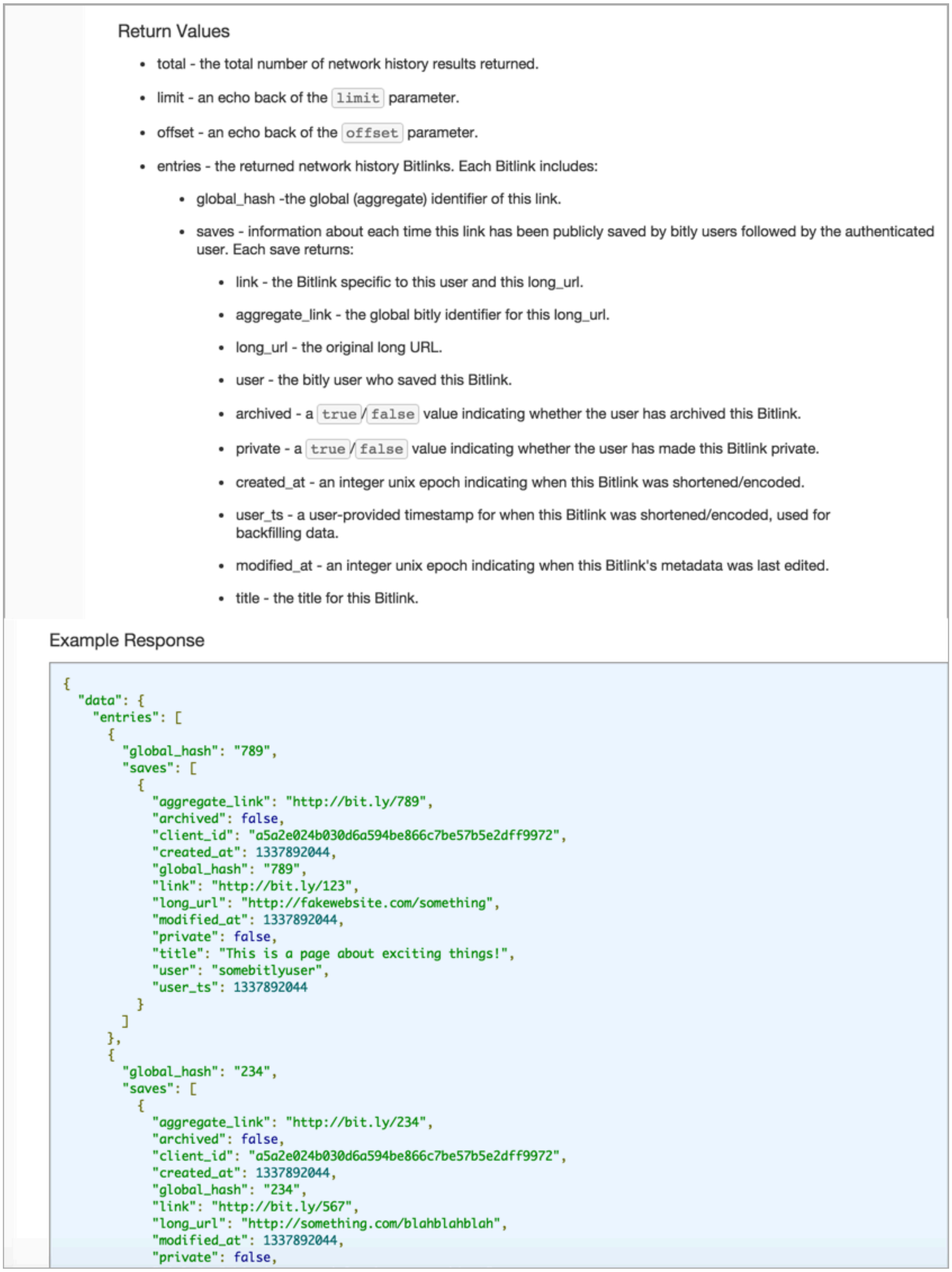
The Dropbox API represents the nesting with a slash. For example, `name_details/`, `team/`, and `quota_info` indicate the multiple object levels.



Return value definitions

field	description
`uid`	The user's unique Dropbox ID.
`display_name`	The user's display name.
`name_details/given_name`	The user's given name.
`name_details/surname`	The user's surname.
`name_details/familiar_name`	The locale-dependent familiar name for the user.
`referral_link`	The user's referral link.
`country`	The user's two-letter country code, if available.
`locale`	Locale preference set by the user (e.g. en-us).
`email`	The user's email address.
`email_verified`	If `true`, the user's email address has been verified to belong to that user.
`is_paired`	If `true`, there is a paired account associated with this user.
`team`	If the user belongs to a team, contains team information. Otherwise, null.

Other APIs will nest the response definitions to imitate the JSON structure. Here’s an example from bit.ly’s API:



Return Values

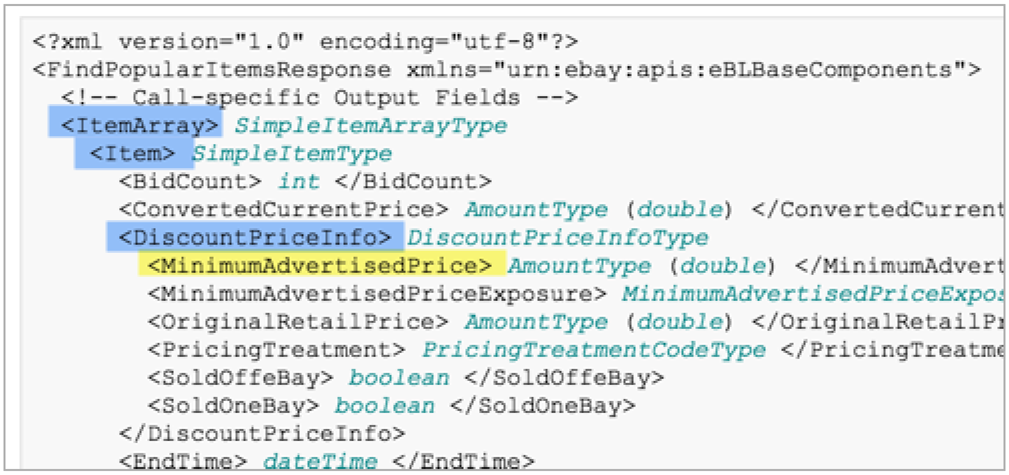
- total - the total number of network history results returned.
- limit - an echo back of the `limit` parameter.
- offset - an echo back of the `offset` parameter.
- entries - the returned network history Bitlinks. Each Bitlink includes:
  - global_hash -the global (aggregate) identifier of this link.
  - saves - information about each time this link has been publicly saved by bitly users followed by the authenticated user. Each save returns:
    - link - the Bitlink specific to this user and this long_url.
    - aggregate_link - the global bitly identifier for this long_url.
    - long_url - the original long URL.
    - user - the bitly user who saved this Bitlink.
    - archived - a `true`/`false` value indicating whether the user has archived this Bitlink.
    - private - a `true`/`false` value indicating whether the user has made this Bitlink private.
    - created_at - an integer unix epoch indicating when this Bitlink was shortened/encoded.
    - user_ts - a user-provided timestamp for when this Bitlink was shortened/encoded, used for backfilling data.
    - modified_at - an integer unix epoch indicating when this Bitlink's metadata was last edited.
    - title - the title for this Bitlink.

Example Response

```
{
  "data": {
    "entries": [
      {
        "global_hash": "789",
        "saves": [
          {
            "aggregate_link": "http://bit.ly/789",
            "archived": false,
            "client_id": "a5a2e024b030d6a594be866c7be57b5e2dff9972",
            "created_at": 1337892044,
            "global_hash": "789",
            "link": "http://bit.ly/123",
            "long_url": "http://fakewebsite.com/something",
            "modified_at": 1337892044,
            "private": false,
            "title": "This is a page about exciting things!",
            "user": "somebitlyuser",
            "user_ts": 1337892044
          }
        ]
      },
      {
        "global_hash": "234",
        "saves": [
          {
            "aggregate_link": "http://bit.ly/234",
            "archived": false,
            "client_id": "a5a2e024b030d6a594be866c7be57b5e2dff9972",
            "created_at": 1337892044,
            "global_hash": "234",
            "link": "http://bit.ly/567",
            "long_url": "http://something.com/blahblahblah",
            "modified_at": 1337892044,
            "private": false,
```

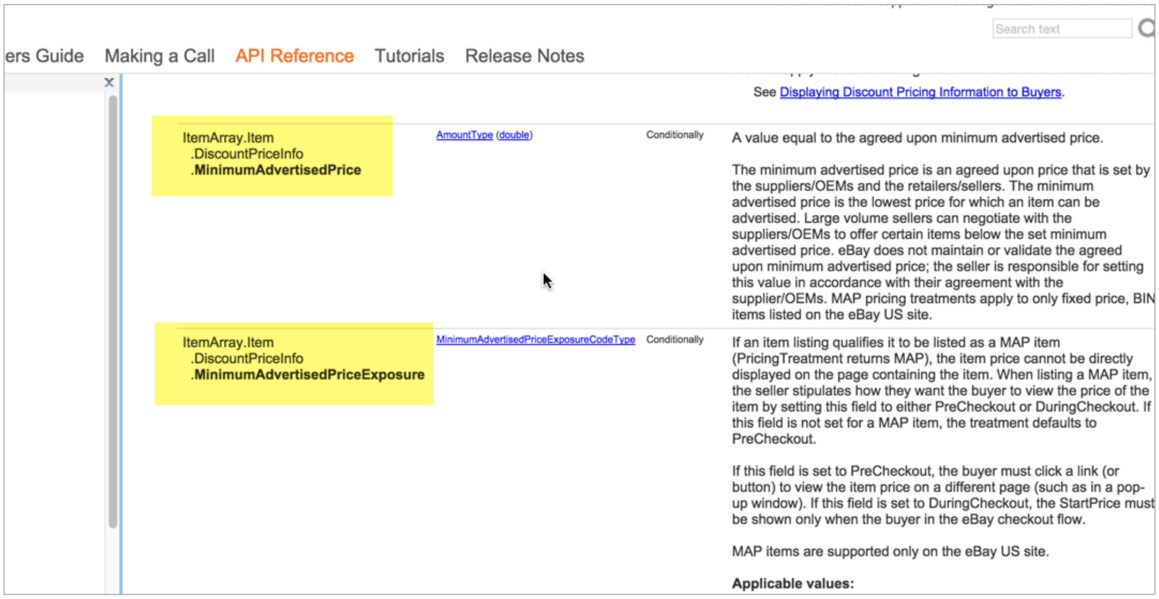
Multiple levels of bullets is usually an eyesore, but here it serves a purpose that works well without requiring sophisticated styling.

eBay's approach is a little more unique. In this case, `MinimumAdvertisedPrice` is nested inside `DiscountPriceInfo`, which is nested in `Item`, which is nested in `ItemArray`. (Note also that this response is in XML instead of JSON.)



```
<?xml version="1.0" encoding="utf-8"?>
<FindPopularItemsResponse xmlns="urn:ebay:apis:eBLBaseComponents">
  <!-- Call-specific Output Fields -->
  <ItemArray> SimpleItemArrayType
    <Item> SimpleItemType
      <BidCount> int </BidCount>
      <ConvertedCurrentPrice> AmountType (double) </ConvertedCurrent
      <DiscountPriceInfo> DiscountPriceInfoType
        <MinimumAdvertisedPrice> AmountType (double) </MinimumAdvert
        <MinimumAdvertisedPriceExposure> MinimumAdvertisedPriceExpos
        <OriginalRetailPrice> AmountType (double) </OriginalRetailPr
        <PricingTreatment> PricingTreatmentCodeType </PricingTreatme
        <SoldOffeBay> boolean </SoldOffeBay>
        <SoldOneBay> boolean </SoldOneBay>
      </DiscountPriceInfo>
      <EndTime> dateTime </EndTime>
```

Here's the response documentation:



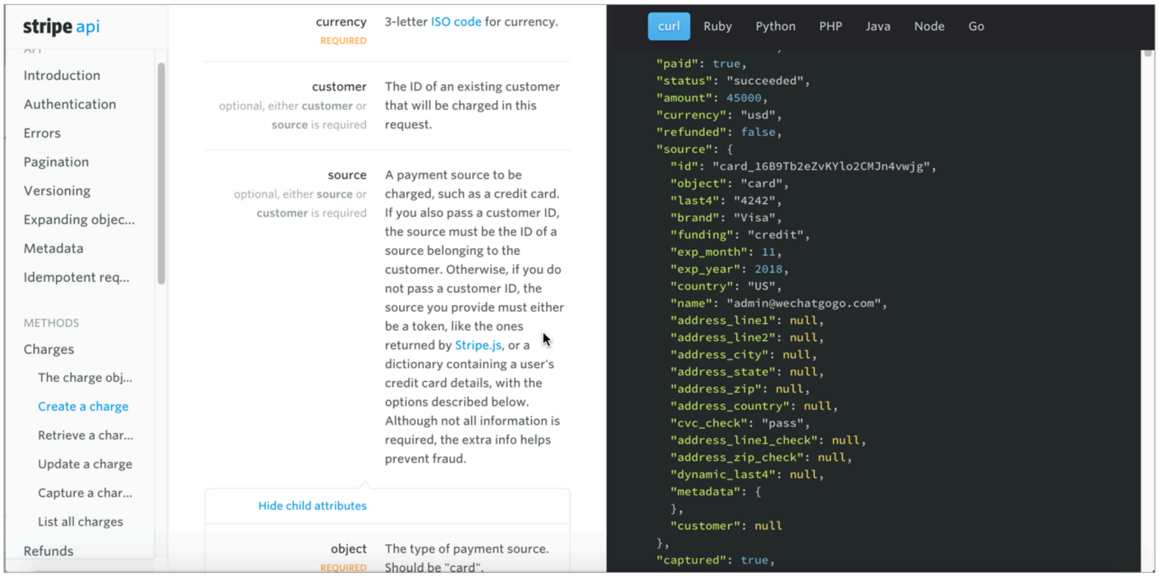
See Displaying Discount Pricing Information to Buyers.

ItemArray.Item .DiscountPriceInfo **.MinimumAdvertisedPrice**	AmountType (double)	Conditionally	A value equal to the agreed upon minimum advertised price. The minimum advertised price is an agreed upon price that is set by the suppliers/OEMs and the retailers/sellers. The minimum advertised price is the lowest price for which an item can be advertised. Large volume sellers can negotiate with the suppliers/OEMs to offer certain items below the set minimum advertised price. eBay does not maintain or validate the agreed upon minimum advertised price; the seller is responsible for setting this value in accordance with their agreement with the supplier/OEMs. MAP pricing treatments apply to only fixed price, BIN items listed on the eBay US site.
ItemArray.Item .DiscountPriceInfo **.MinimumAdvertisedPriceExposure**	MinimumAdvertisedPriceExposureCodeType	Conditionally	If an item listing qualifies it to be listed as a MAP item (PricingTreatment returns MAP), the item price cannot be directly displayed on the page containing the item. When listing a MAP item, the seller stipulates how they want the buyer to view the price of the item by setting this field to either PreCheckout or DuringCheckout. If this field is not set for a MAP item, the treatment defaults to PreCheckout. If this field is set to PreCheckout, the buyer must click a link (or button) to view the item price on a different page (such as in a pop-up window). If this field is set to DuringCheckout, the StartPrice must be shown only when the buyer in the eBay checkout flow. MAP items are supported only on the eBay US site. **Applicable values:**

It's also interesting how much detail eBay includes for each item. Whereas the Twitter writers appear to omit descriptions, the eBay authors write small novels describing each item in the response.

Three-column designs

Some APIs put the response in a right column so you can see it while also looking at the resource description and parameters. Stripe's API made this three-column design popular:



currency REQUIRED	3-letter ISO code for currency.
customer optional, either customer or source is required	The ID of an existing customer that will be charged in this request.
source optional, either source or customer is required	A payment source to be charged, such as a credit card. If you also pass a customer ID, the source must be the ID of a source belonging to the customer. Otherwise, if you do not pass a customer ID, the source you provide must either be a token, like the ones returned by Stripe.js, or a dictionary containing a user's credit card details, with the options described below. Although not all information is required, the extra info helps prevent fraud.
object REQUIRED	The type of payment source. Should be "card".

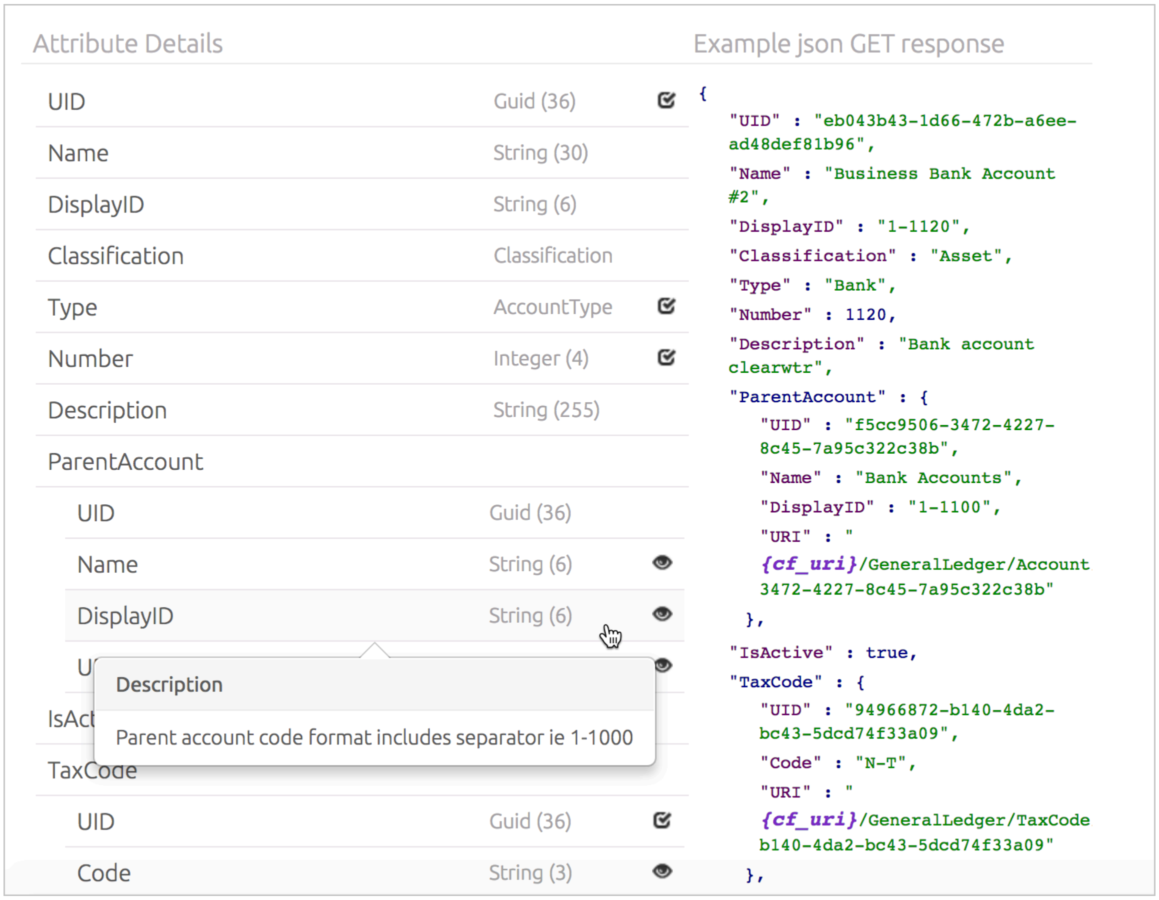
```
  "paid": true,
  "status": "succeeded",
  "amount": 45000,
  "currency": "usd",
  "refunded": false,
  "source": {
    "id": "card_16B9Tb2eZvKYlo2CMJn4vwjg",
    "object": "card",
    "last4": "4242",
    "brand": "Visa",
    "funding": "credit",
    "exp_month": 11,
    "exp_year": 2018,
    "country": "US",
    "name": "admin@wechatgogo.com",
    "address_line1": null,
    "address_line2": null,
    "address_city": null,
    "address_state": null,
    "address_zip": null,
    "address_country": null,
    "cvc_check": "pass",
    "address_line1_check": null,
    "address_zip_check": null,
    "dynamic_last4": null,
    "metadata": {
    },
    "customer": null
  },
  "captured": true,
```

Stripe's design juxtaposes the sample response in a right side pane with the response schema in the main window. The idea is that you can see both at the same time. The description won't always line up with the response, which might be confusing. Still, separating the response example from the response schema in separate columns helps differentiate the two.

A lot of APIs have modeled their design after Stripe's. For example, see Slate or Spectacle. Should you use a three-column layout with your API documentation? Maybe. But if the response example and description don't line up, the viewer's focus is somewhat split, and the user must resort to more up-and-down scrolling. Additionally, if your layout uses three columns, your middle column will have some narrow constraints that don't leave much room for screenshots and code examples.

The MYOB Developer Center takes an interesting approach in documenting the JSON in their APIs. They list the JSON structure in a table-like way, with different levels of indentation. You can move your mouse over a field for a tooltip description, or you can click it to have a description expand below. The use of tooltips enables the rows containing the example and the description to align perfectly.

To the right of the JSON definitions is a code sample with real values. When you select a value, both the element in the table and the element in the code sample highlight at the same time.



Attribute Details		
UID	Guid (36)	☑
Name	String (30)	
DisplayID	String (6)	
Classification	Classification	
Type	AccountType	☑
Number	Integer (4)	☑
Description	String (255)	
ParentAccount		
UID	Guid (36)	
Name	String (6)	👁
DisplayID	String (6)	👁
U		👁
IsAct		
TaxCode		
UID	Guid (36)	☑
Code	String (3)	👁

Description

Parent account code format includes separator ie 1-1000

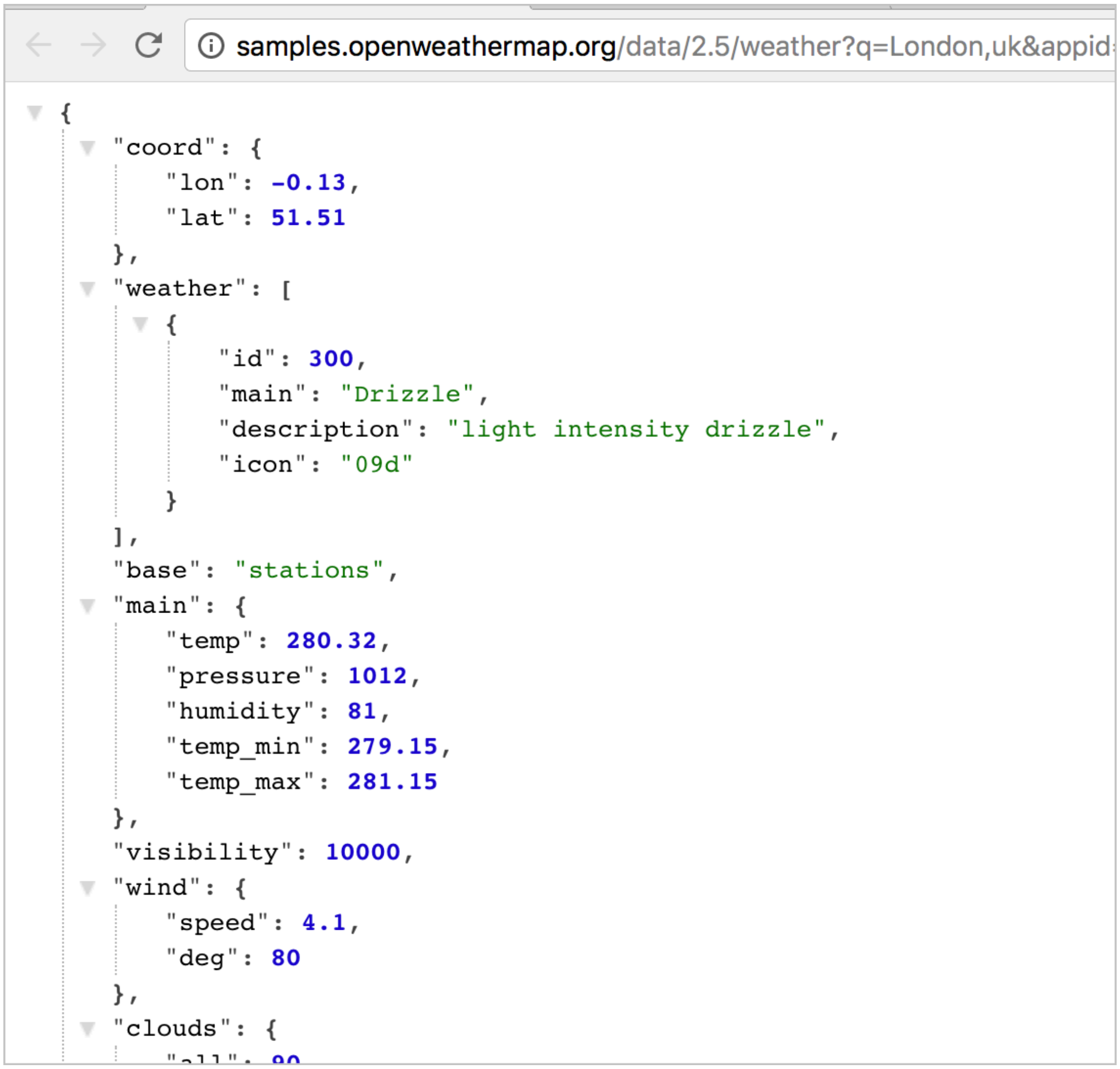
Example json GET response

```
{
   "UID" : "eb043b43-1d66-472b-a6ee-
   ad48def81b96",
   "Name" : "Business Bank Account
   #2",
   "DisplayID" : "1-1120",
   "Classification" : "Asset",
   "Type" : "Bank",
   "Number" : 1120,
   "Description" : "Bank account
   clearwtr",
   "ParentAccount" : {
      "UID" : "f5cc9506-3472-4227-
      8c45-7a95c322c38b",
      "Name" : "Bank Accounts",
      "DisplayID" : "1-1100",
      "URI" : "
      {cf_uri}/GeneralLedger/Account
      3472-4227-8c45-7a95c322c38b"
    },
   "IsActive" : true,
   "TaxCode" : {
      "UID" : "94966872-b140-4da2-
      bc43-5dcd74f33a09",
      "Code" : "N-T",
      "URI" : "
      {cf_uri}/GeneralLedger/TaxCode
      b140-4da2-bc43-5dcd74f33a09"
    },
```

This approach facilitates scanning, and the popover + collapsible approach allows you to compress the table so you can jump to the parts that interest you. However, this approach requires more manual work from a documentation point of view. Still, if you have long JSON objects, it might be worth it.

Embedding dynamic responses

Sometimes responses are generated dynamically based on API calls to a test system. Or if not generated dynamically, they appear to be dynamic. For example, look at OpenWeatherMap API (which we used in earlier activities). When you click a link in the “Examples of API calls” section, such as http://samples.openweathermap.org/data/2.5/weather?q=London, you see the response returned in the browser.



samples.openweathermap.org/data/2.5/weather?q=London,uk&appid

```
{
  "coord": {
    "lon": -0.13,
    "lat": 51.51
  },
  "weather": [
    {
      "id": 300,
      "main": "Drizzle",
      "description": "light intensity drizzle",
      "icon": "09d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 280.32,
    "pressure": 1012,
    "humidity": 81,
    "temp_min": 279.15,
    "temp_max": 281.15
  },
  "visibility": 10000,
  "wind": {
    "speed": 4.1,
    "deg": 80
  },
  "clouds": {
    "all": 90
```

Actually, the OpenWeatherMap response isn't generated dynamically — it just looks that way.

This dynamic approach works well for GET requests that return public information. However, it probably wouldn't scale for other methods (such as POST or DELETE) or which request authorization.

What about status codes?

The responses section sometimes briefly lists the possible status and error codes returned with the responses. However, because these codes are usually shared across all endpoints in the API, status and error codes are often documented in their own section, apart from a specific endpoint's documentation. For this reason, I cover this topic in Documenting status and error codes (p. 402).

Response example and schema for the surfreport endpoint

For the `surfreport/{beachId}` endpoint that we've been exploring in our sample API scenario (p. 115), let's create a section that shows the response example and schema. Here's my approach to these sections:

Sample response

The following is a sample response from the `surfreport/{beachId}` endpoint:

```
{
    "surfreport": [
        {
            "beach": "Santa Cruz",
            "monday": {
                "1pm": {
                    "tide": 5,
                    "wind": 15,
                    "watertemp": 80,
                    "surfheight": 5,
                    "recommendation": "Go surfing!"
                },
                "2pm": {
                    "tide": -1,
                    "wind": 1,
                    "watertemp": 50,
                    "surfheight": 3,
                    "recommendation": "Surfing conditions are okay, not grea
t."
                },
                "3pm": {
                    "tide": -1,
                    "wind": 10,
                    "watertemp": 65,
                    "surfheight": 1,
                    "recommendation": "Not a good day for surfing."
                }
                ...
            }
        }
    ]
}
```

Response definitions

The following table describes each item in the response.

Response item	Description	Data type
beach	The beach you selected based on the beach ID in the request. The beach name is the official name as described in the National Park Service Geodatabase.	String

Response item	Description	Data type
{day}	The day of the week selected. A maximum of 3 days gets returned in the response.	Object
{time}	The time for the conditions. This item is included only if you include a time parameter in the request.	String
{day}/{time}/tide	The level of tide at the beach for a specific day and time. Tide is the distance inland that the water rises to, and can be a positive or negative number. When the tide is out, the number is negative. When the tide is in, the number is positive. The 0 point reflects the line when the tide is neither going in nor out but is in transition between the two states.	Integer
{day}/{time}/wind	The wind speed at the beach, measured in knots (nautical miles per hour). Wind affects the surf height and general wave conditions. Wind speeds of more than 15 knots make surf conditions undesirable because the wind creates white caps and choppy waters.	Integer
{day}/{time}/watertemp	The temperature of the water, returned in Fahrenheit or Celsius depending upon the units you specify. Water temperatures below 70 F usually require you to wear a wetsuit. With temperatures below 60, you will need at least a 3mm wetsuit and preferably booties to stay warm.	Integer
{day}/{time}/surfheight	The height of the waves, returned in either feet or centimeters depending on the units you specify. A surf height of 3 feet is the minimum size needed for surfing. If the surf height exceeds 10 feet, it is not safe to surf.	Integer

Response item	Description	Data type
{day}/{time}/recommendation	An overall recommendation based on a combination of the various factors (wind, watertemp, surfheight). Three responses are possible: (1) "Go surfing!", (2) "Surfing conditions are okay, not great", and (3) "Not a good day for surfing." Each of the three factors is scored with a maximum of 33.33 points, depending on the ideal for each element. The three elements are combined to form a percentage. 0% to 59% yields response 3, 60% - 80% and below yields response 2, and 81% to 100% yields response 1.	String

Next steps

Now that you've completed each of the sections, take a look at all the sections together: Putting it all together (p. 169).

Putting it all together

Let's pull together the various parts we've worked on and bring them together to showcase the full example.

Surfreport

Contains information about surfing conditions, including the surf height, water temperature, wind, and tide. Also provides an overall recommendation about whether to go surfing.

Endpoints

GET **surfreport/{beachId}**

Gets the surf conditions for a specific beach ID.

Parameters

Path parameters

Path parameter	Description
`{beachId}`	The value for the beach you want to look up. Valid `beachId` values are available from our site at sampleurl.com.

Query string parameters

Query string parameter	Required / optional	Description	Type
`days`	Optional	The number of days to include in the response. Default is 3.	Integer
`time`	Optional	If you include the time, then only the current hour will be returned in the response.	Integer. Unix format (ms since 1970) in UTC.

Sample request

```
curl -I -X GET "https://api.openweathermap.org/data/2.5/surfreport?zip=9505
0&appid=APIKEY&units=imperial&days=2"
```

(In the above code, replace `APIKEY` with your actual API key.)

Sample response

The following is a sample response from the `surfreport/{beachId}` endpoint:

```
{
    "surfreport": [
        {
            "beach": "Santa Cruz",
            "monday": {
                "1pm": {
                    "tide": 5,
                    "wind": 15,
                    "watertemp": 80,
                    "surfheight": 5,
                    "recommendation": "Go surfing!"
                },
                "2pm": {
                    "tide": -1,
                    "wind": 1,
                    "watertemp": 50,
                    "surfheight": 3,
                    "recommendation": "Surfing conditions are okay, not grea
t."
                },
                "3pm": {
                    "tide": -1,
                    "wind": 10,
                    "watertemp": 65,
                    "surfheight": 1,
                    "recommendation": "Not a good day for surfing."
                }
                ...
            }
        }
    ]
}
```

Response definitions

The following table describes each item in the response.

Response item	Description	Data type
beach	The beach you selected based on the beach ID in the request. The beach name is the official name as described in the National Park Service Geodatabase.	String
{day}	The day of the week selected. A maximum of 3 days gets returned in the response.	Object
{time}	The time for the conditions. This item is included only if you include a time parameter in the request.	String
{day}/{time}/tide	The level of tide at the beach for a specific day and time. Tide is the distance inland that the water rises to, and can be a positive or negative number. When the tide is out, the number is negative. When the tide is in, the number is positive. The 0 point reflects the line when the tide is neither going in nor out but is in transition between the two states.	Integer
{day}/{time}/wind	The wind speed at the beach, measured in knots (nautical miles per hour). Wind affects the surf height and general wave conditions. Wind speeds of more than 15 knots make surf conditions undesirable because the wind creates white caps and choppy waters.	Integer
{day}/{time}/watertemp	The temperature of the water, returned in Fahrenheit or Celsius depending upon the units you specify. Water temperatures below 70 F usually require you to wear a wetsuit. With temperatures below 60, you will need at least a 3mm wetsuit and preferably booties to stay warm.	Integer
{day}/{time}/surfheight	The height of the waves, returned in either feet or centimeters depending on the units you specify. A surf height of 3 feet is the minimum size needed for surfing. If the surf height exceeds 10 feet, it is not safe to surf.	Integer

Response item	Description	Data type
{day}/{time}/recommendation	An overall recommendation based on a combination of the various factors (wind, watertemp, surfheight). Three responses are possible: (1) "Go surfing!", (2) "Surfing conditions are okay, not great", and (3) "Not a good day for surfing." Each of the three factors is scored with a maximum of 33.33 points, depending on the ideal for each element. The three elements are combined to form a percentage. 0% to 59% yields response 3, 60% - 80% and below yields response 2, and 81% to 100% yields response 1.	String

And that's it. If you have a lot of endpoints to document, you'll probably want to create templates that follow a common structure. Better yet, when you push your reference documentation into the OpenAPI specification, it makes publishing your content even easier. We'll dive more into the OpenAPI specification in the next section, OpenAPI spec and generated reference docs (p. 179).

Next steps

Now that you've completed the API reference tutorial (p. 118), you're ready to do some activities.

There are two activities in this section:

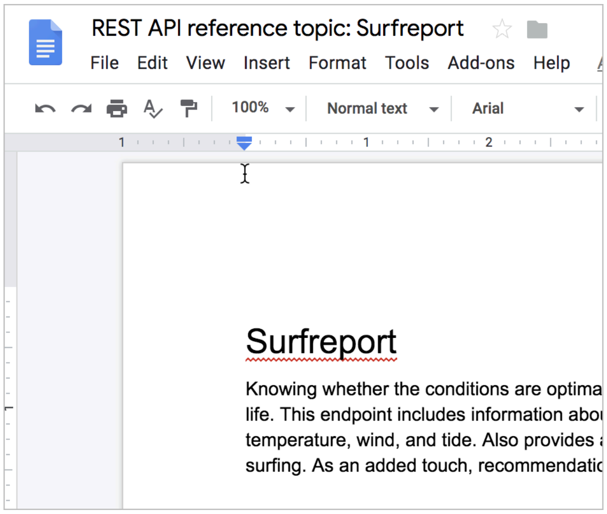
- Activity: What's wrong with this API reference topic (p. 173)
- Activity: Evaluate API reference docs (p. 177)

Activity: What's wrong with this API reference topic

In this activity, you'll critique an existing API reference topic to identify what's wrong with it.

Activity: What's wrong with this API reference topic

The following is a sample API reference topic for an endpoint called `surfreport`. There are about 25 things wrong in the topic. A copy of this same content is available in a read-only Google Doc here. In Google Docs, go to **File > Make a Copy** to create your own instance. Then make comments in Google docs to note as many errors as you can find.



Surfreport

Knowing whether the conditions are optima
life. This endpoint includes information abou
temperature, wind, and tide. Also provides a
surfing. As an added touch, recommendatio

Make a copy of this Google doc and make comments on it.

Here's the same doc (with problems) displayed on the web:

Surfreport

Knowing whether the conditions are optimal for surfing is a necessary detail in every surfer's life. This endpoint includes information about surfing conditions, including the surf height, water temperature, wind, and tide. Also provides an overall recommendation about whether to go surfing. As an added touch, recommendations are expressed in surfer lingo. Surf's up!

Endpoints

GET/POST `surfreport/{:beachId}`

Gets the surf conditions for a specific beach ID.

Parameters

Parameter	Use	Description	Type of data
`{beachId}`	Required	Refers to the ID for the beach you want to look up.	Number
`days`	Optional	The number of days to include in the response. Default is 3. Max 10.	Integer
`time`	Optional	The time you want the report for.	Integer. ISO 8601 format. Example: `20180915T155300+0500`

Sample request

```
https://api.openweathermap.org/data/2.5/surfreport/12345?zip=95050&appid=API
KEY&days=1
```

(In the above code, replace `APIKEY` with your actual API key.)

Sample response

```
{
    "surfreport": [
        {
            "beach": "Santa Cruz",
            "monday": {
                "1pm": {
                    "tide": 5,
                    "wind": 15,
                    "watertemp": 80,
                    "surf_height": 5,
                  "riptide":  "moderate",
                    "recommendation": "Carve it up, brah! The waves are cran
kin' wild out there."
                },
                "2pm": {
                    "tide": -1,
                    "wind": 1,
                    "watertemp": 50,
                    "surf_height": 3,
                 "riptide": extreme
                    "recommendation": "Waves are foam and frothy but rideabl
e in places. Gravitate to the impact zone, due, and hang loose."
                },
                "3pm": {
                    "tide": -1,
                    "wind": 10,
                    "watertemp": 65,
                    "surf_height": 1,
                    "recommendation": "Scene is blown out. Bail inland and c
hill on the beach instead or you'll the one who'll be shredded, due."
                }
                ...
            }
        }
    ]
}
```

Response definitions

The following table describes each item in the response.

Response item	Description	Data type
`beach`	The beach you selected based on the beach ID in the request. The beach name is the official name as described in the National Park Service Geodatabase.	String

`{day}`	The day(s) of the week requested.	object
`{time}`	The time for the conditions.	string
`tide`	The level of tide at the beach for a specific day and time. Tide is the distance inland that the water rises to, and can be a positive or negative number. When the tide is out, the number is negative. When the tide is in, the number is positive. The 0 point reflects the line when the tide is neither going in nor out but is in transition between the two states.	String
`wind`	The wind speed at the beach. Wind affects the surf height and general wave conditions. Wind speeds of more than 15 make surf conditions undesirable because the wind creates white caps and choppy waters.	Int
`watertemp`	The temperature of the water. Water temperatures below 70 usually require you to wear a wetsuit. With temperatures below 60, you will need at least a 3mm wetsuit and preferably booties to stay warm.	String
`surfheight`	The height of the waves, returned in either feet or centimeters depending on the units you specify. A surf height of 3 feet is the minimum size needed for surfing. If the surf height exceeds 10 feet, it is not safe to surf.	Map
`recommendation`	An overall recommendation based on a combination of the various factors (wind, watertemp, surfheight), etc.	String

Answers

You can view the answer key here: What's wrong with this topic answer key.

Activity: Evaluate API reference docs

After completing the API reference tutorial (p. 118), you're ready to start an activity that gives you more experience in creating and editing API reference documentation.

In this activity, you'll evaluate some API reference topics to identify issues.

Activity: Evaluate API reference docs for core elements

In this activity, you'll review API reference documentation and identify the common elements. To evaluate the API reference docs:

1. Choose three of the following sites. In the three you choose, analyze the API reference sections (where the endpoints are listed):
    - Nexmo API reference ↗
    - Twitter API reference ↗
    - Stripe API reference ↗
    - Azure API reference ↗
    - Flickr API reference ↗
    - YouTube API reference ↗
    - Splunk API reference ↗
    - Paypal API reference ↗
    - Basecamp API reference ↗
    - OpenSecrets API reference ↗
    - Dropbox API reference ↗
    - DBS Developers API ↗
    - Adyen Docs ↗
    - NBG Technology Hub ↗
2. In the reference documentation, identify each of the following sections (if they exist):
    - Resource description (p. 120)
    - Endpoints and methods (p. 126)
    - Parameters (p. 132)
    - Request example (p. 141)
    - Response example and schema (p. 154)

    The sections might be named differently in the API doc sites you browse, but they're usually recognizable to some degree (if included). If you're finding it somewhat difficult to locate them, this is part of the wild west of terminology and organization when it comes to API documentation.
3. Assess the API reference documentation by answering the following questions for each section:

    Resource description:

    - Is the description action-oriented?
    - Is it a brief 1-3 sentence summary?

    Endpoints and methods:

    - How are the endpoints grouped? (Are they listed all on the same page, or on different pages? Are they grouped by method, or by resource?)
    - How are the methods specified for each endpoint?

Parameters:

- How many types of parameters are there (header, path, query string) or request body for the endpoints?
- Are the data types (string, boolean, etc.) defined for each parameter? Are required/optional values noted?

Request example:

- In what format or language is the request shown (e.g. curl, specific languages, other)?
- How many parameters does the sample request include?

Response example:

- Is there both a sample response *and* a response schema? (And is each element in the response actually described?)
- How does the doc site handle nested hierarchies in the response definitions?

Next steps

Now that you understand the essential sections to cover in documenting API endpoints, let's look at standardized approaches for describing these sections, primarily with the Overview of REST API specification formats (p. 180).

The OpenAPI standard will help make sure you cover all the necessary details in these sections, and it will present the information to users in a way that users have become accustomed to.

Chapter 4: OpenAPI spec and generated reference docs

The OpenAPI specification provides a formal way of describing your REST API and includes all the reference sections mentioned in the previous section, Documenting API endpoints. Display frameworks such as Swagger UI can parse the OpenAPI specification and generate interactive documentation that lets users try out endpoints while learning about the API.

Chapter sections:

Overview of REST API specification formats 180
Introduction to the OpenAPI specification 181
Stoplight: Visual modeling tools for creating your spec 195
Getting started tutorial: Using Stoplight Studio to create an OpenAPI specification document . 201
Swagger UI tutorial 218
SwaggerHub introduction and tutorial 230
Swagger UI Demo 240
Integrating Swagger UI with the rest of your docs 241
Redocly tutorial -- authoring and publishing API docs with Redocly's command-line tools 249
Redoc Community Edition Basic Demo 268

Overview of REST API specification formats

When I introduced REST APIs (p. 36), I mentioned that REST APIs follow an architectural style, not a specific standard. Several REST specifications were initially developed to provide standards in the way that REST APIs are described. The initial three specs were OpenAPI (formerly called Swagger), RAML, and API Blueprint.

In the early years of specifications, there was healthy competition between the formats. But now the OpenAPI specification is the most popular, with the largest community, momentum, and tooling. Because of this, I spend the most time on OpenAPI in this course. In fact, this entire section focuses on the OpenAPI specification. (I moved RAML (p. 924) and API Blueprint (p. 938) into the Additional resources section at the end.)

“OpenAPI” refers to the specification, while “Swagger” refers to the API tooling that reads and displays the information in the specification. The OpenAPI specification is a vendor-neutral format led by a steering committee comprised of many companies. I’ll dive into both OpenAPI and Swagger in much more depth in the pages to come.

Overall, specifications for REST APIs lead to better reference documentation for your API. Keep in mind that these REST API specifications mostly describe the *reference endpoints* (p. 114) in an API. While the reference topics are important, you will likely have a lot more documentation to write. (This is why I created an entire section of conceptual topics (p. 358).)

Nevertheless, the reference documentation that the specification covers often constitutes the core value of your API, since it addresses the endpoints and what they return.

Writing to a specification introduces a new dimension to documentation that makes API documentation substantially unique. By mastering the OpenAPI specification format, you can distinguish yourself in significant ways from other technical writers.

Introduction to the OpenAPI specification

OpenAPI is a specification for describing REST APIs. If you're familiar with tech comm specifications, you can think of the OpenAPI specification like the DITA specification. With DITA, there are specific XML elements used to define help components, and a required order and hierarchy to those elements. Different tools can read DITA and build out a documentation website from the elements.

With OpenAPI, instead of XML elements, you have a set of JSON objects, with a specific schema that defines their naming, order, and contents. This JSON file (often expressed in YAML instead of JSON) describes each part of your API. By describing your API in a standard format, publishing tools can programmatically parse the information about your API and display each component in a stylized, interactive display.

If you want to jump straight into a step-by-step tutorial for creating the OpenAPI specification document, see Getting started tutorial: Using Stoplight Studio to create an OpenAPI specification document (p. 201), or to code it manually, see the OpenAPI tutorial using Swagger Editor and Swagger UI – overview (p. 272).

Glancing at the OpenAPI specification

To get a better sense of the OpenAPI specification, let's take a glance at some specification excerpts. We'll dive deeper into each element in an upcoming tutorial.

The official description of the OpenAPI specification is available in a Github repository here. Some of the OpenAPI elements are `paths`, `parameters`, `responses`, and `security`. Each of these elements is a JSON object that holds some properties and arrays.

In the OpenAPI specification, your endpoints are `paths`. If you had an endpoint called "pets", your OpenAPI specification for this endpoint might look as follows:

```
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: An paged array of pets
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pets"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
```

This YAML code comes from the Swagger Petstore demo.

Here's what these objects mean:

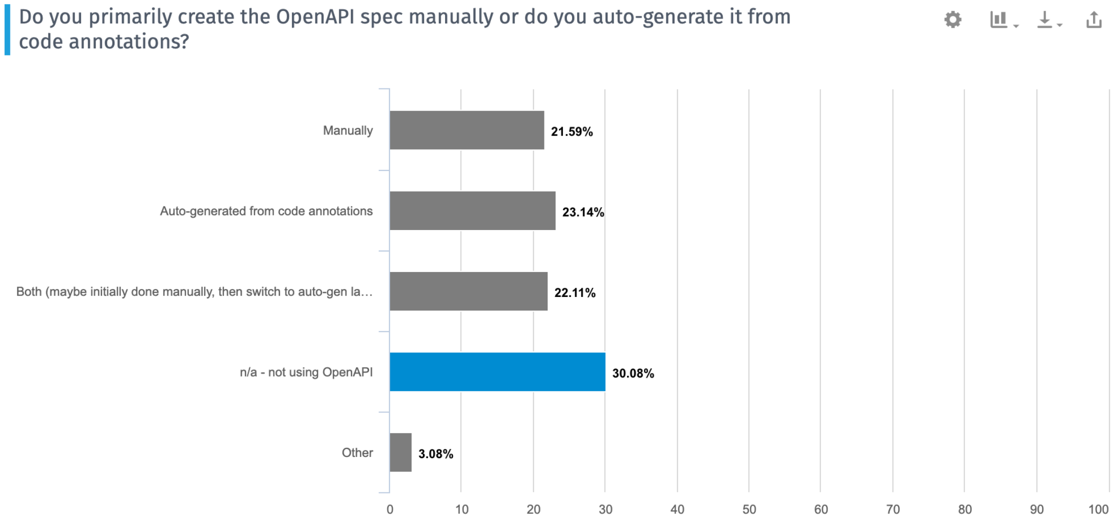
- `/pets` is the endpoint path.
- `get` is the HTTP method.
- `parameters` lists the parameters for the endpoint.
- `responses` lists the response from the request.
- `200` is the HTTP status code.
- `$ref` is a reference to another part of your implementation where the response is defined (in `components`). OpenAPI has a lot of `$ref` markers like this to keep your code clean and to facilitate re-use.

Learning the OpenAPI spec and constructing the YAML or JSON code by hand the first time, when you're also trying to sort out what the values should be, can be daunting. It might be easier to start in a visual editor like Stoplight and then, if desired, transition into more of a direct code-based approach. For this reason I have arranged the sequence of this course to start out first in the visual editor, with Create an

OpenAPI document using a visual editor (Stoplight Studio) (p. 195), and then later in the course provide an OpenAPI tutorial overview using Swagger Editor and Swagger UI (p. 272). However, you can probably do most everything from within the visual editor.

Auto-generation options for creating the OpenAPI spec

Before we dive into ways to manually create the OpenAPI specification document, it's worth noting some other approaches to the task, beyond choosing whether to use a visual editor or manually write the code. You can also choose to auto-generate the OpenAPI spec from annotations in your programming code. This developer-centric approach may make sense if you have a large number of APIs or if it's not practical for technical writers to create this documentation. In my 2020 Developer Documentation Trends survey (p. 44), approaches for auto-generating the spec versus manually generating it were split:



Percentage of auto-generation versus manual creation

If you want to go the code-generation route, Swagger offers a variety of libraries that you can add to your programming code to generate the specification document. These Swagger libraries then parse the annotations that developers add and generate the OpenAPI specification document. These libraries are considered part of the "Swagger Codegen" project. The annotation methods vary based on the programming language. For example, here's a tutorial on annotating code with Swagger for Scalatra.

See There's No Reason to Write OpenAPI By Hand by Phil Sturgeon for some more discussion. For example, Phil provides this sample of code annotations in Java:

```
class UserController {
  @OpenApi(
      path = "/users",
      method = HttpMethod.POST,
      // ...
  )
  public static void createUser(Context ctx) {
      // ...
  }
}
```

And the approach in JavaScript:

```
/**
 * @swagger
 * /users:
 *   get:
 *     description: Returns users
 *     produces:
 *      - application/json
 *     responses:
 *       200:
 *         description: users
 *         schema:
 *           type: array
 *           items:
 *             $ref: '#/definitions/User'
 */
app.get('/users', (req, res) => {
  // implementation logic ...
});
```

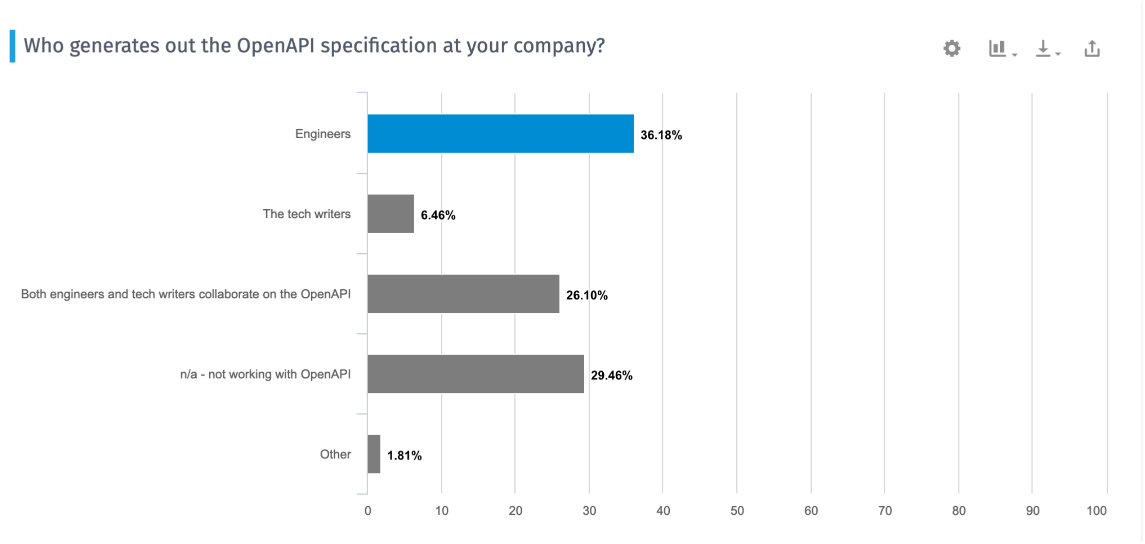
You can see that the annotations differ by language, mostly following the allowed annotations and traditional syntax for that language. So if you want to go this route, first find out what Swagger Codegen libraries are available for that language. For more information on Codegen, see Comparison of Automatic API Code Generation Tools For Swagger by Kin Lane, aka the API Evangelist. For additional tools and libraries, see Swagger services and tools and Open Source Integrations. Note that currently, most of the Codegen libraries only work with version 2.0 of the OpenAPI. Codegen tool have been slow to transition to version 3.0.

Although this code-generation approach “automates” the spec’s generation, someone still has to know what annotations to add and how to add them (the process isn’t too unlike Javadoc’s comments and annotations). Then someone has to write content for each of the annotation’s values (describing the endpoint, the parameters, and so on).

In short, this process isn’t without effort — the automated part is having the Codegen libraries generate the model definitions and the valid specification document that conforms to the OpenAPI schema. The end result is an OpenAPI specification document that you’ll then need to feed into another tool such as Swagger UI (p. 218), Redocly (p. 249), or another tool to generate out the documentation.

Still, many developers get excited about this approach because it offers a way to generate documentation from code annotations, which is what developers have been doing for years with other programming languages such as Java (using Javadoc) or C++ (using Doxygen). They usually feel that generating documentation from the code results in less documentation drift. Docs are likely to remain up to date if the doc is tightly coupled with the code.

It's common that engineers generate the OpenAPI spec, so you wouldn't be out of place to simply ask developers to generate out this file from their code, writing and maintaining their own reference documentation. In my 2020 Developer Documentation Trends survey (p. 44), I asked respondents who typically generates the OpenAPI spec. Their answers trends towards either engineers or engineers + tech writers in collaboration:



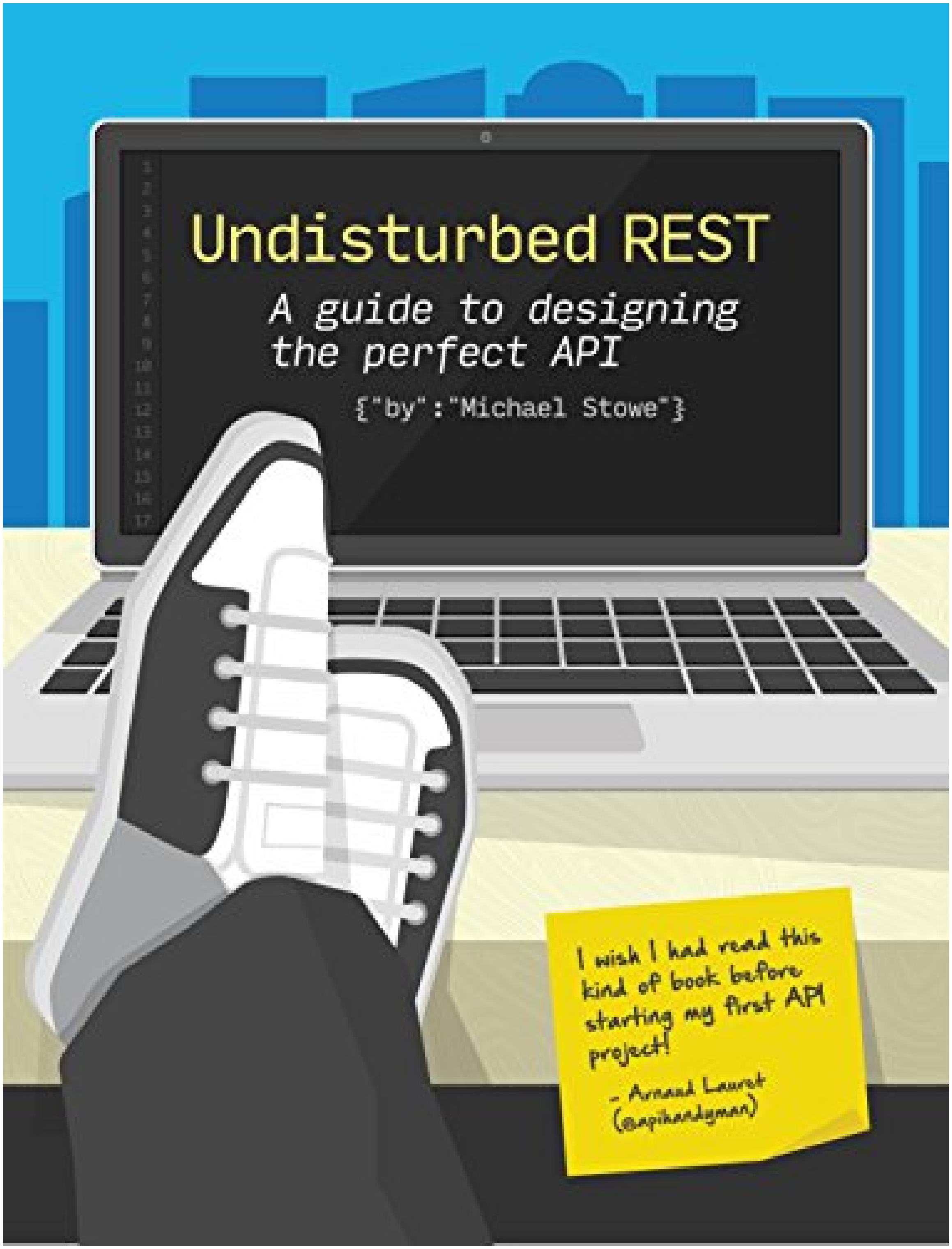
Who generates the OpenAPI spec

If engineers are generating the OpenAPI spec and they want to auto-generate it from annotations in their code, you might want to follow their preference. If you go the auto-generation route, make sure you get access to the source code to make edits to the annotations and ramp up on Git. Otherwise, your developers will be writing your reference docs.

In this course, however, I will not be covering code annotations and ways to auto-generate the spec. I focus exclusively on manually creating the code, since this follows a best practice in the industry. The main weakness of generating your OpenAPI spec from code is that the spec can no longer serve as the blueprint for creating the API, and instead becomes an afterthought. I'll explain this in more depth in the next section, Spec-first development of the OpenAPI spec (p. 185).

Spec-first development of the OpenAPI spec

Now let's talk about the spec-first approach. Many feel that auto-generation, described in the previous section, is *not* the best approach. In *Undisturbed REST: A Guide to Designing the Perfect API*, Michael Stowe recommends that teams implement the OpenAPI specification by hand and then treat the specification document as a contract that developers use when doing the actual coding. This approach is often referred to as "spec-first development."


Undisturbed REST
A guide to designing the perfect API
{"by":"Michael Stowe"}
I wish I had read this kind of book before starting my first API project!
- Arnaud Lauret (@apihandyman)

Spec-first development is a philosophy about how to develop APIs more efficiently. If you follow a spec-first philosophy, you write the spec first and use it as a contract that developers code to.

In other words, developers consult the OpenAPI specification document to see what the parameter names should be called, what the responses should be, and so on. After this “contract” or “blueprint” has been established, Stowe says you can then put the annotations in your code (if you want) to generate the specification document in a more automated way. But don’t code without first having a spec.

Too often, development teams quickly jump to coding the API endpoints, parameters, and responses without doing much user testing or research into whether the API aligns with what users want. Since versioning APIs is extremely difficult (you have to support each new version going forward with full backward-compatibility to previous versions), you want to avoid the “fail fast” approach that agile enthusiasts so commonly celebrate. There’s nothing worse than releasing a new version of your API that invalidates endpoints or parameters used in previous releases. Constant versioning in APIs can become a documentation nightmare.

Even before the API has been coded, your spec can generate a mock response (p. 236) by adding response definitions in your spec. The mock server generates a response that looks like it’s coming from a real server, but in reality it’s just a pre-defined response in your code and appears to be dynamic to the user.

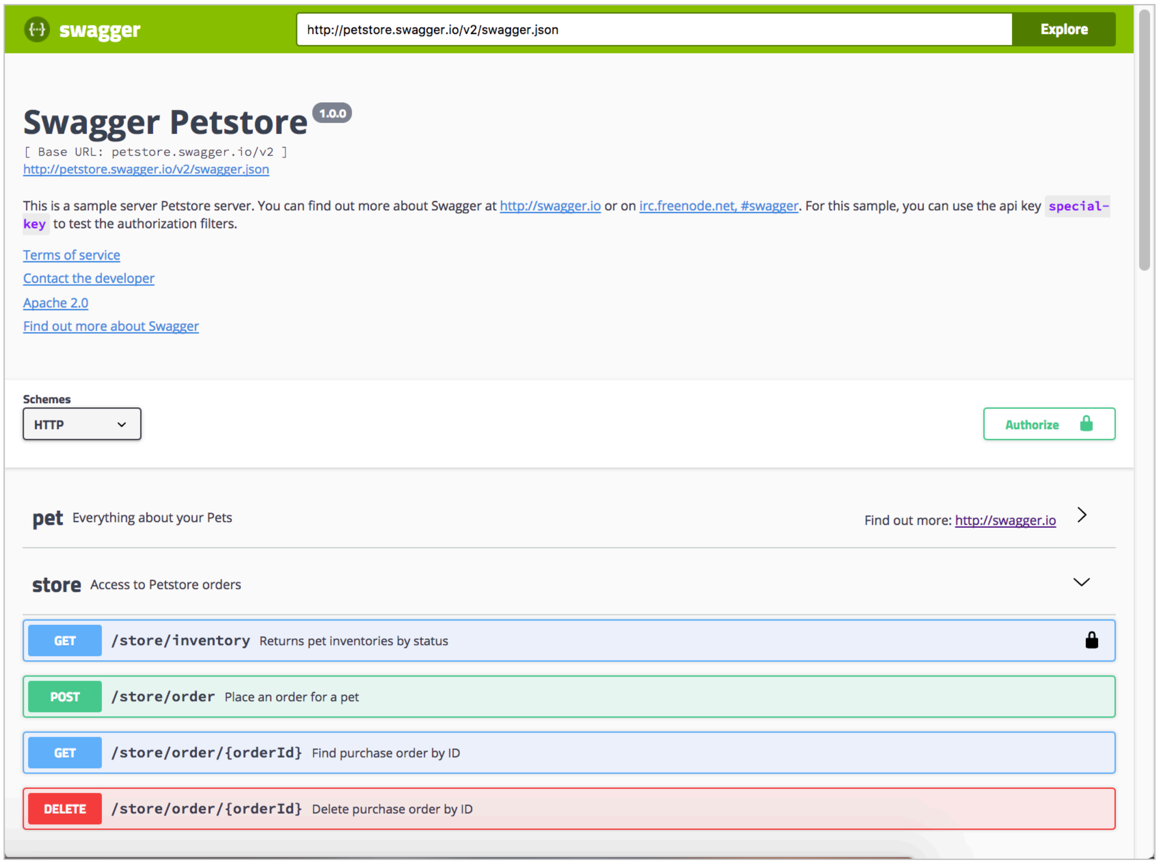
If the OpenAPI spec isn’t used as the blueprint for the API, what is? Most likely, the descriptions about the endpoints and what they return will be stored on a wiki page or Word document, often incomplete in terms of all the details (e.g., casing of the responses, the structure and what’s included), so developers will make these decisions on the fly while coding.

Rendering Your OpenAPI specification with Swagger UI

After you have a valid OpenAPI specification document that describes your API, you can then feed this specification to different tools to parse it and generate the interactive documentation similar to the Petstore demo.

Probably the most common tool used to parse the OpenAPI specification is Swagger UI. (Remember, “Swagger” refers to API tooling, whereas “OpenAPI” refers to the vendor-neutral, tool agnostic specification.) After you download Swagger UI, it’s pretty easy to configure it with your own specification file. I provide a Swagger UI tutorial (p. 218) in an upcoming section.

The Swagger UI code generates a display that looks like this:



Swagger Petstore 1.0.0

[Base URL: petstore.swagger.io/v2]

http://petstore.swagger.io/v2/swagger.json

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service

Contact the developer

Apache 2.0

Find out more about Swagger

pet Everything about your Pets — Find out more: http://swagger.io

store Access to Petstore orders

GET `/store/inventory` Returns pet inventories by status

POST `/store/order` Place an order for a pet

GET `/store/order/{orderId}` Find purchase order by ID

DELETE `/store/order/{orderId}` Delete purchase order by ID

The Swagger Petstore demo shows how Swagger UI renders the OpenAPI spec

You can also check out the sample Swagger UI integration with a simple weather API used as a course example.

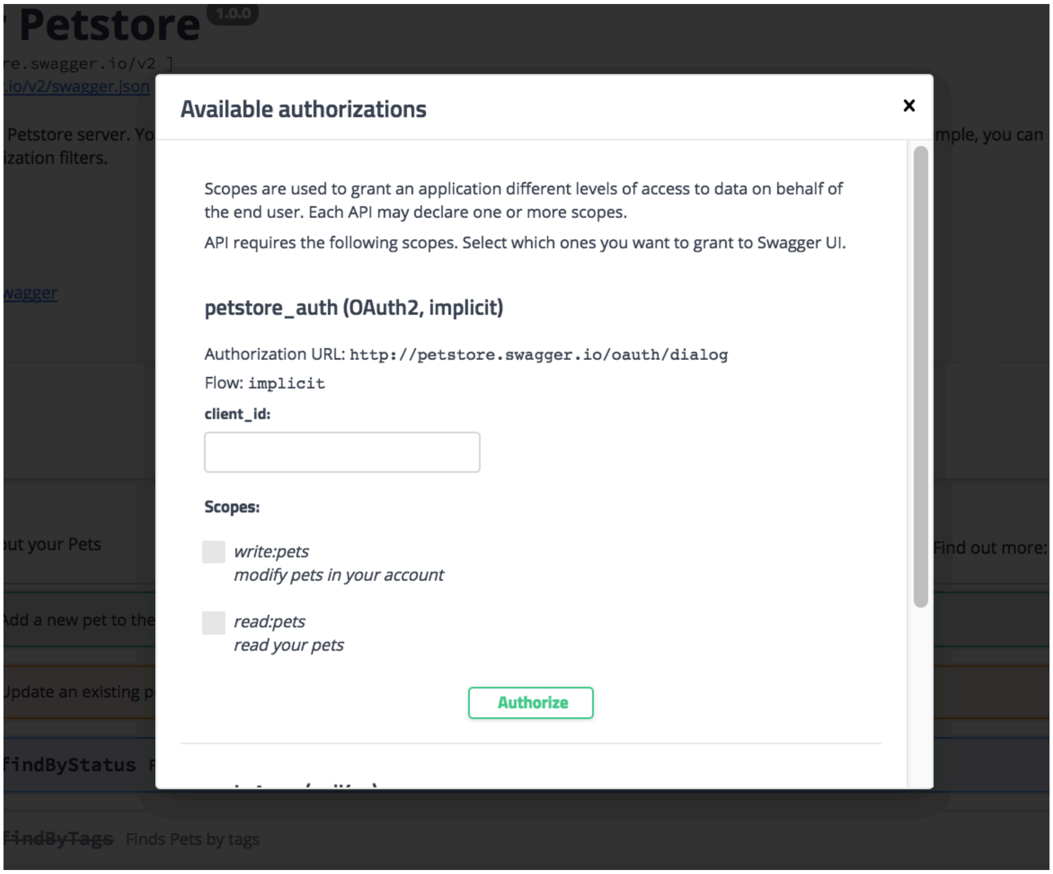
Some designers criticize Swagger UI's expandable/collapsible output as being dated. At the same time, developers find the one-page model attractive and like the ability to zoom out or in for details. By consolidating all endpoints on the same page in one view, users can take in the whole API at a glance. This display gives users a glimpse of the whole, which helps reduce complexity and enables them to get started. In many ways, the Swagger UI display is a quick-reference guide (p. 412) for your API.

Activity: Explore Swagger UI through the Petstore Demo

Let's get some hands-on experience with Swagger UI using the Petstore demo. The Petstore demo provides a good example of how the OpenAPI specification can be rendered visually.

1. Go to the Swagger Pet Store Demo.

   As with most Swagger-based outputs, Swagger UI provides a "Try it out" button. To make it work, you must first authorize Swagger by clicking **Authorize** and entering your API key in the Authorization modal. However, the Petstore authorization modal is just for demo purposes. There isn't any real code authorizing those requests, so you can close the Authorization modal or skip it altogether.



Available authorizations

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.

API requires the following scopes. Select which ones you want to grant to Swagger UI.

petstore_auth (OAuth2, implicit)

Authorization URL: `http://petstore.swagger.io/oauth/dialog`

Flow: `implicit`

client_id:

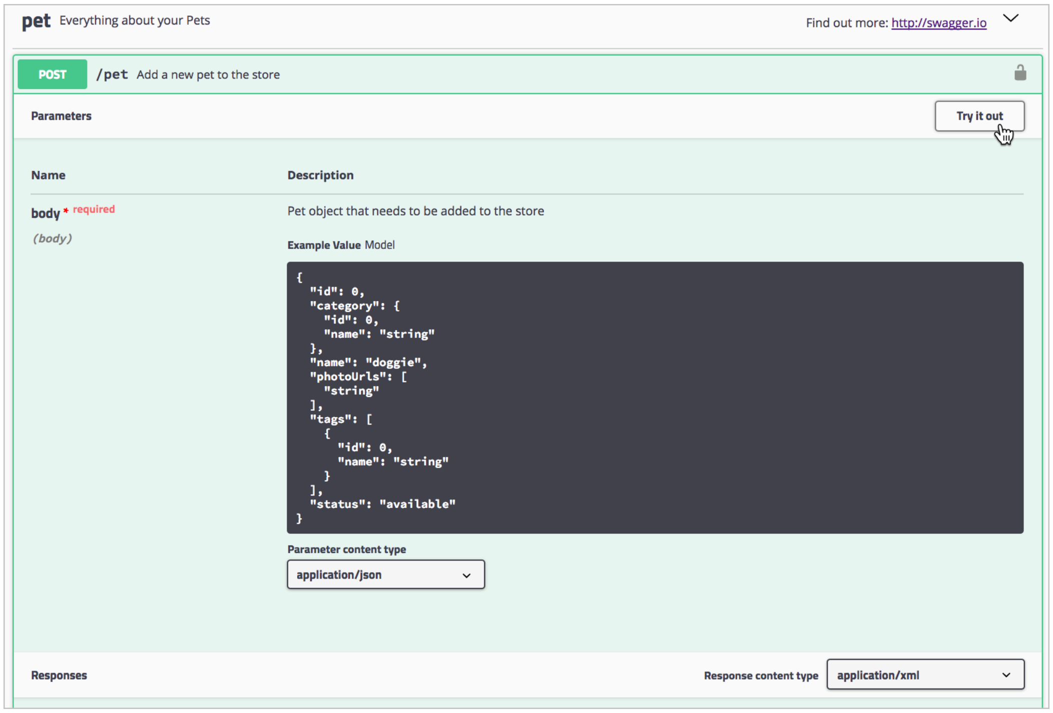
Scopes:

☐ *write:pets*
modify pets in your account

☐ *read:pets*
read your pets

Authorization modal in Swagger UI

2. Expand the **POST** `/pet` endpoint.


pet Everything about your Pets
Find out more: http://swagger.io
POST
/pet Add a new pet to the store
Parameters
Try it out
Name
Description
body * required
(body)
Example Value Model
Responses

Pet object that needs to be added to the store

```
{
  "id": 0,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

Parameter content type: application/json

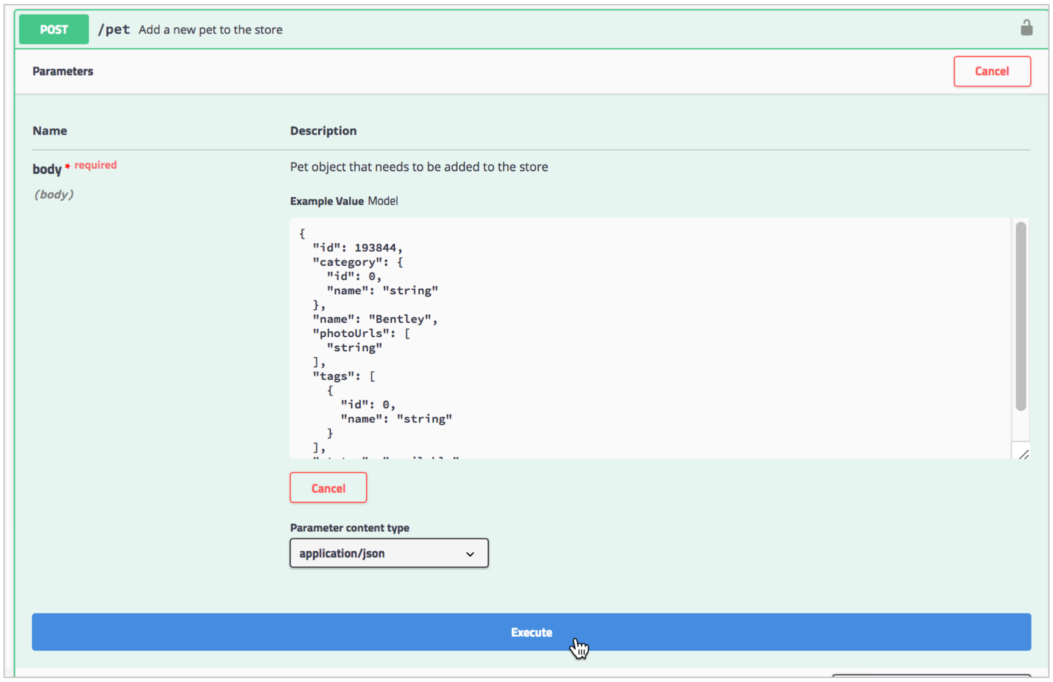
Response content type: application/xml

POST /pet endpoint and Try it out button in Swagger UI

3. Click **Try it out**.

   After you click Try it out, the example value in the Request Body field becomes editable.

4. In the example value, change the first `id` value to a unique (and unlikely to be repeated) whole number (such as `24329`). Change the name `doggie` to a pet name you can remember (e.g., `Bentley`).
5. Click **Execute**.



Name	Description
body * required (body)	Pet object that needs to be added to the store

```
{
  "id": 193844,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "Bentley",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
```

Parameter content type: application/json

Executing a sample Petstore request

Swagger UI submits the request and shows the curl (p. 80) that was submitted. For example, here's the curl Swagger UI sent:

```
curl -X POST "https://petstore.swagger.io/v2/pet" -H "accept: applica
tion/xml" -H "Content-Type: application/json" -d "{ \"id\": 1000, \"c
ategory\": { \"id\": 0, \"name\": \"string\" }, \"name\": \"Bentle
y\", \"photoUrls\": [ \"string\" ], \"tags\": [ { \"id\": 0, \"nam
e\": \"string\" } ], \"status\": \"available\"}"
```

Notice that, with the `-d` (data) parameter, the request body is escaped and added directly into the curl command rather than being loaded from a file (as explained in Common curl commands related to REST (p. 85)).

The Responses section in Swagger UI shows the response from the server. By default, the response returns JSON:

```
{
  "id": 1000,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "Bentley",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
  }
```

6. The Petstore is a functioning API, and you have actually created a pet. For fun, expand the **GET /pet/{petId}** endpoint, click **Try it out**, enter the pet `id` you used in the previous operation, and then execute the request. You should see your pet's name returned.

There are other tools besides Swagger UI that can parse your OpenAPI specification document. Some of these tools include Apiary, Apigee, Lucybot, Gelato, Readme.com, swagger2postman, swagger-ui responsive theme, and more.

Some web designers have created integrations of OpenAPI with static site generators such as Jekyll (see Carte and Readme). You can also embed Swagger UI into web pages as well. More tools roll out regularly for parsing and displaying content from an OpenAPI specification document.

In fact, once you have a valid OpenAPI specification, using a tool called API Transformer, you can even transform it into other API specification formats, such as RAML or API Blueprint. These additional formats allow you to expand your tool horizons even wider.

The tech writer's role with the OpenAPI spec

In many of my projects, developers haven't been that familiar with OpenAPI, so I usually create the OpenAPI specification document manually. Additionally, I don't always have access to the programming source code, and our developers speak English as a second or third language only. They aren't eager to be in the documentation business.

You will most likely find that engineers in your company aren't familiar with OpenAPI but are interested in using it as an approach to API documentation. (I think the spec's schema-based approach fits the engineering mindset, so it's not hard to get buy-in for the approach.) But you'll probably need to take the lead to guide engineers in the needed information, approach, and other details that align with best practices toward creating the spec.

In this regard, tech writers can play a key role in collaborating with the API team in producing the spec. If you're following a spec-first development philosophy, this leading role can help you shape the API before it gets coded and locked down. This means you might have the opportunity to influence the names of the endpoints, the consistency and patterns, simplicity, and other factors that go into the design of an API (which tech writers are usually absent from). See API design and usability (p. 488) for details on how to influence good API design.

Should I even use the OpenAPI spec? What if I just created the reference content manually in my own format and tool?

One of the main reasons for tech writers to use the OpenAPI spec is to auto-generate out the reference documentation in a neat display that lets users try out requests. There are a host of tools that will read an OpenAPI spec and then generate out docs. I'll go into more depth with these approaches in later sections (such as the Stoplight tutorial (p. 201), Swagger UI tutorial (p. 218), and Redoc tutorial (p. 249)).

However, up front you should be aware that there are tradeoffs to using the OpenAPI approach. If you write your reference docs using the OpenAPI spec and then settle on your preferred way of generating out the interactive docs, you're probably going to run into these challenges:

- **Tutorial/how-to docs and reference docs become fragmented**: The OpenAPI specification and the rendering tool typically cover only reference documentation (p. 114). The OpenAPI provides the basics of each endpoint, including a description, the parameters, a sample request, and a response. It doesn't provide space for a getting started tutorial (p. 379), information about how to get API keys (p. 393), how to download and configure a sample app (p. 480), information about rate limits (p. 408), or the hundred other details that go into the conceptual topics for developers (p. 358). So even though you have this cool, interactive tool for users to explore and learn about your API, you still have to provide a user guide. Your docs will likely have a standalone reference output that acts as a companion to the tutorial content. Some platforms let you mix the two (e.g., Stoplight, Readme.com, and others), but then you'll be paying for a hosted documentation service. I talk more about this issue in Integrating Swagger UI with the rest of your docs (p. 241).
- **Redundancy/duplication of information**: With OpenAPI in the mix, you potentially have *two places* where you're describing your endpoints and parameters (both the reference output and your user guide), and you have to either keep the two in sync, embed one in the other, or otherwise link between the two. The OpenAPI lets you re-use parameter and response information through the `components` object, but you couldn't easily re-use the information outside the reference material.
- **Complexity of API workflows**: The complexity of your API can also be a factor to consider in your approach. Peter Gruenbaum, who has published several tutorials on writing API documentation on Udemy, says that automated tools work best when the APIs are simple. When you have endpoints that have complex interdependencies and require special setup workflows or other unintuitive treatment, the straightforward nature of a Try-it-out interface (by this I mean the generated output from tools such as Swagger UI (p. 218) or Redoc (p. 249)) may likely leave users scratching their heads. For example, if you must first configure an API service before an endpoint returns anything, and then use one endpoint to get a certain object that you pass into the parameters of another endpoint, and so on, the Try-it-out features in the output won't make a lot of sense to users without a detailed tutorial to follow.
- **Executing requests against real data**: Some users may not realize that clicking "Try it out" makes actual calls against their own accounts based on the API keys they're using. Mixing an invitation to use an exploratory sandbox with real data can create some headaches later on when users ask how they can remove all of the test data, or why their actual data is now messed up. For these scenarios, it's best to either remove the Try-it-out options or set up a sandbox or test account for users. But this is easier said than done. You might find that your company doesn't provide a sandbox for testing out the API. All API calls execute against real data.

- **Extensive request bodies problematic**: Finally, I found that endpoints with lengthy request bodies tend to be problematic. One API I had to document included requests with request bodies that were *hundreds of lines long* (the request body was used to configure an API server). With this sort of request body, many of these tools that create docs from the OpenAPI spec fall short of being usable. You might have an unwieldy list of parameters or some other custom/complex setup and might want more control over your docs.

Despite these shortcomings, OpenAPI is quickly becoming a way for more and more tools (from Postman Run buttons (p. 382) to nearly every API platform) to quickly ingest the information about your API and make it discoverable and interactive with robust, instructive tooling. Through your OpenAPI specification, you can port your API onto many platforms and systems as well as automatically set up unit testing and prototyping. Your OpenAPI spec is machine-readable and will give much more life and flexibility to your API docs.

Additionally, the generated docs from the OpenAPI provide a nice visual shape for an API. You can easily see all the endpoints and their parameters &dash; it acts like a quick-reference guide (p. 412). Based on this framework, you can help users grasp the basics of your API at a glance.

I’ve also found that learning the OpenAPI specification and describing my API with these objects and properties helped inform my own API vocabulary. For example, I realized that there were four main types of parameters (p. 132): “path” parameters, “header” parameters, “query” parameters, and “request body” parameters. I learned that parameter data types with REST were a “Boolean,” “number,” “integer,” or “string.” I learned that responses (p. 154) provided “objects” containing “strings” or “arrays.” In short, implementing the specification gave me an education about API terminology, which in turn helped me describe the various components of my API in credible ways.

OpenAPI may not be the right approach for every API, but it’s an industry standard and can be a powerful complement to your API documentation. You can give users the ability to try out requests and responses for themselves. With this interactive element, your documentation becomes more than just information. Through OpenAPI and the interactive docs you generate from it, you create a space for users to both read your documentation and experiment with your API at the same time. That combination tends to provide a powerful learning experience for users.

Stoplight: visual modeling tools for creating your OpenAPI spec

Before jumping into the tutorial for creating an OpenAPI spec using Stoplight (p. 201), let me introduce Stoplight a bit more conceptually. As a GUI editor for creating an OpenAPI, Stoplight Studio simplifies the authoring of a highly technical syntax that is often challenging to make valid.

Among other services, Stoplight provides visual modeling tools to create an OpenAPI document for your API — without requiring you to know the OpenAPI spec details or code the spec line by line. This API specification document can act as a single source of truth that empowers the whole API lifecycle, from UX prototyping to testing, development, documentation, sales, and more.

Note that Stoplight is one of the sponsors of my site.

Limits to line-by-line spec coding

Before jumping into details, let me provide some background about why I think Stoplight is a valuable tool, and why I've both listed it in this course and created an activity for using Stoplight (p. 201). At the 2018 TC Camp conference ↗ in Santa Clara, the camp organizers put on a full-day API workshop focused on OpenAPI and Swagger. I was excited to see this topic addressed in a workshop because I think coding the spec is both the most complicated and most important part of API documentation.

I didn't attend the workshop myself, but I was chatting with a few who did. One attendee was a little frustrated that they spent *so much time in YAML (p. 276)* working on different parts of the OpenAPI spec definition (p. 272). He said they spent most of the day in YAML, and it was frustrating, tedious, and boring. For this participant, this isn't what he imagined when he signed up to learn how to create interactive API docs.

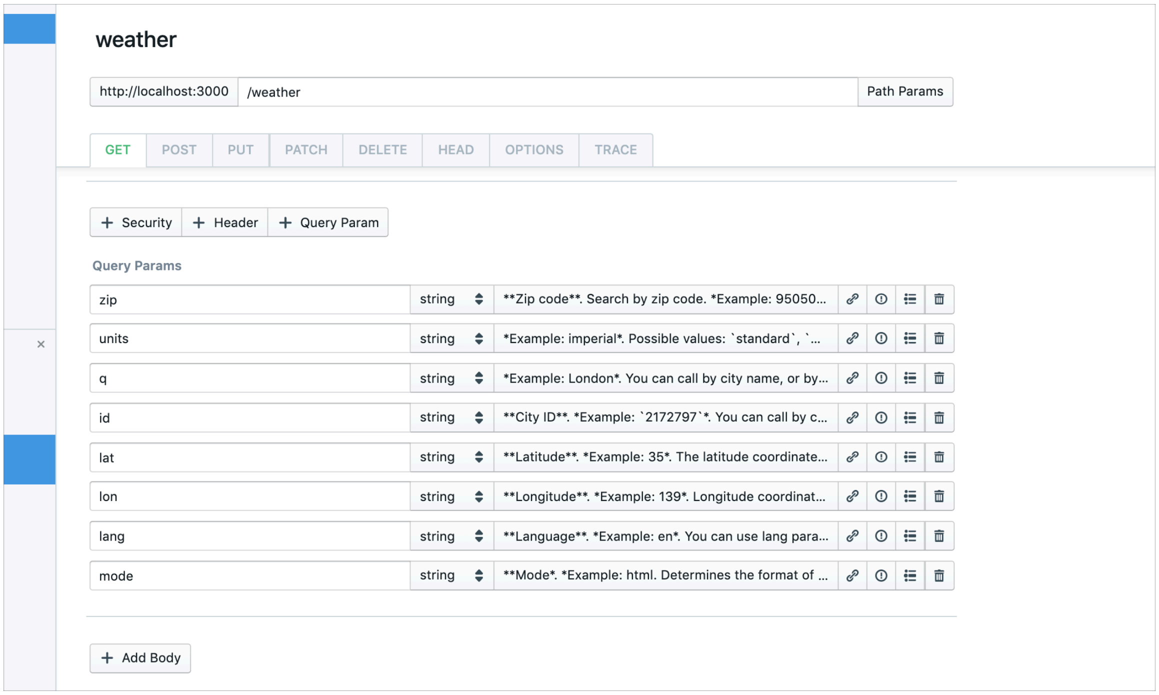
In the instructor's defense, I told my friend that describing an API using the OpenAPI spec *does* pretty much involve living in YAML all day, and it *is* tedious, highly prone to error, and technical. One of my favorite API bloggers, API evangelist Kin Lane, explains that "hand crafting even the base API definition for any API is time-consuming." It is an activity "that swells quickly to being hours when you consider the finish work that's required" (Automated Mapping Of The API Universe … ↗).

Lane says he was exploring ways to automate the API definition using different tools such as Charles Proxy. During this time, he started exploring Stoplight.io ↗, a platform for modeling APIs and more, and he became engrossed in the workflow and design tools. He says,

> I stayed up way too late playing with some of the new features in Stoplight.io. If you aren't familiar with what the Stoplight team has been cooking up — they have been hard at work crafting a pretty slick set of API modeling tools. I feel the platform provides me with a new way to look at the API life cycle — a perspective that spans multiple dimensions, including design, definition, virtualization, documentation, testing, discovery, orchestration, and client. … I am curious to see what API designers and architects do with Stoplight — I feel like it has the potential to shift the landscape pretty significantly, something I haven't seen any API service provider do in a while. (Automagically Defining Your API Infrastructure As You Work Using Stoplight.io ↗)

I also started playing around with Stoplight. I was curious to see whether the visual modeling tools for describing an API could take the tedium out of working in YAML on a line-by-line level with the spec. While using Stoplight to create an OpenAPI description for a recent web API at my work, I found Stoplight to be useful. It made it much easier to create the OpenAPI specification document.

Stoplight's visual modeling tools eliminate the need to be familiar with the format of the OpenAPI specification. You don't have to know the data type for each property, whether the property needs to be nested or defined directly and so forth. That level of complexity has been abstracted away in a GUI for designing your API.



zip	string	**Zip code**. Search by zip code. *Example: 95050...
units	string	*Example: imperial*. Possible values: `standard`, `...
q	string	*Example: London*. You can call by city name, or by...
id	string	**City ID**. *Example: `2172797`*. You can call by c...
lat	string	**Latitude**. *Example: 35*. The latitude coordinate...
lon	string	**Longitude**. *Example: 139*. Longitude coordinat...
lang	string	**Language**. *Example: en*. You can use lang para...
mode	string	**Mode*. *Example: html. Determines the format of ...

Stoplight provides visual modeling tools to describe your API. The screenshot above shows the UI for documenting parameters. Stoplight's UI produces a valid OpenAPI specification document, which is then used to drive other API services and activities on their site.

As part of the visual modeling tools, Stoplight's interface for describing JSON schemas (used in request bodies or responses) is especially welcome. Details about how to document JSON schemas aren't fully described in the OpenAPI spec, so they can be particularly tricky. What's especially neat about Stoplight is that you can paste in a chunk of JSON and it will automatically describe the JSON in the right syntax for you. You do this using the **Generate from JSON** button. For more details, see this section in the Stoplight getting started tutorial: Step 5: Enter the responses and response schema information (p. 213). Here's a short video clip showing this auto-generation:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/pubapis_stoplight.html] to view the content.

Additionally, you can toggle between the visual tools (the Form view) and the specification code (the Code view) easily. If you want to work in the code, your changes will update the content in the UI as well. The two sync perfectly when you make updates in either mode. Here's a short video I made showing this bidirectional synchronization:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/pubapis_stoplight.html] to view the content.

Not just simpler tools, but a design-first philosophy

After playing around with Stoplight, I had the opportunity to chat with Marc Macleod, founder of Stoplight, about how Stoplight differs from SwaggerHub (p. 230) and Readme (p. 609). Marc said when the spec was first introduced, he saw value in having a standard specification for APIs, but at the time, all the tooling required users to write the specification line by line. This hand coding was error-prone, slow, and tedious.

Marc and his team designed Stoplight with visual modeling tools that don't require teams to know the details of the OpenAPI spec. This simplification of tooling opens up the spec's development to a broader number of team players — to product managers, developers, UX designers, technical writers, and more. The barrier to entry in the design and prototyping of the API grows beyond the scope of just engineers.

Building the specification document is probably the most important activity in API development because once you have this API description, you have a **single source of truth**. This single source of truth can then inform and empower a variety of other roles: developers, testers, user experience designers, technical writers, sales, and more. Marc's core philosophy is that the OpenAPI specification document is central to API development. After you have this specification document, you can build tools around it to empower these other teams. For example:

- UX designers can prototype the API using a mock server to let users execute requests and see sample responses — before developers even write one line of code.
- Developers can write code by following a specific contract, like a construction team following a blueprint. All the decisions and questions have been put into the specification document to make it actionable.
- QA can automate unit testing from the API description to speed up endpoint testing across a variety of parameters and scenarios.
- Technical writers can add descriptions and other examples to the specification description, and then generate interactive documentation without worrying about developing templates, styles, or other formatting and organization.

If the OpenAPI specification really powers all of these other activities, doesn't it make sense to build your platform around the specification? And from a larger view, to build your business around the specification? That's what Stoplight is doing. It's what makes them fundamentally different from other API platforms. I think it's what Kin Lane meant when he said Stoplight provides "new way to look at the API life cycle — a perspective that spans multiple dimensions, including design, definition, virtualization, documentation, testing, discovery, orchestration, and client." From a technical writer's perspective, this puts documentation as the center of API development activities.

Not a post-design artifact to generate documentation

The OpenAPI specification isn't just an artifact that describes what the developers already coded. Nor is it just a way to create interactive docs featuring a built-in API explorer, or even to make your API machine readable for other systems to consume. The OpenAPI specification is a way to *design and model* your API. Given this purpose, the tools for designing and modeling your API need to be more flexible, easier to manipulate, and accessible to designers and product managers.

Consider this analogy. When I write blog posts, I often write in an editor like Google Docs or Bear or Ulysses or even Word, because these tools make it easy to express myself. I can edit and move content around, or insert notes and half-formed thoughts. Only after I've finished the content do I move it into Markdown or HTML and then populate the structured YAML in my post's frontmatter. It's the same with the

API specification. When you're designing and modeling your API, you don't want to be worrying about whether your YAML syntax is valid or be constantly consulting the reference documentation to remember what properties are required at each level of the specification.

If we're ever going to embrace modeling and designing APIs in a collaborative way, we can't do it using a YAML editor writing in the spec's rigid syntax. If we don't have tools to design and model collaboratively, the spec gets designed and developed elsewhere (such as in the pages of Confluence or a Word document on a product manager's computer). The specification document becomes an afterthought to design, something that a techie (such as a developer or technical writer) comes along later to create *post-development*.

Spec-last development has, unfortunately, been the approach in most of the APIs I've worked on — I create the spec *after* the API has already been designed and coded. The spec becomes just a way to generate reference documentation for the existing API, rather than as a single source of truth that empowers the whole API lifecycle from beginning to end. Invariably, as soon as user testing begins, the project team discovers shortcomings in the API's design they don't have time to fix.

This practice of putting the spec last (rather than first) in the API's development limits the scope of what the OpenAPI specification can do. Lane explains:

> Many developers still see OpenAPI (fka Swagger) about generating API documentation, not as the central contract that is used across every stop along the API life cycle. Most do not understand that you can mock instead of deploying, and even provide mock data, errors, and other scenarios, allowing you to prototype applications on top of API designs. (Code Generation Of OpenAPI (fka Swagger) Still The Prevailing Approach ↗)

To counter poor practices with *spec-last* development, Lane says more and more platforms are pushing code development further down in the API lifecycle. In other words, design and testing are done first, code development is done later.

In a design-first model, technical writers can also insert themselves early in on the API design process, providing input about the shape and model of the API at a time when their input might get traction. Once the API gets coded by developers, it's hard enough to change a parameter name, much less the parameter itself.

Documentation hosting features on Stoplight

In addition to putting the OpenAPI specification at the center of the API lifecycle process, Stoplight has some other features of particular interest to technical writers. Stoplight offers a hosted docs solution, where you can integrate your conceptual content (the tutorials, guides, and other how-to's) with the reference API docs. Here's a sample page from Stoplight for a product called Shopgate ↗.

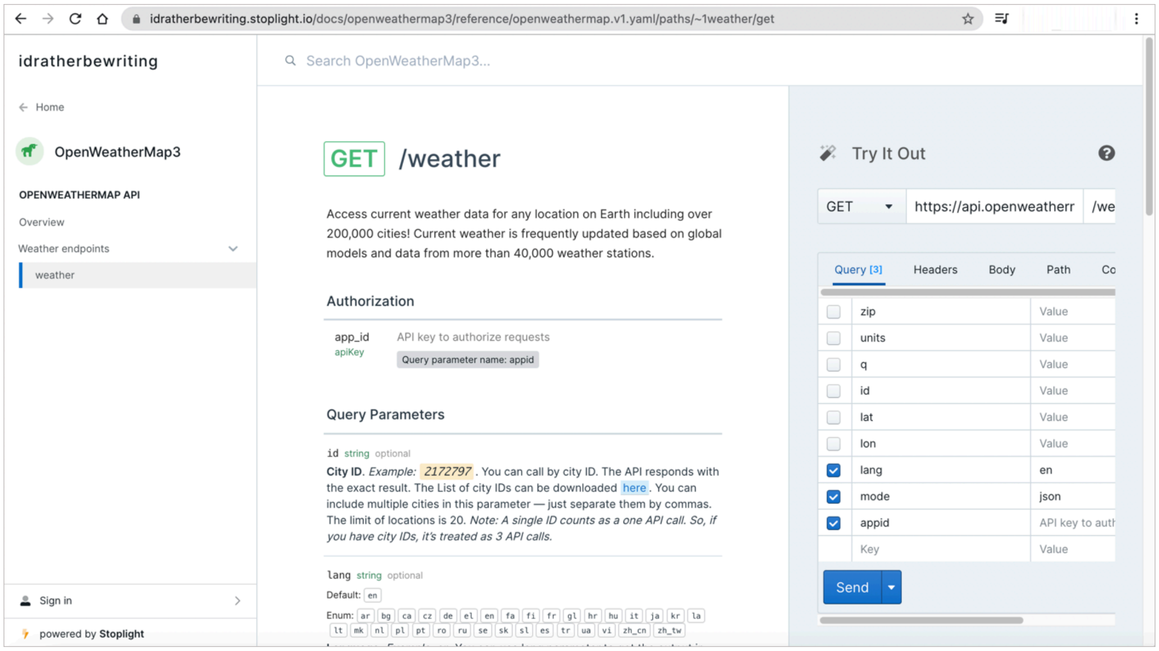
Note that you can integrate your reference documentation (p. 114) and conceptual documentation (p. 358) in Stoplight's hosted doc solution. Integrating these two content types has been a longtime challenge for tech writers in the API doc space. (For more on integration challenges with API documentation, see Integrating Swagger UI with the rest of your docs (p. 241).)

Although I generally like working directly in the code, I've found that Stoplight lets me focus more on the content and less on the details of the spec's format. Ideally, you can probably get developers and other project team members to populate reference content in Stoplight themselves since this is an activity that needs to happen much earlier in the API design process anyway.

If you're documenting an API, Stoplight and their hosted doc solution are worth checking out. But don't think of Stoplight as just a documentation platform or an easy way to generate the OpenAPI description. Consider Stoplight a way to design the single source of truth that will empower all other teams in a more efficient toward a successful API.

Stoplight demos

Here's a short demo of how Stoplight renders the OpenAPIWeatherMap OpenAPI specification. This Stoplight output uses the same API explored earlier in this course. Here's what the weather endpoint looks like in Stoplight:



GET /weather

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

Authorization

app_id
apiKey
API key to authorize requests
Query parameter name: appid

Query Parameters

`id` string optional
City ID. *Example:* *2172797* . You can call by city ID. The API responds with the exact result. The List of city IDs can be downloaded here. You can include multiple cities in this parameter — just separate them by commas. The limit of locations is 20. *Note: A single ID counts as a one API call. So, if you have city IDs, it's treated as 3 API calls.*

`lang` string optional
Default: `en`
Enum: `ar` `bg` `ca` `cz` `de` `el` `en` `fa` `fi` `fr` `gl` `hr` `hu` `it` `ja` `kr` `la` `lt` `mk` `nl` `pl` `pt` `ro` `ru` `se` `sk` `sl` `es` `tr` `ua` `vi` `zh_cn` `zh_tw`

GET https://api.openweatherr /we

☐	zip	Value
☐	units	Value
☐	q	Value
☐	id	Value
☐	lat	Value
☐	lon	Value
☑	lang	en
☑	mode	json
☑	appid	API key to auth
	Key	Value

Here are a few noteworthy callouts about the output:

- The conceptual information can live seamlessly alongside the reference content. You don't have a fragmented output with your documentation.
- The Try It Out pane is visually prominent and easy for users to access. This helps encourage more of exploratory, hands-on learning with your API. It is simple to work and try out a call.
- In the Responses section, the Schema and Example are placed on separate tabs that make it easy to toggle between. The schema presents the content in expand/collapse sections, allowing you to visually consume the whole structure at a glance.
- The output feels clean, with ample breathing space and an easy-to-read design. The UI isn't cluttered or busy. There's a good visual hierarchy with the design elements, encouraging focus in the middle pane where the main content lives.

This sample API doesn't include many endpoints, so it might be more instructive to see some other samples of Stoplight listed on their Case Studies page. For direct links to the documentation mentioned in case studies, explore the following:

- Zoom API docs
- Nutanix API docs
- MediaMath API docs
- AppointmentPlus API docs

- BigCommerce API docs ↗
- Namely API docs ↗

You can learn more about hosting your documentation with Stoplight here: Create Hosted API Documentation Online With Our API Docs Generator Tool ↗.

Stoplight tutorial

For an in-depth tutorial for using Stoplight to create an OpenAPI specification, including sample content and values, see Getting started tutorial: Using Stoplight Studio to create an OpenAPI specification document (p. 201).

Getting started tutorial: Using Stoplight Studio to create an OpenAPI specification document

You can create an OpenAPI specification document in several ways: (a) you can auto-generate it from annotations in your source code, (b) you can code it by hand in an editor such as the Swagger Editor, and (c) you can use a GUI editor such as Stoplight Studio. In this tutorial, we will walk through the third approach: using Stoplight Studio, a GUI editor for modeling your API. We will create a valid OpenAPI specification, using the OpenWeatherMap as the sample API.

For a higher-level introduction to the OpenAPI specification, see Introduction to the OpenAPI specification (p. 181). For a more conceptual introduction to Stoplight Studio, see Stoplight — visual modeling tools for creating your OpenAPI spec (p. 195).

Using a visual editor

Before we get started, let's address an initial question about the approach we're taking. When creating your OpenAPI specification document, why should you use a visual/GUI editor, which hides the code (unless you toggle the view open) instead of coding every detail of the spec by hand?

All approaches have their merits. But if you don't work with the OpenAPI specification enough to remember all the fields and syntax for each property, or if you're learning it for the first time, trying to get the syntax valid in the raw code can be challenging. Coding by hand leads to many errors and forces you to spend a lot of time troubleshooting invalid syntax.

Using an editor helps you avoid these errors and focus instead on the content. The less time you spend fiddling with syntax and more time working on content, the better the outcome of the documentation. For more on the philosophy of not getting lost in the syntactical details of the spec, but instead focusing on content, read this post by Phil Sturgeon, who says Stoplight Studio is "an absolute game changer for API Design, and something I've been waiting for since I found out about it somewhere in 2018" (Reinventing API Design with Stoplight Studio ↗).

As a technical writer who likes working in Markdown and other raw formats, I can see the appeal of working directly in the code. But the OpenAPI syntax is quite complex, and YAML is fussy. I'd rather be worrying more about the content than addressing syntax issues. But even so, Stoplight lets you switch into the code view to work in that mode when you want, and switch back into the visual view when you want. Changes you make in one mode sync to the other. This gives you the best of both worlds.

Before we begin

Before we begin the tutorial, note the following:

- Stoplight Studio isn't the only GUI editor for creating OpenAPI specifications. You can also use OpenAPI-GUI ↗, Apibldr ↗, SwaggerHub's visual editor ↗, and others. However, I think Stoplight's editor is one of the most mature and user-friendly.
- Although Stoplight has a commercial offering, their editor and documentation publishing are free (the free tier ↗ includes 1 project and 3 authors). But in general, just because you're using the Stoplight Studio editor here, it doesn't mean you're wed to their documentation output. You can

use the Stoplight Studio editor to create a valid specification, export it, and then choose the tool you want to use to render it (for example, Stoplight, Redoc, Spectacle, Swagger UI, or some other display framework).

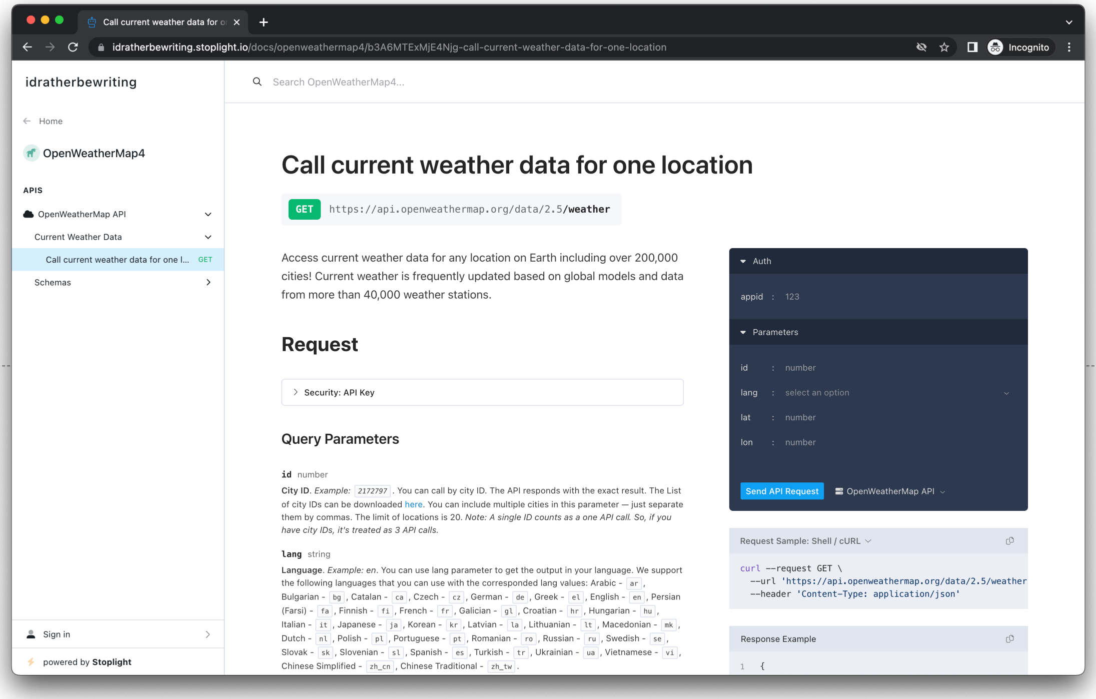
- Stoplight's platform gives you a nice path toward more comprehensive documentation, including not only API reference content but also conceptual documentation (p. 358) and tutorial content (p. 432). In short, you can create Markdown files similar to Stoplight Studio's documentation. Their platform lets you create a more seamless documentation experience for your readers across many different types of content, providing you with a complete publishing solution — not just a way to create reference content. (This tutorial will only cover creating API reference content, though.)
- This tutorial should take about 30 minutes to an hour. No special technical knowledge is required.

Disclaimer: Stoplight is one of the sponsors of my site. I would still recommend this approach regardless of any site sponsorship. It makes sense to use a visual editor when your content must conform to a highly technical syntax.

What you'll build

In this tutorial, we'll use sample information from the OpenWeatherMap API (more specifically, the Current weather data endpoint), which is the same API we've been using in other tutorials, such as the OpenAPI tutorial (p. 272).

Here's the documentation output that you'll build in this tutorial:


Call current weather data for o
idratherbewriting.stoplight.io/docs/openweathermap4/b3A6MTExMjE4Njg-call-current-weather-data-for-one-location
Incognito
idratherbewriting
Search OpenWeatherMap4...
Home
OpenWeatherMap4
APIS
OpenWeatherMap API
Current Weather Data
Call current weather data for one l... GET
Schemas
Sign in
powered by Stoplight
Auth
Parameters
Send API Request
OpenWeatherMap API
Request Sample: Shell / cURL
Response Example

Call current weather data for one location

GET `https://api.openweathermap.org/data/2.5/weather`

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

Request

Security: API Key

Query Parameters

`id` number

City ID. *Example:* `2172797`. You can call by city ID. The API responds with the exact result. The List of city IDs can be downloaded here. You can include multiple cities in this parameter — just separate them by commas. The limit of locations is 20. *Note: A single ID counts as a one API call. So, if you have city IDs, it's treated as 3 API calls.*

`lang` string

Language. *Example: en*. You can use lang parameter to get the output in your language. We support the following languages that you can use with the corresponded lang values: Arabic - `ar`, Bulgarian - `bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English - `en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `gl`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Korean - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `nl`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedish - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`, Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese Traditional - `zh_tw`.

appid : 123

id : number
lang : select an option
lat : number
lon : number

```
curl --request GET \
  --url 'https://api.openweathermap.org/data/2.5/weather
  --header 'Content-Type: application/json'
```

```
1  {
```

The tutorial has the following steps:

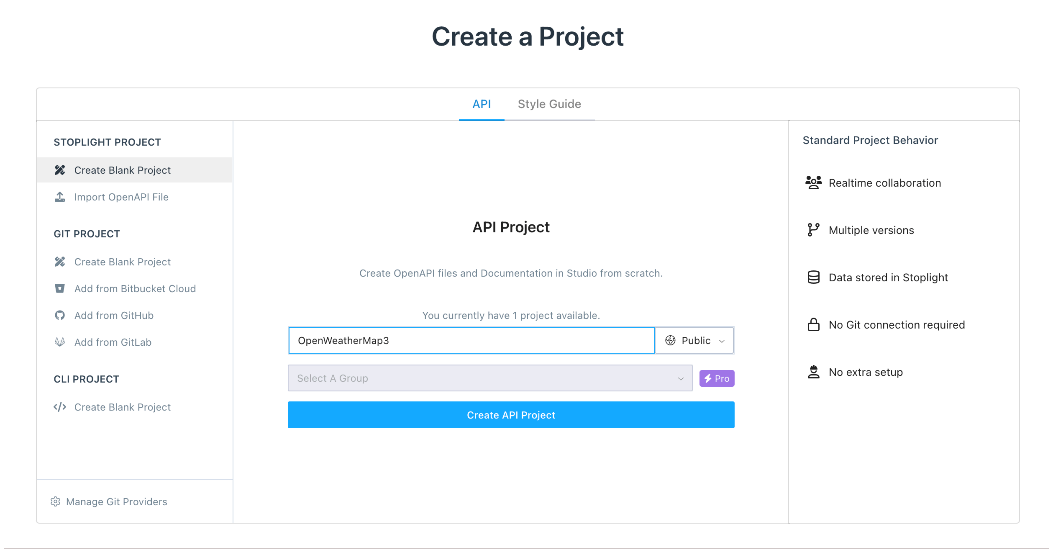
- Step 1: Set up a project in Stoplight Studio (p. 203)

- Step 2: Enter the API overview information (p. 206)
- Step 3: Enter the path and parameter information (p. 208)
- Step 4: Learn how to re-use parameters (p. 211)
- Step 5: Enter the responses and response schema information (p. 213)
- Step 6: Preview, test, and publish the output (p. 215)

Stoplight provides a variety of options for working with projects — desktop editor versus web editor, and syncing from GitHub or saving directly to a Stoplight workspace. In this tutorial, I take the simplest path: using the web editor and saving to the Stoplight workspace. But recognize that you have more options for how you approach projects. See Working with Projects for more details.

Step 1: Set up a project in Stoplight Studio

1. Go to https://stoplight.io/studio/ and click **On the Web**.
2. Follow the on-screen instruction to sign in, create a workspace, and then log in to the workspace.
3. Click the **+** button at the top to add a new project. Give the project a name, and then click **Create API Project**.

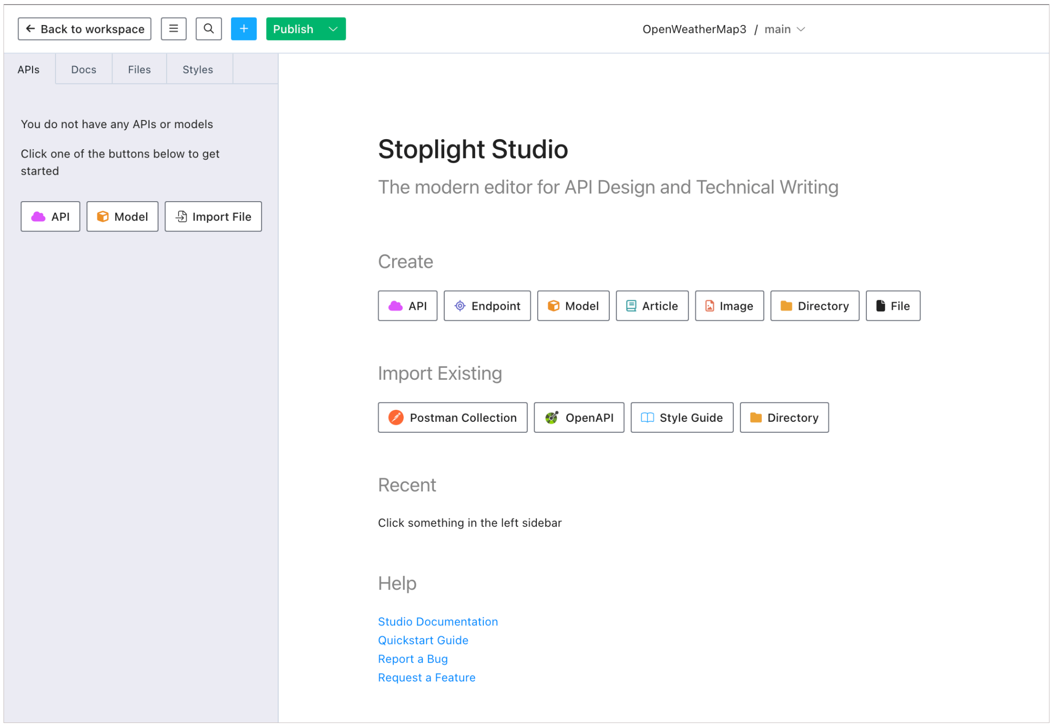


Create OpenAPI files and Documentation in Studio from scratch.

You currently have 1 project available.

OpenWeatherMap3

The Stoplight Studio editor opens. The left pane says, “You do not have any APIs or models.”



You do not have any APIs or models

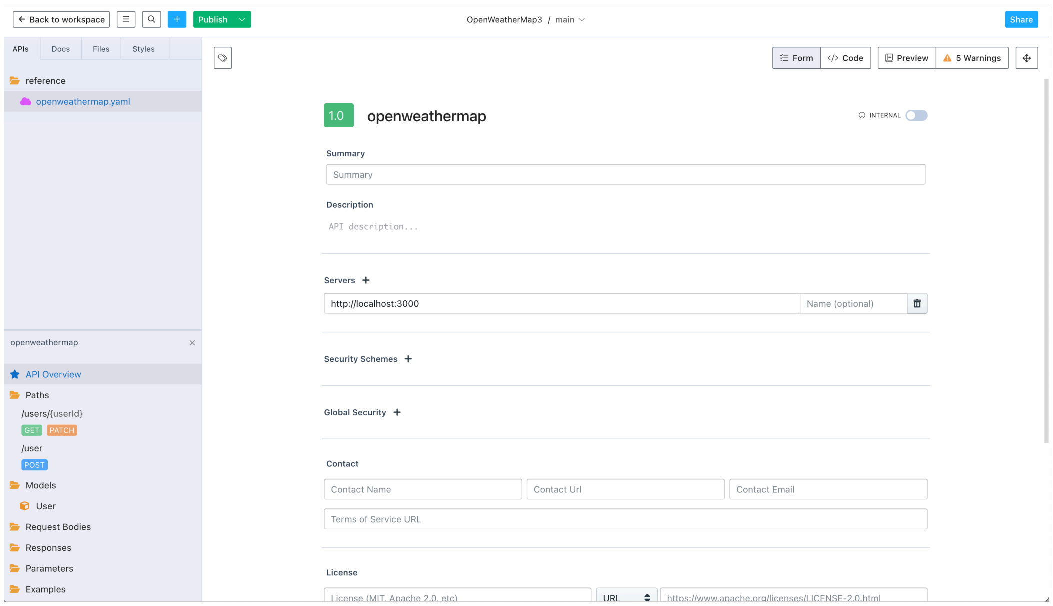
Click one of the buttons below to get started

The modern editor for API Design and Technical Writing

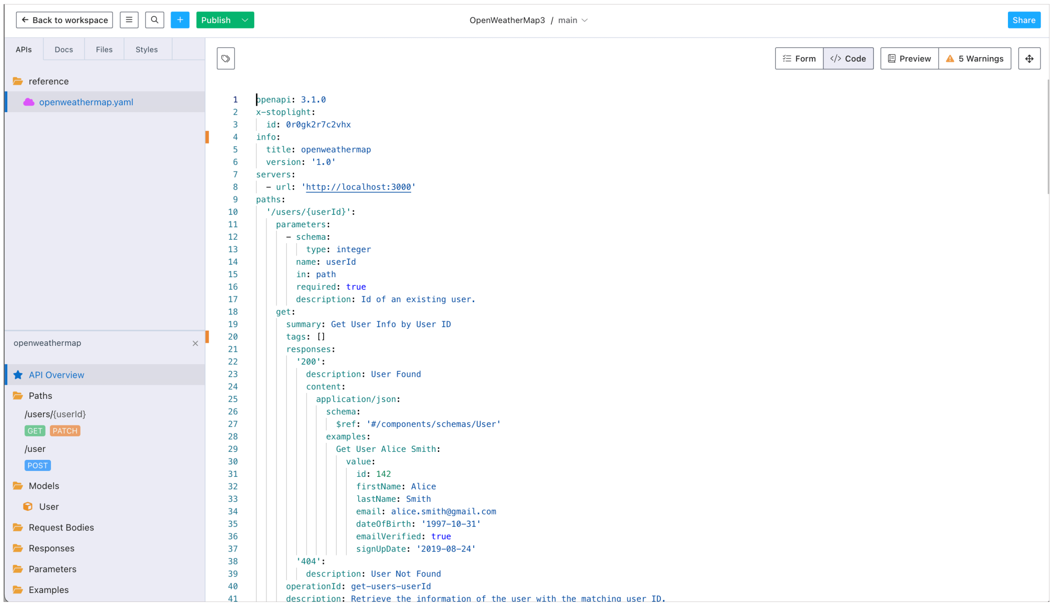
Click something in the left sidebar

4. Click the **API** button in the sidebar.
5. In the New API dialog box, name your API (e.g., “openweathermap”). By default, the editor will use OpenAPI v3.1 and YAML — leave the defaults as is. Click **Create**.

   Stoplight Studio creates an OpenAPI (OAS) specification file called openweathermap.yml and loads it as follows:



6. Stoplight lets you toggle between a form and code editor. The above screenshot shows the Form view. Click the **Code** button in the upper-right corner to see the code automatically created.



```
openapi: 3.1.0
x-stoplight:
  id: 0r0gk2r7c2vhx
info:
  title: openweathermap
  version: '1.0'
servers:
  - url: 'http://localhost:3000'
paths:
  '/users/{userId}':
    parameters:
      - schema:
          type: integer
        name: userId
        in: path
        required: true
        description: Id of an existing user.
    get:
      summary: Get User Info by User ID
      tags: []
      responses:
        '200':
          description: User Found
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
              examples:
                Get User Alice Smith:
                  value:
                    id: 142
                    firstName: Alice
                    lastName: Smith
                    email: alice.smith@gmail.com
                    dateOfBirth: '1997-10-31'
                    emailVerified: true
                    signUpDate: '2019-08-24'
        '404':
          description: User Not Found
      operationId: get-users-userId
      description: Retrieve the information of the user with the matching user ID.
```

The sample code includes some content for some fictitious user endpoints called `/users/{userid}` and `/user`.

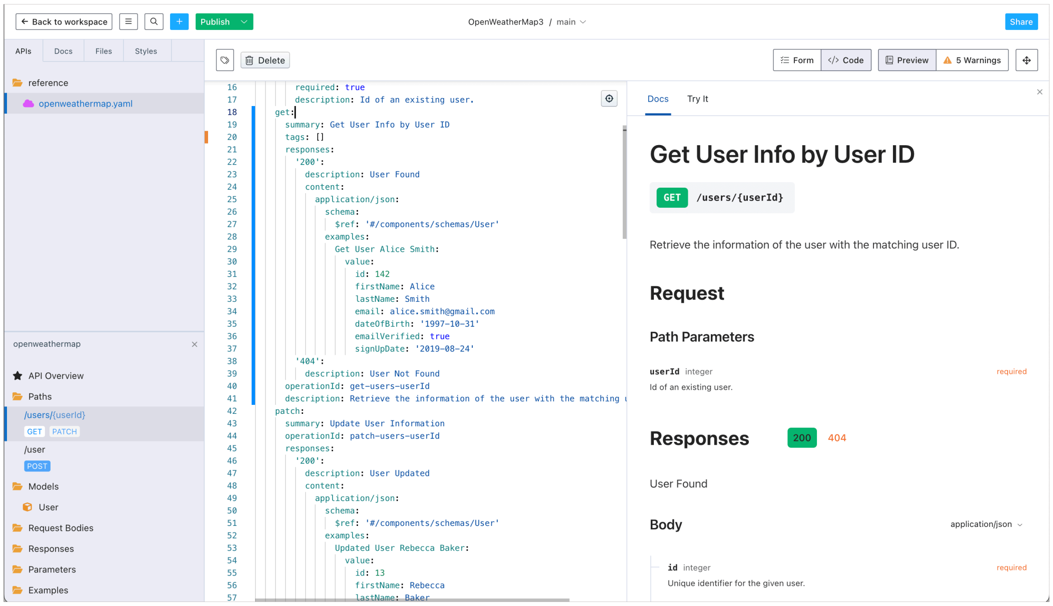
You could simply paste in a valid OpenAPI spec into the Code view, and it would populate the Form view. But presumably you don't already have an OpenAPI spec to paste into the code — this is why you're using the editor, to build out the specification document.

7. You can seamlessly switch between Form and Code views. Try adding a word in the Form view, switch to the Code view and find it, then remove it from the Code view and see how it updates in the Form view. Here's a short video clip showing this:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/pubapis_openapis_quickstart_stoplight.html] to view the content.

```
<frame width="560" height="315" src="https://www.youtube.com/embed/ngTq3aGZF
VA" title="YouTube video player" frameborder="0" allow="accelerometer; autop
lay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowf
ullscreen></iframe>
```

1. Select `users/{userId}` under the Paths folder, and then click the **Preview** button in the upper-right corner to see what the content would look like when published:



```
        required: true
        description: Id of an existing user.
  get:
    summary: Get User Info by User ID
    tags: []
    responses:
      '200':
        description: User Found
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
            examples:
              Get User Alice Smith:
                value:
                  id: 142
                  firstName: Alice
                  lastName: Smith
                  email: alice.smith@gmail.com
                  dateOfBirth: '1997-10-31'
                  emailVerified: true
                  signUpDate: '2019-08-24'
      '404':
        description: User Not Found
    operationId: get-users-userId
    description: Retrieve the information of the user with the matching 
  patch:
    summary: Update User Information
    operationId: patch-users-userId
    responses:
      '200':
        description: User Updated
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
            examples:
              Updated User Rebecca Baker:
                value:
                  id: 13
                  firstName: Rebecca
                  lastName: Baker
```

Get User Info by User ID

GET /users/{userId}

Retrieve the information of the user with the matching user ID.

Request

Path Parameters

userId integer required
Id of an existing user.

Responses 200 404

User Found

Body application/json

id integer required
Unique identifier for the given user.

The ability to switch views between Form, Code, and Preview gives you a lot of authoring power.

2. Click the **Form** button to switch back to the Form view.

Step 2: Enter the API overview information

In this section, you will populate the fields in the API Overview section, which include the version, name, description, security, contact, license, and other general details. For this tutorial, I've prepared sample information for you to easily insert into the Stoplight Editor. (Note: In the sample information in the orange expandable buttons, for fields that don't have names, I put the assumed titles of these fields in brackets.)

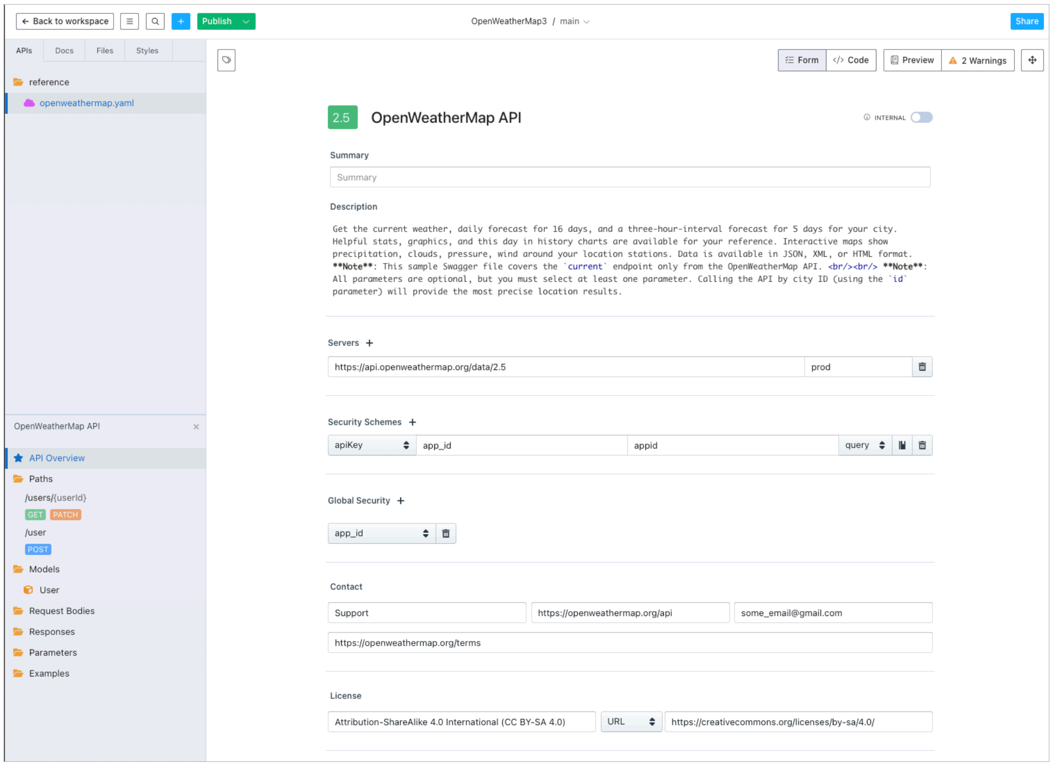
To populate the API overview:

1. In the Stoplight Studio sidebar, click **API Overview.**
2. Enter information in each of the form fields using information provided in the orange button below:

   This content contains JavaScript. Go to https://idratherbewriting.com/learnapidoc/pubapis_openapis_quickstart_stoplight.html to see it.

   Note: The Markdown source formatting in the sample text is intentional. When you copy and paste in the source, Stoplight will convert the Markdown into HTML when displaying the published page. Also note that Stoplight allows you to use Stoplight-flavored Markdown, which allows all the same tags as CommonMark but also includes some special tags for callouts, alerts, and other formatting.

   When finished, the form should look as follows:



2.5 OpenWeatherMap API

Summary

Summary

Description

```
Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city.
Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show
precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format.
**Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API. <br/><br/> **Note**:
All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id`
parameter) will provide the most precise location results.
```

Servers +

https://api.openweathermap.org/data/2.5 prod

Security Schemes +

apiKey app_id appid query

Global Security +

app_id

Contact

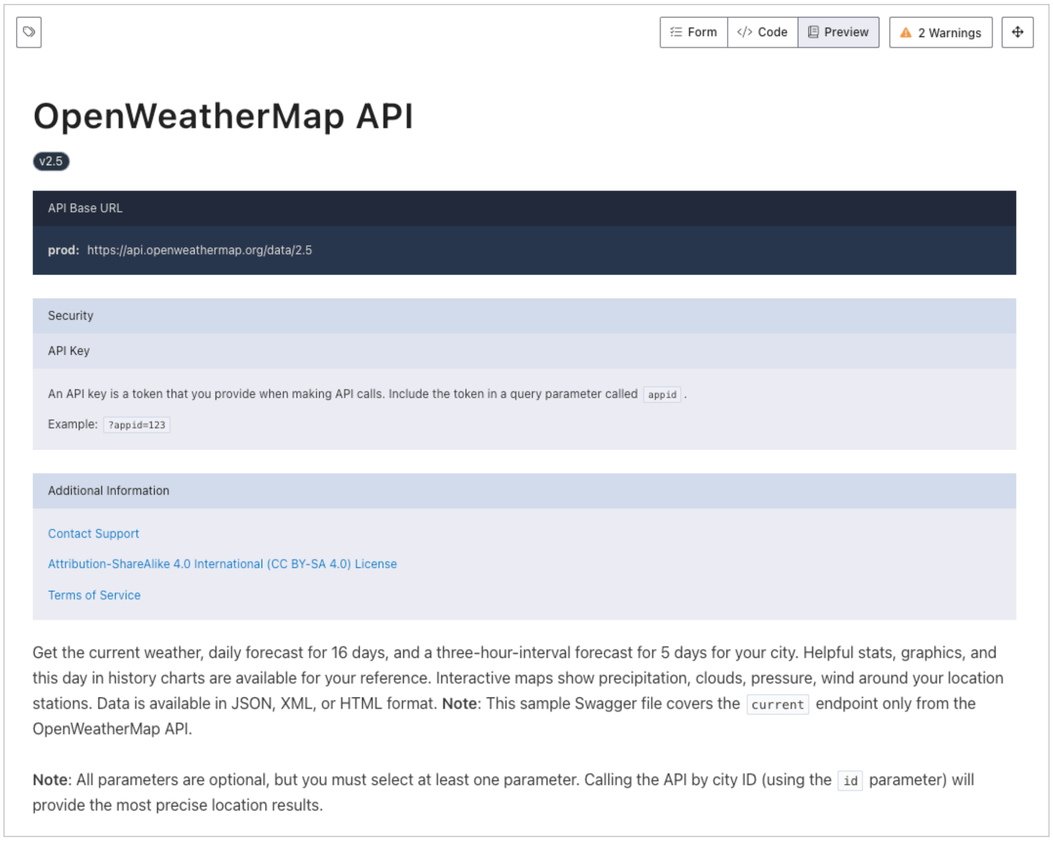
Support https://openweathermap.org/api some_email@gmail.com

https://openweathermap.org/terms

License

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) URL https://creativecommons.org/licenses/by-sa/4.0/

3. Now that you entered information for the first section, check out how it looks. Click the **Preview** button in the upper-right corner. It should look as follows:



OpenWeatherMap API

v2.5

API Base URL

prod: https://api.openweathermap.org/data/2.5

Security

API Key

An API key is a token that you provide when making API calls. Include the token in a query parameter called `appid`.

Example: `?appid=123`

Additional Information

Contact Support

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) License

Terms of Service

Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

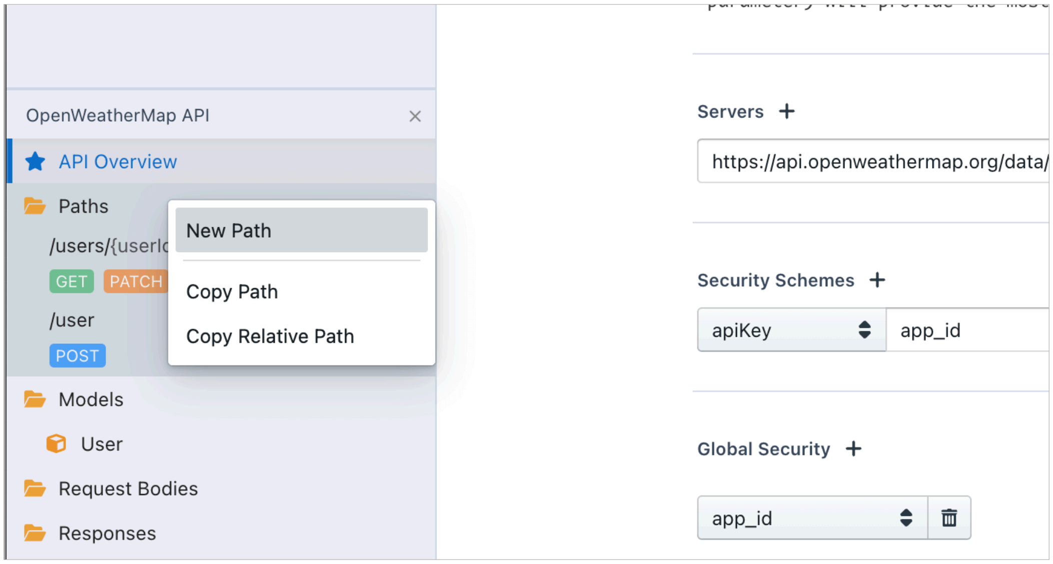
If you are seeing the Preview window side-by-side with the Form or Code view, you can make your browser window smaller so you can view the Preview window in full size.

4. Return to the Form view by clicking the **Form** button again.

Step 3: Enter the path and parameter information

Now enter the path and parameter information:

1. In the sidebar, right-click the **Paths** folder and select **New Path**.



Servers: https://api.openweathermap.org/data/

Security Schemes: apiKey app_id

Global Security: app_id

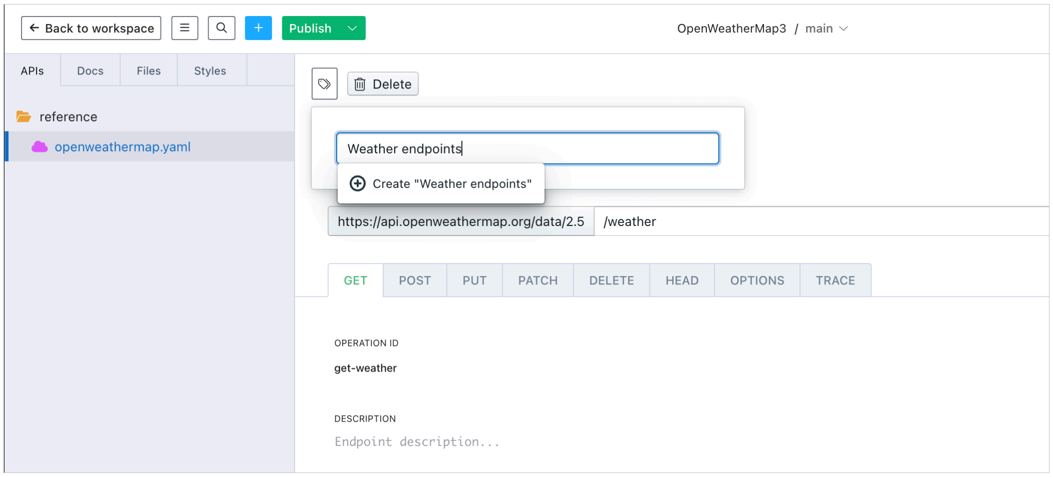
2. Type **weather** and press enter.

   By default Stoplight creates a GET operation for the weather endpoint. In our case, this works out great because this endpoint has a GET endpoint. But if not, you could just click the other operations available to create details for those operations.

   You can remove the default `/users/{userid}` and `/user` and endpoints. Right-click each endpoint and select **Delete Path**.

3. In the main window (not sidebar), change the path's title from "Your GET endpoint" to "weather."

4. Click the **Tags** button in the upper-left corner and add a tag called "Weather endpoints."



Weather endpoints

https://api.openweathermap.org/data/2.5 /weather

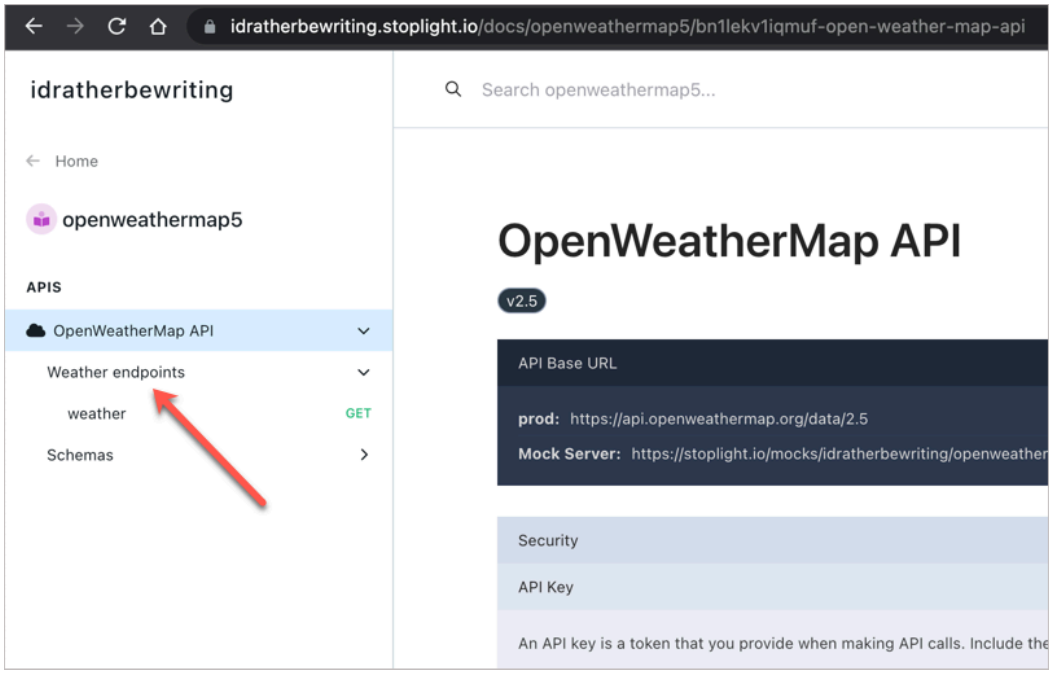
OPERATION ID

get-weather

DESCRIPTION

Endpoint description...

This tag will group the endpoints in the sidebar by this tag into a collapsed section called “Weather endpoints.” This is how you build navigation hierarchy for your endpoints. You won’t see this grouping until you publish your API, though. Here’s a screenshot of what that grouping will look like:



prod: https://api.openweathermap.org/data/2.5

Mock Server: https://stoplight.io/mocks/idratherbewriting/openweather

An API key is a token that you provide when making API calls. Include the

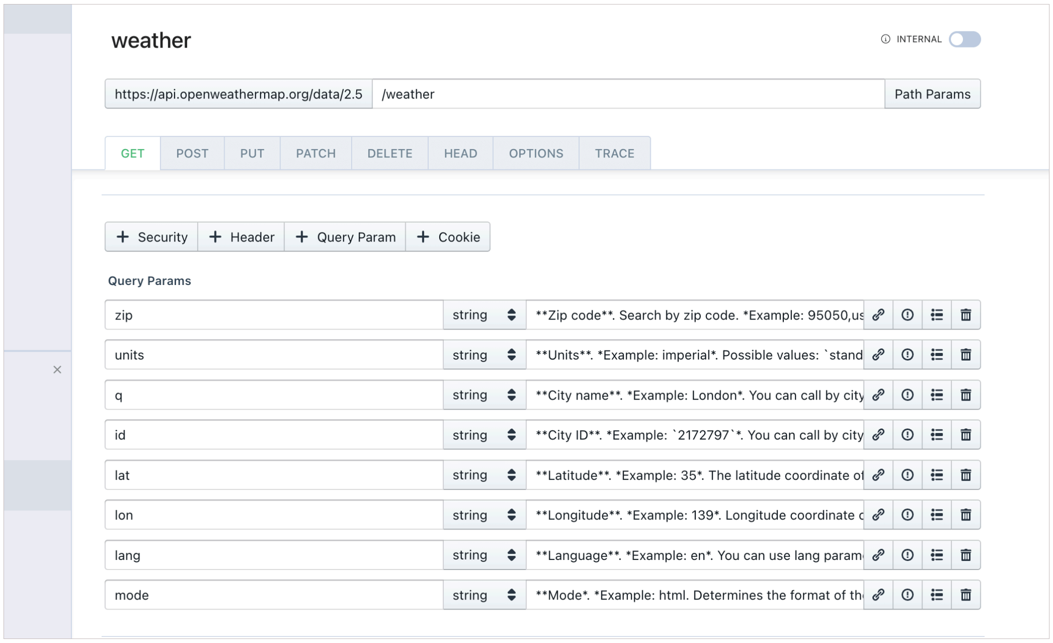
5. In the Description field, add the description for this endpoint. Expand the orange button below to get a description to paste here.

   This content contains JavaScript. Go to https://idratherbewriting.com/learnapidoc/pubapis_openapis_quickstart_stoplight.html to see it.

6. In Stoplight Studio, click the **+ Query Param** button to expand the query parameters options. (Note that our sample API has query parameters only, no path parameters.) Expand the orange button below for sample query parameter information. Then populate all the query parameter information into the Stoplight Studio editor.

   This content contains JavaScript. Go to https://idratherbewriting.com/learnapidoc/pubapis_openapis_quickstart_stoplight.html to see it.

   When you’re finished entering the parameters, Stoplight should look like this:



https://api.openweathermap.org/data/2.5 /weather

Query Params

zip	string	**Zip code**. Search by zip code. *Example: 95050,us
units	string	**Units**. *Example: imperial*. Possible values: `stand
q	string	**City name**. *Example: London*. You can call by city
id	string	**City ID**. *Example: `2172797`*. You can call by city
lat	string	**Latitude**. *Example: 35*. The latitude coordinate of
lon	string	**Longitude**. *Example: 139*. Longitude coordinate c
lang	string	**Language**. *Example: en*. You can use lang param
mode	string	**Mode*. *Example: html. Determines the format of th

You don't need to add security for this endpoint because you already configured global security in the previous section. However, if this endpoint had a unique security method, you could override the global security here by clicking the **+ Security** button.

Step 4: Learn how to re-use parameters

The OpenAPI spec allows to re-use parameters by storing the information in the `components` (p. 302) property. In the Stoplight Studio editor, you reuse parameters by entering the parameter information in the Parameters section in the sidebar. (In the code behind the scenes, this information gets stored in `components`.)

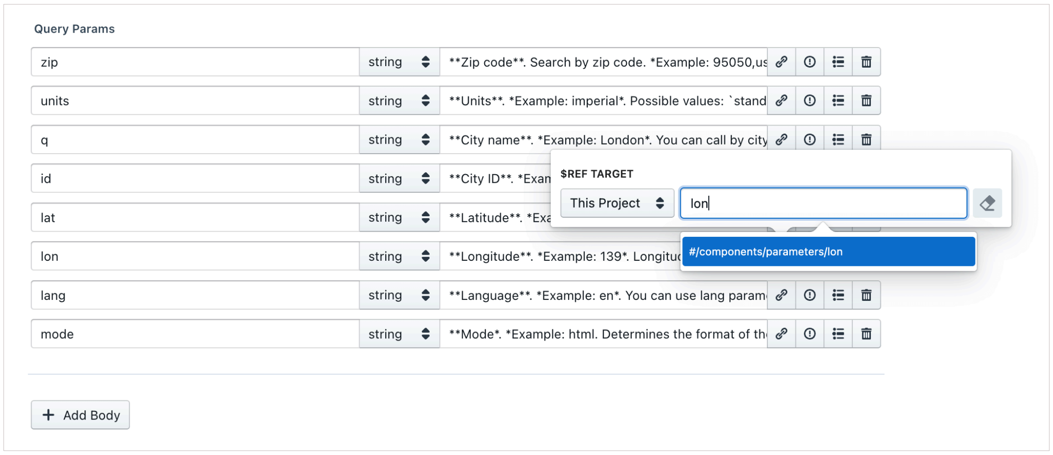
To understand how this works, let's re-use one of the query parameters.

To re-use a parameter:

1. Right-click the **Parameters** folder in the lower-left corner of the sidebar and select **New Query Parameter**.

   A parameter section similar to the query parameter sections you just populated appears.

2. Type **lon** and press enter.
3. Populate the `lon` parameter here using the same information as before.
4. Return to the previous query parameters section in the weather GET endpoint where you entered the `lon` parameter, and click the **Reference** button 🔗 for this parameter.
5. Search for "lon" and select it.



Query Params

zip	string	**Zip code**. Search by zip code. *Example: 95050,us
units	string	**Units**. *Example: imperial*. Possible values: `stand
q	string	**City name**. *Example: London*. You can call by city
id	string	**City ID**. *Exan
lat	string	**Latitude**. *Exa
lon	string	**Longitude**. *Example: 139*. Longituc
lang	string	**Language**. *Example: en*. You can use lang param
mode	string	**Mode*. *Example: html. Determines the format of th

$REF TARGET

This Project: lon

#/components/parameters/lon

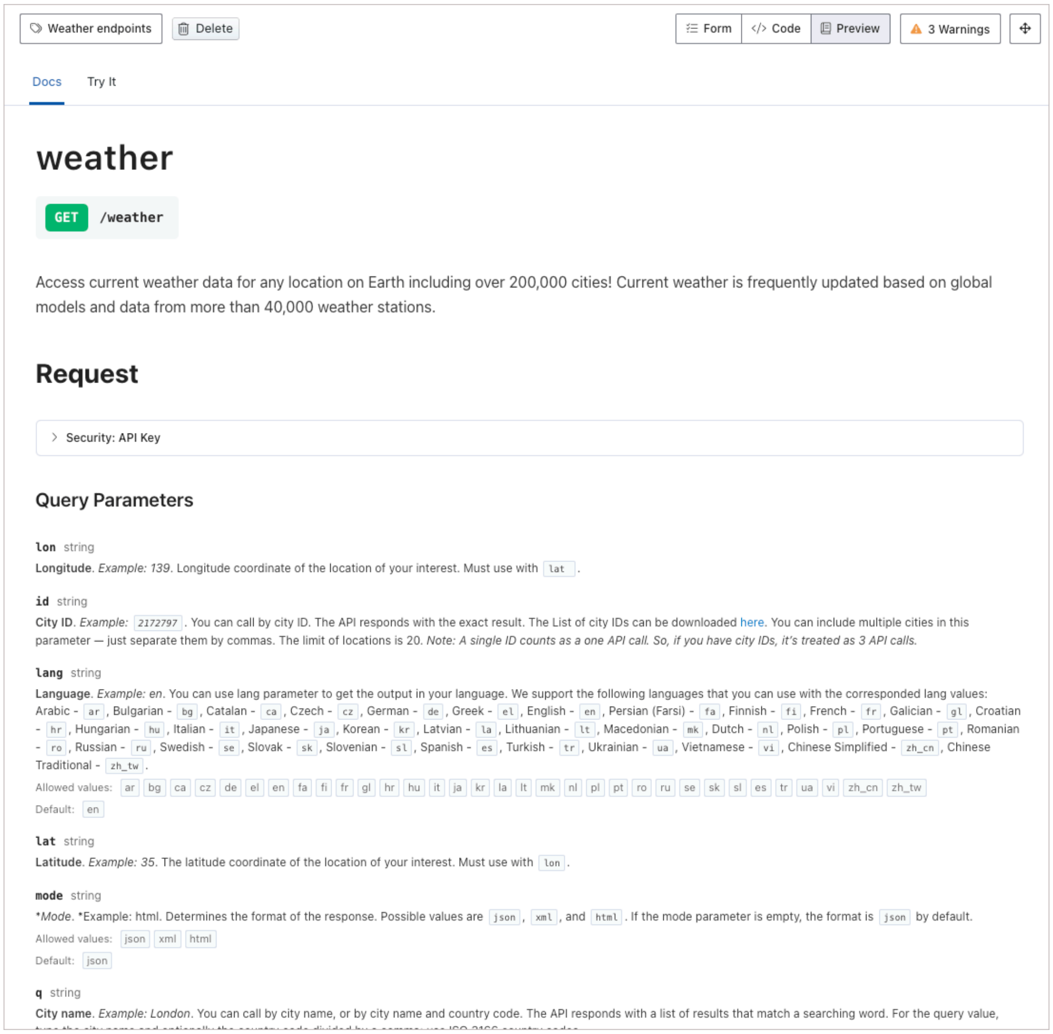
6. When you link the parameter to the reference like this, the icon turns red: 🔗

7. Switch to the Code editor and search for `lon`. You'll now see a reference to a component:

```
- $ref: '#/components/parameters/lon'
```

   You don't need to do anything in the code here — I'm just pointing out what changed behind the scenes. You can use this same technique to re-use other parameters as well as other types of content, such as responses.

8. Preview your updates by clicking the **Preview** button. If you select the /weather endpoint in the sidebar, the preview should look like this:



weather

GET /weather

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

Request

Security: API Key

Query Parameters

`lon` string

Longitude. *Example: 139*. Longitude coordinate of the location of your interest. Must use with `lat`.

`id` string

City ID. *Example:* `2172797`. You can call by city ID. The API responds with the exact result. The List of city IDs can be downloaded here. You can include multiple cities in this parameter — just separate them by commas. The limit of locations is 20. *Note: A single ID counts as a one API call. So, if you have city IDs, it's treated as 3 API calls.*

`lang` string

Language. *Example: en*. You can use lang parameter to get the output in your language. We support the following languages that you can use with the corresponded lang values: Arabic - `ar`, Bulgarian - `bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English - `en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `gl`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Korean - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `nl`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedish - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`, Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese Traditional - `zh_tw`.

Allowed values: ar bg ca cz de el en fa fi fr gl hr hu it ja kr la lt mk nl pl pt ro ru se sk sl es tr ua vi zh_cn zh_tw

Default: en

`lat` string

Latitude. *Example: 35*. The latitude coordinate of the location of your interest. Must use with `lon`.

`mode` string

**Mode*. *Example: html. Determines the format of the response. Possible values are `json`, `xml`, and `html`. If the mode parameter is empty, the format is `json` by default.

Allowed values: json xml html

Default: json

`q` string

City name. *Example: London*. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value,

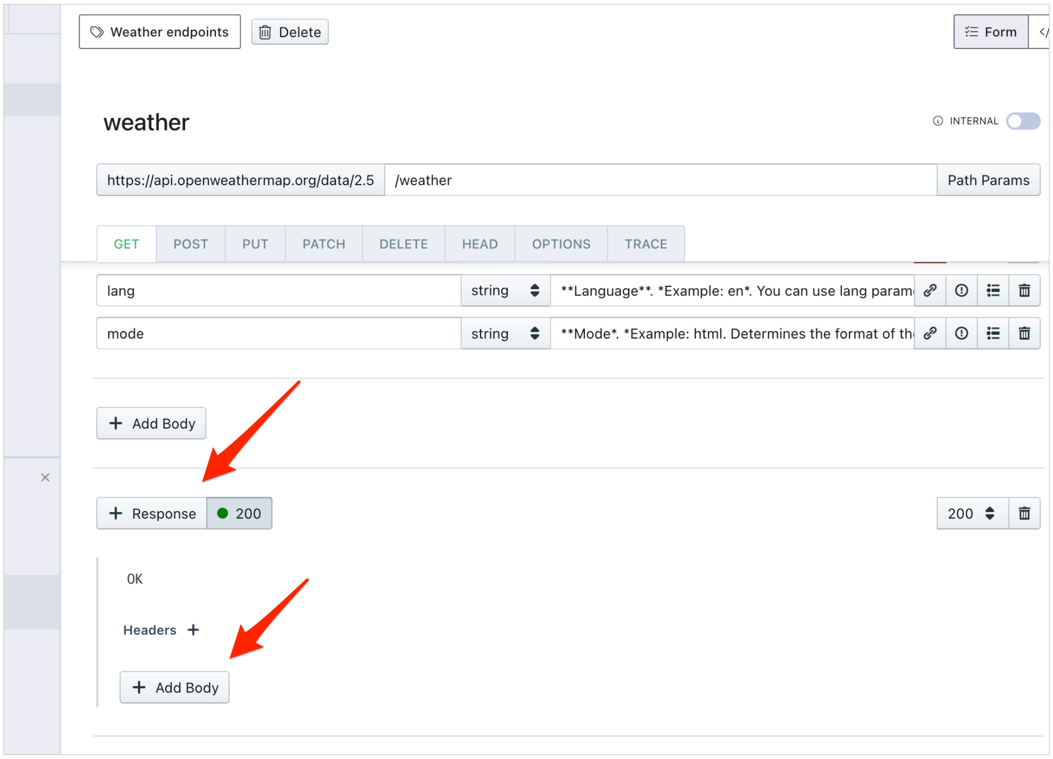
This endpoint in the OpenWeatherMap API doesn't have any path, header, or body parameters, so you can skip those elements.

Step 5: Enter the responses and response schema information

As a final step for this endpoint, you need to enter the response information. The response includes not only an example response but also the schema describing the response. The schema describes all possible values in the response, their data types, and whether the values are required.

To enter the response information in Stoplight Studio:

1. Below the parameter section for the endpoint, click the **+ Response** button. Then click the **+ Add Body** button that appears within this expanded Response section.



weather

https://api.openweathermap.org/data/2.5 /weather

lang string **Language**. *Example: en*. You can use lang param

mode string **Mode*. *Example: html. Determines the format of th

200

OK

2. On the Schema tab, click **Generate from JSON**.

   Here you'll experience one of the coolest features in the Stoplight Studio editor, and one reason why I like this editor so much. The editor will automatically build out the response schema from a sample JSON response.

3. Expand the orange button below and copy the sample response.

   This content contains JavaScript. Go to https://idratherbewriting.com/learnapidoc/pubapis_openapis_quickstart_stoplight.html to see it.

4. Paste the JSON you copied into the gray area where it says "Paste or write a JSON example below…" (Remove the empty curly braces `{}` first, and then paste in your sample.)

5. Click **Generate**. Stoplight builds the schema for you! Here's a short video clip showing this:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/pubapis_openapis_quickstart_stoplight.html] to view the content.

1. Add some more detail about the response elements:

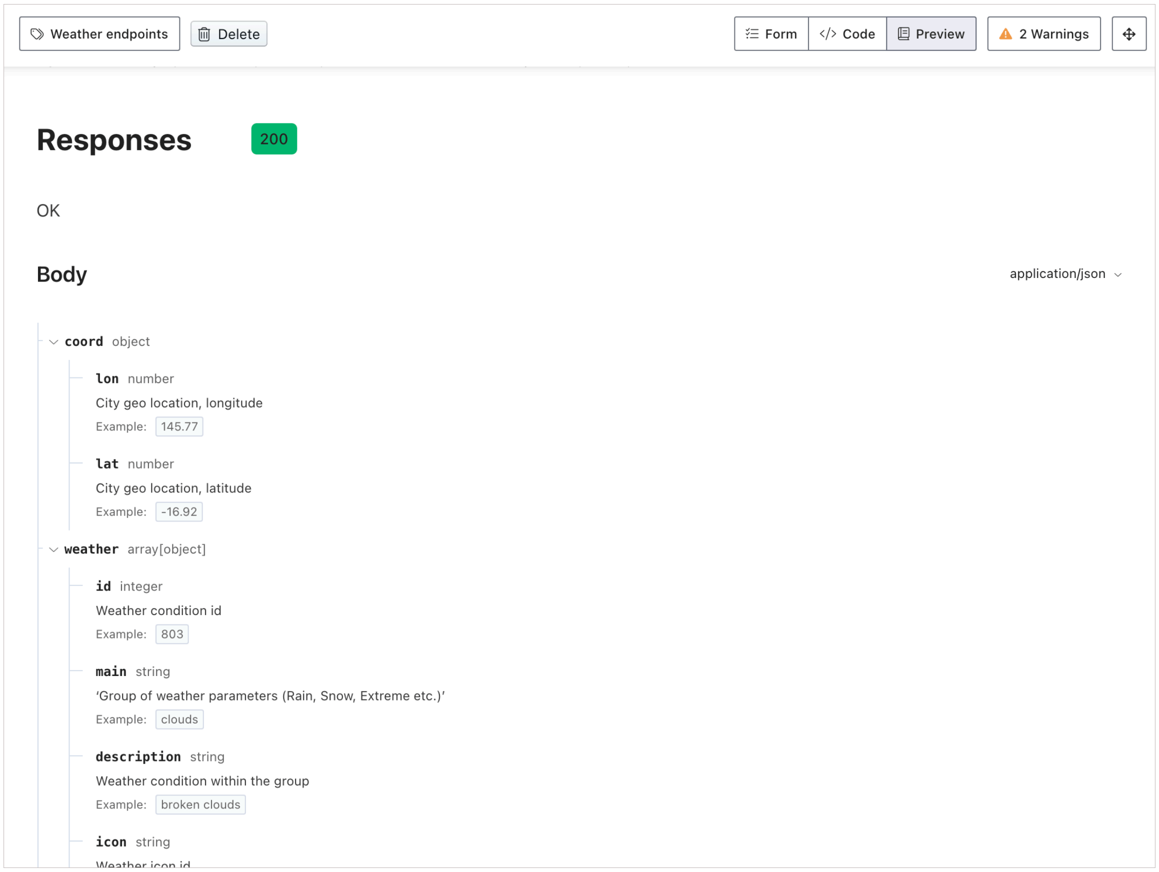
   - Click the **Description** button 📖 next to each property and add a description.

   - Click the **Other Properties** button ☰ and add an example in the **example** field.

   For the description and examples, expand the orange button below and copy over the information.

> This content contains JavaScript. Go to https://idratherbewriting.com/learnapidoc/pubapis_openapis_quickstart_stoplight.html to see it.

To view the responses output, you'll need to click **Preview** in Stoplight and view the rendered output. See my example here.



Responses 200

OK

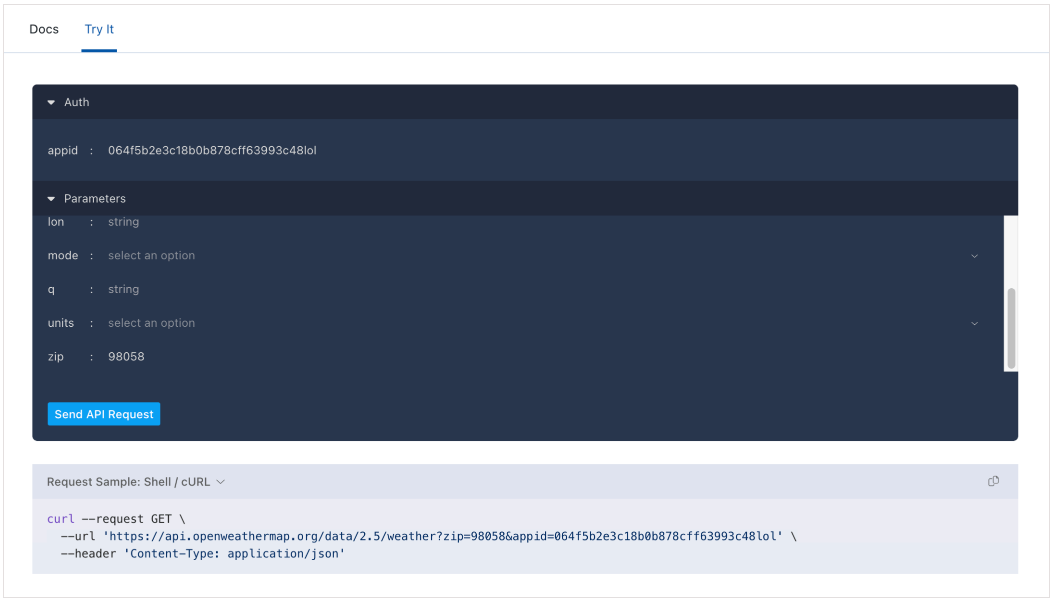
Body application/json

- **coord** object
  - **lon** number
    City geo location, longitude
    Example: 145.77
  - **lat** number
    City geo location, latitude
    Example: -16.92
- **weather** array[object]
  - **id** integer
    Weather condition id
    Example: 803
  - **main** string
    'Group of weather parameters (Rain, Snow, Extreme etc.)'
    Example: clouds
  - **description** string
    Weather condition within the group
    Example: broken clouds
  - **icon** string
    Weather icon id

Step 6: Preview, test, and publish the output

You've finished creating your OpenAPI spec, so let's preview, test, and publish test the documentation:

1. Click the **Preview** button in the upper-right corner.
2. In the sidebar, select the **/weather** endpoint.
3. Click the **Try It** tab near the top of the screen. (See note below — this feature won't work now due to CORS restrictions with the API.)
4. Enter your API key for the `appid` parameter. For details on getting the OpenWeatherMap API key, see Get an OpenWeatherMap API Key (p. 69). (You can also type out the API key pictured in the image below. It's not text because any API keys printed in text get exploited and then disabled by the vendor.)
5. Enter your zip code for the `zip` parameter (for example, 98058). You might have to scroll to find the zip parameter as they are sorted alphabetically. The screen should look like this:



appid : 064f5b2e3c18b0b878cff63993c48lol

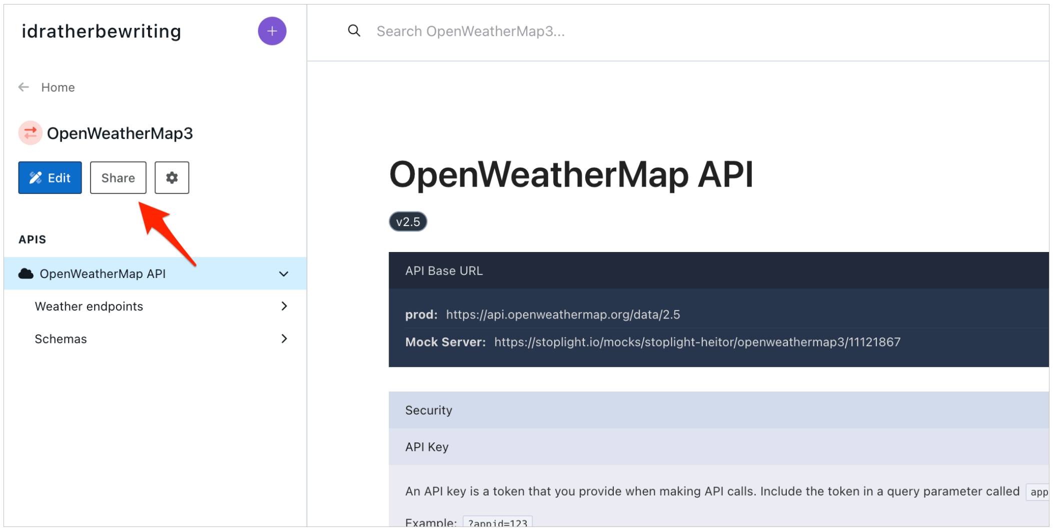
lon : string
mode : select an option
q : string
units : select an option
zip : 98058

```
curl --request GET \
  --url 'https://api.openweathermap.org/data/2.5/weather?zip=98058&appid=064f5b2e3c18b0b878cff63993c48lol' \
  --header 'Content-Type: application/json'
```

6. If you try to click **Send API Request**, you will get an error that says "Network Error occurred."

   You get this error because of CORS issues (restrictions from the API and browser with making requests on a non-authorized domain). This will unfortunately prevent you from getting a response from the API request. You can, however, copy and paste the cURL request sample to your terminal and run it, and that should give you a response from the OpenWeatherMap API. (This assumes you have curl installed. On a Mac, it's usually available by default.)

7. At this point, you can either publish or export your API documentation. To publish, click the **Publish** button.
8. To set the visibility:
   - Click on **Back to workspace** on the top-left to exit Stoplight Studio.
   - From your workspace, look at the admin sidebar on the left. Click the **Share** button and choose the visibility settings for your documentation (it is set to Public by default). You can also invite other team members from the share dialog.



OpenWeatherMap API

v2.5

API Base URL

prod: https://api.openweathermap.org/data/2.5

Mock Server: https://stoplight.io/mocks/stoplight-heitor/openweathermap3/11121867

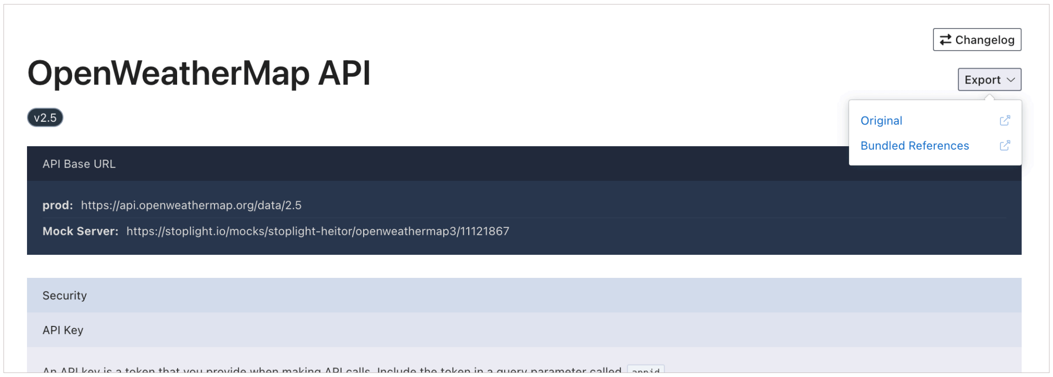
Security

API Key

An API key is a token that you provide when making API calls. Include the token in a query parameter called `app`

Example: `?appid=123`

9. If you don't want to publish docs through Stoplight, you can export your OpenAPI spec file from Stoplight. From the workspace home, click the **Export** button and choose the type of code you want:



prod: https://api.openweathermap.org/data/2.5

Mock Server: https://stoplight.io/mocks/stoplight-heitor/openweathermap3/11121867

The Bundled References will resolve any `$ref` code and place them in the components section so there's no duplication. The Original will export any `$ref` code, but not resolve them. The option you choose depends on what's supported in the editor you're bringing it to.

For publishing options outside of Stoplight, you can use your OpenAPI definition with Redoc, Swagger UI (p. 218), Spectacle, Docsy, and other tools. See Openapi.tools for a list of possibilities. You can find a Redoc tutorial in this course here (p. 249).

You can view my OpenWeatherMap API documentation on Stoplight here. If you got stuck in the tutorial somewhere along the way, you can click Export to download my code and paste it into your code. For various demos of Stoplight with different API docs, see Stoplight demos (p. 199).

Stoplight has several pricing tiers. The free version supports up to 3 users and 1 project, while the next level up, Starter, supports up to 3 users and 10 projects.

Swagger UI tutorial

Swagger UI provides a display framework that reads an OpenAPI specification document and generates an interactive documentation website. The following tutorial shows you how to integrate an OpenAPI specification document into Swagger UI.

For a more conceptual overview of OpenAPI and Swagger, see Introduction to the OpenAPI specification (p. 181), or see this article I wrote for ISTC a few years ago: Implementing Swagger with API docs (PDF). For a step-by-step tutorial on creating an OpenAPI specification document, see the OpenAPI tutorial (p. 272). Or for an easier approach using a visual editor to create the OpenAPI spec, see Getting started tutorial: Using Stoplight Studio to create an OpenAPI specification document (p. 201).

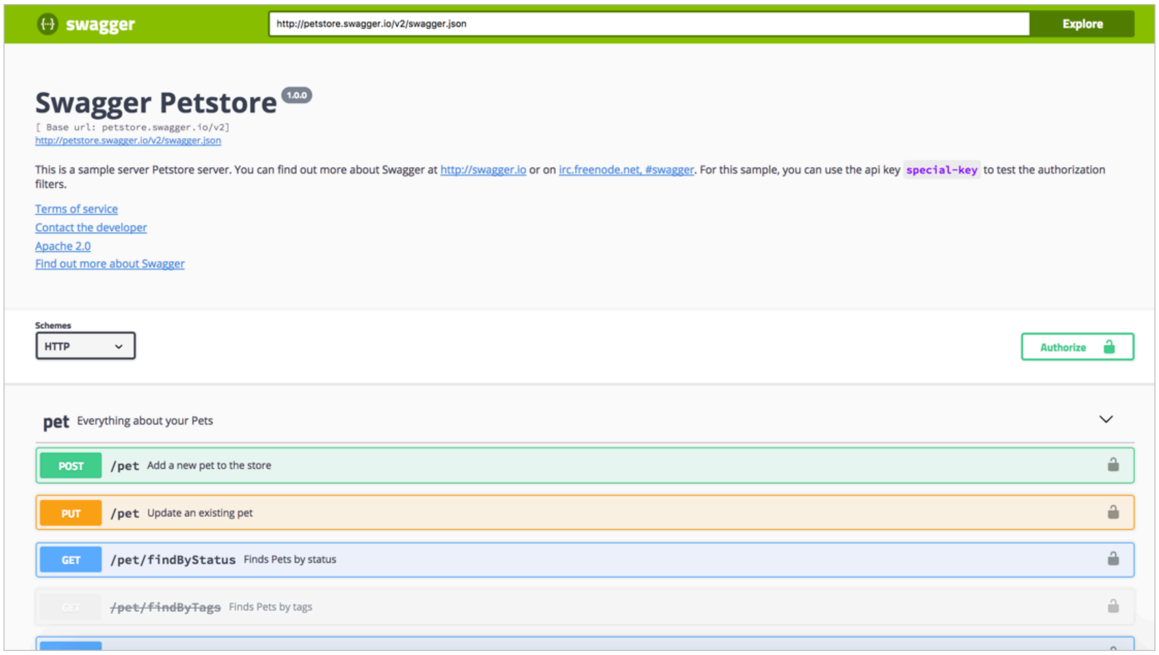
Swagger UI overview

Swagger UI is one of the most popular tools for generating interactive documentation from your OpenAPI document. Swagger UI generates an interactive API console for users to quickly learn about your API and experiment with requests. Additionally, Swagger UI (which is an actively managed project with an Apache 2.0 license) supports the latest version of the OpenAPI spec (3.x) and integrates with other Swagger tooling.

For definitions of common terms, see Key terms (p. 228) at the end of this article.

Get familiar with Swagger UI through the Petstore demo

To get a better understanding of Swagger UI, let's explore the Swagger Petstore example. In the Petstore example, the site is generated using Swagger UI.



Swagger Petstore 1.0.0

[Base url: petstore.swagger.io/v2]

http://petstore.swagger.io/v2/swagger.json

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

swagger
http://petstore.swagger.io/v2/swagger.json
Explore
Terms of service
Contact the developer
Apache 2.0
Find out more about Swagger
Schemes
HTTP
Authorize
pet Everything about your Pets
POST /pet Add a new pet to the store
PUT /pet Update an existing pet
GET /pet/findByStatus Finds Pets by status
GET ~~/pet/findByTags~~ Finds Pets by tags

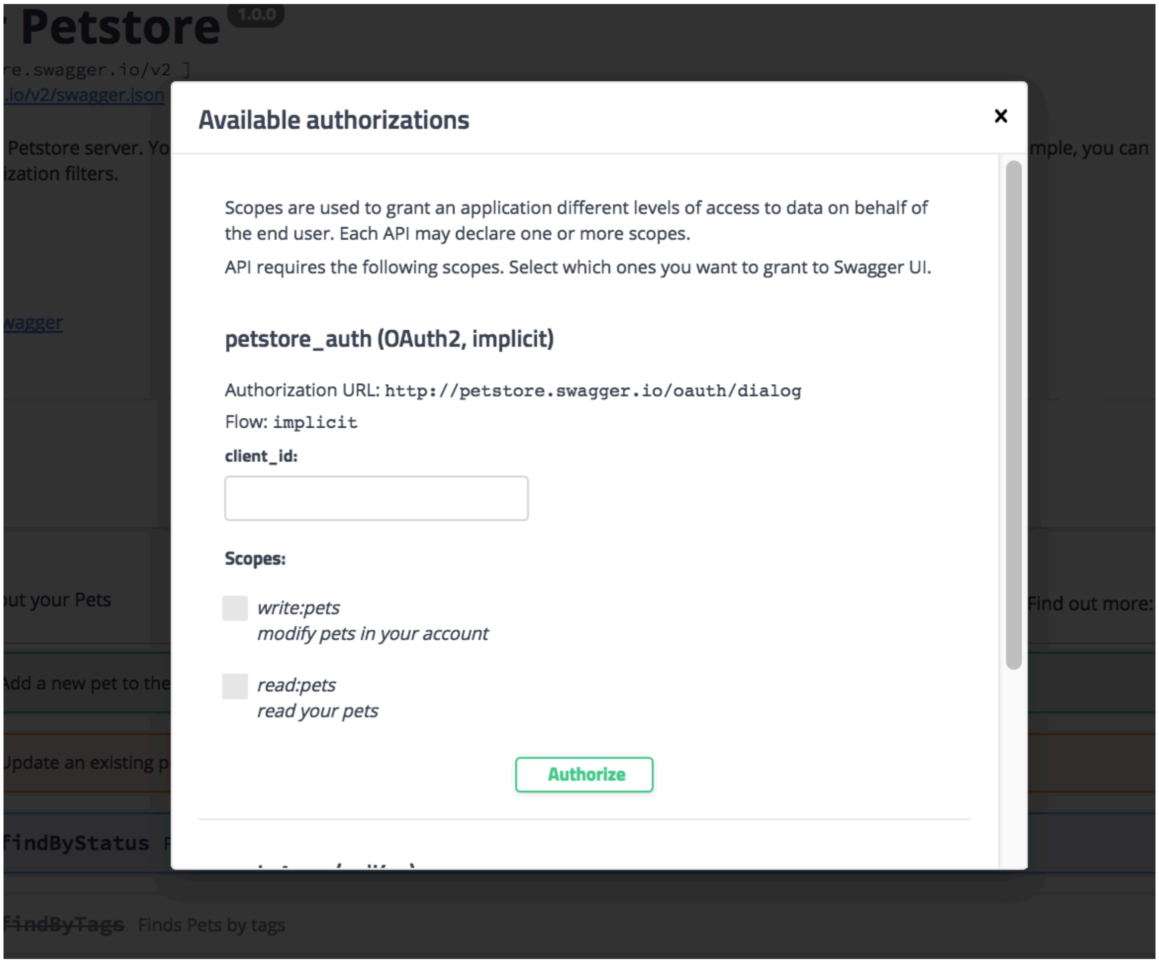
The endpoints are grouped as follows:

- pet

- store
- user.

Authorize your requests

Before making any requests, you would normally authorize your session by clicking the **Authorize** button and completing the information required in the Authorization modal pictured below:



Available authorizations

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.

API requires the following scopes. Select which ones you want to grant to Swagger UI.

petstore_auth (OAuth2, implicit)

Authorization URL: `http://petstore.swagger.io/oauth/dialog`

Flow: `implicit`

client_id:

Scopes:

☐ *write:pets*
modify pets in your account

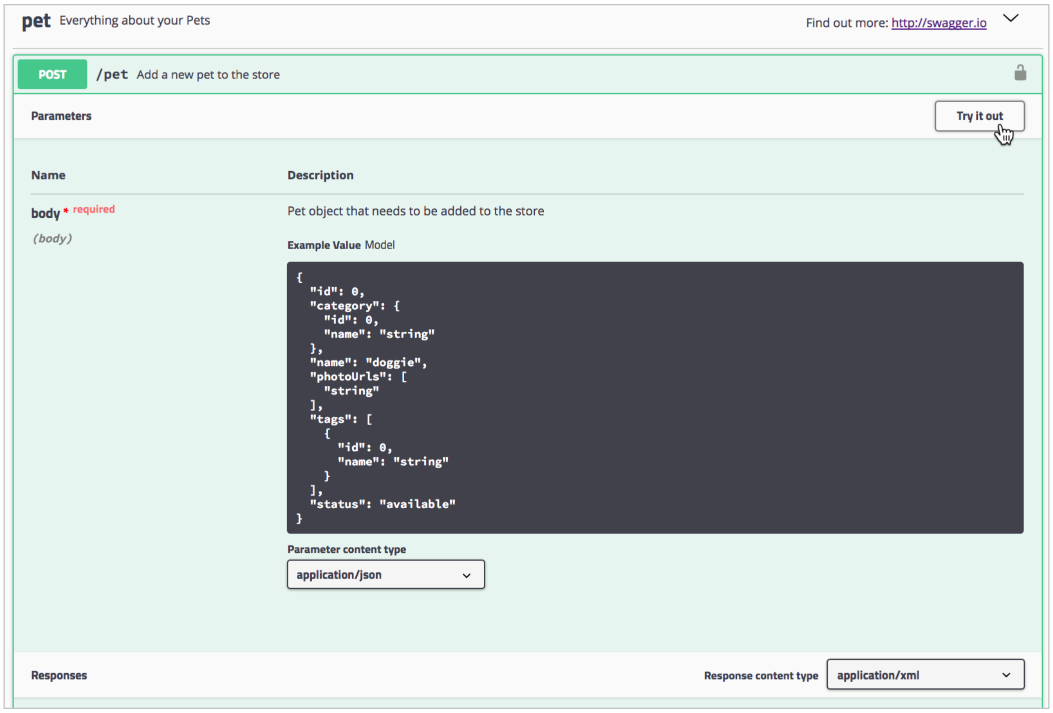
☐ *read:pets*
read your pets

The Petstore example has an OAuth 2.0 security model. However, the authorization code is just for demonstration purposes. There isn't any real logic authorizing those requests, so you can simply close the Authorization modal.

Make a request

Now let's make a request:

1. Expand the **POST Pet** endpoint.
2. Click **Try it out**.



Pet object that needs to be added to the store

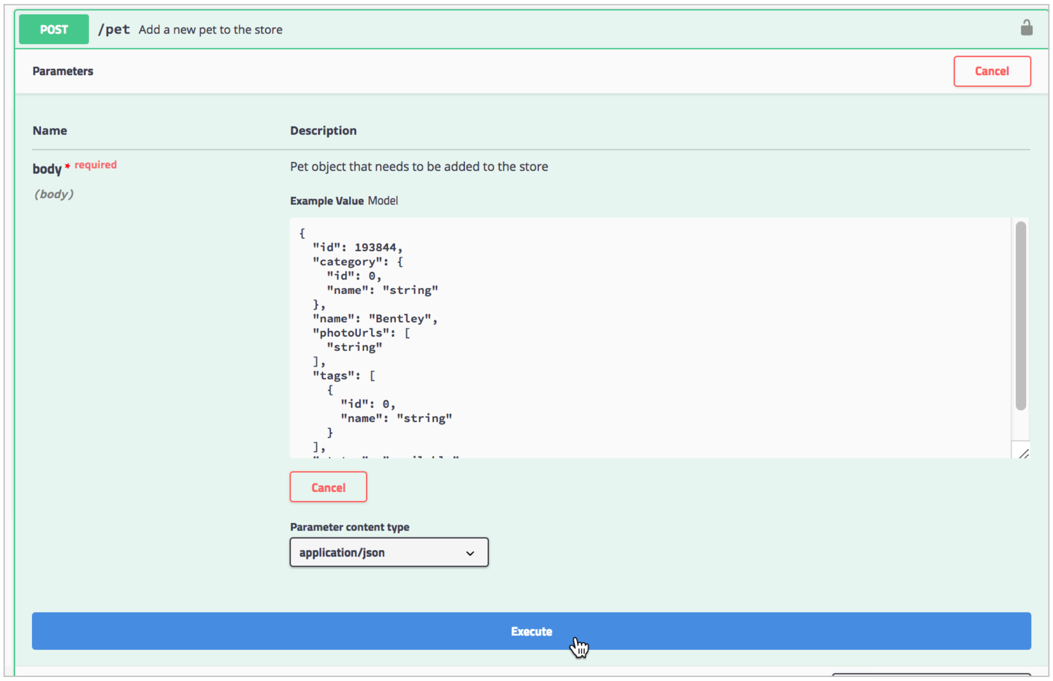
```
{
  "id": 0,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

Parameter content type: application/json

Response content type: application/xml

After you click Try it out, the example value in the Request Body field becomes editable.

3. In the Example Value field, change the first `id` value to a random integer, such as `193844`. Change the second `name` value to something you'd recognize (your pet's name).
4. Click **Execute**.



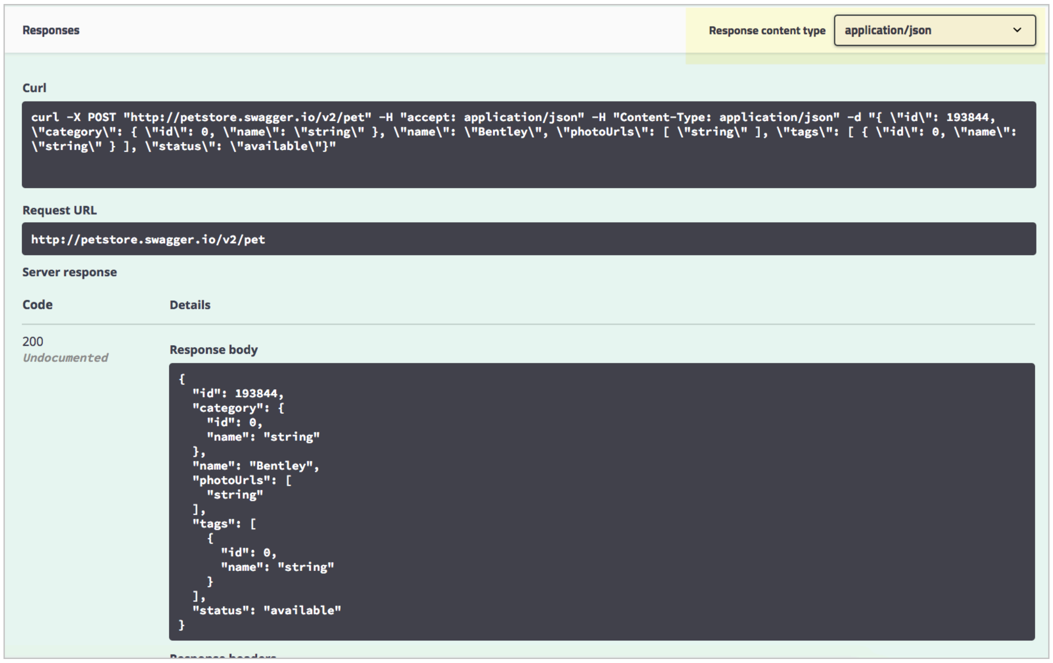
Pet object that needs to be added to the store

```
{
  "id": 193844,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "Bentley",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
```

Parameter content type: application/json

Executing a sample Petstore request

Swagger UI submits the request and shows the curl (p. 80) that was submitted. The Responses section shows the response (p. 154). (If you select JSON rather than XML in the "Response content type" drop-down box, the response's format will be shown in JSON.)



Curl

```
curl -X POST "http://petstore.swagger.io/v2/pet" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"id\": 193844,
\"category\": { \"id\": 0, \"name\": \"string\" }, \"name\": \"Bentley\", \"photoUrls\": [ \"string\" ], \"tags\": [ { \"id\": 0, \"name\":
\"string\" } ], \"status\": \"available\"}"
```

Request URL

```
http://petstore.swagger.io/v2/pet
```

Server response

Code: 200 *Undocumented*

Response body

```
{
  "id": 193844,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "Bentley",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

> **Important:** The Petstore is a functioning API, and you have actually created a pet. You now need to take responsibility for your pet and begin feeding and caring for it! All joking aside, most users don't realize they're playing with real data when they execute responses in an API (especially when using their own API key). This test data may be something you have to wipe clean when you transition from exploring and learning about the API to eventually using the API for production use.

Verify that your pet was created

1. Expand the **GET /pet/{petId}** endpoint.
2. Click **Try it out**.
3. Enter the pet ID you used in the previous operation. (If you forgot it, look back in the **POST Pet** endpoint to check the value.)
4. Click **Execute**. You should see your pet's name returned in the Response section.

Some sample Swagger UI doc sites

Before we get into this Swagger tutorial with another API (other than the Petstore demo), check out a few Swagger implementations:

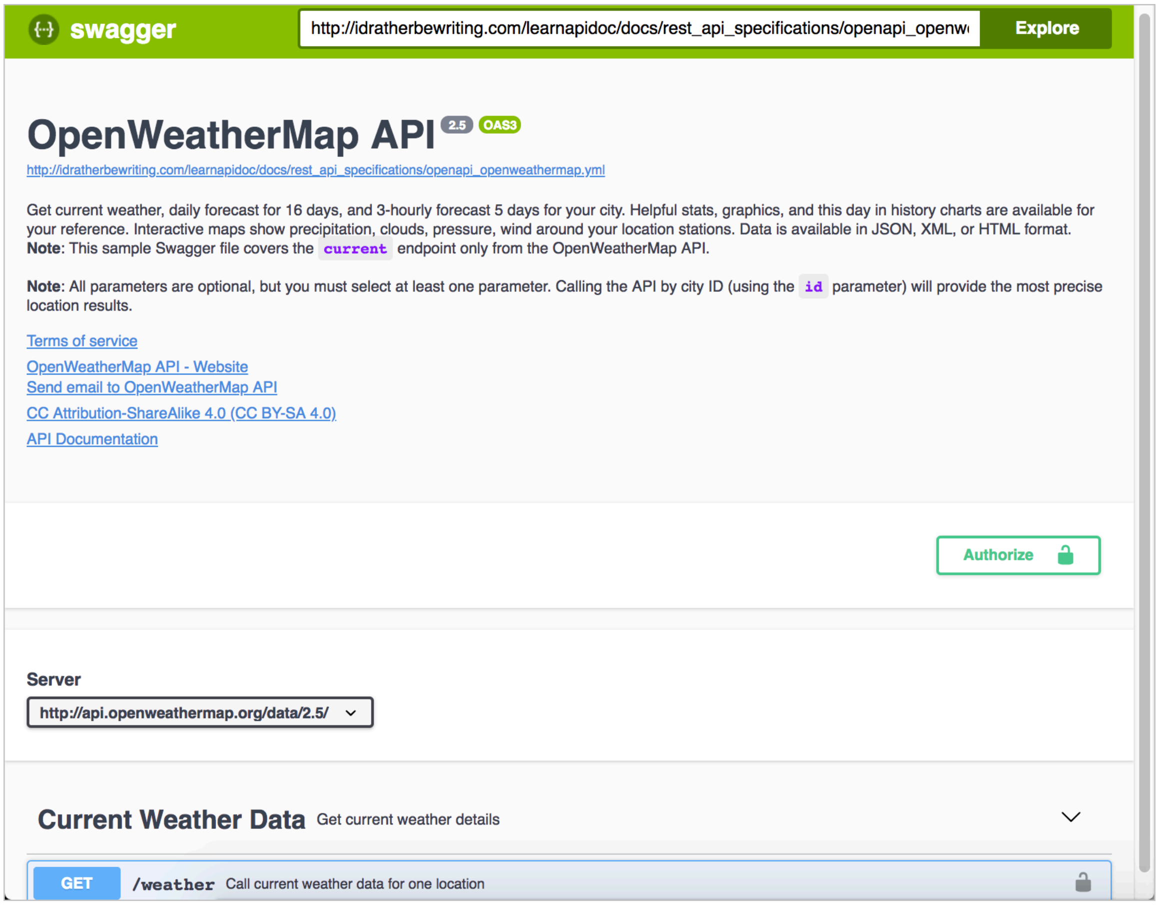
- Reverb
- VocaDB
- IBM Watson IoT Admin REST APIs
- The Movie Database API

Some of these sites look the same, but others, such as The Movie Database API and Zomato, have been integrated seamlessly into the rest of their documentation website.

Looking at the examples, you'll notice the documentation is short and sweet in a Swagger implementation. This brevity is because the Swagger display is meant to be an interactive experience where you can try out calls and see responses — using your own API key to see your own data. It's the learn-by-doing-and-seeing-it approach. Also, Swagger UI only covers the reference topics (p. 114) of your documentation. The conceptual topics (p. 358) are usually covered in a separate guide.

Activity: Create a Swagger UI display with an OpenAPI spec document

In this activity, you'll create a Swagger UI display for an OpenAPI specification document. If you're using one of the pre-built OpenAPI files, you can see a demo of what we'll build here: OpenWeatherMap Swagger UI.


swagger
http://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_openw
Explore

OpenWeatherMap API 2.5 OAS3

http://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_openweathermap.yml

Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

Terms of service

OpenWeatherMap API - Website

Send email to OpenWeatherMap API

CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)

API Documentation

Server: http://api.openweathermap.org/data/2.5/

Current Weather Data Get current weather details

GET `/weather` Call current weather data for one location

Demo of Swagger UI rendering an OpenWeatherMap OpenAPI specification document

First, you'll make sure you can view Swagger locally. Then you'll switch the Petstore OpenAPI document URL with an OpenWeatherMap OpenAPI document URL.

1. Go to the Swagger UI GitHub project.
2. Click **Code**, and then click **Download ZIP**. Download the files to a convenient location on your computer and extract the files.

The only folder you'll be working with in the downloaded zip is the `dist` folder (short for distribution). Everything else is used only if you're recompiling the Swagger files, which is beyond the scope of this tutorial. (If desired, you can drag the `dist` folder out of the `swagger-ui-master` folder so that it stands alone.)

3. In your Chrome browser, press **Cmd+O** (Mac) or **Ctrl+O** (Windows), browse to the `dist` folder, and select the `swagger-initializer.js` file,

   You should see the Petstore Swagger content. Now you'll customize the OpenAPI spec file with another file.

4. Inside your `dist` folder, open `swagger-initializer.js` in a text editor such as Sublime Text ↗.
5. Look for the following code:

```
url: "https://petstore.swagger.io/v2/swagger.json",
```

6. Change the `url` value to an online web URL to your Swagger file. For example: `https://idratherbewriting.com/docs/openapi_spec_and_generated_ref_docs/openapi_openweathermap.yml`. Then save the file.

   If the `url` reference isn't to an *online* URL (but rather to a local file), Swagger UI will create an CORS (cross-origin resource sharing) error. To view Swagger UI with a local OpenAPI file, you can run a simple Python server locally ↗ to simulate a web server (this requires you to install Python).

7. Refresh the `index.html` file in your Chrome browser. The content should show the OpenWeatherMap API content instead of Swagger Petstore content.

   When you're ready to publish your Swagger UI file, you just upload the `dist` folder (or whatever you want to call it) to a web server and go to the `index.html` file. For example, if you called your directory `dist` (leaving it unchanged), you would go to `http://myserver.com/dist/`.

   For more instructions in working with Swagger UI, see the Swagger.io docs ↗.

Activity: View local OpenAPI file in Swagger UI

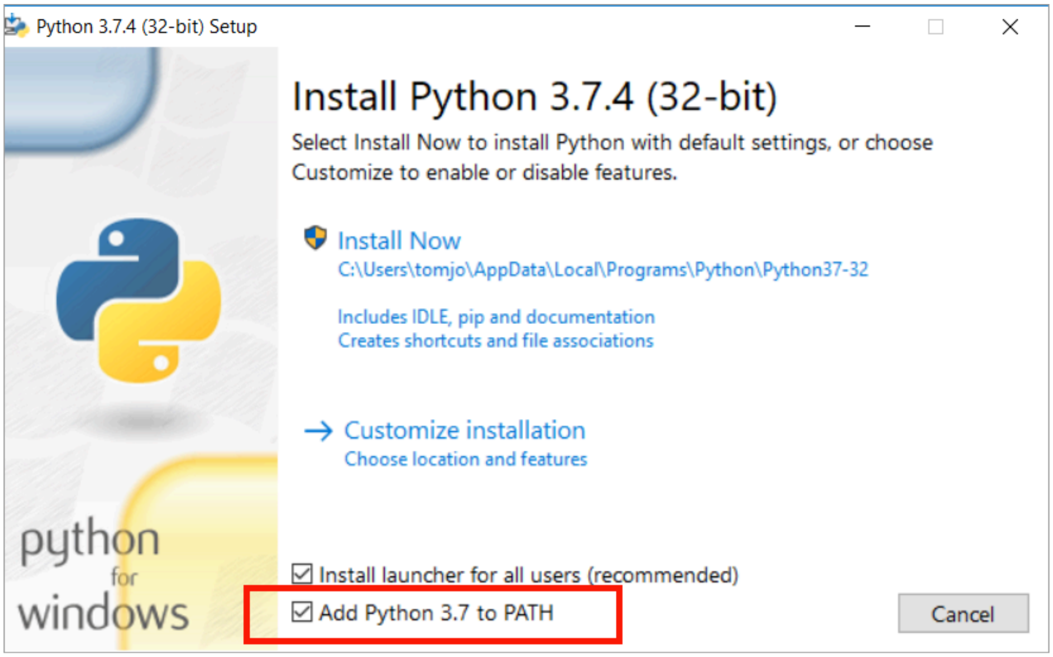
In order to view a *local* OpenAPI file (rather than an OpenAPI file hosted on a web server), you'll need to run an HTTP server on your computer. This is because CORS (cross-origin resource sharing) security restrictions in Chrome will block Swagger UI from running. Swagger UI needs to load on a web server to fulfill the security requirements.

You can create a local web server running on your computer through Python's SimpleHTTPServer module ↗. Mac has a system version of Python installed by default, but Windows computers will need to install Python.

Windows: Run the Python simple HTTP server

1. Download and install Python 3x ↗.

   When you install Python, be sure to select the check box that says "Add Python 3.7 to PATH." *This check box isn't selected by default. If you don't select it, your command prompt won't recognize the word "python".*



Python 3.7.4 (32-bit) Setup

Install Python 3.7.4 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\tomjo\AppData\Local\Programs\Python\Python37-32

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.7 to PATH

Add Python 3.7 to PATH

2. After installing Python, close your command prompt and reopen it.
3. In your command prompt, browse to the Swagger UI `dist` directory.

   To browse in the Windows command prompt, type `cd <folder name>` to move into the folder. Type `cd ..` to move up a directory. Type `dir` to see a list of the current directory's contents.

   If you're having trouble locating the `dist` directory in the command prompt, try this trick: type `cd`, press the spacebar, and then drag the `dist` folder directly into the command prompt. The path will be printed automatically.

4. After you've navigated into the `dist` folder, launch the server:

   ```
   python3 -m http.server
   ```

   If this command doesn't work, try it without the 3:

   ```
   python -m http.server
   ```

   The server starts:

   ```
   Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
   ```

   (If your Command Prompt doesn't recognize `python`, then you probably need to add Python to your PATH. Instructions for doing that are outside the scope of this tutorial.)

5. Go to `http://localhost:8000/` in your address bar. This address lets you view the local web server.

By default, web servers default to the `index.html` file in the directory, so it will show the Swagger UI file automatically. If the browser doesn't direct to `index.html`, add it manually: `http://localhost:8000/index.html`.

To stop the server, press **Ctrl+C** in your command prompt. If you closed your Command Prompt before stopping the service, type `ps`, find the process ID, then type `kill -9 <process ID>`.

Mac: Run the Python simple HTTP server

1. In your terminal, browse to the Swagger UI `dist` directory.

   To browse in your terminal, type `cd <folder name>` to move into the folder. Type `cd ..` to move up a directory. Type `ls` to see a list of the current directory's contents.

   If you're having trouble locating the `dist` directory in the command prompt, try this trick: type `cd`, press the spacebar, and then drag the `dist` folder directly into the command prompt. The path will be printed automatically.

2. Since Mac already has Python, you can just run the following in your terminal to launch simple server:

   ```
   python -m http.server
   ```

   If this command doesn't work, try it with the 3 in case you already have Python3 installed:

   ```
   python3 -m http.server
   ```

   The server starts:

   ```
   Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
   ```

3. Go to `http://localhost:8000/` in your address bar. This address lets you view the local web server.

   By default, web servers default to the `index.html` file in the directory, so it will show the Swagger UI file automatically. If the browser doesn't direct to `index.html`, add it manually: `http://localhost:8000/index.html`.

   To stop the server, press **Ctrl+C** in your command prompt. If you closed your Command Prompt before stopping the service, type `ps`, find the process ID, then type `kill -9 <process ID>`.

For more details on using the Python simple server, see How do you set up a local testing server? for more details.

Customize the OpenAPI file

By default, SwaggerUI has the Petstore OpenAPI document configured in the `url` parameter in the `index.html` file. You need to swap in your local file instead.

1. Download the this OpenAPI document (right-click the link and save the YAML file to your desktop.).

2. Drag your OpenAPI specification file, `openapi_openweathermap.yml`, into the `dist` folder. Your file structure should look as follows:

```
├── dist
│   ├── favicon-16x16.png
│   ├── favicon-32x32.png
│   ├── index.html
│   ├── oauth2-redirect.html
│   ├── swagger-ui-bundle.js
│   ├── swagger-ui-bundle.js.map
│   ├── swagger-ui-standalone-preset.js
│   ├── swagger-ui-standalone-preset.js.map
│   ├── swagger-ui.css
│   ├── swagger-ui.css.map
│   ├── swagger-ui.js
│   ├── swagger-ui.js.map
│   ├── swagger30.yml
│   └── openapi_openweathermap.yml
```

3. Inside your `dist` folder, open `swagger-initializer.js` in a text editor such as Sublime Text.
4. Look for the following code:

   ```
   url: "http://petstore.swagger.io/v2/swagger.json",
   ```

5. Change the `url` value from `http://petstore.swagger.io/v2/swagger.json` to a relative path to your YAML file, and then save the file. For example:

   ```
   url: "openapi_openweathermap.yml",
   ```

6. View the `index.html` file locally in your browser using the Python simple server. For example, go to `http://0.0.0.0:8000/` or `http://0.0.0.0:8000/index.html`.

   If you go to the file path, such as `file:///Users/tomjoht/Downloads/dist/index.html`, you might see a message that says “Failed to load API definition” note in the JavaScript Console that says “URL scheme must be “http” or “https” for CORS request.” The SimpleServer simulates this http or https.

7. To stop the Python simpler server, press **Ctrl+C** in your terminal or command prompt.

Configuring Swagger UI parameters

Swagger UI provides various configuration parameters (unrelated to the OpenAPI parameters (p. 293)) that you can use to customize the interactive display. For example, you can set whether each endpoint is expanded or collapsed, how tags and operations are sorted, whether to show request headers in the response, whether to include the Models section after the list of endpoints, and more.

We won’t get too much into the details of these configuration parameters in the tutorial. I just want to call attention to these parameters here for awareness.

If you look at the source of my Swagger UI demo (go to View > Source), you’ll see the parameters listed in the `// Build a system` section:

```
  // Build a system
const ui = SwaggerUIBundle({
  url: "openapi_openweathermap.yml",
  dom_id: '#swagger-ui',
  defaultModelsExpandDepth: -1,
  deepLinking: true,
  presets: [
    SwaggerUIBundle.presets.apis,
    SwaggerUIStandalonePreset
  ],
  plugins: [
    SwaggerUIBundle.plugins.DownloadUrl
  ],
  layout: "StandaloneLayout"
})
```

The parameters there (e.g., `deepLinking`, `dom_id`, etc.) are defaults. However, I've added `defaultModelsExpandDepth: -1` to hide the "Models" section at the bottom of the Swagger UI display (because I think that section is unnecessary).

You can also learn about the Swagger UI configuration parameters in the Swagger documentation ↗.

Challenges with Swagger UI

As you explore Swagger UI, you may notice a few limitations:

- There's not much room to describe in detail the workings of the endpoints. If you have several paragraphs of details and gotchas about a parameter, it's best to link out from the description to another page in your docs. The OpenAPI spec provides a way to link to external documentation in both the paths object (p. 291), the info object (p. 285), and the externalDocs object (p. 337)
- The Swagger UI looks mostly the same for each API. You can customize Swagger UI ↗ with your own branding, but it will require some more in-depth UX skills. It is, however, relatively easy to change the color and image in the top navigation bar.
- The Swagger UI might be a separate site from your other documentation. This separate output means that in your regular docs, you'll probably need to link to Swagger as the reference for your endpoints. You don't want to duplicate your parameter descriptions and other details in two different sites. See Integrating Swagger UI with the rest of your docs (p. 241) for strategies on unifying your reference docs and user guide.

Troubleshooting issues with Swagger UI

When you're setting up Swagger UI, you might run into some issues. The following issues are the most common:

CORS issues:

If you have security correctly configured, but the requests are rejected, it could be due to a CORS (cross-origin resource sharing) issue. CORS is a security measure that websites implement to make sure other scripts and processes cannot take their content through requests from remote servers. See CORS Support ↗ in Swagger UI's documentation for details.

If the requests aren't working, open your browser's JavaScript console when you make the request and see if the error relates to cross-origin requests. If this is the error, ask your developers to enable CORS on the endpoints. (To open the JavaScript Console, on Chrome on a Mac, go to **View > Developer > Javascript Console**; on Windows, click the **menu** button (vertical ellipses) and go to **More tools > Developer tools**. Then click the **Console** tab.)

Host URL issues:

The host for your test server might be another reason that requests are rejected. Some APIs (like Aeris Weather) require you to create an App ID that is based on the host URL where you'll be executing requests. If the host URL you registered is `http://mysite.com`, but you're submitting the test from `https://editor.swagger.io/`, the API server will reject the requests.

If you need help, the Swagger Google Group is a helpful resource for troubleshooting.

Embedding Swagger UI within an existing site

In addition to publishing your Swagger UI output as a standalone site, you can also embed the Swagger file within an existing site. See the following:

- Standalone Swagger UI Demo
- Embedded Swagger UI Demo (p. 240)

Since the Swagger UI site is responsive, it resizes well to fit into most any space. Even so, embedding Swagger into an existing site still looks like a website within a website.

Key terms

Swagger

Refers to API tooling related to the OpenAPI spec. Some of these tools include Swagger Editor, Swagger UI, Swagger Codegen, SwaggerHub, and others. These tools are managed by Smartbear. For more tools, see Swagger Tools. "Swagger" was the original name of the OpenAPI spec, but the name was later changed to OpenAPI to reinforce the open, non-proprietary nature of the standard. People sometimes refer to both names interchangeably (especially on older web pages), but "OpenAPI" is how the spec should be referred to. For more on naming conventions between OpenAPI and Swagger, see What Is the Difference Between Swagger and OpenAPI?.

OpenAPI

The official name for the OpenAPI specification. The OpenAPI specification provides a set of properties that can be used to describe your REST API. When valid, the specification document can be used to create interactive documentation, generate client SDKs, run unit tests, and more. You can read the specification details on GitHub at https://github.com/OAI/OpenAPI-Specification. Under the Open API Initiative with the Linux Foundation, the OpenAPI specification aims to be vendor neutral (many companies steer its development, not just one).

Swagger Editor

An online editor that validates your OpenAPI document against the rules of the OpenAPI specification. The Swagger Editor will flag errors and give you formatting tips. See Swagger Editor.

Swagger UI

An open-source web framework (on GitHub) that parses an OpenAPI specification document and generates an interactive documentation website. Swagger UI is the tool that transforms your spec into the Petstore-like site.

Swagger Codegen

Generates client SDK code for a lot of different platforms (such as Java, JavaScript, Scala, Python, PHP, Ruby, Scala, and more). The client SDK code helps developers integrate your API on a specific platform and provides for more robust implementations that might include more scaling, threading, and other necessary code. In general, SDKs are toolkits for implementing the requests made with an API. Swagger Codegen generates the client SDKs in nearly every programming language. See Swagger Codegen for more information. See also SDKs (p. 480).

For more terms, see the Glossary (p. 55).

SwaggerHub introduction and tutorial

Previously, I explored using the open-source Swagger UI project (p. 218) as a way to render your OpenAPI specification document (p. 272). The same company that offers the free, open-source version of Swagger Editor and Swagger UI (Smartbear) also offers a premium version with more robust features. This premium version of Swagger Editor is called SwaggerHub. You can see a comparison of features between the open source and premium versions here.

You can see a demo of the sample OpenWeatherMap API on SwaggerHub here.

Note that Smartbear is one of the sponsors of my site.

Advantages of SwaggerHub

While the open-source Swagger Editor coupled with Swagger UI works, you'll run into several problems:

- It's challenging to collaborate with other project members on the spec.
- It's difficult to gather feedback from reviewers about specific parts of the spec.
- You can't automatically provide the API in the myriad code frameworks your users might want it in.

When you're working on REST API documentation, you need tools specifically designed for REST APIs — tools that allow you to create, share, collaborate, version, test, and publish the documentation in ways that don't require extensive customization or time.

There's a point at which experimenting with the free Swagger UI tooling hits a wall, and you'll need to find another way to move to the next level. This next level is where SwaggerHub from Smartbear comes in. SwaggerHub provides a complete solution for designing, managing, and publishing documentation for your API in ways that will simplify your life as an API technical writer.

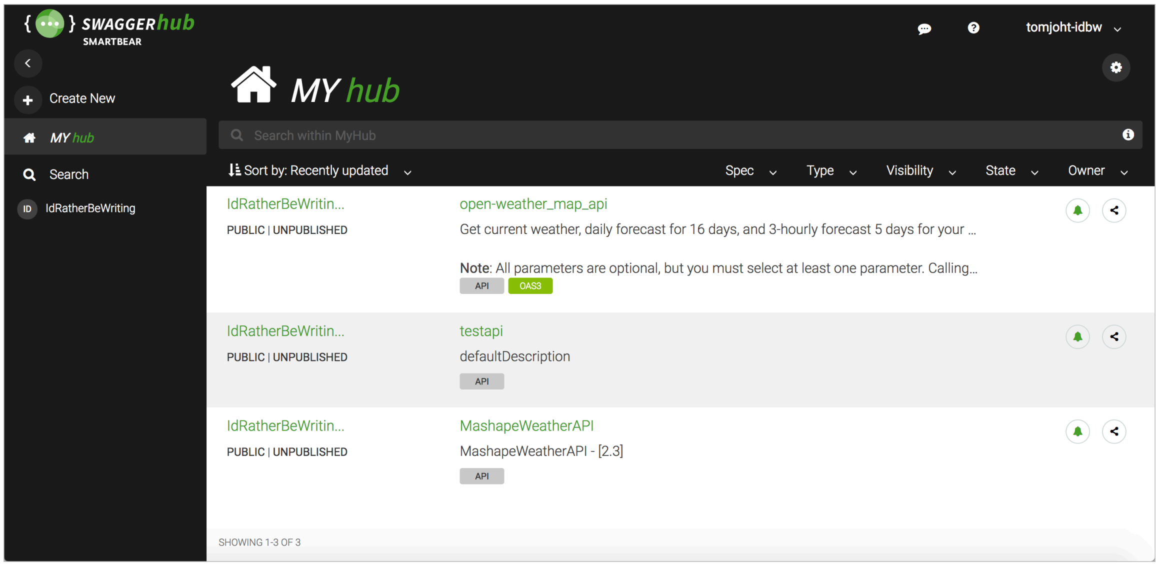
More than 15,000 software teams across the globe use SwaggerHub. As the OpenAPI spec becomes more of an industry standard for API documentation, SwaggerHub's swagger-specific tooling can be essential.

SwaggerHub Intro and Dashboard

Smartbear is the company that maintains and develops the open source Swagger tooling (Swagger Editor, Swagger UI, Swagger Codegen, and others.) They also formed the OpenAPI Initiative, which leads the evolution of the Swagger (OpenAPI) specification.

Smartbear developed SwaggerHub as a way to help teams collaborate around the OpenAPI spec. Many of the client and server SDKs can be auto-generated from SwaggerHub, and there are a host of additional features you can leverage as you design, test, and publish your API.

To get started with SwaggerHub, go to swaggerhub.com and create an account or sign in with your GitHub credentials. After signing in, you see the SwaggerHub dashboard.



IdRatherBeWritin...
PUBLIC | UNPUBLISHED
open-weather_map_api
Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your ...

Note: All parameters are optional, but you must select at least one parameter. Calling...
API OAS3

IdRatherBeWritin...
PUBLIC | UNPUBLISHED
testapi
defaultDescription
API

IdRatherBeWritin...
PUBLIC | UNPUBLISHED
MashapeWeatherAPI
MashapeWeatherAPI - [2.3]
API

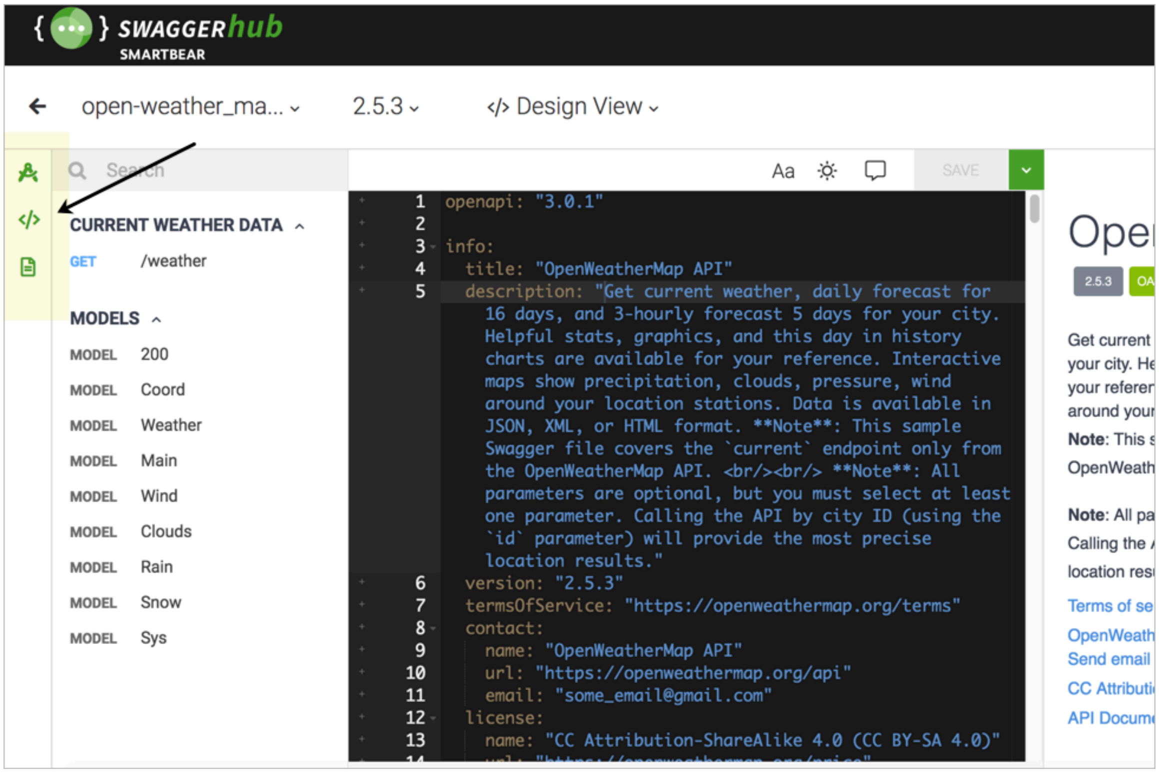
SHOWING 1-3 OF 3

The dashboard shows a list of the APIs you've created. In this example, you see the OpenWeatherMap API (p. 65) that I've been using throughout this course.

SwaggerHub Editor

SwaggerHub contains the same Swagger Editor that you can access online. This editor provides you with real-time validation as you work on your API spec. However, unlike the standalone Swagger Editor, with SwaggerHub's Swagger Editor, you can toggle between several modes:

- Hide Navigation
- Hide Editor and Navigation
- Hide UI Docs



```
openapi: "3.0.1"

info:
  title: "OpenWeatherMap API"
  description: "Get current weather, daily forecast for
    16 days, and 3-hourly forecast 5 days for your city.
    Helpful stats, graphics, and this day in history
    charts are available for your reference. Interactive
    maps show precipitation, clouds, pressure, wind
    around your location stations. Data is available in
    JSON, XML, or HTML format. **Note**: This sample
    Swagger file covers the `current` endpoint only from
    the OpenWeatherMap API. <br/><br/> **Note**: All
    parameters are optional, but you must select at least
    one parameter. Calling the API by city ID (using the
    `id` parameter) will provide the most precise
    location results."
  version: "2.5.3"
  termsOfService: "https://openweathermap.org/terms"
  contact:
    name: "OpenWeatherMap API"
    url: "https://openweathermap.org/api"
    email: "some_email@gmail.com"
  license:
    name: "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)"
```

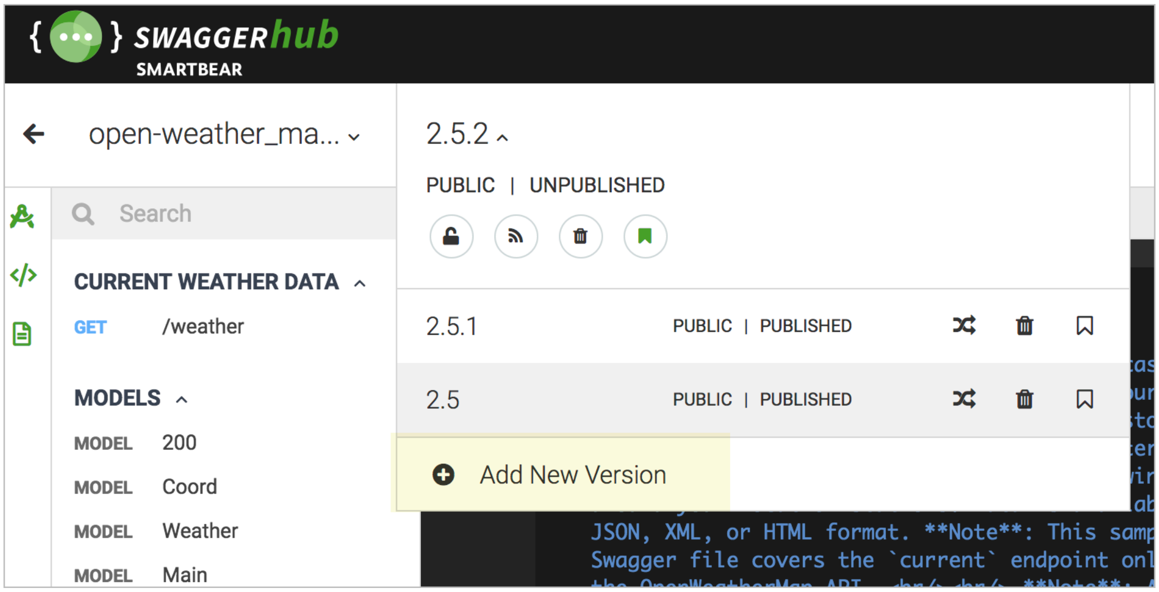
SwaggerHub's editor gives you more flexible viewing options

Most importantly, as you're working in the Editor, SwaggerHub allows you to *save your work*. With the free Swagger Editor, your content is kept in the browser cache, with no ability to save the file in the cloud. When you clear your cache, your content is gone. As a result, if you use the standalone Swagger Editor, you have to regularly copy the content from the Swagger Editor into a file on your own computer each time you finish.

With SwaggerHub, you can save your specification document directly on SwaggerHub's servers, or you can reference and store it in an external source such as GitHub.

Versions

SwaggerHub not only allows you to save your OpenAPI spec but also save different versions of your spec. As a result, you can experiment with new content by adding a new version. You can return to any version you want, and you can also publish or unpublish any version.



Versioning options for your OpenAPI file

When you publish a version, the published version becomes Read Only. If you want to make changes to a published version (rather than creating a new version), you can unpublish the version and make edits to it.

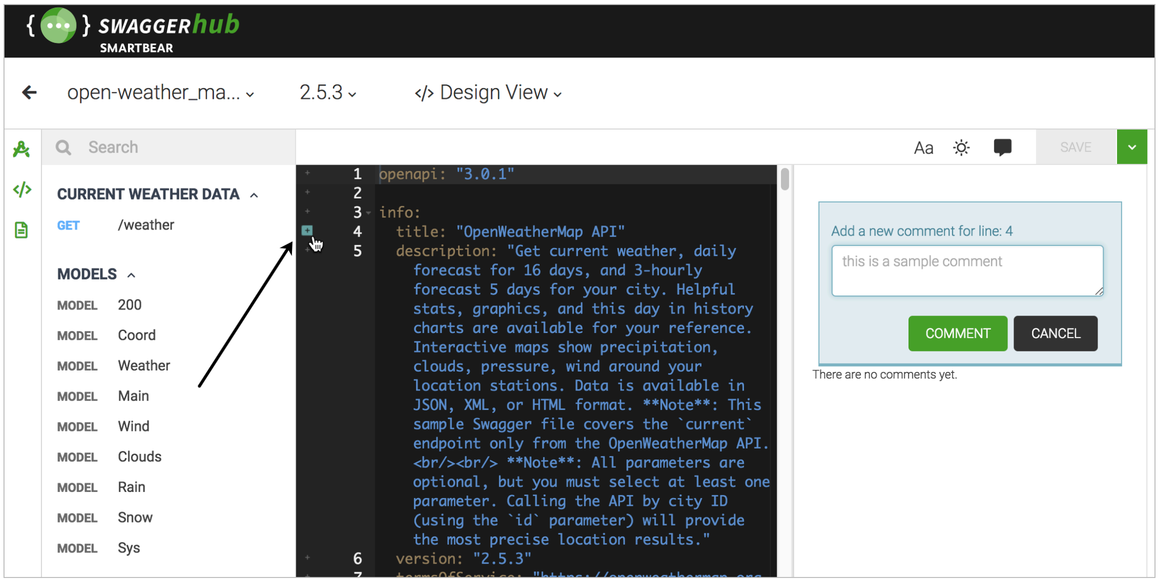
You can link to specific versions of your documentation, or you can use a more general link path that will automatically forward to the latest version. Here's a link to the OpenWeatherMap API published on SwaggerHub that uses version 2.5.1 of the documentation: https://app.swaggerhub.com/apis/IdRatherBeWriting/open-weather_map_api/2.5.1/ . To link to a specific version, include the version number in the URL. In contrast, this more general link (which omits a version number) automatically forwards to the latest version (which is 2.5.2): https://app.swaggerhub.com/apis/IdRatherBeWriting/open-weather_map_api/ .

Versioning is helpful when you're collaborating on the spec with other team members. For example, suppose the original version was drafted by an engineer, and you want to make major edits. Rather than directly overwriting the content (or making a backup copy of an offline file), you can create a new version and then take more ownership to overhaul that version with your own updates, without fear that the engineer will react negatively about overwritten/lost content.

When you publish your Swagger documentation on SwaggerHub, Swagger's base URL (`app.swaggerhub.com`) remains in the URL. Although this base URL isn't customizable, you can add your company logo and visual branding as desired.

Inline commenting/review

Key to the review process is the ability for team members to comment on the spec inline, similar to Google Docs and its margin annotations. When you're working in SwaggerHub's editor, a small plus sign appears to the left of every line. Click the plus button to add a comment inline at that point.



```
openapi: "3.0.1"

info:
  title: "OpenWeatherMap API"
  description: "Get current weather, daily
    forecast for 16 days, and 3-hourly
    forecast 5 days for your city. Helpful
    stats, graphics, and this day in history
    charts are available for your reference.
    Interactive maps show precipitation,
    clouds, pressure, wind around your
    location stations. Data is available in
    JSON, XML, or HTML format. **Note**: This
    sample Swagger file covers the `current`
    endpoint only from the OpenWeatherMap API.
    <br/><br/> **Note**: All parameters are
    optional, but you must select at least one
    parameter. Calling the API by city ID
    (using the `id` parameter) will provide
    the most precise location results."
  version: "2.5.3"
```

Add a new comment for line: 4

this is a sample comment

There are no comments yet.

Inline commenting and reply features on SwaggerHub

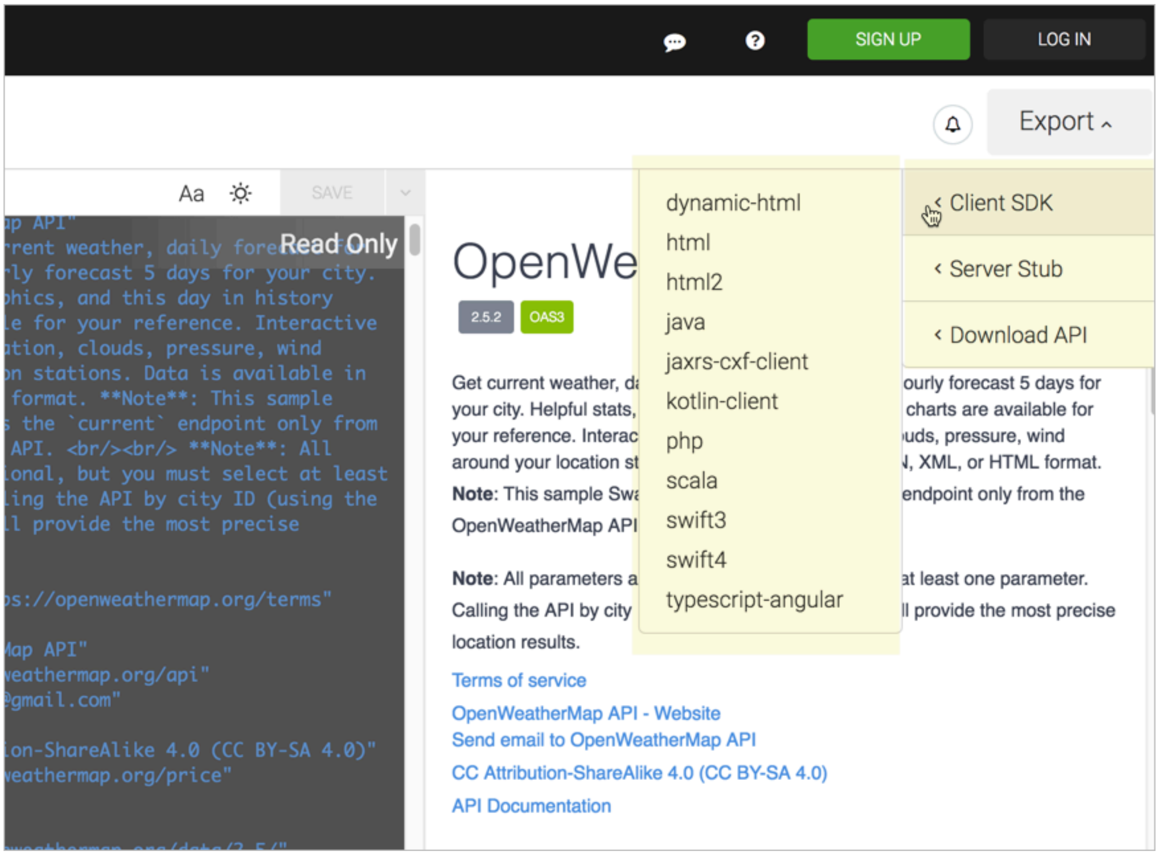
When you click the plus sign, a comment pane appears on the right where you can elaborate on comments, and where others can reply. Users can edit, delete, or resolve the comments. This commenting feature helps facilitate the review process in a way that tightly integrates with your content. You can also collapse or show the comments pane as desired.

Few tech comm tools support inline annotations like this, and it wouldn't be possible without a database to store the comments, along with profiles associated with the reviewers. This feature would be tedious to implement on your own, as it would require both a database and an authentication mechanism. This is all included in SwaggerHub.

Auto-Generate Client SDKs

Another benefit to SwaggerHub is the ability to auto-generate the needed client or server code from your specification. Client SDKs provide the tooling needed to make API requests in specific programming languages (like Java or Ruby).

In the upper-right corner, click the down-arrow and select **Client** or **Server**. Users have access to generate client and server SDKs (p. 480) in more than 30 formats.



Client and server SDK export capabilities

For example, suppose a user is implementing your REST API in a Java application. The user can choose to download the Java client SDK for extensive code that shows a Java implementation of your API. Other options include Ruby, Android, Go, CSharp, JavaScript, Python, Scala, PHP, Swift, and many more.

Some API documentation sites look impressive for showing implementations in various programming languages. SwaggerHub takes those programming languages and multiplies them tenfold to provide every possible output a user could want.

The output includes more than a simple code sample showing how to call a REST endpoint in that language. The output includes a whole SDK that includes the various nuts and bolts of an implementation in that language. (For more information on SDKs, see SDKs (p. 480).)

Providing this code speeds implementation for developers and helps you scale your language-agnostic REST API to a greater variety of platforms and users, reducing the friction in adoption.

Export to HTML

Among SwaggerHub's many options for generating client and SDK files is an HTML option. You can export your OpenAPI spec as a static HTML file in one of two styles: HTML or HTML2.

You can see a demo export of the OpenWeatherAPI API here: HTML ↗ or HTML2 ↗. Both exports generate all the content into an index.html file.

The HTML export is a more basic output than HTML2. You could potentially incorporate the HTML output into your other documentation, such as what Cherryleaf did in importing Swagger into Flare. (You might have to strip away some of the code and provide styles for the various documentation elements, and there wouldn't be any interactivity for users to try it out, but it could be done.) In another part of the course, I expand on ways to integrate Swagger UI's output with the rest of your docs (p. 241).

The HTML2 export is more intended to stand on its own, as it has a fixed left sidebar to navigate the endpoints and navtabs showing six different code samples:

You can see a demo export of the OpenWeatherAPI API here: HTML or HTML2. Both exports generate all the content into an index.html file.

Both outputs would need a healthy dose of custom styling to be usable.

Mocking Servers

Another cool feature of SwaggerHub is the ability to create mock API servers. Suppose you have an API in which you don't want users to generate real requests. (Maybe it's an ordering system where users might be ordering products through the API, or you don't have test accounts/systems). Even so, you can still simulate responses that let users get a sense of how your API works.

Assuming you have example responses in your API spec, you can set your API to "auto-mock." When a user tries out a request, SwaggerHub will return the example response from your spec. The response won't contain the custom parameters the user entered in the UI but will instead return the example responses coded into your spec as if returned from a server.

Providing an auto-mock for your API solves the problem of potentially complicating user data by having users interact with their real API keys and data. In many cases, you don't want users junking up their data with tests and other experiments. At the same time, you also want to simulate the API response.

Simulating the API can be especially useful for testing your API with beta users. One reason many people code their API with the spec before writing any lines of code (following a spec-first philosophy such as that described by Michael Stowe (p. 185)) is to avoid coding an API with endpoints and responses that users don't want.

Using the mock server approach, SwaggerHub not only provides documentation but also acts as a beta-testing tool to get the design of your API right before sinking thousands of hours of time into actual coding. You can enable auto-mocking for different versions of your API, creating variants and testing each of the variants.

To set up a mocking server in SwaggerHub, click the plug icon and select to add a new integration. Select the **API Auto Mocking** service and complete the configuration details. Make sure you have `examples` for each of the endpoint responses in your spec. See API Auto Mocking for more details.

Content Re-use (Domains)

Another feature exclusively available in SwaggerHub is the concept of domains. Domains are re-useable code snippets that you can leverage to avoid duplication in your spec.

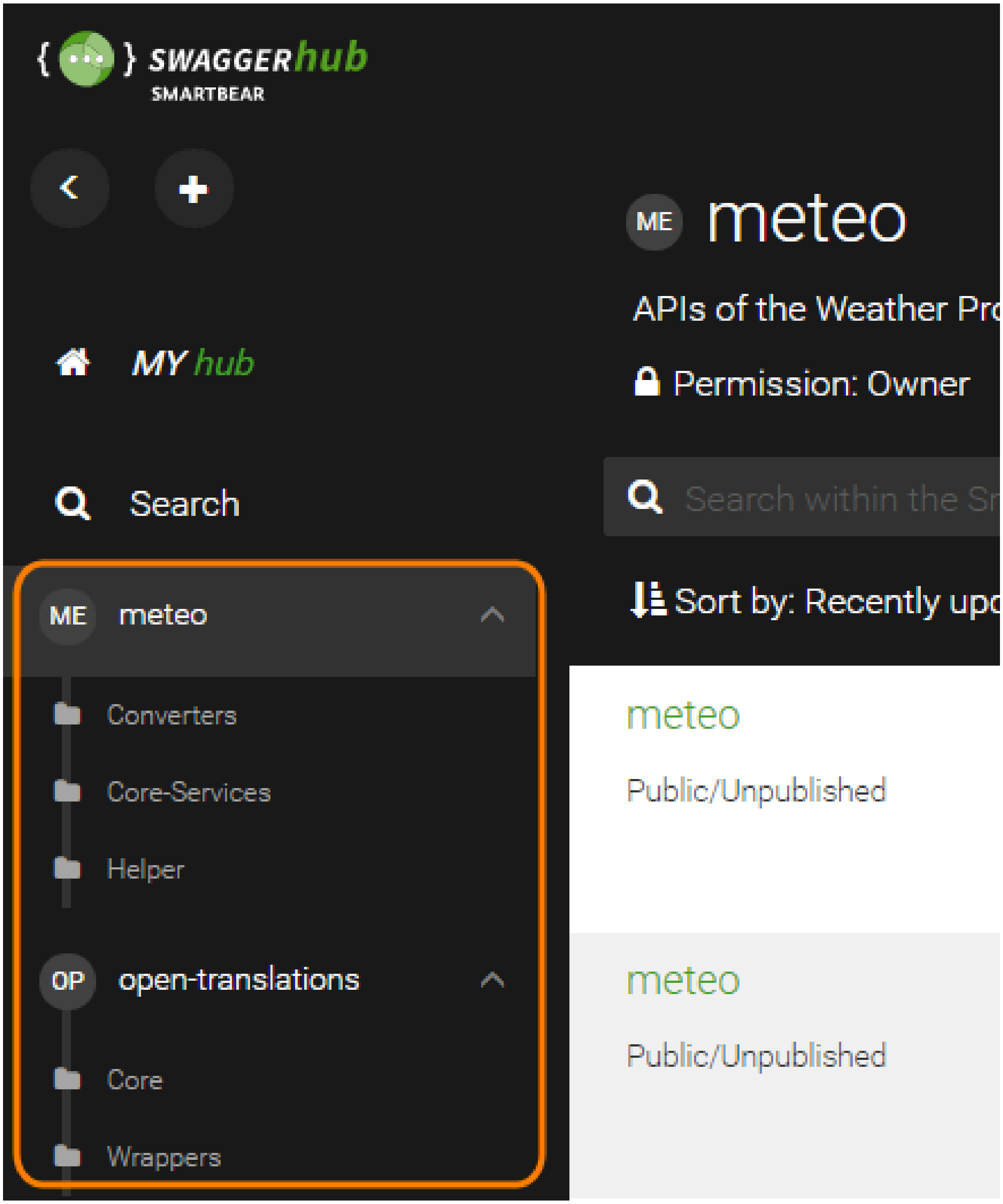
When you create definitions for your requests and responses, you may find yourself re-using the same code over and over. Rather than duplicating this code, you can save it as a domain. When you want to re-use the code, you select this domain.

Using the domain minimizes duplicate content and enables you to be more consistent and efficient. You can read more in Domains.

Organizations and projects

The collaborative aspect of SwaggerHub is the most common reason people move from the open source tools to SwaggerHub. You might have a lot of different engineers working on a variety of APIs in SwaggerHub. To organize the work, you can group APIs into organizations, and then assign members to the appropriate organization. When that member logs in to SwaggerHub, he or she will see only the organizations to which he or she has access.

Additionally, within an organization, you can further group APIs into different projects. This way teams working in the same organization but on different projects can have visibility into other APIs.



Organization of projects by team

This aspect of organizations and projects may not seem essential if you have just one or two APIs, but consider how you'll scale and grow as you have dozens of APIs and multiple teams. In these more robust scenarios, the organization and project features become essential.

Expanding the tech writer’s role with APIs

Tech writers are positioned to be power players in the spec-first philosophy with OpenAPI design. By becoming adept at coding the OpenAPI spec and familiar with robust collaborative tools like SwaggerHub, tech writers can lead engineering teams not only through the creation and refinement of the API documentation but also pave the way for beta testing, spec review, client/server SDK generation, and more.

Designing a fully-featured, highly functioning OpenAPI spec is at the heart of this endeavor. Few engineers are familiar with creating these specs, and technical writers who are skilled at both creating the spec and setting up Swagger tooling can fill critical roles on API teams.

Great tools aren’t free. SwaggerHub does cost money, but this is a good thing since free tools are frequently abandoned, poorly maintained, and lack documentation and support. By using a paid tool from a robust API company (the same company that maintains the Swagger tools and sponsors the OpenAPI specification), you can plug into the infrastructure you need to scale your API documentation efforts.

To read more about SwaggerHub, check out my blog post SwaggerHub: A collaborative platform for working on OpenAPI/Swagger specification files, and more.

Related resources

- Getting Started With SwaggerHub

Swagger UI Demo

When you use Swagger UI, it's not necessary for the Swagger UI output to be a standalone site. You can also embed Swagger into an existing web page. The following is an embedded instance of the Swagger UI showing the OpenAPI file for the OpenWeatherMapAPI.

This page can only be viewed online through a browser on your computer. Go to https://idratherbewriting.com/learnapidoc/pubapis_swagger_demo.html to view it.

While the Swagger UI display is designed to be responsive, the collapse/expand sections in the Model views still have overflow issues in responsive views, so you might run into issues with embedding. For more discussion on how to integrate Swagger with the rest of your docs, see Integrating Swagger UI with the rest of your docs (p. 241).

Integrating Swagger UI with the rest of your docs

Whenever discussions about Swagger and other REST API specifications take place, technical writers invariably ask how they can integrate the Swagger output with the rest of their documentation. This question dominates tech writer discussions perhaps more than any other when it comes to Swagger.

Background to the single source of truth idea

One of the core principles of tech writing is to generate content from a single source rather than copying and pasting duplicate information. When you start pushing your API documentation into another source file — in this case, a YAML or JSON file that is included in a Swagger UI file set, you end up splitting your single source of truth into multiple sources.

For example, you might have defined your endpoints and parameters in your regular documentation, and now the OpenAPI spec asks you to provide the same endpoints and descriptions in the spec. Do you copy and paste the same parameters and other information across both sites? Do you somehow generate the descriptions from the same source? In this topic, I tackle this conundrum.

Option 1: Put all the info into your spec through expand/collapse sections

You can try to squeeze all your documentation into the specification document itself. This only works if you have a relatively small amount of conceptual information to accompany your API (p. 358). You may be surprised about how much information you can actually include in the spec. Any `description` element (not just the `description` property in the `info` object) allows you to use Markdown and HTML.

For example, here's the `info` object in the OpenAPI spec where a description appears. (If desired, you can type a pipe `>` to break the content onto the next line and then indent two spaces. You can actually add a lot of content in `description` elements.)

```
info:
  title: OpenWeatherMap API
  description: 'Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `weather` endpoint only from the OpenWeatherMap API. <br/><br/> **Tip**: We recommend that you call the API by city ID (using the `id` parameter) to get unambiguous results for your city.'
  version: '2.5'
```

With one Swagger API project I worked on, I referenced Bootstrap CSS and JS in the header of the `index.html` of the Swagger UI project, and then incorporated Bootstrap alerts and expand/collapse buttons in this `description` element. Here's an example:

```
info:
  description: >
    ACME offers a lot of configuration options...
    <div class="alert alert-success" role="alert"><i class="fa fa-info-circle"></i> <b>Tip: </b>See the resources available in the portal for more detail.</div>
    <div class="alert alert-warning" role="alert"><i class="fa fa-info-circle"></i> <b>Note: </b>The  network includes a firewall that protects your access to the resources...</div>

    <div class="container">
    <div class="apiConfigDetails">
    <button type="button" class="btn btn-warning" data-toggle="collapse" data-target="#demo">
    <span class="glyphicon glyphicon-collapse-down"></span> See API Configuration Details
    </button>
    <div id="demo" class="collapse">

    <h2>Identifiers Allowed</h2>

    <p>Based on this configuration, ACME will accept any of the following identifiers in requests.</p>

    <table class="table">
    <thead>
    <tr>
    <th>Request Codes</th>
    <th>Data Type</th>
    <th>Comparison Method</th>
    </tr>
    </thead>
    <tbody>
    <tr>
    ...
```

(Unfortunately, I can’t point to the output as this project was behind a firewall.)

The result was to compress much of the information into a single button that, when clicked, expanded with more details. By incorporating expand/collapse sections from Bootstrap, you can add a ton of information in this `description` object. (For the JavaScript you need, add `script` references in the header or footer of the same `index.html` file where you referenced your `openapi.yaml` file.)

Additionally, you can include modals that appear when clicked. Modals are dialog windows that dim the background outside the dialog window. Again, you can include all the JavaScript you want in the `index.html` file of the Swagger UI project.

If you incorporate Bootstrap, you will likely need to restrict the namespace so that it doesn’t affect other elements in the Swagger UI display. (See How to Isolate Bootstrap CSS to Avoid Conflicts for details on how to do this.)

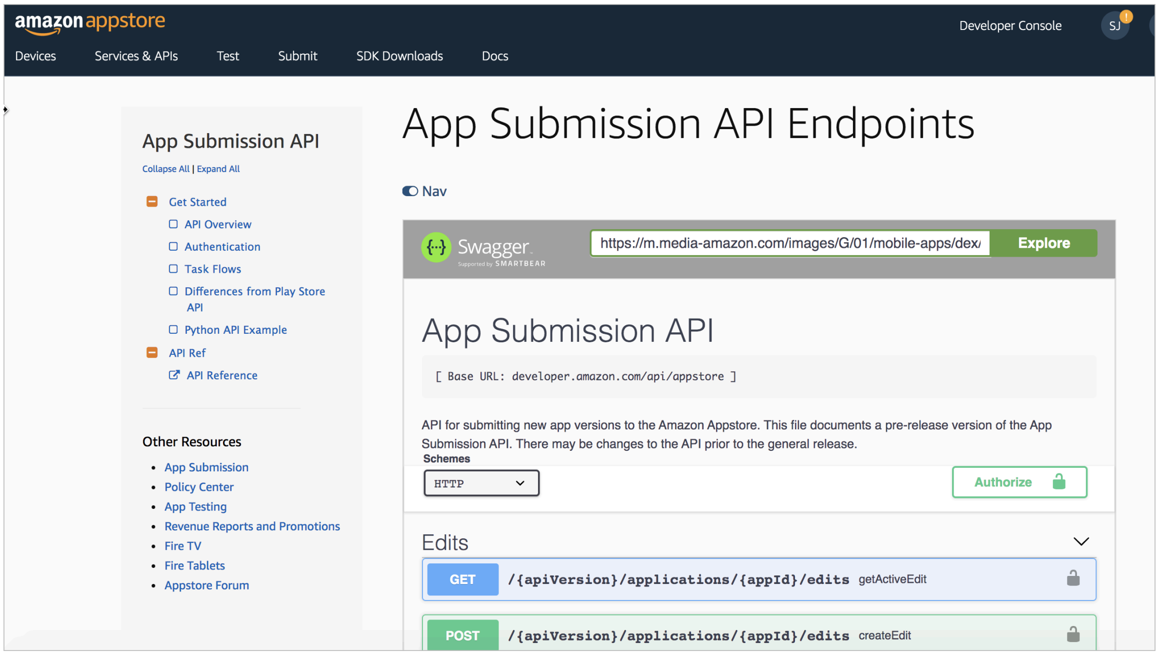
Overall, if your API docs are relatively small, you can try putting all your information in the spec first. If you have a complex API or just an API that has a lot of extra information not relevant to the spec, look for alternative approaches. But try to fit it into the spec first. This keeps your information in one space.

There are many benefits to using a spec that you will miss out on if you choose another approach. When you store your information in a spec, many other tools can parse the spec and generate interactive displays. Putting your content in the OpenAPI spec format allows you to separate your content from the presentation layer, instantly taking advantage of any new API tooling or platform that can parse the spec.

Option 2: Embed Swagger UI in your docs

Another solution is to embed Swagger UI in your regular docs. You can see an example of this here: Swagger UI Demo (p. 240). It's pretty easy to embed Swagger into an HTML page — just copy the code from the `index.html` file from the `dist` folder into your doc page (more or less). The latest version of Swagger has a more responsive, liquid design. It almost looks *designed* to be embedded into another site.

However, the effect is still kind of clunky and is obvious that the content is embedded from some other document generator. It's not a seamlessly branded experience. Here an example where Swagger is embedded directly in the docs: App Submission API



[Base URL: developer.amazon.com/api/appstore]

API for submitting new app versions to the Amazon Appstore. This file documents a pre-release version of the App Submission API. There may be changes to the API prior to the general release.

`/{apiVersion}/applications/{appId}/edits` getActiveEdit

`/{apiVersion}/applications/{appId}/edits` createEdit

Embedding Swagger into your existing doc site

Another example of the same embedding technique is here: Moments Developer Guide

Notice that I've included a **"Nav"** toggle at the top of the embedded Swagger content. This little JS trick will collapse your sidebar, giving your embedded Swagger display full width (which is usually needed for readability). This Nav toggle isn't part of the Swagger UI display but is something that's easy to add.

To include a Nav toggle, first make sure you're including Font Awesome and jQuery in your site. Then add the Nav icon with this class at the top of your page:

```
<p><a id="tg-sb-link" href="#"><i id="tg-sb-icon" class="fa fa-toggle-on"></i> Nav</a></p>
```

Now add a `toggleClass` script that will toggle your sidebar. You'll need to customize this a bit with the classes used in your own site. My script looks like this:

```
<script>
        $(document).ready(function() {
            $("#tg-sb-link").click(function() {
                $("#sidebar").toggleClass('navToggle');
                $(".container").toggleClass('expand');
                $("#tg-sb-icon").toggleClass('fa-toggle-on');
                $("#tg-sb-icon").toggleClass('fa-toggle-off');
            });
        });
</script>
```

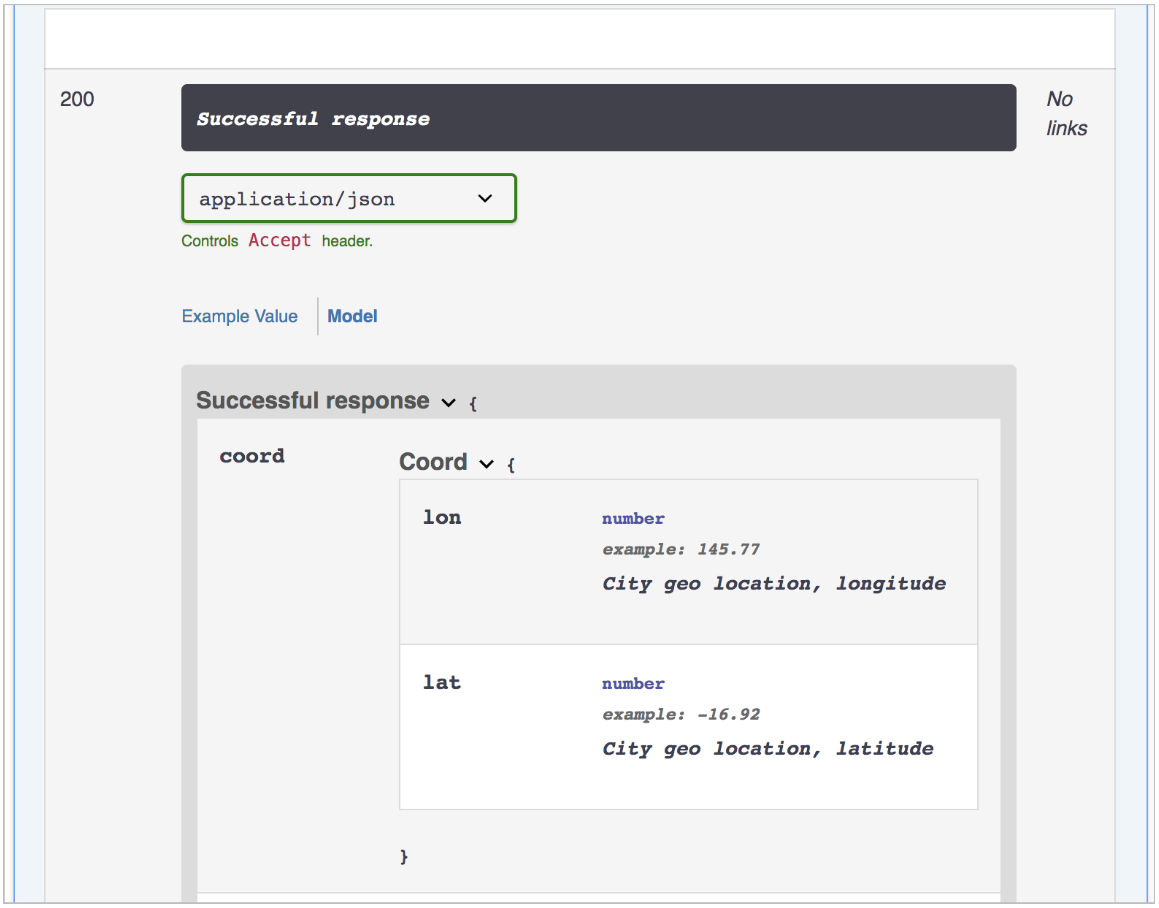
Then embed this style:

```
<style>
.navToggle {
  display: none !important;
}
.expand {
  width: 100%;
  margin-left: 10%;
  margin-right: 10%;
}
#tg-sb-link:hover, #tg-sb-link:active, #tg-sb-link {
  text-decoration: none !important;
}
</style>
```

Here's how this script works. When users click the element ID `tg-sb-link`, the anonymous function fires above. The `toggleClass` looks for the `sidebar` element and injects a class called `navToggle` into it. The embedded styles define `navToggle` with a `display: none` property. This makes your `sidebar` element disappear. (If your sidebar has some other class, customize `sidebar` with the name of your website's sidebar element.)

Continuing on with the script, the `toggleClass` function looks for the `container` class and injects an element called `expand`. The embedded styles for this element expand the main container to a larger width. (Again, you'll need to customize `container` to use the main container element for your website.) Hopefully you get the gist of how this `toggleClass` jQuery function is working. When you click the Nav button again, the same function removes the injected classes. For more details, check out the source code of the Swagger UI Demo (p. 240) on GitHub here.

Besides the clunkiness of the "site-within-a-site" look, an additional problem with embedding is that some of the Models aren't constrained within their container, so they expand beyond their limits. Try expanding the Model section in the demo (p. 240) — you'll see what I'm talking about.



Expanding Models -- if you have multiple levels of hierarchy, this section will not expand well when constrained within a fixed space

I'm not sure if some ninja styling prowess could simply overcome this uncontained behavior. Probably, but I'm not a CSS ninja, and I haven't fiddled around with this enough to say that it can actually be done. I did end up adding some custom styles to make some adjustments to Swagger UI in various places. If you view the source of the demo page (p. 240) and check out the second `<style>` block, you can see the styles I added.

With the embedded option, you can still use the official Swagger UI tooling to read the spec, and you can include the Swagger UI output in your main documentation. Swagger UI reads the latest version of the OpenAPI specification (p. 272), which is something many tools don't yet support. Additionally, Swagger UI has the familiar interface that API developers are probably already familiar with. However, if the styling overflows in ugly ways in your Model sections, you might want to avoid the embedded approach.

Option 3: Parse the OpenAPI specification document

If you're using a tool such as Jekyll, which incorporates a scripting language called Liquid, you can use Jekyll's instance of Liquid to read the OpenAPI specification document (which is, after all, just YAML syntax). For example, you could use a `for` loop to iterate through the OpenAPI spec values. Here's a code sample. In this example, the `swagger.yml` file is stored inside Jekyll's `_data` directory.

```
<table>
    <thead>
    <tr><th>Name</th><th>Type</th><th>Description</th><th>Required?</th></tr>
    </thead>
    {% for parameter in site.data.swagger.paths.get.parameters %}
        {% if parameter.in == "query" %}
        <tr>
            <td><code>{{ parameter.name }}</code></td>
            <td><code>{{ parameter.type }}</code></td>
            <td>
            {% assign found = false %}
            {% for param in site.data.swagger.paths.get.parameters %}
                {% if parameter.name == param.name %}
                    {{ param.description }}
                    {% assign found = true %}
                {% endif %}
            {% endfor %}
            {% if found == false %}
                ** New parameter **
            {% endif %}
            </td>
            <td><code>{{ parameter.required }}</code></td>
        </tr>
        {% endif %}
    {% endfor %}
</table>
```

Special thanks to Peter Henderson for sharing this technique and the code. With this approach, you may have to figure out the right Liquid syntax to iterate through your OpenAPI spec, and it may take a while. But this might work if you're looking for tight integration into your authoring tool. (Note that many static site generators (p. 591) can parse YAML, not just Jekyll.)

For more information on this approach, see Peter's write-up at Integrating Autogenerated Content Into Your Documentation Site Using Swagger and Jekyll and this sample GitHub code.

Option 4: Store content in YAML files that are sourced to both outputs

Another approach to integrating Swagger's output with your other docs might be to store your descriptions and other info in YAML data files in your project, and then include the data references in your specification document. I'm most familiar with Jekyll, so I'll describe the process using Jekyll (but similar techniques exist for other static site generators).

In Jekyll, you can store content in YAML files in your `_data` folder. For example, suppose you have a file called `parameters.yml` inside `_data` with the following content:

```
acme_parameter: >
  This is a description of my parameter ...
```

You can then include that reference using tags like this:

```
{{site.data.parameters.acme_parameter}}
```

In your Jekyll project, you would include this reference your spec like this:

```
info:
  description: >
    {{site.data.parameters.acme_parameter}}
```

You would then take the output from Jekyll that contains the content pushed into each spec property. In this model, you're generating the OpenAPI spec from your Jekyll project.

I've tried this approach. It's not a bad way to go, but it's hard to ensure that your OpenAPI spec remains valid as you write content. When you have references like this in your spec content (`{{site.data.parameters.acme_parameter}}`), you can't benefit from the real-time spec validation that you get when using the Swagger Editor.

Most likely, you'd need to include the entire Swagger UI project in your Jekyll site. At the top of your Swagger.yml file, add frontmatter dashes with `layout: null` to ensure Jekyll processes the file:

```
---
layout: null
---
```

In your `jekyll serve` command, configure the `destination` to build your output into an `htdocs` folder where you have a simple local HTTP server running. With each build, check the display to see whether it's valid or not.

By storing the values in data files, you can then include them elsewhere in your doc as well. For example, you might have a parameters section in your doc where you would also include the `{{site.data.parameters.acme_parameter}}` description.

Again, although I experimented with this approach, I grew frustrated at not being able to validate my spec immediately. It was more challenging to track down the exact culprits behind my validation errors, and I eventually gave up.

Option 5: Use a tool that imports Swagger and allows additional docs

Another approach is to use a tool like Readme.com or Stoplight that allows you to both import your OpenAPI spec and also add your own separate documentation pages. Readme provides one of the most attractive outputs and is fully inclusive of almost every documentation feature you could want or need. I explore Readme with more depth in Hybrid documentation systems (p. 609). Readme.com requires third-party hosting, but some other doc tools allow you to incorporate Swagger as well. I explored Stoplight in depth in a previous topic: Stoplight — visual modeling tools for creating your OpenAPI spec (p. 195).

Sites like Apiary and Mulesoft also let you import your OpenAPI spec while also adding custom documentation pages. These sites offer full-service management for APIs, so if your engineers are already using one of these platforms, it could make sense to store your docs there too.

Cherryleaf has an interesting post called Example of API documentation portal using MadCap Flare. In the post, Ellis Pratt shows a proof of concept with a Flare project that reads an OpenAPI spec and generates static HTML content from it. If you're using Flare, it might be worth exploring.

Option 6: Change perspectives — Having two sites isn’t so bad

Finally, ask yourself, what’s so bad about having two different sites? One site for your reference information, and another for your conceptual documentation (p. 358). Programmers might find the Swagger-based reference information convenient because it distills and simplifies the body of information. Rather than having a massive site to navigate, the Swagger output provides the core reference information they need. When they want conceptual information, they can consult the accompanying guide. Think of the Swagger UI output as your API’s quick reference guide (p. 412).

The truth is that programmers have been operating this way for years with Javadocs (p. 707), Doxygen (p. 723), and other document-generator tools ↗ that generate documentation from Java, C++, C#, Python, Ruby, and other programming sources. Auto-generating the reference information from source code into a standalone output is extremely common and wouldn’t be viewed as a fragmented information experience by programmers.

So in the end, instead of feeling that having two outputs is fragmented or disjointed, reframe your perspective. Your Swagger output provides a clear go-to source for reference information about the endpoints, parameters, requests, and responses. The rest of your docs provide tutorials and other conceptual information. Your two outputs just became an organizational strategy for your documentation.

Next steps

Now that you’ve had your head buried in API reference documentation, it’s time to dive into testing. As you work with API endpoints and other code, you’ll need to test these endpoints yourself, both to gather and verify the information in your documentation. Testing isn’t always straightforward, so I devote an entire section to this topic. Continue to Overview of testing your docs (p. 343).

Redocly tutorial -- authoring and publishing API docs with Redocly's command-line tools

Redocly ↗ provides a variety of tools for working with API docs. Using Redocly's command-line tools, you can split the OpenAPI definition into many sub-files, and then later bundle up the discrete files into a single file during the publishing stage. You can generate your docs into one of the most attractive outputs available for REST API docs, including integration with conceptual topics as well. Redocly also offers more robust developer portals and SaaS offerings that cover the full authoring and publishing lifecycle.

Note that Redocly is one of the sponsors of my site.

Background about Redocly's approach to simplifying the author experience with OpenAPI

Before Redocly became Redocly, the team worked on a product called Rebilly ↗ that included a 22,000 line OpenAPI definition file. Working in such a long file and authoring in YAML or JSON was especially challenging. Not only was maneuvering in a single massive file difficult, one small indentation error would require them to constantly troubleshoot syntax issues. (The single YAML file for all OpenAPI definition elements was analogous to software developers storing all code for an application in a single file — an unthinkable way of working with code.)

When Redocly innovated on their approach to simplifying OpenAPI spec development, instead of putting a graphical user interface (GUI) on top of the OpenAPI code and requiring users to write the OpenAPI definition in forms, menus, and other widgets, Redocly decided on a more docs-as-code approach that aligns with the developer audience.

Redocly developed command-line tools to split the OpenAPI definition into smaller files to make it easier to work with. Files are stored as separate files and referenced through JSON reference tag (`$ref`) in the main OpenAPI file. Working in smaller files reduces the likelihood of syntax issues and other formatting errors. It also makes it easier to focus on the content.

The CLI tools also let you check your OpenAPI definition against a ruleset to ensure you're following best practices. You can configure the ruleset with different levels of strictness, either including or ignoring optional best practices in the OpenAPI specification. This linting lets you identify issues in your OpenAPI definition that might not technically be errors, but would better if included.

When you're ready to publish your OpenAPI definition, you can also use a command line tool to bundle the small files up into a single file again. You can then feed your OpenAPI definition file into any OpenAPI documentation tool, including Redoc's documentation output: Redocly API Reference ↗ (premium version), or the community edition of Redoc ↗ (free version). There's also a Developer Portal ↗ option, or a more cloud-based Workflows integration ↗. Some of these tools (especially the command-line tools) will come into focus through the tutorial below.

Overview of Redocly products

Redocly has a variety of products, so let's first clarify these tools before diving more deeply into the the tutorial.

Tool	Description	Cost
Redoc	The API reference documentation output generated from the OpenAPI definition. Compare Redoc with Swagger UI (p. 240) or other document generators that parse through an OpenAPI file and render documentation. The free version has limited functionality.	Free
API Docs	The premium version of Redoc – the API docs output from the OpenAPI definition file. This premium version includes Try-it functionality, enhanced search, fast loading, special tags, and other features. You can also compare the difference between Redoc and Redocly here.	Premium
`create-openapi-repo`	CLI tool for splitting a single OpenAPI definition file into multiple files. You can also use this CLI tool to start a new API definition.	Free
`redocly-cli`	CLI tool that provides linting against a customizable ruleset, as well as bundling of the OpenAPI files into a single file. You can also preview the output of your docs with the Redocly API Reference through this tool.	Free
Developer Portal	A sample starter developer portal, built on top of Gatsby. Intended for more robust documentation scenarios. See Developer portal: Introduction for the Redocly documentation, and `developer-portal-starter` for some of the code.	Premium
Workflows	An online Redocly app that provides the full authoring and publishing toolset in the cloud, with GitHub integration and docs-as-code workflows. See https://app.redoc.ly	Premium

This article will focus mostly on the authoring experience working with the OpenAPI definition (using `create-openapi-repo` and `redocly-cli`), as well as publishing using Redoc. For more details about Redocly's Developer Portal and Workflows product, see the Products menu on Redocly's site.

Redocly CLI Walkthrough tutorial

This tutorial will give you a sense of how the Redocly CLI tools work and how to publish the OpenAPI using Redoc. You'll break up an OpenAPI file, make some changes, bundle it back up, and then publish it with Redoc. The tutorial has the following sections:

- 1. Install the prerequisites (p. 251)
- 2. Break up an OpenAPI file (p. 251)
- 3. Check your OpenAPI definition against rules (p. 254)
- 4. Preview the Redoc output (p. 256)
- 5. Bundle up the OpenAPI definition into a single file (p. 258)
- 6. Combine conceptual docs with reference docs (p. 259)
- 7. Explore the premium version: Redocly API Reference (p. 262)
- 8. Publish the reference output (p. 266))

The sections aren't necessarily sequential, but they proceed through Redocly's tools in a logical way to become familiar with different aspects of the products. Also, the tutorial uses a sample OpenAPI definition related to the OpenWeatherMap API (used elsewhere in the course). Also, the instructions were written using a Mac. There might not be any differences with Windows, but note that Windows wasn't used here.

This tutorial is a brief exposure to Redocly's tools only. For more comprehensive documentation, see Redocly's documentation ↗.

1. Install the prerequisites

Before you can work with the Redocly CLI tools, you need to install them. node.js ↗ is a prerequisite to installing both Redocly CLI tools.

1. Install `create-openapi-repo` ↗ following the instructions in How to generate your repository ↗ .

   "npx" is the npm package runner and is the recommended way of running packages that you would run occasionally (see Downloading and installing packages globally ↗). Redocly recommends using npx when installing the `create-openapi-repo` because it's typically used only once.

2. Install `openapi-cli` ↗ following the instructions here: Installation ↗.
3. After you finish installing these CLI tools, restart your terminal.
4. Check that your terminal recognizes the `create-openapi-repo` and `openapi-cli` commands.

2. Break up an OpenAPI file

In this step, you'll break up an existing OpenAPI file into smaller files. We'll use a sample OpenAPI file that we've been using in other places in this course.

1. Download the openapi_openweathermap.yml ↗ file.

   Right-click the file and choose **Save Link As**. Save it in a folder called `redocly_project` in a convenient location. This OpenAPI definition is a small OpenAPI 3.0 file for one API in the OpenWeatherMap API ↗. Note that you can use another OpenAPI definition in this tutorial if you prefer. There's nothing special about this file — it just gives us an OpenAPI definition to work with.

2. Open a terminal, browse to your `redocly_project` directory, and type the following:

   ```
   create-openapi-repo
   ```

   The CLI asks, "Do you already have an OpenAPI/Swagger 3.0 definition for your API? (y/N)"

3. Type **y** and press **return**.

   The CLI asks, "Please specify the path to the OpenAPI definition (local file):"

4. Type `openapi_openweathermap.yml` and press **return**.

   The CLI asks, “API Name: (OpenWeatherMap API).”

5. Press **return**.

   The CLI asks, “The following folders will be created: openapi and docs. You can change them by running `create-openapi-repo <openapiDir> <docsDir>` Proceed? (Y/n)”

6. Type **Y** and press **return**.

   The CLI breaks up your OpenAPI file into smaller files inside an `openapi` file. Redocly also installs adds some `node_modules` folders and initializes a git repository.

7. In a text editor (such as VS Code, Atom, or Sublime Text), open the `redocly_project` folder and browse the contents.

   The folder structure looks like this:

```
└── redocly_project
    ├── .git ...
    ├── .redocly
    ├── LICENSE
    ├── node_modules ...
    ├── README.md
    ├── docs
    │   ├── favicon.png
    │   └── index.html
    ├── openapi
    │   ├── README.md
    │   ├── components
    │   │   ├── README.md
    │   │   ├── parameters
    │   │   │   ├── id.yaml
    │   │   │   ├── lang.yaml
    │   │   │   ├── lat.yaml
    │   │   │   ├── lon.yaml
    │   │   │   ├── mode.yaml
    │   │   │   ├── q.yaml
    │   │   │   ├── units.yaml
    │   │   │   └── zip.yaml
    │   │   ├── schemas
    │   │   │   ├── 200.yaml
    │   │   │   ├── Clouds.yaml
    │   │   │   ├── Coord.yaml
    │   │   │   ├── Main.yaml
    │   │   │   ├── Rain.yaml
    │   │   │   ├── Snow.yaml
    │   │   │   ├── Sys.yaml
    │   │   │   ├── Weather.yaml
    │   │   │   └── Wind.yaml
    │   │   └── securitySchemes
    │   │       └── app_id.yaml
    │   ├── openapi.yaml
    │   └── paths
    │       ├── README.md
    │       └── weather.yaml
    ├── openapi_openweathermap.yml
    ├── package-lock.json
    └── package.json
```

The original OpenAPI definition remains the same: `openapi_openweathermap.yml`. Redocly has extracted each of the paths, parameters, components, and other elements into individual files inside the `openapi` folder.

Redocly creates a new OpenAPI definition file at `openapi/openapi.yaml`. This OpenAPI definition contains references to all of the files that it extracted as external files. If you view this `openapi.yaml` file, you'll see that Redocly has separated out paths with external JSON references (`$ref`), such as the following:

```
paths:
  /weather:
    $ref: paths/weather.yaml
```

The `weather` path is now stored in `openapi/paths/weather.yaml`. When you open this `weather.yaml` file, you'll see that its parameters are also stored in external files:

```
parameters:
  - $ref: ../components/parameters/q.yaml
  - $ref: ../components/parameters/id.yaml
  - $ref: ../components/parameters/lat.yaml
  - $ref: ../components/parameters/lon.yaml
  - $ref: ../components/parameters/zip.yaml
  - $ref: ../components/parameters/units.yaml
  - $ref: ../components/parameters/lang.yaml
  - $ref: ../components/parameters/mode.yaml
```

The OpenAPI definition allows for breaking up files like this and referencing the external files with `$ref`.

Note that `../` refers to another file, whereas `#/` refers to another section on the same page (such as with references to `components`).

Although the `$ref` syntax is standard, most tools don't work well with OpenAPI files separated out. Usually, when you want to generate API documentation, you'll need to feed the preferred API documentation generator a single OpenAPI YAML or JSON file, not a folder of separate files. But when your working on the API definition, it's helpful to separate files to more easily work with the content.

In this OpenAPI scenario, there's just one simple path: `weather`. However, suppose you had a slash, such as `weather/surfing` or something similar. In that case, because `/` is an illegal character in file names, Redocly would store this as `weather@surfing` under the `paths` folder. Forward slashes (`/`) become `@` symbols.

Additionally, if you had a path parameter, such as `weather/{beach_id}`, Redocly uses curly braces `{}` to denote path parameters (rather than a colon `:` which is sometimes used).

See the `README.md` in the `openapi` folder for more details on these conventions. These conventions are recommendations only. You can decide how you reference the paths from `openapi.yaml`.

3. Check your OpenAPI definition against rules

Redocly lets you run a linter against your OpenAPI definition to check for best practices. In this step, you'll break some rules, run the linter, and see the warnings displayed.

1. Open the `.redocly.yaml` file (inside the root directory) to see Redocly's configuration details, including the linting rulesets, themes, and OpenAPI file used:

```yaml
apiDefinitions:
  main: openapi/openapi.yaml
lint:
  extends:
    - recommended
  rules:
    no-unused-components: warning
referenceDocs:
  htmlTemplate: ./docs/index.html
  theme:
    colors:
      primary:
        main: "#32329f"
```

See Configuration file for details about each of these fields. Notice that `main` references the OpenAPI definition file. You could list multiple OpenAPI definitions here. Under `lint`, the `recommended` ruleset is specified.

You can customize how strict the linting rules are, whether to ignore some rules, and make other configurations. For example, the recommended rules check that you follow best practices for completing the information (even if it's optional in the spec) because this information leads to a better output when you generate the Redoc output. You will see warnings if you're missing a `contact` object under `info`, if you're missing a `description` for `tags`, if you're missing `description` properties for each `parameter`, and so on. You can learn more about the linting rules here: Rules.

2. In a terminal prompt, `cd` to your `redocly_project` folder.

3. Run the linter:

   ```
   redocly lint
   ```

   By default, all OpenAPI files listed in your `.redocly.yaml` file's `apiDefinitions` object will be used. These are the command's `entrypoints`. For example, although you just typed `openapi lint`, the command fills in the implied defaults as follows:

   ```
   redocly lint main openapi/openapi.yaml
   ```

   Here's the response when the linter finds that your OpenAPI definition is valid:

   ```
   validating /Users/tomjoht/projects/redocly_project/openapi/openapi.yaml...

   /Users/tomjoht/projects/redocly_project/openapi/openapi.yaml: validated in in 35ms

   Woohoo! Your OpenAPI definition is valid.
   ```

4. To see a generated warning, let's violate some best practices. Open your `openapi/openapi.yaml` file and remove the `license` object:

```
license:
  name: CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)
  url: 'https://openweathermap.org/price'
```

Then save the file.

5. Now run the `openapi lint` command again, and this time you'll see that Redocly identifies a missing license and provides recommendations:

```
validating /Users/tomjoht/projects/redocly_project/openapi/openapi.yaml...
[1] openapi/openapi.yaml:3:3 at #/info

Info object should contain `license` field.

 1 | openapi: 3.0.2
 2 | info:
 3 |   title: OpenWeatherMap API
 4 |   description: >-
 … |   < 14 more lines >
19 |     email: some_email@gmail.com
20 | servers:
21 |   - url: 'https://api.openweathermap.org/data/2.5'

Warning was generated by the info-license rule.


/Users/tomjoht/projects/redocly_project/openapi/openapi.yaml: validated in in 126ms

Woohoo! Your OpenAPI definition is valid. 🎉
You have 1 warning.
```

As your OpenAPI file gets larger and more complex, having a linter to check the definitions against a ruleset can be highly valuable.

(Before continuing on, restore the `license` object in the `openapi.yaml` file that you previously removed.)

4. Preview the Redoc output

The CLI contains a Redoc documentation preview server that listens for changes and rebuilds when it detects a change. To run the preview server:

1. Type the following command:

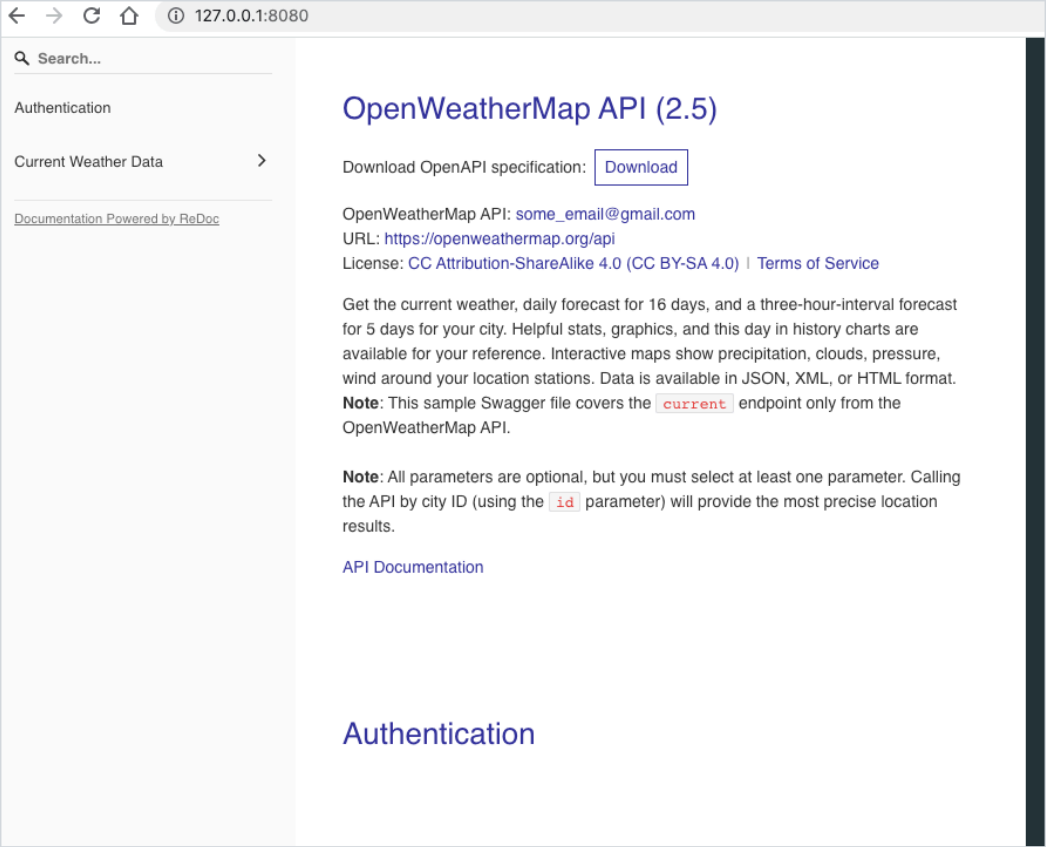
```
redocly preview-docs
```

The response is as follows:

```
Using Redoc community edition.
Login with openapi-cli login or use an enterprise license key to prev
iew with the premium docs.


  🔎  Preview server running at http://127.0.0.1:8080

Bundling...


  👀  Watching /Users/tomjoht/projects/redocly_project/openapi/openap
i.yaml and all related resources for changes

Created a bundle for /Users/tomjoht/projects/redocly_project/openapi/
openapi.yaml successfully
GET /: 43.289ms
GET /hot.js: 22.313ms
GET /simplewebsocket.min.js: 34.003ms
GET /openapi.json: 0.507ms
GET /favicon.png: 1.720ms
```

Redoc bundles up your individual files into a single OpenAPI definition and passes it into the Redoc documentation generator. By default, you're using the community edition of Redoc, which lacks some of the features of the pro version. (The differences are explained in more detail later in this section and also in 7. Explore the premium version: Redocly API Reference (p. 262).)

2. Go to `http://127.0.0.1:8080` in your browser to see a preview of the output:



OpenWeatherMap API (2.5)

Download OpenAPI specification: Download

OpenWeatherMap API: some_email@gmail.com
URL: https://openweathermap.org/api
License: CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0) | Terms of Service

Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

API Documentation

Authentication

3. To see the auto-regeneration, go into one of your files in `openapi` and make a change. For example go to `openapi/components/parameters/lat.yaml` and change a line in the description, and then save the file.

   The preview server rebuilds the output and reloads the page.

4. Press **Ctrl+C** to stop the preview server.

   Note that we’re currently using the Community Edition of Redoc. We’ll explore more features in the Pro version in 7. Explore the premium version: Redocly API Reference (p. 262).

5. Bundle up the OpenAPI definition into a single file

Let’s bundle the separate files into a single OpenAPI definition file. You might want to do this if you’re done working with your file and want to generate out the output to feed into another tool to render the documentation. To bundle up the OpenAPI files:

1. Run the following command:

   ```
   redocly bundle main -o dist.json
   ```

   `main` refers to the OpenAPI definition you want bundled, as specified in your `.redocly.yaml` file’s `apiDefinitions` object. The `-o dist.json` specifies the output file. After running the command, you see the following:

```
bundling openapi/openapi.yaml...
📦 Created a bundle for openapi/openapi.yaml at dist.json in 38ms.
```

2. Open `dist.json` to see that it contains all the references packaged into a single file.

Now let’s dive into some more features related to the authoring experience: how to integrate conceptual docs with reference docs.

6. Combine conceptual docs with reference docs

One common question in writing API tech docs is how to combine conceptual docs (e.g., getting started tutorials, information about authorization, troubleshooting, etc.) with the reference docs. Most API documentation generators generate reference docs only.

Redocly has some unique offerings when it comes to organizing and working with conceptual content. One integration strategy is to put all of your conceptual information into the `description` field of your `info` object, and then store that `description` content in an external file. Any heading 1 and 2 tags (`#` and `##`) in your `description` will become separate entries in the sidebar. Let’s walk through this.

1. In our sample redocly project, open `openapi.yaml` and change the `description` to a `$ref` pointing to a separate Markdown file. Specifically, identify this section:

```
openapi: 3.0.2
info:
  title: OpenWeatherMap API
  description: >-
    Get the current weather, daily forecast for 16 days, and a
    three-hour-interval forecast for 5 days for your city. Helpful stats,
    graphics, and this day in history charts are available for your reference.
    Interactive maps show precipitation, clouds, pressure, wind around your
    location stations. Data is available in JSON, XML, or HTML format. **Note**:
    This sample Swagger file covers the `current` endpoint only from the
    OpenWeatherMap API. <br/><br/> **Note**: All parameters are optional, but
    you must select at least one parameter. Calling the API by city ID (using
    the `id` parameter) will provide the most precise location results.
```

And change it to this:

```
openapi: 3.0.2
info:
  title: OpenWeatherMap API
  description:
  $ref: description.md
```

Note that `$ref: description.md` is technically incorrect with the OpenAPI spec, but the Redocly bundle process places that Markdown inline to correct it.

For more detail on using `$ref`, see Using JSON references ($refs).

2. Create a file called `description.md` under the `openapi` folder and paste the description content there.
3. Run the preview server:

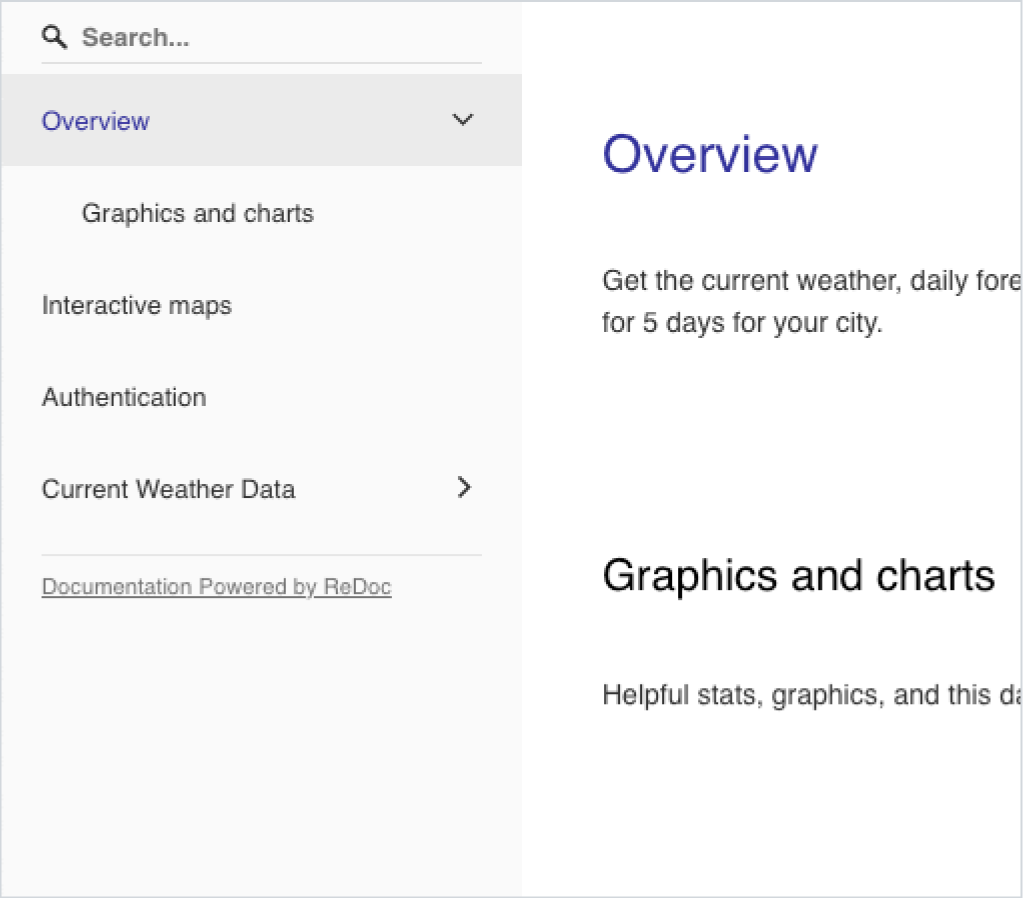
```
redocly preview-docs
```

4. Go to http://127.0.0.1:8080. The description appears the same as before, but it's now a bit easier to work with because it's in a standalone file. You don't have worry about maintaining proper indentation in YAML. You can write more freely. As with the OpenAPI spec, the Markdown flavor supported is Commonmark Markdown.
5. In your `description.md` file, add some heading tags, like this:

```
# Overview

Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city.

## Graphics and charts

Helpful stats, graphics, and this day in history charts are available for your reference.

# Interactive maps

Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format.

**Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API. <br/><br/> **Note**: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.
```

6. Save the file and check the reload in the preview server.

In the preview server, notice how the heading levels are reflected in the sidebar.



Overview

Get the current weather, daily fore
for 5 days for your city.

Graphics and charts

Helpful stats, graphics, and this da

First-level headings become parent-level sidebar entries. Second-level headings become child-level sidebar entries to the parents. (Deeper levels aren't supported.)

7. You can also re-use some content from your OpenAPI definition. In your description file, add this tag:

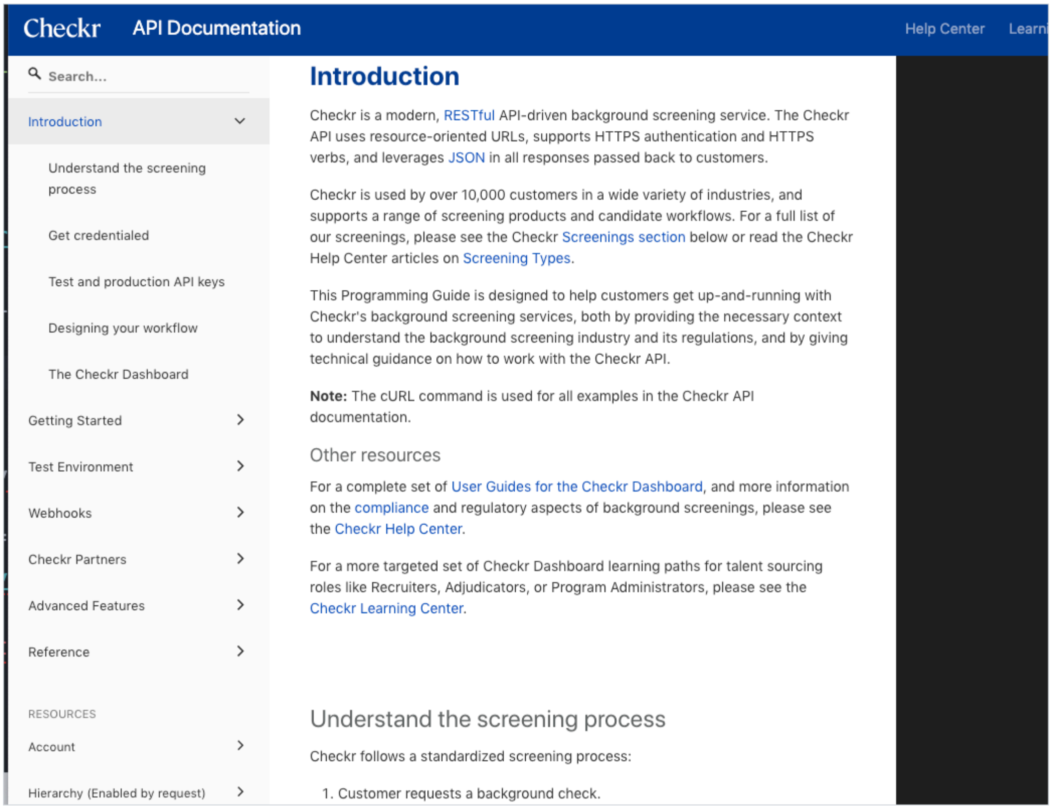
   ```
   <SecurityDefinitions />
   ```

8. Save the file and check the reload in the preview server.

   The security object from your OpenAPI definition appears directly in your description content.

   The pro version lets you embed other tags and also push content into the right menu. We'll explore that in the next section.

   For an example of fairly robust API documentation that uses the sidebar for a lot of conceptual documentation, see the Checkr documentation.



Introduction

Checkr is a modern, RESTful API-driven background screening service. The Checkr API uses resource-oriented URLs, supports HTTPS authentication and HTTPS verbs, and leverages JSON in all responses passed back to customers.

Checkr is used by over 10,000 customers in a wide variety of industries, and supports a range of screening products and candidate workflows. For a full list of our screenings, please see the Checkr Screenings section below or read the Checkr Help Center articles on Screening Types.

This Programming Guide is designed to help customers get up-and-running with Checkr's background screening services, both by providing the necessary context to understand the background screening industry and its regulations, and by giving technical guidance on how to work with the Checkr API.

Note: The cURL command is used for all examples in the Checkr API documentation.

Other resources

For a complete set of User Guides for the Checkr Dashboard, and more information on the compliance and regulatory aspects of background screenings, please see the Checkr Help Center.

For a more targeted set of Checkr Dashboard learning paths for talent sourcing roles like Recruiters, Adjudicators, or Program Administrators, please see the Checkr Learning Center.

Understand the screening process

Checkr follows a standardized screening process:

1. Customer requests a background check.

If integrating your conceptual content like this doesn't work for your doc needs, Redocly offers a more robust option: Redocly Developer Portals. The Redocly Developer Portals use Gatsby as a static site generator to give you more complete control and customization for non-reference content. Gatsby also provides many pre-built React components, which makes building out other features of a developer portal, such as marketing landing pages and other UI elements in a console, much easier.

7. Explore the premium version: Redocly API Reference

The lack of the Try-it explorer in the Redoc community edition is a major reason to upgrade to the premium version (Redocly API Reference). As I've argued elsewhere in this course, the try-it functionality makes API documentation both interactive and interesting. In this tutorial, we've been using the community edition of Redoc. The premium version, called Redocly API Reference, provides several key benefits:

- **The Try-it-out explorer**. You can execute calls in the browser. This interactivity is what makes REST API docs so powerful and instructive.
- **Optimized loading**. The output is split into multiple files to render loading more quickly (up to 10x faster).
- **Special tags**. You can use special tags to pull in content from your reference content (such as your security definition or a response).
- **Right pane**. You can push code into the right pane so that it appears juxtaposed with the conceptual info in the main column.
- **Enhanced search**. Search is more comprehensive and detailed, including results for request and response parameters, among other enhancements.

Here's a short video explaining the top 3 reasons why people choose the pro version:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/pubapis_redocly_tutorial.html] to view the content.

If you have a license for the pro version, you would generate an API key and then use the `openapi login` command to unlock the pro features. (The `openapi logout` command clears the API key, reverting you to the community edition.) Pricing for different versions of Redocly is available on their Pricing page.

Let's enable the premium version features and see how they work. (You won't have to purchase anything to explore the features locally.)

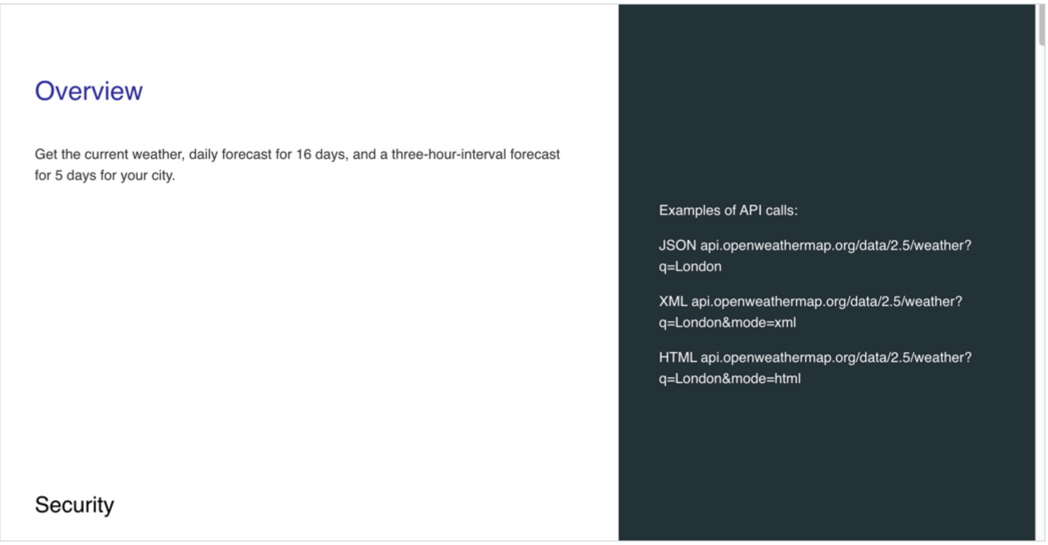
1. Sign in to Redocly.
2. Go to your Redocly profile at https://app.redoc.ly/profile and create a Personal API key. Copy the key.
3. Type the following:

   ```
   redocly login
   ```

4. When prompted for a password, paste the password into the terminal and press **Enter**. Then, authorization begins. [Q: Are you prompted for your API key at this point?]
5. In your `description.md` file, add some `PullRight` tags with some sample code, such as the following:

   ```
   <PullRight>
   Examples of API calls:

   JSON api.openweathermap.org/data/2.5/weather?q=London

   XML api.openweathermap.org/data/2.5/weather?q=London&mode=xml

   HTML api.openweathermap.org/data/2.5/weather?q=London&mode=html
   </PullRight>
   ```

6. View the preview server (http://127.0.0.1:8080/) and see the code populate in the right column.



Overview

Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city.

Examples of API calls:

JSON api.openweathermap.org/data/2.5/weather?q=London

XML api.openweathermap.org/data/2.5/weather?q=London&mode=xml

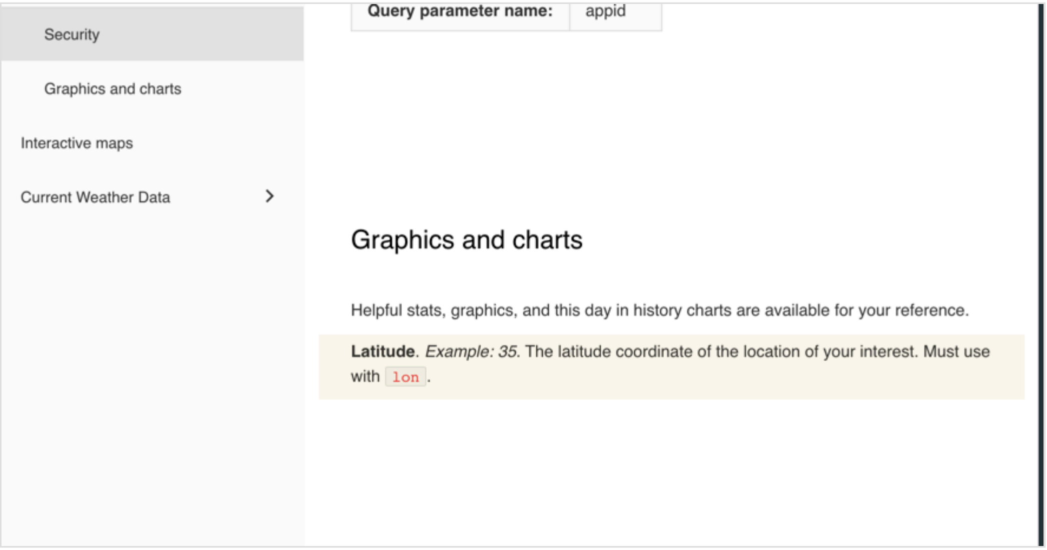
HTML api.openweathermap.org/data/2.5/weather?q=London&mode=html

Security

7. In your `description.md` file, re-use a parameter description from your reference:

```
<RedocResponse pointer={"#/components/parameters/lat"} />
```

8. View the preview server and see the `lat` parameter appear (highlighted below).



Graphics and charts

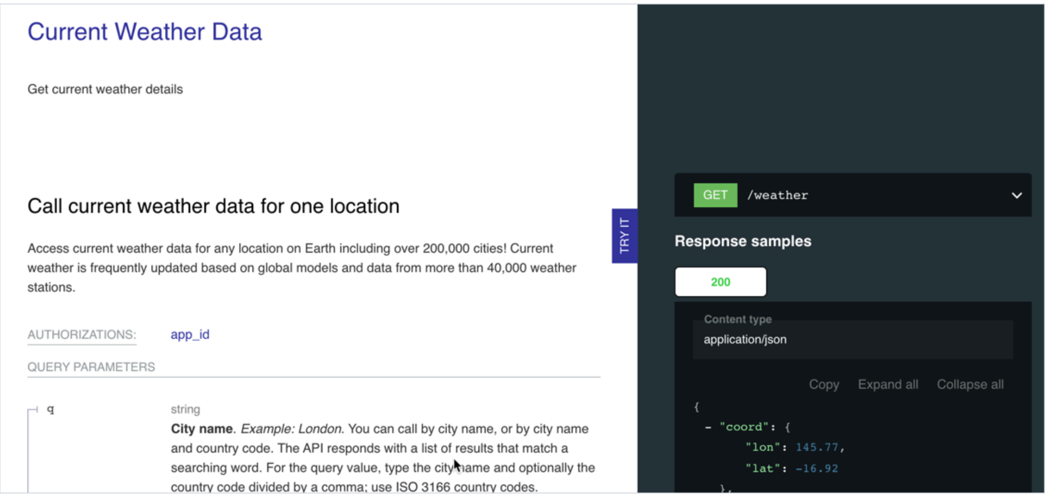
Helpful stats, graphics, and this day in history charts are available for your reference.

Latitude. *Example: 35*. The latitude coordinate of the location of your interest. Must use with `lon`.

9. Open `.redocly.yaml` and add `showConsole: true` under `referenceDocs`:

```
referenceDocs:
  htmlTemplate: ./docs/index.html
  showConsole: true
```

10. View the preview server and look for the blue “TRY IT” tab next to the weather path:



Current Weather Data

Get current weather details

Call current weather data for one location

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

AUTHORIZATIONS: app_id

QUERY PARAMETERS

q string

City name. *Example: London*. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by a comma; use ISO 3166 country codes.

Content type application/json

```
{
 - "coord": {
       "lon": 145.77,
       "lat": -16.92
```

11. Click the **TRY IT** tab. An API explorer replaces the right pane with options to configure a request.
12. For the `zip` parameter, enter `95050` (or another zip code).

13. Click the **Auth (app_id)** tab in the upper-right and enter your OpenWeatherMap API key. (If you don't have one, use `126cac1a482f51de0f1287b45ae2bf9a`.)
14. Click **Send**.

    A successful response returns JSON such as the following:

```
{
  "coord": {
    "lon": -121.98,
    "lat": 37.35
  },
  "weather": [
    {
      "id": 711,
      "main": "Smoke",
      "description": "smoke",
      "icon": "50n"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 65.48,
    "feels_like": 62.38,
    "temp_min": 63,
    "temp_max": 68,
    "pressure": 1016,
    "humidity": 68
  },
  "visibility": 10000,
  "wind": {
    "speed": 8.05,
    "deg": 10
  },
  "clouds": {
    "all": 40
  },
  "dt": 1596517361,
  "sys": {
    "type": 1,
    "id": 5845,
    "country": "US",
    "sunrise": 1596460471,
    "sunset": 1596510811
  },
  "timezone": -25200,
  "id": 0,
  "name": "Santa Clara",
  "cod": 200
}
```

    You can learn more about the Try it feature here: The try it API console.

8. Publish the reference output

So far we've been exploring the API reference output from within the preview server. Now let's publish the reference output outside of the preview server:

1. Bundle the OpenAPI definition.

   ```
   redocly bundle main -o dist.json
   ```

2. Inside a folder called `redoc`, create an HTML file called `index.html`.
3. In your `index.html` file, insert the following template for rendering the OpenAPI definition using Redoc JS delivered through a CDN.

   ```
   <!DOCTYPE html>
   <html>
     <head>
       <title>ReDoc</title>
       <!-- needed for adaptive design -->
       <meta charset="utf-8"/>
       <meta name="viewport" content="width=device-width, initial-scale=1">
       <link href="https://fonts.googleapis.com/css?family=Montserrat:300,400,700|Roboto:300,400,700" rel="stylesheet">

       <!--
       ReDoc doesn't change outer page styles
       -->
       <style>
         body {
           margin: 0;
           padding: 0;
         }
       </style>
     </head>
     <body>
       <redoc spec-url='http://petstore.swagger.io/v2/swagger.json'></redoc>
       <script src="https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js"> </script>
     </body>
   </html>
   ```

   If you have a license for the premium version (the Redocly API Reference) and you want the Try-it console enabled, you would use the code described here: Example with "Try It" console enabled in the JavaScript CDN usage (self-hosted) page. However, because we're just publishing with the community edition version of Redoc, we'll use the code described in the Redoc GitHub page.

4. Replace `http://petstore.swagger.io/v2/swagger.json` with a reference to `dist.json`.

   ```
     <redoc spec-url='dist.json'></redoc>
   ```

5. Drag the `dist.json` file generated in step 1 into your `redoc` folder so that the reference is valid.
6. To view the output, you can either upload the files onto a web server, or you can simulate an HTTP server locally. To simulate an HTTP server locally, you can use Python Simple HTTP Server. You can view detailed steps for running this server here: Activity: View local OpenAPI file in Swagger UI (p. 223). In short, if you already have Python3 installed, `cd` to your `redoc` folder and run this command: `python3 -m http.server`. Then browse to port `http://localhost:8000/` and view the Redoc output.

Redoc provides a lot of options to configure the output. Options to configure the community edition of Redoc are here: `<redoc>` options object. Options to configure the Redocly API Reference version are here: Customization.

For sample API reference docs built with Redoc, see Some Real-life usages on the Redoc GitHub page.

Conclusion

Redoc provides one of the most attractive outputs for REST API docs. In a recent survey about trends in developer documentation (p. 44), I asked developers “How do you render the OpenAPI specification into documentation?” Although Swagger UI (p. 218) was the most common (used 27% of the time), this is likely because it’s free. The premium version of Swagger (called SwaggerHub (p. 230)) was used only 4% of the time. In contrast, Redoc was used 8% of the time.

Note: This content doesn’t embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/pubapis_redocly_tutorial.html] to view the content.

Overall, the output from Redoc does seem more polished and user-friendly than other API reference doc outputs. For the interactivity feature alone I would definitely recommend using the premium version (Redocly API Reference) rather than the community edition. Redocly’s CLI-based toolset provides a way to work more efficiently in the code, supporting a docs-as-code workflow (p. 551). If this command-line-based toolset and three-column output appeal to you, Redoc might be the way to go. You can start a free trial here or learn more here: Redocly.

Redoc Community Edition Basic Demo

Redocly offers robust tools for authoring and publishing API docs using the command-line and docs-as-code workflows. For a full tutorial, see Redocly tutorial – authoring and publishing API docs with Redocly's command-line tools (p. 249). The tutorial that follows here is a simple way to use the community/free edition of Redoc.

Redoc is a popular display framework for the OpenAPI as well and is a favorite by many API gurus. Redoc started out as a API doc site for Rebilly docs but with many customizations evolved into its own unique framework. Redoc provides a three-panel layout made popular by companies such as Stripe.

The makers of Redoc spun off into their own company (Redocly) and developed a more robust hosted solution offering (see From ReDoc to Redoc.ly). As with both Stoplight and Swagger UI, Redoc offers both free and pro versions to adjust to your needs. And you can add topics beyond just reference docs.

Activity: Use Redoc Community Edition to render OpenAPI spec

In this activity, you'll render your OpenAPI spec with Redoc.

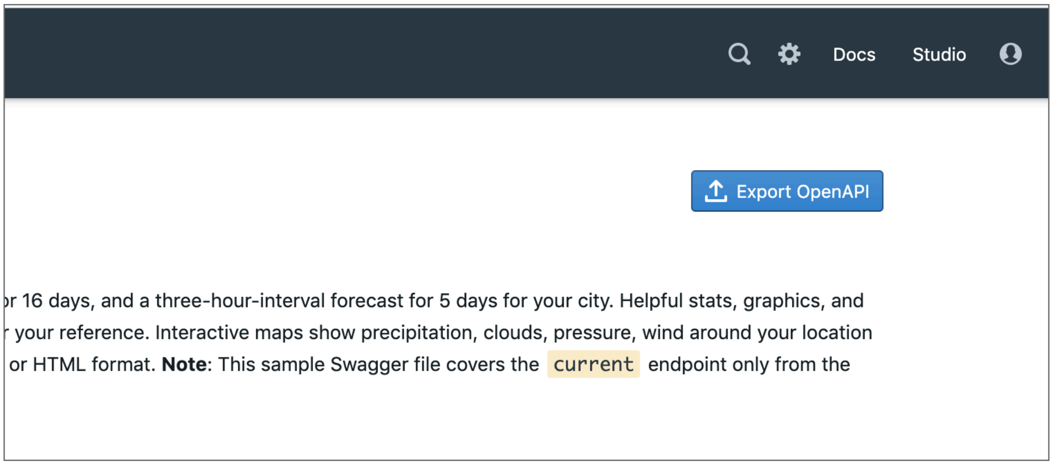
1. Copy the following code into an empty text file (in a text editor such as Sublime Text) and save the file as **redoc.html**. (Note: This code comes from Redoc's "TL;DR" readme instructions on GitHub.)

```
<!DOCTYPE html>
<html>
  <head>
    <title>ReDoc</title>
    <!-- needed for adaptive design -->
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="https://fonts.googleapis.com/css?family=Montserrat:300,400,700|Roboto:300,400,700" rel="stylesheet">

    <!--
    ReDoc doesn't change outer page styles
    -->
    <style>
      body {
        margin: 0;
        padding: 0;
      }
    </style>
  </head>
  <body>
    <redoc spec-url='https://idratherbewriting.com/docs/openapi_spec_and_generated_ref_docs/openapi_openweathermap.yml'></redoc>
    <script src="https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js"> </script>
  </body>
</html>
```

Make sure you save the file with a `.html` extension so that your browser will render it as HTML.

2. Change the value for `redoc spec-url` to a valid OpenAPI spec that is available at an online URL.

   If you were following the previous tutorial on creating an OpenAPI spec through Stoplight Studio (Getting started tutorial: Using Stoplight Studio to create an OpenAPI specification document (p. 201)), you can use the OpenAPI URL that Stoplight provides. From your Stoplight Studio display, go to the Overview topic and click **Export OpenAPI**. Choose either original or dereferenced. Copy the URL. Here's an example URL.

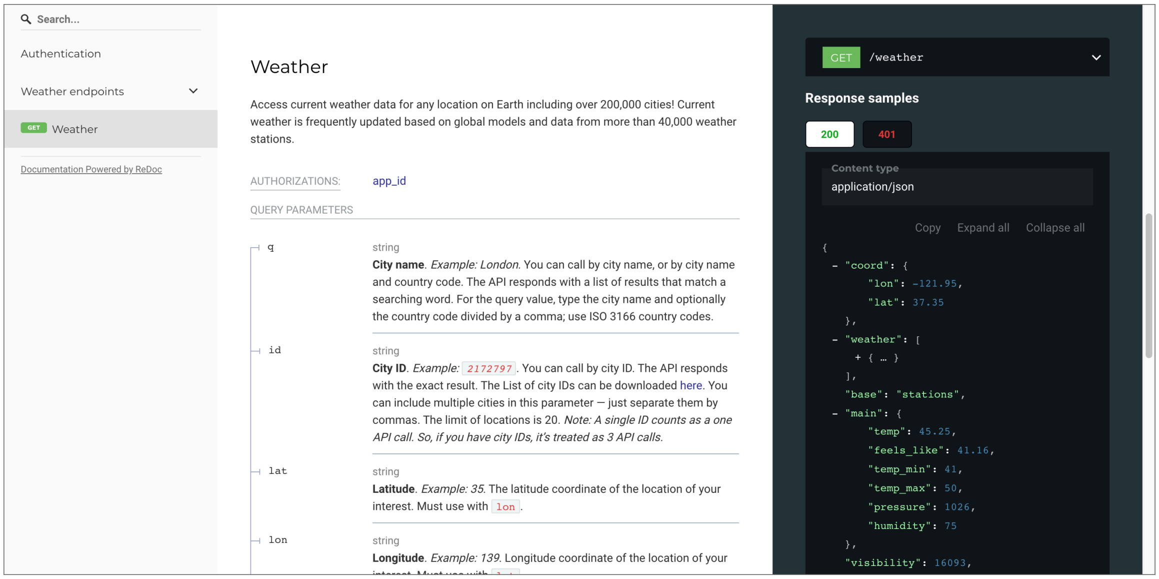


r 16 days, and a three-hour-interval forecast for 5 days for your city. Helpful stats, graphics, and
your reference. Interactive maps show precipitation, clouds, pressure, wind around your location
or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the

Exporting an OpenAPI URL

3. In your web browser, click **Cmd + O** (Mac) or **Ctrl + O** (Windows), locate your **redoc.html** file, and select it.

The display should look as follows:



Weather

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

AUTHORIZATIONS: app_id

QUERY PARAMETERS

q string
City name. *Example: London*. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by a comma; use ISO 3166 country codes.

id string
City ID. *Example:* `2172797`. You can call by city ID. The API responds with the exact result. The List of city IDs can be downloaded here. You can include multiple cities in this parameter — just separate them by commas. The limit of locations is 20. *Note: A single ID counts as a one API call. So, if you have city IDs, it's treated as 3 API calls.*

lat string
Latitude. *Example: 35*. The latitude coordinate of the location of your interest. Must use with `lon`.

lon string
Longitude. *Example: 139*. Longitude coordinate of the location of your

Content type
application/json

```
{
 - "coord": {
       "lon": -121.95,
       "lat": 37.35
   },
 - "weather": [
     + { … }
   ],
   "base": "stations",
 - "main": {
       "temp": 45.25,
       "feels_like": 41.16,
       "temp_min": 41,
       "temp_max": 50,
       "pressure": 1026,
       "humidity": 75
   },
   "visibility": 16093,
```

Sample Redoc display

Chapter 5: Step-by-step OpenAPI code tutorial

If you want to build the OpenAPI spec by hand in a text editor, this tutorial will help you get started. Even though documentation isn't necessarily linear, I break down the various sections in the OpenAPI YAML file and go through them line by line. Approaching the OpenAPI file manually requires a more advanced understanding of the spec, YAML or JSON syntax, and how to approach strategies specific to your API. If you're a beginner or want a visual editor, check out the Getting started tutorial: Using Stoplight Studio to create an OpenAPI specification document (p. 201) instead.

Chapter sections:

OpenAPI tutorial using Swagger Editor and Swagger UI: Overview 272
Working in YAML 276
Step 1: The openapi object 282
Step 2: The info object 285
Step 3: The servers object 288
Step 4: The paths object 291
Step 5: The components object 302
Step 6: security object 327
Step 7: The tags object 334
Step 8: The externalDocs object 337
Activity: Create an OpenAPI specification document 340

OpenAPI tutorial using Swagger Editor and Swagger UI: Overview

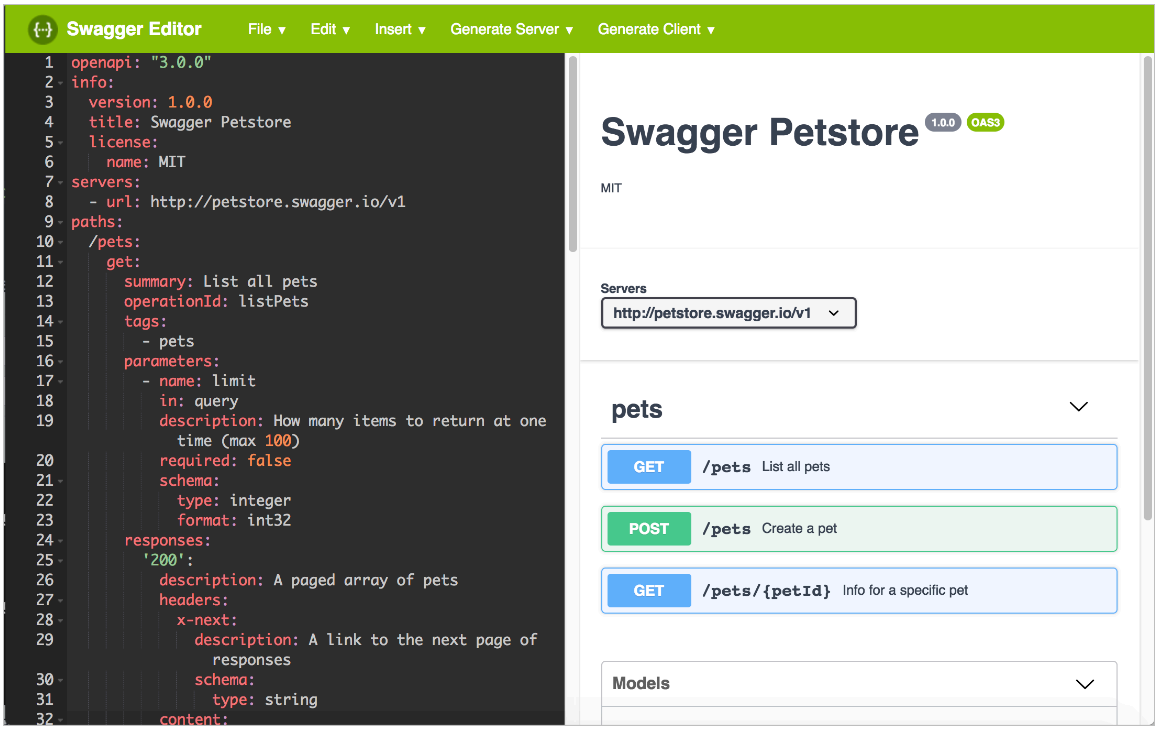
In the previous section, I explained how to create the OpenAPI spec using a visual editor with Stoplight Studio (p. 201). In this tutorial, I'll explain how to work in a text editor such as Swagger Editor to write the OpenAPI code by hand. For a higher-level introduction to the OpenAPI spec, see Introduction to the OpenAPI specification (p. 181).

In this tutorial, we'll dive deeply into the OpenAPI specification. We'll use the same OpenWeatherMap API that we've been using throughout other parts of this course as the content for our OpenAPI document. Using this API, we'll create a valid OpenAPI specification document and then render it using Swagger UI.

To see a presentation that covers the same concepts in this article, see https://goo.gl/n4Hvtq.

Swagger Editor

When choosing an editor to write OpenAPI code by hand, the most common is the Swagger Editor because it dynamically validates your content as you write. It lets you determine whether the specification document you're creating is valid. The Swagger Editor looks like this, with the left pane showing the code and the right pane showing the output:



```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: A paged array of pets
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
```

The Swagger Editor validates your specification content dynamically and shows you the display on the right

While you're coding in the Swagger Editor, if you make an error, you can quickly fix it before continuing rather than waiting until a later time to run a build and sort out errors.

JSON or YAML format

For your specification document's format, you have the choice of working in either JSON or YAML. The code sample is in the previous screenshot shows YAML. YAML refers to "YAML Ain't Markup Language," meaning YAML doesn't have any markup tags (`< >`), as would be common with other markup languages such as XML.

YAML depends on spacing and colons to establish the object syntax. This space-sensitive formatting makes the code more human-readable, but it's also sometimes trickier to get the spacing right.

You can also write in JSON, if you prefer that. There are more curly braces to deal with, but it isn't a space-sensitive format. The OpenAPI specification documentation on GitHub shows code samples in both YAML and JSON in nearly every example. I'll go into more detail about YAML in the next step, Working with YAML (p. 276).

General resources for learning the OpenAPI specification

Learning the OpenAPI specification will take some time. As an estimate, if you're coding it manually, plan about two weeks of immersion, working with a specific API in the context of the specification before you become comfortable with it. Remember that the OpenAPI specification is general enough to describe nearly every REST API, so some parts may be more applicable to your API than others.

As you learn the OpenAPI specification, use the following resources:

- Sample OpenAPI specification documents. These sample specification documents provide a good starting point as a basis for your specification document. They give you a big picture of the general shape of a specification document.
- Swagger user guide. The Swagger user guide is more friendly, conceptual, and easy to follow. It doesn't have the detail and exactness of the specification documentation on GitHub, but in many ways, it's clearer and contains more examples.
- OpenAPI specification documentation. The specification documentation is technical and takes a little getting used to, but you'll no doubt consult it frequently when describing your API. It's a long, single page document to facilitate findability through Ctrl+F.

There are other Swagger/OpenAPI tutorials online, but make sure you follow tutorials for the 3.0 version of the API rather than 2.0. Version 3.0 was released in July 2017. 3.0 is substantially different from 2.0. (Version 3.0.2 was released in December 2017 and makes minor improvements to 3.0. Note that whenever I refer to 3.0, I'm referring to 3.x, meaning any incremental dot release from the 3.0 line.)

How my OpenAPI/Swagger tutorial is different

You can find many Swagger tutorials online. What makes mine different? Besides the end-to-end walkthrough using the OpenAPI 3.0 (rather than 2.0) version of the spec, and using an actual API for context, I also show how OpenAPI fields map to Swagger UI. Specifically, I demonstrate how and where each of the OpenAPI fields gets displayed.

Many other display frameworks besides Swagger UI can parse and display information in an OpenAPI specification document, and you can even create your own custom parsing tools.

Swagger UI is sponsored by Smartbear, the same company that is heavily invested in the OpenAPI initiative and which develops SwaggerHub (p. 230) (the premium version of Swagger UI) and drives other Swagger tooling (including Swagger Editor, Swagger UI, Swagger Codegen, and others). Note that SmartBear does not own the OpenAPI specification, as the Linux Foundation drives this initiative. The OpenAPI spec's development is driven by many companies and organizations.

By showing you how the fields in the spec appear in the Swagger UI output, I hope the specification objects and properties will take on more relevance and meaning.

Note that SmartBear is one of the sponsors of this site.

Terminology for Swagger and OpenAPI

Before continuing, I want to clarify the difference between "Swagger" and "OpenAPI" terms for those who may be unfamiliar with this landscape. "Swagger" ↗ was the original name of the OpenAPI specification, but the specification was later changed to "OpenAPI" ↗ to reinforce the open, non-proprietary nature of this standard. Now, "Swagger" refers to API tooling that supports the OpenAPI spec, not the spec itself.

People still often refer to both names interchangeably, but "OpenAPI" is how the spec should be referred to. The "OpenAPI specification document" or "OpenAPI document" is the Swagger YAML file that you create to describe your API. For other terms, see the API Glossary (p. 55). See also What Is the Difference Between Swagger and OpenAPI? ↗.

Customizing Swagger UI

You might be concerned that Swagger UI outputs look similar. First, note that you can render the OpenAPI spec using many different tools outside of Swagger UI (e.g., using Stoplight (p. 201) or Redoc (p. 249)). However, Swagger UI tends to be popular because it's free and was one of the first in this space.

With my OpenAPI projects, I usually customize the Swagger UI's colors a bit, add a custom logo and a few other custom styles. With one project, I integrated Bootstrap ↗ so that I could have modals ↗ where users could generate their authorization codes. You can even add collapse-and-expand features ↗ in the `description` element to provide more information to users.

Beyond these simple modifications, however, it takes a bit of web-developer prowess to significantly alter the Swagger UI display. It's possible, but you need web development skills.

Start by looking at the big picture

If you would like to get a big picture of the specification document, take a look at the 3.0 examples here ↗, specifically the Petstore OpenAPI specification document ↗. It probably won't mean much at first, but try to get a sense of the whole before we dive into the details. Look at some of the other samples in the v.3.0 folder as well.

Follow the OpenAPI tutorial

The OpenAPI tutorial has a brief into to YAML followed by eight steps. Each step corresponds with one of the root-level objects in the OpenAPI document.

- Working in YAML (p. 276)
- Step 1: The openapi object (p. 282)
- Step 2: The info object (p. 285)
- Step 3: The servers object (p. 288)
- Step 4: The paths object (p. 291)
- Step 5: The components object (p. 302)
- Step 6: security object (p. 327)
- Step 7: The tags object (p. 334)
- Step 8: The externalDocs object (p. 337)

You don't have to create the specification document in this order; I've merely chosen this order to provide more of a specific path and series of steps to the process.

In the following sections, we'll proceed through each of these objects one by one and document the OpenWeatherMap current API. Tackling each root-level object individually (rather than documenting everything at once) helps reduce the complexity of the spec.

`components` is more of a storage object for schemas defined in other objects, but to avoid introducing too much at once, I'll wait until the `components` tutorial (p. 302) to fully explain how to reference a schema in one object (using `$ref`) that points to the full definition in `components`.

With each step, you'll paste the object you're working on into the Swagger Editor. The right pane of the Swagger Editor will show the Swagger UI display. (Remember that the specification document alone does nothing with your content. Other tools are required to read and display the spec document.)

Later, when I talk more about publishing, I'll explain how to configure Swagger UI with your specification document as a standalone output. For our sample OpenWeatherMap API, you can see the OpenAPI spec (`openapi_openweathermap.yml`) rendered by the Swagger UI at the following links:

- Standalone Swagger UI with OpenWeatherMap API
- Embedded Swagger with OpenWeatherMap API (p. 240)

Migrating from OpenAPI 2.0 to 3.0

If you have an existing specification document that validates against version OpenAPI 2.0 and you want to convert it to OpenAPI 3.0 (or vice versa), you can use APIMATIC's Transformer to convert it automatically. (You can also use APIMATIC to transform your specification document into many other outputs, such as RAML (p. 924), API Blueprint (p. 938), or Postman (p. 71).)

To see the difference between the 2.0 and the 3.0 code, you can copy these code samples to separate files and then use an application like Diffmerge to highlight the differences. The Readme.com blog has a nice post that provides A Visual Guide to What's New in Swagger 3.0.

Helpful resources

As you embark on creating an OpenAPI specification file, you might find the recording of Peter Gruenbaum's Swagger/OpenAPI presentation to the STC Puget Sound chapter helpful, as well as his Udemy course.

Brace yourself — this is where you'll find out if you're cut out for API technical writing.

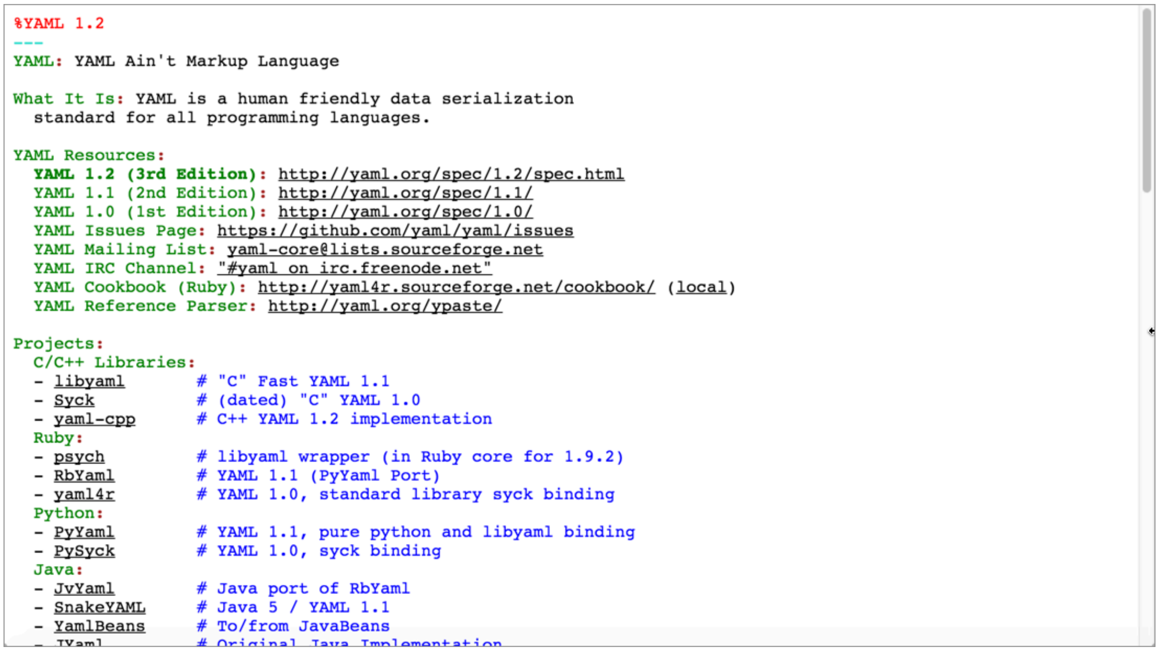
Working in YAML (OpenAPI tutorial)

Before we dive into the steps of the OpenAPI Tutorial (p. 272), it will help to have a better grounding in YAML, since this is the most common syntax for the OpenAPI specification document. (You can also use JSON (p. 94), but the prevailing trend with the OpenAPI document format is YAML.)

YAML stands for "YAML Ain't Markup Language." This means that the YAML syntax doesn't have markup tags such as `<` or `>`. Instead, it uses colons to denote an object's properties and hyphens to denote an array.

Working with YAML

YAML is easier to work with because it removes the brackets, curly braces, and commas that get in the way of reading content.



```
%YAML 1.2
---
YAML: YAML Ain't Markup Language

What It Is: YAML is a human friendly data serialization
  standard for all programming languages.

YAML Resources:
  YAML 1.2 (3rd Edition): http://yaml.org/spec/1.2/spec.html
  YAML 1.1 (2nd Edition): http://yaml.org/spec/1.1/
  YAML 1.0 (1st Edition): http://yaml.org/spec/1.0/
  YAML Issues Page: https://github.com/yaml/yaml/issues
  YAML Mailing List: yaml-core@lists.sourceforge.net
  YAML IRC Channel: "#yaml on irc.freenode.net"
  YAML Cookbook (Ruby): http://yaml4r.sourceforge.net/cookbook/ (local)
  YAML Reference Parser: http://yaml.org/ypaste/

Projects:
  C/C++ Libraries:
  - libyaml       # "C" Fast YAML 1.1
  - Syck          # (dated) "C" YAML 1.0
  - yaml-cpp      # C++ YAML 1.2 implementation
  Ruby:
  - psych         # libyaml wrapper (in Ruby core for 1.9.2)
  - RbYaml        # YAML 1.1 (PyYaml Port)
  - yaml4r        # YAML 1.0, standard library syck binding
  Python:
  - PyYaml        # YAML 1.1, pure python and libyaml binding
  - PySyck        # YAML 1.0, syck binding
  Java:
  - JvYaml        # Java port of RbYaml
  - SnakeYAML     # Java 5 / YAML 1.1
  - YamlBeans     # To/from JavaBeans
  - JYaml         # Original Java Implementation
```

The YAML site itself is written using YAML, which you can immediately see is not intended for coding web pages.

YAML is an attempt to create a more human-readable data exchange format. It's similar to JSON (which is actually a subset of YAML) but uses spaces, colons, and hyphens to indicate the structure.

Many computers ingest data in a YAML or JSON format. It's a syntax commonly used in configuration files and an increasing number of platforms (like Jekyll), so it's a good idea to become familiar with it.

YAML is a superset of JSON

For the most part, YAML and JSON are different ways of structuring the same data. Dot notation accesses the values the same way. For example, the Swagger UI can read the `openapi.json` or `openapi.yaml` files equivalently. Pretty much any parser that reads JSON will also read YAML. However, some JSON parsers might not read YAML because there are a few features YAML has that JSON lacks (more on that below (p. 279)).

YAML syntax

With a YAML file, spacing is significant. Each two-space indent represents a new level:

```
level1:
  level2:
    level3:
```

Each new level is an object. In this example, the `level1` object contains the `level2` object, which contains the `level3` object.

With YAML, you generally don't use tabs (since tab spacing is non-standard). Instead, you space twice.

Each level can contain either a single key-value pair (also referred to as a *dictionary* in YAML lingo) or a *sequence* (a list of hyphens):

```
level3:
  -
    itema: "one"
    itemameta: "two"
  -
    itemb: "three"
    itembmeta: "four"
```

The values for each key can optionally be enclosed in quotation marks. If your value has a colon or quotation mark in it, enclose it in quotation marks.

Comparing JSON to YAML

Earlier in the course, we looked at various JSON structures involving objects and arrays (p. 94). So let's look at the equivalent YAML syntax for each of these same JSON objects.

You can use Unserialize.me to make the conversion from JSON to YAML or YAML to JSON.

Here are some key-value pairs in JSON:

```
{
"key1":"value1",
"key2":"value2"
}
```

Here's the same structure expressed in YAML syntax:

```
key1: value1
key2: value2
```

Here’s an array (list of items) in JSON:

```
["first", "second", "third"]
```

In YAML, the array is formatted as a list with hyphens:

```
- first
- second
- third
```

Here’s an object containing an array in JSON:

```
{
"children": ["Avery","Callie","lucy","Molly"],
"hobbies": ["swimming","biking","drawing","horseplaying"]
}
```

Here’s the same object with an array in YAML:

```
children:
  - Avery
  - Callie
  - lucy
  - Molly
hobbies:
  - swimming
  - biking
  - drawing
  - horseplaying
```

Here’s an array containing objects in JSON:

```
[
   {
      "name":"Tom",
      "age":43
   },
   {
      "name":"Shannon",
      "age":41
   }
]
```

Here’s the same array containing objects converted to YAML:

```
-
  name: Tom
  age: 42
-
  name: Shannon
  age: 41
```

Hopefully, by seeing the syntax side by side, it will begin to make more sense. Is the YAML syntax more readable? It might be difficult to see in these simple examples, but generally it is.

JavaScript uses the same dot notation techniques to access the values in YAML as it does in JSON. (They're pretty much interchangeable formats.) The benefit to using YAML, however, is that it's more readable than JSON.

However, YAML might be more tricky because it depends on getting the spacing just right. Sometimes that spacing is hard to see (especially with a complex structure), and that's where JSON (while maybe more cumbersome) is perhaps easier to troubleshoot.

Some features of YAML not present in JSON

YAML has some features that JSON lacks. You can add comments in YAML files using the `#` sign. YAML also allows you to use something called "anchors." For example, suppose you have two definitions that are similar. You could write the definition once and use a pointer to refer to both:

```
api: &apidef Application programming interface
application_programming_interface: *apidef
```

If you access the value, the same definition will be used for both. The `*apidef` acts as an anchor or pointer to the definition established at `&apidef`.

You won't use these unique YAML features in the OpenAPI tutorial, but they're worth noting because JSON and YAML aren't entirely equivalent. For details on other differences between JSON and YAML, see Learn YAML in Minutes ↗. To learn more about YAML, see this YAML tutorial ↗.

YAML is also used with Jekyll (p. 637). See my YAML tutorial in the context of Jekyll ↗ for more details.

JSON versus YAML for the spec format

Let's clear up some additional descriptors around JSON and YAML as well. The specification document in my OpenAPI tutorial uses YAML (which I introduced briefly here (p. 276)), but it could also be expressed in JSON. JSON is a subset of YAML, so the two are practically interchangeable formats (for the data structures we're using). Ultimately, though, the OpenAPI spec is a JSON object. The specification notes:

> An OpenAPI document that conforms to the OpenAPI Specification is itself a JSON object, which may be represented either in JSON or YAML format. (See Format ↗)

In other words, the OpenAPI document you create is a JSON object, but you have the option of expressing the JSON using either JSON or YAML syntax. YAML is more readable and is a more common format (see API Handyman's take on JSON vs YAML ↗ for more discussion), so I've used YAML exclusively in code samples here. You will see that the OpenAPI specification documentation on GitHub always shows both the JSON and YAML syntax when showing specification formats. (For a more detailed comparison of YAML versus JSON, see "Relation to JSON" in the YAML spec.)

YAML refers to data structures with three main terms: "mappings (hashes/dictionaries), sequences (arrays/lists) and scalars (strings/numbers)" (see "Introduction" in YAML 1.2). However, because the OpenAPI spec is a JSON object, it uses JSON terminology — such as "objects," "arrays," "properties," "fields," and so forth. As such, I'll be showing YAML-formatted content but describing it using JSON terminology.

Review and summary

So that we're on the same page with terms in the upcoming tutorial, let's briefly review. Each level in YAML (defined by a two-space indent) is an object. In the following code, `california` is an object. `animal`, `flower`, and `bird` are properties of the `california` object.

```
california:
  animal: Grizzly Bear
  flower: Poppy
  bird: Quail
```

Here's what this looks like in JSON:

```
{
  "california": {
    "animal": "Grizzly Bear",
    "flower": "Poppy",
    "bird": "Quail"
  }
}
```

The specification often uses the term "field" in the titles and table column names when listing the properties for a specific object. (Further, it identifies two types of fields — "fixed" fields are declared, unique names while "patterned" fields are regex expressions.) *Fields* and *properties* are used synonymously in the OpenAPI spec.

In the following code, `countries` contains an object called `united_states`, which contains an object called `california`, which contains several properties with string values:

```
countries:
  united_states:
    california:
      animal: Grizzly Bear
      flower: Poppy
      bird: Quail
```

In the following code, `demographics` is an object that contains an array:

```
demographics:
 - population
 - land
 - rivers
```

Here's what the above code looks like in JSON:

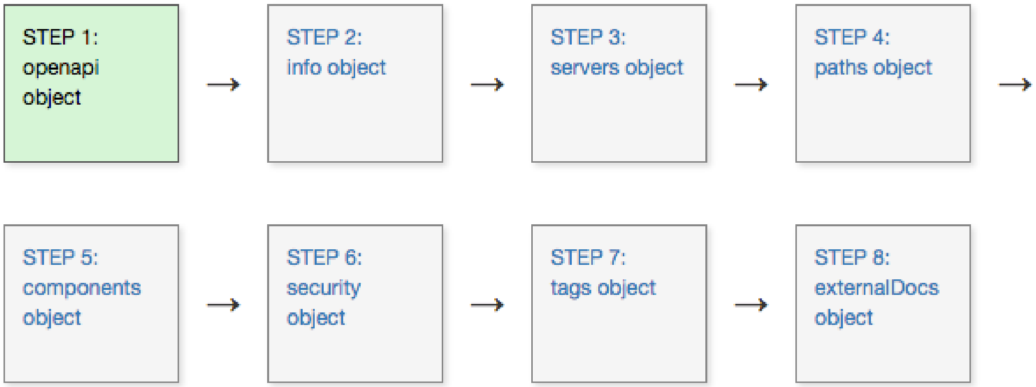
```
{
  "demographics": [
    "population",
    "land",
    "rivers"
  ]
}
```

Hopefully, those brief examples will help align us with the terminology used in the tutorial.

Let's get started

With that information about YAML, hopefully the upcoming step-by-step sections that walk through each section in the OpenAPI spec, using YAML as the primary format, will make more sense. Let's get started with Step 1: The openapi object (OpenAPI tutorial) (p. 282).

Step 1: The openapi object (OpenAPI tutorial)



Before diving into the first step of the OpenAPI tutorial here, read the OpenAPI tutorial overview (p. 272) (if you haven't already) to get a sense of the scope of this tutorial. In brief, this OpenAPI tutorial is unique in the following ways:

- This OpenAPI tutorial shows the spec in context of a simple weather API introduced earlier (p. 65) in this course.
- This OpenAPI tutorial shows how the spec information gets populated in two different tools and display frameworks: Stoplight and Swagger UI.
- This OpenAPI tutorial is a subset of the information in both the OpenAPI specification and the OpenAPI specification commentary. In other words, it's not comprehensive of all the possible information in the spec.
- This OpenAPI tutorial covers the *3.0 version* of the OpenAPI spec, which is the latest version. (Surprisingly, many tools still support only the 2.0 version.)

The root-level objects in OpenAPI spec

The OpenAPI 3.0 spec has eight objects at the root level. These eight objects have many nested objects inside them, but at the root level, there are just these parent objects:

- `openapi`
- `info`
- `servers`
- `paths`
- `components`
- `security`
- `tags`
- `externalDocs`

By “root level,” I mean the first level in the OpenAPI document. This level is also referred to as the global level because some object properties declared here (namely `servers` and `security`) are applied to each of the operation objects unless overridden at a lower level.

The whole document (the object that contains these eight root-level objects) is called an OpenAPI document. The convention is to name the document **openapi.yml**.

“OpenAPI” refers to the specification; “Swagger” refers to the tooling (at least from SmartBear) that supports the OpenAPI specification.

Although one could approach the assembly of an OpenAPI document in many ways, I’ve decided to structure each step in this tutorial based on each of these root-level objects. You’ll see two approaches: one using Swagger working at a code-level, and another using Stoplight Studio working in a user interface.

Swagger

Before we jump into coding, let’s first talk about where you’ll write the code and what your options are.

Where to write your spec code

The easiest option is to use the online Swagger Editor. The Swagger Editor provides a split view: on the left where you write your spec code, and on the right, you see a fully functional Swagger UI display. You can even submit requests from the Swagger UI display in this editor.

The Swagger Editor will validate your content in real-time, and you will see validation errors until you finish coding the specification document. Don’t worry about the errors unless you see X marks in the code you’re working on.

I usually keep a local text file (using a text editor such as Atom or Sublime Text) where I keep the specification document offline, but I work with the document’s content in the online Swagger Editor. When I’m done working for the day, I copy and save the content back to my local file. Even so, the Swagger Editor caches the content quite well (just don’t clear your browser’s cache), so you probably won’t need your local file as a backup.

If you want to purchase a subscription to SwaggerHub (p. 230), you could keep your spec content in the cloud (SwaggerHub has an editor almost identical to Swagger UI) associated with your personal login. SwaggerHub is the premium tooling for the open-source and free Swagger Editor.

Another option to work locally is to use Visual Studio Code with two extensions: openapi-lint and Swagger Viewer. These extensions let you work locally and preview a live version of Swagger. You can also download and run the Swagger Editor locally.

For the sake of simplicity, for the Swagger sections in this tutorial, we’ll just use the online Swagger Editor.

Add the openapi object

From the Swagger Editor and go to **File > Clear editor**. Keep this tab open throughout the OpenAPI tutorial, as you’ll be adding to your specification document with each step.

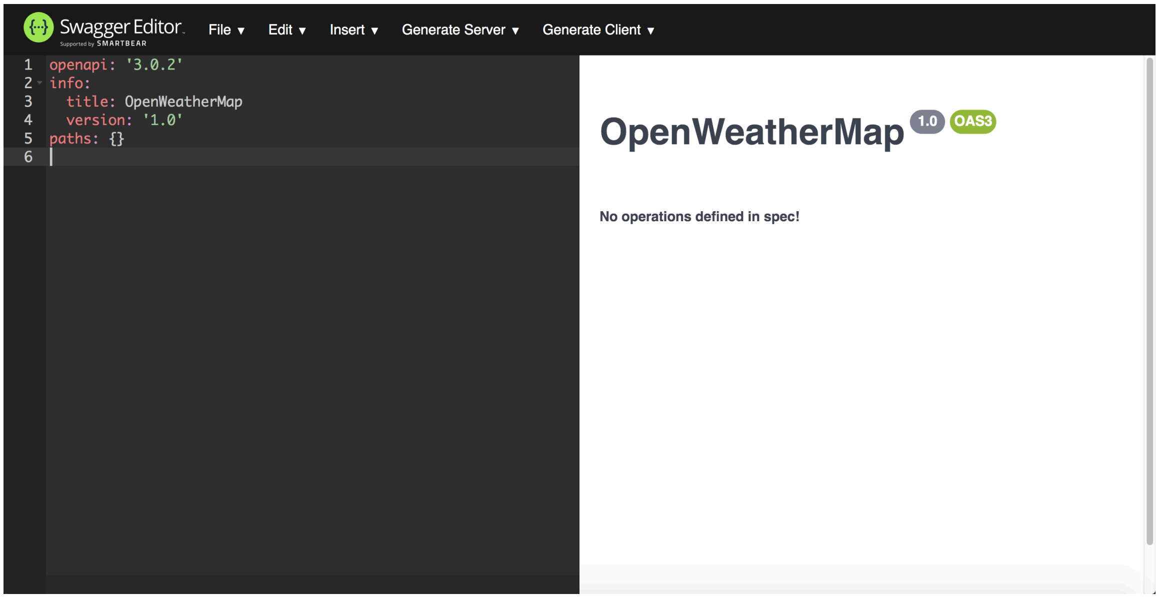
Add the first root-level property for the specification document: `openapi`. In the openapi object, indicate the version of the OpenAPI spec to validate against. The latest version is `3.0.2`.

```
openapi: "3.0.2"
```

Until you add more information in here, you'll see error messages and notes such as "No operations defined in spec!" To avoid these errors, add some placeholder info here like this:

```yaml
openapi: '3.0.2'
info:
  title: OpenWeatherMap
  version: '1.0'
paths: {}
```

The editor renders the display as follows.



```yaml
openapi: '3.0.2'
info:
  title: OpenWeatherMap
  version: '1.0'
paths: {}

```

OpenWeatherMap 1.0 OAS3

No operations defined in spec!

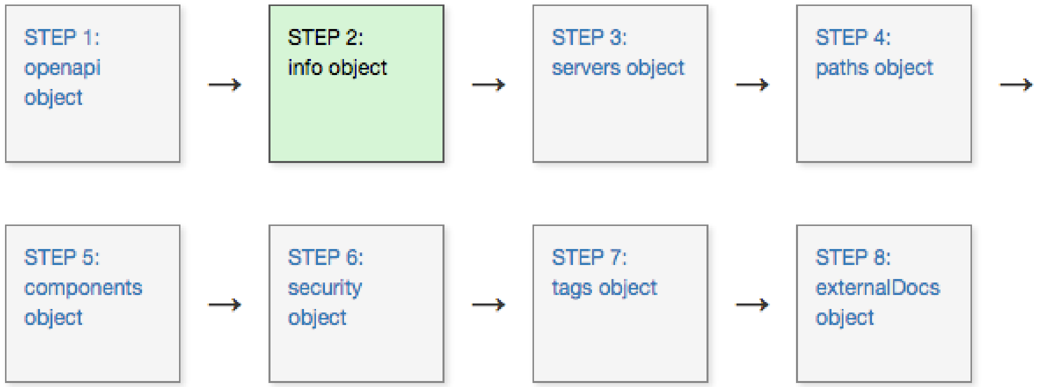
openapi object

If you get stuck, see the sample OpenAPI spec here for the fully working sample. This will help you spot and troubleshoot indentation or other errors.

On the backend, Swagger UI uses the 3.0.2 version of the spec to validate your content. In the above screenshot, the gray "1.0" version refers to the version of the API here, not the version of the OpenAPI spec.

There's not much to the `openapi` object except to elaborate on the versions. OAS 3.0 was released on 2017-07-26, and OAS 3.0.2 was released on 10-08-2018 (see Version History). Much of the information and examples online, as well as supporting tools, often focus only on 2.0. Even if you're locked into publishing in a 2.0 tool or platform, you can code the spec in 3.0 and then use a tool such as APIMATIC Transformer to convert the 3.0 spec to 2.0. You can also convert a spec from 2.0 to 3.0.

Step 2: The info object (OpenAPI tutorial)



The info object contains basic information about your API, including the title, a description, version, link to the license, link to the terms of service, and contact information. Many of the properties are optional.

Sample info object

Here's an example of the `info` object and its properties. (The `openapi` object and the empty `paths` object are commented out to maintain the focus on the `info` object.)

```yaml
# openapi: 3.0.2
info:
  title: "OpenWeatherMap API"
  description: "Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API. <br/><br/> **Note**: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results."
  version: "2.5"
  termsOfService: "https://openweathermap.org/terms"
  contact:
    name: "OpenWeatherMap API"
    url: "https://openweathermap.org/api"
    email: "some_email@gmail.com"
  license:
    name: "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)"
    url: "https://openweathermap.org/price"
# paths: {}
```

If you get stuck, see the sample OpenAPI spec here ↗ for the fully working sample. This will help you spot and troubleshoot indentation or other errors.

Description properties and Markdown

Note that in any `description` property, you can use CommonMark Markdown ↗, which is much more precise, unambiguous, and robust than the original Markdown.

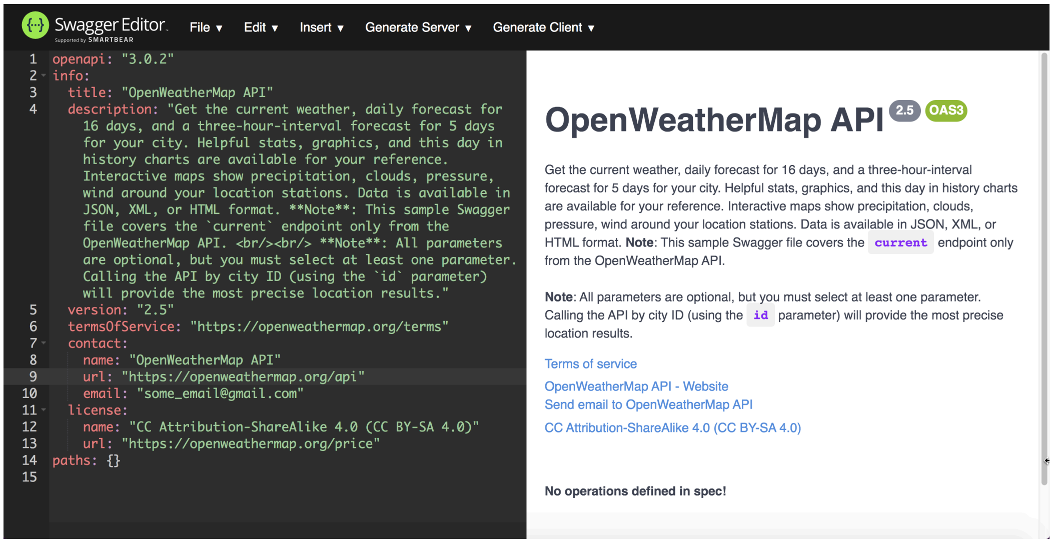
For example, CommonMark markdown offers some backslash escapes ↗, and it specifies exactly how many spaces you need in lists and other punctuation. You can also break to new lines with `\n` and escape problematic characters like quotation marks or colons with a backslash.

As you write content in `description` properties, note that colons are problematic in YAML because they signify new levels. Either enclose the `description` value in quotation marks or escape colons with a backslash. (If you enclose the values in quotation marks, syntax highlighters in text editors can display better color coding between the properties and values.)

Update your file in Swagger Editor

To update the spec file in Swagger Editor:

1. Paste the code from the preceding section (“Sample info object”) containing the `info` object into the Swagger Editor.
2. Uncomment the `openapi` and `paths` objects (remove the “ `#` ”). The display looks as follows:



```yaml
openapi: "3.0.2"
info:
  title: "OpenWeatherMap API"
  description: "Get the current weather, daily forecast for
    16 days, and a three-hour-interval forecast for 5 days
    for your city. Helpful stats, graphics, and this day in
    history charts are available for your reference.
    Interactive maps show precipitation, clouds, pressure,
    wind around your location stations. Data is available in
    JSON, XML, or HTML format. **Note**: This sample Swagger
    file covers the `current` endpoint only from the
    OpenWeatherMap API. <br/><br/> **Note**: All parameters
    are optional, but you must select at least one parameter.
    Calling the API by city ID (using the `id` parameter)
    will provide the most precise location results."
  version: "2.5"
  termsOfService: "https://openweathermap.org/terms"
  contact:
    name: "OpenWeatherMap API"
    url: "https://openweathermap.org/api"
    email: "some_email@gmail.com"
  license:
    name: "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)"
    url: "https://openweathermap.org/price"
paths: {}
```

OpenWeatherMap API

Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

Terms of service

OpenWeatherMap API - Website
Send email to OpenWeatherMap API
CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)

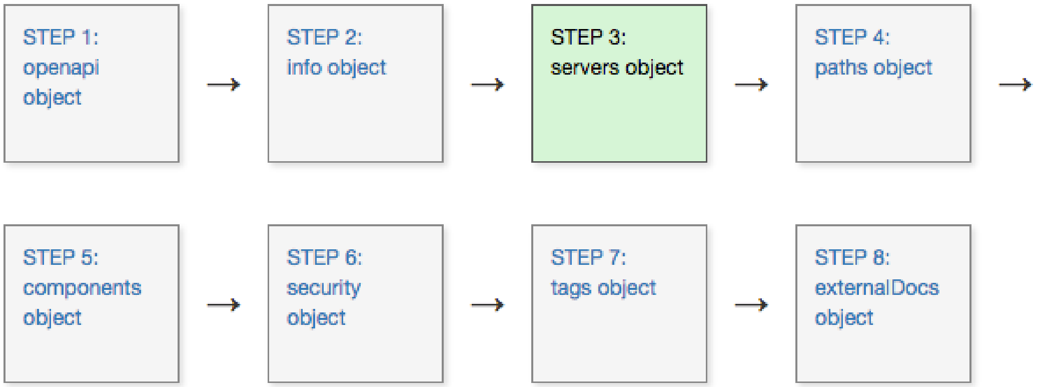
No operations defined in spec!

openapi, info, and empty paths object in Swagger Editor

In the Swagger UI display, the `info` object's information appears below the title.

In the `description` property, in addition to describing your overall API, you might want to provide some basic instructions to users on how to use Swagger UI. If there's a test account they should use, you can provide the information they need in this space.

Step 3: The servers object (OpenAPI tutorial)



In the `servers` object, you specify the basepath used in your API requests. The basepath is the part of the URL that appears before the endpoint.

Sample servers object

The following is a sample `servers` object:

```
servers:
- url: https://api.openweathermap.org/data/2.5/
```

Each of your endpoints (called "paths" in the spec) will be appended to the server URL when users make "Try it out" requests. For example, if one of the paths is `/weather`, when Swagger UI submits the request, it will submit the path to `{server URL}{path}` or `https://api.openweathermap.org/data/2.5/weather`.

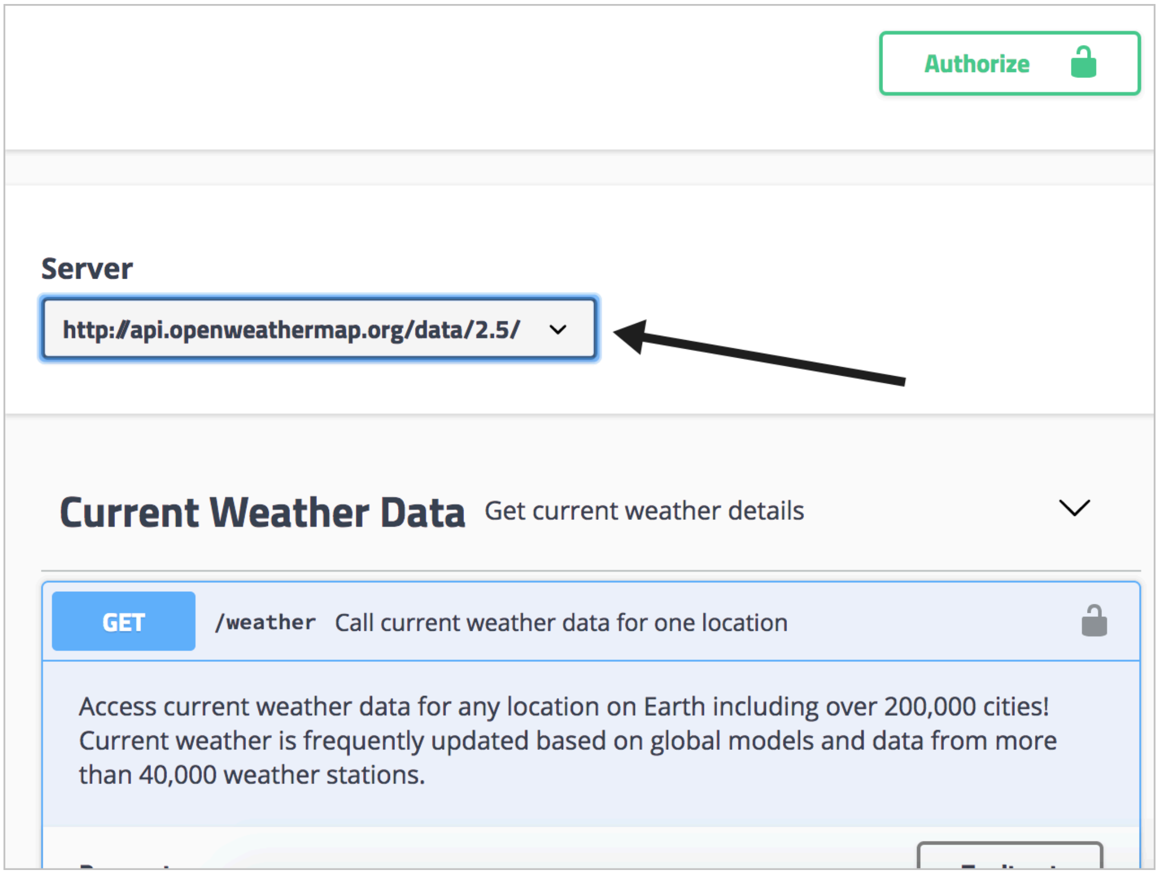
Options with the server URL

You have some flexibility and configuration options for your server URL. You can specify multiple server URLs that might relate to different environments (test, beta, production). If you have multiple server URLs, users can select the environment from a servers drop-down box. For example, you can specify multiple server URLs like this:

```
servers:
- url: https://api.openweathermap.org/data/2.5/
  description: Production server
- url: http://beta.api.openweathermap.org/data/2.5/
  description: Beta server
- url: http://some-other.api.openweathermap.org/data/2.5/
  description: Some other server
```

If you get stuck, see the sample OpenAPI spec here for the fully working sample. This will help you spot and troubleshoot indentation or other errors.

In Swagger UI, the multiple servers appear as options users can select in a drop-down list:


Authorize
Server
http://api.openweathermap.org/data/2.5/
Current Weather Data
Get current weather details
GET
/weather
Call current weather data for one location

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

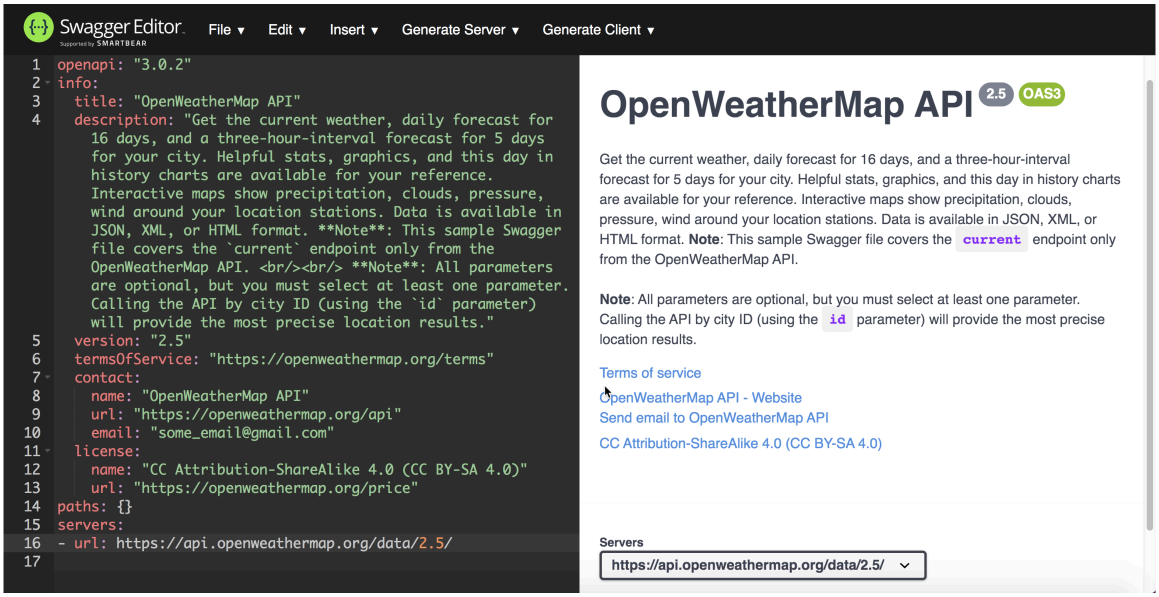
If you have just one URL, you still see a drop-down box but with just one option.

You can also incorporate variables into the server URL that can be populated at runtime by your server. Additionally, if different paths (endpoints) require different server URLs, you can add the `servers` object as a property in the `path` (p. 291) object's operation object. The locally declared servers URL will override the global servers URL.

See "Overriding Servers" in "API Server and Base URL" (Swagger's docs) for more details.

Swagger

Paste the `servers` object (the first code sample above (p. 288) showing just one `url`) into your Swagger Editor, adding to the code you already have there. Swagger UI will look as follows.



```
openapi: "3.0.2"
info:
  title: "OpenWeatherMap API"
  description: "Get the current weather, daily forecast for
    16 days, and a three-hour-interval forecast for 5 days
    for your city. Helpful stats, graphics, and this day in
    history charts are available for your reference.
    Interactive maps show precipitation, clouds, pressure,
    wind around your location stations. Data is available in
    JSON, XML, or HTML format. **Note**: This sample Swagger
    file covers the `current` endpoint only from the
    OpenWeatherMap API. <br/><br/> **Note**: All parameters
    are optional, but you must select at least one parameter.
    Calling the API by city ID (using the `id` parameter)
    will provide the most precise location results."
  version: "2.5"
  termsOfService: "https://openweathermap.org/terms"
  contact:
    name: "OpenWeatherMap API"
    url: "https://openweathermap.org/api"
    email: "some_email@gmail.com"
  license:
    name: "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)"
    url: "https://openweathermap.org/price"
paths: {}
servers:
- url: https://api.openweathermap.org/data/2.5/

```

OpenWeatherMap API 2.5 OAS3

Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

Terms of service

OpenWeatherMap API - Website
Send email to OpenWeatherMap API

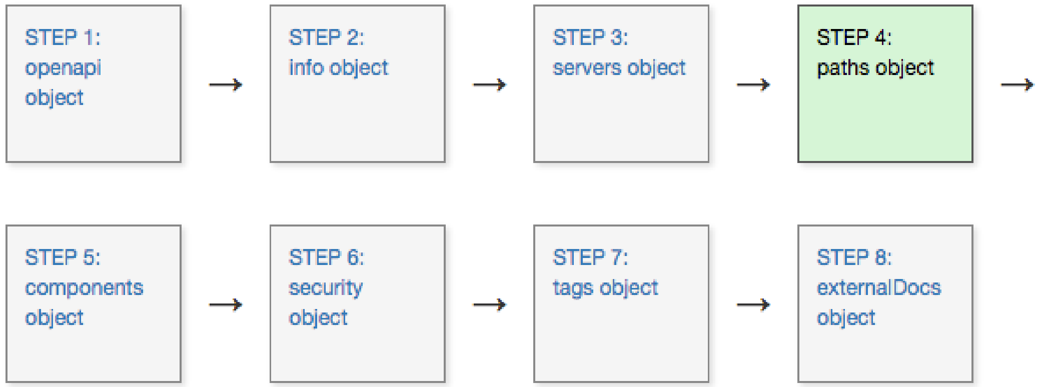
CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)

Servers
https://api.openweathermap.org/data/2.5/ ⌄

Swagger UI with the servers object

Notice the drop-down menu that appears in the lower-right. (Even if you have just one URL, it still appears in a drop-down menu.)

Step 4: The paths object (OpenAPI tutorial)



The `paths` object contains the meat of your API information. The `paths` object has several sub-objects: a path items object, an operations object, and more.

We've been moving along at about 5 mph in the previous steps but are going to speed up to 60 mph here quickly. It's okay if the content that follows doesn't entirely sink in. You can paste the example code that follows into Swagger UI for now and later go back to study it in more detail.

Paths objects

My preferred term is "endpoint" rather than "path," but to be consistent with the terminology of the OpenAPI spec, I use the term "paths" here.

Each item in the `path` object contains an operation object. (Operations are the GET, POST, PUT, and DELETE methods we explored in the Endpoints section (p. 126) of the API reference tutorial.)

Start by listing the paths (endpoints) and their allowed operations (methods). For the `weather` endpoint in the OpenWeatherMap API, there is just one path (`/weather`) and one operation (`get`) for that path:

```
paths:
  /weather:
    get:
```

Operation Objects

The operation object (`get` in the code above) contains various properties and objects:

- `tags`: A group name to organize paths in the Swagger UI. Swagger UI will group endpoints under tag headings.
- `summary`: A brief overview of the path. Swagger UI shows the summary next to the path name. Limit the summary to 5-10 words only. The display appears even when this section is collapsed.

- `description`: A full description of the path. Include as much detail as you want. There's a lot of space in the Swagger UI for these details. CommonMark Markdown is allowed.
- `externalDocs` (object): Links to documentation for more information about the path.
- `operationId`: A unique identifier for the path.
- `parameters` (object): Parameters accepted by the path. Does not include request bodies, which are instead detailed in the `requestBody` object (p. 291). The `parameters` object can also include a reference object that contains a pointer to the description in the `components` object (this is explained in Step 5: The components object (p. 302)).
- `requestBody` (object): The request body details for this path. The `requestBody` object can also include a reference object that contains a pointer to the description in the `components` object (explained in step 5 (p. 302)). (You can find an example of a `requestBody` parameter in the Swagger Petstore demo. The `/pet` endpoint submits a request body when adding a pet. Check out the `requestBody` YAML syntax in petstore-expanded.yml — look at `post` under `/pets`. Also see Describing Request Body.)
- `responses` (object): Responses provided from requests with this path. The `responses` object can also include a reference object that contains a pointer to the description in the `components` object. Responses use standard status codes.
- `callbacks` (object): Callback details to be initiated by the server if desired. Callbacks are operations performed after a function finishes executing. The `callbacks` object can also include a reference object that contains a pointer to the description in the `components` object.
- `deprecated`: Whether the path is deprecated. Omit unless you want to indicate a deprecated field. Boolean.
- `security` (object): Security authorization method used with the operation. Include this object at the path level only if you want to overwrite the `security` object at the root level. The name is defined by the `securitySchemes` object in the `components` object. More details about this are provided in the security object (p. 327).
- `servers` (object): A servers object that might differ from the global `servers` object (p. 288) for this path.

Each of the above hyperlinked properties that say "(object)" contain additional levels. Their values aren't just simple data types like strings but are rather objects that contain their own properties.

You'll undoubtedly need to consult the OpenAPI spec to see what details are required for each of the values and objects here. I can't replicate all the details you need, nor would I want to. I'm just trying to introduce you to the OpenAPI properties at a surface level.

Let's add a skeleton of the operation object details to our existing code:

```
paths:
  /weather:
    get:
      tags:
      summary:
      description:
      operationId:
      externalDocs:
      parameters:
      responses:
      deprecated:
      security:
      servers:
      requestBody:
      callbacks:
```

At this point, if you paste this content into the Swagger Editor, you will get errors until some additional properties are added.

Now we can remove a few unnecessary fields that we don't need for our OpenWeatherMap API documentation:

- There's no need to include `requestBody` object because none of the OpenWeatherMap API paths contain request bodies.
- There's no need to include the `servers` object because the paths use the same global `servers` URL that we defined globally (p. 288) at the root level.
- There's no need to include security because all the paths use the same `security` object, which we will define globally at the root level later (see Step 6: The security object (p. 327)).
- There's no need to include `deprecated` because none of the paths are deprecated.
- There's no need to include `callbacks` because none of the paths use callbacks.

As a result, we can reduce the number of relevant fields to the following:

```
paths:
  /weather:
    get:
      tags:
      summary:
      description:
      operationId:
      externalDocs:
      parameters:
      responses:
```

Most of the properties for the operation object either require simple strings or include relatively simple objects. The most detailed object here is the `parameters` object (p. 293) and the `responses` object (p. 296).

Parameters object

The `parameters` object contains an array with these properties:

- `name` : Parameter name.

- `in`: Where the parameter appears. Possible values are `header`, `path`, `query`, or `cookie`. (Request bodies are not described here.)
- `description`: Description of the parameter.
- `required`: Whether the parameter is required.
- `deprecated`: Whether the parameter is deprecated.
- `allowEmptyValue`: Whether the parameter allows an empty value to be submitted.
- `style`: How the parameter's data is serialized (converted to bytes during data transfer).
- `explode`: Advanced parameter related to arrays.
- `allowReserved`: Whether reserved characters are allowed.
- `schema` (object): The schema or model for the parameter. The schema defines the input or output data structure. Note that the `schema` can also contain an `example` object.
- `example`: An example of the media type. If your `examples` object contains examples, those examples appear in Swagger UI rather than the content in the `example` object.
- `examples` (object): An example of the media type, including the schema.

Here's the `paths` object that includes the `parameters` details:

```
paths:
  /weather:
    get:
      tags:
      - Current Weather Data
      summary: "Call current weather data for one location."
      description: "Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations."
      operationId: CurrentWeatherData
      parameters:
      - name: q
        in: query
        description: "**City name**. *Example: London*. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by a comma; use ISO 3166 country codes."
        schema:
          type: string

      - name: id
        in: query
        description: "**City ID**. *Example: `2172797`*. You can call by city ID. The API responds with the exact result. The List of city IDs can be downloaded [here](http://bulk.openweathermap.org/sample/). You can include multiple cities in this parameter &mdash; just separate them by commas. The limit of locations is 20. *Note: A single ID counts as a one API call. So, if you have 3 city IDs, it’s treated as 3 API calls.*"
        schema:
          type: string

      - name: lat
        in: query
        description: "**Latitude**. *Example: 35*. The latitude coordinate of the location of your interest. Must use with `lon`."
        schema:
          type: string

      - name: lon
        in: query
        description: "**Longitude**. *Example: 139*. Longitude coordinate of the location of your interest. Must use with `lat`."
        schema:
          type: string

      - name: zip
        in: query
        description: "**Zip code**. Search by zip code. *Example: 95050,us*. Please note that if the country is not specified, the search uses USA as a default."
        schema:
          type: string

      - name: units
        in: query
        description: '**Units**. *Example: imperial*. Possible values: `standard`, `metric`, and `imperial`. When you do not use the `units` parameter, the format is `standard` by default.'
        schema:
          type: string
          enum: [standard, metric, imperial]
          default: "imperial"

      - name: lang
        in: query
        description: '**Language**. *Example: en*. You can use lang parameter to get the output in your language. We support the following languages that you can use with the corresponded lang values: Arabic - `ar`, Bulgarian - `bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English - `en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `gl`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Korean - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `nl`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedish - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`, Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese Traditional - `zh_tw`.'
        schema:
          type: string
          enum: [ar, bg, ca, cz, de, el, en, fa, fi, fr, gl, hr, hu, it, ja, kr, la, lt, mk, nl, pl, pt, ro, ru, se, sk, sl, es, tr, ua, vi, zh_cn, zh_tw]
          default: "en"

      - name: mode
        in: query
        description: "**Mode**. *Example: html*. Determines the format of the response. Possible values are `xml` and `html`. If the mode parameter is empty, the format is `json` by default."
        schema:
          type: string
          enum: [json, xml, html]
          default: "json"
```

If you get stuck, see the sample OpenAPI spec here for the fully working sample. This will help you spot and troubleshoot indentation or other errors.

Responses object

The other substantial property in the operations object is the `responses` object. For the `responses` property, you typically reference a full definition in the `components` object, so I'll cover the `responses` object in the next section — Step 5: The components object (p. 302). (There's already too much detail in this step as is.)

For now, so that the Swagger Editor will validate and show our path, let's just add some placeholder content for `responses`:

```
responses:
  200:
    description: Successful response
    content:
      application/json:
        schema:
          title: Sample
          type: object
          properties:
            placeholder:
              type: string
              description: Placeholder description

  404:
    description: Not found response
    content:
      text/plain:
        schema:
          title: Weather not found
          type: string
          example: Not found
```

See Describing Parameters in Swagger's OpenAPI documentation for more details.

Paths object code

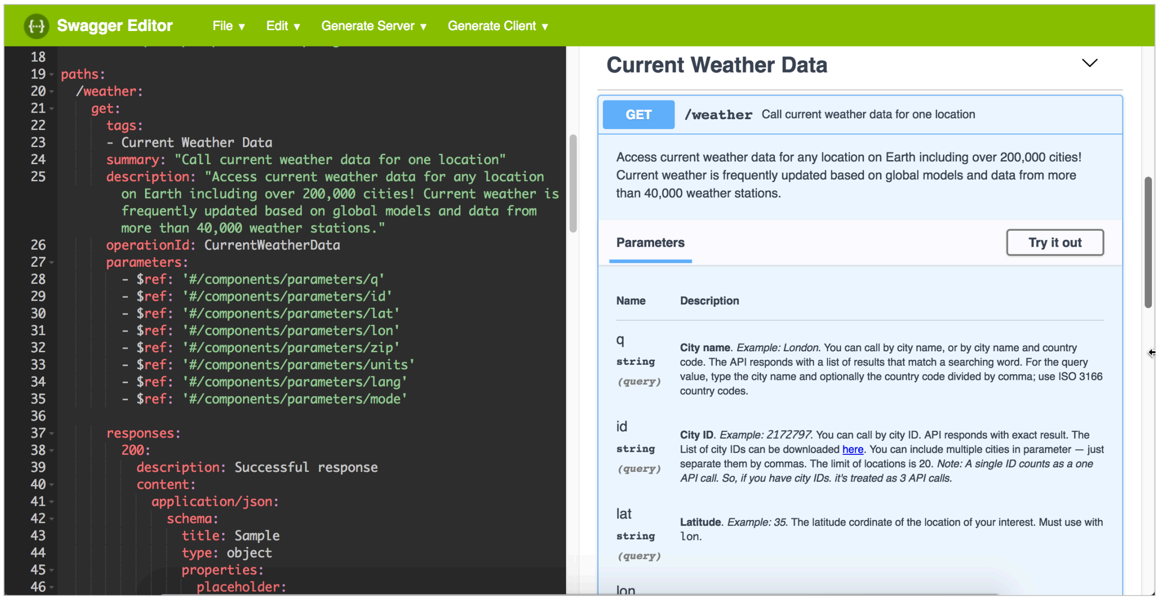
Now let's combine the above two code blocks (both `parameters` and `responses`) for our `paths` object. You can paste the following code into the Swagger Editor — add this `paths` object below the `openapi`, `info`, and `servers` code you added in the previous tutorials.

```
paths:
  /weather:
    get:
      tags:
      - Current Weather Data
      summary: "Call current weather data for one location."
      description: "Access current weather data for any location on Earth in
cluding over 200,000 cities! Current weather is frequently updated based on
global models and data from more than 40,000 weather stations."
      operationId: CurrentWeatherData
      parameters:
      - name: q
        in: query
        description: "**City name**. *Example: London*. You can call by cit
y name, or by city name and country code. The API responds with a list of re
sults that match a searching word. For the query value, type the city name a
nd optionally the country code divided by a comma; use ISO 3166 country code
s."
        schema:
          type: string

      - name: id
        in: query
        description: "**City ID**. *Example: `2172797`*. You can call by cit
y ID. The API responds with the exact result. The List of city IDs can be do
wnloaded [here](http://bulk.openweathermap.org/sample/). You can include mul
tiple cities in this parameter &mdash; just separate them by commas. The lim
it of locations is 20. *Note: A single ID counts as a one API call. So, if y
ou have 3 city IDs, it’s treated as 3 API calls.*"
        schema:
          type: string

      - name: lat
        in: query
        description: "**Latitude**. *Example: 35*. The latitude coordinate o
f the location of your interest. Must use with `lon`."
        schema:
          type: string

      - name: lon
        in: query
        description: "**Longitude**. *Example: 139*. Longitude coordinate o
f the location of your interest. Must use with `lat`."
        schema:
          type: string

      - name: zip
        in: query
        description: "**Zip code**. Search by zip code. *Example: 95050,u
s*. Please note that if the country is not specified, the search uses USA a
s a default."
```

```
        schema:
          type: string

      - name: units
        in: query
        description: '**Units**. *Example: imperial*. Possible values: `stan
dard`, `metric`, and `imperial`. When you do not use the `units` parameter,
the format is `standard` by default.'
        schema:
          type: string
          enum: [standard, metric, imperial]
          default: "imperial"

      - name: lang
        in: query
        description: '**Language**. *Example: en*. You can use lang paramete
r to get the output in your language. We support the following languages tha
t you can use with the corresponded lang values: Arabic - `ar`, Bulgarian -
`bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English -
`en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `g
l`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Kore
an - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `n
l`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedi
sh - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`,
Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese T
raditional - `zh_tw`.'
        schema:
          type: string
          enum: [ar, bg, ca, cz, de, el, en, fa, fi, fr, gl, hr, hu, it, j
a, kr, la, lt, mk, nl, pl, pt, ro, ru, se, sk, sl, es, tr, ua, vi, zh_cn, z
h_tw]
          default: "en"

      - name: mode
        in: query
        description: "**Mode**. *Example: html*. Determines the format of th
e response. Possible values are `xml` and `html`. If the mode parameter is e
mpty, the format is `json` by default."
        schema:
          type: string
          enum: [json, xml, html]
          default: "json"

      responses:
        200:
          description: Successful response
          content:
            application/json:
              schema:
                title: Sample
                type: object
                properties:
                  placeholder:
                    type: string
                    description: Placeholder description

        404:
          description: Not found response
          content:
            text/plain:
              schema:
                title: Weather not found
                type: string
                example: Not found
```

View the Appearance in Swagger UI

Swagger UI displays the `paths` object like this:



```
paths:
  /weather:
    get:
      tags:
      - Current Weather Data
      summary: "Call current weather data for one location"
      description: "Access current weather data for any location
        on Earth including over 200,000 cities! Current weather is
        frequently updated based on global models and data from
        more than 40,000 weather stations."
      operationId: CurrentWeatherData
      parameters:
        - $ref: '#/components/parameters/q'
        - $ref: '#/components/parameters/id'
        - $ref: '#/components/parameters/lat'
        - $ref: '#/components/parameters/lon'
        - $ref: '#/components/parameters/zip'
        - $ref: '#/components/parameters/units'
        - $ref: '#/components/parameters/lang'
        - $ref: '#/components/parameters/mode'

      responses:
        200:
          description: Successful response
          content:
            application/json:
              schema:
                title: Sample
                type: object
                properties:
                  placeholder:
```

Current Weather Data

GET `/weather` Call current weather data for one location

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

Name	Description
q `string` *(query)*	**City name**. *Example: London.* You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by comma; use ISO 3166 country codes.
id `string` *(query)*	**City ID**. *Example: 2172797.* You can call by city ID. API responds with exact result. The List of city IDs can be downloaded here. You can include multiple cities in parameter — just separate them by commas. The limit of locations is 20. *Note: A single ID counts as a one API call. So, if you have city IDs. it's treated as 3 API calls.*
lat `string` *(query)*	**Latitude**. *Example: 35.* The latitude cordinate of the location of your interest. Must use with `lon`.
lon	

Expand the Current Weather Data section to see the details. When you click **Try it out**, you'll notice that the field populates with the description. If you want the field to populate with a value, add a `default` property under `schema` (as shown with the `mode` parameter in the code above).

However, with this API, the parameters can't all be passed with the same request — you use only the parameters you want for the request you're making. (For example, you can't pass zip code *and* city name *and* lat/long, etc. in the same request.) As a result, it wouldn't make sense to use defaults for each parameter because the user would then need to remove most of them.

Swagger's UI collapses each path by default. You can set whether the initial display is collapsed or open using the `docExpansion` parameter in Swagger UI ↗. This `docExpansion` parameter is for Swagger UI and not part of the OpenAPI spec. Swagger UI has more than 20 different parameters ↗ of its own that control the display. For example, if you don't want the `Models` section to appear, add the parameter `defaultModelsExpandDepth: -1` in your Swagger UI file.

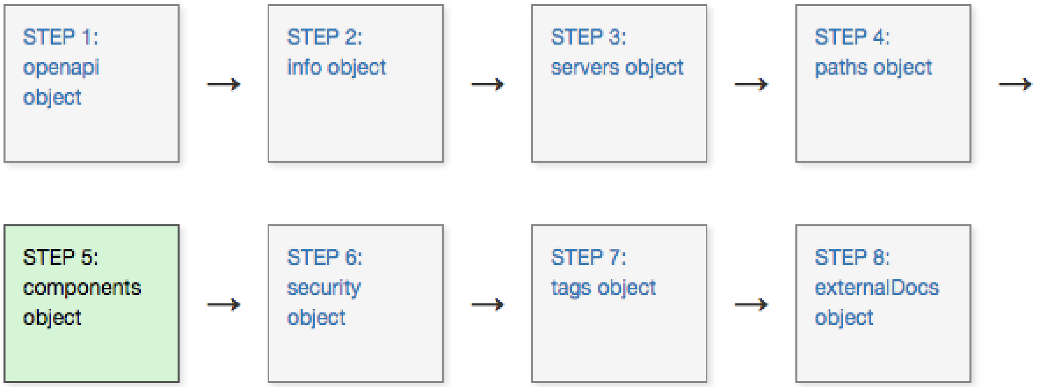
Note about parameter dependencies

The OpenAPI specification doesn't allow you to declare dependencies with parameters, or mutually exclusive parameters. According to the Swagger OpenAPI documentation,

> OpenAPI 3.0 does not support parameter dependencies and mutually exclusive parameters. There is an open feature request at https://github.com/OAI/OpenAPI-Specification/issues/256. What you can do is document the restrictions in the parameter description and define the logic in the 400 Bad Request response. (Parameter Dependencies)

In the case of the weather endpoint with the OpenWeatherMap, most of the parameters are mutually exclusive. You can't search by City ID *and* zip code simultaneously. Although the parameters are optional, you must use at least one parameter. Also, if you use the `lat` parameter, you must also use the `lon` parameter because they're a pair. The OpenAPI spec can't programmatically reflect that structured logic, so you have to explain it in the `description` property or in other more conceptual documentation.

Step 5: The components object (OpenAPI tutorial)



The `components` object is unique from the other objects in the OpenAPI specification. In `components`, you store re-usable definitions that might appear in multiple places in your specification document. In our API documentation scenario, we'll store details for both the `parameters` and `responses` object in `components`.

Reasons to use the components object

Describing the details of your parameters and describing the schema of complex responses can be the most challenging aspects of the OpenAPI spec. Although you can define the parameters and responses directly in the `parameters` and `responses` objects, you typically don't list them there for two reasons:

- You might want to re-use parts of these definitions in other requests or responses. It's common to have the same parameter or response used in multiple places in an API. Through the `components` object, OpenAPI allows you to re-use these same definitions in multiple places.
- You might not want to clutter up your `paths` object with too many parameter and response details, since the `paths` object is already somewhat complex with several levels of objects.

Instead of listing the schema for your requests and responses in the `paths` object, for more complex schemas (or for schemas that are re-used in multiple operations or paths), you typically use a reference object (referenced with `$ref`) that points to a specific definition in the `components` object. (For more details on `$ref`, see Using $ref.)

Think of the `components` object like a document appendix where the re-usable details are provided. If multiple parts of your spec have the same schema, you point each of these references to the same object in your `components` object, and in so doing you single source the content. The `components` object can even be stored in a separate file if you have a large API and want to organize the information that way. (However, with multiple files, you wouldn't be able to use the online Swagger Editor to validate the content.)

Objects in components

You can store a lot of different re-usable objects in the `components` object. The `components` object can contain these objects:

- `schemas`
- `responses`
- `parameters`
- `examples`
- `requestBody`
- `headers`
- `securitySchemes`
- `links`
- `callbacks`

The properties for each object inside `components` are the same as they are when used in other parts of the OpenAPI spec. You use a reference pointer (`$ref`) to point to more details in the `components` object. `$ref` stands for `reference` object and is part of JSON.

Re-using parameters across multiple paths

For the parameters in the previous step, we listed all the details directly in the `parameters` object. To facilitate re-use of the same parameters in other paths, let's store the `parameters` content in `components`. The code below shows how to make these references:

```yaml
paths:
  /weather:
    get:
      tags:
      - Current Weather Data
      summary: "Call current weather data for one location"
      description: "Access current weather data for any location on Earth in
cluding over 200,000 cities! Current weather is frequently updated based on
global models and data from more than 40,000 weather stations."
      operationId: CurrentWeatherData
      parameters:
        - $ref: '#/components/parameters/q'
        - $ref: '#/components/parameters/id'
        - $ref: '#/components/parameters/lat'
        - $ref: '#/components/parameters/lon'
        - $ref: '#/components/parameters/zip'
        - $ref: '#/components/parameters/units'
        - $ref: '#/components/parameters/lang'
        - $ref: '#/components/parameters/mode'

      responses:
        200:
          description: Successful response
          content:
            application/json:
              schema:
                title: Sample
                type: object
                properties:
                  placeholder:
                    type: string
                    description: Placeholder description

        404:
          description: Not found response
          content:
            text/plain:
              schema:
                title: Weather not found
                type: string
                example: Not found

components:

  parameters:
    q:
      name: q
      in: query
      description: "**City name**. *Example: London*. You can call by city n
ame, or by city name and country code. The API responds with a list of resul
ts that match a searching word. For the query value, type the city name and
```

```
optionally the country code divided by a comma; use ISO 3166 country codes."
      schema:
        type: string
    id:
      name: id
      in: query
      description: "**City ID**. *Example: `2172797`*. You can call by city
ID. The API responds with the exact result. The List of city IDs can be down
loaded [here](http://bulk.openweathermap.org/sample/). You can include multi
ple cities in this parameter &mdash; just separate them by commas. The limi
t of locations is 20. *Note: A single ID counts as a one API call. So, if yo
u have 3 city IDs, it’s treated as 3 API calls.*"
      schema:
        type: string

    lat:
      name: lat
      in: query
      description: "**Latitude**. *Example: 35*. The latitude coordinate of
the location of your interest. Must use with `lon`."
      schema:
        type: string

    lon:
      name: lon
      in: query
      description: "**Longitude**. *Example: 139*. Longitude coordinate of t
he location of your interest. Must use with `lat`."
      schema:
        type: string

    zip:
      name: zip
      in: query
      description: "**Zip code**. Search by zip code. *Example: 95050,us*. P
lease note that if the country is not specified, the search uses USA as a de
fault."
      schema:
        type: string

    units:
      name: units
      in: query
      description: '**Units**. *Example: imperial*. Possible values: `standa
rd`, `metric`, and `imperial`. When you do not use the `units` parameter, th
e format is `standard` by default.'
      schema:
        type: string
        enum: [standard, metric, imperial]
        default: "imperial"
    lang:
      name: lang
      in: query
      description: '**Language**. *Example: en*. You can use lang parameter to get the output in your language. We support the following languages that you can use with the corresponded lang values: Arabic - `ar`, Bulgarian - `bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English - `en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `gl`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Korean - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `nl`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedish - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`, Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese Traditional - `zh_tw`.'
      schema:
        type: string
        enum: [ar, bg, ca, cz, de, el, en, fa, fi, fr, gl, hr, hu, it, ja, kr, la, lt, mk, nl, pl, pt, ro, ru, se, sk, sl, es, tr, ua, vi, zh_cn, zh_tw]
        default: "en"

    mode:
      name: mode
      in: query
      description: "**Mode**. *Example: html*. Determines the format of the response. Possible values are `xml` and `html`. If the mode parameter is empty, the format is `json` by default."
      schema:
        type: string
        enum: [json, xml, html]
        default: "json"
```

Replace the existing `paths` object in the Swagger Editor with the above code sample, include the new `components` object, and observe that the rendered display still looks the same.

If you get stuck, see the sample OpenAPI spec here for the fully working sample. This will help you spot and troubleshoot indentation or other errors.

Re-using response objects

In Step 4: The paths object (p. 291), when we described the `responses` object in the `paths` object, even with just a simple placeholder, we used a `schema` object to describe the model for the request or response. The `schema` refers to the data structure (the fields, values, and hierarchy of the various objects and properties of a JSON or YAML object — see What is a schema?).

Let’s dive deeply into how to use the schema properties to document the `responses` object. We will also store this content in `components` so that it can be re-used in other parts of the specification document. If you recall in the previous step (OpenAPI tutorial Step 4: The paths object (p. 291)), the `responses` object for the `weather` endpoint looked like this:

```
paths:
  /current:
    get:
      parameters:

      ...

      responses:
        200:
          description: Successful response
          content:
            application/json:
              schema:
                title: Sample
                type: object
                properties:
                  placeholder:
                    type: string
                    description: Placeholder description

        404:
          description: Not found response
          content:
            text/plain:
              schema:
                title: Weather not found
                type: string
                example: Not found
```

Now let's move the `schema` description for the `200` response into the `components` object:

```yaml
paths:
  /weather:
    get:
      tags:
      - Current Weather Data
      summary: "Call current weather data for one location"
      description: "Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations."
      operationId: CurrentWeatherData
      parameters:
        - $ref: '#/components/parameters/q'
        - $ref: '#/components/parameters/id'
        - $ref: '#/components/parameters/lat'
        - $ref: '#/components/parameters/lon'
        - $ref: '#/components/parameters/zip'
        - $ref: '#/components/parameters/units'
        - $ref: '#/components/parameters/lang'
        - $ref: '#/components/parameters/mode'

      responses:
        200:
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/200'
        404:
          description: Not found response
          content:
            text/plain:
              schema:
                title: Weather not found
                type: string
                example: Not found
```

Then in `components/schemas`, we'll define the `200` schema.

Before we describe the response in the `components` object, it might be helpful to review what the `weather` response from the OpenWeatherMap API looks like. The JSON response contains multiple nested objects at various levels.

```
{
  "coord": {
    "lon": 145.77,
    "lat": -16.92
  },
  "weather": [
    {
      "id": 803,
      "main": "Clouds",
      "description": "broken clouds",
      "icon": "04n"
    }
  ],
  "base": "cmc stations",
  "main": {
    "temp": 293.25,
    "pressure": 1019,
    "humidity": 83,
    "temp_min": 289.82,
    "temp_max": 295.37,
    "sea_level": 984,
    "grnd_level": 990
  },
  "wind": {
    "speed": 5.1,
    "deg": 150
  },
  "clouds": {
    "all": 75
  },
  "rain": {
    "3h": 3
  },
  "snow": {
    "3h": 6
  },
  "dt": 1435658272,
  "sys": {
    "type": 1,
    "id": 8166,
    "message": 0.0166,
    "country": "AU",
    "sunrise": 1435610796,
    "sunset": 1435650870
  },
  "id": 2172797,
  "name": "Cairns",
  "cod": 200
}
```

There are a couple of ways to go about describing this response. You could create a long description that contains all the hierarchy reflected. One challenge with this approach, however, is that it's difficult to keep all the levels straight. With so many nested objects, it's dizzying and confusing. Additionally, it's easy to make mistakes. Worst of all, you can't re-use the individual objects. This undercuts one of the main reasons for storing this object in `components` in the first place.

Another approach is to make each object its own entity in the `components`. Whenever an object contains an object, add a `$ref` value that points to the new object. This way, the objects remain shallow (rather than having multiple levels of nesting), and you won't get lost in a sea of confusing sublevels. (If there's no sub-object, just provide the description directly, without using `$ref`.

Here's the description of the `200` response for the `weather` endpoint. I included the `paths` tag to maintain some context:

Responses object with components documentation:

```yaml
paths:
  /weather:
    get:
      tags:
      - Current Weather Data
      summary: "Call current weather data for one location"
      description: "Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations."
      operationId: CurrentWeatherData
      parameters:
        - $ref: '#/components/parameters/q'
        - $ref: '#/components/parameters/id'
        - $ref: '#/components/parameters/lat'
        - $ref: '#/components/parameters/lon'
        - $ref: '#/components/parameters/zip'
        - $ref: '#/components/parameters/units'
        - $ref: '#/components/parameters/lang'
        - $ref: '#/components/parameters/mode'

      responses:
        200:
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/200'
        404:
          description: Not found response
          content:
            text/plain:
              schema:
                title: Weather not found
                type: string
                example: Not found

components:

  parameters:
    # not shown for the sake of brevity -- see the earlier code block for details
    ...

  schemas:
    200:
      title: Successful response
      type: object
      properties:
        coord:
          $ref: '#/components/schemas/Coord'
        weather:
          type: array
          items:
            $ref: '#/components/schemas/Weather'
          description: (more info Weather condition codes)
        base:
          type: string
          description: Internal parameter
          example: cmc stations
        main:
          $ref: '#/components/schemas/Main'
        visibility:
          type: integer
          description: Visibility, meter
          example: 16093
        wind:
          $ref: '#/components/schemas/Wind'
        clouds:
          $ref: '#/components/schemas/Clouds'
        rain:
          $ref: '#/components/schemas/Rain'
        snow:
          $ref: '#/components/schemas/Snow'
        dt:
          type: integer
          description: Time of data calculation, unix, UTC
          format: int32
          example: 1435658272
        sys:
          $ref: '#/components/schemas/Sys'
        id:
          type: integer
          description: City ID
          format: int32
          example: 2172797
        name:
          type: string
          example: Cairns
        cod:
          type: integer
          description: Internal parameter
          format: int32
          example: 200
    Coord:
      title: Coord
      type: object
      properties:
        lon:
          type: number
          description: City geo location, longitude
          example: 145.77000000000001
        lat:
          type: number
          description: City geo location, latitude
          example: -16.920000000000002
    Weather:
      title: Weather
      type: object
      properties:
        id:
          type: integer
          description: Weather condition id
          format: int32
          example: 803
        main:
          type: string
          description: Group of weather parameters (Rain, Snow, Extreme etc.)
          example: Clouds
        description:
          type: string
          description: Weather condition within the group
          example: broken clouds
        icon:
          type: string
          description: Weather icon id
          example: 04n
    Main:
      title: Main
      type: object
      properties:
        temp:
          type: number
          description: 'Temperature. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.'
          example: 293.25
        pressure:
          type: integer
          description: Atmospheric pressure (on the sea level, if there is no sea_level or grnd_level data), hPa
          format: int32
          example: 1019
        humidity:
          type: integer
          description: Humidity, %
          format: int32
          example: 83
        temp_min:
          type: number
          description: 'Minimum temperature at the moment. This is deviation from current temp that is possible for large cities and megalopolises geographically expanded (use these parameter optionally). Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.'
          example: 289.81999999999999
        temp_max:
          type: number
          description: 'Maximum temperature at the moment. This is deviation from current temp that is possible for large cities and megalopolises geographically expanded (use these parameter optionally). Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.'
          example: 295.37
        sea_level:
          type: number
          description: Atmospheric pressure on the sea level, hPa
          example: 984
        grnd_level:
          type: number
          description: Atmospheric pressure on the ground level, hPa
          example: 990
    Wind:
      title: Wind
      type: object
      properties:
        speed:
          type: number
          description: 'Wind speed. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour.'
          example: 5.0999999999999996
        deg:
          type: integer
          description: Wind direction, degrees (meteorological)
          format: int32
          example: 150
    Clouds:
      title: Clouds
      type: object
      properties:
        all:
          type: integer
          description: Cloudiness, %
          format: int32
          example: 75
    Rain:
      title: Rain
      type: object
      properties:
        3h:
          type: integer
          description: Rain volume for the last 3 hours
          format: int32
          example: 3
    Snow:
      title: Snow
      type: object
      properties:
        3h:
          type: number
          description: Snow volume for the last 3 hours
          example: 6
    Sys:
      title: Sys
      type: object
      properties:
        type:
          type: integer
          description: Internal parameter
          format: int32
          example: 1
        id:
          type: integer
          description: Internal parameter
          format: int32
          example: 8166
        message:
          type: number
          description: Internal parameter
          example: 0.0166
        country:
          type: string
          description: Country code (GB, JP etc.)
          example: AU
        sunrise:
          type: integer
          description: Sunrise time, unix, UTC
          format: int32
          example: 1435610796
        sunset:
          type: integer
          description: Sunset time, unix, UTC
          format: int32
          example: 1435650870
```

I'll explain a bit more in the next sections how to describe the response. In looking at the above code, you may have noticed that not only can you use `$ref` properties in other parts of your spec but also within `components` too.

Notice how the schema definition includes an `example` property for each element? Swagger UI will take this `example` and use it to dynamically build a full code sample in the Responses section in the Swagger UI output. Thus, you don't need big chunks of code for the sample responses in your spec. Instead, these sample responses get built automatically from the schema. It's one of the neat things about Swagger UI. This way, your schema documentation and sample response remain consistent.

Describing a schema

For most of the sections in `components`, you follow the same object descriptions as detailed in the rest of the spec. However, when describing a `schema` object, you use standard keywords and terms from the JSON Schema, specifically from the JSON Schema Specification Wright Draft 00.

In other words, you aren't merely using terms defined by the OpenAPI spec to describe the models for your JSON. As you describe your JSON models (the data structures for input and output objects), the terminology in the OpenAPI spec feeds into the larger JSON definitions and description language for modeling JSON. The OpenAPI's usage of the JSON Schema is just a subset of the full JSON Schema.

The OpenAPI specification doesn't attempt to document how to model JSON schemas. This would be redundant with what's already documented in the JSON Schema site and outside of the scope of the OpenAPI spec. Therefore you might need to consult JSON Schema for more details. (One other helpful tutorial is Advanced Data from API Handyman.)

To describe your JSON objects, you might use the following identifiers:

- `title`
- `multipleOf`
- `maximum`
- `exclusiveMaximum`
- `minimum`
- `exclusiveMinimum`
- `maxLength`
- `minLength`
- `pattern`
- `maxItems`
- `minItems`
- `uniqueItems`
- `maxProperties`
- `minProperties`
- `required`
- `enum`
- `type`
- `allOf`
- `oneOf`
- `anyOf`
- `not`
- `items`
- `properties`
- `additionalProperties`
- `description`
- `format`
- `default`

These data types are also available:

- `integer`
- `long`
- `float`
- `double`
- `string`
- `byte`
- `binary`

- `boolean`
- `date`
- `dateTime`
- `password`

When you start documenting your own schema, start by looking in the OpenAPI's schema object, and then consult the JSON Schema if something isn't covered.

Additionally, look at some example schemas. You can view 3.0 examples here. I usually find a spec that resembles what I'm trying to represent and mimic the same properties and structure.

The `schema` object in 3.0 differs slightly from the schema object in 2.0 — see this post on Nordic APIs for some details on what's new. However, example schemas from 2.0 specs (which are a lot more abundant online) would probably also be helpful as long as you just look at the schema definitions and not the rest of the spec.

A way to cheat – automatically generate the schema from JSON using Stoplight

Describing a JSON response can be complicated and confusing. Fortunately, there's a somewhat easy workaround. To be honest, this is the approach I use when I'm documenting JSON responses. With the Stoplight Studio Editor, you can automatically generate the specification schema syntax from a sample response. See this section in the Stoplight getting started tutorial: Step 5: Enter the responses and response schema information (p. 213).

Using GUI editors to work with the specification code

At this point, you're probably thinking how impractical and error-prone it's going to be as you work directly in the YAML code like this. For this reason, several companies have developed GUI editors to make it easier to work with the specification code. In particular, check out Stoplight (p. 195), which provides an editor that lets you toggle between code and a GUI display. Smartbear also offers SwaggerHub (p. 230), which doesn't necessarily provide a GUI but which gives you inline commenting and versioning tools. (Note that both Stoplight and Smartbear are sponsors of the site, but I would mention them here anyway.)

View the Appearance in Swagger UI

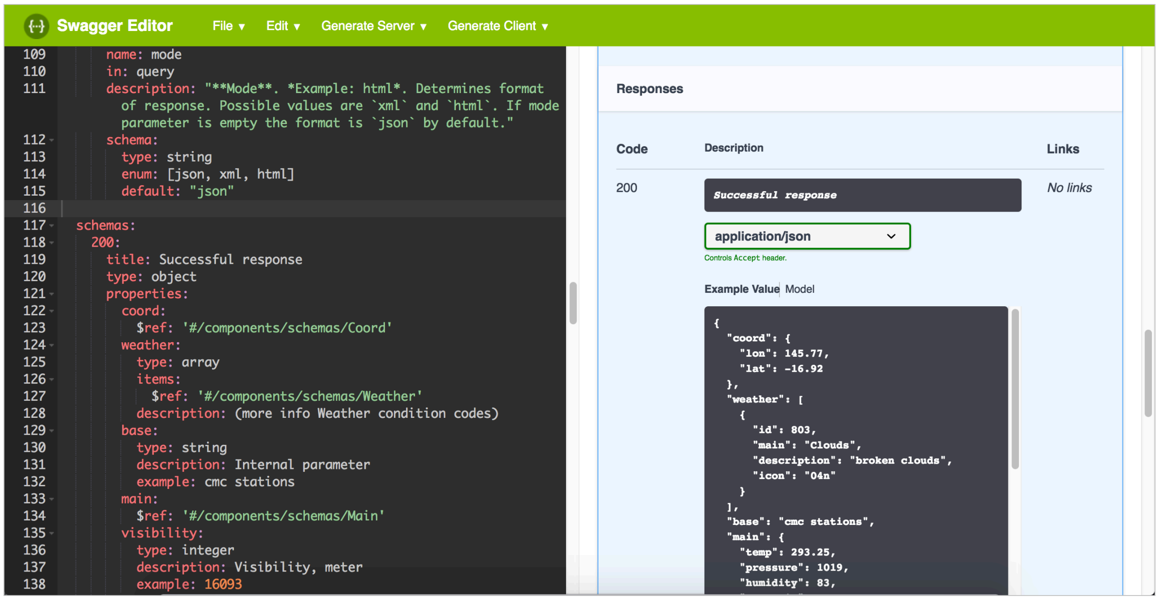
Copy the following code and paste it into the Swagger Editor below your `openapi`, `info` and `servers` objects:

```yaml
paths:
  /weather:
    get:
      tags:
      - Current Weather Data
      summary: "Call current weather data for one location"
      description: "Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations."
      operationId: CurrentWeatherData
      parameters:
        - $ref: '#/components/parameters/q'
        - $ref: '#/components/parameters/id'
        - $ref: '#/components/parameters/lat'
        - $ref: '#/components/parameters/lon'
        - $ref: '#/components/parameters/zip'
        - $ref: '#/components/parameters/units'
        - $ref: '#/components/parameters/lang'
        - $ref: '#/components/parameters/mode'

      responses:
        200:
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/200'
        404:
          description: Not found response
          content:
            text/plain:
              schema:
                title: Weather not found
                type: string
                example: Not found

components:

  parameters:
    q:
      name: q
      in: query
      description: "**City name**. *Example: London*. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by a comma; use ISO 3166 country codes."
      schema:
        type: string
    id:
      name: id
      in: query
      description: "**City ID**. *Example: `2172797`*. You can call by city ID. The API responds with the exact result. The List of city IDs can be downloaded [here](http://bulk.openweathermap.org/sample/). You can include multiple cities in this parameter &mdash; just separate them by commas. The limit of locations is 20. *Note: A single ID counts as a one API call. So, if you have 3 city IDs, it’s treated as 3 API calls.*"
      schema:
        type: string

    lat:
      name: lat
      in: query
      description: "**Latitude**. *Example: 35*. The latitude coordinate of the location of your interest. Must use with `lon`."
      schema:
        type: string

    lon:
      name: lon
      in: query
      description: "**Longitude**. *Example: 139*. Longitude coordinate of the location of your interest. Must use with `lat`."
      schema:
        type: string

    zip:
      name: zip
      in: query
      description: "**Zip code**. Search by zip code. *Example: 95050,us*. Please note that if the country is not specified, the search uses USA as a default."
      schema:
        type: string

    units:
      name: units
      in: query
      description: '**Units**. *Example: imperial*. Possible values: `standard`, `metric`, and `imperial`. When you do not use the `units` parameter, the format is `standard` by default.'
      schema:
        type: string
        enum: [standard, metric, imperial]
        default: "imperial"

    lang:
      name: lang
      in: query
      description: '**Language**. *Example: en*. You can use lang parameter to get the output in your language. We support the following languages that you can use with the corresponded lang values: Arabic - `ar`, Bulgarian - `b
```

```
g`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English - `en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `gl`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Korean - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `nl`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedish - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`, Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese Traditional - `zh_tw`.'
      schema:
        type: string
        enum: [ar, bg, ca, cz, de, el, en, fa, fi, fr, gl, hr, hu, it, ja, kr, la, lt, mk, nl, pl, pt, ro, ru, se, sk, sl, es, tr, ua, vi, zh_cn, zh_tw]
        default: "en"

    mode:
      name: mode
      in: query
      description: "**Mode**. *Example: html*. Determines the format of the response. Possible values are `xml` and `html`. If the mode parameter is empty, the format is `json` by default."
      schema:
        type: string
        enum: [json, xml, html]
        default: "json"

  schemas:
    200:
      title: Successful response
      type: object
      properties:
        coord:
          $ref: '#/components/schemas/Coord'
        weather:
          type: array
          items:
            $ref: '#/components/schemas/Weather'
          description: (more info Weather condition codes)
        base:
          type: string
          description: Internal parameter
          example: cmc stations
        main:
          $ref: '#/components/schemas/Main'
        visibility:
          type: integer
          description: Visibility, meter
          example: 16093
        wind:
          $ref: '#/components/schemas/Wind'
        clouds:
          $ref: '#/components/schemas/Clouds'
```

```
         rain:
           $ref: '#/components/schemas/Rain'
         snow:
           $ref: '#/components/schemas/Snow'
         dt:
           type: integer
           description: Time of data calculation, unix, UTC
           format: int32
           example: 1435658272
         sys:
           $ref: '#/components/schemas/Sys'
         id:
           type: integer
           description: City ID
           format: int32
           example: 2172797
         name:
           type: string
           example: Cairns
         cod:
           type: integer
           description: Internal parameter
           format: int32
           example: 200
     Coord:
       title: Coord
       type: object
       properties:
         lon:
           type: number
           description: City geo location, longitude
           example: 145.77000000000001
         lat:
           type: number
           description: City geo location, latitude
           example: -16.920000000000002
     Weather:
       title: Weather
       type: object
       properties:
         id:
           type: integer
           description: Weather condition id
           format: int32
           example: 803
         main:
           type: string
           description: Group of weather parameters (Rain, Snow, Extreme et
c.)
           example: Clouds
         description:
           type: string
           description: Weather condition within the group
           example: broken clouds
         icon:
           type: string
           description: Weather icon id
           example: 04n
     Main:
       title: Main
       type: object
       properties:
         temp:
           type: number
           description: 'Temperature. Unit Default: Kelvin, Metric: Celsius,
 Imperial: Fahrenheit.'
           example: 293.25
         pressure:
           type: integer
           description: Atmospheric pressure (on the sea level, if there is n
 o sea_level or grnd_level data), hPa
           format: int32
           example: 1019
         humidity:
           type: integer
           description: Humidity, %
           format: int32
           example: 83
         temp_min:
           type: number
           description: 'Minimum temperature at the moment. This is deviatio
 n from current temp that is possible for large cities and megalopolises geog
 raphically expanded (use these parameter optionally). Unit Default: Kelvin,
 Metric: Celsius, Imperial: Fahrenheit.'
           example: 289.81999999999999
         temp_max:
           type: number
           description: 'Maximum temperature at the moment. This is deviatio
 n from current temp that is possible for large cities and megalopolises geog
 raphically expanded (use these parameter optionally). Unit Default: Kelvin,
 Metric: Celsius, Imperial: Fahrenheit.'
           example: 295.37
         sea_level:
           type: number
           description: Atmospheric pressure on the sea level, hPa
           example: 984
         grnd_level:
           type: number
           description: Atmospheric pressure on the ground level, hPa
           example: 990
     Wind:
       title: Wind
       type: object
       properties:
         speed:
           type: number
           description: 'Wind speed. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour.'
           example: 5.0999999999999996
         deg:
           type: integer
           description: Wind direction, degrees (meteorological)
           format: int32
           example: 150
     Clouds:
       title: Clouds
       type: object
       properties:
         all:
           type: integer
           description: Cloudiness, %
           format: int32
           example: 75
     Rain:
       title: Rain
       type: object
       properties:
         3h:
           type: integer
           description: Rain volume for the last 3 hours
           format: int32
           example: 3
     Snow:
       title: Snow
       type: object
       properties:
         3h:
           type: number
           description: Snow volume for the last 3 hours
           example: 6
     Sys:
       title: Sys
       type: object
       properties:
         type:
           type: integer
           description: Internal parameter
           format: int32
           example: 1
         id:
           type: integer
           description: Internal parameter
           format: int32
           example: 8166
         message:
           type: number
           description: Internal parameter
           example: 0.0166
         country:
           type: string
           description: Country code (GB, JP etc.)
           example: AU
         sunrise:
           type: integer
           description: Sunrise time, unix, UTC
           format: int32
           example: 1435610796
         sunset:
           type: integer
           description: Sunset time, unix, UTC
           format: int32
           example: 1435650870

   securitySchemes:
     app_id:
       type: apiKey
       description: API key to authorize requests.
       name: appid
       in: query
```

You should see the following populate in the Swagger UI display:



```
      name: mode
      in: query
      description: "**Mode**. *Example: html*. Determines format
        of response. Possible values are `xml` and `html`. If mode
        parameter is empty the format is `json` by default."
      schema:
        type: string
        enum: [json, xml, html]
        default: "json"

  schemas:
    200:
      title: Successful response
      type: object
      properties:
        coord:
          $ref: '#/components/schemas/Coord'
        weather:
          type: array
          items:
            $ref: '#/components/schemas/Weather'
          description: (more info Weather condition codes)
        base:
          type: string
          description: Internal parameter
          example: cmc stations
        main:
          $ref: '#/components/schemas/Main'
        visibility:
          type: integer
          description: Visibility, meter
          example: 16093
```

Responses

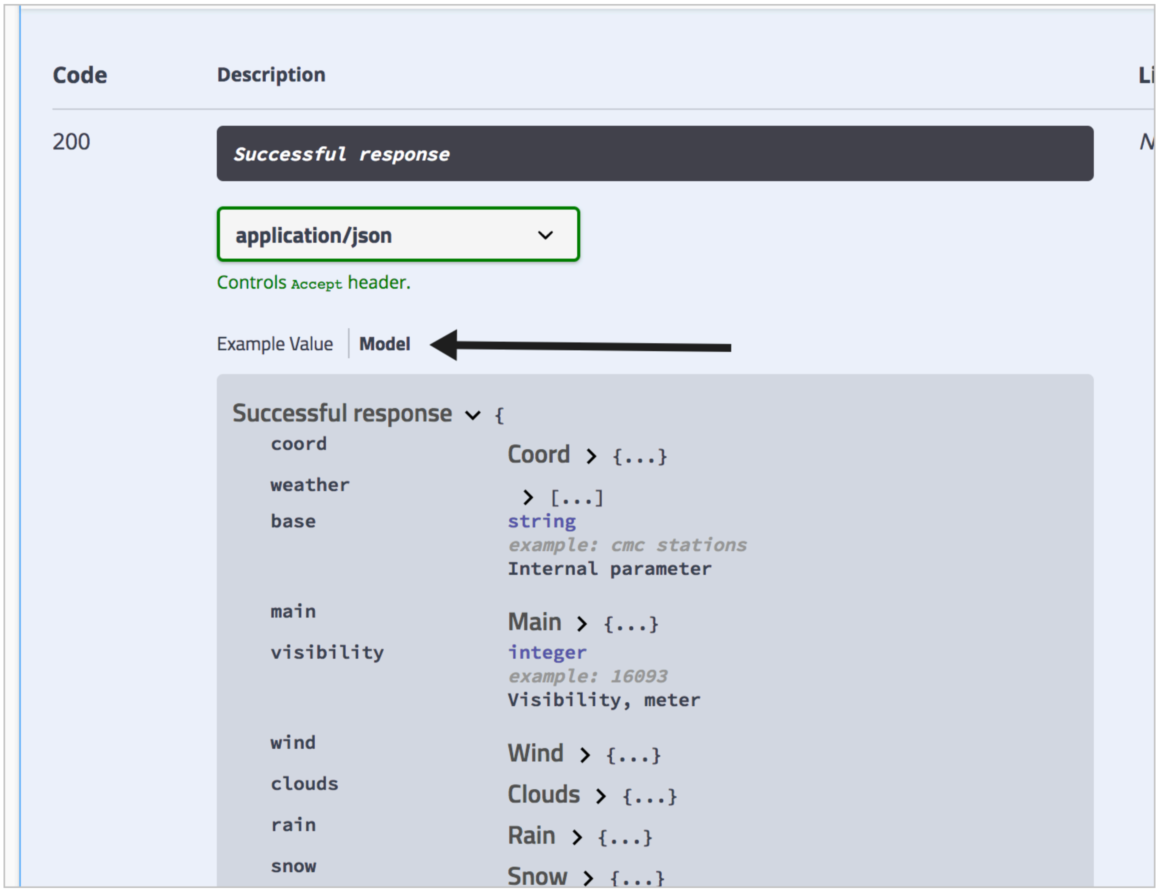
Code	Description	Links
200	*Successful response*	*No links*

```
{
  "coord": {
    "lon": 145.77,
    "lat": -16.92
  },
  "weather": [
    {
      "id": 803,
      "main": "Clouds",
      "description": "broken clouds",
      "icon": "04n"
    }
  ],
  "base": "cmc stations",
  "main": {
    "temp": 293.25,
    "pressure": 1019,
    "humidity": 83,
```

Responses object defined in components

In the Response section, observe how the Example Value code has been dynamically built from the `example` values in the schema to show a sample response.

Also, click the **Model** link to see how the descriptions of each element appear in an expandable/collapsible way:

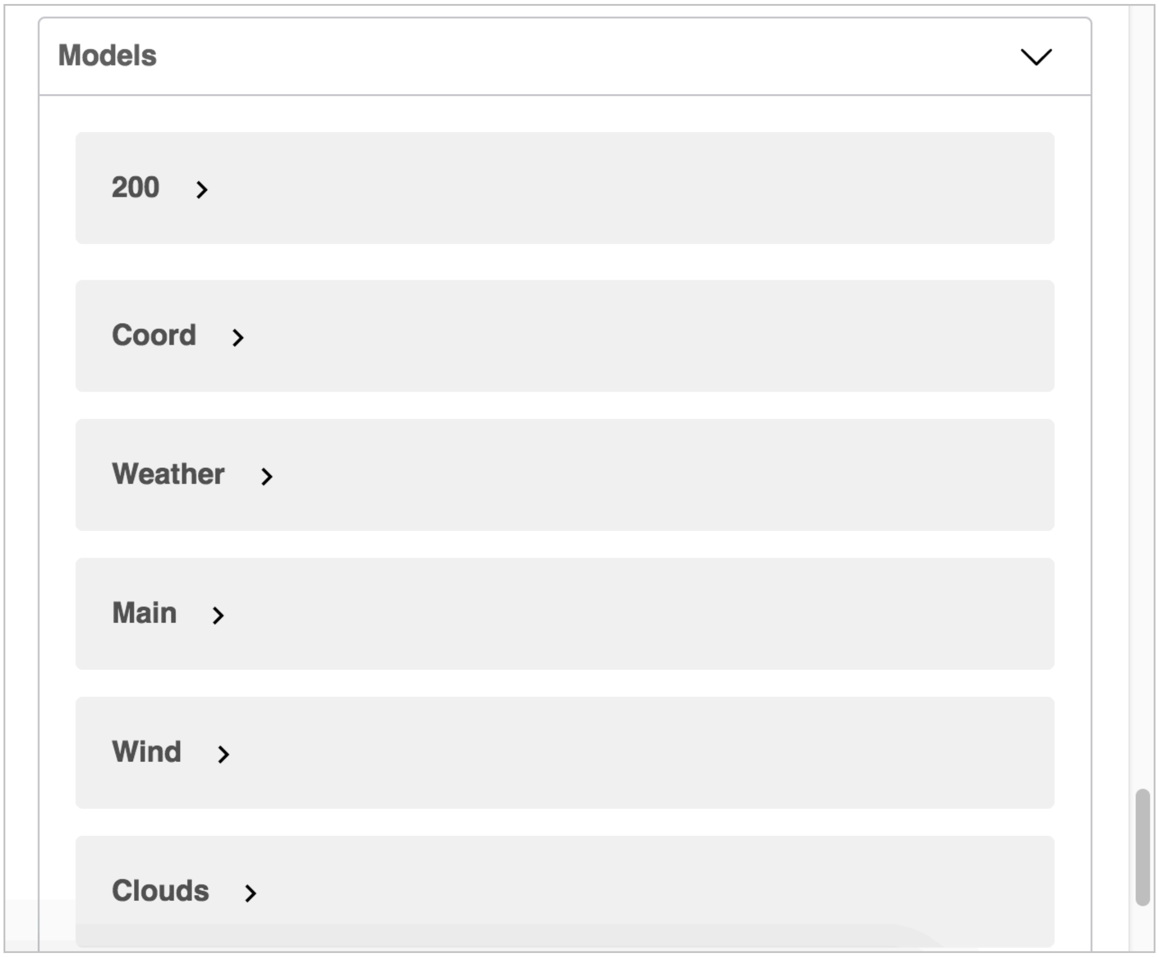


```
Successful response {
    coord          Coord > {...}
    weather          > [...]
    base           string
                   example: cmc stations
                   Internal parameter
    main           Main > {...}
    visibility     integer
                   example: 16093
                   Visibility, meter
    wind           Wind > {...}
    clouds         Clouds > {...}
    rain           Rain > {...}
    snow           Snow > {...}
```

Descriptions appear in the Model

The Models section – why it exists, how to hide it

You'll also notice another "Models" section below all the other paths:



By default, Swagger UI displays each object in `components` in a section called “Models” at the end of your Swagger UI display. If you consolidate all schemas into a single object, without using the `$ref` property to point to new objects, you will see just one object in Models. If you split out the objects, then you see each object listed separately, including the object that contains all the references.

Because I want to re-use objects, I’m going to define each object in `components` separately. As a result, the Models section looks like this:

Why is there a Models section here? Apparently, it was added by popular request because the online Swagger Editor showed the display, and many users asked for it to be incorporated into Swagger UI.

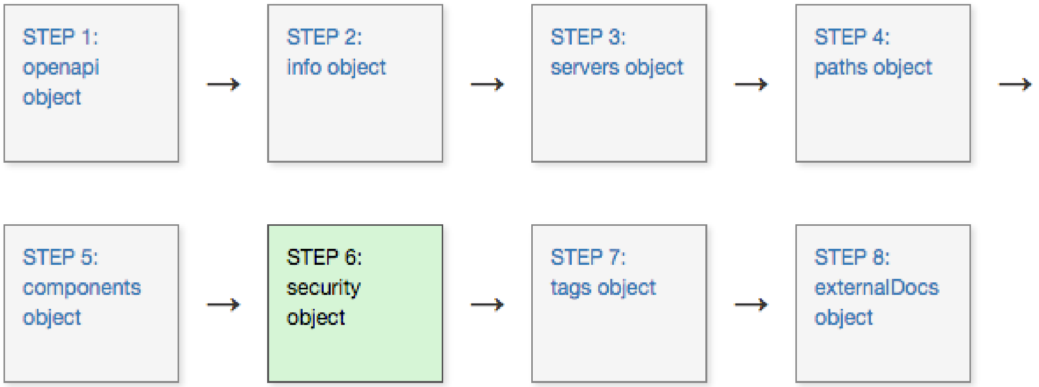
You don’t need this Models section in Swagger UI because both the request and response sections of Swagger UI provide a “Model” link that lets the user toggle to this view.

To hide the Models section, you can add the parameter `defaultModelsExpandDepth: -1` parameter in your Swagger UI project. I provide a Swagger UI tutorial (p. 222) in an upcoming section in this course, with details about the Swagger UI parameters ↗ where you could configure this parameter.

Security definitions

The `components` object also contains a `securitySchemes` object ↗ that defines the authorization method used with each `path`. Rather than dive into the security configuration details here, I explore security in Step 6: The security object (p. 327).

Step 6: The security object (OpenAPI tutorial)



Swagger UI provides a “Try it out” feature that lets users submit actual requests. To submit requests that are authorized by your API server, the spec must contain security information that will authorize the request. The `security` object specifies the security or authorization protocol used when submitting requests.

Which security scheme?

REST APIs can use different security approaches to authorize requests. I explored the most common authorization methods in Authentication and authorization requirements (p. 393). Swagger UI supports four authorization schemes:

- API key
- HTTP
- OAuth 2.0
- Open ID Connect

In this step of the OpenAPI tutorial, we’ll use the API key approach, since this is what the OpenWeatherMap API uses. If your API uses OAuth 2.0 (p. 397) or another method, you’ll need to read the Security Scheme information for details on how to configure it. However, all the security methods mostly follow the same pattern.

API key authorization

The sample OpenWeatherMap API we’re using in this course uses an API key passed in the URL’s query string (rather than in the header). If you submit a request without the API key in the query string (or without a valid API key), the server denies the request. For details on the OpenWeatherMap’s authorization model, see How to start.

Security object

At the root level of your OpenAPI document, add a `security` object that defines the global method for the API's security:

```
security:
- app_id: []
```

`app_id` is the arbitrary name we gave to this security scheme in our `securitySchemes` object. We could have named it anything. We'll define `app_id` in `components`.

All paths will use the `app_id` security method by default unless it's overridden by a value at the `path` object level (p. 291). For example, at the path level, we could overwrite the global security method as follows:

```
/current:
  get:
    ...
    security:
    - some_other_key: []
```

Then the `weather` path would use the `some_other_key` security method, while all other paths would use the globally declared security, `app_id`.

If you get stuck, see the sample OpenAPI spec here for the fully working sample. This will help you spot and troubleshoot indentation or other errors.

Referencing the security scheme in components

In the `components` object (p. 302), add a `securitySchemes` object that defines details about the security scheme the API uses:

```
components:
  ...

  securitySchemes:
    app_id:
      type: apiKey
      description: API key to authorize requests.
      name: appid
      in: query
```

Properties you can use for each item in the `securitySchemes` object include the following:

- `type`: The authorization protocol — `apiKey`, `http`, `oauth2`, or `openIdConnect`.
- `description`: A description of your security method. In Swagger UI, this description appears in the Authorization modal (see the screenshot below). CommonMark Markdown is allowed.
- `name`: The name of the header value submitted in the request. Used only for `apiKey` type security.
- `in`: Specifies where the security key is applied. Options are `query`, `header` or `cookie`. Used only for `apiKey` type security.
- `scheme`. Used with `http` type authorization.

- `bearerFormat`. Used with `http` type authorization.
- `flows` ↗ (object): Used with `oauth2` type authorization.
- `openIdConnectUrl`: Used with `openIdConnect` type authorization.

View the Appearance in Swagger UI

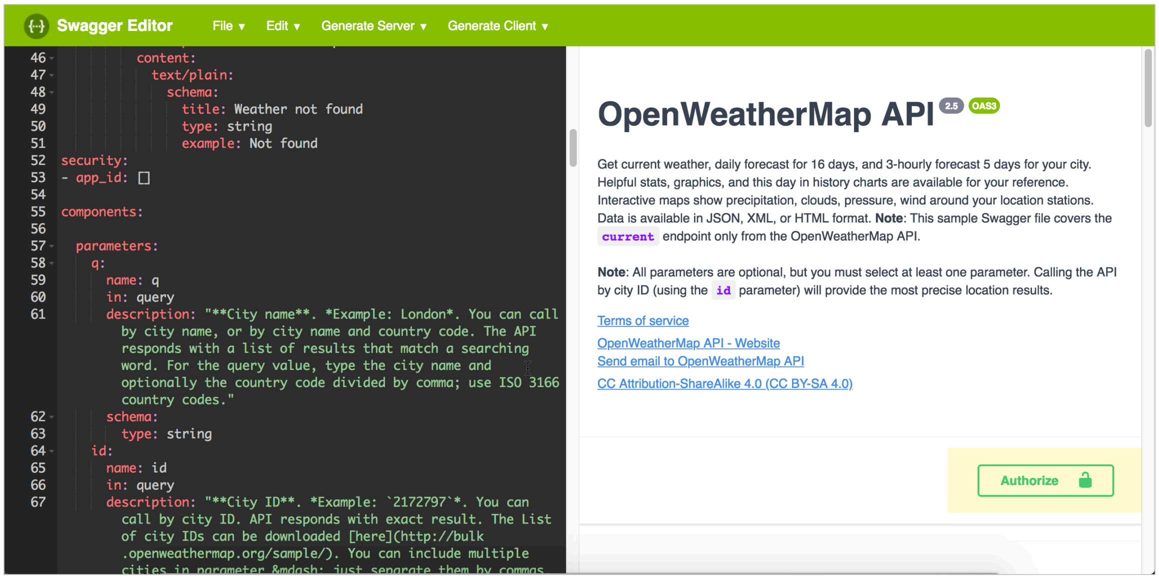
In Swagger Editor, if you haven't already done so, insert the `security` object at the root level:

```
security:
- app_id: []
```

And insert the `securitySchemes` object into `components` (indented at the same level as `parameters` and `responses`):

```
components:
  parameters:
  ...
  responses:
  ...

  securitySchemes:
    app_id:
      type: apiKey
      description: API key to authorize requests.
      name: appid
      in: query
```

Then check out the Swagger UI display. You'll see an "Authorize" button appear.



```
          content:
            text/plain:
              schema:
                title: Weather not found
                type: string
                example: Not found
security:
- app_id: []

components:

  parameters:
    q:
      name: q
      in: query
      description: "**City name**. *Example: London*. You can call
        by city name, or by city name and country code. The API
        responds with a list of results that match a searching
        word. For the query value, type the city name and
        optionally the country code divided by comma; use ISO 3166
        country codes."
      schema:
        type: string
    id:
      name: id
      in: query
      description: "**City ID**. *Example: `2172797`*. You can
        call by city ID. API responds with exact result. The List
        of city IDs can be downloaded [here](http://bulk
        .openweathermap.org/sample/). You can include multiple
        cities in parameter &mdash; just separate them by commas.
```

OpenWeatherMap API

Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

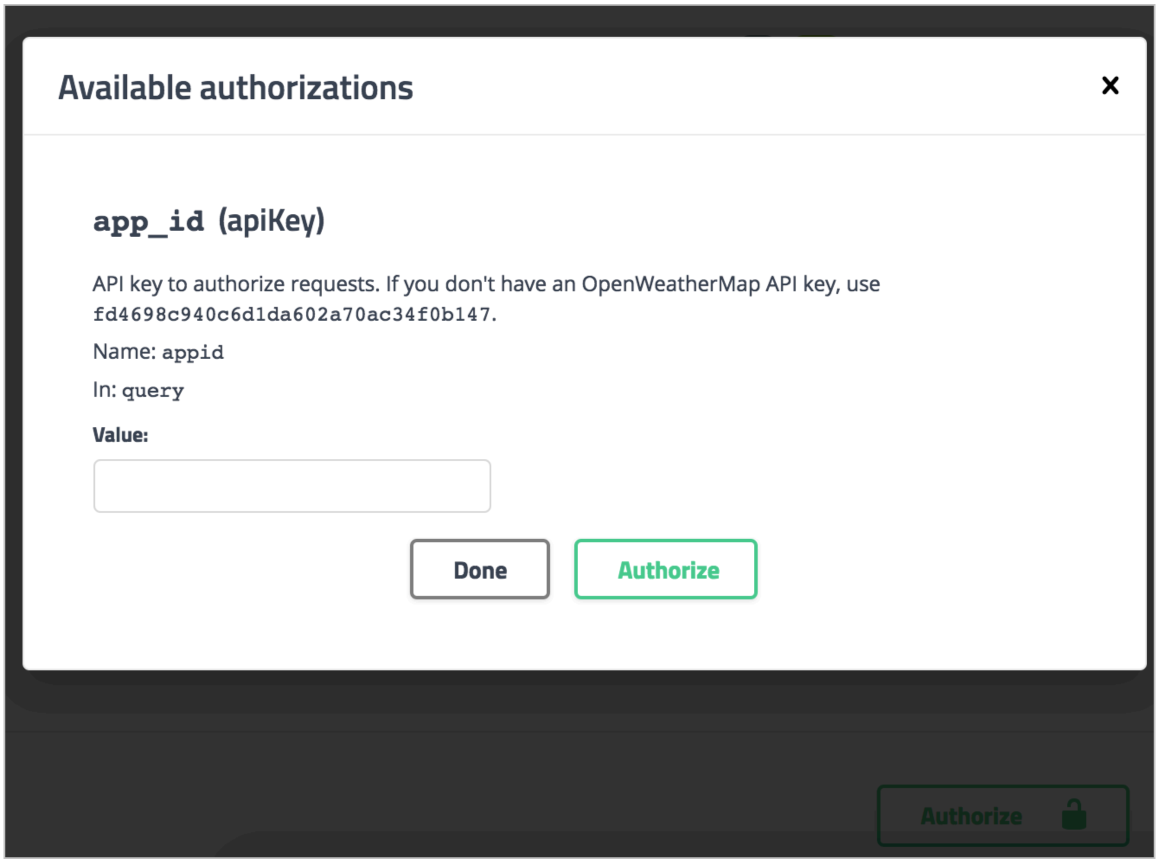
Terms of service

OpenWeatherMap API - Website
Send email to OpenWeatherMap API

CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)

Adding security information into the spec

When you click Authorize, the `description` and other security details appear:



Available authorizations

app_id (apiKey)

API key to authorize requests. If you don't have an OpenWeatherMap API key, use `fd4698c940c6d1da602a70ac34f0b147`.

Name: `appid`

In: `query`

Value:

After users enter an API key and click **Authorize**, the authorization method is set for as many requests as they want to make. The authorization session expires only when users refresh the page.

Test out how authorization works

Now that we've added authorization, let's try making an actual API request. In the Swagger Editor (the right pane), click the **Authorize** button, paste the sample API key shown in the description into the **Value** field (or use your own OpenWeatherMap API key (p. 69)), and click **Authorize**. Then click **Close** to close the authorization modal.

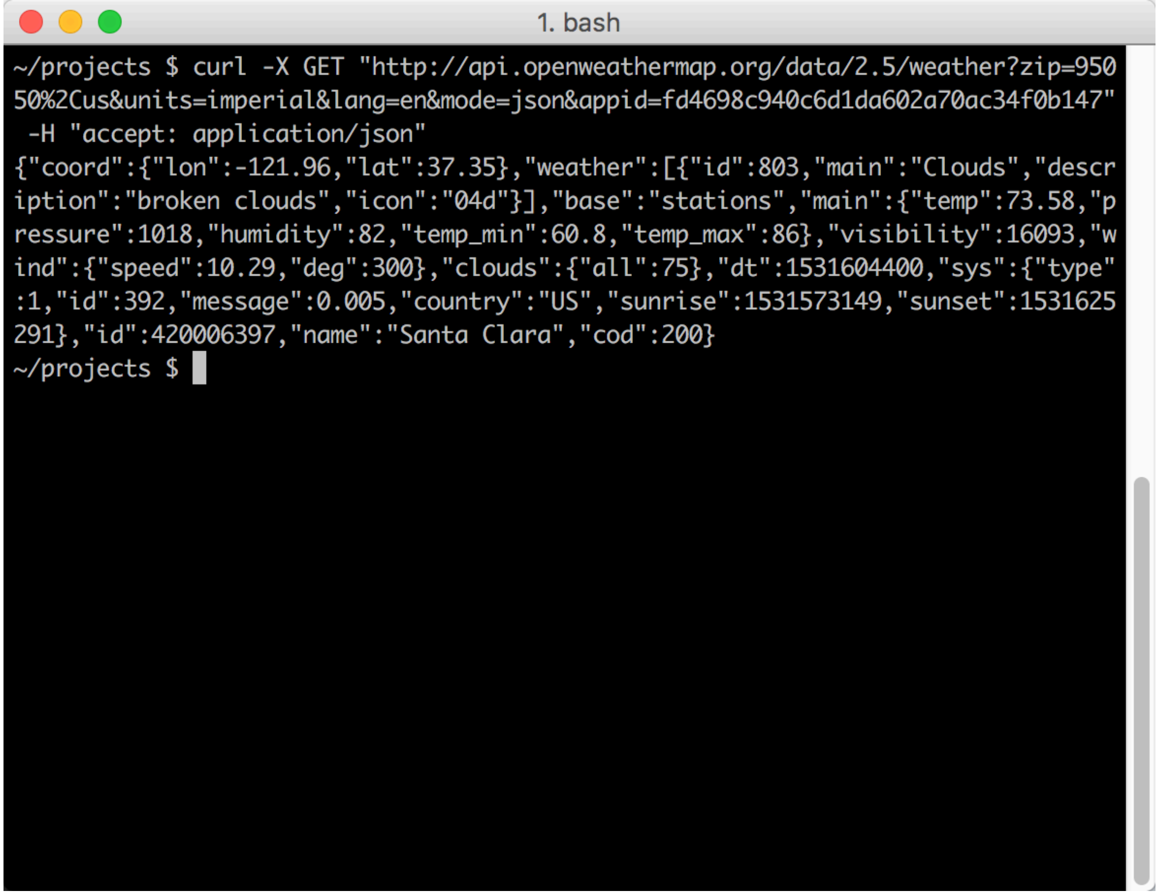
In the Current Weather Data section, expand the **GET weather** endpoint and click **Try it out**. In the **zip** field, enter your zip code and country abbreviation (such as `95050,us`), and then click **Execute**.

When you execute the request, Swagger UI shows you the curl request (p. 80) submitted. For example, after executing a weather request, the curl is as follows:

```
curl -X GET "https://api.openweathermap.org/data/2.5/weather?zip=95050%2Cu
s&units=imperial&lang=en&mode=json&appid=APIKEY" -H "accept: application/jso
n"
```

(Replace APIKEY with your actual API key.)

The `&appid=APIKEY"` indicates that the API key is being included in the query string, so the request will be authorized. If you copy the curl submitted, customize the API key, and paste it into the command line, you'll see a successful response:



```
~/projects $ curl -X GET "http://api.openweathermap.org/data/2.5/weather?zip=950
50%2Cus&units=imperial&lang=en&mode=json&appid=fd4698c940c6d1da602a70ac34f0b147"
 -H "accept: application/json"
{"coord":{"lon":-121.96,"lat":37.35},"weather":[{"id":803,"main":"Clouds","descr
iption":"broken clouds","icon":"04d"}],"base":"stations","main":{"temp":73.58,"p
ressure":1018,"humidity":82,"temp_min":60.8,"temp_max":86},"visibility":16093,"w
ind":{"speed":10.29,"deg":300},"clouds":{"all":75},"dt":1531604400,"sys":{"type"
:1,"id":392,"message":0.005,"country":"US","sunrise":1531573149,"sunset":1531625
291},"id":420006397,"name":"Santa Clara","cod":200}
~/projects $
```

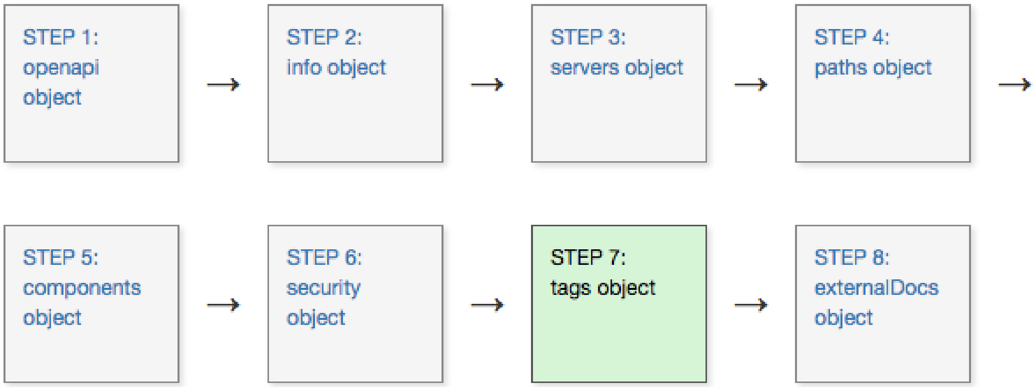
Successful curl response

The server response also appears directly in the Swagger UI display, with a link to download it:

```
{
  "coord": {
    "lon": -121.96,
    "lat": 37.35
  },
  "weather": [
    {
      "id": 500,
      "main": "Rain",
      "description": "light rain",
      "icon": "10d"
    },
    {
      "id": 701,
      "main": "Mist",
      "description": "mist",
      "icon": "50d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 55.24,
    "pressure": 1012,
    "humidity": 77,
    "temp_min": 51.08,
    "temp_max": 59
  },
  "visibility": 16093,
  "wind": {
    "speed": 5.82,
    "deg": 320
  },
  "rain": {
    "1h": 0.25
  },
  "clouds": {
    "all": 40
  },
  "dt": 1544039760,
  "sys": {
    "type": 1,
    "id": 5122,
    "message": 0.0052,
    "country": "US",
    "sunrise": 1544022470,
    "sunset": 1544057391
  },
  "id": 420006397,
  "name": "Santa Clara",
  "cod": 200
}
```

Note that when you implement Swagger UI, if you find that the curl request works but the response doesn’t appear in Swagger UI, there might be a CORS issue with your API blocking requests from web applications like Swagger. See Troubleshooting issues with Swagger UI (p. 227) for details.

Step 7: The tags object (OpenAPI tutorial)



The `tags` object allows you to arrange the `paths` (endpoints) into named groups in the Swagger UI display.

Defining tags at the root level

At the root level, the `tags` object lists all the tags that are used in the operation objects (which appear within the `paths` object, as explained in Step 4: The paths object (p. 291)). Here's an example of the `tags` object for our OpenWeatherMap API:

```
tags:
  - name: Current Weather Data
    description: "Get current weather details"
```

We have just one tag, but you could have as many as you want (if you have a lot of endpoints, it would make sense to create multiple tags to group them). You can list both the `name` and a `description` for each tag. The `description` appears as a subtitle for the tag name in the Swagger UI display.

If you get stuck, see the sample OpenAPI spec here for the fully working sample. This will help you spot and troubleshoot indentation or other errors.

Tags at the path object level

The `tags` object at the root level should list all tags (groups) that you want in your API. Then in each path object under `paths`, you list the tag you want that path grouped under.

By “root level,” I mean the first level in the OpenAPI document. This level is also referred to as the global level because some object properties declared here (namely servers and security) are applied to each of the operation objects unless overridden at a lower level.

For example, in the operations object for the `/current` path, we used the tag `Current Weather Data`:

```
paths:
  /weather:
    get:
      tags:
      - Current Weather Data
```

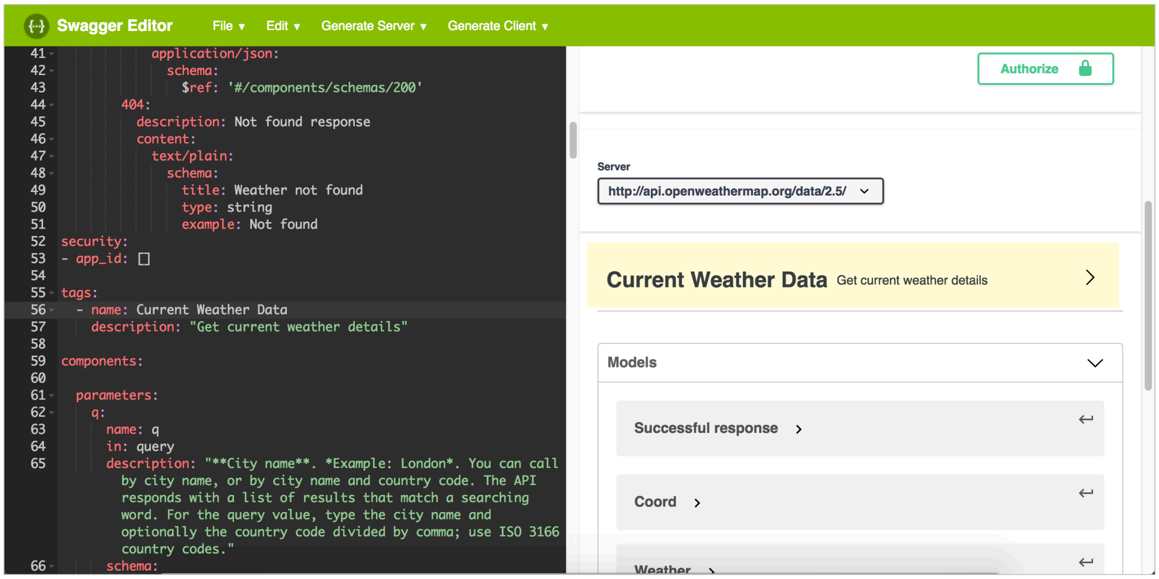
This tag is defined at the global level, so the `/weather` path will be grouped here.

View the Appearance in Swagger UI

Add the following to the root level of your OpenAPI document in Swagger Editor:

```
tags:
  - name: Current Weather Data
    description: "Get current weather details"
```

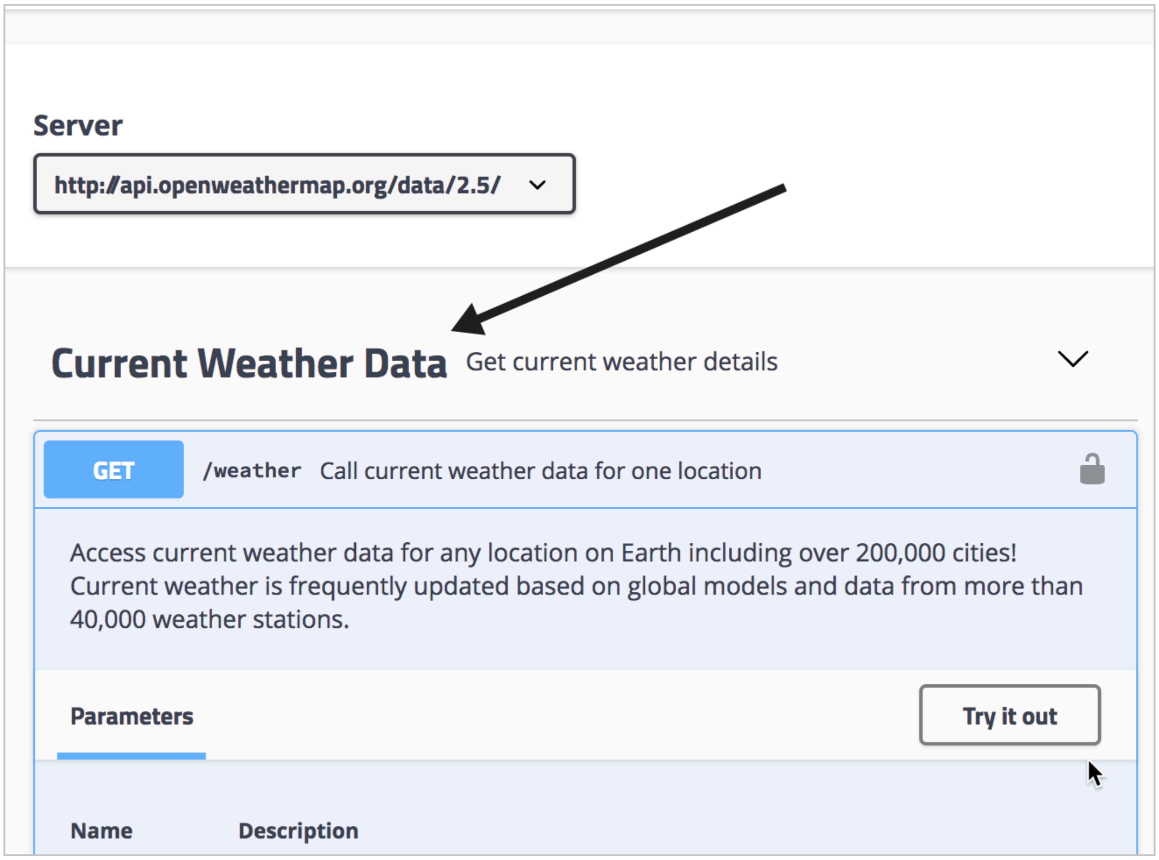
Observe how the description appears next to the collapsed Current Weather Data section.



```
            application/json:
              schema:
                $ref: '#/components/schemas/200'
        404:
          description: Not found response
          content:
            text/plain:
              schema:
                title: Weather not found
                type: string
                example: Not found
security:
- app_id: []

tags:
  - name: Current Weather Data
    description: "Get current weather details"

components:

  parameters:
    q:
      name: q
      in: query
      description: "**City name**. *Example: London*. You can call
        by city name, or by city name and country code. The API
        responds with a list of results that match a searching
        word. For the query value, type the city name and
        optionally the country code divided by comma; use ISO 3166
        country codes."
      schema:
```

Tags defined at the root level

All paths that have the same tag are grouped together in the display. For example, paths that have the `Current Weather Data` tag will be grouped together under the title `Current Weather Data`. Each group title is a collapsible/expandable toggle.

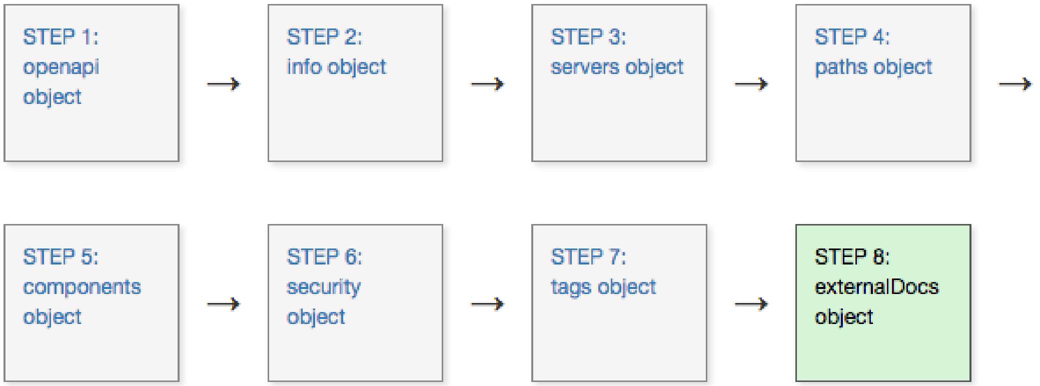

Server
http://api.openweathermap.org/data/2.5/
Current Weather Data
Get current weather details
GET
/weather
Call current weather data for one location
Parameters
Try it out
Name
Description

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

The order of the tags in the `tags` object at the root level determines their order in Swagger UI. Additionally, the `descriptions` appear to the right of the tag name.

In our sample OpenAPI spec, tags don't seem all that necessary since we're just documenting one path/endpoint. (Additionally, I configured the Swagger UI demo (p. 240) to expand the section by default.) But imagine if you had a robust API with 30+ paths to describe. You would certainly want to organize the paths into logical groups for users to navigate.

Step 8: The externalDocs object (OpenAPI tutorial)



The `externalDocs` object lets you link to external documentation. You can also provide links to external docs in the `paths` object.

Example externalDocs object

Here's an example of an `externalDocs` object:

```
externalDocs:
  description: API Documentation
  url: https://openweathermap.org/api
```

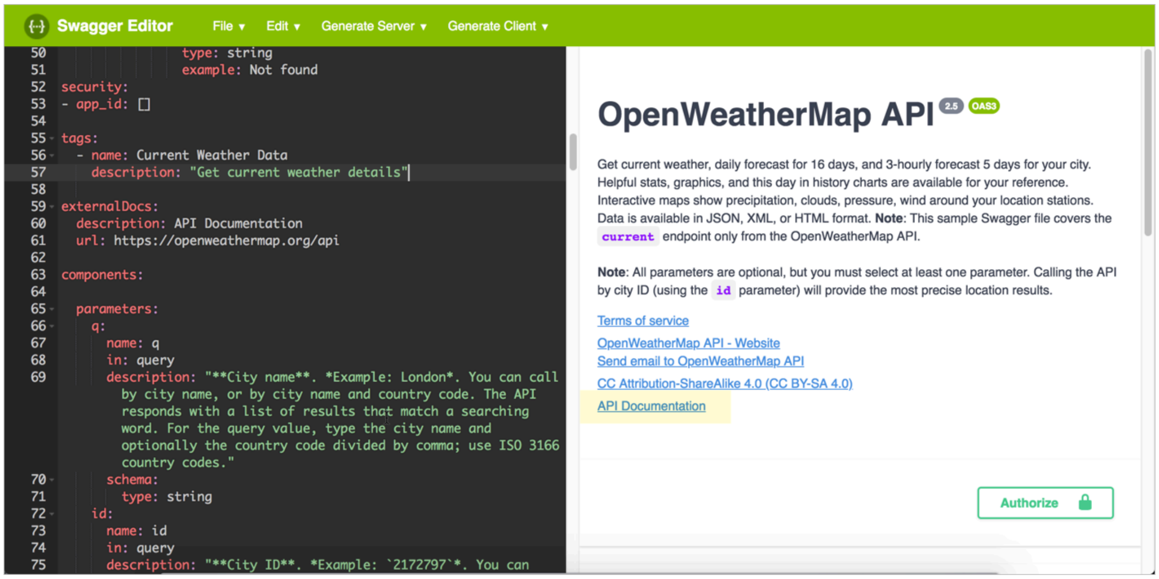
Note that this documentation should relate to the API as a whole. To link a specific parameter to more documentation, you can add an `externalDocs` object to the operation object, as noted in Operation objects (p. 291) section in Step 4: The paths object.

If you get stuck, see the sample OpenAPI spec here for the fully working sample. This will help you spot and troubleshoot indentation or other errors.

View the Appearance in Swagger UI

Add the above code to the root level of your OpenAPI document in Swagger UI.

When you do, in the Swagger UI, a link appears after the API description along with other info about the API:



```
                type: string
                example: Not found
security:
- app_id: []

tags:
  - name: Current Weather Data
    description: "Get current weather details"

externalDocs:
  description: API Documentation
  url: https://openweathermap.org/api

components:

  parameters:
    q:
      name: q
      in: query
      description: "**City name**. *Example: London*. You can call
        by city name, or by city name and country code. The API
        responds with a list of results that match a searching
        word. For the query value, type the city name and
        optionally the country code divided by comma; use ISO 3166
        country codes."
      schema:
        type: string
    id:
      name: id
      in: query
      description: "**City ID**. *Example: `2172797`*. You can
```

OpenWeatherMap API 2.5 OAS3

Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

Terms of service

OpenWeatherMap API - Website
Send email to OpenWeatherMap API

CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)

API Documentation

External documentation link

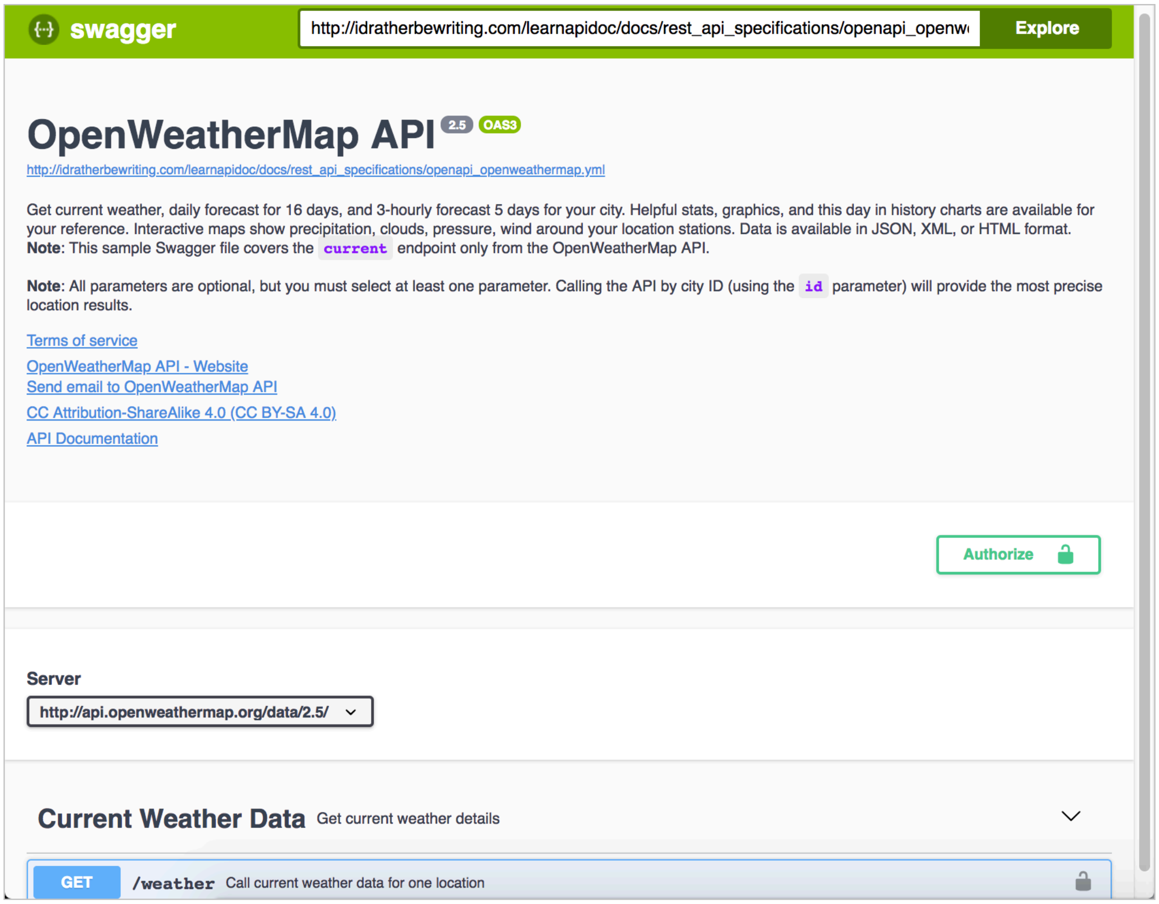
At this point, you can probably anticipate some challenges with integrating Swagger UI with the rest of your documentation. It seems that you will likely have two outputs and a semi-fragmented user experience. The `externalDocs` object at least gives you a predictable place to link back to your other conceptual topics (p. 358). See Integrating Swagger UI with the rest of your docs (p. 241) for more information on integration strategies.

Seeing the finished result

Now that we've completed all the steps in the tutorial, we're finished building our OpenAPI specification document.

You can see the complete specification document here: openapi_openweathermap.yml.

Here's the specification document rendered by Swagger UI:


swagger
http://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_openw
Explore
2.5
OAS3
Authorize
GET

OpenWeatherMap API

http://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_openweathermap.yml

Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

Terms of service

OpenWeatherMap API - Website
Send email to OpenWeatherMap API

CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)

API Documentation

Server

http://api.openweathermap.org/data/2.5/

Current Weather Data Get current weather details

`/weather` Call current weather data for one location

Try executing a request in the version above and look at the result. In the result, locate the `temp` value in the `main` object. Then take a break by going outside and evaluate whether the temperature outside matches the response.

You can insert any valid path to an OpenAPI specification document in the “Explore” box in Swagger UI (assuming the version of Swagger UI supports your OpenAPI version), and it will display the API documentation. For example, you could insert `https://petstore.swagger.io/v2/swagger.json` (then click **Explore**) and it would show the Petstore API.

Activity: Create an OpenAPI specification document

The OpenAPI tutorial (p. 272) walked you through 8 steps in building the OpenAPI specification document. Now it's your turn to practice first editing and then creating an OpenAPI specification document.

Activity: Edit an existing OpenAPI specification document

Use this simple Sunrise and sunset times API ↗ to get more familiar with the process of creating an OpenAPI specification file. This Sunrise and sunset times API doesn't require authentication with requests, so it removes some of the more complicated authentication workflows (the spec file doesn't require the security object (p. 327)). In this activity, you'll edit some of the existing values in an OpenAPI specification document that's already written.

To edit the OpenAPI specification file:

1. Copy the code from this pre-built OpenAPI specification ↗.
2. Paste the YAML content into the Swagger Editor ↗.
3. Identify each of the root-level objects of the OpenAPI spec:
    - Step 1: The openapi object (p. 282)
    - Step 2: The info object (p. 285)
    - Step 3: The servers object (p. 288)
    - Step 4: The paths object (p. 291)
    - Step 5: The components object (p. 302)
    - Step 8: The externalDocs object (p. 337)
4. In the `info` object (near the top), make some changes to the `description` property and see how the visual display in the right column gets updated.
5. In the `parameters` object, make some changes to one of the `description` properties and see how the visual editor gets updated.
6. Look for the `$ref` pointer in the `responses` object. Identify what it points to in `components`.
7. Change some spacing in a way that makes the spec invalid (such as inserting a space before `info`), and look at the error that appears. Then revert the invalid space.
8. Expand the **Get** section and click **Try it out**. Then click **Execute** and look at the response.

Create the OpenAPI specification document for an API of your choosing

In an earlier activity, you found an open-source API project (p. 686) with some documentation needs. Try creating an OpenAPI specification for this API. Depending on the API you choose to work with, you could potentially use this specification document as part of your portfolio.

If your open-source project doesn't have an API, or if it's API already has an OpenAPI specification, find another API (perhaps from this list of 100+ APIs (p. 536)) and create the OpenAPI specification.

Go through each step of the OpenAPI specification tutorial to build out the specification document:

- Step 1: The openapi object (p. 282)
- Step 2: The info object (p. 285)
- Step 3: The servers object (p. 288)
- Step 4: The paths object (p. 291)

- Step 5: The components object (p. 302)
- Step 6: security object (p. 327)
- Step 7: The tags object (p. 334)
- Step 8: The externalDocs object (p. 337)

Validate your specification document in the Swagger Editor. Execute a request to make sure it's working correctly.

Chapter 6: Testing API documentation

Testing your documentation is critical to providing accurate, thorough information. With API and developer docs, due to the high level of complexity and engineering requirements, technical writers might be inclined to simply take information that engineers give them and incorporate it wholesale, without personally testing it. Merely playing an editorial/publishing function, however, can reduce your role to that of an engineer's secretary.

Chapter sections:

Overview of testing your docs.......... 343

Set up a test environment.......... 345

Test all instructions yourself.......... 348

Test your assumptions.......... 353

Activity: Test your project's documentation.......... 357

Overview of testing your docs

Up until now, you've been focused intensely on API documentation. In this section, I'll talk about an aspect of API documentation that is perhaps more applicable to all types of documentation but which is especially relevant to developer docs, where testing and experimenting with the products and services is not always straightforward.

Walking through all the steps in documentation yourself is critical to producing high-quality, accurate instructions. The more complex setup you have, the more difficult it can be to test all of the steps. Still, if you want to move beyond merely editing and publishing engineer-written documentation, you'll need to build sample apps or set up the systems necessary to test the API docs. These tests should mirror what actual users will do as closely as possible.

Leveraging test cases from QA

When you start setting up tests for your documentation, you typically interact with the quality assurance (QA) team. Developers might be helpful too, but the quality assurance team already has, presumably, a test system in place, usually a test server, and test cases. "Test cases" are the various scenarios that the product needs to be tested against.

You'll want to make friends with the quality assurance team and find out best practices for testing scenarios relevant to your documentation. They can usually help you get started in an efficient way, and they'll be excited to have more eyes on the system. If you find bugs, you can either forward them to QA or log them yourself in the team's issue tracker.

If you can hook into a set of test cases that QA teams use to run tests, you can often get a jump start on the tasks you're documenting. Good test cases usually list the steps required to produce a result, and the scripts can inform the documentation you write.

Ways to test content

Testing your API doc content is so critical, I've created an entire section devoted to this topic. This section includes three topics:

- Set up a test environment (p. 345)
- Test all instructions yourself (p. 348)
- Test your assumptions (p. 353)

Photo from Flickr, CC BY 2.0 — City water testing laboratory, 1948. When I think about testing docs, I like to think of myself as a scientist in a laboratory, carefully setting up tests to measure reactions and outcomes.

Set up a test environment

The first step to testing your instructions is to set up a test environment. Without this test environment, it will be difficult to make any progress in testing your instructions.

Kinds of test environments

The kind of test environment you set up depends on your product and company. In the following sections, I explain testing setup details for different scenarios:

- Testing on a test server (p. 345)
- Testing local builds (p. 345)
- Testing sample apps in specific programming languages (p. 346)
- Testing hardware products (p. 346)

Testing on a test server

The easiest way to test an API is by making requests to a test server where the API service is configured. QA can usually help you with access to the test server. With the test server, you'll need to get the appropriate URLs, login IDs, roles, etc. Ask QA if there's anything you shouldn't alter or delete because sometimes the same testing environment is shared among groups.

Additionally, make sure your logins correspond with the permissions users will have. If you have an admin login, you might not experience the same responses as a regular user.

You may also need to construct certain files necessary to configure a server with the settings you want to test. Understanding exactly how to create the files, the directories to upload them to, the services to stop and restart, and so on, can require a lot of initial investigation.

Exactly what you have to do depends on your product, the environment, the company, and security restrictions, etc. No two companies are alike. Sometimes it's a pain to set up your test system, and other times it's a breeze.

At one company, to gain access to the test system, we had to jump over a series of security hurdles. For example, connections to the web services from internal systems required developers to go through an intermediary server. To connect to the web server test instance, you had to SSL to the intermediary server, and then connect from the intermediary to the web server. (This wasn't something users would need to do, just internal engineers.)

The first time I attempted this, I asked a developer to help me set this up. I carefully observed the commands and steps he went through on my computer. I later documented it for future knowledge purposes, and other engineers used my doc to set up the same access.

Testing local builds

Many times, developers work on a local instance of the system before pushing it to a test server. In other words, they build the app or web server entirely on their own machines and run through test code there, long before pushing it to a test server. If you're plugged into the project this early, great — you might have more influence on the API's design and can influence changes as needed. To build code locally, you may need to install special utilities or frameworks, become familiar with various command line operations to build the code, and more.

If you can get the local builds running on your own machine, it's usually worthwhile because it can empower you to document content ahead of time, long before the release.

If it's too complicated to set up a local environment, you can ask an engineer to install the local system on your machine. Sometimes developers like to just sit down at your computer and take over the task of installing and setting up a system. They can work quickly on the command terminal and troubleshoot systems or quickly proceed through installation commands that would otherwise be tedious to walk you through.

Many times, developers aren't too motivated to set up your system, so they may give you a quick explanation about installing this and that tool. But never let a developer say "Oh, you just do a, b, and c." Then you go back to your desk and nothing works, or it's much more complicated than he or she said it would be. It can take persistence to get everything set up and working the first time.

If a developer is knee-deep in sprint tasks and heavily backlogged, he or she may not have time to help you properly get set up. Be patient and ask the developer to indicate a good time to go over the setup.

With local builds, setting up a functional system is much more challenging than using a test server. Still, if you want to write good documentation, setting up a test system is essential. Good developers know and recognize this need, and so they're usually accommodating (to an extent) in helping set up a test environment to get you started.

Testing sample apps

Depending on the product, you might also have a sample app (p. 464) or SDK (p. 480) in your code deliverables. You often include a sample app or SDK (or multiple apps and SDKs in various programming languages) with a product to demonstrate how to integrate and call the API. If you have a test app that integrates the API, you'll probably need to install some programs or frameworks on your own machine to get the sample app working.

For example, you might have to build a sample Java app to interact with the system — so you'd likely need to have the Java Development Kit and a Java IDE installed on your computer to make it work. If the app is in PHP, you probably need to install PHP. Or if it's an Android app, you will probably need to download Android Studio and connect it to virtual (or actual) device.

There are usually fewer instructions about how to run a sample app because developers assume users will already have these environments set up on their machines. (It wouldn't make sense for a user to choose the Java app if they didn't already have a Java environment, for example.)

The sample app is among the most helpful pieces of documentation. As you set up the sample app and get it working, look for opportunities to add documentation in the code comments. At the very least, get the sample app working on your own computer and include the setup steps in your documentation.

Testing hardware products

If you're documenting a hardware product, you'll want to secure a device that has the right build installed on it. Big companies often have prototype devices available. At some companies, there may be kiosks where you can "flash" (quickly install) a specific build number on the device. Or you may send your device's serial number to someone who manages a pool of devices that receive beta-version updates from the cloud.

With some hardware products, it may be difficult to get a test instance of the product to play with. I once worked at a government facility documenting a million-dollar storage array. The only time I was allowed to see the storage array was by signing into a special data server room environment, accompanied by an engineer, who wouldn't dream of letting me actually touch the array, much less swap out a storage disk, run commands in the terminal, replace a RAID, or do some other task (for which I was writing instructions).

I learned early on to steer my career towards jobs where I could actually get my hands on the product, usually software code, and play around with it. When I was working at Amazon, I had a drawer full of Fire TV devices and prototypes. I happened to work in the Amazon devices section, so I had almost every model of Fire TV, Echo Dot, Echo Spot, Echo Show, TVs with Fire OS, and more. Through Android Studio and Android Debug Bridge (ADB), I would run an app on one of these physical devices to test how it works. It's a lot of fun to play around with different devices.

If you're documenting hardware, you need access to the hardware to provide reliable documentation on how to use it. You'll need to understand how to run apps on the device and how to interface with it. Hopefully, the product is one that you can access to play around with in a direct, hands-on way.

Working at Google, the product I was documenting included Google services, such as Google Maps, Play Store, and Assistant, embedded directly in the car's infotainment system. These services are built on the Android operating system. The group I was in maintained a fleet of vehicles, mostly Volvo Polestars and XC-90s, with test installs of Android and these Google services. One day I decided to check a fleet car out and drove it around for a couple of weeks. The experience gave me a much better understanding of the product, one that I didn't entirely possess before when I was just writing about the features. I could experience first-hand in many real situations what the user experience was with the product, and while in some cases I was underwhelmed, in others I was kind of like, wow, this is cool. I logged about 25 bugs while I had the car during this time.

If you encounter developer resistance …

Many times developers don't expect that a technical writer will be doing anything more than just transcribing and relaying the information given to him or her by engineers. With this mindset, a developer might not immediately think that you need or want a sample app to test out the calls or other code. You might need to ask (or even petition) the developers for it.

I've found that most of the time, developers respect technical writers much more if the technical writers can test out the code themselves. Engineers also appreciate any feedback you may have as you run through the system. Technical writers, along with QA, are usually the first users of the developer's code.

If a developer or QA person can't give you access to any such test server or sample code, be suspicious. How can a development and QA team create and test their code without a sample system where they expect it to be implemented? And if there's a sample system, why can't you also have access so you can write accurate, thorough documentation on how to use it?

Sometimes developers don't want to go through the effort of getting something working on your machine, so you may have to explain more about your purpose and goals in testing. If you run into friction, be persistent. It might take one or more days to get your test environment set up. For example, it took me several days to get an app framework to build successfully on my Fire TV device. But once you have a test system set up, it makes it much easier to create documentation because you can start to answer your own questions.

Next steps

After you get the test environment set up, it's time to test your instructions (p. 348).

Test all instructions yourself

After setting up your test environment (p. 345), the next step is to test your instructions. This will likely involve testing API endpoints with various parameters along with other configurations. Testing all your docs can be challenging, but it's where you'll get the most useful insights when creating documentation.

Benefits of testing your instructions

One benefit of testing your instructions is that you can start to answer your own questions. Rather than taking the engineer's word for it, you can run a call, see the response, and learn for yourself. (This assumes the application is behaving correctly, though, which may not be the case.)

A lot of times, when you discover a discrepancy in what's supposed to happen, you can confront an engineer and tell him or her that something isn't working correctly. Or you can make suggestions for improving workflows, terms, responses, error messages, etc. You can't do this if you're just taking notes about what engineers say, or if you're just copying information from wiki specs or engineer-written pages.

When things don't work, you can identify and log bugs in issue tracking systems such as JIRA. Logging bugs is helpful to the team overall and increases your credibility with the engineers. It's also immensely fun to log a bug against an engineer's code because it shows that you've discovered flaws and errors in what the "gods of code" have created.

Other times, the bugs are within your documentation. For example, on one project, through testing API calls I realized I had one of my parameters wrong. Instead of `verboseMode`, the parameter was simply `verbose`. This subtle discrepancy is one of those details you don't discover unless you test something, find it doesn't work, and then set about figuring out what's wrong.

If you're testing a REST API, you can submit the test calls using curl (p. 80), Postman (p. 71), or another REST client. Save the calls so that you can quickly run a variety of scenarios.

When you start to run tests and experiments, you'll begin to discover what does and does not work. For example, at one company, after setting up a test system and running some calls, I learned that part of my documentation was unnecessary. I thought that field engineers would need to configure a database with a particular code themselves, when it turned out that IT operations would actually be doing this configuration.

I didn't realize this until I started to ask how to configure the database, and an engineer said that my audience wouldn't be able to do that configuration, so it shouldn't be in the documentation.

It's little things like that, which you learn as you're going through the process yourself, that reinforce the importance of testing your docs. Testing is vital to writing good developer documentation. Never just take an engineer's word for how something works. If you follow this advice and test all your docs, you'll be successful in the API doc field. But if you just transcribe what engineers tell you, you'll basically end up being an engineer's secretary. (For more, see my blog post How to avoid being a secretary for engineers ↗.)

Going through the whole process

In addition to testing individual endpoints and other features, it's also important to go through the whole user workflow from beginning to end.

While working at one company, it wasn't until I built my own app and submitted it to the Appstore that I discovered some bugs. I was documenting an app template designed for third-party Android developers building streaming media apps for the Amazon Appstore. To get a better understanding of the developer's tasks and process, I needed to be familiar with the steps I was asking developers to do. For me, that meant building an app and submitting my app to the Appstore — the whole workflow from beginning to end.

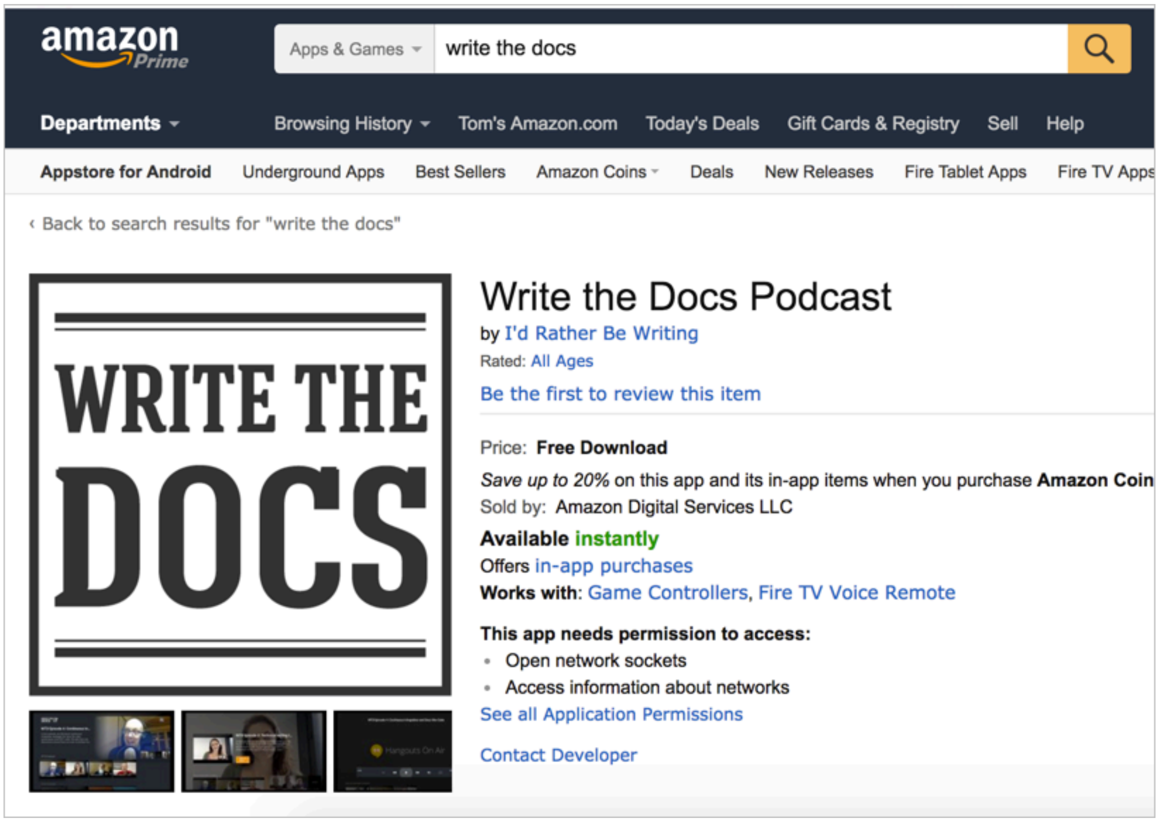
To build my sample app, first I had to figure out how to get content for my app. I decided to take the video recordings of podcasts that we had through the Write the Docs podcast and use that media for the app.

Since the app template didn't support YouTube as a web host, I downloaded the MP4s from YouTube and uploaded them directly to my web host. Then I needed to construct the media feed that I would use to integrate with the app template. The app template could read all the media from a feed by targeting it with Jayway Jsonpath or XPath expression syntax.

I used Jekyll to build my feed. (You can view my JSON-based feed at podcast.writethedocs.org/fab.json .) The most challenging part in setting up this feed was configuring the `recommendations` object. Each video has some `tags`. The `recommendations` object needed to show other videos that have the same tag. Getting the JSON valid there was challenging, and I relied on some support from the Jekyll forum.

After I had the media and the feed, integrating it into Fire App Builder was easy because, after all, I had written the documentation for that.

Submitting the app into the Appstore was fun and illuminated parts of the developer's workflow that I hadn't previously understood. You can view the Write the Docs podcast app in the Amazon Appstore website here.



Write the Docs Podcast

by I'd Rather Be Writing

Rated: All Ages

Be the first to review this item

Price: **Free Download**

Save up to 20% on this app and its in-app items when you purchase **Amazon Coin**

Sold by: Amazon Digital Services LLC

Available instantly

Offers in-app purchases

Works with: Game Controllers, Fire TV Voice Remote

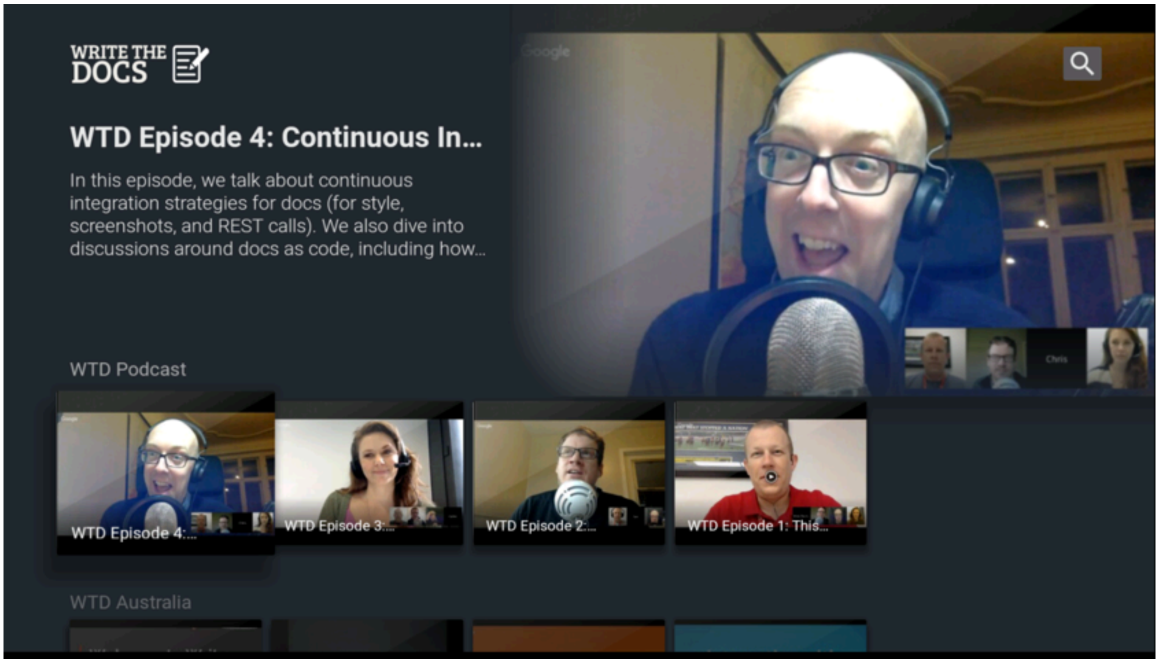
This app needs permission to access:

- Open network sockets
- Access information about networks

See all Application Permissions

Contact Developer

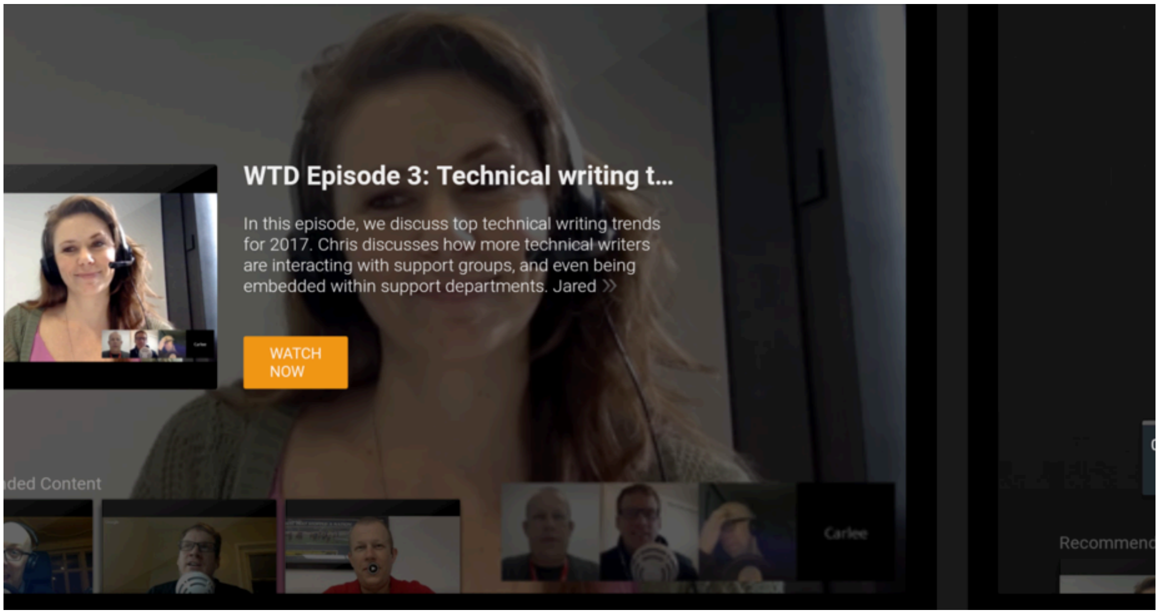
Here's what the app screens look like on your Fire TV:



WTD Episode 4: Continuous In...

In this episode, we talk about continuous integration strategies for docs (for style, screenshots, and REST calls). We also dive into discussions around docs as code, including how...

When you select a video, you see a video preview screen:



WTD Episode 3: Technical writing t...

In this episode, we discuss top technical writing trends for 2017. Chris discusses how more technical writers are interacting with support groups, and even being embedded within support departments. Jared »

All seemed to go well, but then I discovered some bugs that I would not have discovered had I not actually submitted the app into the Appstore. First, I found that device targeting (listing certain features in your Android manifest to identify which Fire devices your app supports) didn't work correctly for Fire TV apps. (This issue wasn't directly related to the app template, though.)

I also discovered other issues. Although developers had tested the app template for many months, they hadn't tested pushing apps into the Appstore with the app template. It turns out the template's in-app purchases component (not active or configured by default) automatically triggered the Appstore to add a tag indicating that the app contained in-app purchases.

This in-app-purchasing tag surprised the dev team, and it would have caused a lot of issues if all apps that third-party developers were building showed this tag.

The developers said users could simply deregister the component from the app. So I modified the doc to indicate this. Then I tried deregistering the component from the app and submitted a new version, but the in-app-purchases tag issue persisted.

This experience reinforced to me how vital it is to get your hands on the code you're documenting and run it through as real of a situation as you can. It blew my mind that the engineering team did not actually have an app in the Appstore that they published with this template. I was the only one.

It's not always possible to run code through real situations, and there are times when I might limit my role to editing and publishing only, but that's not the scenario I prefer to work in. I love getting my hands on the code, trying to make it work in the scenario it was designed for. Really, how else can you write good documentation?

The team also asserted that the same app could be submitted into the Google Play Appstore. However, this was an untested assumption. When I submitted my app into the Play Store, Google rejected it due to missing banner assets declared in the manifest. It also triggered "dangerous permission" warnings. I relayed the information to engineers, who created JIRA tickets to address the issues. More than just creating better documentation, this testing allowed me to improve the products I was documenting. (It also improved my credibility with the engineers.)

This same type of scenario repeats itself frequently with other projects as well. Another engineering team I was supporting also developed an app template (web instead of Android) for publishing apps in the Appstore. This tool was designed for non-technical end users and was supposed to be easy. The project team hadn't even planned any documentation except for a brief FAQ.

I tested the tool from beginning to end by creating and submitting an app with it. By the time I finished, I had more than 30 questions along with several significant issues that I had discovered. I uncovered many previously unknown bugs, called attention to a problematic synchronization issue, brought together teams from across organizations to troubleshoot some issues, and generally raised my value from mere documentation writer to more of a power player on multiple teams.

Empowered to test additional features

Testing documentation for developers is difficult because we often just provide reference APIs for users to integrate into their own apps. We assume that developers already have apps, and so all they need is the API integration information. But many times you can't know what issues the API has until you integrate it into a sample app and use the API in a full scenario from beginning to end.

For example, for general Fire TV users who weren't using the app template, I also wrote documentation on how to integrate and send recommendations. But since I didn't have my own general Fire TV app (not one built with Fire App Builder) to test this with, I didn't play around with the code to send recommendations. I had to take on faith much of my information based on the engineer's instructions and the feedback we were getting from beta users.

As you can imagine, I later discovered gaps in the documentation that I needed to address. It turns out when you, in fact, send recommendations to the Fire TV home screen, Fire TV uses only *some* of the recommendations information that you submit. But in my original docs, I didn't indicate which fields actually get used. This lack of information left developers wondering if they integrated the recommendations correctly. Unsurprisingly, in our forums, a third-party developer soon asked what he was doing wrong because a field he was passing seemed to be ignored in the display.

Putting together an app from scratch that leverages all the recommendation API calls requires more effort, for sure. But to write better documentation, it's the step I needed to take to ferret out all the potential issues users would face. If creating the sample app is beyond your skill level, ask the engineers for a demo app or to schedule a meeting where they will demonstrate the feature in a live way.

Overall, make sure to test the code you're documenting in as real of a situation as you can. You'll be surprised what you discover. Reporting back the issues to your team will make your product stronger and increase your value to the team.

The pleasures of testing

Testing your instructions makes the tech writing career a lot more engaging. I'd even say that testing all the docs is what converts tech writing from a dull, semi-isolated career to an engaging, interactive role with your team and users.

There's nothing worse than ending up as a secretary for engineers, where your main task is to record what engineers say, write up notes, send it to them for review, and then listen to their every word as if they're emperors who give you a thumbs up or thumbs down. That's not the kind of technical writing work that motivates me.

Instead, when I can walk through the instructions myself, and confirm whether they work or not, adjusting them with more clarity or detail as needed, that's when things become interesting. (And actually, the more I learn about the knowledge domain itself — the technology, product landscape, business, and industry, etc. — the appeal of technical writing increases dramatically.)

In contrast, if you just stick to technical editing, formatting, publishing, and curating, these activities will likely not fulfill you in your technical writing career (even though these activities are still worthwhile). Only when you get your synapses firing in the knowledge domain you're writing in, as well as get your hands dirty testing and trying out all the steps and processes, does the work of technical writing come alive.

Accounting for the necessary time

Note that it takes time to try out the instructions yourself and with users. It probably doubles or triples the documentation time. Writing thorough, accurate instructions that address users with different setups, computers, and goals is tedious. You don't always have this time before release.

But don't assume that once your product is released, all documentation is done. You can always go back over your existing, already-published documentation and improve it. Consider the first release a kind of "Day 1" for your documentation. It's the first iteration. Your documentation will get better with each iteration. If you couldn't get your test system up and running before the first release, that's okay. Build the test system for the upcoming release.

With the first release, if you can capture feedback as your documentation get used (feedback from forums, contact email, logs, and other means), you can improve your documentation and see gaps that you likely missed. In some ways, each time users consult the documentation to perform a task, they are testing your documentation. (For more on capturing feedback, see my post on Reconstructing the absent user ↗.)

Beyond just testing documentation yourself, you also need to test it against users (p. 353).

Test your assumptions against users

The previous two sections talked about testing from the perspective of the tech writer merely running through the steps. However, remember that you, the tech writer, are not the user. Almost all documentation builds on assumptions (about capabilities, setup, previous knowledge) that may or may not be shared with your audience. While testing your documentation (p. 348), recognize that what may seem clear to you may be confusing to your users. And what might be confusing to you might be clear to users. Learn to identify assumptions that can interfere with your audience's ability to follow the instructions in your documentation.

Assumptions about terminology

You might assume that your audience already knows how to SSH onto a server, create authorizations in REST headers (p. 393), use curl to submit calls (p. 80), and so on. Usually documentation doesn't hold a user's hand from beginning to end, but rather jumps into a specific task that depends on concepts and techniques that you assume the user already knows. However, making assumptions about concepts and techniques your audience knows can be dangerous. These assumptions are exactly why so many people get frustrated by instructions.

For example, my 10-year-old daughter is starting to cook. She feels confident that if the cookbook's instructions are clear, she can follow almost anything (assuming we have the ingredients to make it). However, she says sometimes the instructions tell her to do something that she doesn't know how to do — such as *sauté* something.

To *sauté* an onion, you cook onions in butter until they turn soft and translucent. To *julienne* a carrot, you cut them in the shape of little fingers. To *grease* a pan, you spray it with Pam or smear it with butter. To add an egg *white* only, you use the shell to separate out the yolk. To *dice* a pepper, you chop it into little tiny pieces.

The terms can all be confusing if you haven't done much cooking. Sometimes you must *knead* bread, or *cut* butter, or *fold in* flour, or add a *pinch* of salt, or add a cup of *packed* brown sugar, or add some *confectioners* sugar, or bring liquid to a *roiling boil*, and so on.

Sure, these terms are cooking 101, but if you're 10-years-old and baking for the first time, this is a world of new terminology. Even measuring a cup of flour is difficult — does it have to be *exact*, and if so, how do you get it exact? You could use the flat edge of a knife to knock off the top peaks, but someone has to teach you how to do that. When my 10-year-old first started measuring flour, she went to great lengths to get it exactly 1 cup, as if the success of the entire recipe depended on it.

The world of software instruction is full of similarly confusing terminology. For the most part, you have to know your audience's general level so that you can assess whether something will be clear. Does a user know how to *clear their cache*, or update *Flash*, or ensure the *JDK* is installed, or *clone* or *fork* a git repository? Do the users know how to open a *terminal*, *deploy* a web app, import a *package*, *cd* on the command line, submit a *PR*, or *chmod* file permissions?

This is why checking over your own instructions by walking through the steps yourself becomes problematic. The first rule of usability is to know the user, and also to recognize that you aren't the user.

With developer documentation, usually the audience's skill level is beyond my own, so adding little notes that clarify obvious instruction (such as saying that the `$` in code samples signals a command prompt and shouldn't be typed in the actual command, or that ellipses `...` in code blocks indicates truncated code and shouldn't be copied and pasted) isn't essential. But adding these notes can't hurt, especially when some users of the documentation are product marketers rather than developers.

We must also remember that users may have deep knowledge in another technical area outside of the domain we're writing in. For example, the user may be a Java expert but a novice when it comes to JavaScript, and vice versa.

To read more about how specialized language makes technical documentation difficult to understand, see Reducing the complexity of technical language in my series on Simplifying Complexity.

Solutions for addressing different audiences

The solution to addressing different audiences doesn't involve writing entirely different sets of documentation (although conceptually, that might be a good strategy in some situations). You can link potentially unfamiliar terms to a glossary (p. 418) or getting started tutorial (p. 379) where beginners can ramp up on the basics.

You can likewise provide links to separate, advanced topics for those scenarios when you want to give some power-level instruction but don't want to hold a user's hand through the whole process. You don't have to offer just one path through the doc set.

The problem, though, is learning to see the blind spots. If you're the only one testing your instructions, the instructions might seem perfectly clear to you. (I think most developers also feel this way after they write something; they usually take the approach of rendering the instruction in the most concise way possible, assuming their audience knows exactly what they do.) But the audience *doesn't* know exactly what you know, and although you might feel like what you've written is crystal clear, because c'mon, everyone knows how to clear their cache, in reality you won't know until you **test your instructions against an audience**.

Testing your docs against an audience

Almost no developer can push out their code without running it through QA, but for some reason technical writers usually don't follow the same QA processes with their documentation. There are some cases where tech docs are "tested" by QA, but in my experience, when I do get feedback from QA, the reviewers rarely assess aspects of clarity, organization, or communication. They just highlight any errors they find (which is still helpful on some level).

In general, QA people don't test whether a user would understand the instructions or whether concepts are clear. They just look for accuracy. QA team members are also poor testers because they already know the system too well in the first place.

Before publishing, it's ideal to submit your documentation through a testing process of some kind, that is, a "quality assurance" process. Companies wouldn't dream of setting up an IT shop without a quality assurance group for developers — why should docs be any different?

When there are editors for a team, the editors usually play a style-only role, checking to make sure the content conforms to a consistent voice, grammar, and diction in line with the company's official style guide.

While conforming to the same style guide is important, it's not as important as having someone actually test the instructions. Users can overlook poor speech and grammar — blogs and YouTube are proof of that. But users can't overlook instructions that don't work, that don't accurately describe the real steps and challenges the user faces.

I haven't had an editor for years. In fact, the only time I've ever had an editor was at my first tech writing job, where we had a dozen writers. The editor focused mostly on style.

I remember one time our editor was on vacation, and I was asked to fill in as editor during her absence. As interim editor, I tried testing out the instructions and found that about a quarter of the time, I got lost. The instructions either missed a step, needed a screenshot, built on assumptions I didn't know, or had other issues.

The response, when you give instructions back to the writer, is usually, "Oh, users will know that." The problem is that we're usually so disconnected with the actual user experience (since we rarely see users trying out docs), we can't recognize the "users-will-know" statement for the fallacy that it is.

Using your colleagues as test subjects

If you have access to real users who can test your docs, great. Take advantage of this situation. But if you don't, how do you test instructions without a dedicated editor, without a group of users, and without any formal structure in place? At the very least, you can ask a colleague or product team member to try out the instructions.

Other technical writers are usually both curious and helpful when you ask them to try out your instructions. And when other technical writers start to walk through your steps, they recognize discrepancies in style that are worthy of discussion in themselves.

Although other technical writers don't always have time to go through your instructions, and they usually share your same level of technical expertise, having *someone* test your instructions is better than no one.

Tech writers are good testing candidates precisely because they are writers instead of developers. As writers, they usually lack the technical assumptions that a lot of developers have (assumptions that can cripple documentation). Also, developers may feel embarrassed if they don't already understand a concept or process referenced in the docs.

Tech writers who test your instructions know exactly the kind of feedback you're looking for. They won't feel ashamed and dumb if they get stuck and can't follow your instructions. They'll usually let you know where your instructions are poor. They might say, *I got confused right here because I couldn't find the X function, and I didn't know what Y meant.* They know what you need to hear.

In general, it's always good to have a non-expert test something rather than an expert. Experts can often compensate for shortcomings in documentation by relying on their own expertise. In fact, experts may pride themselves in being able to complete a task *despite the poor instruction*. Novices can't compensate.

Also, when you have other technical writers test your docs, the activity fosters good team building and knowledge sharing. At a previous job, I worked in a large department that had, at one time, about 30 UX engineers. The UX team held periodic meetings during which they submitted a design for general feedback and discussion. The camaraderie and team-building during the design reviews was immediately apparent to me, and I wished we had something similar with our tech pubs team.

By giving other technical writers the opportunity to test your documentation, you create the same kind of sharing and review of content. You build a community rather than having each technical writer always work on independent projects.

The outcomes from having colleagues test your docs might include more than just feedback about the shortcomings in a specific doc set. You might bring up matters of style, or you might foster great team discussions about innovative approaches to your help. Maybe you've integrated a glossary tooltip that is simply cool, or a button for an embedded series. When other writers test your instructions, they not only see your demo, they understand how helpful a feature is in a real context, and they can incorporate similar techniques.

Observing users as they test your docs

One question in testing docs with your users is whether you should watch them in test mode. Undeniably, when you watch users, you put some pressure on them. Users don't want to look incompetent or dumb when they're following what should be clear instructions.

But if you don't watch users, the whole testing process is much more nebulous. Exactly *when* is a user trying out the instructions? How much time are they spending on the tasks? Are they asking others for help, googling terms, and going through a process of trial and error to arrive at the right answer?

If you watch a user, you can see exactly where they're getting stuck. Usability experts prefer to have users actually share their thoughts in a running monologue. They tell users to let them know what's running through their head every now and then.

In other usability setups, you can turn on a web cam to capture the user's expression while you view the screen in an online meeting screenshare. This can allow you to give the user some privacy while also watching him or her directly.

Agile testing

In my documentation projects, I admit that I don't do nearly as much user testing as I should. At some point in my career, someone talked me into the idea of "agile testing." When you release your documentation, you're basically submitting it for testing. Each time you get a question from users, or a support incident gets logged, or someone sends an email about the doc, you can consider that feedback about the documentation. (*And if you don't hear anything, then the doc must be great, right? Hmmm.*)

Agile testing methods are okay. You should definitely act on this feedback. But hopefully you can catch errors *before* they get to users. The whole point of a quality assurance process is to ensure users get a quality product prior to release.

Additionally, the later that errors are identified in the software development process, the more costly they are. For example, suppose you discover that a button or label or error message is really confusing. It's much harder to change it post-release rather than pre-release. Changing a button might require the app to undergo retesting, error messages to be adjusted, retraining, UI modifications, and more. Changes to APIs are even worse, because presumably third-party developers have already integrated the calls into their apps. If you change the endpoint or parameters, the team might have to support both the old and new API, which then doubles the testing efforts.

Conclusion

No matter how extensively or minimally you do it, look for opportunities to test your instructions against an actual audience. You don't need to do a lot of tests (even the usability pros say 4-5 test subjects is usually enough to identify 80% of the problems), but try to do *some user testing*. When you treat docs like code (p. 551), it naturally follows that just as we should test code, we should also test docs.

Activity: Test the docs in your open-source project

Now that you've read about testing, it's time to get some more hands-on practice. In this activity, you'll do some doc testing with an open-source project (or another project you identified).

Test a topic

1. With the open-source API project you're working with (p. 686), find the following:
   - Getting started tutorial (or similar tutorial)
   - API endpoint
2. Now test the documentation by walking through the documentation details. More specifically:
   - For the Getting started tutorial, proceed through all the steps in the tutorial. Make a note of any assumptions that you found confusing, new terms or acronyms that were unfamiliar, or other questions you had. How long did it take to complete the getting started tutorial?
   - For the API endpoint, make a request and examine the response. Does the response match what's in the documentation? Try different variations of parameters in the endpoint and see if the responses are what you expect. Identify any incorrect or missing or inaccurate information.
3. If you found information that was incorrect, either log an issue or reach out to the QA lead with the feedback.

Find out test details

Identify who performs the testing on the project. Reach out and interact with the QA lead for the project to gather as much information as you can about how testing is done.

Find answers to the following questions:

- Are there test cases used to run through various scenarios in the project?
- Where are the test cases stored?
- How are the tests executed? Automatically? Manually?
- What kind of testing does the team do before a release?
- If you encounter a bug while testing, how should you report it?

Chapter 7: Conceptual topics in API docs

While reference topics in APIs generally receive the most attention, the conceptual topics, such as getting started tutorials, information about authorization, rate limiting, status and error codes, quick reference guides, and other topics constitute about half of the documentation. These topics are usually handled by technical writers more than engineers. You can evaluate the quality of API documentation in part by looking to see whether it includes these conceptual topics.

Chapter sections:

API conceptual topics overview 359
API product overviews 360
API getting started tutorials 379
API authentication and authorization 393
API status and error codes 402
API rate limiting and thresholds 408
API quick reference 412
API glossary 418
API best practices 426
Activity: Complete the SendGrid Getting Started tutorial 429
Activity: Assess the conceptual content in your project 430

Conceptual topics overview

Up until this point, we've mostly been focusing on the reference aspects of API documentation (p. 114) (the endpoints). The reference documentation is only one aspect of API documentation.

In this section, I'll cover the main conceptual topics that are commonly found in API documentation. Rather than "conceptual topics," you might consider this type of information the "user guide."

The following are common conceptual topics commonly found in API documentation:

- API product overview (p. 360)
- Getting started (p. 379)
- Authentication and authorization requirements (p. 393)
- Status and error codes (p. 402)
- Rate limiting and thresholds (p. 408)
- Quick reference guide (p. 412)
- API glossary (p. 418)
- API best practices (p. 426)

Beyond the sections outlined above, you might want to include other tasks and tutorials specific to your API, based on what you expect your users to do and the business scenarios for which they'll use your API.

In each concept topic, I'll provide general descriptions and overviews of what these sections contain, followed by examples from actual API documentation sites.

With each topic, there's an activity for you to go into your open-source project (p. 686) and evaluate the conceptual topic at hand. Although many of the conceptual topics are straightforward, when you look at the information in context, that is, when you look to see how the information is actually implemented, it gets a lot more interesting.

API product overviews

The product overview tells your product's story at a high level, including what you can do with the product, the market needs or pain points it solves, requirements to use it, who the product or other features are for, and other introductory information. A company with multiple products will have distinct product overview pages for each product, with a more general umbrella overview for them all. In contrast, smaller companies with fewer products might have a single, consolidated product overview page for everything the company offers.

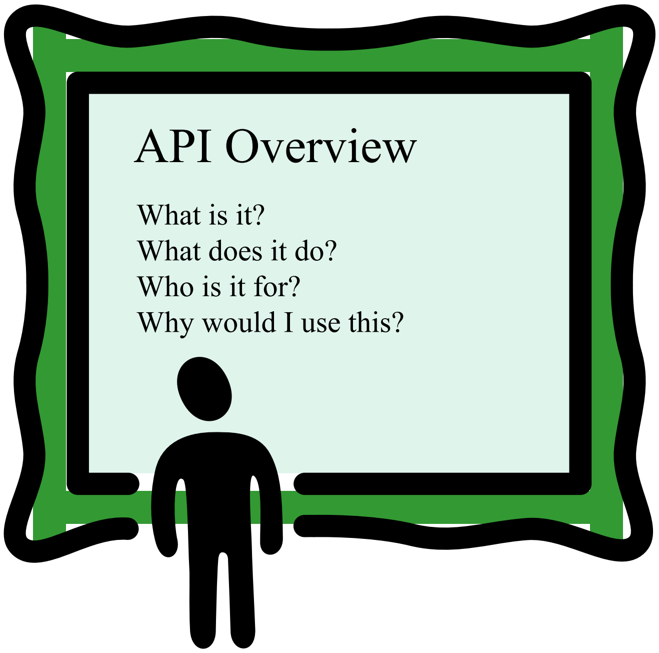
Although a seemingly simple page, the product overview page can overlap into marketing domains, create redundancies with README's, and pose challenges in connecting with a more diverse audience (both engineers and bizdev people) than the rest of your technical docs. Overall, the product overview is an area where documentation and marketing intersect in interesting ways. The product overview is one of the hardest topics to write, but it's also likely the most important.

Key questions a product overview should answer

Too often with developer documentation, the documentation gets quickly mired in technical details without ever explaining clearly what the product is used for. It's easy for writers to lose sight of the overall purpose and business goals of the API by getting lost in the endpoints and other technical details. Unfortunately, many documentation sites never seem to explain the story of their product, thus missing out on a foundational aspect of the documentation. The product overview should let users get a good understanding of the following:

- What does the product do?
- what are some examples where you'd use it?
- Who is it for?
- What problem does the product solve?
- How do you get started?

These are essentially who/what/when/where/why/how questions — not rocket science here, just the basic fundamentals of expository writing.



The product overview provides high-level detail about the purpose, audience, and market need for your API.

Keep in mind that there are thousands of APIs. If people are browsing your API, their first and most pressing question is, what information does it provide? Is this information relevant and useful to my needs? How does it differ from other products in this same space? The user's first question is usually not "How do I configure this endpoint."

Because the product overview is really one of the few places where you can tell a story, the product overview space should appeal to tech writers and be one of the content areas where we excel.

Presentation on product overviews

I recently gave a presentation on getting started tutorials (p. 379) and product overviews. You can watch the presentation here:

Telling your product’s story

To tell your product’s story, consider identifying a market need that your product solves. This is the basics of storytelling — there is some conflict that a protagonist (in this case, your product) addresses and solves. In The Top 20 Reasons Startups Fail, one of the main reasons startups fail is their inability to solve a market problem. The authors explain:

> Startups fail when they are not solving a market problem. We were not solving a large enough problem that we could universally serve with a scalable solution. We had great technology, great data on shopping behavior, great reputation as a thought leader, great expertise, great advisors, etc, but what we didn’t have was technology or business model that solved a pain point in a scalable way. (*CB Insights*)

To encapsulate this overarching story, the product overview focuses on the market problem that the product solves. If the product doesn’t solve a market problem, that could be a red flag about the product itself. So the first step in the product overview is to describe the pain point your product solves.

Common use cases

To make the product’s market-solving characteristics more concrete, list some common use cases or business scenarios in which the product and API are relevant. These scenarios will give users the context they need to evaluate whether the API is relevant to their needs. Too often, product descriptions are general and high-level (e.g., “X product helps companies collaborate more effectively…”). These higher-level abstract descriptions fail to resonate with users.

Use cases are concrete examples of how the product might be used. Continuing with the above example, a use case might be “X product allows writers to work on the same document simultaneously through a remote browser interface,” or something. Usually, a product manager has already defined a list of key use cases for a product and would have these available.

Product overview vs overviews (plural)

A developer portal usually has documentation for many different products, not just one. Each product will have its own product overview page. In fact, the homepage of the developer portal rarely has a product overview. Instead, the developer portal's homepage often serves as a routing board to the product overviews and other developer journeys.

Even a single product might have multiple overviews for each of the features. The overview is just a term for the opening page of a product, the landing page or starting page. Whenever you need a high-level description of your product, you need an overview page.

Audience includes decision-makers

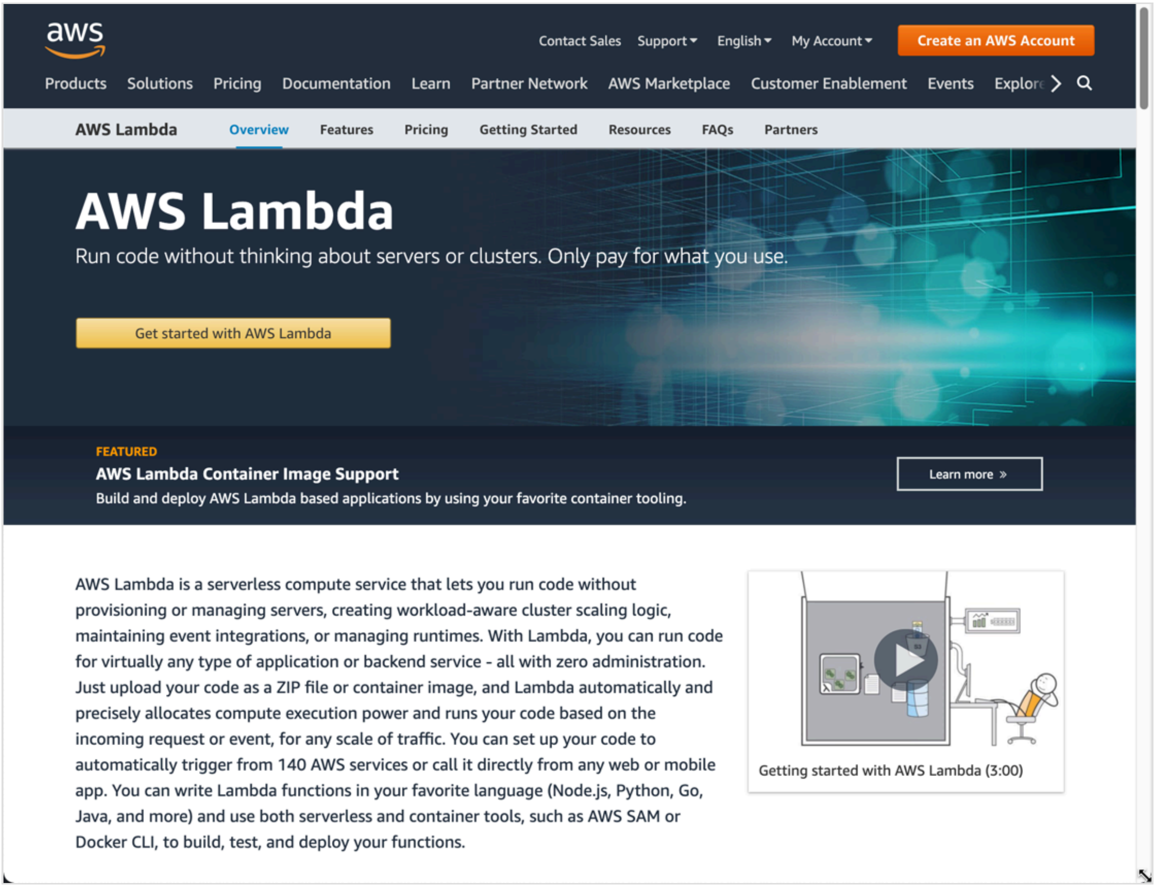
One important dimension to keep in mind with product overviews is the expanded audience. Product overviews aren't just read by your target documentation users, i.e., usually engineers. Product overviews are frequently read by product managers, executives, and other decision-makers who are trying to decide whether to purchase or move forward with the product. These decision-makers might be trying to size up the complexity of integration, how many person-hours the effort will consume, and whether the product will solve the problem they have. Only documentation can truly answer this question, not marketing material.

For example, a high-level executive might be trying to decide if implementing your product will require one-week integration effort by a single developer or a team of 50 developers working heads-down for six months. The product overview should give some indication of the development effort. Even if you don't call out the estimated development time, by browsing through the tasks in the docs, it should become clear what level of effort is involved in implementing the product.

In the overview, the high-level executive will be less interested in the technical details and more interested in conceptual and bigger picture content. List out the main components involved in the system, followed by an architectural diagram and an explanation. Save the excruciating technical details for the inner pages of the documentation.

Overlap with marketing

Another facet of product overviews is their frequent overlap with marketing content. In many organizations, there is a developer marketing group that handles higher-level product content, creating some overlap with documentation-based product overviews. For example, if you browse the AWS Lambda documentation, you'll find that a higher-level product overview appears before the actual documentation. Here's the marketing layer ↗ to the product:



AWS Lambda

Run code without thinking about servers or clusters. Only pay for what you use.

FEATURED

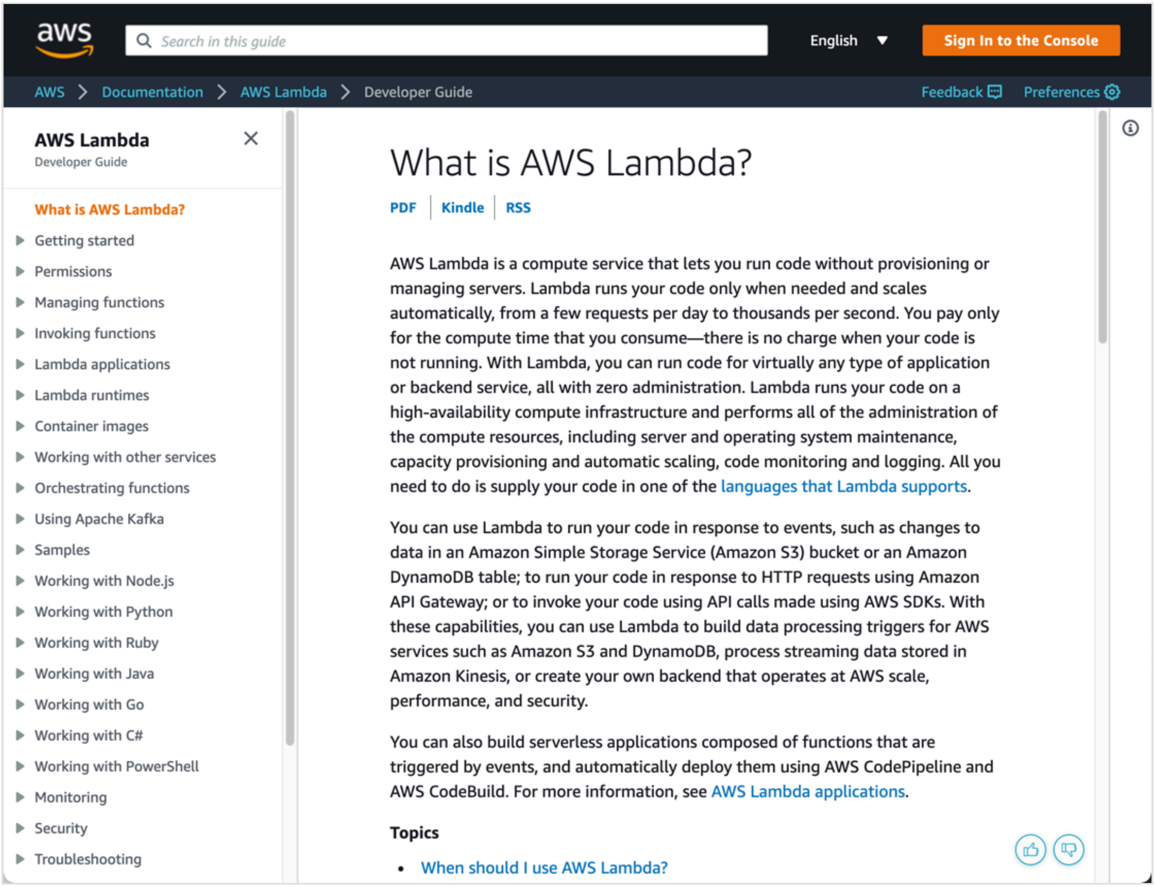
AWS Lambda Container Image Support

Build and deploy AWS Lambda based applications by using your favorite container tooling.

AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers, creating workload-aware cluster scaling logic, maintaining event integrations, or managing runtimes. With Lambda, you can run code for virtually any type of application or backend service - all with zero administration. Just upload your code as a ZIP file or container image, and Lambda automatically and precisely allocates compute execution power and runs your code based on the incoming request or event, for any scale of traffic. You can set up your code to automatically trigger from 140 AWS services or call it directly from any web or mobile app. You can write Lambda functions in your favorite language (Node.js, Python, Go, Java, and more) and use both serverless and container tools, such as AWS SAM or Docker CLI, to build, test, and deploy your functions.

Getting started with AWS Lambda (3:00)

This marketing layer covers these topics: Overview, Features, Pricing, Getting Started, Resources, FAQs, and Partners. The actual Lambda documentation is presented on another layer of the site:



What is AWS Lambda?

AWS Lambda is a compute service that lets you run code without provisioning or managing servers. Lambda runs your code only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time that you consume—there is no charge when your code is not running. With Lambda, you can run code for virtually any type of application or backend service, all with zero administration. Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. All you need to do is supply your code in one of the languages that Lambda supports.

You can use Lambda to run your code in response to events, such as changes to data in an Amazon Simple Storage Service (Amazon S3) bucket or an Amazon DynamoDB table; to run your code in response to HTTP requests using Amazon API Gateway; or to invoke your code using API calls made using AWS SDKs. With these capabilities, you can use Lambda to build data processing triggers for AWS services such as Amazon S3 and DynamoDB, process streaming data stored in Amazon Kinesis, or create your own backend that operates at AWS scale, performance, and security.

You can also build serverless applications composed of functions that are triggered by events, and automatically deploy them using AWS CodePipeline and AWS CodeBuild. For more information, see AWS Lambda applications.

Topics

- When should I use AWS Lambda?

If you read the first paragraph from each screenshot, you'll see how similar yet different the two descriptions are. They repeat many of the same points but in different ways. The documentation product overview addresses these questions:

- When should I use AWS Lambda?
- Are you a first-time user of AWS Lambda?

Beyond the product description, in this Lambda example, the marketing content even has its own Getting Started page with tutorials, parallel to the documentation's Getting started section, which is more robust.

Typically, developer marketing teams write the marketing product overview, while developer documentation teams write the documentation product overview. But as you can see, this is an area where content overlaps and where some coordination across teams becomes essential. Suppose the doc team for Lambda wanted to emphasize certain points that the marketing team did not — it would create a confusing transition between the two sets of content.

But even with differences, the idea is that business decision-makers read the marketing content, while engineers read the documentation content. Marketers are primarily writing for these decision-makers while tech writers are primarily addressing the implementers (engineers).

Your organization might have multiple teams writing content like this, or you might be tasked with creating both the higher-level marketing layer and the documentation yourself (especially in startups). In some ways, having a single team or writer handle both types of content might lead to a more streamlined, unified content experience. When you’re the sole writer, you’re less likely to repeat yourself in different places in contradictory ways. You can simply devote a section of your documentation to the marketing content rather than housing it on another site.

If you are stuck with the two-site model (marketing on one site, documentation on another), you could try to share content between these two sites, but usually marketing has a different system for managing and publishing content than the documentation teams. Marketers don’t usually adopt docs-as-code systems but rather prefer more CMS-driven systems. These systems rarely share content with each other, and even if they did, the marketing versions might be written using another style, perspective, or approach that contrasts with your docs, making it difficult to single-source the content. For example, I once tried to re-use marketing content (a product brief) that was written entirely in third-person point of view (“the partner does X”) rather than the traditional second-person point of view in docs (“you do X”). It didn’t work out well.

Strategies for the documentation’s product overview

What strategies should you implement when you’re faced with writing a product overview for docs, especially when a product marketing team has their own product overview on a separate site? Consider the following:

- Find out who the marketing group is and what messaging they are focusing on for the product.
- If marketing content already exists and you want to leverage it (rather than just link to it), consider creating a condensed/streamlined version of the marketing overview content, and point users to the marketing overview for more details. You don’t want to duplicate all the content because invariably, docs will go out of date as the other content evolves (not to mention the confusion of presenting two sets of overviews to users).
- Avoid copying any overblown promises about simplicity or ease of installation from marketing copy.

Differences between marketing and documentation content

Here it’s worth diving into some differences between documentation copy and marketing copy. While both genres might appear to share similar purposes in the product overview, avoid falling into marketing style in docs. For example, suppose you find a few pages of product descriptions that the marketing team already wrote, and you want to just copy it into the docs for the documentation product overview. Should you?

If you do this, strip out mention of the word “easy” or “just,” as in “the implementation is so easy, you *just* have to do X….” To sell a product, marketing often gravitates toward promises about ease of implementation. This is perhaps the hallmark of marketing content (from a tech writer’s perspective anyway). And many bizdevs or execs are trying to scope the difficulty of the implementation, so marketing’s message about ease of implementation makes sense.

But as a technical writer, you not only have an obligation to be honest about implementation complexity, you must also recognize that what is easy for one user might be insurmountable to another. (If you’ve ever done DIY projects at home, you know what I’m talking about.)

Never say something is “easy” — instead, you might qualify the degree of development effort based on the role. For example, you could say that for a seasoned engineer familiar with Java and who has been developing cloud-based apps for years, implementing this product will likely not require more than a week of integration effort. But for someone new to cloud-based app development or less familiar with Java, there will be a much steeper learning curve and might require several months or more of preparation and learning to implement.

If you can qualify the level of development effort based on different audience types, this will provide more realistic information. You can still answer the exec's question — how difficult is this product to implement — without falling prey to promises of implementation ease.

By the way, it's worth noting that most marketers have a superficial technical understanding of a product, so they usually cannot make judgments about the implementation difficulty anyway. They might be going off of an internal engineer's observation that it's "straightforward" or that it "should be easy to implement" or that "most engineers should find this familiar." What the marketer might not realize is that engineers usually make these estimations by assuming the audience has the same knowledge level and background as the engineer. Unfortunately, most marketers remain within the pre-sales context and so rarely see the post-sales realities, where many support cases and threads spring up with confused and frustrated users.

Beyond adjectives about easiness, in the previous Lambda examples, the marketing copy uses terms like "virtually any," "automatically," "precisely," and "favorite." These superlatives aren't usually used in documentation, which tends to be more factual and plain-speaking. Marketing tries to get users excited about a product by embracing these extreme adjectives, while documentation isn't overtly trying to sell or hype anything.

Key differentiators in product overviews

As I've been arguing, the product overview space places you into murky territory where marketing and documentation blend. If you were to put on a marketing hat for a moment, what angle would you take in your writing (beyond language)?

Although it would be awesome to compare your product against competitor products, most likely your legal group will not allow it (mentioning competitors is usually taboo). And you might not have a deep understanding of other products to make a fair comparison. Or you might feel that readers will assume you're too biased and wouldn't trust your comparison anyway.

But what you can do is focus on your product's key differentiators. These are features your product has that make it unique in the market. For example, maybe users can access your app from the browser rather than installing it locally. You don't need to create a comparison chart showing how products X, Y, and Z lack online browser access. But by emphasizing this differentiating feature, you help establish a selling point and a potential reason for buying the product.

Remember that the product overview, unlike other documentation, often addresses a *pre-sales* scenario. As such, the reader is likely wondering how your product compares with other products in the same category. Why should they go with your product rather than another? What advantage does your product provide? Unless you know the competitive advantage of your product, you'll have a difficult time writing marketing-esque content in your product overview.

Then again, you might want to leave that topic alone entirely and just point users to marketing material. You will need to make a judgment call about where marketing ends and where documentation begins. If you do try to veer into the marketing domain, however, reading through competitive analysis docs from marketers could help inform your writing.

Overlap with README's

Another challenge with product overviews is the overlap with README content. A README is an introductory overview file (a homepage) placed in a code repository such as GitHub. The README often has many elements of an overview similar to a product overview in documentation site. If your documentation references a code repo, that code repo needs a README. But do you duplicate the same information in the README that you do in the documentation overview?

Hopefully not. The README might have a high-level summary and information about installation, configuration, and usage. But this information should be much more condensed/abbreviated than more detailed documentation.

Many guides about writing README content assume that the README is the only documentation for the code in the repo. As a professional technical writer, I rarely work on projects that are so small that the documentation can be handled by a single page that lives in a code repo. If that's all you need for your product, great. However, chances are the README is only a glimpse of many more pages of configuration, installation, and usage detail that live in a more robust documentation site separate from the repo. If that's the case, you might want to just link to your docs in the README.

Although the README and product overview overlap a bit, the README has some elements that don't necessarily belong in regular documentation. Content elements specific to the README in the code repo might be the following:

- Code of conduct
- Contributor how-to protocol
- Filing issues
- Pull requests
- License information
- Team details/contributors

See The Essential Sections for Better Documentation of Your README Project ↗ by Thomas Parisot for a good guide about writing README content.

I admit that my preferences for the README might deviate from general recommendations from developers in this space. I am not a fan of duplicating the same information from the documentation into a README. Instead, I prefer to provide brief summaries only in the README and then point users to the main documentation for more details. For example, you could provide 1-2 sentences for each of the main sections and then point users back to your main docs for details. As a rule of thumb, a README might be the length of a poem while your docs are the length of a novel.

README's have the additional complication of being difficult to maintain. Unless you have rights to publish to the code repository, it might be cumbersome to update the README content. If you're an engineer who is writing the code and docs at the same time, this likely isn't an issue. But in many organizations, technical writers are separate from engineering teams, and technical writers usually don't publish code to GitHub repos. I've published to GitHub repos in the past (in an effort to speed along the publication of a sample app, I jumped through the hoops of the company's approval process and pushed out the content into the repo), but later I regretted doing so. I learned that the person who pushes content into a repo owns that content and all the filed issues, pull requests, and other responsibilities that come with repo management. I didn't want to be in that position — I wanted the engineers to own and maintain the code and control pushes to this space.

Overall, README files shouldn't contain so many doc details that the information begins to conflict or become outdated with your main documentation. As long as you have only brief, condensed information in your README, it likely won't go out of date with each release.

Good Docs project template

If you're looking for more inspiration and guidance about product overviews, see the API overview template from the Good Docs project ↗. They recommend similar sections as those I've been recommending here — establish who the docs are for, what problems the product solves, what market/industry the product is intended for, and so on. In the body of the overview, the Good Docs team recommends covering the following questions:

> - What is it supposed to do? (What problem does it solve, and for whom?)
> - What exact capabilities are available to the user? What services does it offer?
> - What does it not do that developers should know about?
> - What are the typical use cases?
> - How does it work? (What do users need to know about architecture and internal components?)
> - What dependencies does the developer need to know about before installing?
> - What technical requirements do readers need? Include development environment and licensing requirements.
> - What knowledge prerequisites does the developer need to know about before using the API?
>
> (See The overview ↗)

This is all good information to include. Consider auditing your overview by asking each of these questions. Does your product overview provide answers? If not, add a section that answers the question.

Sample structure of a product overview

Product overviews vary from product to product, but here's the general flow that I like to follow:

- Description of the product
- Intended audience and assumptions about knowledge
- Sample use cases
- Requirements to use the product
- List of components
- High-level architectural diagram of components + explanation
- Development effort/scope
- How to get support/help
- Link to getting started tutorial

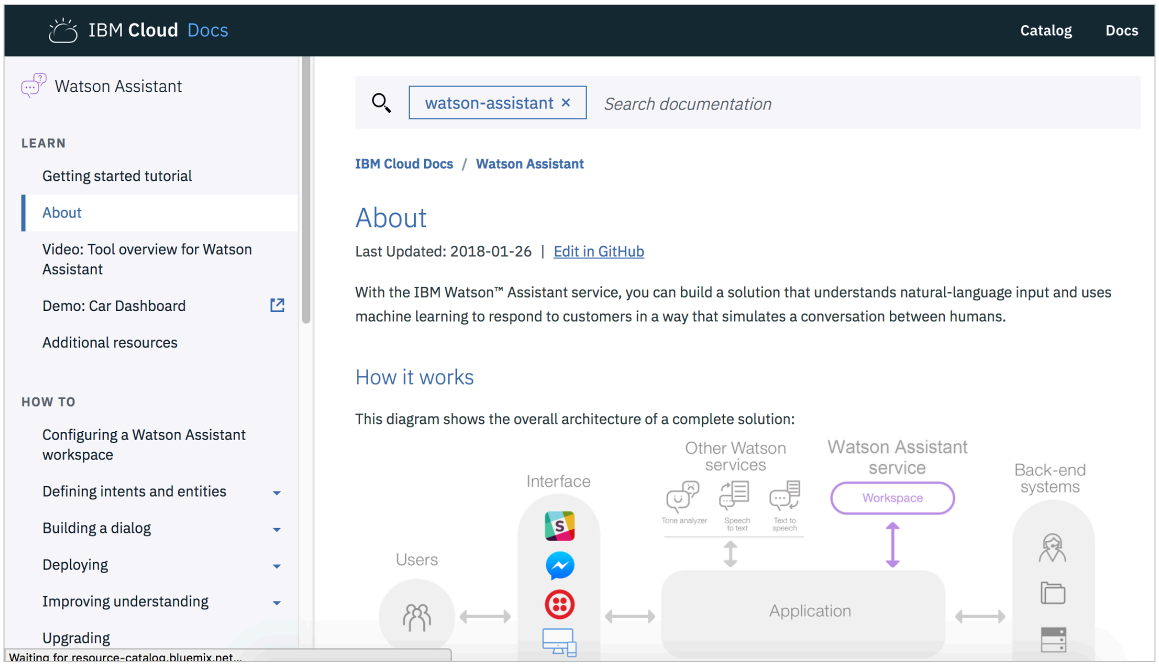
These topics don't need to be standalone sections but can be interwoven into similar sections as you see fit.

At the end of the product overview, be sure to transition into the next logical step: getting started! Here's where your getting started tutorial (p. 379) gets handed off to the user. It's your call to action, so to speak.

Sample product overviews

Here are a few sample product overviews.

IBM Watson Assistant



About

Last Updated: 2018-01-26 | Edit in GitHub

With the IBM Watson™ Assistant service, you can build a solution that understands natural-language input and uses machine learning to respond to customers in a way that simulates a conversation between humans.

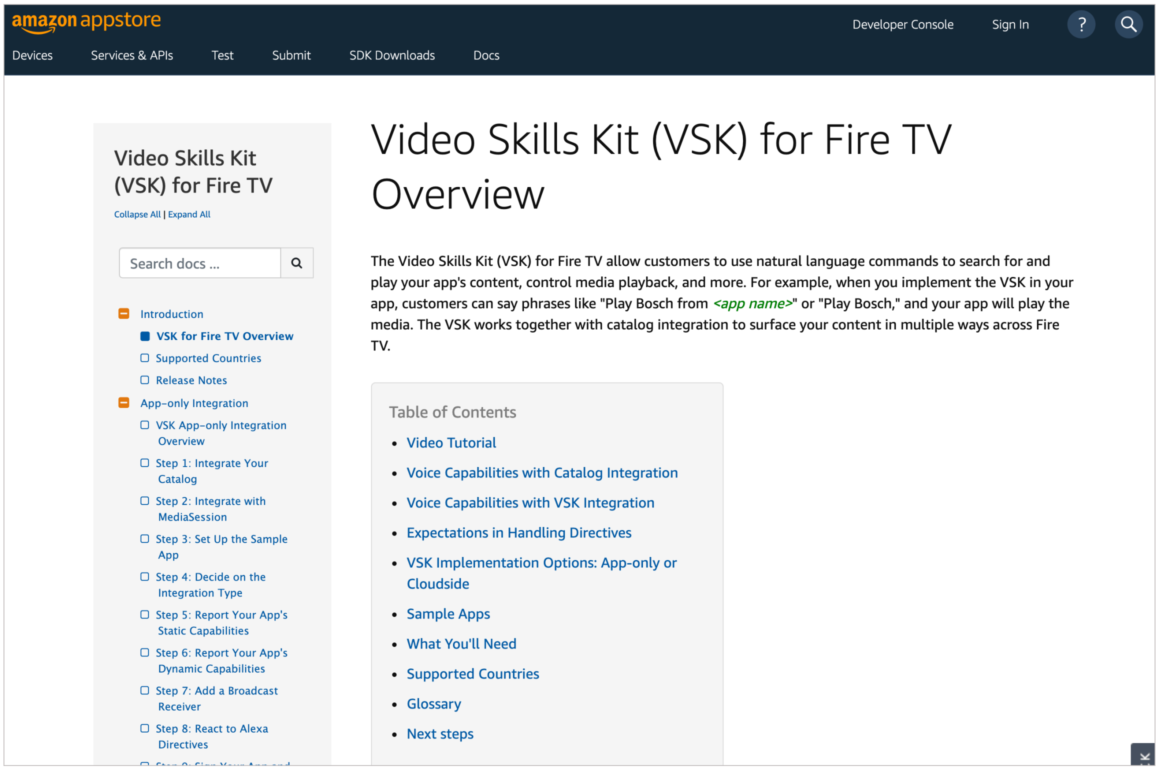
How it works

This diagram shows the overall architecture of a complete solution:

IBM Watson Assistant overview

IBM Watson Assistant starts off with a brief summary of the service, followed by a high-level diagram of the system and a summary about how to implement it. Including a diagram of your API gives users a good grounding about what to expect, such as the level of complexity and time it will take to incorporate the API.

Video Skills Kit for Fire TV



Video Skills Kit (VSK) for Fire TV

Collapse All | Expand All

- Introduction
  - VSK for Fire TV Overview
  - Supported Countries
  - Release Notes
- App-only Integration
  - VSK App-only Integration Overview
  - Step 1: Integrate Your Catalog
  - Step 2: Integrate with MediaSession
  - Step 3: Set Up the Sample App
  - Step 4: Decide on the Integration Type
  - Step 5: Report Your App's Static Capabilities
  - Step 6: Report Your App's Dynamic Capabilities
  - Step 7: Add a Broadcast Receiver
  - Step 8: React to Alexa Directives

Video Skills Kit (VSK) for Fire TV Overview

The Video Skills Kit (VSK) for Fire TV allow customers to use natural language commands to search for and play your app's content, control media playback, and more. For example, when you implement the VSK in your app, customers can say phrases like "Play Bosch from *<app name>*" or "Play Bosch," and your app will play the media. The VSK works together with catalog integration to surface your content in multiple ways across Fire TV.

Table of Contents

- Video Tutorial
- Voice Capabilities with Catalog Integration
- Voice Capabilities with VSK Integration
- Expectations in Handling Directives
- VSK Implementation Options: App-only or Cloudside
- Sample Apps
- What You'll Need
- Supported Countries
- Glossary
- Next steps

VSK for Video Skills Kit for Fire TV

This is an overview I wrote for a product called “Video Skills Kit for Fire TV.” The product overview stays at a high level by describing the capabilities the product provides, general implementation options, sample apps available, requirements to complete the implementation, supported countries, and next steps. There’s a parallel product overview page called Video Skills Kit for Echo Show.

Both of these product overviews are like product landing pages within a larger developer portal that covers many different products. In fact, if you go to the developer portal homepage, the page just routes you to different product overview areas.

Activity with product overviews

With the open-source project you identified (p. 686), identify the API overview. Then answer the following questions:

1. Does the documentation have a product overview?
2. Does the overview clarify who the API is for?
3. Does the overview indicate the market need or business problem the API solves?
4. What questions do you still have about the API after reading the overview?
5. How does the overview transition into a getting started tutorial or other first steps with the API?

Summary of best practices for product overviews

As a summary, consider including these general sections in a product overview:

- Description of the product
- Sample use cases
- Intended audience and technical level
- Requirements for use (system requirements, geo-requirements)
- List of components involved
- High-level architectural diagram of components, workflow
- Development effort and scope
- How to get support
- Most popular topics
- Known limitations, release notes
- Link to getting started tutorial

Every product seems to elicit its own unique sections on the overview, but these sections will give you a good starting point. Now let's explore the reasons why product overviews frequently fail.

Reasons why product overviews are often minimal or nonexistent

Have you ever found yourself reading documentation for a product and wondered, what exactly is the product? What does it do? Who is this for? Why isn't it more clear? You look for the big picture and higher-level understanding, but every topic seems to assume that you already know more than you actually do. The nature and use of the tool remains muddy.

In general, a product overview should allow users to get a good sense of what the product does, who it's for, why they might use it, the pain point the product solves, requirements and availability, how to get started, how to get help when needed, and other foundational concepts. Ideally, the product overview should give you a solid understanding of the product and what it's used for.

Yet in so many cases, when I start reading through documentation for a product, I'm often left confused and without a clear sense of what it's for or how I might actually use it, let alone how to get started. Why are some product overviews so unfulfilling, so brief, disappointing, and weak? In this section, I'll explore several reasons for anemic product overviews.

Cause 1: The reader isn't the intended audience, so the overview fails for the reader

Perhaps the main reason that product overviews fail is because the reader (for example, a tech writer reading a product overview about some API for developers) isn't the intended audience for the product. As such, the overview might fail for that particular reader but actually be fine for the intended audience. This mismatch of actual reader versus intended reader makes it difficult to make judgments about product overviews.

As an example, take a look at some of the product overviews in Microsoft's Azure docs ↗, which look exemplary to me. You could use any product as an example, but let's start with the first product, the Anomaly Detector ↗. The starting topic is What is the Anomaly Detector API? ↗. (In fact, all docs seem to start out with "What is … [product]?" This frequent pattern creates a nice sense of predictability to the various doc sets in their portal.) The first two paragraphs start as follows:

> The Anomaly Detector API enables you to monitor and detect abnormalities in your time series data without having to know machine learning. The Anomaly Detector API's algorithms adapt by automatically identifying and applying the best-fitting models to your data, regardless of industry, scenario, or data volume. Using your time series data, the API determines boundaries for anomaly detection, expected values, and which data points are anomalies.
>
> Using the Anomaly Detector doesn't require any prior experience in machine learning, and the RESTful API enables you to easily integrate the service into your applications and processes.

Although the sentences seem clear, and there are screenshots, interactive demos, descriptions of features, getting started topics, and more, I'm still lost because I'm not the intended audience for the product. What is "time series data"? What kind of data is appropriate to analyze here? Why would I want to look for abnormalities in my data? What kind of application or process would I integrate this anomaly detection service into? *I dunno…*

As good as this product overview is written, it doesn't make sense to me because I'm not the intended audience. I'm not a developer working with large data sets, nor am I involved in machine learning algorithms. I can't even understand what scenario would make sense where I'd have "time series data" with anomalies that I need to detect as part of a machine learning model that I'm building, even though this scenario is apparently applicable across industries.

It's not the writer's responsibility to bring non-target users up from ground zero here, holding my hand through this knowledge domain and assuming I know nothing. But it would help to perhaps explicitly identify the audience here. Even without identifying the audience, though, it's pretty clear reading this overview that I'm not the user envisioned for this product.

So how do you, as a technical writer, a person who is most likely an outsider to the domain you're working in, know if the overview makes sense to the intended audience? This is the whole crux of writing documentation: most of the time, you're an outsider to the knowledge domain, so it's hard to know what the audience already knows or does not know, and what to explain or assume.

As technical writers, we usually spin our lack of domain awareness as a positive, because we don't end up assuming our audience knows so much already. We aren't hampered by the curse of knowledge, numb to the jargon and concepts our audience also isn't familiar with. So we explain the basics, we define terms, we start a few rungs lower on the knowledge ladder than people expect. And users often appreciate it.

But without closer interaction with users, we can only guess what users might know or need to know. Typically, we end up relying on feedback from those who do interact closely with users (such as devrel groups). Through them, we try to better understand the user's knowledge level, but even so, many times these groups can only speak from their own limited interactions. Most of us have experienced situations where engineers tell us that users will know this or that concept, only to learn later that users don't and the assumptions confused them. At the end of the day, we find ourselves staring at a product overview and, even if it fails for us, we hope it works for the right audience.

As we read through product overviews, we have to remember that we're usually not the intended audience. It might fail to orient us, but does it fail for the intended audience? At the very least, try to be clear about the intended audience in the overview, as this will set expectations for knowledge levels. You can also add a "Background Knowledge and Assumptions" section. This section could link out to some preparatory documentation (perhaps on other websites) that users should consult if they get lost.

Cause 2: UX's influence on intuitiveness implies that long overviews indicate bad design

Another reason why product overviews are anemic is due to UX's influence with intuitiveness. (This cause is related to the previous point but a separate facet.) The idea is that products should be intuitive and naturally address mental models that customers have, without the need for extensive explanations. Why would you need to explain a product in depth to the users who you built it for? If something needs a deep explanation, it probably isn't well-designed and intuitive for that audience.

Achieving intuitiveness in your product is a common goal of UX design. In What makes intuitive products intuitive? ↗, Scott Kitchell argues that a product is intuitive when it matches the mental model of the user. Scott says, "Intuitiveness can be created by designing every part of a product in reference to a mental model, and then promoting the mental model through the UI and marketing."

Mental models are the logic and theories in our heads that make sense of the world around us. For example, in mountain biking, a common product for seats is a "dropper post," which lets bikers dynamically raise or lower the seat post height by pressing a button on their handlebars. Why would one need such a button and the ability to quickly raise or lower the seat height while riding? If you're into mountain biking, you know that climbing dirt/gravel hills requires you to sit low while keeping weight on the back tire for traction, so you might need to lower the seat quickly on the climb, but then revert to regular height for other scenarios.

In short, if you're part of the intended audience, you already have a rationale for the feature and don't need extensive conceptual docs explaining the scenario and reason for the product. You won't see extensive conceptual docs for dropper posts on product detail pages. The need is already felt by the intended audience.

The problem in tech comm is that tech writers are usually outsiders to the domain, looking in at the product. We don't share the same mental model as our users. As a result, many details don't immediately make sense. Kitchell says:

> Mental models are literally the logic within our heads, so if it's in there, you'll see the logic in it. From the outside however, others will not. Unintuitive mental models are like irregular looking blocks — They don't fit well with other mental models which makes them harder to remember, and problem-solve with.

Ideally, the product should just make sense for users, without a need for in-depth explanation. If you don't share the same mental model as your users, it's difficult to assess how much users will actually need the who, what, and why of the product. It might just make sense to them based on their mental model and problem set, like a jigsaw piece that fits perfectly into the space for which the piece was created. To evaluate whether a product needs an overview, you have to evaluate it not from an outsider's mental model, but from the mental model of users.

In some cases, your product might require some new learning even for the target user. Mark Baker says, "… learning is about rearranging our own mental furniture, finding our way through the thickets of our own minds. The expert can help us enormously at certain key junctures in that process, but most of it we simply have to do for ourselves" (Chatbots are not the future of Technical Communication ↗). It's not always the case that users will intuitively understand the product. Some learning frequently needs to take place, and that learning often involves some mental strain (the learning of a new model), even for the target audience.

For more on mental models, see the Schemas and learning ↗ section in Principle 5: Conform to the patterns and expectations of the genre and schemas ↗ *Schemas* are a more scientific term referring to the mental models in our heads that make sense of the world. See also Script theory ↗ in the same article. Script theory argues that if designers create experiences that match the schemas by which users operate, users will naturally know what to do and act in an almost scripted way. For example, Kirk St. Amant says if you design your hospital waiting room in an archetypical, expected way, then users will naturally know what to do when entering the space.

Define the stories that your audience uses to think about the scenario your product addresses. What mental model or schema organizes their thinking about the problem? If your product overview already naturally fits into this mental model, then you might not need to make the details more explicit in an overview — it might already make sense for the user.

Cause 3: Overview pages are hard to write, so they're often neglected

Another reason product overviews often fail for users is because, put simply, product overviews are hard to write, and so they are often poorly executed. The product overview requires you to be thoroughly familiar with the product, comfortable enough to summarize the product at a high level, describe the overall architecture, use cases, how to get started, requirements and limitations, and more.

As technical writers, we're often incrementing our understanding of a complex product little by little — we're piecing together what it's about, how it works, how to perform various tasks, the reference material about the APIs, and so on. We're slowly identifying puzzle pieces and trying to fit them together into the right picture. At any given time, there are likely many unidentified and unused puzzle pieces, making our current picture incomplete. It might not be until several weeks or months that we have a light bulb moment and glimpse the full puzzle picture.

To use another metaphor, I often like to think of projects as a monster that I battle and slay ↗. That moment when I slay the monster is when I unlock its secret and suddenly grasp how it ticks, how to unlock its data and have it returned to me. It's at that point, near the end, that I can properly write the overview page. In general, *I usually can't finish the first page of documentation until I write the last page of documentation.*

And when am I writing that last page of documentation? Sometimes right near crunch-time, about two weeks before release.

If you're working more as an editor and publisher rather than an author, the overview might be even more challenging. You might be reliant on general product descriptions from internal documents, without the additional context and detail that you get by struggling with the product for months with hands-on exploration and experimentation.

Recognize that you typically acquire the full knowledge to write the overview only after you've written all the other documentation. To avoid last-minute efforts, keep running notes on an overview draft that you add to as you work through the other documentation. Place section holders on the page, and then fill them in as you go.

Cause 4: Agile's co-development influence makes it difficult to surface higher-level content needs

Another cause for poor product overviews is agile's co-development influence with products. Agile software development prescribes close interactions with users as software teams develop and build out the product. When users are so intimately involved in product development, essentially co-collaborators with each iteration, they don't need the higher level overview, story, and purpose of the product. They need only the technical details for implementing it.

In Agile Principle 1: Active User Involvement Is Imperative ↗, Kelly Waters lists out 10 principles of agile and says "active user involvement is the first principle of agile development." Why is active user involvement so fundamental to agile development? User involvement is essential because software teams want to build the product in a way that matches users needs, and you can't do that without closely working with users, checking in regularly with each build to see if it matches their expectations, and course correcting to fine-tune the alignment needed to build the right product.

But just as product team members become somewhat numb to product jargon and the reasons behind decisions, the docs follow somewhat the same suit. There's no need to explain why the product is needed because, with active user involvement, these needs are communicated from the user from day one and throughout in regular meetings and other interactions. As such, there isn't a strong need for this higher-level overview and understanding. The conceptual basis for the feature is already understood by users because the product team iterated with users to develop the feature.

Once the feature is complete, some brief technical docs get added that explain how to use the feature. But the feature itself, the reason it was created, the problem it solves, the high-level overview and description of the feature, etc., is not documented because the initial user didn't need that high-level. The documentation likely follows a similar trend elsewhere, and soon you end up with lots of little building blocks and technical how-to's but no higher-level descriptions and glue between all of these tasks. New users who didn't participate in the feature's development have to try to derive back what the feature is and why/what/who the product is for.

Product overview anemia is a byproduct of the agile development process itself. This is where a technical writer's perspective as an outsider becomes so important. If you're an outsider to both the product and domain, you won't have this co-development history and won't have seen the product evolve from a sketch on a napkin to a fully released product. You'll see the lack of connecting glue between topics, the absence of a larger story that connects with your needs, and more. The problem is, without an audience asking for this higher-level information, you might be facing an uphill battle to generate the content. You might be writing an overview for an imagined future audience that hasn't yet materialized.

If the users were co-developers of the product and features (or frequent sounding boards during the design phase), don't use that group as a barometer for assessing content needs. Find someone who is new to the product.

Cause 5: Higher-level content is already handled by developer marketing content, making it redundant in docs

Another reason for anemic product overviews is because many of these higher-level questions are usually handled in the developer marketing layer, and tech writers don't usually operate in that pre-sales space. In many doc portals, there's a marketing layer that sits on top of the documentation. This marketing layer is supposed to articulate the larger story of the product — the problem the product solves, the target audience, use cases, case studies, and more — to a pre-sales audience.

As an example, see the example with AWS Lambda that I explained in the product overviews (p. 363) topic. In fact, the product overviews in the marketing layer pose challenges for overviews in technical documentation because tech writers usually try to avoid redundancy. Since many tech writers assume the marketing layer handles this larger story and overview about the product, this type of content is often absent or minimized in the documentation's product overview.

Additionally, the higher-level overview often gets more into pre-sales territory than many technical writers are comfortable with. In this space, you're trying to tell the who, what, and how of the product in a way that resonates with user pain points. In The importance of "how" in developer messaging, Matthew Revell argues that developer messaging needs to start with the *what* and *how* before the *why*. He touches on the need to build confidence with your audience, to align your goals with theirs. Revell says:

> The origin myth of a product provides a framework that enables people to form their own feelings and thoughts about it. Without ‘why’ there’s no developer community, no champions, no advocacy.

Origin myths are not typical content that technical writers create. For example, you will not find a tutorial on origin myths in any technical writing handbook.

Developer messaging focuses on building trust with users, finding an emotional connection with them, addressing the developer journey, and telling the product story. Most tech writers don’t think about this type of content in docs — this is the land of content marketing. For example, suppose you worked as a technical writer for Red Bull. Your primary task would be to describe the product’s ingredients, not to construct a story about Red Bull being the drink of extreme sports enthusiasts, for as helicopter skiers and daredevil mountain bikers.

As such, if there is not a marketing layer for the product, tech writers are unlikely to create one because this space entails writing that tech writers might not want to dabble in. Or the marketing layer might fully address questions, so they don’t need to be redundantly handled in docs.

However, any good content strategy should have alignment with each content touchpoint, from pre-sales to post-sales. In Principle 8: Align the product story with the user story, a series on how to simplify complexity, I argued that products often fail because the story the company tells doesn’t align with the story the user tells. Developer docs have the added complexity of having three stories: a company story, an end-user story, and a developer story. If all three groups are playing off different stories, the product likely won’t succeed.

Even if a technical writer’s job is to focus on the how and what, more than the why behind the product, technical writers should have a larger sense of product story that helps structure and direct the technical content. Ideally, the shape of documentation should be constructed around the developer journey, and that journey should connect with the product story.

Look to see if marketing content covers the higher-level content needs in the documentation overview. If so, you could simply link to the marketing content, or alternatively, put a more technical, matter-of-fact spin on the same content. Either way, think about the developer’s journey and story they tell themselves, and consider using this journey/story to shape your documentation.

Cause 6: Tech comm buys in to the “reading to do” paradigm for docs, minimizing the need for longer conceptual docs

Another reason for lack of product overviews, even when outsiders like tech writers create the product docs, is due to tech comm’s strategy preference for task-oriented docs. There’s a strong belief among most tech writers that users turn to docs only when they have a task-related problem they’re trying to solve.

As a result, docs are usually problem-oriented, focused on what users want to do and achieve. Conceptual docs are often seen as a sideshow to the task-oriented docs. This idea is so pervasive, it hardly needs explaining. The hallmark of good technical docs, most tech writers believe, is a list of numbered steps that takes users through a complex task.

This more action-oriented, experiential approach to learning has its roots in a movement called “minimalism” that John Carroll, author of the *Nurnberg Funnel* identified in the 1980s. Describing John Carroll’s minimalism approach, scholars David Farkas and Thomas Williams write:

> The premise behind minimalism is that people learning to use computer software are impatient, mentally active, and curious. They want to begin right away getting their work done; they want to exercise their problem-solving abilities; and they are apt to utterly reject or diverge from highly constraining instruction such as tutorials. Training material, therefore, must not impede the natural impulses of computer users, as systems approach documentation does. It should be as brief as possible, support the accomplishment of real work, help leaners recognize and recover from errors, and, when possible, permit non-sequential reading. Such documentation cannot be generated mechanically from a theory of instruction but requires careful attention to the needs and behavior of the intended users of the software and reiterative testing of the design. (See John Carroll's *The Nurnberg Funnel* and Minimalist Documentation. *IEEE Transactions on Professional Communication*, Vol. 33, Nov. 4, Dec 1990.)

If people are always anxious to do tasks, not read conceptual overviews, then why spend time on these conceptual overviews? What purpose do they solve when users just want a list of steps for the task they're trying to perform? With this mindset, the product overview gets shortchanged.

Don't get me wrong — I support task-oriented docs and agree that it's generally the right approach. There's merit behind experiential, action-oriented learning (which is explored more in the Reasons why getting started tutorials fail or don't exist (p. 390)). Explanation docs without a hands-on sense of the product often fall flat. We need context and experience with a product to better understand it. If you try to learn something without first tinkering with the product, it's hard because names don't mean much unless you have something to hang them on. I learn best by mixing the two experiences — tinkering and reading, back and forth.

But task-oriented docs often swing too far toward tasks, resulting in minimal or anemic overview information. When that happens, you often end up confusing users with various tasks and no higher-level content that helps their decision-making about which tasks to follow and why.

In research about how developers use APIs (p. 438), researchers have identified "opportunistic behaviors" (try-first), "systematic behaviors" (read-first), and hybrids of the two. When users are observed, there's much hybrid behavior than solely opportunistic or systematic. Just because you might be an opportunistic user, it doesn't mean you always skip conceptual explanations — it's just that you might not start with concepts. A hybrid reader might start with code, trying it out on their own, and circle back to the introductory conceptual information when the code doesn't work as expected.

Deciding to cater to one type of behavior at the expense of the other might not be practical, since the learning behaviors and approaches seem to be in constant flux.

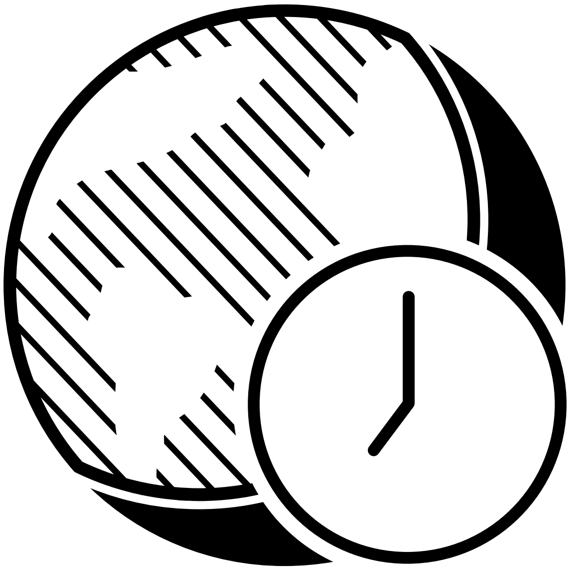
Remember that user behavior isn't night and day when it comes to opportunistic versus systematic behavior. Users flip back and forth between these two modes as needed. As such, try to link between the task-based topic and concepts where relevant to accommodate this fluctuating behavior.

API getting started tutorials

Following the product overview section (p. 360), you usually have a “Getting started” section that details the first steps users need to start using the API. This section often includes an end-to-end process of some starter task, complete in itself, to help users get started with your product.

Purpose of the getting started tutorial

The Getting Started tutorial is somewhat like the typical Hello World tutorial in developer documentation, but with an API. Your goal is to hold a user’s hand from start to finish in producing the simplest possible output with the system. For Hello World tutorials, the simplest output might just be a message that says “Hello World.” For an API, it might be a successful response from the most basic request.



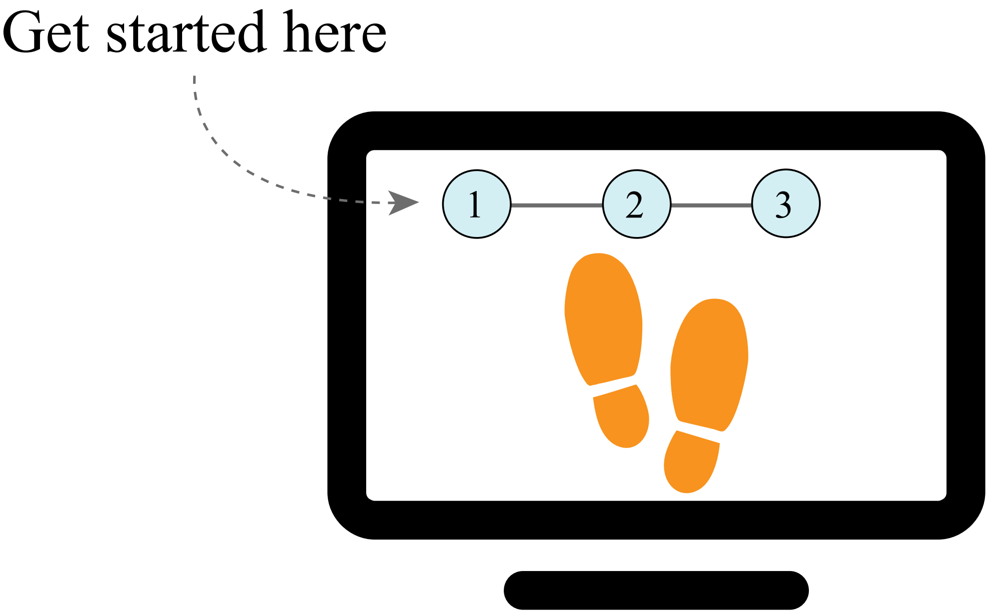
Think of getting started tutorials as a kind of Hello World tutorial with the API. How long would it take for a developer to get the simplest possible response using your API?

Both hello-world tutorials and getting-started tutorials share the same goal: to show a user how to use a framework, API, or some other system to get the simplest and easiest result, so they get a end-to-end sense of how it works.

As an example, you could take a common, basic use case for your API and show how to construct a request, as well as what response returns. If developers can make that call successfully, they can probably be successful with the other calls too.

The Getting Started tutorial might involve the following:

- Signing up for an account
- Getting API keys
- Making a request
- Evaluating the response



The Getting started tutorial usually walks users through the process from beginning to end but in a compressed, simple way

Put a link to your Getting Started tutorial on your documentation homepage. Make it as easy as possible for developers to use the API to get some result. This might mean using pre-provisioned accounts or setup configurations.

Presentation on getting started tutorials

I recently gave a presentation on product overviews (p. 360) and getting started tutorials. You can watch the presentation here:

Philosophic foundations for getting started tutorials

A getting started tutorial is more than just another type of content to include. Prioritizing a getting started tutorial in your documentation supports an action-oriented approach to learning that aligns with the preferences of many users, especially engineering-oriented developers.

In general, consider how people prefer to learn a new system. Do they prefer to sit passively while reading a manual? Or do they want to jump in, write some code, run it, see what it does, tweak a few buttons, and learn through some trial and error, experimentation and feedback?

While user learning types can fall all along this spectrum, from “systematic” (read first) to “opportunistic” (try first), as some researchers call it, the engineering mindset tends to favor the opportunistic (hands-on, experiential) approach. For more on different learning behaviors, see How Developers Use API Documentation: An Observation Study (p. 444).

The more action-oriented, experiential approach to learning has its roots in a movement called “minimalism” that John Carroll identified long ago. Describing John Carroll’s minimalism approach, David Farkas and Thomas Williams write:

> The premise behind minimalism is that people learning to use computer software are impatient, mentally active, and curious. They want to begin right away getting their work done; they want to exercise their problem-solving abilities; and they are apt to utterly reject or diverge from highly constraining instruction such as tutorials. Training material, therefore, must not impede the natural impulses of computer users, as systems approach documentation does. It should be as brief as possible, support the accomplishment of real work, help leaners recognize and recover from errors, and, when possible, permit non-sequential reading. Such documentation cannot be generated mechanically from a theory of instruction but requires careful attention to the needs and behavior of the intended users of the software and reiterative testing of the design. (See John Carroll's *The Nurnberg Funnel* and Minimalist Documentation. *IEEE Transactions on Professional Communication*, Vol. 33, Nov. 4, Dec 1990.)

In other words, minimize the amount of explanatory text that you require readers to wade through before getting to a more hands-on activity. Instead of constructing long walls of text, provide activities that get users interacting with the system in a direct way, learning as they go. Integrate some of your instruction within the tutorial. Make the learning action-oriented.

When users can experiment with your system and see the result, and learn by doing, something magical happens. Not only do users start to understand the product, they feel confidence with it. There's nothing more exhilarating than learning something new and getting it to work. This inspired confidence can add the motivation users need to continue learning and using your product.

Run in Postman button

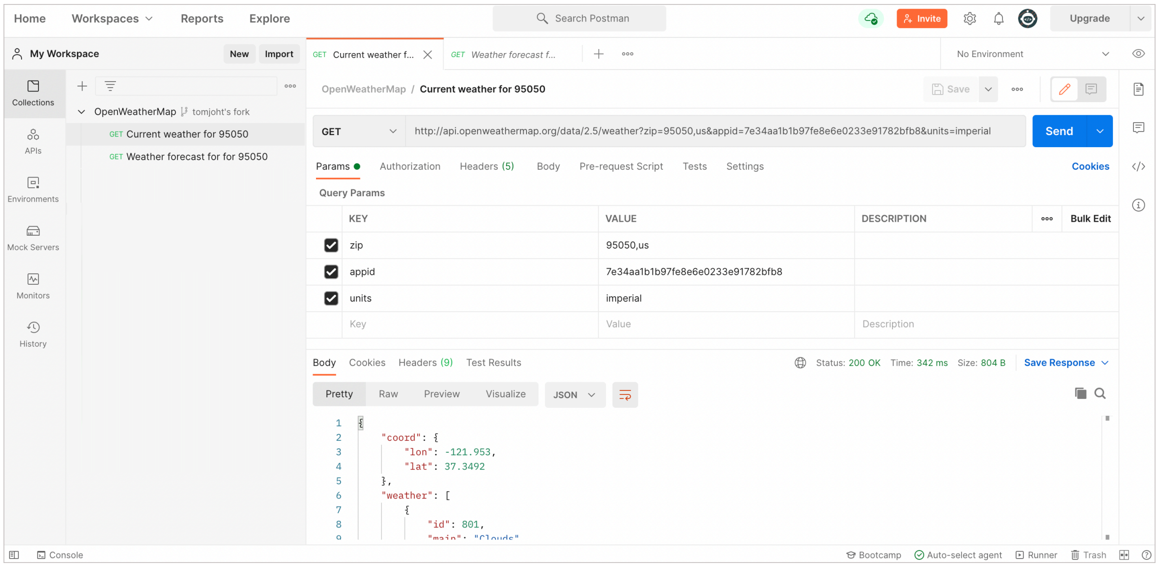
The quicker you can get users started, the better. One way to jumpstart the configuration of an API is by using Run in Postman buttons. This can reduce more lengthy and tedious explanations about parameters and other configurations for an API.

Postman is a REST API GUI client that we explored earlier in Submit requests through Postman (p. 71). If you have your API endpoints integrated with Postman, you can either create a workspace that contains your API endpoints, or you can export your Postman collections as a widget to embed in an HTML page. When users click the Run in Postman button, they will fork your collection into their own workspace.

To see how these Run in Postman buttons work, click this button:

Note: This content doesn't embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/docapis_doc_getting_started_section.html] to view the content.

When prompted, click **Fork Collection**. You'll then be prompted to sign into Postman first (create an account if you don't have one). Then you'll see the Postman web interface like this:



OpenWeatherMap / Current weather for 95050

GET http://api.openweathermap.org/data/2.5/weather?zip=95050,us&appid=7e34aa1b1b97fe8e6e0233e91782bfb8&units=imperial

	KEY	VALUE	DESCRIPTION
☑	zip	95050,us	
☑	appid	7e34aa1b1b97fe8e6e0233e91782bfb8	
☑	units	imperial	
	Key	Value	Description

Status: 200 OK Time: 342 ms Size: 804 B

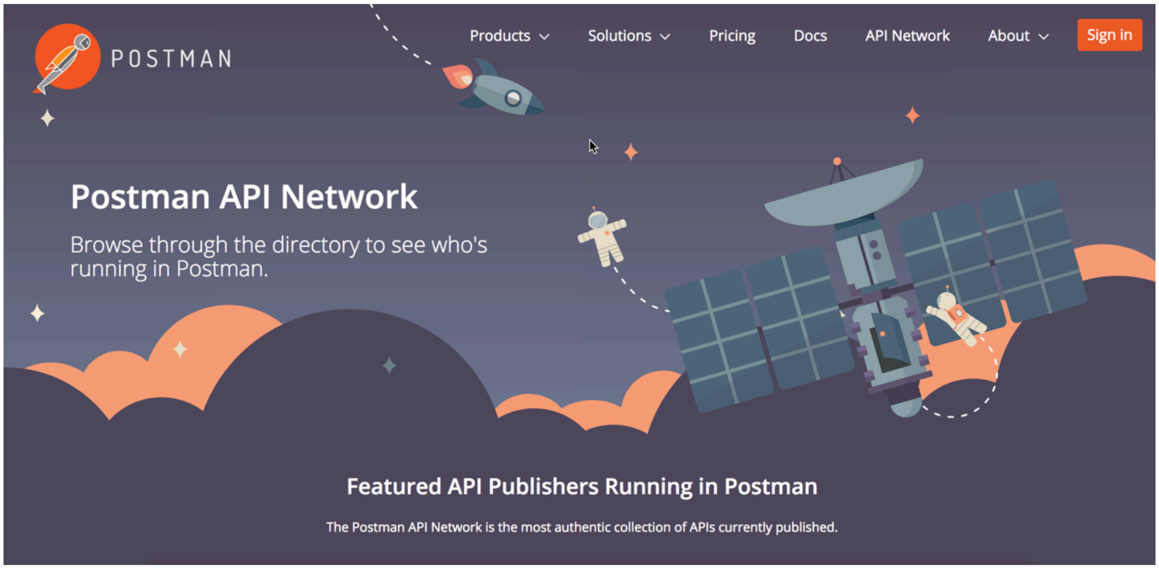
```
{
    "coord": {
        "lon": -121.953,
        "lat": 37.3492
    },
    "weather": [
        {
            "id": 801,
            "main": "Clouds"
```

Postman web interface

Expand the collection to view the endpoints, and click **Send**. You'll see the response generated.

Now change the `zip` code parameter to your own zip code and click **Send** again. In the response, look for `temp` and see if it matches the weather outside your home right now.

You can see the many demos of Run in Postman here ↗. Many of these demos are listed in Postman's API Network ↗.



Postman API Network

Browse through the directory to see who's running in Postman.

Featured API Publishers Running in Postman

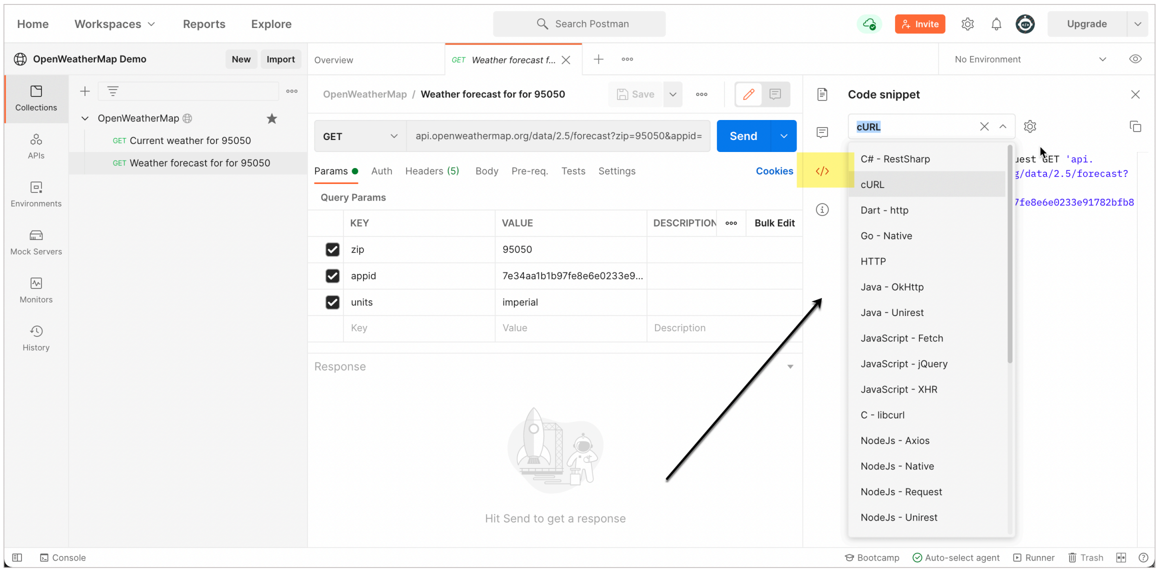
The Postman API Network is the most authentic collection of APIs currently published.

Postman API network

For details on how to share your Postman collections like this, see Creating Run in Postman buttons ↗ in the Postman documentation. To try out Run in Postman, you can either import an OpenAPI spec into Postman ↗ or enter your API information manually.

What's especially good about Postman is that it allows users to customize the API key and parameters and save those values. You can also safely share an API key. Although Postman doesn't provide same direct, in-documentation experience as Swagger UI (p. 240), in many ways the Postman client is more useful because it lets users configure and save the calls they make. They can easily save many different variations of the same endpoint, with different configurations. Postman is what internal developers often use to save and store API calls as they test and explore the functionality.

Postman also allows users to easily generate the needed code to make requests in practically any language. In the same Postman workspace, click the **</>** (Code) button on the right side (highlighted in yellow) to expand the Code pane.



GET api.openweathermap.org/data/2.5/forecast?zip=95050&appid=

Query Params

	KEY	VALUE	DESCRIPTION
☑	zip	95050	
☑	appid	7e34aa1b1b97fe8e6e0233e9...	
☑	units	imperial	
	Key	Value	Description

```
uest GET 'api.
g/data/2.5/forecast?

7fe8e6e0233e91782bfb8
```

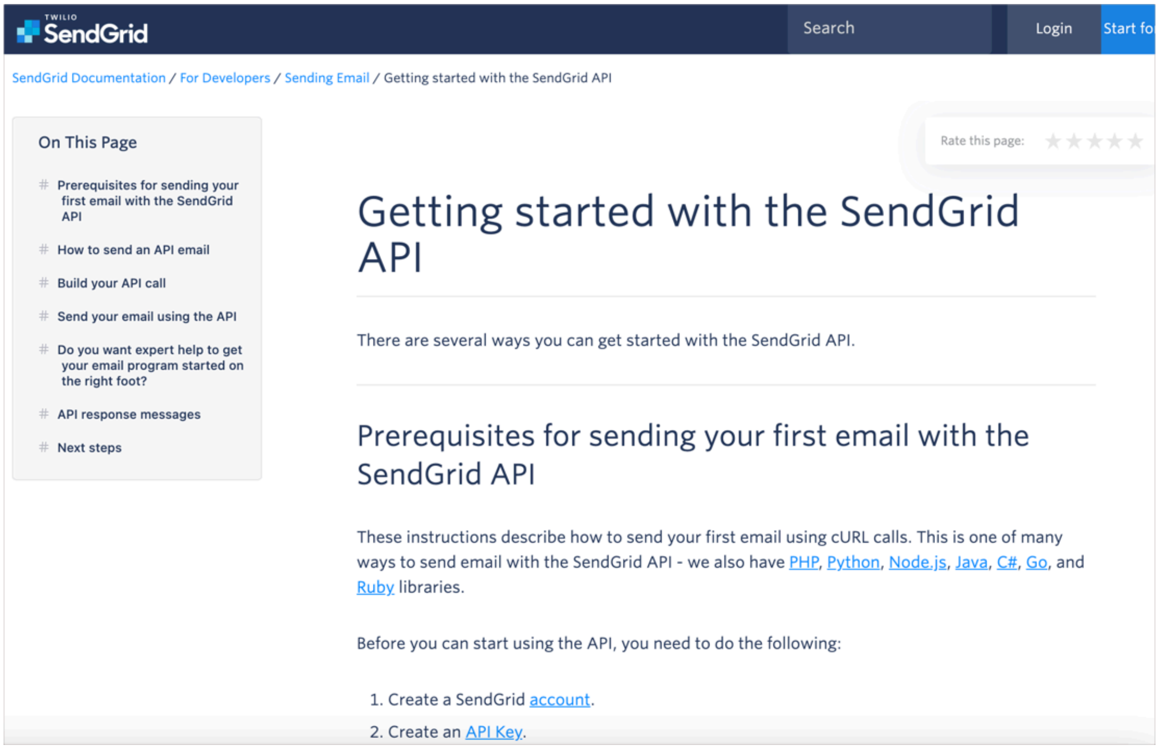
You can see the many options for running this same code in other languages. This gives users a jumping off point where they can build on your information to create more detailed and customized calls.

If you don't already have a "Try it out" feature in your docs, the Run in Postman button gives you this interactivity in an easy way, without requiring you to sacrifice the single source of truth for your docs.

Samples of getting started topics

Here are a few sample Getting Started topics in APIs. If you compare the various Getting Started sections, you'll see that some are detailed and some are high-level and brief. In general, the more you can hold the developer's hand, the better. However, the tutorial should still be brief and not simply redundant with the other documentation. The key is that you show the user the beginning-to-end, full process for working with the API.

SendGrid



SendGrid Documentation / For Developers / Sending Email / Getting started with the SendGrid API

On This Page

- Prerequisites for sending your first email with the SendGrid API
- How to send an API email
- Build your API call
- Send your email using the API
- Do you want expert help to get your email program started on the right foot?
- API response messages
- Next steps

Getting started with the SendGrid API

There are several ways you can get started with the SendGrid API.

Prerequisites for sending your first email with the SendGrid API

These instructions describe how to send your first email using cURL calls. This is one of many ways to send email with the SendGrid API - we also have PHP, Python, Node.js, Java, C#, Go, and Ruby libraries.

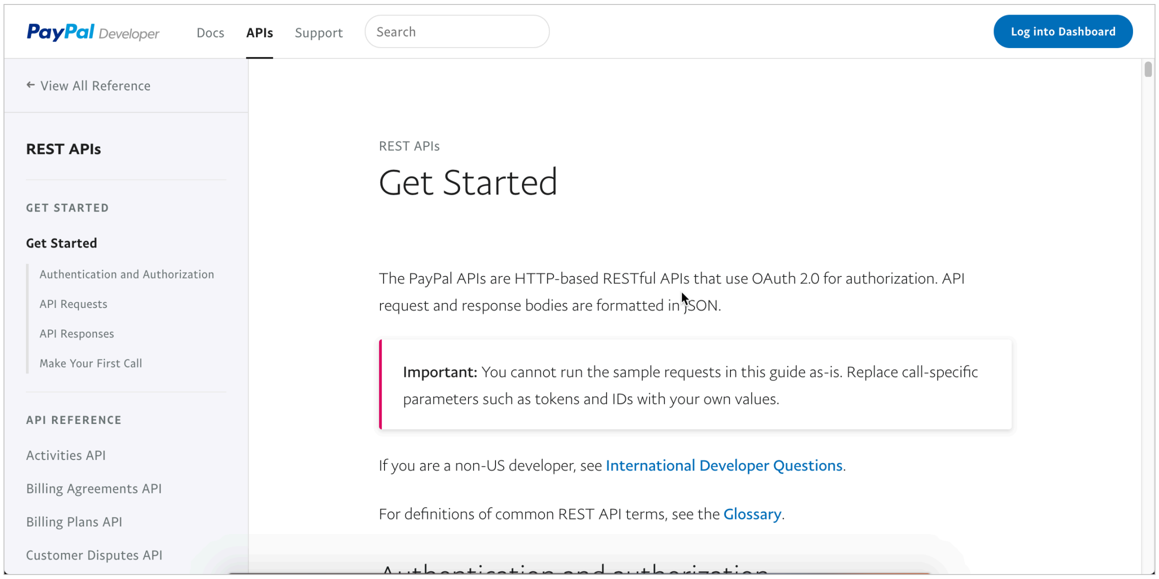
Before you can start using the API, you need to do the following:

1. Create a SendGrid account.
2. Create an API Key.

SendGrid getting started tutorial

This getting started tutorial from SendGrid is one of my favorites and one that I’ve had workshop participants try to complete in about 10 minutes. Interestingly, although it seems rather simple, only few participants could successfully send the email.

Paypal



REST APIs

Get Started

The PayPal APIs are HTTP-based RESTful APIs that use OAuth 2.0 for authorization. API request and response bodies are formatted in JSON.

Important: You cannot run the sample requests in this guide as-is. Replace call-specific parameters such as tokens and IDs with your own values.

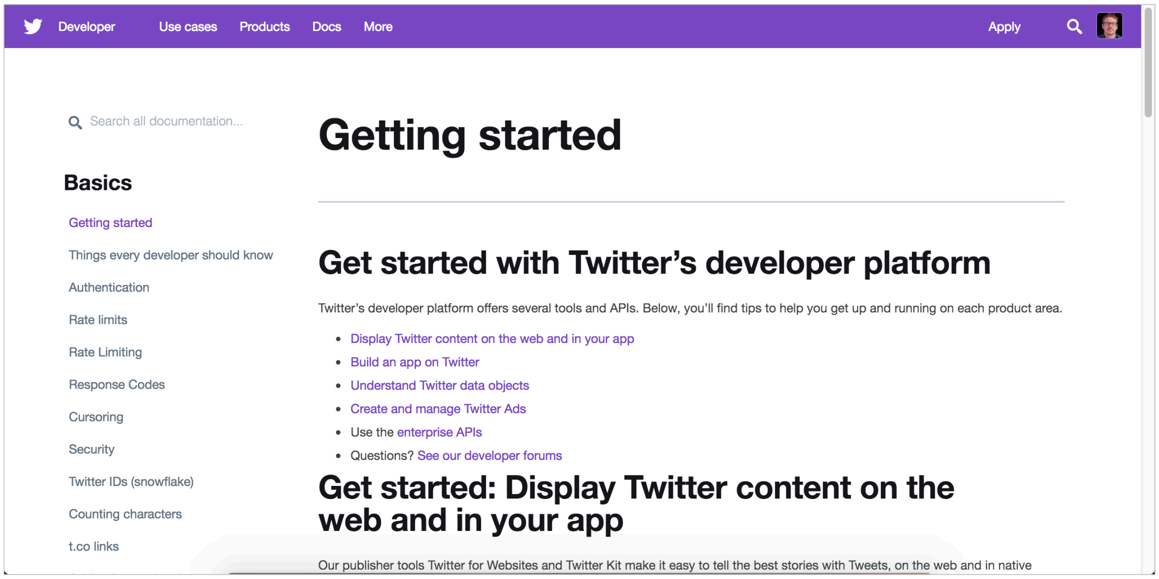
If you are a non-US developer, see International Developer Questions.

For definitions of common REST API terms, see the Glossary.

Paypal getting started tutorial

Paypal's getting started tutorial contains quite a bit of detail, starting with authorization, requests, and other details before making the first call. Although not so brief, this level of details helps orient users with the information they need. The format is clean and easy to follow.

Twitter



Getting started

Get started with Twitter's developer platform

Twitter's developer platform offers several tools and APIs. Below, you'll find tips to help you get up and running on each product area.

- Display Twitter content on the web and in your app
- Build an app on Twitter
- Understand Twitter data objects
- Create and manage Twitter Ads
- Use the enterprise APIs
- Questions? See our developer forums

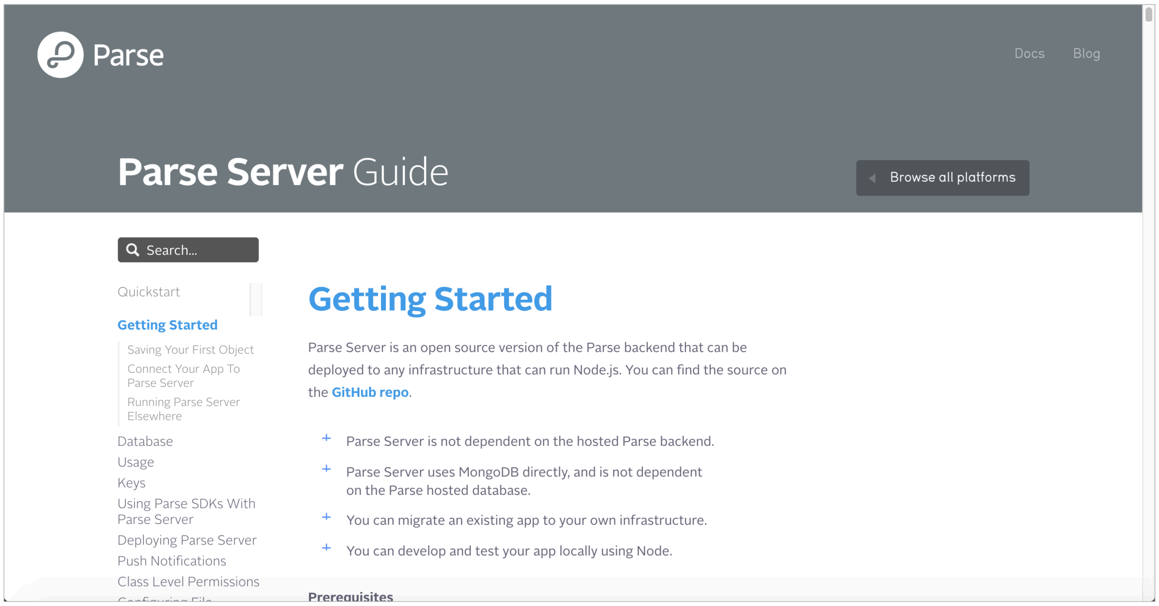
Get started: Display Twitter content on the web and in your app

Our publisher tools Twitter for Websites and Twitter Kit make it easy to tell the best stories with Tweets, on the web and in native

Twitter getting started

Twitter's getting started page has several getting started sections for different development goals. The text is concise and easy to follow. The tutorial links frequently to other documentation for more details. In the need for brevity, you might need to follow this same strategy — being brief and linking out to other pages that have more detail.

Parse Server



Getting Started

Parse Server is an open source version of the Parse backend that can be deployed to any infrastructure that can run Node.js. You can find the source on the GitHub repo.

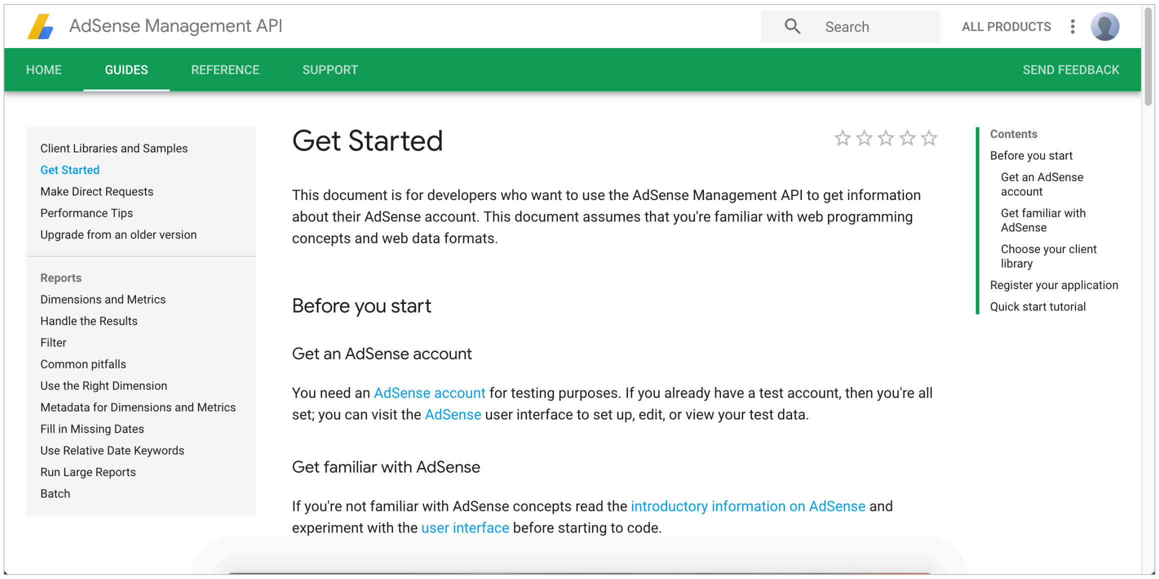
- Parse Server is not dependent on the hosted Parse backend.
- Parse Server uses MongoDB directly, and is not dependent on the Parse hosted database.
- You can migrate an existing app to your own infrastructure.
- You can develop and test your app locally using Node.

Prerequisites

Parse Server getting started

The Parse Server tutorial provides a fair amount of detail and handholding through the various steps. For the more detailed steps on connecting your app and running the server elsewhere, the tutorial links out to more information.

Adsense



Get Started

This document is for developers who want to use the AdSense Management API to get information about their AdSense account. This document assumes that you're familiar with web programming concepts and web data formats.

Before you start

Get an AdSense account

You need an AdSense account for testing purposes. If you already have a test account, then you're all set; you can visit the AdSense user interface to set up, edit, or view your test data.

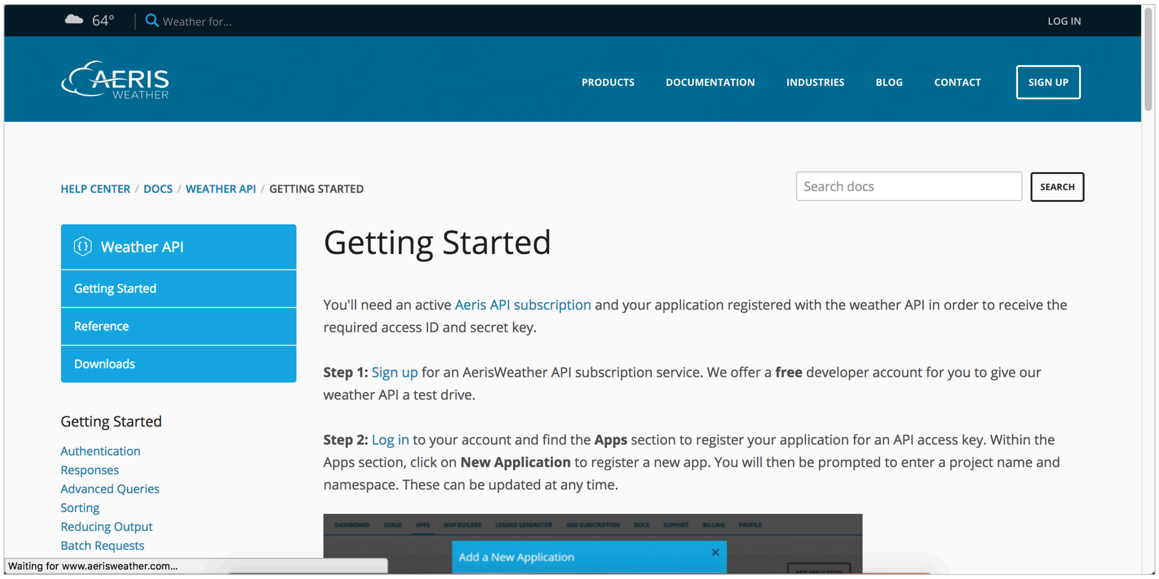
Get familiar with AdSense

If you're not familiar with AdSense concepts read the introductory information on AdSense and experiment with the user interface before starting to code.

Adsense getting started

The Adsense tutorial separates out some foundational prerequisites for getting started on the platform. After you get set up, it then provides a “quick start tutorial.” The tutorial walks users through a simple scenario from end to end, helping them get a sense of the product and its capabilities.

Aeris



Getting Started

You'll need an active Aeris API subscription and your application registered with the weather API in order to receive the required access ID and secret key.

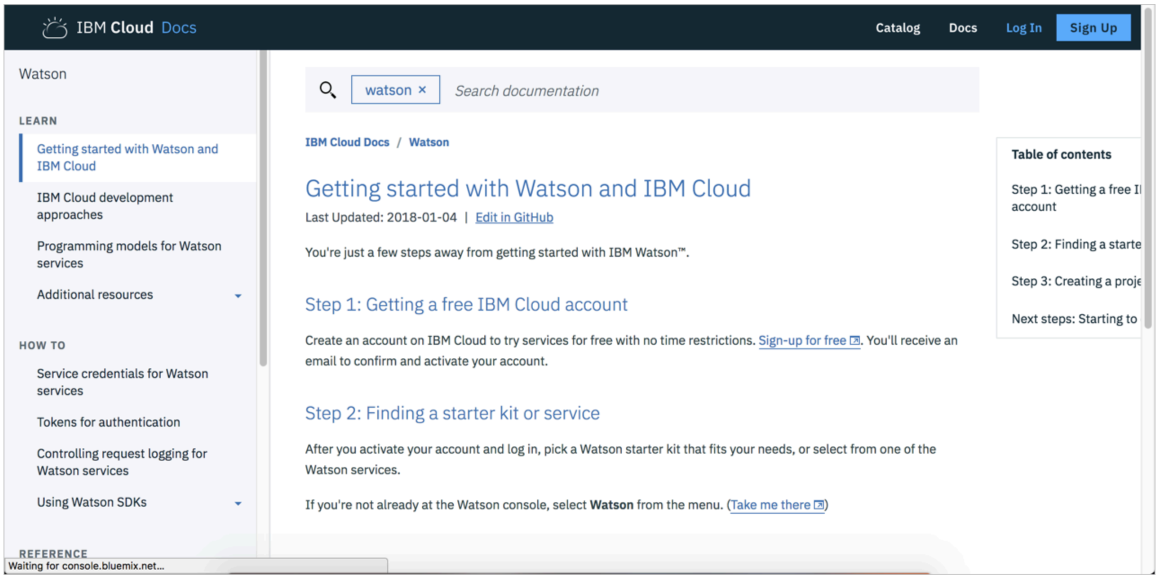
Step 1: Sign up for an AerisWeather API subscription service. We offer a **free** developer account for you to give our weather API a test drive.

Step 2: Log in to your account and find the **Apps** section to register your application for an API access key. Within the Apps section, click on **New Application** to register a new app. You will then be prompted to enter a project name and namespace. These can be updated at any time.

Aeris getting started tutorial

The Aeris weather getting started provides information for setting up an application and then making a request in one of several popular languages. While showing code in specific languages is undoubtedly more helpful for programmers coding in those languages, the code samples might be irrelevant to other users (e.g., Java devs might find the Python code irrelevant and vice versa). Focusing in a specific language is often a tradeoff.

Watson and IBM Cloud



Getting started with Watson and IBM Cloud

Last Updated: 2018-01-04 | Edit in GitHub

You're just a few steps away from getting started with IBM Watson™.

Step 1: Getting a free IBM Cloud account

Create an account on IBM Cloud to try services for free with no time restrictions. Sign-up for free. You'll receive an email to confirm and activate your account.

Step 2: Finding a starter kit or service

After you activate your account and log in, pick a Watson starter kit that fits your needs, or select from one of the Watson services.

If you're not already at the Watson console, select **Watson** from the menu. (Take me there)

Watson and IBM Cloud getting started tutorial

The Watson and IBM Cloud getting started tutorial lists three steps. It's not an end-to-end getting started tutorial, though. It just gets the user started in selecting a service for your project. At the end, you start to code using the Watson Dashboard. Ideally, a getting started tutorial should help a user see some tangible output, but whether that's possible or not depends on your API.

Other getting started tutorials

For other inspiration with getting started tutorials, explore the links below. Most of these samples come from the Pronovix Dev Portal awards in the Best Onboarding category. Pronovix lists the nominees here.

- Onfido: Getting started integrating
- Deutsche Bank: Getting Started Guide
- Yolt Technology Services: Getting started
- PlatformOS: Get Started
- Getting Started with Hazelcast Cloud
- Seon: Getting started
- Fedex: Getting started
- Vonage Developer: Getting Started
- Shopify: Getting started building payments apps
- Amadeus: Self-Service APIs
- Tapix
- Thingworx: Getting Started

- Codat: Getting Started

Summary of best practices for getting started tutorials

As a recap, include following characteristics in your getting started tutorial:

- Allow a new user to have some success with your product, even if the success is small, like getting a one-line value back from an API call.
- Remove the burden about setup requirements as much as possible through sandbox accounts, pre-configured data that users can load, Postman workspaces, sample apps, or more.
- Take a user from beginning to end through the tutorial, including registration, authorization, making a call with the API, inspecting the result, and more. As much as possible, the tutorial shouldn't just link out to longer docs but rather provide a compressed and condensed path with the product.
- Make sure the tutorial actually works and provides the advertised result, with as high of a success rate as possible.
- During the tutorial, take the opportunity to teach users about concepts, linking out to other parts of the docs. Include teaching moments and asides, assuming that the user is in a learning mode.
- Include a troubleshooting section that covers common issues users might see (e.g., 403 authorization errors).

Activity with getting started tutorials

With the open-source project you identified (p. 686), identify the getting started tutorial. Then answer the following questions:

1. Does the API have a getting started tutorial?
2. Does the getting-started tutorial walk you through an end-to-end scenario?
3. Can you follow all the steps in the getting started tutorial successfully?
4. How long does the getting started tutorial take to complete?
5. In an attempt to simplify the instruction and keep it brief, does the documentation make assumptions about your technical level and familiarity with the domain?

Reasons why getting started tutorials fail or don't exist

So far I've covered best practices for getting started tutorials. Now let's explore some reasons why getting started tutorials are often poor or non-existent. In other words, why do getting started tutorials fail? Why are they sometimes hard to follow?

Cause 1: Getting started tutorials are seen as an optional extra, and few writers have time for optional work at release crunch times

One reason getting started tutorials are often absent is because they're seen as going the extra mile, delivering content that exceeds normal expectations. Given time constraints and bandwidth, creating a getting started tutorial might not make the cut against the pressure for more pressing documentation needs. For example, if you're still working on reference documentation for your API, that's probably more important than a getting started tutorial. If you haven't even explained what information gets returned in an API call, that is likely more important.

Without question, there's a hierarchy of needs with docs, and a getting started tutorial sits more towards the top (meaning, it isn't essential). If you're still working on the core docs, the getting started tutorial might not be achievable within the release timeframe.

If writing a getting started tutorial isn't possible within the release timeframe, plan for it in the next iteration. You can usually continue improving and adding to docs after a release.

Cause 2: The product setup might be too involved or impractical for a getting started tutorial

Your product might require more setup than is practical for a getting started tutorial. For example, the user might need to be officially authorized or credentialed, which might be possible only after signing a contract. Some requirements such as creating a database, setting up a server, or other configuring a service might be necessary before the user can call an API. And you might not have this infrastructure readily available for the user, especially not in a sandbox/temporary way.

One of the products I documented at Amazon was developer docs for Fire TV. Getting things setup for Alexa integration literally took 3-4 hours, with no simple shortcuts to speed things up. Developers had to set up AWS accounts, identity access management credentials, configure permissions for buckets, set up other services, add code in various files, and more. There really wasn't any quick shortcut even with a sample app (though the sample app made the tutorial possible in an afternoon rather than a week). In short, your product might not facilitate getting started in an easy way.

If your product doesn't allow users to get started in an easy way, consider scaling back the result. Maybe the user can't call the API easily. Maybe all you can do is teach a user how to register for API keys, or show them how to issue a command in a terminal that brings up the help information. Either scale the expectations down a bit, or go the other route and lengthen the tutorial so that it actually does provide value but might require more time and effort to get through.

Cause 3: There's no sample app to demonstrate how to call the API

Another obstacle to getting started tutorials occurs with native APIs, such as Java APIs, that require users to have an app already set up. If you don't have a sample app for the user, it will likely be difficult to show how to make a call to the API. Most native API documentation involves various classes and other app configuration across various files. In short, you need the context of an app to run the code in the tutorial.

But getting engineers to make a sample app (p. 476) available is a tall order with many risks involved. Even though most QA teams have a simple sample app to facilitate testing, product teams don't usually like to give external developers these sample apps for fear that they'll use them for real scenarios. The sample apps are usually so primitive that they are in no way production scalable.

In scenarios where you lack a sample app, you could still provide code snippets and instruct users to make the code configurations in their own apps. Assume that your target users will actually have apps and be familiar with the programming language; tell them to add a class and instantiate it with certain parameters in their app, etc. If you find that the instructions go beyond your audience's skill level, double down on those efforts to make a sample app available. (You can always add detailed notes to the sample app that state the app is not intended for production use and is a simplified form of an app to demonstrate API calls only.)

Cause 4: The getting started tutorial omits details for the sake of brevity that might lead to confusion

Another obstacle to creating the getting started tutorial is the basic task of writing this content, as it requires more writing skill. Ideally, the getting started tutorial condenses longer, more detailed information into a succinct one-page tutorial. Yet this tutorial is supposed to take a user through registration, setup, authorization, execution, viewing results, interpretation, and more. It's like being asked to create a poem out of a novel, with the requirement that the same general story gets told.

Being brief at the expense of creating a short tutorial might lead to a confusing user experience. Leaving out more specifics, notes, and other more explicit instructions for the sake of brevity goes against our tech-writing principles of clear, explicit documentation.

The ability to reduce and distill information down to short, crisp summaries for a getting started tutorial is a higher-level writing skill. It also requires more effort and subject mastery. For more information on distilling information, see Principle 4: Reduce and distill vast information down to its essence ↗.

In places where you choose to be succinct and omit detail, link out to your regular docs for more information. In these cases, consider the approach to be a progressive information disclosure strategy. If a user needs more information, he or she can click the link and get it. But if not, the concise version suffices.

Cause 5: The tech might be too complicated for tech writers to walk through themselves

Another reason why getting started tutorials might be absent from docs is because the tech itself might be beyond the tech writer's reach. Suppose you have an advanced product that requires senior-engineering-level knowledge to use. In this scenario, the tech writer might mainly be acting as an editor, curator, and publisher — facilitating content mostly written by engineers. Since getting started tutorials are mostly hands-on, experience-driven deliverables, it might be too advanced for the tech writer. This is the nature of working in developer docs, especially without an engineering background. The tools are really designed for engineers who have a lot more training, experience, and knowledge in areas that we lack.

For example, suppose the integration assumes access to a vehicle's hardware abstraction layer, as well as knowledge about the way this layer connects to the vehicle's custom infotainment system in the head unit. In this case, the system environment might be too complex to wrangle into a simple getting started tutorial. It can be hard to walk through all of these same tasks in an end-to-end way.

Or putting it more optimistically, the tasks might be within the tech writers reach at *some* point, but not immediately. The tech writer might need more time to set up environments and walk through all of the tasks — time that's not available at present.

Consider asking an engineer to write the getting started tutorial. If you can't follow it, this could form the basis of homework or other learning that you chip away at little by little in your path of continual learning.

Cause 6: The content isn't tested against real users

Another reason for poor getting started tutorials is that they are rarely tested against actual users. (This could explain why a lot of documentation is poor, not just getting started tutorials.) During one of my API workshops, I had participants go through this 5 minute Sendgrid tutorial ↗. Out of about 20 workshop participants, only 1 managed to do the tutorial successfully. It seemed simple on the surface, but in practice, users got hung up many details. (Granted, this was a room full of tech writers, not engineers, but still.)

Testing docs against users isn't usually something tech writers do, though most admit it would be a good idea. User testing usually comes in the form of feedback to product teams, support cases, or through training sessions where users provide feedback in real-time.

If you want to make getting started tutorials a better experience, ask a few engineers (preferably not on the product team) to spend an hour or so going through the getting started tutorial. Unlike other content in docs that might require more setup and context, the getting started tutorial is supposed to be self-contained, short, and accessible to a new user. It's the perfect type of documentation to test.

API authentication and authorization

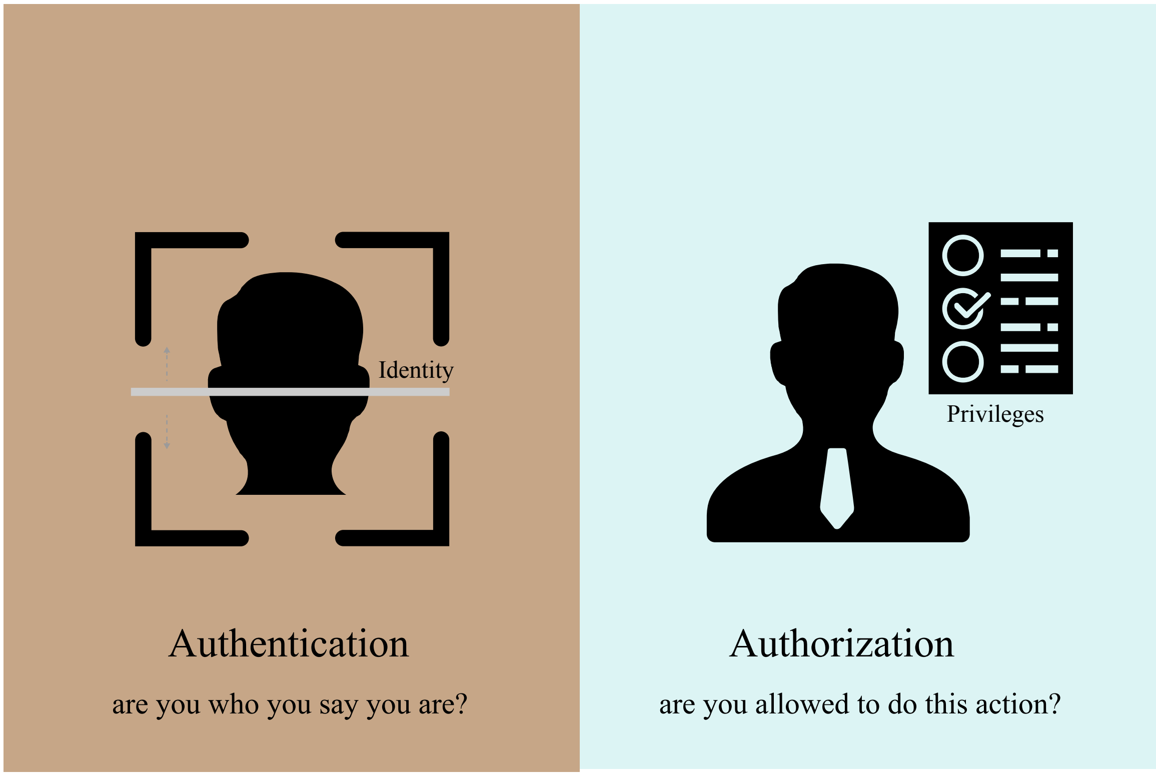
Before users can make requests with your API, they'll usually need to register for an API key or learn other ways to authenticate the requests. APIs vary in the way they authenticate users. Some APIs require you to include an API key in the request header, while other APIs require elaborate security due to the need to protect sensitive data, prove identity, and ensure the requests aren't tampered with. In this section, you'll learn more about authentication and authorization and what you should focus on in documentation.

Defining terms

First, let's define some key terms:

- **Authentication**: Refers to proving correct identity
- **Authorization**: Refers to allowing a certain action

An API might authenticate you but not authorize you to make a certain request.


Identity
Privileges
Authentication
are you who you say you are?
Authorization
are you allowed to do this action?

Authentication and authorization

Consequences if an API lacks security

Why do APIs even need authentication? For read-only APIs, sometimes users don't need keys. But most commercial APIs do require authorization in the form of API keys or other methods. If you *didn't* have any security with your API, users could make unlimited amounts of API calls without any kind of registration. Allowing unrestricted requests would make a revenue model for your API difficult.

Additionally, without authentication, there wouldn’t be an easy way to associate requests with specific user data. And there wouldn’t be a way to protect against requests from malicious users that might delete another user’s data (such as by making DELETE requests on another’s account).

Finally, you couldn’t track who is using your API, or what endpoints are most used. Clearly, API developers must think about ways to authenticate and authorize requests made to their API.

Overall, authentication and authorization with APIs serves the following purposes:

- Authenticate calls to the API to registered users only
- Track who is making the requests
- Track usage of the API
- Block or throttle any requester who exceeds the rate limits (p. 408)
- Apply different permission levels to different users

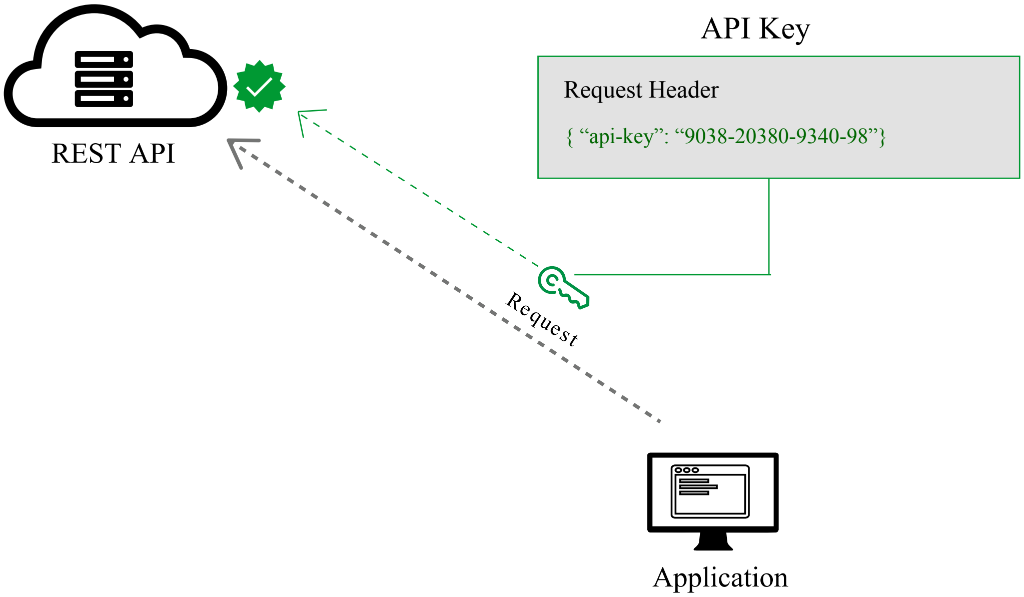
Different types of authorization

There are several methods for authorization. The following are various types of API authorization you might encounter:

- API keys (p. 394)
- Basic Auth (p. 395)
- HMAC (p. 396)
- OAuth (p. 397)

API keys

Most APIs require you to sign up for an API key in order to use the API. The API key is a long string that you usually include either in the request URL or request header. The API key mainly functions as a way to identify the person making the API call (authenticating you to use the API). The API key might also be associated with a specific app that you register.



APK keys use a string in a header property to authorize requests

APIs might give you both a public and private key. The public key is usually included in the request, while the private key is treated more like a password and used only in server-to-server communication. For some API documentation sites, when you're logged into the site, your API key automatically gets populated into the sample code and API Explorer.

Basic Auth

Another type of authorization is called Basic Auth. With this method, the sender places a `username:password` into the request header. The username and password are encoded with Base64, which is an encoding technique that converts the username and password into a set of 64 characters to ensure safe transmission. Here's an example of a Basic Auth in a request header:

```
Authorization: Basic bG9sOnNlY3VyZQ==
```

APIs that use Basic Auth will also use HTTPS, which means the message content will be encrypted within the HTTP transport protocol. (Without HTTPS, it would be easy for people to decode the username and password.)

When the API server receives the message, it decrypts the message and examines the header. After decoding the string and analyzing the username and password, it then decides whether to accept or reject the request.

In Postman, you can configure Basic Authorization by clicking the **Authorization** tab, selecting **Basic Auth** from the drop-down selector, and then typing the **username** and **password** on the right of the colon on each row. The Headers tab will show a key-value pair that looks like this:

```
Authorization: Basic RnJlZDpteXBhc3N3b3Jk
```

Postman handles the Base64 encoding for you automatically when you enter a username and password with Basic Auth selected.

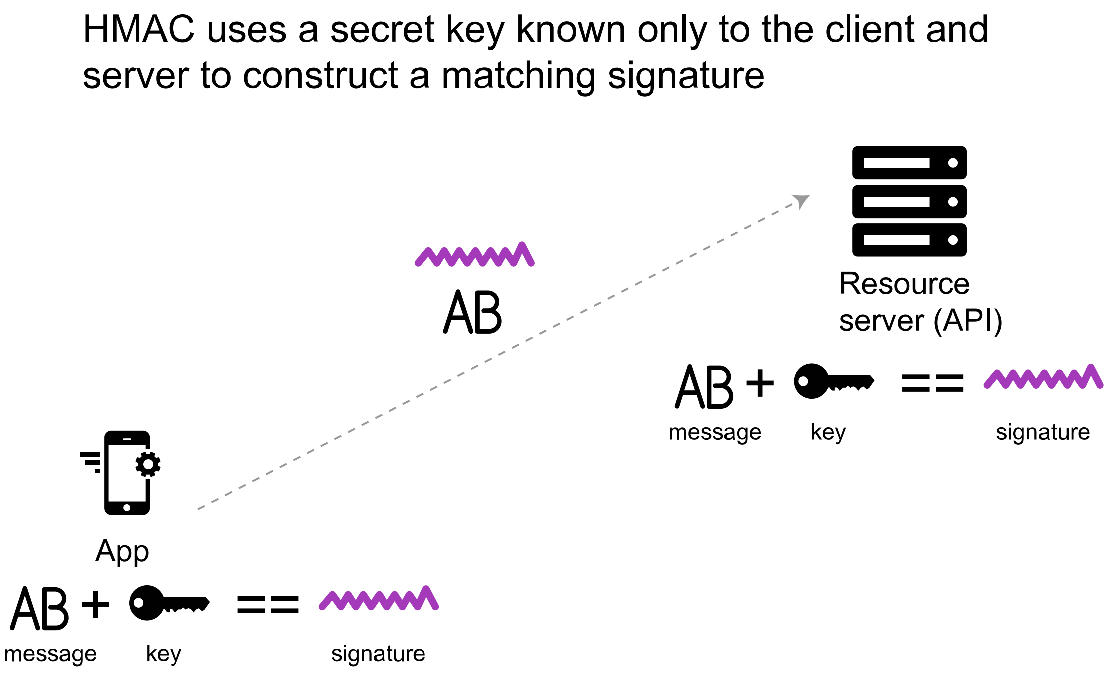
HMAC (Hash-based message authorization code)

HMAC stands for Hash-based message authorization code and is a stronger type of authentication, more common in financial APIs. With HMAC, both the sender and receiver know a secret key that no one else does. The sender creates a message based on some system properties (for example, the request timestamp plus account ID).

The message is then encoded by the secret key and passed through a secure hashing algorithm (SHA). (A hash is a scramble of a string based on an algorithm.) The resulting value, referred to as a signature, is placed in the request header.

When the receiver (the API server) receives the request, it takes the same system properties (the request timestamp plus account ID) and uses the secret key (which only the requester and API server know) and SHA to generate the same string. If the string matches the signature in the request header, it accepts the request. If the strings don't match, then the request is rejected.

Here's a diagram depicting this workflow:

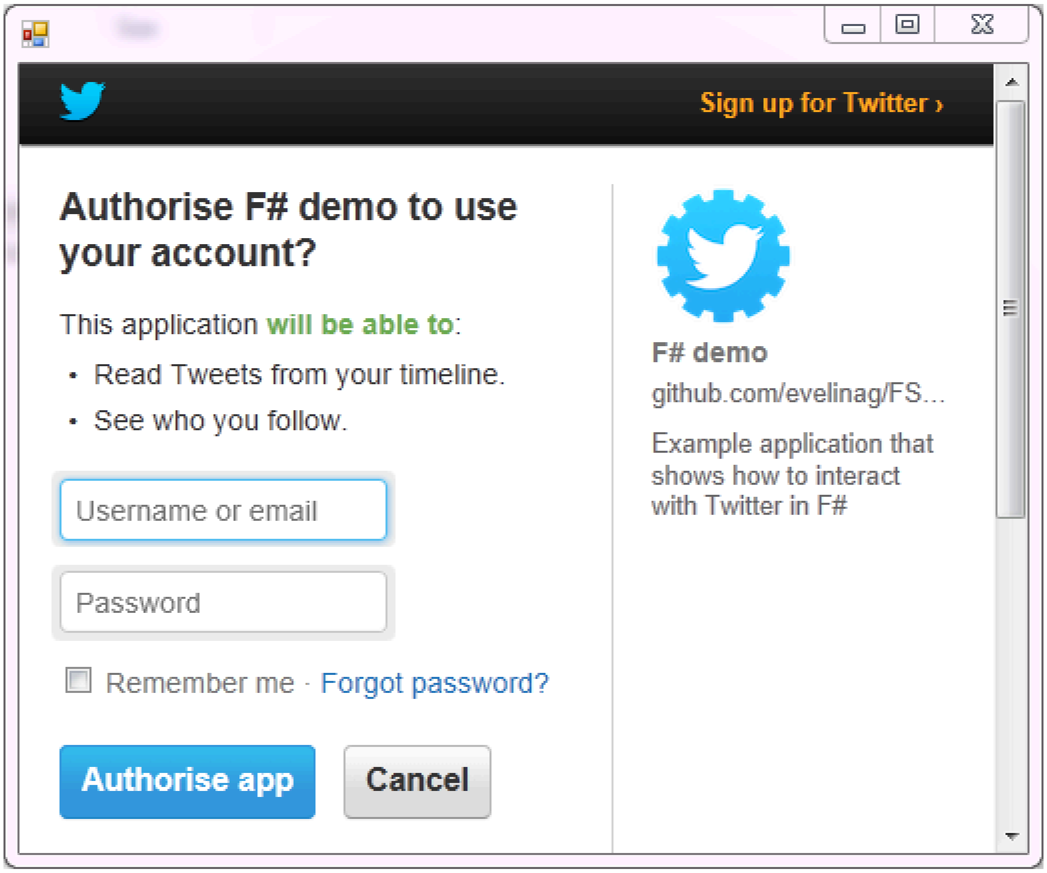

HMAC uses a secret key known only to the client and server to construct a matching signature
AB
Resource server (API)
AB + ==
message
key
signature
App
AB + ==
message
key
signature

HMAC workflow

The important point is that the secret key (critical to reconstructing the hash) is known only to the sender and receiver. The secret key is not included in the request. HMAC security is used when you want to ensure the request is both authentic and hasn't been tampered with.

OAuth 2.0

One popular method for authenticating and authorizing users is OAuth 2.0. This approach relies on an authentication server to communicate with the API server to grant access. You often see OAuth 2.0 when you're using a site and are prompted to log in using a service like Twitter, Google, or Facebook.



Authorise F# demo to use your account?

This application **will be able to**:

- Read Tweets from your timeline.
- See who you follow.

F# demo

github.com/evelinag/FS…

Example application that shows how to interact with Twitter in F#

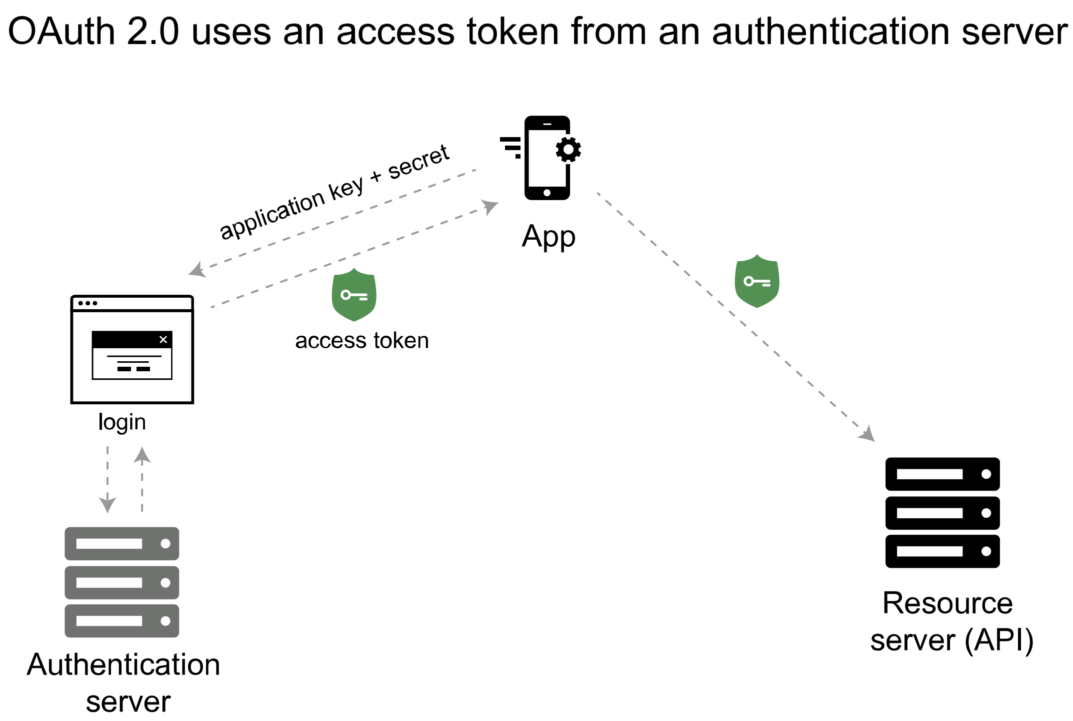
OAuth login window

There are a few varieties of OAuth — namely, "one-legged OAuth" and "three-legged OAuth." One-legged OAuth is used when you don't have sensitive data to secure. This might be the case if you're just retrieving general, read-only information.

In contrast, three-legged OAuth is used when you need to protect sensitive data. Three groups are interacting in this scenario:

- The authentication server
- The resource server (API server)
- The user or app

Here's the basic workflow of OAuth 2.0:



OAuth authentication

First, the consumer application sends over an application key and secret to a login page at the authentication server. If authenticated, the authentication server responds to the user with an access token.

The access token is packaged into a query parameter in a response redirect (302) to the request. The redirect points the user's request back to the resource server (the API server).

The user then makes a request to the resource server (API server). The access token gets added to the header of the API request with the word `Bearer` followed by the token string. The API server checks the access token in the user's request and decides whether to authenticate the user.

Access tokens not only provide authentication for the requester but also define the permissions of how the user can use the API. Additionally, access tokens usually expire after a period of time and require the user to log in again. For more information about OAuth 2.0, see these resources:

- Learn API Technical Writing 2: REST for Writers (Udemy), by Peter Gruenbaum
- OAuth simplified, by Aaron Parecki

What to document with authentication

In API documentation, you don't need to explain how your authentication works in detail to outside users. In fact, *not* explaining the internal details of your authentication process is probably a best practice as it would make it harder for hackers to abuse the API.

However, you do need to explain some necessary information such as:

- How to get API keys
- How to authenticate requests

- Error messages related to invalid authentication
- Sensitivity around authentication information
- Token expiration times

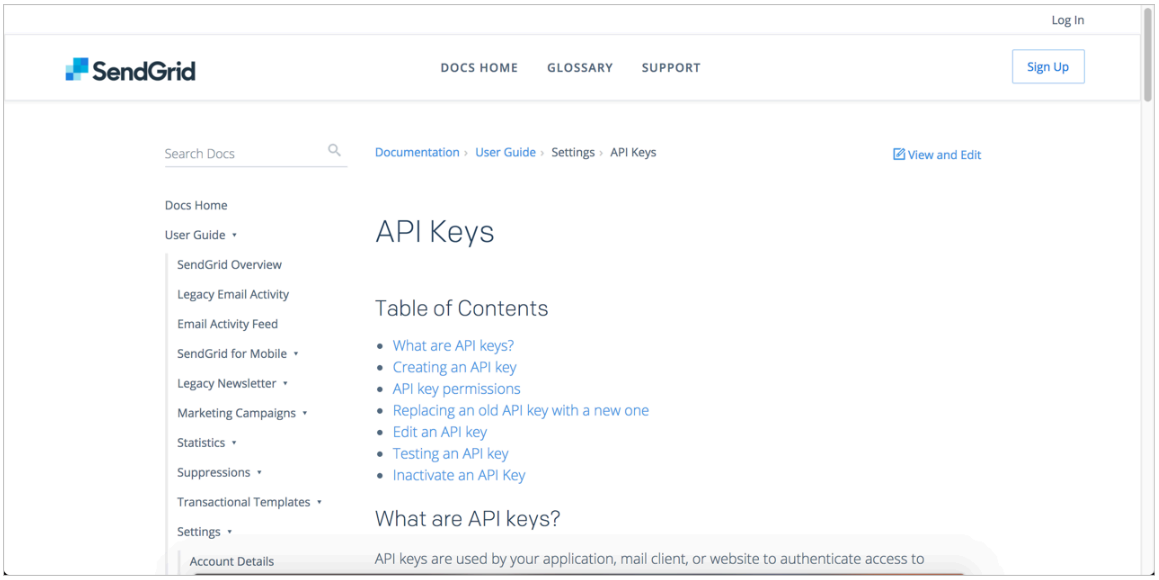
If you have public and private keys, you should explain where each key should be used, and note that private keys should not be shared. If different license tiers provide different access to the API calls, these licensing tiers should be explicit in your authorization section or elsewhere.

Since the API keys section is usually essential before developers can start using the API, this section needs to appear at the beginning of your help.

Samples of authorization sections

The following are a few samples of authorization sections in API documentation.

SendGrid



API Keys

Table of Contents

- What are API keys?
- Creating an API key
- API key permissions
- Replacing an old API key with a new one
- Edit an API key
- Testing an API key
- Inactivate an API Key

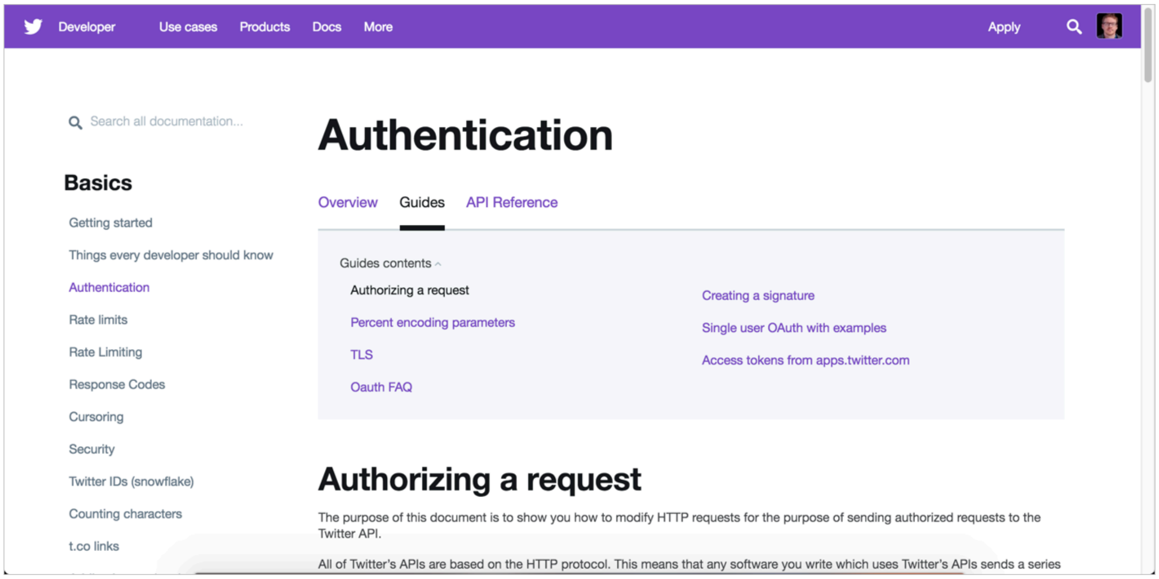
What are API keys?

API keys are used by your application, mail client, or website to authenticate access to

SendGrid API keys

SendGrid offers a detailed explanation of API keys, starting with the basics by explaining, “What are API keys?” Contextually, the topic on API keys appears with other account management topics.

Twitter



Authentication

Authorizing a request

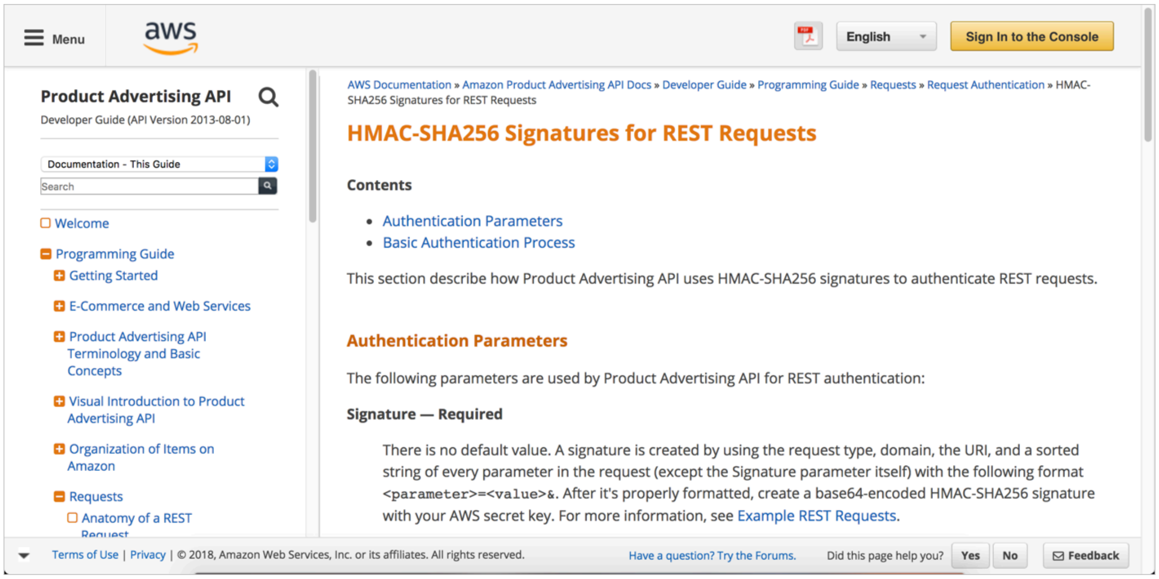
The purpose of this document is to show you how to modify HTTP requests for the purpose of sending authorized requests to the Twitter API.

All of Twitter's APIs are based on the HTTP protocol. This means that any software you write which uses Twitter's APIs sends a series

Twitter authorization

With Twitter, because the OAuth 2.0 authorization requirements are a bit more involved, a detailed example is warranted and provided.

Amazon Web Services


Menu
aws
English
Sign In to the Console
Product Advertising API
Developer Guide (API Version 2013-08-01)
Documentation - This Guide
Search
Welcome
Programming Guide
Getting Started
E-Commerce and Web Services
Product Advertising API Terminology and Basic Concepts
Visual Introduction to Product Advertising API
Organization of Items on Amazon
Requests
Anatomy of a REST Request
AWS Documentation » Amazon Product Advertising API Docs » Developer Guide » Programming Guide » Requests » Request Authentication » HMAC-SHA256 Signatures for REST Requests
Terms of Use | Privacy | © 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Have a question? Try the Forums.
Did this page help you?
Yes
No
Feedback

HMAC-SHA256 Signatures for REST Requests

Contents

- Authentication Parameters
- Basic Authentication Process

This section describe how Product Advertising API uses HMAC-SHA256 signatures to authenticate REST requests.

Authentication Parameters

The following parameters are used by Product Advertising API for REST authentication:

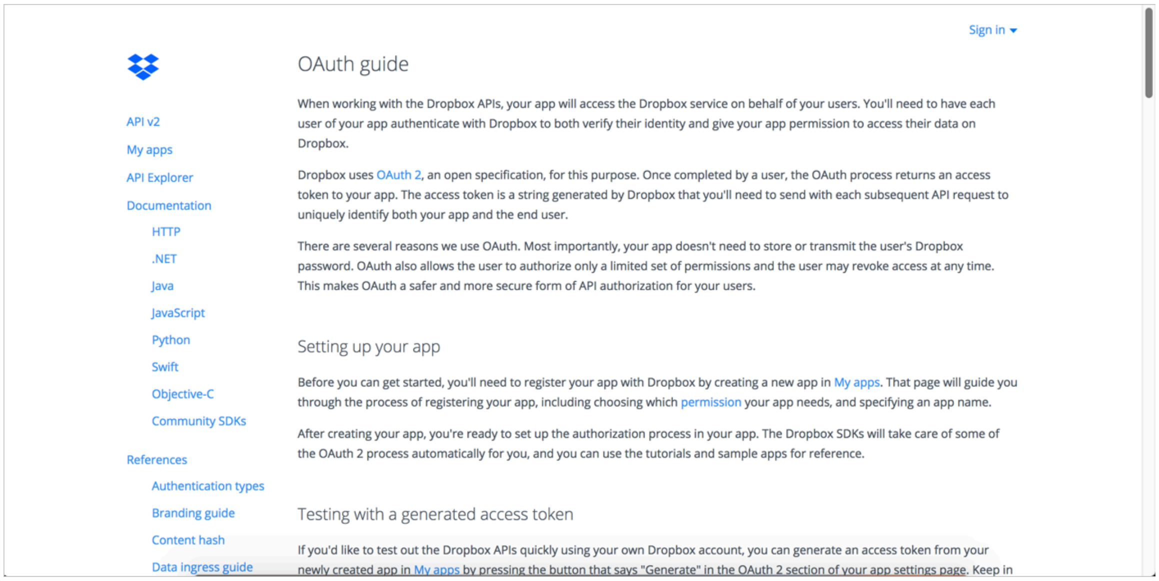
Signature — Required

There is no default value. A signature is created by using the request type, domain, the URI, and a sorted string of every parameter in the request (except the Signature parameter itself) with the following format `<parameter>=<value>&`. After it's properly formatted, create a base64-encoded HMAC-SHA256 signature with your AWS secret key. For more information, see Example REST Requests.

Amazon authorization

The Amazon example uses HMAC. The process is complex enough that a full-fledged diagram is included to show the steps users need to perform.

Dropbox



OAuth guide

When working with the Dropbox APIs, your app will access the Dropbox service on behalf of your users. You'll need to have each user of your app authenticate with Dropbox to both verify their identity and give your app permission to access their data on Dropbox.

Dropbox uses OAuth 2, an open specification, for this purpose. Once completed by a user, the OAuth process returns an access token to your app. The access token is a string generated by Dropbox that you'll need to send with each subsequent API request to uniquely identify both your app and the end user.

There are several reasons we use OAuth. Most importantly, your app doesn't need to store or transmit the user's Dropbox password. OAuth also allows the user to authorize only a limited set of permissions and the user may revoke access at any time. This makes OAuth a safer and more secure form of API authorization for your users.

Setting up your app

Before you can get started, you'll need to register your app with Dropbox by creating a new app in My apps. That page will guide you through the process of registering your app, including choosing which permission your app needs, and specifying an app name.

After creating your app, you're ready to set up the authorization process in your app. The Dropbox SDKs will take care of some of the OAuth 2 process automatically for you, and you can use the tutorials and sample apps for reference.

Testing with a generated access token

If you'd like to test out the Dropbox APIs quickly using your own Dropbox account, you can generate an access token from your newly created app in My apps by pressing the button that says "Generate" in the OAuth 2 section of your app settings page. Keep in

Dropbox authorization

Like Twitter, Dropbox also uses OAuth 2.0. Their documentation includes not just one but two diagrams and an extended explanation of the process.

Activity with authorization

With the open-source project you identified (p. 686), identify the information about authorization for requests to the API. Answer the following questions:

1. What kind of authorization is required to make requests to the API?
2. Are there different access levels within the authorization (for example, free versus pro tiers) that determine how many requests you can make or the types of information you can access?
3. Are you able to get an API key or whatever authorization method is required to make test calls to the API?
4. How is the information about authorization integrated into the getting started tutorial?

API status and error codes

Status and error codes refer to a code number in the response header that indicates the general classification of the response — for example, whether the request was successful (200), resulted in a server error (500), had authorization issues (403), and so on. Standard status codes don't usually need much documentation, but custom status and error codes specific to your API do. Error codes in particular help in troubleshooting bad requests.

Sample status code in curl header

Status codes don't appear in the response body. They appear in the response header, which you might not see by default.

Remember when you submitted the curl call back in Make a curl call (p. 80)? To get the response header, you add `--include` or `-i` to the curl request. If you want *only the response header returned* in the response (and nothing else), capitalize the `-I`, like this:

```
curl -I -X GET "https://api.openweathermap.org/data/2.5/weather?zip=95050&ap
pid=APIKEY&units=imperial"
```

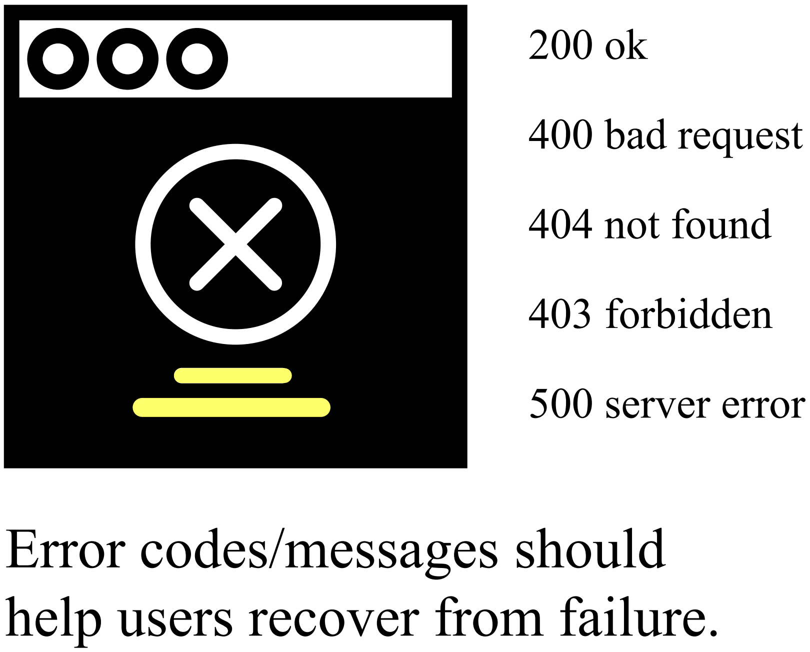
Replace `APIKEY` with your actual API key.

The response header looks as follows:

```
HTTP/1.1 200 OK
Server: openresty
Date: Thu, 06 Dec 2018 22:58:41 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 446
Connection: keep-alive
X-Cache-Key: /data/2.5/weather?units=imperial&zip=95050
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: GET, POST
```

The first line, `HTTP/1.1 200 OK`, tells us the status of the request (`200`). Most REST APIs follow a standard protocol for response headers. For example, `200` isn't just an arbitrary code decided upon by the OpenWeatherMap API developers. `200` is a universally accepted code for a successful HTTP request. (If you change the method, you'll get back a different status code.)

With a GET request, it's pretty easy to tell if the request is successful because you get back the expected response. But suppose you're making a POST, PUT, or DELETE call, where you're changing data contained in the resource. How do you know if the request was successfully processed and received by the API? HTTP response codes in the header of the response will indicate whether the operation was successful. The HTTP status codes are just abbreviations for longer messages.



Status codes are pretty subtle, but when a developer is working with an API, these codes may be the only "interface" the developer has. If you can control the messages the developer sees, it can be a huge win for usability.

All too often, status codes are uninformative, poorly written, and communicate little or no helpful information to the user to overcome the error. Ultimately, status codes should assist users in recovering from errors.

You can see a list of common REST API status codes here and a general list of HTTP status codes here. Although it's probably good to include a few standard status codes, comprehensively documenting *all standard* status codes, especially if rarely triggered by your API, is unnecessary.

Where to list the HTTP response and error codes

Most APIs should have a general page listing response and error codes across the entire API. A standalone page listing the status codes (rather than including these status codes with each endpoint) allows you to expand on each code with more detail without crowding the other documentation. It also reduces redundancy and the sense of information overload.

On the other hand, if some endpoints are prone to triggering certain status and error codes more than others, it makes sense to highlight those status and error codes on same API reference pages. One strategy might be to call attention to any particularly relevant status or error codes for a specific endpoint, and then link to the centralized "Response and Status Codes" page for full information.

Where to get status and error codes

Status and error codes may not be readily apparent when you're documenting your API. You'll probably need to ask developers for a list of all the status and error codes that are unique to your API. Sometimes developers hard-code these status and error codes directly in the programming code and don't have easy ways to hand you a comprehensive list (this makes localization problematic as well).

As a result, you may need to experiment a bit to ferret out all the codes. Specifically, you might need to try to break the API (p. 348) to see all the potential error codes. For example, if you exceed the rate limit (p. 408) for a specific call, the API might return a special error or status code. You would especially need to document this custom code. A troubleshooting section in your API might make special use of the error codes.

How to list status codes

You can list your status and error codes in a basic table or definition list, somewhat like this:

Status code	**Meaning**
200	Successful request and response.
400	Malformed parameters or other bad request.

Status/error codes can assist in troubleshooting

Status and error codes can be particularly helpful when it comes to troubleshooting. As such, you can think of these error codes as complementary to a section on troubleshooting.

Almost every set of documentation could benefit from a section on troubleshooting. In a troubleshooting topic, you document what happens when users get off the happy path and stumble around in the dark forest. Status codes are like the poorly illuminated trail signs that will help users get back onto the right path.

A section on troubleshooting could list error messages related to the following situations:

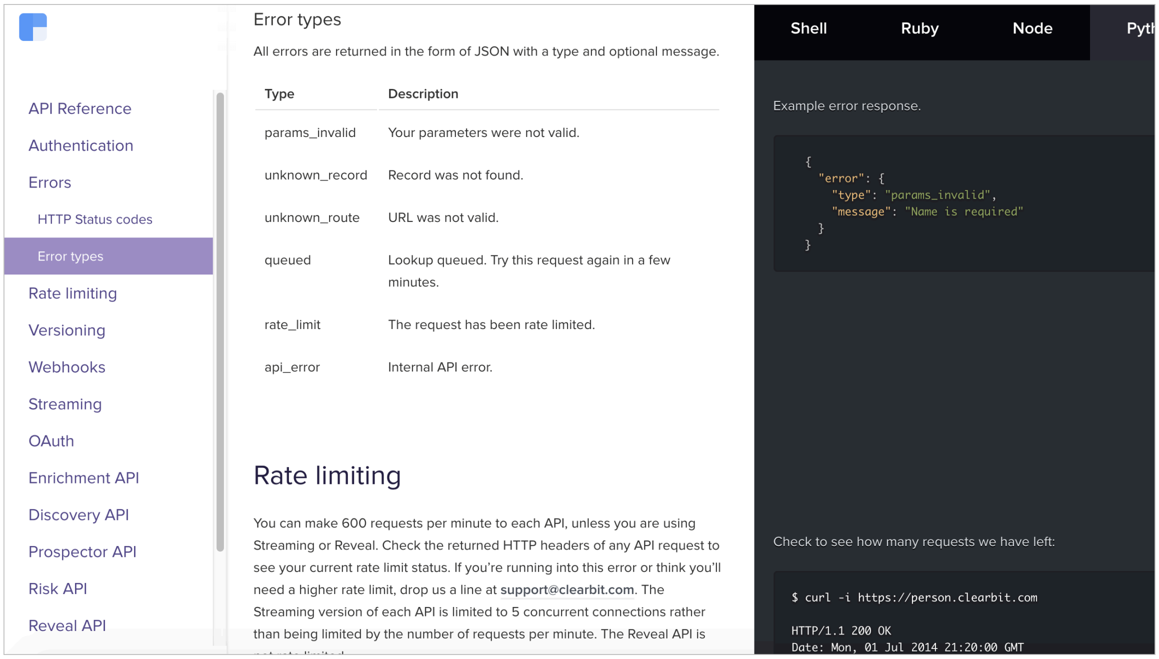
- The wrong API keys are used
- Invalid API keys are used
- The parameters don't fit the data types
- The API throws an exception
- There's no data for the resource to return
- The rate limits have been exceeded
- The parameters are outside the max and min boundaries of what's acceptable
- A required parameter is absent from the endpoint

Where possible, document the exact text of the error in the documentation so that it easily surfaces in searches.

Example of status and error codes

The following are some sample status and error code pages in API documentation.

Context.io



Error types

All errors are returned in the form of JSON with a type and optional message.

Type	Description
params_invalid	Your parameters were not valid.
unknown_record	Record was not found.
unknown_route	URL was not valid.
queued	Lookup queued. Try this request again in a few minutes.
rate_limit	The request has been rate limited.
api_error	Internal API error.

Rate limiting

You can make 600 requests per minute to each API, unless you are using Streaming or Reveal. Check the returned HTTP headers of any API request to see your current rate limit status. If you're running into this error or think you'll need a higher rate limit, drop us a line at **support@clearbit.com**. The Streaming version of each API is limited to 5 concurrent connections rather than being limited by the number of requests per minute. The Reveal API is

Example error response.

```
{
  "error": {
    "type": "params_invalid",
    "message": "Name is required"
  }
}
```

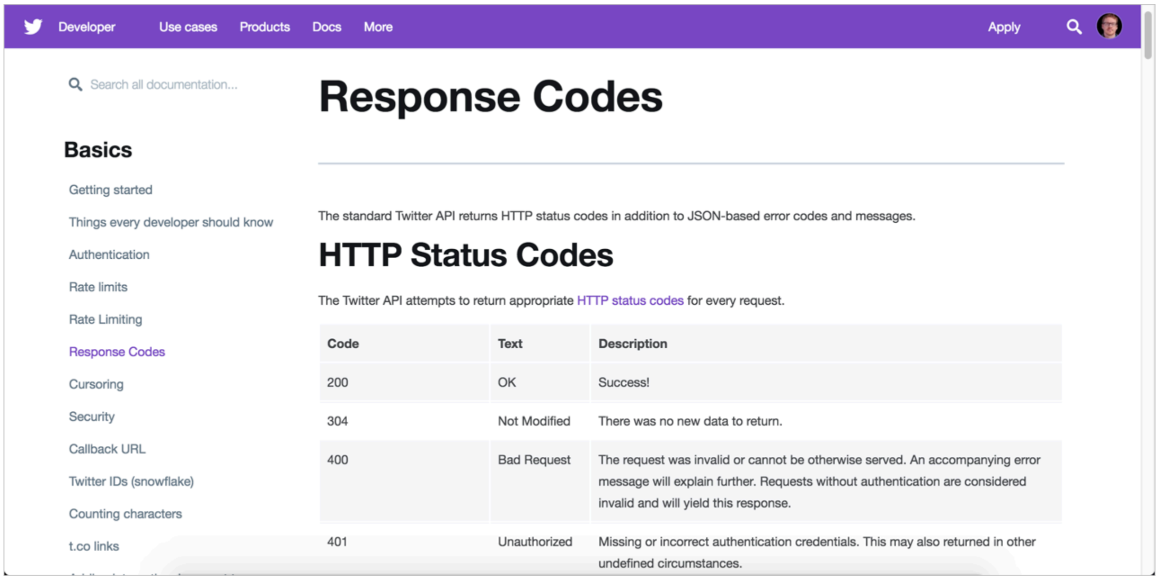
Check to see how many requests we have left:

```
$ curl -i https://person.clearbit.com

HTTP/1.1 200 OK
Date: Mon, 01 Jul 2014 21:20:00 GMT
```

Context.io status and error codes

Clearbit not only documents the standard status codes but also describes the unique parameters returned by their API. Most developers will probably be familiar with 200, 400, and 500 codes, so these codes don't need a lot of explanatory detail. But if your API has unique codes, make sure to describe these adequately.

Twitter



Response Codes

The standard Twitter API returns HTTP status codes in addition to JSON-based error codes and messages.

HTTP Status Codes

The Twitter API attempts to return appropriate HTTP status codes for every request.

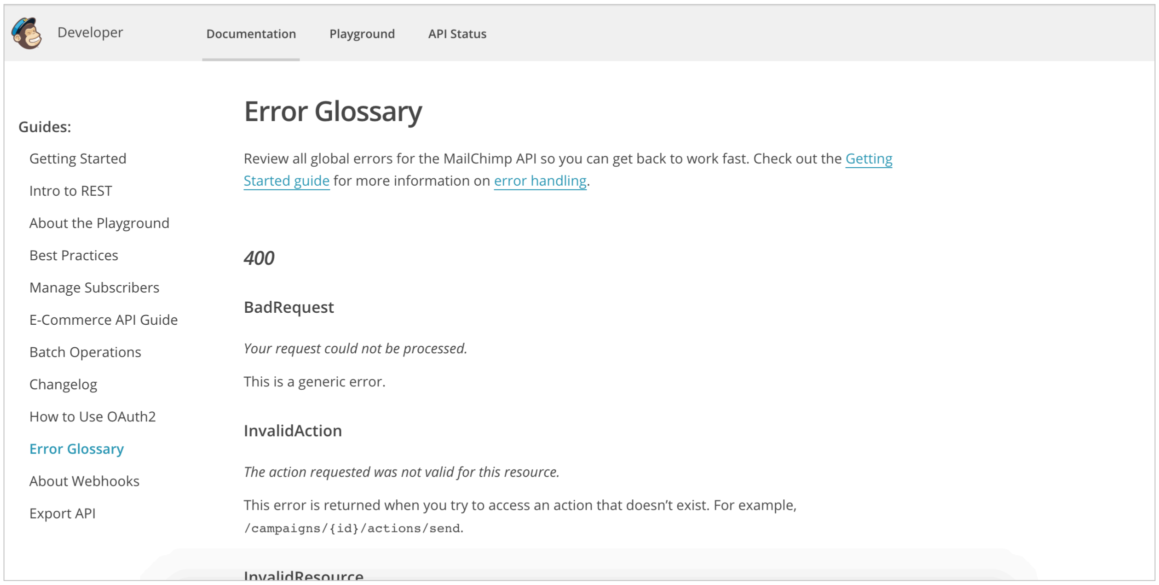
Code	Text	Description
200	OK	Success!
304	Not Modified	There was no new data to return.
400	Bad Request	The request was invalid or cannot be otherwise served. An accompanying error message will explain further. Requests without authentication are considered invalid and will yield this response.
401	Unauthorized	Missing or incorrect authentication credentials. This may also returned in other undefined circumstances.

Twitter status and error codes

With Twitter's status code documentation, they not only describe the code and status but also provide helpful troubleshooting information, potentially assisting with error recovery. For example, with the `500` error, the authors don't just say the status refers to a broken service, they explain, "This is usually a temporary error, for example in a high load situation or if an endpoint is temporarily having issues. Check in the developer forums in case others are having similar issues, or try again later."

This kind of helpful message is what tech writers should aim for with status codes (at least for those codes that indicate problems).

Mailchimp



Error Glossary

Review all global errors for the MailChimp API so you can get back to work fast. Check out the Getting Started guide for more information on error handling.

400

BadRequest

Your request could not be processed.

This is a generic error.

InvalidAction

The action requested was not valid for this resource.

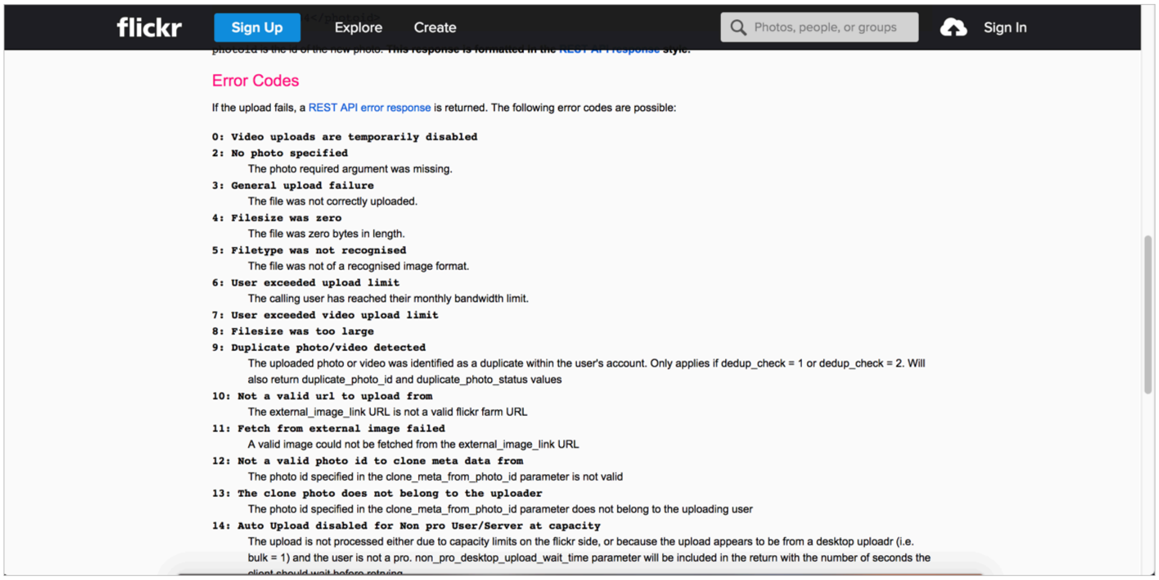
This error is returned when you try to access an action that doesn't exist. For example, `/campaigns/{id}/actions/send`.

InvalidResource

Mailchimp status and error codes

Mailchimp provides readable and friendly descriptions of the error message. For example, with the `403` errors, instead of just writing "Forbidden," Mailchimp explains reasons why you might receive the Forbidden code. With Mailchimp, there are several types of 403 errors. Your request might be forbidden due to a disabled user account or request made to the wrong data center. For the "WrongDataCenter" error, Mailchimp notes that "It's often associated with misconfigured libraries" and they link to more information on data centers. This is the type of error code documentation that is helpful to users.

Flickr



Error Codes

If the upload fails, a REST API error response is returned. The following error codes are possible:

0: Video uploads are temporarily disabled
2: No photo specified
The photo required argument was missing.
3: General upload failure
The file was not correctly uploaded.
4: Filesize was zero
The file was zero bytes in length.
5: Filetype was not recognised
The file was not of a recognised image format.
6: User exceeded upload limit
The calling user has reached their monthly bandwidth limit.
7: User exceeded video upload limit
8: Filesize was too large
9: Duplicate photo/video detected
The uploaded photo or video was identified as a duplicate within the user's account. Only applies if dedup_check = 1 or dedup_check = 2. Will also return duplicate_photo_id and duplicate_photo_status values
10: Not a valid url to upload from
The external_image_link URL is not a valid flickr farm URL
11: Fetch from external image failed
A valid image could not be fetched from the external_image_link URL
12: Not a valid photo id to clone meta data from
The photo id specified in the clone_meta_from_photo_id parameter is not valid
13: The clone photo does not belong to the uploader
The photo id specified in the clone_meta_from_photo_id parameter does not belong to the uploading user
14: Auto Upload disabled for Non pro User/Server at capacity
The upload is not processed either due to capacity limits on the flickr side, or because the upload appears to be from a desktop uploadr (i.e. bulk = 1) and the user is not a pro. non_pro_desktop_upload_wait_time parameter will be included in the return with the number of seconds the

Flickr's status and error codes

With Flickr, the Response Codes section is embedded within each API reference topic. As such, the descriptions are short. While embedding the Response Codes in each topic makes the error codes more visible, in some ways it's less helpful. Because it's embedded within each API topic, the descriptions about the error codes must be brief, or the content would overwhelm the endpoint request information.

In contrast, a standalone page listing error codes allows you to expand on each code with more detail without crowding out the other documentation. The standalone page also reduces redundancy and the appearance of a heavy amount of information (information which is just repeated).

If some endpoints are prone to triggering certain status and error codes more than others, it makes sense to highlight those status and error codes on their relevant API reference pages. I recommend calling attention to any particularly relevant status or error codes on an endpoint's page and then linking to the centralized page for full information.

Activity with status and error codes

With the open-source project you identified (p. 686), identify the status and error code information. Answer the following questions:

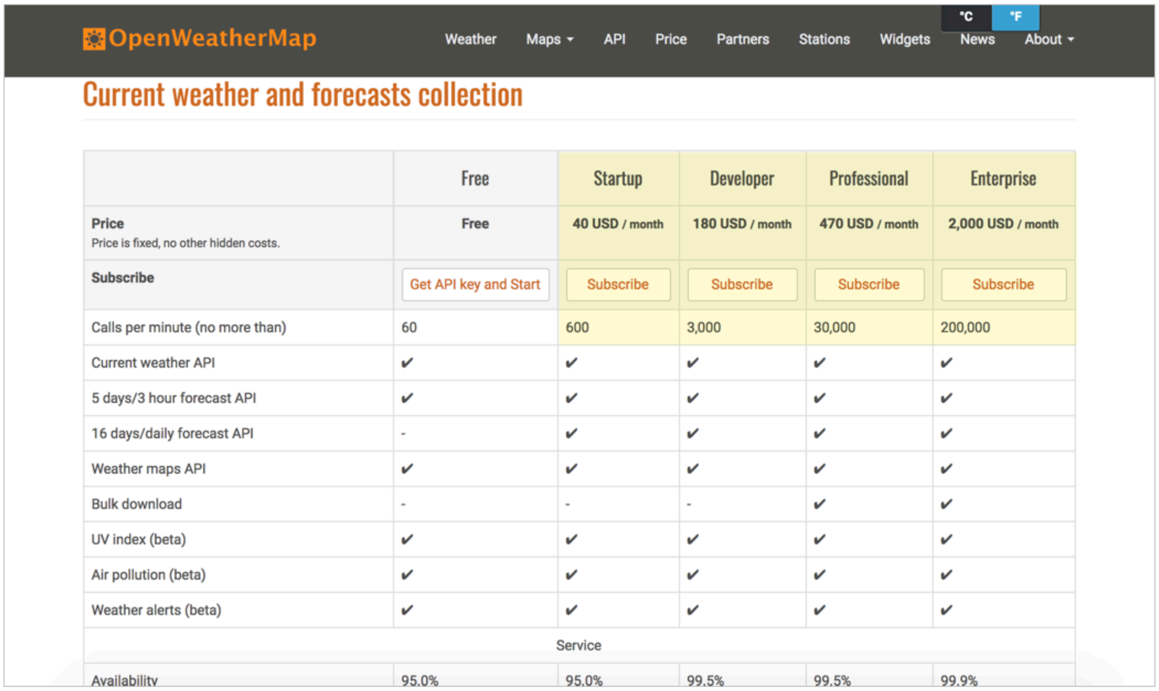
- Does the project describe status and error codes?
- Where is the status and error code information located within the context of the documentation? As a standalone topic? Below each endpoint? Somewhere else?
- Does the API have any unique status and error codes?
- Do the error codes help users recover from errors?
- Make a request to one of the endpoints. Then purposefully change a parameter so that it invalidates the call. What status code gets returned in the response? Is this status code documented?

API rate limiting and thresholds

Rate limits determine how frequently you can call a particular endpoint. Usually, companies have different tiers (for example, free versus pro) and licenses (open-source, business, commercial) corresponding to different capabilities or rate limits with the API.

What to cover with rate limiting

Companies with APIs make money by charging for access to the API, but they usually distinguish between low usage and high usage, often making the low usage options free so that developers can explore and experiment with the API. With the sample OpenWeatherMap Weather API that we've been using in this course, you can see where the pricing tier begins:



Current weather and forecasts collection

	Free	Startup	Developer	Professional	Enterprise
Price Price is fixed, no other hidden costs.	**Free**	**40 USD** / month	**180 USD** / month	**470 USD** / month	**2,000 USD** / month
Subscribe	Get API key and Start	Subscribe	Subscribe	Subscribe	Subscribe
Calls per minute (no more than)	60	600	3,000	30,000	200,000
Current weather API	✔	✔	✔	✔	✔
5 days/3 hour forecast API	✔	✔	✔	✔	✔
16 days/daily forecast API	-	✔	✔	✔	✔
Weather maps API	✔	✔	✔	✔	✔
Bulk download	-	-	-	✔	✔
UV index (beta)	✔	✔	✔	✔	✔
Air pollution (beta)	✔	✔	✔	✔	✔
Weather alerts (beta)	✔	✔	✔	✔	✔
Service					
Availability	95.0%	95.0%	99.5%	99.5%	99.9%

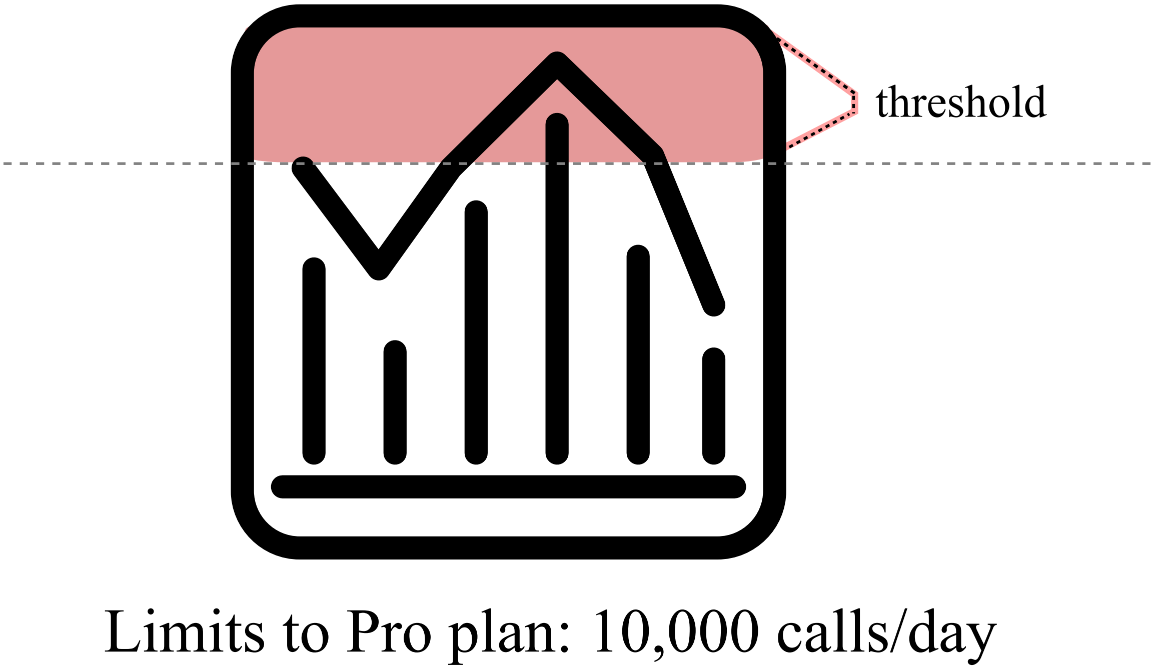
Pricing tiers for OpenWeatherMap API. Each call is a request to the API. If your page makes just one call for weather, and you get more than 60 visitors per second, you'll need to move past the free tier.

If your site has hundreds of thousands of visitors a day, and with each page reload calls an API endpoint, you want to be sure the API can support that kind of traffic.

Pricing related to rate limiting is probably information that's within the marketing domain rather than documentation domain. However, developers will still want to know a few key behaviors around the rate-limiting thresholds. For example:

- When you exceed the threshold, do your calls get throttled with slower responses?
- Do you get overcharges for every extra call?
- If you exceed the limit, do the responses return a particular status code (if so, which one)?

Also, when developers implement the code into their applications, how are they handling situations where the API doesn't respond due to rate limit violations? Are there conditions and checks to handle these throttled scenarios? Does the widget (or whatever might be implementing the API) freeze or hang, display empty or crash?

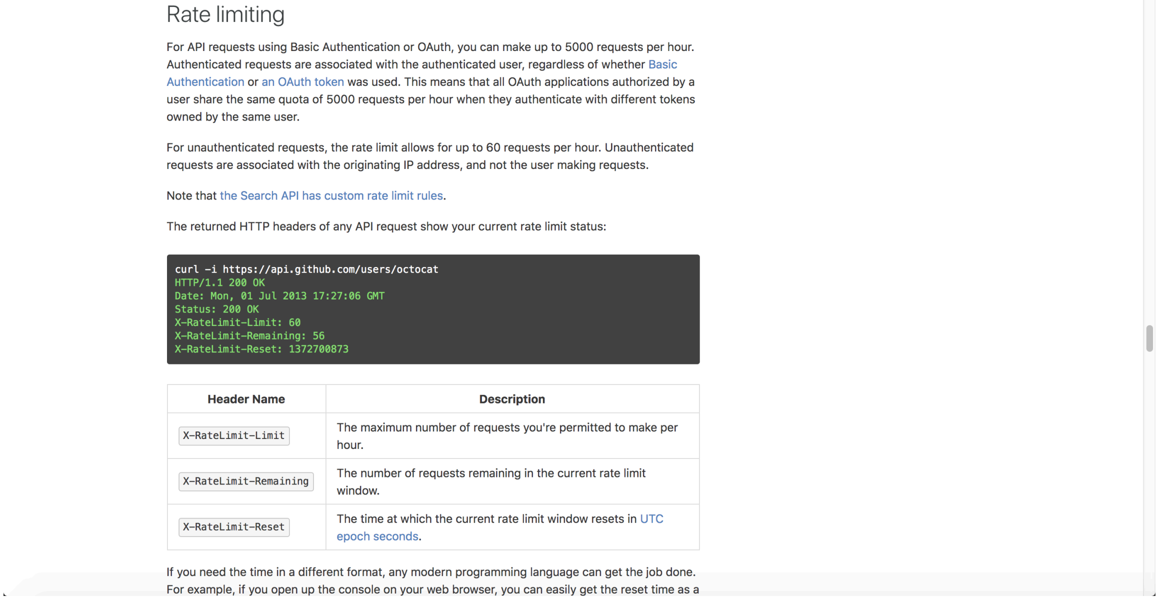


Rate limiting might seem like a marketing topic, but the rate limiting policies and how they affect API calls can have a significant impact on development.

Examples rate limiting sections

Here are a few examples of rate limiting sections in API documentation.

GitHub



Rate limiting

For API requests using Basic Authentication or OAuth, you can make up to 5000 requests per hour. Authenticated requests are associated with the authenticated user, regardless of whether Basic Authentication or an OAuth token was used. This means that all OAuth applications authorized by a user share the same quota of 5000 requests per hour when they authenticate with different tokens owned by the same user.

For unauthenticated requests, the rate limit allows for up to 60 requests per hour. Unauthenticated requests are associated with the originating IP address, and not the user making requests.

Note that the Search API has custom rate limit rules.

The returned HTTP headers of any API request show your current rate limit status:

```
curl -i https://api.github.com/users/octocat
HTTP/1.1 200 OK
Date: Mon, 01 Jul 2013 17:27:06 GMT
Status: 200 OK
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 56
X-RateLimit-Reset: 1372700873
```

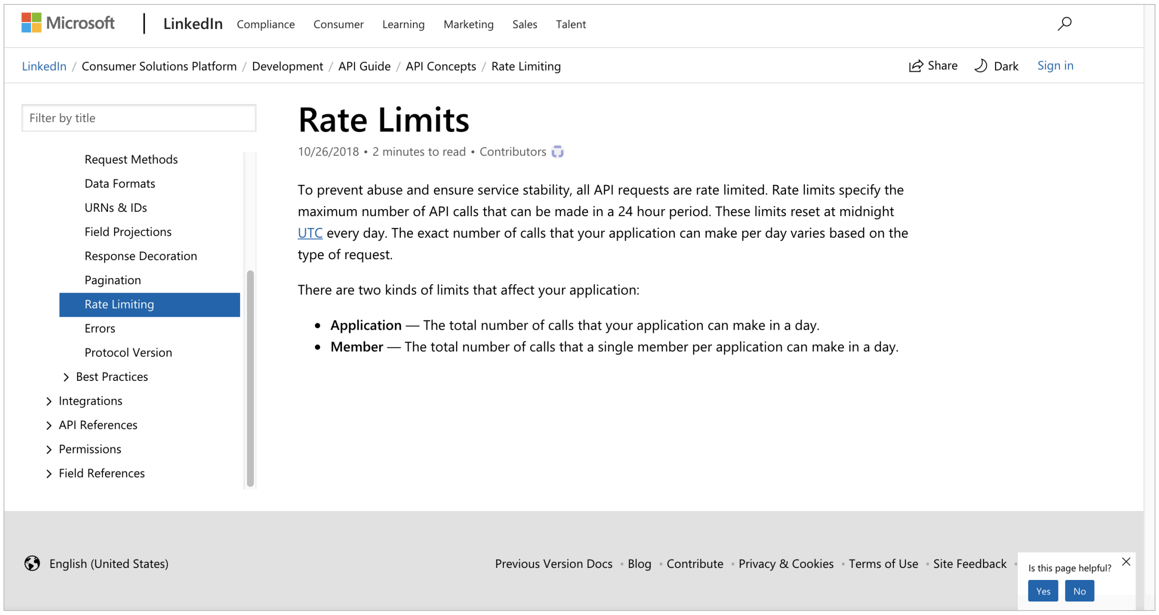
Header Name	Description
`X-RateLimit-Limit`	The maximum number of requests you're permitted to make per hour.
`X-RateLimit-Remaining`	The number of requests remaining in the current rate limit window.
`X-RateLimit-Reset`	The time at which the current rate limit window resets in UTC epoch seconds.

If you need the time in a different format, any modern programming language can get the job done. For example, if you open up the console on your web browser, you can easily get the reset time as a

GitHub rate limiting

GitHub's documentation explains rate limits for authenticated versus unauthenticated requests, the header returned, the meaning of the rate-limiting titles (`X-RateLimit-Limit`, `X-RateLimit-Remaining`, and `X-RateLimit-Reset`), how to check your current usage, how to increase rate limits for a specific application, what happens when rate limits are abused, and more.

Linkedin



Rate Limits

10/26/2018 • 2 minutes to read • Contributors

To prevent abuse and ensure service stability, all API requests are rate limited. Rate limits specify the maximum number of API calls that can be made in a 24 hour period. These limits reset at midnight UTC every day. The exact number of calls that your application can make per day varies based on the type of request.

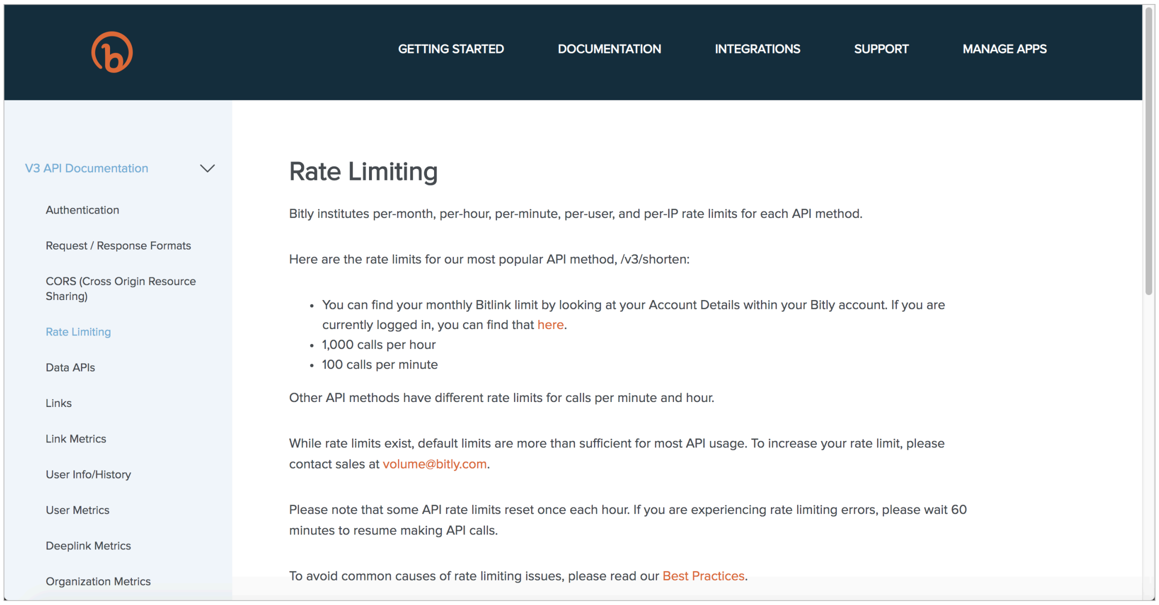
There are two kinds of limits that affect your application:

- **Application** — The total number of calls that your application can make in a day.
- **Member** — The total number of calls that a single member per application can make in a day.

Linkedin rate throttling section

Linkedin’s rate limiting documentation explains that different API endpoints have different limits. There are three different types of throttling: Application throttling, User throttling, and Developer throttling. Their documentation also explains the time zone used to track the day’s beginning and end.

Bitly



Rate Limiting

Bitly institutes per-month, per-hour, per-minute, per-user, and per-IP rate limits for each API method.

Here are the rate limits for our most popular API method, /v3/shorten:

- You can find your monthly Bitlink limit by looking at your Account Details within your Bitly account. If you are currently logged in, you can find that here.
- 1,000 calls per hour
- 100 calls per minute

Other API methods have different rate limits for calls per minute and hour.

While rate limits exist, default limits are more than sufficient for most API usage. To increase your rate limit, please contact sales at volume@bitly.com.

Please note that some API rate limits reset once each hour. If you are experiencing rate limiting errors, please wait 60 minutes to resume making API calls.

To avoid common causes of rate limiting issues, please read our Best Practices.

Bitly's rate limiting

Bitly provides basic information on the page above but also links to best practices for avoiding rate-limiting issues. These best practices include tips such as caching, security issues, long page loads, batch processing, high-volume requests, URL encoding, and more.

By looking at these examples, you can see that while rate limiting might seem like a straightforward topic, there are layers of depth and complexity to cover. The relevance of the topic depends on your API and the rate-limiting policies your company sets, but this information cannot be entirely offloaded to Marketing to handle. So much of the information around rate limiting directly affects development.

Activity with rate limits

With the open-source project you identified (p. 686), identify the information about rate limits with the API. Answer the following questions:

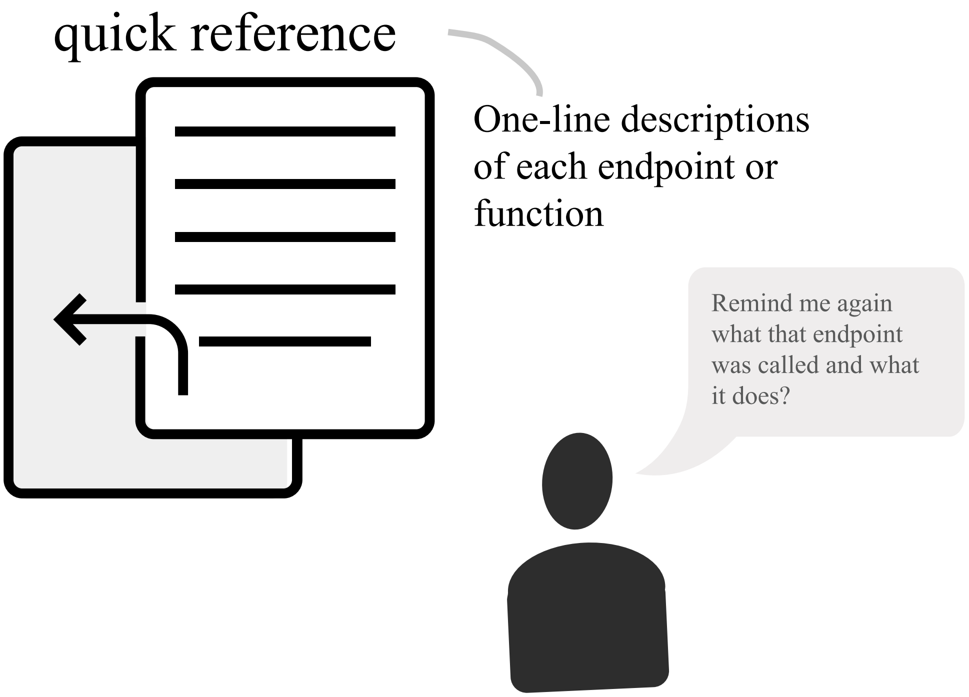
1. Does the API have rate limits?
2. What happens if users exceed the rate limits?
3. What status codes are sent if rate limits are exceeded?
4. What kind of rate limits exist with the free (or development) tiers of the API?

API quick reference

Quick reference guides serve a different function than getting started tutorials (p. 379). While the getting started tutorial helps beginners get oriented by providing an end-to-end instruction to make a simple API request, the quick reference guide helps users get a glimpse of the system as a whole, often by providing a list of the API's endpoints.

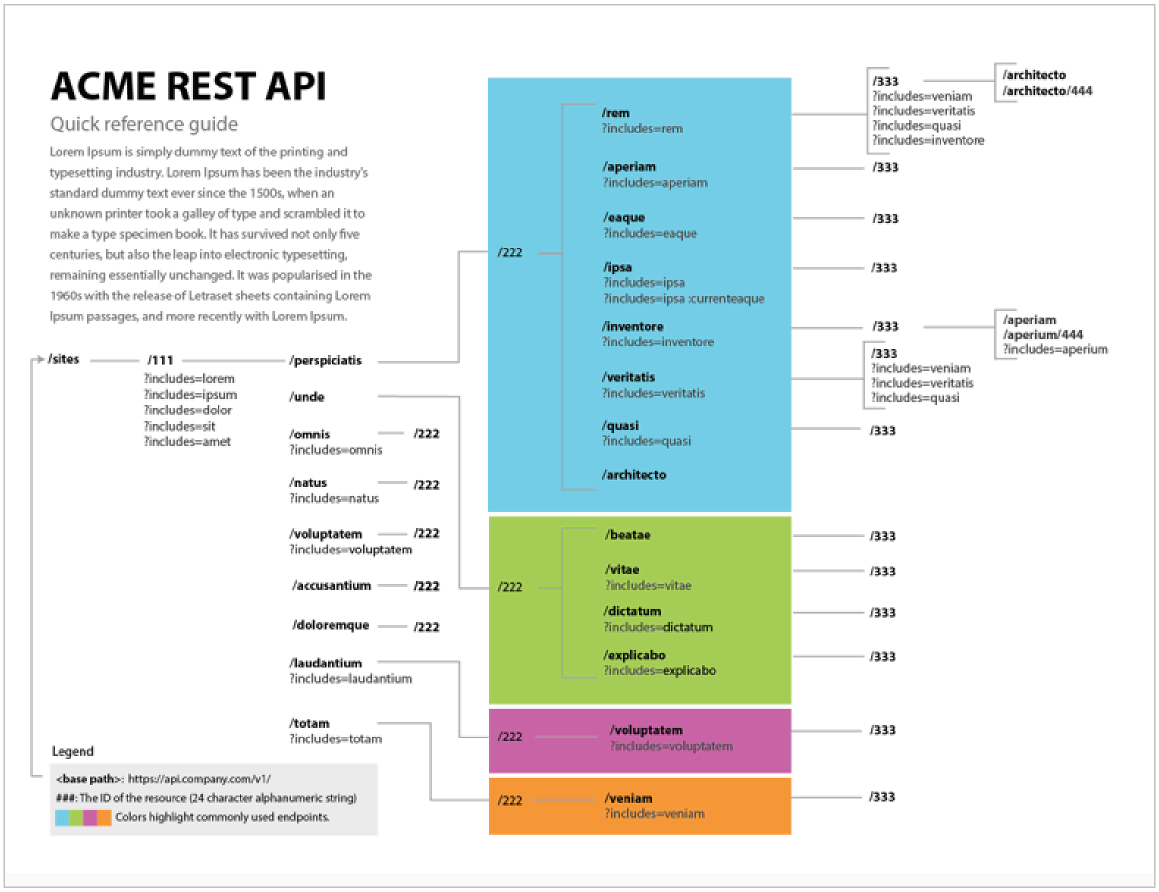
The need for quick reference guides

Whether for end-user documentation or developer documentation, the quick reference guide provides a 1-2-page guide that provides a brief summary of the core tasks and features in the system.


quick reference
One-line descriptions of each endpoint or function
Remind me again what that endpoint was called and what it does?

Quick reference guides compress the key information into a brief format for easy consumption

The quick reference guide should provide the user with just enough information to get the gist of what the system is about, including the key endpoints and tasks. Often with APIs, the endpoints have relationships with each other that you can depict visually. Here's an API diagram I created at a previous company:



ACME REST API

Quick reference guide

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with Lorem Ipsum.

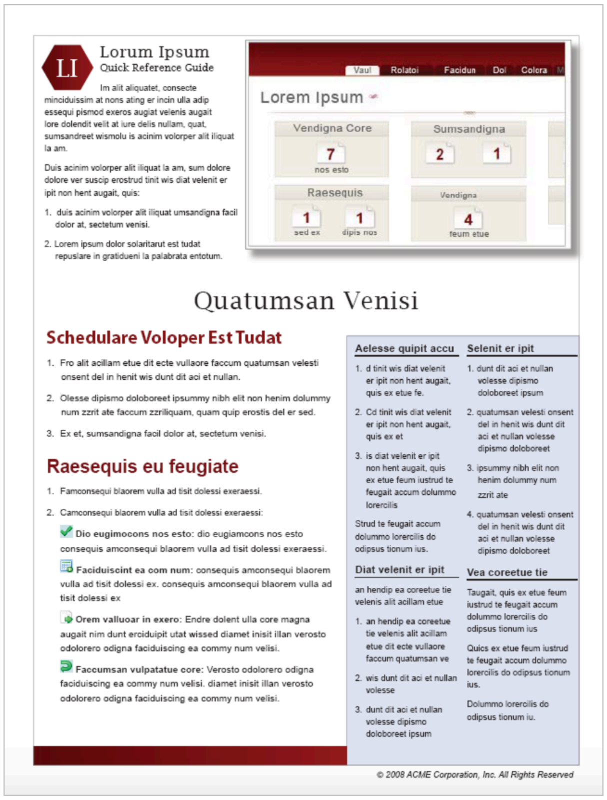
Legend

<base path>: https://api.company.com/v1/
###: The ID of the resource (24 character alphanumeric string)
🟦🟩🟪🟧 Colors highlight commonly used endpoints.

A quick reference guide format

The text is Latin filler for privacy reasons, so the logic may not be entirely apparent. But with this API, the endpoints could be organized into different groups. Some of the groups had multiple levels within the endpoint, and multiple include options for each endpoint. I created this diagram in Adobe Illustrator and distributed it as a PDF. Developers found it useful because it tried to make sense of the API as a whole, showing how all the endpoints fit together in logical harmony. Most commonly with API documentation, the quick reference guide lists abbreviated descriptions of the endpoints. For this reason, the Swagger UI output (p. 240) can often function as a quick reference guide.

Outside of API documentation, quick reference guides tend to focus more on tasks. If you have a service to set up or configure, a more narrative rather than visual format might make sense. Here's a sample layout for such a guide:



Lorum Ipsum
Quick Reference Guide

Im alit aliquatet, consecte minciduissim at nons ating er incin ulla adip essequi pismod exeros augiat velenis augait lore dolendit velit at iure delis nullam, quat, sumsandreet wismolu is acinim volorper alit iliquat la am.

Duis acinim volorper alit iliquat la am, sum dolore dolore ver suscip erostrud tinit wis diat velenit er ipit non hent augait, quis:

1. duis acinim volorper alit iliquat umsandigna facil dolor at, sectetum venisi.
2. Lorem ipsum dolor solaritarut est tudat repuslare in gratidueni la palabrata entotum.

Quatumsan Venisi

Schedulare Voloper Est Tudat

1. Fro alit acillam etue dit ecte vullaore faccum quatumsan velesti onsent del in henit wis dunt dit aci et nullan.
2. Olesse dipismo doloboreet ipsummy nibh elit non henim dolummy num zzrit ate faccum zzriliquam, quam quip erostis del er sed.
3. Ex et, sumsandigna facil dolor at, sectetum venisi.

Raesequis eu feugiate

1. Famconsequi blaorem vulla ad tisit dolessi exeraessi.
2. Camconsequi blaorem vulla ad tisit dolessi exeraessi:
   - ✓ **Dio eugimocons nos esto:** dio eugiamcons nos esto consequis amconsequi blaorem vulla ad tisit dolessi exeraessi.
   - **Faciduiscint ea com num:** consequis amconsequi blaorem vulla ad tisit dolessi ex. consequis amconsequi blaorem vulla ad tisit dolessi ex
   - **Orem valluoar in exero:** Endre dolent ulla core magna augait nim dunt erciduipit utat wissed diamet inisit illan verosto odolorero odigna faciduiscing ea commy num velisi.
   - **Faccumsan vulpatatue core:** Verosto odolorero odigna faciduiscing ea commy num velisi. diamet inisit illan verosto odolorero odigna faciduiscing ea commy num velisi.

Aelesse quipit accu

1. d tinit wis diat velenit er ipit non hent augait, quis ex etue fe.
2. Cd tinit wis diat velenit er ipit non hent augait, quis ex et
3. is diat velenit er ipit non hent augait, quis ex etue feum iustrud te feugait accum dolummo lorercilis

Strud te feugait accum dolummo lorercilis do odipsus tionum ius.

Diat velenit er ipit

an hendip ea coreetue tie velenis alit acillam etue

1. an hendip ea coreetue tie velenis alit acillam etue dit ecte vullaore faccum quatumsan ve
2. wis dunt dit aci et nullan volesse
3. dunt dit aci et nullan volesse dipismo doloboreet ipsum

Selenit er ipit

1. dunt dit aci et nullan volesse dipismo doloboreet ipsum
2. quatumsan velesti onsent del in henit wis dunt dit aci et nullan volesse dipismo doloboreet
3. ipsummy nibh elit non henim dolummy num zzrit ate
4. quatumsan velesti onsent del in henit wis dunt dit aci et nullan volesse dipismo doloboreet

Vea coreetue tie

Taugait, quis ex etue feum iustrud te feugait accum dolummo lorercilis do odipsus tionum ius

Quics ex etue feum iustrud te feugait accum dolummo lorercilis do odipsus tionum ius.

Dolummo lorercilis do odipsus tionum iu.

© 2008 ACME Corporation, Inc. All Rights Reserved

This quick reference guide format focuses more on tasks than API endpoints

However, with API documentation, usually the quick reference guide focuses on some visual grouping or display of the endpoints since this is what constitutes the core functionality in an API.

Advantages of distilled information for learning

The information in the quick reference usually can’t be single sourced, since it’s not just an excerpt from the docs but rather a more briefly written summary or depiction of the entire system. As a result, many times it seems like yet another deliverable technical writers don’t have time to write. But for the best user experience, the quick reference guide shouldn’t be skipped because it provides incalculable value to users.

When you create the quick reference guide, try to condense the most important information into one or two pages that users can print and pin up on their wall. By “condense” I don’t mean shrink the font to 6-point, decrease the leading, and eliminate all white space. With the quick reference guide, you take something that’s robust and complex, and distill it down to its essence in a way that still maintains clarity to users.

Through this distillation, quick reference guides provide a unique advantage for users to understand the material. Providing a high-level overview of a system helps users get a sense of the whole before drilling into the details.

For a deep dive into the importance of distilling information for users, see Reduction, layering, and distillation as a strategy for simplicity in my series on Simplifying Complexity. Reading overviews, summaries, and other high-level information to see the whole at a glance can help users understand a complex system in significant ways. Too often, users get thrown into the technical details without more grounding and orientation about the whole.

Distilling large amounts of information into concisely worded titles, summaries, headings, mini-TOCs, and topic sentences can facilitate information consumption and comprehension. Quick reference guides take the principle of distillation to another level by compressing the whole system into a bite-sized information deliverable.

Quick reference guides are like the poetry of technical writing. The goal is not just to be brief or concise. With poetry, the poet attempts to evoke a mood or paint a moment, and in that brief moment, capture the essence of the whole. Writing a quick reference guide involves much the same effort. It's not that you merely cut words to make the documentation shorter, or restrict the output to a few topics, but that you try to compress the documentation as a whole and express its minimalist equivalent.

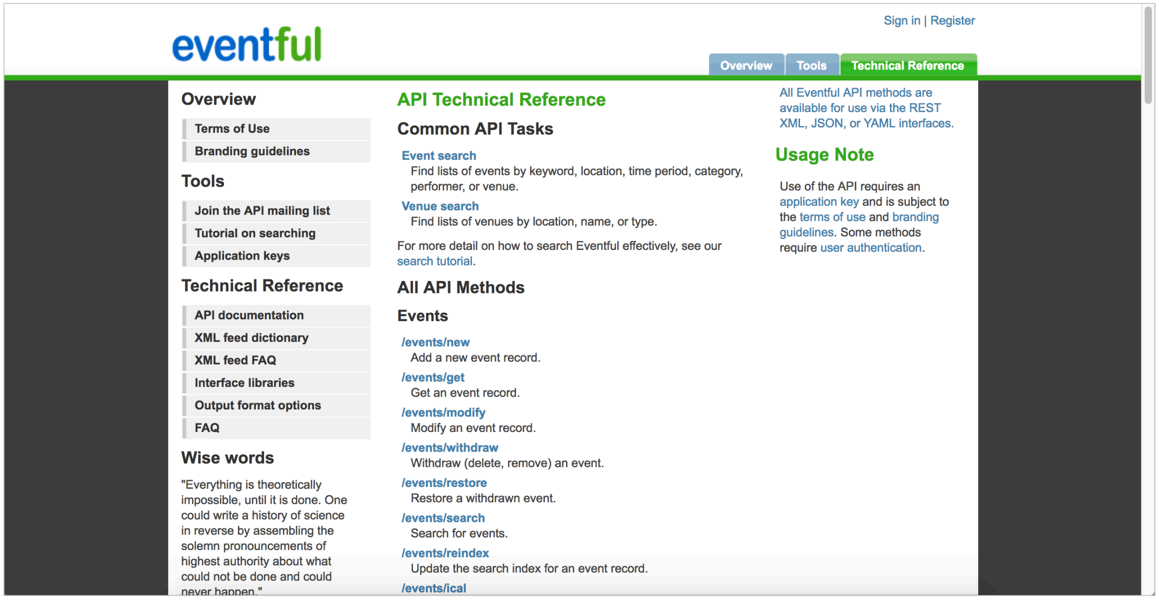
I'll grant that the task is probably impossible for technical material. Still, the attempt is worthwhile and the philosophy remains the same. Quick reference guides teach each us how to use the system in 5 minutes rather than 5 hours. It's a philosophy of simplification and linguistic efficiency.

Don't be deceived by the brevity and scope of the quick reference guide. In wrangling with layout, scope, and concision to create this guide, you might spend several days writing just one page. But when you're done, you can practically frame the result.

Sample quick reference guides

The following are sample quick reference guides from various API documentation sites.

Eventful



Wise words

"Everything is theoretically impossible, until it is done. One could write a history of science in reverse by assembling the solemn pronouncements of highest authority about what could not be done and could never happen."

API Technical Reference

Common API Tasks

Event search
Find lists of events by keyword, location, time period, category, performer, or venue.

Venue search
Find lists of venues by location, name, or type.

For more detail on how to search Eventful effectively, see our search tutorial.

All API Methods

Events

/events/new
Add a new event record.

/events/get
Get an event record.

/events/modify
Modify an event record.

/events/withdraw
Withdraw (delete, remove) an event.

/events/restore
Restore a withdrawn event.

/events/search
Search for events.

/events/reindex
Update the search index for an event record.

/events/ical

All Eventful API methods are available for use via the REST XML, JSON, or YAML interfaces.

Usage Note

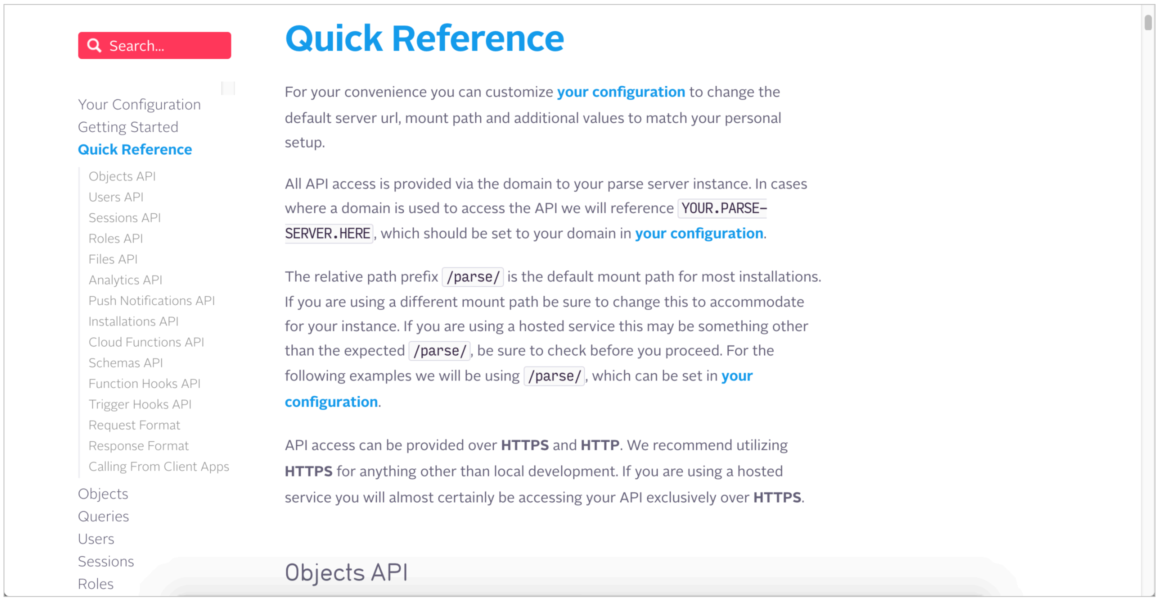
Use of the API requires an application key and is subject to the terms of use and branding guidelines. Some methods require user authentication.

Eventful quick reference guide

Eventful provides a one-page quick list of all the endpoints in the API, organized by resource group. Each endpoint is described in about half a line, so you can get a gist of them all quickly. For example, the description for the `/events/get` in their quick reference is "Get an event record." But if you click for more details, the more detailed definition is "Given an event ID, returns the event data associated with that event. See http://eventful.com/events/E0-001-000278174-6 for an example interface."

There’s a certain understanding you get from seeing all the endpoints at a glance. By looking from high-above at the forest, you can see the shape of the forest as a whole. You may not know what kinds of trees the forest contains, but you can comprehend other details that aren’t apparent when you’re looking at a single tree.

Parse



Quick Reference

For your convenience you can customize **your configuration** to change the default server url, mount path and additional values to match your personal setup.

All API access is provided via the domain to your parse server instance. In cases where a domain is used to access the API we will reference `YOUR.PARSE-SERVER.HERE`, which should be set to your domain in **your configuration**.

The relative path prefix `/parse/` is the default mount path for most installations. If you are using a different mount path be sure to change this to accommodate for your instance. If you are using a hosted service this may be something other than the expected `/parse/`, be sure to check before you proceed. For the following examples we will be using `/parse/`, which can be set in **your configuration**.

API access can be provided over **HTTPS** and **HTTP**. We recommend utilizing **HTTPS** for anything other than local development. If you are using a hosted service you will almost certainly be accessing your API exclusively over **HTTPS**.

Objects API

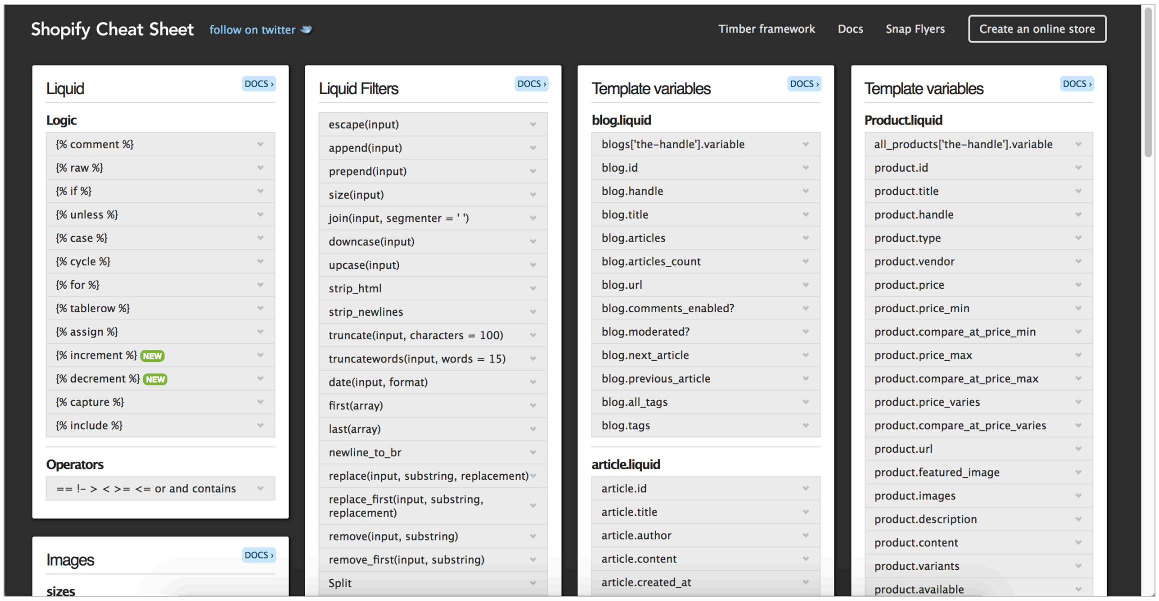
Parse quick reference guide

The quick reference for Parse is similar to Eventful in that it’s a long list of endpoints, this time grouped in tables. Notice that this quick reference page is just a section within one long, single page of docs. In their approach, all documentation is on the same page, but as you scroll down, different entries in the sidebar highlight.

Sometimes developers like the one-page approach because it reduces information fragmentation and lets them use Ctrl+F to find all instances of a keyword. I explored the tradeoffs in this one-page approach in Single-page docs versus “Click Insanity.”

If you use the OpenAPI reference docs on GitHub, you’ll notice the docs are also contained on a single page. Developers might like to use Ctrl+F to quickly see all instances of a topic. However, I’m not a fan of single-page documentation like this because it provides a lot of visual complexity for users to sort out.

Shopify



Liquid

Logic

{% comment %}
{% raw %}
{% if %}
{% unless %}
{% case %}
{% cycle %}
{% for %}
{% tablerow %}
{% assign %}
{% increment %}
{% decrement %}
{% capture %}
{% include %}

Operators

== != > < >= <= or and contains

Images

sizes

Liquid Filters

escape(input)
append(input)
prepend(input)
size(input)
join(input, segmenter = ' ')
downcase(input)
upcase(input)
strip_html
strip_newlines
truncate(input, characters = 100)
truncatewords(input, words = 15)
date(input, format)
first(array)
last(array)
newline_to_br
replace(input, substring, replacement)
replace_first(input, substring, replacement)
remove(input, substring)
remove_first(input, substring)
Split

Template variables

blog.liquid

blogs['the-handle'].variable
blog.id
blog.handle
blog.title
blog.articles
blog.articles_count
blog.url
blog.comments_enabled?
blog.moderated?
blog.next_article
blog.previous_article
blog.all_tags
blog.tags

article.liquid

article.id
article.title
article.author
article.content
article.created_at

Template variables

Product.liquid

all_products['the-handle'].variable
product.id
product.title
product.handle
product.type
product.vendor
product.price
product.price_min
product.compare_at_price_min
product.price_max
product.compare_at_price_max
product.price_varies
product.compare_at_price_varies
product.url
product.featured_image
product.images
product.description
product.content
product.variants
product.available

Shopify quick reference guide

The Shopify quick reference guide isn't for an API, but it does show the filters, variables, and other functions available in Liquid, which is a scripting language for developers. Here Shopify takes advantage of collapse-and-expand functionality to compress the information.

This quick reference guide is handy because it lets you browse all the available functionality in Liquid at once, so you can know what to dive into for more information. It's like a map of the Liquid terrain. The map lets you know all the functions that exist.

Activity with quick reference guides

With the open-source project you identified (p. 686), identify the information about quick reference guides for the API. Answer the following questions:

1. Is there a quick reference guide for the API? Perhaps a quick list of the API endpoints?
2. Is there a Swagger UI output that acts as a quick reference for the API?
3. If there isn't a quick reference guide, would the API benefit from one? Why or why not?
4. Besides listing abbreviated descriptions of the endpoints, what else would you put in the API's quick reference guide? Common tasks?
5. Are there several essential tasks users need to perform with the API? Are these core tasks also communicated in the getting started tutorial?

API glossary

The glossary defines all the terms that might be helpful for users to know — and especially all terms unique to your company or product. Glossaries are often overlooked or skipped, but their importance should not be understated since much of the user's understanding of API documentation depends on the clarity and alignment of specific terms. A glossary will help you, as a writer, be more precise about terminology. Additionally, sourcing definitions from a central glossary file can make authoring more efficient and consistent.

Defining specialized terms

Unlike most other professional writing disciplines, tech docs are notorious for the number of specialized terms in their content. We have unique terms specific to our company as well as industry and business domain jargon.

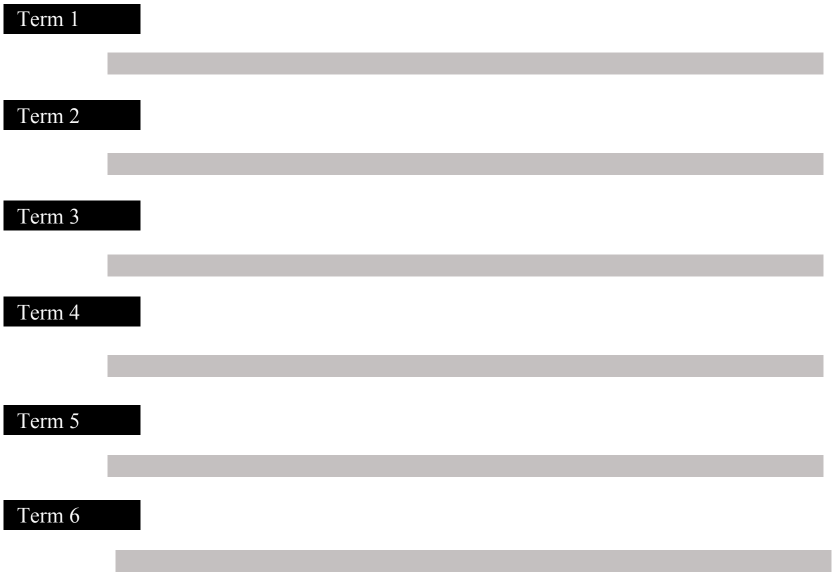
API evangelist Kin Lane noted his frustration with an API's language when he encountered an undefined acronym in some API docs he was browsing. Lane explains,

> I came across a set of API resources for managing a DEG the other day. You could add, updated, delete and get DEGs. You can also pull analytics, history, and other elements of a DEG. I spent about 10-15 minutes looking around their developer portal, documentation, and even Googling, but never could figure out what a DEG was. Nowhere in their documentation did they ever tell consumers what a DEG was, you just had to be in the know I guess. The API designer (if that occurred) and developer had never stopped to consider that maybe someone would stumble across their very public API and not know what a DEG was. (Using Plain Language In Your API Paths ↗)

In this case, DEG must have been an acronym specific to the developer's API. In these cases, there's no excuse for not defining your own pet acronyms and unique vocabulary.

But many times, acronyms and unfamiliar terms are an unavoidable part of a specialized domain. As a technical writer, using the correct terms for your knowledge domain and your product is necessary, and those terms are often appropriate for that particular knowledge domain.

To simplify the language, you can't just omit the necessary terminology for the domain and substitute in more friendly names. You have to teach the user the right language so they can participate in the conversation. One approach for teaching users how to speak this language involves defining unfamiliar words for the user by way of a glossary.



Glossaries not only provide clarity about terms for users but also help the writers think more clearly and consistently about the topic.

I provide an API glossary here (p. 55) that defines many terms related to API documentation. If you run into unfamiliar jargon that I don't define in the topic, you can always consult the glossary.

Should you provide a glossary?

Usually, providing a glossary in your documentation seems like going above and beyond the call of duty. However, defining these terms has tremendous benefit for the writer too, not just for the users.

Defining terms helps you recognize the specialized terminology that might be used with your API — terms that you may have become immune to by hearing them so frequently in project meetings. By defining the terms, you ensure that you're using these terms more precisely and accurately (rather than switching around with synonyms for variety).

Localization requirements and glossaries

If you're planning to localize your API documentation, most translators will require a glossary. Especially with API documentation, there are many technical terms that translators need to to be aware of so they can decide whether the terms should be translated.

For example, suppose in your sample Android app, you have terms like `PlaybackState` and "media session" and "callback" and `BroadcastReceiver`. Translators need to be technical enough to know whether these terms should be translated or left as is. Terms set off in `code` tags normally wouldn't be translated, but many other terms referring to technical elements might also best be left untranslated. It can be a tricky judgment call when the technical term isn't set off as code but refers to a technical concept or class (such as "MediaSession").

After translators finish translating the content, the translation needs to be checked over by a subject matter expert in that language — usually by a field engineer who works with clients in that locale. The glossary will also assist the subject matter experts who review the translation.

Specialized versus non-specialized terms

One challenge in coming up with a glossary is distinguishing between specialized terms unique to your API and terms that are common within the industry or knowledge domain. You don't need to define terms that are common within the knowledge domain you're working in. For example, with a sample Android app, you wouldn't need to define adb (Android Debug Bridge) in your glossary because most anyone who develops on Android would presumably already be familiar with adb.

However, if you think listing some industry terms in the glossary would be helpful for users, who might need a little more information, you can do so. Glossaries can easily accommodate dozens or even hundreds of terms. In your definitions, you can also include links to external sites where more information is available.

Separating your API's terminology from the general knowledge domain gets at the heart of why developer documentation can be so challenging for technical writers (who often aren't experts in the knowledge domain). You'll likely need to rely on engineers to help identify terms that need to be defined.

I explored glossaries in depth in Reducing the complexity of technical language ↗. As a best practice, you can get a good sense as to whether you're using the right terms in a standard way by reading your competitor's documentation.

Integration of the glossary with documentation

One question to consider is how and where to integrate the glossary definition within your technical content. For example, suppose you have the terms "near field" and "far field" in your content, referring to voice interactions with a device. You might use these terms in a number of sections and different pages. Sure, you could define these terms the first instance in your docs when you use them, but what if they appear on half a dozen pages? Users might not start on the initial page where they're defined.

You could incorporate tooltips (such as these tooltips from Bootstrap ↗) over the term in places where you don't define it in-context. You could also link to the glossary definition rather than continually incorporating tooltips.

One principle to consider with connecting terms to glossary definitions is Fitt's Law, which asserts that "the distance between a user's task/attention area and the task-related button should be kept as short as possible" (What is Fitts' Law? ↗). In other words, the more distance between your glossary definition and the term, the harder it is for users to locate the definition. Users have to leave their context to go hunting for the definition, and then return. This context switching is often skipped by users because it's a drag on their efficiency.

In the worst case scenario, the term might not have a tooltip or link, and the glossary page link might not be visible at all in the page's context. In this case, it becomes even harder for a user to locate the definition, requiring more time out of context. In the best case scenario, the term is defined within the context and flow of the sentence; or it has a tooltip that, on mouseover, shows the definition. The tradeoff of the latter scenario, however, is that making too many words styled into tooltip links distracts the user. Each link the user sees presents the user with the question of whether to click the link, which provides more cognitive load on the reading experience. If you do use tooltips, style them in a way that distinguishes them from regular hyperlinks.

Reusing glossary definitions

To reuse glossary definitions, consider storing glossary terms and definitions in a structured file that acts like a database for glossary terms. For example, on my glossary page (p. 55), the data source is a YAML file here: _data/glossary.yml. Each definition follows a structure like this:

```
api:
  term: API
  def: "Application Programming Interface. Enables different systems to interact with each other programmatically. Two types of APIs are REST APIs (web APIs) and native-library APIs."
  link: See <a href='docapis_what_is_a_rest_api.html'>What is a REST API?</a>
```

By storing the data in a structured way like this, I can get the data out and re-use it in different ways. For example, to show all terms in the glossary, I created some for loops here to get the info and format it in the display you see on the glossary page: _includes/glossary_full.html. The logic here is specific to Jekyll, but other docs-as-code solutions have similar templating options available. A lot of these templating languages operate in similar ways.

You could also get the same definition and insert it into a Bootstrap tooltip. To do this, first initialize the Bootstrap code:

```
<script>
$(function () {
  $('[data-toggle="tooltip"]').tooltip()
})
</script>
```

Then populate the tooltip like this to get info from the glossary YAML file:

```
Learning how to document <a href="#" data-toggle="tooltip" title="{{site.data.glossary.api.def}}">APIs</a> is essential in a tech landscape that has shifted to an information economy, with the buying and selling of information more than physical goods.
```

The result:

Note: This content doesn’t embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/docapis_glossary_section.html] to view the content.

Tooltips appear when you mouse over a link and disappear when you move your mouse off the link. Because of this behavior, don’t include any links in tooltip text. If you want to link to text, use a Bootstrap popover. Just like with tooltips, popovers must also be initialized:

```
<script>
$(document).ready(function(){
    $('[data-toggle="popover"]').popover();
});
</script>
```

You can then populate the popover with info from the glossary YAML file:

```
Learning how to document <a data-html="true" href="javascript://" title="{{s
ite.data.glossary.api.term}}" data-toggle="popover" data-placement="top" dat
a-content="{{site.data.glossary.api.def}} {{site.data.glossary.api.link}}">A
PIs</a> is essential in a tech landscape that has shifted to an information
economy, with the buying and selling of information more than physical good
s.
```

The result:

Note: This content doesn't embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/docapis_glossary_section.html] to view the content.

Notice that I separated out the link into its own property in _data/glossary.yml. I did this to provide flexibility as to whether I wanted the link included. Including the link would be problematic if the link pointed to the same page where the popover appears. Also, since tooltips don't support links, I didn't want to always include the links there. If you're using popovers, note the following:

- The Bootstrap popover code has an attribute required to enable html: `data-html="true"`
- In YAML, it's easier to code links in HTML rather than Markdown. Converting the Markdown link to HTML using the `markdownify` filter will include `<p>` tags around the content (which are problematic in popover formatting).
- If clicking the popover link makes the page focus jump to the top, use `javascript://` instead of `#` for the link target.

Finally, to style the link in a distinct way from normal HTML links, add some CSS to links with the `tooltip` and `popover` data attributes:

```
a[data-toggle=tooltip], a[data-toggle=popover] {
    color: cadetblue;
    border-bottom: dashed;
    border-bottom-color: darkgray;
    border-bottom-width: thin;
    font-style: italic;
}

a[data-toggle=tooltip]:hover, a[data-toggle=popover]:hover {
  border-bottom: solid;
  border-bottom-color: darkgray;
  border-bottom-width: thin;
}
```

Glossary style

As you're preparing your glossary source, note a few common stylistic conventions:

- **Definition length**. Definitions should be short enough to appear conveniently in a tooltip (hence should be about the length of a tweet).
- **Definition tense**. Definitions should not repeat the word being defined at the beginning of the definition but should follow a dictionary-style definition.
- **Capitalization**. Capitalization should be consistent. Decide whether to capitalize the first letter of the word. Also note that although acronyms are fully capitalized, the same word spelled out is not always capitalized.

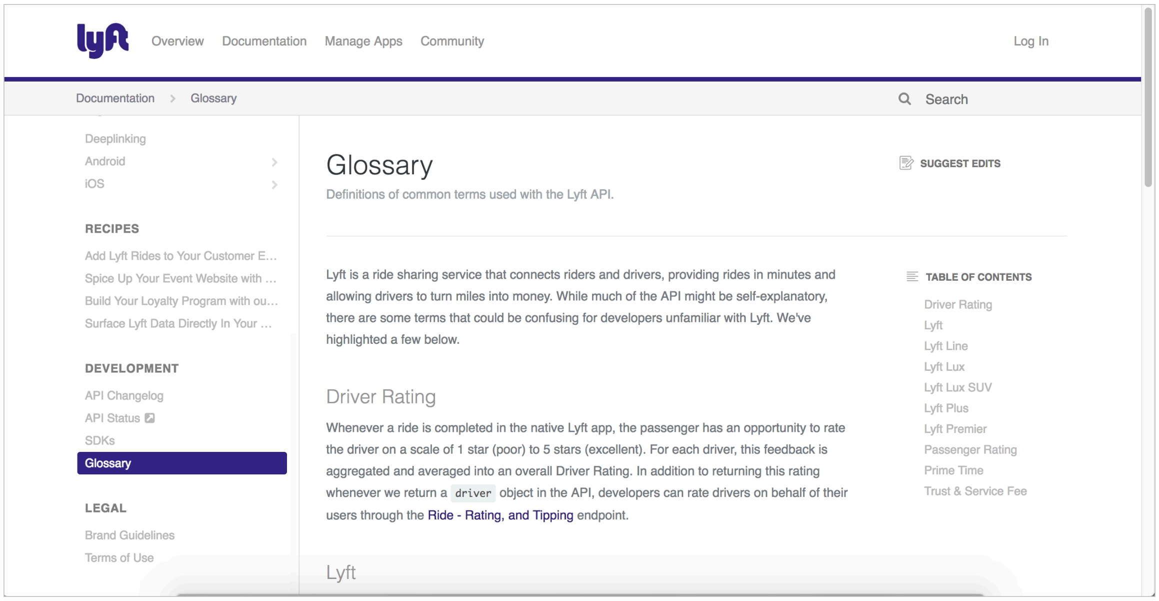
Benefits to writers

I mentioned earlier that one main benefit of a glossary is the heightened precision and awareness around language for the writers. As you write documentation, you'll be more attuned to the language that appears in the articles, and unfamiliar words will stand out more prominently. Add these words to your glossary, and then use them consistently. Word usage might prompt more discussions about whether you're using these unfamiliar terms consistently. For example, people may be using HAL and VHAL interchangeably, but as you add the term to the glossary, it forces you to clarify the difference. In clarifying the difference, you then become more aware as to the correctness of your usage.

Sample glossary pages

The following are sample glossary pages in various APIs.

Lyft



Glossary

Definitions of common terms used with the Lyft API.

Lyft is a ride sharing service that connects riders and drivers, providing rides in minutes and allowing drivers to turn miles into money. While much of the API might be self-explanatory, there are some terms that could be confusing for developers unfamiliar with Lyft. We've highlighted a few below.

Driver Rating

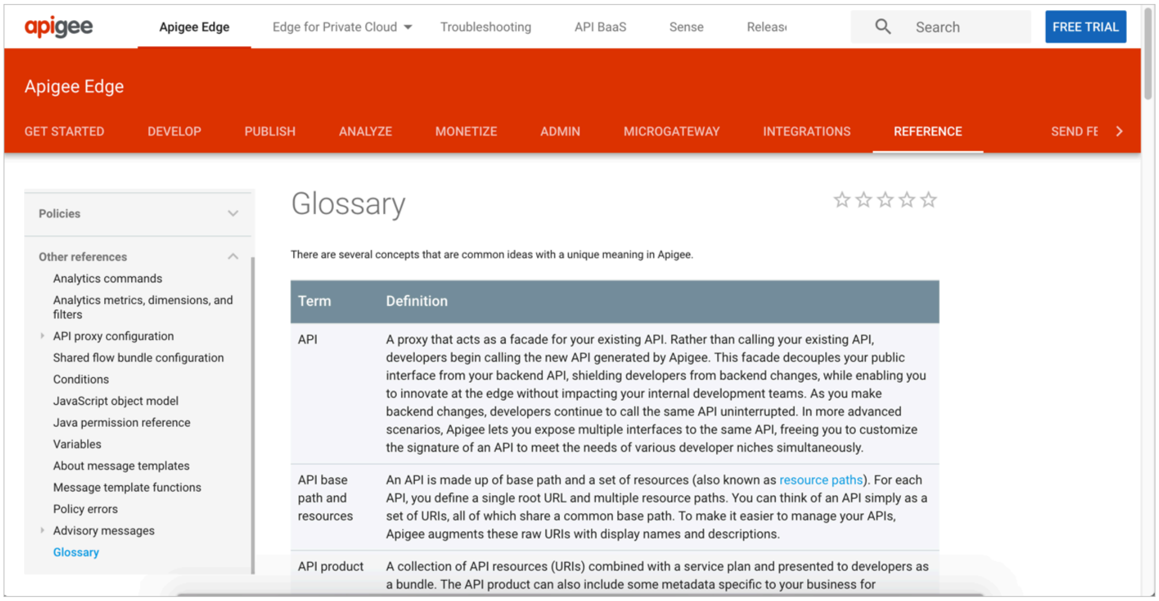
Whenever a ride is completed in the native Lyft app, the passenger has an opportunity to rate the driver on a scale of 1 star (poor) to 5 stars (excellent). For each driver, this feedback is aggregated and averaged into an overall Driver Rating. In addition to returning this rating whenever we return a `driver` object in the API, developers can rate drivers on behalf of their users through the Ride - Rating, and Tipping endpoint.

Lyft

Lyft glossary

Lyft's glossary doesn't have a ton of terms (there are about 10), but they're specific to the Lyft API world. You see terms such as "Lyft Line," "Lyft Lux," "Lyft Plus," and so on. I like that the developer documentation takes pains to define terms that might frequently be used on Marketing and Sales spaces. Whenever a company includes a glossary, it demonstrates a sensitivity the company has toward users. The company doesn't assume that their users understand all the company's specialized terms and vocabulary.

Apigee



Glossary

There are several concepts that are common ideas with a unique meaning in Apigee.

Term	Definition
API	A proxy that acts as a facade for your existing API. Rather than calling your existing API, developers begin calling the new API generated by Apigee. This facade decouples your public interface from your backend API, shielding developers from backend changes, while enabling you to innovate at the edge without impacting your internal development teams. As you make backend changes, developers continue to call the same API uninterrupted. In more advanced scenarios, Apigee lets you expose multiple interfaces to the same API, freeing you to customize the signature of an API to meet the needs of various developer niches simultaneously.
API base path and resources	An API is made up of base path and a set of resources (also known as resource paths). For each API, you define a single root URL and multiple resource paths. You can think of an API simply as a set of URIs, all of which share a common base path. To make it easier to manage your APIs, Apigee augments these raw URIs with display names and descriptions.
API product	A collection of API resources (URIs) combined with a service plan and presented to developers as a bundle. The API product can also include some metadata specific to your business for

Apigee glossary

The Apigee glossary provides another good model to follow. One interesting decision with Apigee's glossary is to format the entries as a table. The format probably doesn't matter much, but given that there are official definition list elements in HTML, it seems a bit odd to reject definition lists and use tables instead. Definition lists usually display better on mobile devices such as phones and tablets, and definition lists are easier to work with in general.

Regardless of the format, I'm usually thrilled to see a glossary. All too often, glossaries are neglected in documentation (not just in API documentation). They shouldn't be. Much of the complexity of technical content is due to the high number of specialized terms — terms that product teams often assume their audiences know (but in reality do not).

Other glossaries

- Squiz
- AWS glossary
- RapidAPI
- Microsoft Teams

Activity with glossaries

With the open-source project you identified (p. 686), identify the glossary (or lack of one). Then answer the following questions:

1. Does the API include a glossary?
2. If the API lacks a glossary, what terms would you suggest including in a glossary?
3. If the API includes a glossary, do term definitions link to any documentation topics for more information?
4. Are unfamiliar terms defined in the context of topics themselves?
5. Are the terms in the glossary consistent with the same terms used in competitor's

documentation?

More reading on glossaries

For more articles on glossaries, see the following:

- Chapter 13. Glossary Guidelines. *Read Me First! A Style Guide for the Computer Industry, Second Edition*. Sun Technical Publications. May 2003.
- Defining Glossaries, Michelle Kidd Tackabery. *Technical Communication.* Volume 52, Number 4, November 2005.
- Techniques for Introducing Unfamiliar Terms, David K. Farkas. *Technical Communication.* Volume 67, Number 1, February 2020.
- The ABCs of Writing a Technical Glossary, Evie Gray, William Ingram, and Dennis Bodson. *Technical Communication.* Volume 45, Number 1, February 1998.
- Principle 7: Reduce the complexity of technical language, Tom Johnson. *Simplifying complexity series*. Idratherbewriting.com.

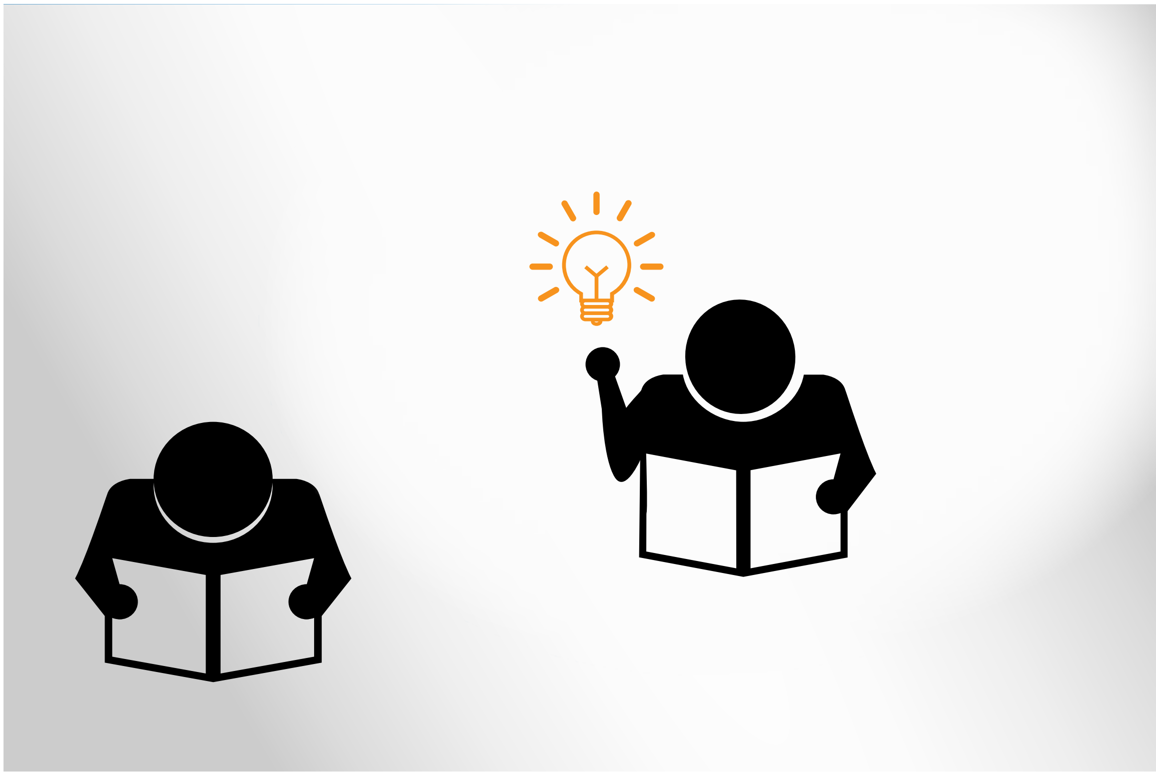
API best practices

API best practices can refer to any general advice your product team wants to communicate to developers about working with the API. There aren't any set number of topics typically covered in API best practices. Instead, the best practices can be a catch-all title for content that doesn't fit anywhere else.

What topics to include in best practices

Although many of the topics in API documentation are standard, there will usually be a laundry list of things to know about working with the API. You can only probably get this information by asking the developers.

The list of topics might include topics such as the following: pagination, time ranges, fault tolerance, cache values, connectivity, timeouts, downtime, SSL, versions, testing and validation, exports, languages, number handling, expanding resources, notifications, CORS, localization, and more.

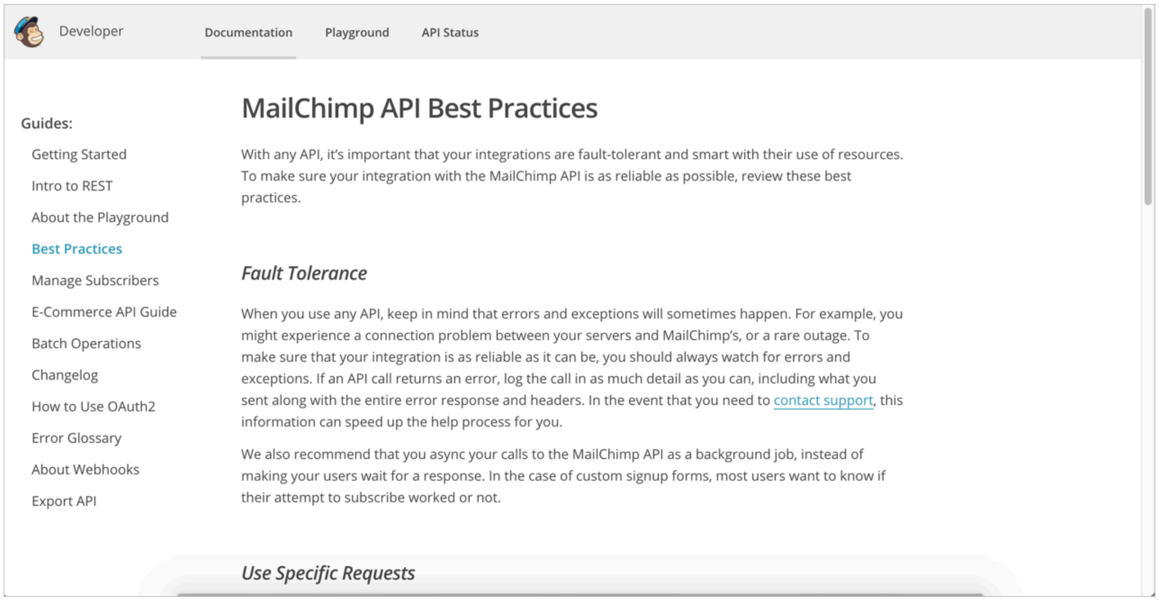


Best practices cover a range of topics usually unique to your API

Sample API best practices

The following are API best practices from several API documentation sites.

Mailchimp



MailChimp API Best Practices

With any API, it's important that your integrations are fault-tolerant and smart with their use of resources. To make sure your integration with the MailChimp API is as reliable as possible, review these best practices.

Fault Tolerance

When you use any API, keep in mind that errors and exceptions will sometimes happen. For example, you might experience a connection problem between your servers and MailChimp's, or a rare outage. To make sure that your integration is as reliable as it can be, you should always watch for errors and exceptions. If an API call returns an error, log the call in as much detail as you can, including what you sent along with the entire error response and headers. In the event that you need to contact support, this information can speed up the help process for you.

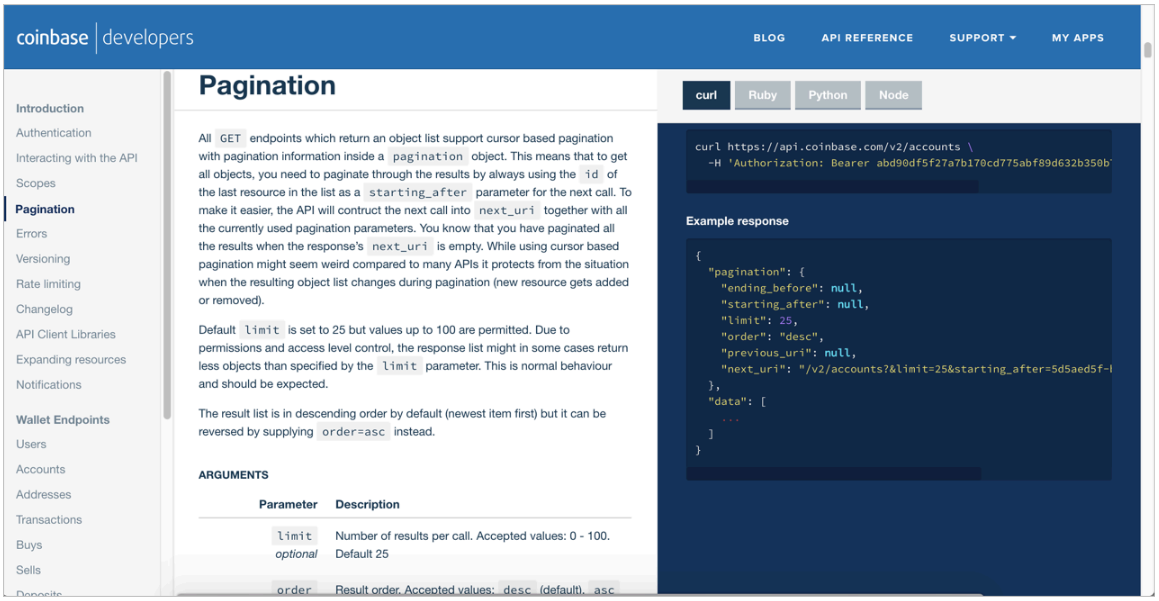
We also recommend that you async your calls to the MailChimp API as a background job, instead of making your users wait for a response. In the case of custom signup forms, most users want to know if their attempt to subscribe worked or not.

Use Specific Requests

Mailchimp best practices

Mailchimp's API best practices include tips about fault tolerance, using specific requests, authentication, cache values, connectivity, and registration. With fault tolerance, Mailchimp reminds developers that outages sometimes happen, so they should plan to handle scenarios accordingly if the API doesn't respond. With specific requests, Mailchimp warns users about the time it can take if the request is too general and hence returns too much information.

Coinbase



Pagination

All `GET` endpoints which return an object list support cursor based pagination with pagination information inside a `pagination` object. This means that to get all objects, you need to paginate through the results by always using the `id` of the last resource in the list as a `starting_after` parameter for the next call. To make it easier, the API will contruct the next call into `next_uri` together with all the currently used pagination parameters. You know that you have paginated all the results when the response's `next_uri` is empty. While using cursor based pagination might seem weird compared to many APIs it protects from the situation when the resulting object list changes during pagination (new resource gets added or removed).

Default `limit` is set to 25 but values up to 100 are permitted. Due to permissions and access level control, the response list might in some cases return less objects than specified by the `limit` parameter. This is normal behaviour and should be expected.

The result list is in descending order by default (newest item first) but it can be reversed by supplying `order=asc` instead.

ARGUMENTS

Parameter	Description
`limit` *optional*	Number of results per call. Accepted values: 0 - 100. Default 25
`order`	Result order. Accepted values: `desc` (default), `asc`

```
curl https://api.coinbase.com/v2/accounts \
  -H 'Authorization: Bearer abd90df5f27a7b170cd775abf89d632b350b
```

Example response

```
{
  "pagination": {
    "ending_before": null,
    "starting_after": null,
    "limit": 25,
    "order": "desc",
    "previous_uri": null,
    "next_uri": "/v2/accounts?&limit=25&starting_after=5d5aed5f-
  },
  "data": [
    ...
  ]
}
```

Mailchimp best practices

Coinbase doesn't specifically refer to these topics as best practices; instead, the navigation shows a laundry list of topics. Pagination is one of these topics worth expanding on here. How does pagination relate to APIs? Suppose your API endpoint returns all items in a user account. There could be thousands of items, and if all items were returned in the same response, it might take a long time for the API to gather and return the large amount of data. As a result, just like with searches on Google, the response returns a limited set, such as the first ten items, and then includes a URL that you can use to go to the next set of responses. Pagination refers to advancing to the next page of responses.

Earlier, when defining the characteristics of REST, I mentioned HATEOS (p. 41), or "Hypermedia as the Engine of Application State." Links in responses that return more results is one example of HATEOS.

Programmatically handling the URL to get more responses can be kind of tricky. If you want to get all items returned and then filter and sort the items, looking for specific values to pull out, how would you do this using the URL returned in the response? Your team might have some advice for developers handling these scenarios. Most likely, the endpoint would offer filters as parameters to apply to the endpoint, so that the initial response would contain the item set you wanted. This kind of advice might be appropriate in API best practices.

Activity with best practices

With the open-source project you identified (p. 686), identify any API best practices type of content. Then answer the following questions:

1. Are there best practices for working with the API that don't fit in any other typical API topics?
2. How are best practices organized in the existing documentation? Are they randomly listed in an FAQ?
3. What actual topics are covered in the best practices?
4. Are there issues logged against the project that should be covered in the API best practices?

Activity: Complete the SendGrid Getting Started tutorial

In this section, you explored Getting started tutorials (p. 379). To get better empathy for these tutorials as a user, try to complete the following tutorial from SendGrid: Getting Started with the SendGrid API ↗.

There are no supplementary instructions or tips for working through the content, as that would be a crutch to the experience. Ideally, you should be able to complete all the steps in about 5 minutes. After you finish, answer the following questions:

- Were you successful?
- If you were successful, how does this make you feel towards the documentation?
- Would you improve any part of this getting started tutorial?
- How could you implement a similar tutorial with your own product?

Activity: Judge conceptual content and decide which is best

With each of the conceptual topics in this section, you've been analyzing the topic in the context of the open-source project you identified (p. 686).

Now broaden your analysis to see how these conceptual topics appear in other API documentation sites.

Remember that sometimes the best way to learn how to create API documentation is to carefully observe how it's done on sites you admire. Following standard practices in the industry helps you create more predictable, easy-to-follow patterns in your own documentation.

Activity: Judge conceptual content and decide which is best

The following are 7 common conceptual topics in API documentation. For each topic, examine the three examples and decide which one is best. This is an activity we'll do together as a group.

Product overview

- Spotify
- Hootsuite
- Watson Assistant

API getting started

- Mailchimp
- Paypal
- Google

API authentication and authorization

- Sendgrid
- Twitter
- AWS

API status and error codes

- Clearbit
- Twitter
- Mailchimp

API rate limiting and thresholds

- GitHub
- Linkedin
- Bitly

API quick reference

- Eventful
- Parse
- Veracode

SDKs

- SAP

- PayPal
- Facebook

Chapter 8: Code tutorials

Documenting code poses one of the most challenging tasks for technical writers, especially if the writers don’t have an engineering background or familiarity with the code. Even without deeper engineering expertise, you can still document code — everything from code snippets to full-fledged sample apps. This section explores techniques and strategies for documenting code.

Chapter sections:

Common characteristics of code tutorials 433
Why documenting code is so difficult 434
What research tells us about documenting code 438
Five strategies for documenting code 451
Code samples 464
Sample apps 476
SDKs (software development kits) 480
API design and usability 488
Developer experience (DevX) usability 497

Common characteristics of code tutorials

Code tutorials describe how developers will use your APIs and other services to achieve some end. These tutorials often involve healthy chunks of code and configuration of various services. These are the “tasks” that would be more common in end-user docs.

Code tutorials have common sections or characteristics (whether explicitly called out as such or not).

Starting with a template that lists these sections can help you gather the right information when you’re developing your own tutorials.

Activity: Analyze two code tutorials

Code tutorials often have the following sections or characteristics:

- **Scenario description** - the “why” behind the tutorial
- **Tutorial outcomes** - objectives or a demo of solution
- **Solution overview** - a birds-eye view of the solution
- **Intended audience** - assumptions about who the tutorial is for, skill level, point in journey
- **Prerequisites** - expectations about required items, configurations, or utilities for the tutorial (e.g., hardware devices, services set up, API keys, software, etc.).
- **Steps** - the tasks involved in the solution, often formatted as sections. The steps show an assembly order working up to final solution, building the code piece by piece as needed (like Legos).
- **Examples to make it real** - examples or personalized info included in the steps to make the steps more meaningful and understandable.
- **Teaching moments** - conceptual explanations peppered in along the way as asides or footnotes.
- **Inline code comments** - explanations inside the code about what’s going on.
- **Conclusion, next steps** - wrap-up, showing fully assembled code, pointing out next logical steps, linking to related tutorials.

The following are sample tutorials. Pick two tutorials and analyze them to see how many of the above common sections and characteristics you can find.

1. Twilio - How to build a chatbot
2. GitHub - Building a CI server
3. Message Buttons with Node.js
4. Dropbox - Quickly integrate file upload in your web app using the Chooser
5. Maps JavaScript API Overview
6. Searching By Seller: Reviewing Information About A Seller
7. Using the Marketing API with the Facebook Pixel
8. Create a Custom Map URL
9. Training Course: How to Build a Multimodal Alexa Skill
10. Introduction to Populating a Website with API Data
11. Video Skills Kit for Fire TV

Why documenting code is so difficult

Software code, often written in Java, C++, or any other language, is challenging to document in part because technical writers often aren't already fluent in the programming language. But even for writers or developers who are fluent in the language, code is hard to document. There isn't a step-by-step process to follow. Code is often arranged in non-linear order, so you can't simply proceed line-by-line through it. There's also the question of how much to document, what to cover, and where to include the documentation. Overall, best practices for documenting code are somewhat fuzzy and undefined, which makes documenting code one of the most challenging and difficult tasks technical writers face.

In a previous topic, Documenting code (p. 464), I briefly touched on the need to document code. But given the importance of this topic, I've expanded this section in more depth with its own dedicated topics here.

Starting with a recent experience documenting code

I like to start abstract topics by grounding them an actual experience. A recent documentation project I worked on involved creating video skills for Fire TV apps — developing the backend logic to make your streaming media apps voice interactive, so users could say "Play *Interstellar*," (or some other movie) and the video would play in your app.

When users say these Alexa commands, Alexa interprets the commands and packages up the info into a structured JSON request that it sends a partner's Lambda code. Lambda, an AWS service, provides serverless computing. Partners are supposed to listen for the incoming requests and then develop their own code to respond appropriately to the requests, presumably retrieving the media identifier for *Instellar* (in this example) and sending this media identifier to their Fire TV app to play.

The implementation involves coding a Lambda function. To help developers, engineers provided a sample Lambda function (in Node JS), but without explanation. As I was working on the documentation for this, I felt a need to explain the logic of the Lambda code.

The Lambda code was only about 450 lines long, and not really complicated. But I wasn't that familiar with Lambda or Node JS, so I learned enough about both to get a sense of what was going on in each line. In my approach to documenting the code, I explained the conditions that triggered the Lambda code to be invoked in the first place. Then I presented the Lambda code in its entirety so the user would have context. Then I split the code into four discrete logical sections. Below the full code sample, I presented each section (labeling them as Section 1 Explanation, Section 2 Explanation, etc.) until I proceeded through the whole code sample.

This approach wasn't particularly great, but I didn't have a better method. I wanted room to elaborate in detail about each of these sections rather than just making brief inline comments. You can see the documentation here: Step 3: Understand the Alexa Directives and Lambda Responses (Multimodal Devices) ↗. No doubt this documentation will soon change, so I'm hesitant to give any more detail beyond the approach I described. But I relate it here as an intro to the challenge of documenting code.

Overall, documenting code is perhaps the most challenging aspect of technical writing, especially for non-engineers. The following sections explain a few reasons why documenting code presents so many challenges.

Challenge 1: Code doesn't follow a step-by-step paradigm

Code itself is non-linear. What appears at the top (e.g., variables) might not be implemented until functions at the bottom. Functions defined at the bottom might be run inside other code blocks in the middle, and so on. When you're given a chunk of code to document, its assembly order isn't apparent at all.

The central paradigm most technical writers follow is a task-based model where you begin with step 1, 2, 3, and so on until you reach the end of the task. This isn't the case with code documentation. Code is non-linear by nature. You can't simply start from the top and proceed to the bottom. Although I attempted a somewhat section-by-section explanation in my code explanation, I had to skip over some lines or note that they were part of logic explained in later sections.

zebra

Overall, this non-linearity is very different from the procedural approach usually followed in tech docs.

Challenge 2: Audience has a high variance of technical levels

Another challenge I faced was deciding what to explain and what to skip over. Would developers already be versed in handlers in Lambda and Node JS? Or would this be new to them, as it was to me?

In documenting code, you have to write to your audience's awareness and knowledge needs, even when their technical levels vary dramatically. But when the audience awareness and needs vary dramatically, we either overwrite for the advanced developers by explaining the obvious, or we alienate less experienced developers by assuming too much.

Implementing progressive information disclosure models (where you reveal a little bit of info and then let them expand for more details) can be tricky. Even if audiences are technical, there's no guarantee that they have expertise in the particular technology you're documenting. As a result of all this haziness, we often end up imagining ourselves as the audience.

Challenge 3: Code requires an understanding of a specific programming language

Related to the above point about audience variation is the fact that we technical writers often lack the familiarity with the programming language to begin with. Or if we do have some familiarity with programming, it's not always the right language for the project. So right away we're at a disadvantage and have to ramp on tutorials just to understand the basics of what's going on in the code.

Additionally, we're not documenting basics — we're documenting how to implement code in a particular context, often at an advanced level. Knowledge about how the code works is simply assumed. We pretty much have to walk into an Advanced Calculus course and explain Legrange Multipliers without having taken anything more than beginning Algebra.

Challenge 4: Making sure code samples work from release to release requires a lot of maintenance

Another challenge is making sure the code samples work from release to release. I pasted in the full Lambda code to provide context, but within a couple of weeks, I tweaked the code sample a bit. I then had to update the Lambda code as well as my section-by-section explanations. If you have dozens of code samples spread throughout your docs, maintaining this code from release to release will be arduous. How do you ensure it works? Do you separate the code from narrative context so that it can be more regularly tested?

Separating the code to enable testing sounds like a good idea, but as soon as you separate code from the conceptual explanation, you risk the possibility that someone will update the code in such a way that it no longer matches the explanation.

Challenge 5: Engineers have a much more trained eye for good versus bad code

Finally, I should note that when I document code, I feel a bit like an outsider writing about a culture or country that I don't belong to. Technical writers are often outsiders to engineering domains. Not being a developer, I might not even realize that the code is bad. Engineers live and breathe code, and many feel that *code is poetry*.

An efficient technique in code (e.g., recursive loops that expand resources as needed) can be beautiful, evoking an aesthetic in the engineer's mind. In my perspective as a technical writer, it's unlikely that I'll approach code with the same reverence and awe. My more mundane approach to code might make it difficult to resonate with developer users.

The importance of code documentation

Despite the difficulties of documenting code, this area of documentation shouldn't be overlooked. Ask developers what the most important element of API docs is, and the answer you're bound to hear time and again is code samples. Include working code samples that developers can easily copy and paste into their documentation. Code samples that demonstrate how to incorporate abstracts into an actual implementation. Code samples, sample apps — whatever the form, just give us more code, code, code.

Check out the following video clip from engineer Ruthie Ben Dor during a Write the Docs conference. Responding to the question, "What are the three most important elements of producing API documentation?" Ruthie emphasizes the need to include code samples.

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_code_difficulty.html] to view the content.

Around the 4:15 mark, Ruthie says,

> I think that if you're going to produce API documentation, there are 3 things that should either be included in it or that you should do in the course of making it. The most important thing, I think, if you just want people to just start using it is code examples. There are many API documentation generators that will generate code samples for you. Sometimes they're not great, so having someone actually go through and vet those code samples to make sure that they actually work correctly – that's huge. It's one thing to be given a method and how it's called, but for the developer to have to take that and figure out how to make working code, hitting that endpoint, it's just a little bit more of a barrier. As opposed to if they can just copy and paste in their browser console to prove that it works to themselves, that's a really nice way to lower the barrier to entry. Code samples are huge, and code samples in a variety of languages that people are likely to use with your API. (Ruthie Ben Dor: API documentation and developer portals 2/3 - YouTube ↗)

Why do engineers so frequently say they want code? Code examples show how to implement an abstract, narrative explanation in an actual way. Code shows how to use reference APIs toward some actionable and tangible end. In that sense, code is an extremely powerful tool to help users understand how to use your API.

In fact, in the 2021 State of API report by SmartBear, "examples" are the most desired form of documentation, topping all other documentation types by a wide margin.

Despite the importance of code samples, they're often neglected or missing from API docs. After all, it's much easier to just cover reference material and let developers figure out how to actually use it constructively. In this section of my API course, I'll provide concrete strategies for successfully incorporating code samples into your documentation.

What to do if you don't understand the code

As you're documenting code examples, you'll probably run into situations where the code is beyond your understanding. For example, the engineer might call out various classes and methods in a Java API and have some fragmented, incomplete code comments on how to use it. In these situations, one strategy is to ask the engineer to talk through the code from beginning to end. Important: Record the meeting where the engineer explains the code. After the meeting, add notes from the engineer's explanation about different aspects of the code. Pepper in your explanations both in regular copy and in code comments.

After reviewing your writeup with the engineer, present the content to other engineers or devrel-type roles, and ask them to review it. Basically, gather enough feedback from other people to get a sense of whether the code and your explanations are clear enough in places or still confusing. Act as a kind of mediator between the author and users. It doesn't take a deep understanding to play this mediator role, and as long as you record the meetings where explanations are flying around, you can go back at a snails pace and listen/re-listen to the comments and convert them into a semi-intelligible writeup.

Of course, the ideal is that you would also be comprehending what's going on in the code to a deeper extent, but this isn't always possible in the timeframes given. Consider using the project as a starting point for deepening your understanding of a programming language or framework. Start taking a course on the programming language on Udemy, for example, and highlight the various concepts in the tutorial that you want to learn more about.

Next topic

Go on to the next topic in this series: What research tells us about documenting code (p. 438).

What research tells us about documenting code

Before diving in to how to document code, let's first explore some research that has been done on best practices for documenting code, as this can inform our direction and approach. A couple of academic articles stand out as noteworthy for this effort:

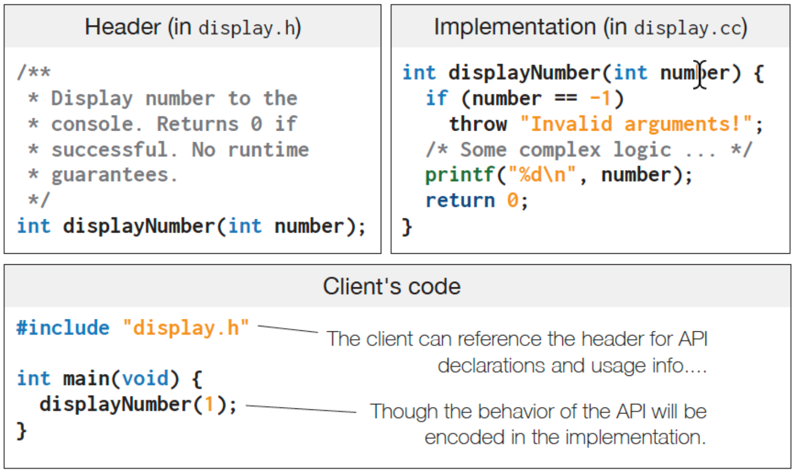
- **"When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects"** by Head et al. This article explores how developers at Google find and use documentation for code. The researchers found that for simple code, sometimes developers prefer to examine the code directly. However, for more complex code, developers consult the code's documentation, often by looking in the formal class declarations for information they need; other times they look at comments in the implementation code. Besides providing guidance on the best location for documentation, the researchers also identify what type of answers and guidance developers want for the content of the documentation.
- **"How Developers Use API Documentation: An Observation Study"** by Meng et al. In this article, the researchers look at how developers interact with API documentation and found a mix of both *systematic* (read-first, explore-later) and *opportunistic* (explore-first, read-later) learning styles. While we often write with systematic developers in mind, focusing on opportunistic behaviors might be more beneficial, and will cause us to look more closely at improving search, navigation, interactive components, troubleshooting, error messages, and other action-oriented features.

Both of these articles come from academic journals. It's actually rare to find research about API documentation in academic journals (not sure why), and when you do find them, they're often in engineering or computer science journals (rather than tech comm journals). Later I'll dive into some articles outside of academic journals.

When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects

First, let's explore "When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects," by Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. This article was published in the *2018 ACM/IEEE 40th International Conference on Software Engineering.* (To read the article, see this ResearchGate link or go here.)

This research coordinates efforts among academic researchers, engineers, usability specialists, and members from Google's Engineering Productivity Research team. Given how important documentation is for understanding code, the researchers want to know the best location for documentation as well as what information engineers want in docs. Specifically, they focused on C++ APIs and asked whether engineers are more inclined to consult the header files (where classes are defined) or the implementation files (where classes are implemented) for the information they need. The following screenshot (from their article) shows the difference between header and implementation files:



Header (in `display.h`)

```
/**
 * Display number to the
 * console. Returns 0 if
 * successful. No runtime
 * guarantees.
 */
int displayNumber(int number);
```

Implementation (in `display.cc`)

```
int displayNumber(int number) {
  if (number == -1)
    throw "Invalid arguments!";
  /* Some complex logic ... */
  printf("%d\n", number);
  return 0;
}
```

Client's code

```
#include "display.h"

int main(void) {
  displayNumber(1);
}
```

The client can reference the header for API declarations and usage info....

Though the behavior of the API will be encoded in the implementation.

Figure 2: Writing docs in .h and implementation in .cc files. For C++ APIs, the member declarations and actual implementation are often split between two distinct files, as shown in this toy example. Clients may have to reference both to understand both an API's intended usage *and* its actual behavior.

Basically, in C++, the header files (`.h`) contain the classes and the main documentation. The implementation files (`.cc`) instantiate and implement the classes from the header files. In short, the header files contain more formal documentation that follows specific annotation conventions, while the implementation files contain the guts of the logic about how the class has been implemented. Implementation files have comments peppered inline with the code, without formally structured doc annotations. A central question the researchers wanted to know is whether users gravitate toward the inline code comments in implementation files or the official documentation in the header files.

The researchers used tracking tools to identify when developers would switch from one type of file to another, and they also interviewed the developers as a follow-up. Google has about a billion lines of code stored in a central code repository that can be used across the company, so thousands of developers might find and discover code in this monorepo to use in their projects. The team that uses an API might not know the team that developed the API, and vice versa.

Even if you don't document C++, this study is helpful because it raises this central question: should you put the bulk of your documentation in formal descriptions about the code, or should the bulk of your documentation appear within the context of the code, peppered in as inline comments.

After gathering information from more than 600 participants in their study, the researchers found that not all code is equal. Complex code needs more formal documentation, but simple code might not need documentation at all.

What type of code actually needs documentation

First, the researchers found that most developers actually looked in the header files for documentation:

> Survey respondents reported it would be most convenient to find answers to many of these questions in header files, though interviewees indicated code could be accurate and quick enough to read in many cases.

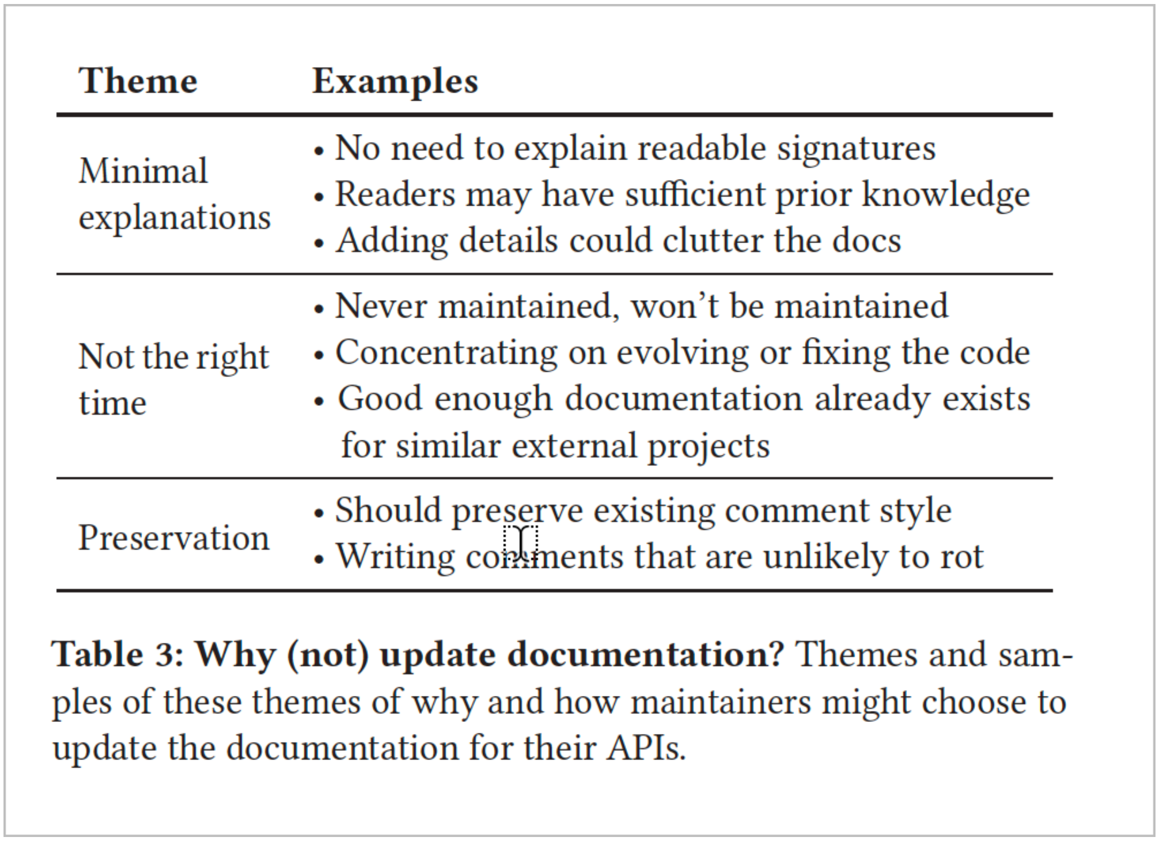
But the researchers also found that for simpler APIs, many developers read the code directly (rather than consulting the docs) to see if they can quickly understand the API. In other words, they see if they can figure out what's going on by looking solely at the code.

Some developers actually have philosophical views about distrusting the accuracy and currency of documentation and prefer to view the code as the primary source of information, like reading a primary source instead of secondary or tertiary sources of information. As many developers who distrust docs know, documentation for code can easily become outdated and neglected, so why trust it? Why even bother to read it?

In fact, some developers feel that documentation for simple code becomes a liability and a hindrance for development. The doc gets in the way of the developer's path to simply reading the code and understanding it on its own terms.

Besides skipping docs when the code is simple enough to understand on its own, the researchers also recommend avoiding writing docs while development is in constant flux (because it makes documentation a constantly evolving target). The researchers say you might also consider skipping writing docs when there aren't sufficient resources to keep the documentation updated. When maintainers can't keep the documentation up to date, it "rots" and becomes more of a liability.

In short, there are valid arguments for not even writing documentation, particularly for simple code. The following chart shows when documentation might not be necessary with code:



Theme	Examples
Minimal explanations	• No need to explain readable signatures • Readers may have sufficient prior knowledge • Adding details could clutter the docs
Not the right time	• Never maintained, won't be maintained • Concentrating on evolving or fixing the code • Good enough documentation already exists for similar external projects
Preservation	• Should preserve existing comment style • Writing comments that are unlikely to rot

Table 3: Why (not) update documentation? Themes and samples of these themes of why and how maintainers might choose to update the documentation for their APIs.

However, for more complex code, especially where multiple files and generated code might be involved, developers still relied on the formal documentation to understand it. The researchers explain:

> When isn't code enough to be self-documenting? Sometimes, developers had no problem reading code, and in fact preferred it for finding more accurate information. However, there are some cases where self-documentation isn't feasible, like code with overly complex method signatures and generated code. Other details, like recommended usage, just can't be conveyed by source code.

In short, when the source code isn't intuitive to understand on its own, developers turned to more lengthy and formal documentation about it. This makes sense and aligns with best practices for GUI documentation as well — one should always focus on the complicated parts of a system, not the obvious elements that no one needs help with.

Granted, recognizing what code is simple versus complex is the challenge because the technical writer is likely not a developer and will likely only have a rudimentary idea about the complexity of the code. Just because the code is short or long does not give a clear indication about its complexity. A short snippet out of context might be confusing, while a lengthier sample that contains a fully functioning sample might be more understandable.

As an analogy, an outsider visiting a city in a faraway country. The outsider wouldn't know whether some observed event is normal or out of the ordinary — you have to be more familiar with the place to gauge whether an event is strange or commonplace. Same with code.

You could ask the developers about the level of complexity of the code, but this assumes that you can trust the judgment of the engineers who designed and created the code (unless you're asking engineers on other teams). Developers almost always overestimate the level of intuitiveness of the code they wrote and assume more capability in their audience than the audience actually has. How many times have you heard engineers say, "Users will understand this — and if they don't, they *shouldn't* be using the API." But are the risks of omitting docs greater than the risk of including them?

More advanced developers can probably extrapolate the API's usage from code, while beginning developers might need more handholding. Do comments interfere with readability for advanced developers but aid readability for new developers? Are we doomed to frustrate one audience in order to help another? And is there a greater risk in omitting documentation than in overburdening docs with *too much* explanation?

When to document code

Let's set aside questions about whether to document or not and focus instead on timing for writing docs. The researchers found that there's an ideal time for writing and updating documentation:

> The ideal time to propose changes to documentation is during code authoring and review, possibly through a surrogate like a code reviewer. Documentation can get updated only infrequently after it is initially written, as future updates may raise questions of whether the information adds clutter or redundancy.

In other words, write the docs when the code development is still fresh in the developer's mind. If you wait too long after active development finishes, the documentation will likely be neglected and forgotten, as developers move on to other projects.

Of course, timing is not always easy to plan. Your availability might not match up with the developer's coding sprints. You're probably juggling several other projects with more pressing timelines, and so you might have to postpone this documentation until one or two months post development. But by that time, the developer may have long ago finished coding and forgotten many details. The nature of complexity is that we hold a plethora of details in our heads (in our short-term memory) while we're elbow-deep in the task, but once we move on, our brains dump the information from short-term memory so that we can load up our brain's RAM with another project.

If you try to prod developers to articulate details no longer at the forefront of their minds, they might not have forgotten it, but their motivation and enthusiasm to explain it and review your docs will likely be poor. They have already moved on. Strike while the iron is hot.

With a recent project I documented, the documentation was mostly written post-release of the product. The initial version had been written by a product manager, and later, after many complaints about the poor quality of the docs, tech writers had been called in to fix it. But months after the product had been released, the team was no longer meeting in regular sprints, engineers were focused on other matters, and it seemed no one had interest in explaining things to tech writers nor reviewing documentation.

There is a right moment to jump into a documentation project — that time is usually pre-release. But just as one can jump in too late and miss out on enthusiasm and availability, jumping in too early can also prove inefficient. You might find that plans are high-level; nothing has actually been coded yet, or ideas are scrapped from one sprint to the next. It might be like trying to document a recipe while cooks are still deciding on the ingredients.

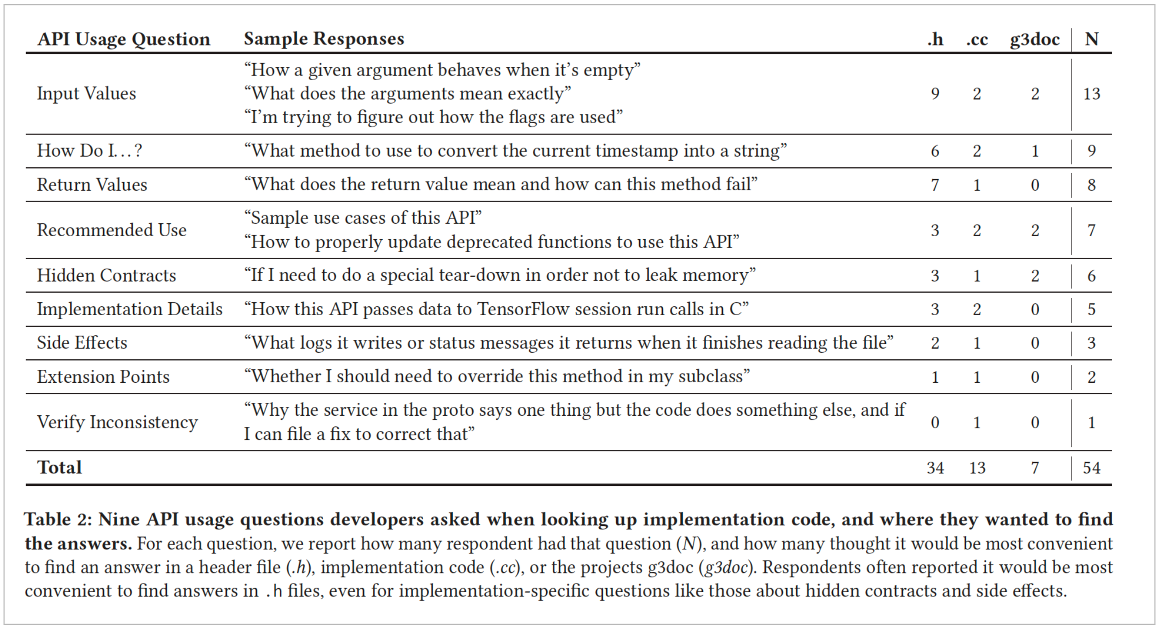
Regardless of whether you can actually do anything about the timing, you should have in mind that timing is a noteworthy element in documenting code. Code is not a standalone artifact independent of time. Time is a relevant factor that determines your ability to document code.

What questions to address in code documentation

So far we've looked at where code documentation should appear and when code documentation should be written. Now let's look at code documentation from another angle: what types of answers and guidance should be in the documentation. This is a more difficult, broad question, and the researchers' answer is "API usage." They write,

> Most searchers and maintainers we interviewed had opinions about what did belong in documentation, at both the level of headers and in-line comments. Maintainers and searchers mentioned the importance of describing how a file relates to other files in the project (S17), the state of the world when a method is called (S8), executable examples (M5, M8), implementation comments for future maintainers of an API (M5), explicit links to external documentation (M5), semantics of a function (M8), main concepts that someone should understand and know to use the API (M8), "what" the code is doing and "why" at a statement level (M6), and even a proof of correctness (M6). It is unsurprising that not all of this information was available for all of the APIs we saw during this study.

The researchers arrange this information into a chart for readability:



API Usage Question	Sample Responses	.h	.cc	g3doc	N
Input Values	"How a given argument behaves when it's empty" "What does the arguments mean exactly" "I'm trying to figure out how the flags are used"	9	2	2	13
How Do I…?	"What method to use to convert the current timestamp into a string"	6	2	1	9
Return Values	"What does the return value mean and how can this method fail"	7	1	0	8
Recommended Use	"Sample use cases of this API" "How to properly update deprecated functions to use this API"	3	2	2	7
Hidden Contracts	"If I need to do a special tear-down in order not to leak memory"	3	1	2	6
Implementation Details	"How this API passes data to TensorFlow session run calls in C"	3	2	0	5
Side Effects	"What logs it writes or status messages it returns when it finishes reading the file"	2	1	0	3
Extension Points	"Whether I should need to override this method in my subclass"	1	1	0	2
Verify Inconsistency	"Why the service in the proto says one thing but the code does something else, and if I can file a fix to correct that"	0	1	0	1
Total		34	13	7	54

Table 2: Nine API usage questions developers asked when looking up implementation code, and where they wanted to find the answers. For each question, we report how many respondent had that question (*N*), and how many thought it would be most convenient to find an answer in a header file (*.h*), implementation code (*.cc*), or the projects g3doc (*g3doc*). Respondents often reported it would be most convenient to find answers in `.h` files, even for implementation-specific questions like those about hidden contracts and side effects.

Nothing particularly stands out here, except that “input values” are read the most. Input values refers to parameters or other arguments that developers often consult to understand data types, casing, or other details. As such, take pains to document your parameters in great detail. I describe the various categories to cover at length with parameters in Step 3: Parameters (p. 132). With REST APIs, some details to note about parameters might include the following:

- For REST API endpoints, the type of parameter: header, query string, and path. Also consider request bodies.
- Default values
- Min or max values
- Data types (boolean, integer, string, etc.)
- Sample values

Other callouts about content include “How do I?…” questions. In other words, rather than just reading reference information, developers want instruction that explains how to implement the reference topics toward a business goal or end. Just as with end-user (GUI) documentation, focusing on tasks rather than simply defining reference information remains an important element of code documentation. Most technical writers already orient their mindset around task-based documentation, so this focus needs no expansion here.

Finally, “Recommended Use” is also an interesting element to surface. “Recommended” paths aren’t that common in GUI documentation — users usually have a task, and there’s a preferred way to achieve it. But with developer docs, there are often a dozen ways to go about a goal, and just because code compiles does not mean it is good. Code needs to scale, be efficient, and cover a multitude of use cases. Therefore, recommendations are in order to help guide a developer down many potential paths of documentation.

In fact, this might be a defining characteristic that separates GUI documentation from developer documentation. GUI documentation typically has a single path to achieve a result. Developer documentation, on the other hand, is more like providing a cabinet of baking goods to put at the developer’s disposal — salt, flour, baking powder, eggs, vanilla, baking soda, spices, and so on. The developer might choose to use one API here, another there, some in combination with each other, all to achieve a particular end. That particular end is more open and flexible depending on what the developer is building/baking. (One difference here is that developers don’t typically eat their code.)

Conclusion

Overall, this research has many insights and conclusions. The article addresses many concerns around code documentation and presents a complex view about each of these facets — when/if to document code, where to put this documentation, when to write the documentation, what questions to address, and more. There’s not always a clear path to follow (it’s messy), and many environmental, product, and audience details must factor into the documentation strategy. Still, this article provides solid research and probes the topic in illuminating ways.

How Developers Use API Documentation: An Observation Study

Now let’s move into another meaty academic article that addresses best practices for documenting code. The January 2019 issue of *Communication Design Quarterly*, a publication of SIGDOC (the Special Interest Group for Design of Communication), features an article called How Developers Use API Documentation: An Observation Study. Several researchers from Merseburg University in Germany — Michael Meng, Stephanie Steinhardt, and Andreas Shubert — set out to “understand how developers use documentation when starting to work with a new API.”

For their research, they "asked software developers to solve a set of pre-defined tasks using a public API unfamiliar to them on the basis of the documentation published by the API provider" — and then observed their behavior. Basically, these users had to figure out how to construct REST API requests with the right parameters and other configurations in order to send requests that would return the needed information. The researchers then observed how the developers used the API documentation to figure out the tasks.

There are a lot of great observations and conclusions in this article. I'm just summarizing and highlighting the information here. I recommend that you read the article ↗ for the full details.

Systematic versus opportunistic behaviors

The authors present some previous research about *systematic* and *opportunistic* learning behaviors. These terms are typically how previous researchers describe the contrasting user behaviors.

You're undoubtedly familiar with these two types of behaviors. Sometimes when you get a new device, you just start pushing the buttons and exploring how it works based on inputs and responses, trial and error, etc.. — this is called "opportunistic" behavior. Other times you might crack open the user guide and start reading from page one before pushing buttons on the device — this is called "systematic" behavior). Other times you blend the two modes ("pragmatic" behavior). Same with developers using an API.

The authors describe the opportunistic behavior patterns in their study as follows:

> … [opportunistic] developers worked in a more intuitive manner and seemed to deliberately risk errors. They often tried solutions without double-checking in the documentation whether the solutions were correct. For example, P10 [person 10] changed parameter values to values that seemed to match based on experience with similar problems, but he did not check in the documentation whether the values were actually correct or even existing. P2 [person 2] inserted parameters that he had noticed at some point in the documentation before, but did not attempt to re-consult the relevant section of the documentation to make sure that the parameters were spelled correctly. …
>
> We found that opportunistic developers in our test started the first task with some example code from the documentation which they then modified and extended. Once a task was completed, the piece of code that solved the task was used as starting point for the next task, which again was a potential source of error. Developers in this group worked in a highly task-driven manner, but also tried things that were not related to the task, but possibly helped them to build a broader understanding of the API in passing. For example, P9 [person 9] submitted a request for a UPS service (United Parcel Service) which was not required by any of the tasks, simply in order to see what would happen.
>
> We noted that developers which we assigned to the opportunistic group did not take time to get a general overview of the API before starting with the first task. They scrolled briefly through some pages of the documentation, checked the tools available and then started with the first task. Developers from the opportunistic group wanted fast and direct access to information. They did not systematically read larger sections of the documentation, but typically searched for a specific piece of information and then scanned the documentation.

In short, opportunistic developers learn by doing. They look at a piece of code, try it out, experiment with parameters, see what gets returned in the response, and more. They learn through experimentation, trial and error.

In contrast, the systematic developers approached tasks by reading first:

> In our test, we note that these [systematic] developers took some time to explore the API and to prepare the development environment before starting with the first task. Moreover, they took some time to get a general orientation. For example, P7 and P8 [persons 7 and 8] studied some sections in the documentation, then sent a GET request to the API and analyzed the response to check whether the request-response process worked as expected.

In short, systematic developers follow a “read-first, try-later” approach (while opportunistic developers follow a “try-first, read-later” approach). Pragmatic developers mix the two: “try-read / read-try.”

Although it might seem convenient to divide learning styles out by systematic, opportunistic, and pragmatic behaviors, researchers also found that the same developers did not always exercise the same behavior. Whether one approaches a task opportunistically versus systematically versus pragmatically might very well depend on the nature of the problem. For simple API requests, carefully reading the documentation probably isn’t warranted. But for complex API requests, where the developer might be totally stumped from the start, reading the documentation systematically might make sense.

The researchers explain that “the strategy a developer follows does not seem to predict a tendency towards using information from the Concepts page in our test.” In other words, just because you’re an opportunistic user, it doesn’t mean you always skip conceptual explanations — it’s just that you might not start with concepts. A non-linear reader might start with code, trying it out on their own, and circle back to the introductory conceptual information when the code doesn’t work as expected.

Deciding to cater to one type of behavior at the expense of the other might not be practical, since the learning behaviors and approaches seem to be in much greater flux than it seems.

When I’m writing docs and structuring my help system, I admit that I often have the more systematic developer in mind — the one that will read the material from start to end, the one who begins at step one, reads conceptual introductions, and then proceeds to the code examples and such. But that learning preference doesn’t describe a huge percentage of learners. It’s probably better to design for the opportunistic behavior, since this behavior pattern tends to go against our natural inclinations for linear and top-down information design. The linear/systematic behavior might be accommodated by default (since we tend to write linearly), while the non-linear/opportunistic behavior pattern is more likely to be neglected.

Designing for opportunistic behavior

How do you design for opportunistic behavior? If you recognize that users learn through experimentation and action, you’ll put more emphasis in code comments and code samples, error messages, troubleshooting, interactive experiences (such as Swagger UI) so developers can try out requests, clear navigation, and search to facilitate the user jumping around for specific information.

The authors call out some of these design patterns in their recommendations. The second half of the article provides recommendations such as:

- “Provide transparent navigation and a powerful search function”
- “Provide clean and working code samples”
- “Enable fast use of the API”

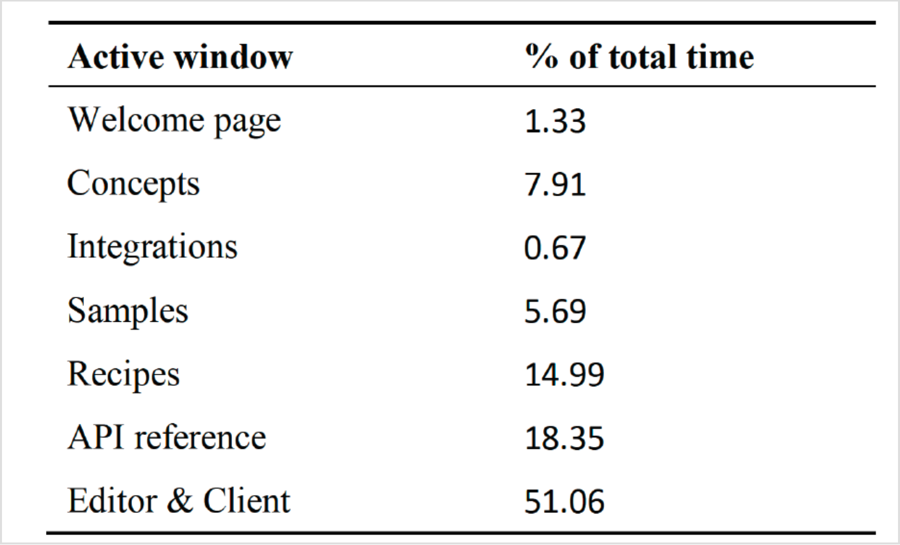
- “Provide important information redundantly”
- “Organize the content according to API functionality”

Note that “opportunistic” isn’t the author’s own terminology choice (it’s a term previous researchers used). The authors say that opportunistic behavior “bears many similarities with the exploratory and active approach described by John Carroll …”, referring to Carroll’s seminal work in *The Nurnberg Funnel*, which ties in with Mark Baker’s “Every Page Is Page One” and other nonlinear reading behaviors. Readers jump around, gathering information after exploring the system with trial and error.

Instead of “opportunistic” (which has a somewhat negative connotation), others have also characterized this behavior as “exploratory” or “active” or “bottom-up” learning. See How to design documentation for non-linear reading behavior ↗ and Principle 2: Make information discoverable as the user needs it ↗ for more information.

Where users spend the most time

If developers jump around from code to concepts and other places, where are they spending most of their time in the docs? If we can measure the time in one type of documentation more than another, we can give more attention to that kind of documentation. The authors measured the time users spent in various parts of the documentation as follows:



Active window	% of total time
Welcome page	1.33
Concepts	7.91
Integrations	0.67
Samples	5.69
Recipes	14.99
API reference	18.35
Editor & Client	51.06

Similar to the previous research from Head et al., ““When Not to Comment,” developers spent most of the time looking at API reference information (e.g., parameters). But here the researchers make an interesting observation that breaks other assumptions: they say developers looked for topics rather than categories of information — in other words, they didn’t necessarily distinguish between concepts versus recipes versus reference information types as they searched for information. They had a problem to solve, and they looked for information related to that problem, regardless of whether that information might be classified as a concept, task, reference, troubleshooting, recipe, or other topic type. As a result, the researchers recommend a more topic-based organization strategy:

> **Organize the content according to API functionality.** A first aspect concerns the high-level organization of the API documentation. From the results of our study, we conclude that API documentation should be structured according to categories that reflect the functionality or content domain of the API rather than using categories that signal the type of information provided. Instead of dividing documentation into “Samples,” “Concepts,” “API reference” and “Recipes,” the API used in our study should be reorganized using categories such as “Shipment Handling,” “Address Handling” and so on. If developers experience a problem while working with the API and turn to the API documentation to find information that solves the problem, they are likely to know the content domain of their problem (such as shipments or address handling), but it is more difficult for them to predict whether the information they are looking for is presented in the API reference, in a section dedicated to presenting code examples, or in a section discussing concepts. Note that this guideline can be viewed as an application of the principle of minimalist documentation according to which the components of the documentation should reflect task structure (van der Meij & Carroll, 1995).

This is a somewhat radical recommendation because almost all API docs clearly separate out the reference information and label it as such. But perhaps the conceptual and recipe-based information can more easily integrate and re-use information from the reference section in seamless, unified ways. That way, if you’re looking for information on Shipping Handling (the example in their study), you might see the relevant Shipping Handling endpoints and parameters as well as Shipping Handling introductions and tutorials right in the same place (instead of jumping over to Reference for the Shipping Handling endpoint, then back to Recipes for how it might be used, and then over to Concepts for other Shipping Handling information).

It makes sense to have a Reference section where all endpoints are listed, but if this is the only place where these endpoints are described, this pattern might not be most convenient for users.

The authors also recommend that you integrate concepts with their related tasks:

> **Present conceptual information integrated with related tasks.** Another aspect relevant in this respect concerns the integration of conceptual information that developers need in order to use the API successfully. Confirming results reported in Meng et al. (2018), our study supports the conclusion that developers vary with respect to whether they use conceptual overviews that introduce important API concepts in a systematic way. While some developers use such offerings, others tend to ignore them. To reach both groups of developers, conceptual information should not be aggregated in a dedicated section or document that signals to focus on conceptual information. We recommend presenting conceptual information integrated with the description of tasks or usage scenarios where knowledge of these concepts is needed. To give an example from the API used in our test, information regarding the representation of a shipment should be introduced in the section describing how to create a new shipment, and specific features of a return shipment should be provided in the section describing how return shipments are handled.

As much as tech writers might like to separate out information into different topic types (following patterns like Information Mapping or DITA), creating topics that clearly separate out concept from task from reference and troubleshooting topics constitutes a big fail in terms of usability, since developers need all of this information as they explore the problems they attempt to solve. If you are separating out this information, hopefully you’ve structured your help system in a way such that the information is closely linked or integrated with each other.

Conclusion

While the learning styles discussed in this article might seem more applicable to overall documentation (rather than our specific focus on code documentation), the takeaway with code documentation is pretty clear: when you document code, the code should be easy to experiment with. Can users copy and paste it into their own IDEs and make it run? Can they copy and paste requests into Postman and get immediate responses that they can learn from?

More than any other type of API documentation, when you document code you find yourself with a direct opportunity to target the opportunistic learning style with experiment-and-try opportunities.

Takeaways from the Research

To summarize the takeaways from the research presented here, here are some key points:

- Omit documentation when the code is simple and understandable on its own.
- Write code documentation while the project is fresh and active in developer’s minds.
- Make sure the reference documentation (e.g., parameters) is complete and accurate, since this documentation is where developers spend most of their time.
- Include documentation both in regularly expected locations as well as peppered in the code itself with inline comments.
- To accommodate non-linear, opportunistic behaviors, design your help system with healthy doses of code, interactive API explorers, search features, troubleshooting information, and cross-references.
- Organize information by the function it provides rather than by its information type (task, concept, reference, etc.).

- Make code easy to experiment with so that people can learn directly through these experiences.

These strategies provide a foundation for best practices that we will explore in more detail with more concrete, tangible techniques in the sections that follows.

Five strategies for documenting code

In this section, we'll dive into approaches for actually documenting code. There are a number of approaches writers and engineers take here, and I've covered five of them with examples and commentary.

Approach 1: Separate the "how" into inline comments, the "why" into external docs

After reading the intro to this section (When code is too complex to understand (p. 432)), one reader shared her approach to documenting code. Morgan Craft said ↗,

> I tend to split my documentation up into the 'how' (inline comments) and the 'why' (external markdown docs).

I think this general division and arrangement of code comments makes sense. Separate your explanations into two general categories: how and why. Then insert the *how* comments inline with the code (every 5-10 lines or so). In contrast, put the *why* comments into external sections before or after the code. Granted, explanations of *how* might prompt details about *why*, and vice versa, so separating the two might not be so easy in practice. But this general pattern likely holds true.

Additionally, I like this approach because it prompts you to specifically consider the why. As technical writers documenting code (that others write), we might fail to consider the "why" behind the choices made. It might be difficult to even see what "why" questions exist. Why use this class instead of another? Why approach it like this instead of like that? There are often many different ways to accomplish a similar end, so why go down this path rather than some other?

When you interview developers about the code samples, include a few "why" questions:

- Are there other approaches that you rejected here? Why?
- Are users free to implement some other approach, or do we specifically want them to follow this method? Why?
- Why use this language/tool/framework/library rather than some other?

As for inserting inline comments regarding how, a best practice is to insert short comments for every 5-10 lines of code — not so many comments that you make the code unreadable, but not so few that you omit explanation of what's going on.

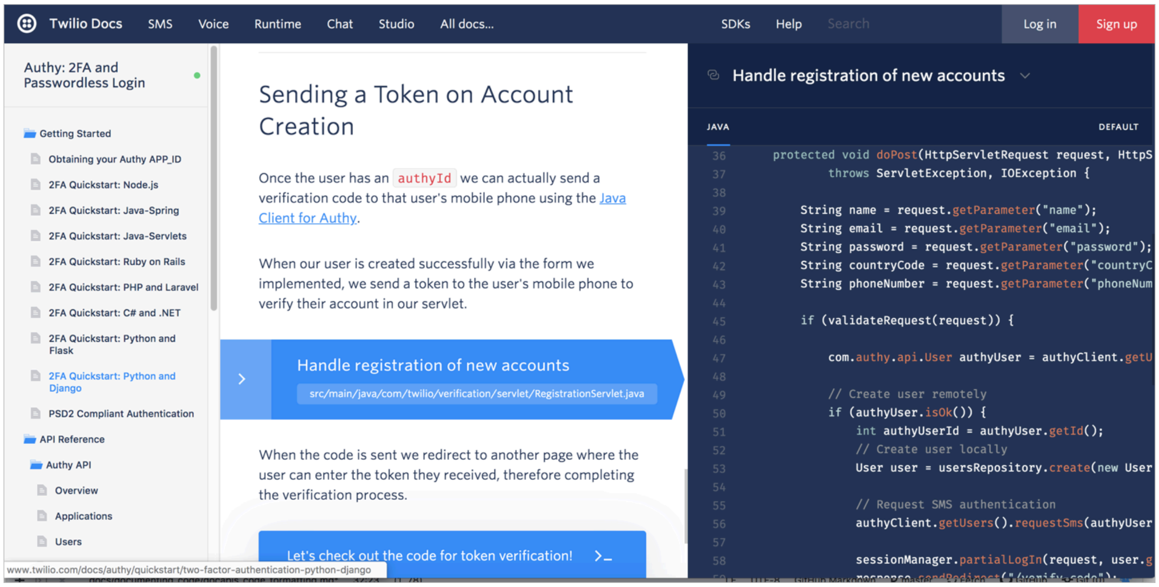
Also, note that inline code comments can be somewhat controversial. If you're just explaining what the code is doing, it can be redundant to someone who reads the language. As discussed in What research tells us about documenting code (p. 438), some developers feel that simple code documents itself — its meaning is clear to those who read the language, without the need for explanation. However, this statement tends to cater to advanced users and doesn't extend to cases where code is complex.

Approach 2: Juxtaposed commentary in a third column

Best practices for documentation in general (not just code documentation) involve placing helpful instruction next to the area of confusion, and within the context of code documentation, this might mean adding inline comments peppered throughout code. But suppose you want a longer running commentary about what's going on in the code (because the level of complexity can't be relayed in a short informal comment). How do you juxtapose your lengthy conceptual/explanatory information next to the code?

If your commentary dwarfs the code, you risk making the code unreadable. If you arrange the commentary in sections that come long before or after the code, you risk creating a chasm between the explanation and the code, such that readers won't know what parts of the code your explanation refers to.

One solution to this location problem for code docs is to create an additional, third column in your layout. You devote the middle column to your conceptual explanation and your right column to the code. This way the code and the narration are juxtaposed in a way that readers can glance at the code while reading your conceptual explanations — in other words, the third column maintains the needed context between the code and explanations. Here's an example from Twilio showing this juxtaposed approach:



Twilio Docs
SMS
Voice
Runtime
Chat
Studio
All docs...
SDKs
Help
Search
Log in
Sign up
Authy: 2FA and Passwordless Login
Getting Started
Obtaining your Authy APP_ID
2FA Quickstart: Node.js
2FA Quickstart: Java-Spring
2FA Quickstart: Java-Servlets
2FA Quickstart: Ruby on Rails
2FA Quickstart: PHP and Laravel
2FA Quickstart: C# and .NET
2FA Quickstart: Python and Flask
2FA Quickstart: Python and Django
PSD2 Compliant Authentication
API Reference
Authy API
Overview
Applications
Users
JAVA
DEFAULT

Sending a Token on Account Creation

Once the user has an `authyId` we can actually send a verification code to that user's mobile phone using the Java Client for Authy.

When our user is created successfully via the form we implemented, we send a token to the user's mobile phone to verify their account in our servlet.

Handle registration of new accounts

src/main/java/com/twilio/verification/servlet/RegistrationServlet.java

When the code is sent we redirect to another page where the user can enter the token they received, therefore completing the verification process.

Let's check out the code for token verification!

www.twilio.com/docs/authy/quickstart/two-factor-authentication-python-django

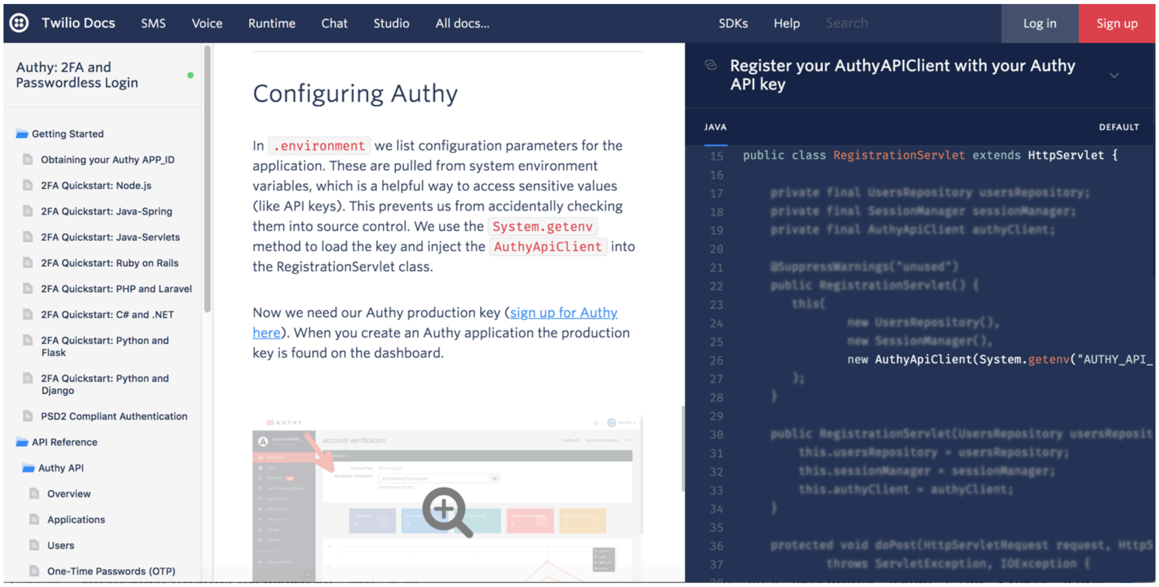
Handle registration of new accounts

```java
    protected void doPost(HttpServletRequest request, HttpS
            throws ServletException, IOException {

        String name = request.getParameter("name");
        String email = request.getParameter("email");
        String password = request.getParameter("password");
        String countryCode = request.getParameter("countryC
        String phoneNumber = request.getParameter("phoneNum

        if (validateRequest(request)) {

            com.authy.api.User authyUser = authyClient.getU

            // Create user remotely
            if (authyUser.isOk()) {
                int authyUserId = authyUser.getId();
                // Create user locally
                User user = usersRepository.create(new User

                // Request SMS authentication
                authyClient.getUsers().requestSms(authyUser

                sessionManager.partialLogIn(request, user.g
```

Twilio documentation juxtaposed with the code through a third column

In this example, the conceptual content and steps appear in the middle column, the code on the right, with a dark background behind the code to create visual contrast.

Some of Twilio's screens actually blur out the irrelevant code, allowing you to focus your attention on the lines articulated in the conceptual area, like this:


Twilio Docs
SMS
Voice
Runtime
Chat
Studio
All docs...
SDKs
Help
Search
Log in
Sign up
Authy: 2FA and Passwordless Login
Getting Started
Obtaining your Authy APP_ID
2FA Quickstart: Node.js
2FA Quickstart: Java-Spring
2FA Quickstart: Java-Servlets
2FA Quickstart: Ruby on Rails
2FA Quickstart: PHP and Laravel
2FA Quickstart: C# and .NET
2FA Quickstart: Python and Flask
2FA Quickstart: Python and Django
PSD2 Compliant Authentication
API Reference
Authy API
Overview
Applications
Users
One-Time Passwords (OTP)
JAVA
DEFAULT

Configuring Authy

In `.environment` we list configuration parameters for the application. These are pulled from system environment variables, which is a helpful way to access sensitive values (like API keys). This prevents us from accidentally checking them into source control. We use the `System.getenv` method to load the key and inject the `AuthyApiClient` into the RegistrationServlet class.

Now we need our Authy production key (sign up for Authy here). When you create an Authy application the production key is found on the dashboard.

Register your AuthyAPIClient with your Authy API key

```java
public class RegistrationServlet extends HttpServlet {

    private final UsersRepository usersRepository;
    private final SessionManager sessionManager;
    private final AuthyApiClient authyClient;

    @SuppressWarnings("unused")
    public RegistrationServlet() {
        this(
                new UsersRepository(),
                new SessionManager(),
                new AuthyApiClient(System.getenv("AUTHY_API_
        );
    }

    public RegistrationServlet(UsersRepository usersReposit
         this.usersRepository = usersRepository;
         this.sessionManager = sessionManager;
         this.authyClient = authyClient;
    }

    protected void doPost(HttpServletRequest request, HttpS
            throws ServletException, IOException {
```

Blurring out irrelevant code

One challenge with this juxtaposed approach is screen space. To pull off a third-column design, you need to occupy the whole screen, without margins. (I'm surprised Twilio doesn't include a switch to collapse the left-side navigation, which would give more space for the code.)

As is, the code is only partially visible (on my 13" Macbook Pro screen). The code extends horizontally with an option to scroll right, but surely the designers must have cringed in developing a UI that involves a healthy dose of back-and-forth horizontal scrolling.

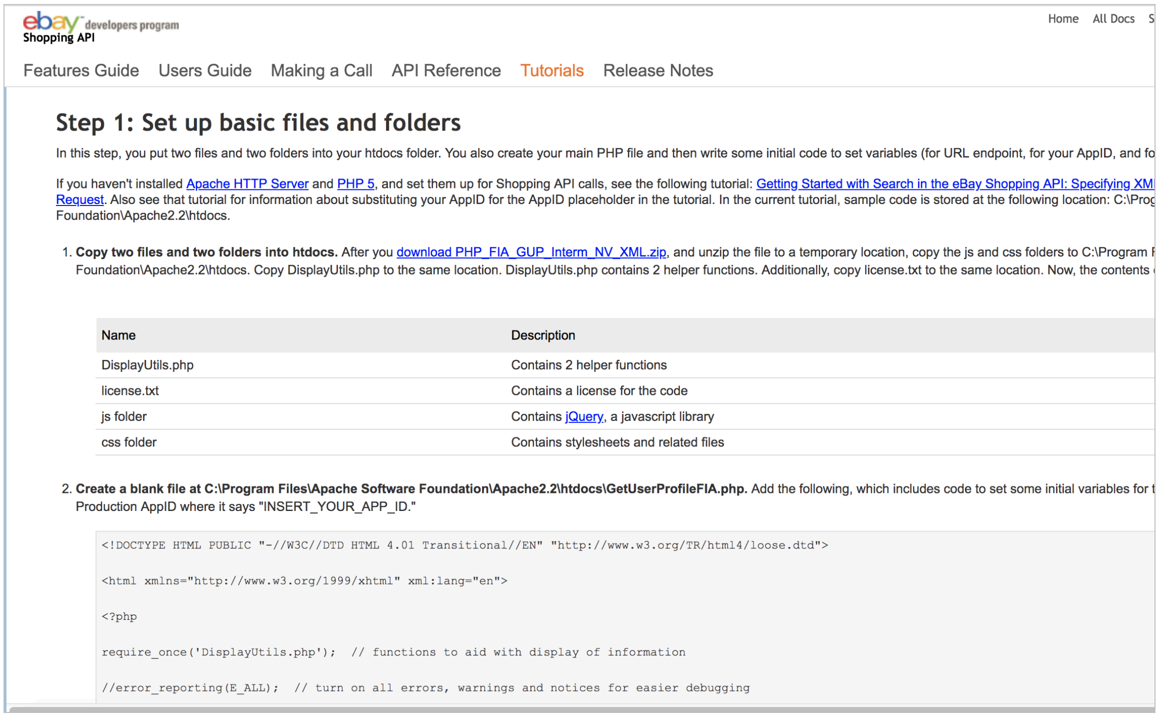
Additionally, implementing blur and focus views based on the line the user is at in the tutorial must be technically challenging and somewhat cumbersome to keep track of.

Another challenge is that the code and explanations of the code rarely line up all the way down. Suppose you have one method in the code that occupies just a single line, but describing this method occupies three long paragraphs of conceptual explanation. By the time the user reaches the bottom of the conceptual explanation, the referenced code is no longer juxtaposed. Now the user not only has to scroll horizontally to see the code but the user also has to scroll up and down the column to locate the relevant code. Designing a UI to accommodate all of these moving parts not only seems challenging, but also puts more burden on the user as well.

Another challenge with this juxtaposed design is that code is often spread out across multiple files. The view on the design column doesn't indicate whether all the code appears in the same Java file, for example, or whether we're seeing code from multiple Java files. Introducing tabs in your code column requires an even more complex documentation format. I doubt you can accomplish all this with Markdown syntax.

Approach 3: The Lego approach

Another approach is to build the code from the ground up level by level, which I'm calling the Lego approach. For an example of the Lego approach, take a look at this example from the eBay Shopping API: Searching By Seller: Reviewing Information About A Seller.



Step 1: Set up basic files and folders

In this step, you put two files and two folders into your htdocs folder. You also create your main PHP file and then write some initial code to set variables (for URL endpoint, for your AppID, and fo

If you haven't installed Apache HTTP Server and PHP 5, and set them up for Shopping API calls, see the following tutorial: Getting Started with Search in the eBay Shopping API: Specifying XM Request. Also see that tutorial for information about substituting your AppID for the AppID placeholder in the tutorial. In the current tutorial, sample code is stored at the following location: C:\Prog Foundation\Apache2.2\htdocs.

1. **Copy two files and two folders into htdocs.** After you download PHP_FIA_GUP_Interm_NV_XML.zip, and unzip the file to a temporary location, copy the js and css folders to C:\Program Foundation\Apache2.2\htdocs. Copy DisplayUtils.php to the same location. DisplayUtils.php contains 2 helper functions. Additionally, copy license.txt to the same location. Now, the contents

Name	Description
DisplayUtils.php	Contains 2 helper functions
license.txt	Contains a license for the code
js folder	Contains jQuery, a javascript library
css folder	Contains stylesheets and related files

2. **Create a blank file at C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\GetUserProfileFIA.php.** Add the following, which includes code to set some initial variables for Production AppID where it says "INSERT_YOUR_APP_ID."

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">

<?php

require_once('DisplayUtils.php');  // functions to aid with display of information

//error_reporting(E_ALL);  // turn on all errors, warnings and notices for easier debugging
```

Lego approach shown through eBay Shopping API

Their tutorial contains five steps:

> - Step 1: Set up basic files and folders
> - Step 2: Add code for making the GetUserProfile call and displaying the results
> - Step 3: Add code for making the FindItemsAdvanced call and displaying the results
> - Step 4: Add HTML and Javascript for the user interface
> - Step 5: Run the code

With the Lego approach, you might start with a blank file. Then with each step, you add more and more code until you have a fully working example. Similar to building with Legos — you start with a base and then start bolting on piece after piece until you finish the design you intended to build.

The problem with this Lego approach is that as a technical writer, you likely won't have the build-up logic that the developer followed. Developers will more likely just send you the finished piece of code to document, and then you might end up approaching it as I described in the intro to this section (p. 434), where I divided my explanations of the code section by section.

Tackling code explanations section by section won't necessarily match the Lego build order of the code, since code is non-linear. The code that appears at the top might have been like the icing on the cake for the developer — e.g., abstracting more complicated lines into variables that he or she uses to reduce the code's working complexity. Finished code often has logic that is abstracted away into variables or other referenced functions so that certain parts of the final code remain cleaner and more concise, but other parts become more opaque. Finished code is often too messy and confusing to document in any teachable way.

Despite the challenges with the Lego approach, if you want to teach someone how to understand code, you have to start simple and work your way up. The next technique explains this simple-to-complex approach through the metaphor of the nautilus.

Approach 4: The Nautilus approach

The Nautilus approach is similar to the Lego approach, but rather than describing chunks of work that are tackled one by one in an assembly order, you describe the core, simple patterns that users need to know. You start with the simplest code and then let the user's project grow larger and larger as needed, like the growing spiral pattern of the nautilus' shell.

Paul Gustafson, who manages a technical writing staffing company in the Bay area called Expert Support ↗, presents us with the metaphor of the nautilus in the approach described here. Paul says that the nautilus provides a good metaphor for technical communication because the nautilus follows a spiral pattern (Fibonacci sequence) that allows it to start small and gradually grow larger and larger as needed:

Nautilus

Paul writes:

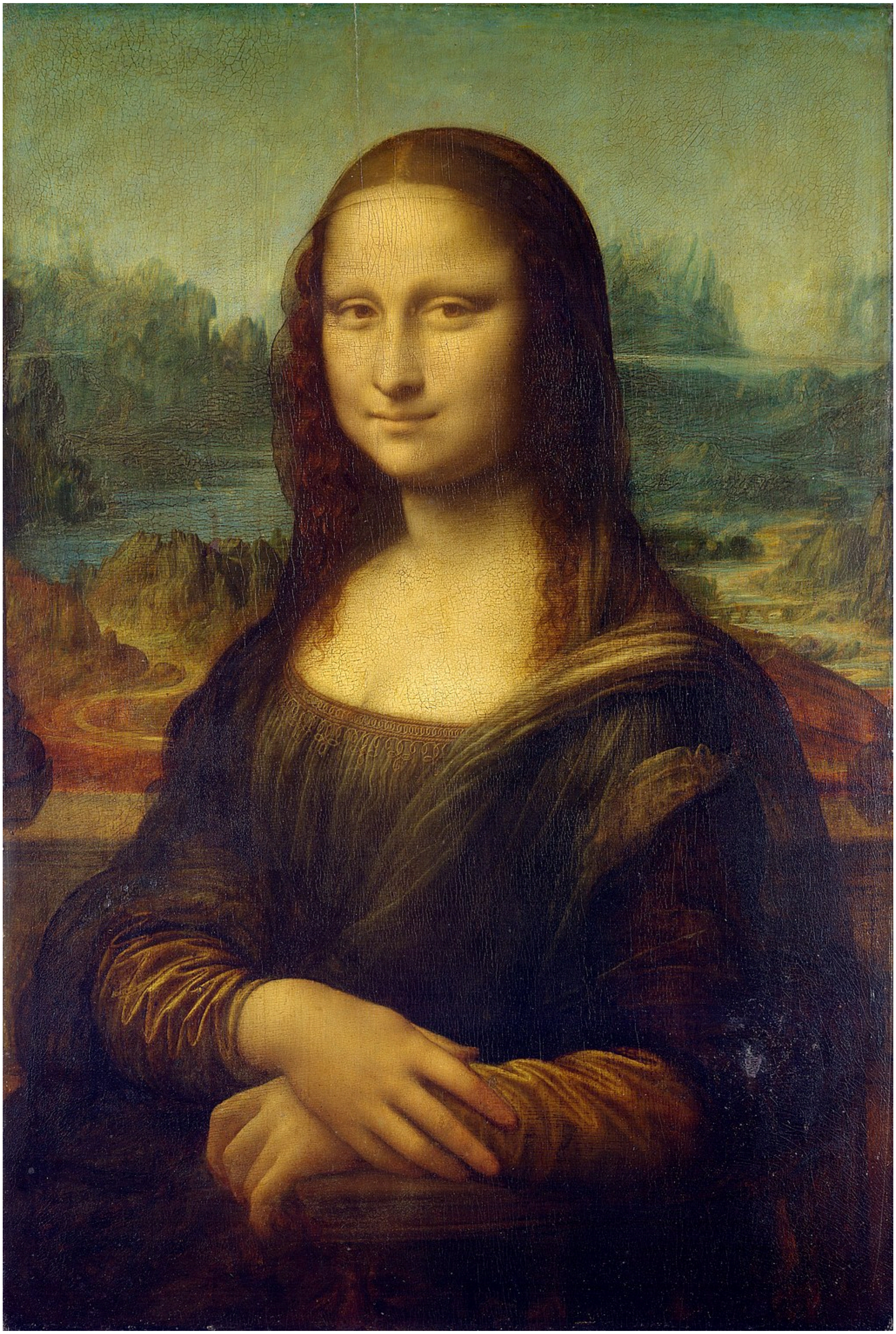
> Fostering understanding, which is what technical communications is all about, happens most efficiently by following a similar pattern….
>
> When your understanding is small, you learn best when the first lesson imparts information for a small, simple task with traits importantly akin to the first nautilus chambers.
>
> … The good news for the nautilus is that the small chambers follow the same basic plan as the bigger chambers. If the first tasks a learner masters are fundamentally similar to more complex tasks farther down the syllabus, the student begins to understand and apply those patterns. The sooner newbies learn to “think about things the right way,” the sooner they “get it,” which is exactly what both the instructor and the student want. (Lessons from a cephalopod)

Following the nautilus approach, you start with the simple, core patterns and then build up more and more code around it gradually as needed. You don’t start by describing the complex finished work from the start. That finished work likely involves all kinds of code abstractions and rearrangements for a clean, finished product.

I like the Nautilus approach. The problem is that we often want to explain how the finished code works, taking a user through it all from start to end. We might have 500 lines of code that we want to articulate, but the Nautilus approach would have us explain just several small pieces of that code (at least to begin with). Hence there’s an A-versus-Z type of problem: We describe A (the simplest core pattern), but the end product is Z. How exactly do you get from A to Z? How do you get from the simple patterns that maybe occupy 20 lines of code to the monstrously complex, finished code that spans 500 lines?

To the technical writer looking at the finalized code, there’s no clear sense of how the developer got there. We often can’t decouple the nautilus-like logic that the developer started with, which led him or her toward this more complex end. All we see is this complex end. How do you decompile the code to reconstruct the logic that the developer started with? How do you know what these initial nautilus patterns were that started the whole process? If you didn’t develop the code, nor are you a developer, it will be nearly impossible to reconstruct the nautilus pattern behind the code in your explanations.

As another analogy, consider teaching others how to paint. Suppose your task is to describe a finished painting to a would-be painter. You need to describe how to paint the Mona Lisa:



How would you document the process of producing a painting?

To document the process of producing this painting, would you start at the top and work your way to the bottom? No, that would be ludicrous. Most likely you would start by creating ovals for the head. Then maybe some general sketches for the eyes, and so on. Maybe you sketch our perspective lines and other basic structure lines first. You wouldn't get to the colors and lighting and shadows until later, right? Same with code. You start with the foundation and then work your way towards more of the finishing detail.

However, if you're not a painter, how would you know how to describe the process of creating a painting? You would need to know the painter's logic — where to start, and how to proceed toward the end. If you instead just started at the end and tried to work your way backward, the tutorial would likely be insanely complex.

To illustrate this point more clearly, let me provide a code example. Although I'm not an engineer, I'm handy with Jekyll and theming, and the other day I set about creating a template that would take a content export from a ticketing tool (like JIRA) and render it as a documentation report on a web page.

My finished template looks like this:

```
<div class="sprintDuration">{{page.duration}}</div>

{% assign sprintYamlFile = page.sprint_yaml_file %}

<div class="metaReportInfo" markdown="span">
Tech writers: {% for member in page.team_members %}<a href="https://somesite.company.com/users/{{member}}">{{member}}</a>{% unless forloop.last %}, {% endunless %}{% endfor %}<br/>
Team: <a href="https://ourteamsite.company.com/">DevComm</a><br/>
Sprint: <a href="{{page.sprint_link}}">Link</a>
</div>

<div id="top"></div>
<div class="all-items">

<h2 id="high-level-summary">High-level Summary</h2>

{{page.high_level_summary}}

{% include sprintdisplaylogic.html %}

{% assign sprintYamlFileOpen = page.sprint_yaml_file_open %}

<h2>Open Issues</h2>
<p>ACME project has <b>{{page.open_items_acme}}</b> open issues. Beta project has has <b>{{page.open_items_beta}}</b> open issues.</p>

{% include sprintdisplaylogic_open.html %}

{% include sprintstylesandscripts.html %}
```

This code looks kind of like gibberish, really. I have some “include” files where I’ve abstracted away the logic because I’ll be repeating it from report to report. I don’t want the scripts and styles showing here, as they’ll clutter up the code, so I’ve abstracted them away into include files as well.

Imagine trying to document this code. If you started from the top and worked your way to the bottom, it would be a real mess. The lengthy explanation would also be hard to read and understand for users. It’s just a confusing ball of yarn. It doesn’t help that I put this together in haste, without much thought for a clean, elegant solution. I needed to get this report out fast, so I hacked together the template as quickly as I could. Developers building applications often implement similar hacks and other quick-fixes using “duct tape and WD-40”, as Joel Spolsky says, to get a working solution shipped to meet a deadline.

This kind of finalized code, with all of its quick hacks and inelegant implementations, is not instructive to someone looking to build their own report template. Instead, it would be more useful if I started from scratch with the core pattern. That pattern involves looping through a JSON file (the ticketing export) and pulling out the key values that I want to display. This key logic is available in the `sprintdisplaylogic.html` include above. Here’s the contents of that include:

```
{% assign shortIdList = page.short_ids %}
{% for item in shortIdList %}
{% assign sprintItems = site.data.sprints[sprintYamlFile]  | where_exp:"entry",
"entry.ShortId contains item" %}

<h2 id="{{item}}">{{item}} Resolved Doc Issues</h2>
{% for entry in sprintItems %}
<div class="entryTitle">{{entry.Title}}</div>
<div class="entryIssueUrl"><a href="{{entry.IssueUrl}}">{{entry.ShortId}}</a></div>
<blockquote>{{entry.Description | markdownify }}</blockquote>
{% endfor %}
<small><a href="#top">↑ Back to top</a></small>
{% endfor %}
```

Even this is confusing, as I have some weird stuff going on here with variables inserted as brackets in YAML file references.

To really pare this down into the core logic, this is what developers would start with:

```
{% assign sprintItems = site.data.sprints.someyamlfile %}
{% for entry in sprintItems %}
* {{entry.Title}}
* {{entry.Description }}
* {{entry.IssueUrl}}
{% endfor %}
```

This is the core logic of the report. It uses a `for` loop to look through items in a data file (accessed through `site.data.sprints.someyamlfile`), and then Jekyll prints these values out through tags like ``. Once users learn these core patterns, they can extend them into creating more complex solutions, such as inserting variables into the loop so that you can repeat the logic without duplicating the loop for every report category.

But if you didn't develop the code, it would be extremely difficult to pinpoint the core, simple logic that is the basic pattern of the code. Where did the developer start? What is the essential pattern to learn?

To gather this information, you need to interview the developer. And when you interview the developer, you need to understand the language and explanations he or she communicates. Alternatively, you can try to steer the developer towards describing the logic of his or her own code through guidance with the right questions. For example, you might ask, What's the fundamental core logic that the user needs to know here? What's the really important line to focus on in all this code? Can you walk me through that specific part in detail?

Just a caution here — chances are if you ask developers these questions, they will quickly slip into advanced explanations and jargon and become impatient if you're unable to follow the logic. If basic concepts are totally unfamiliar, it is unrealistic for a beginning to simply soak in the explanation. Plus, it can be hard to explain code. I even forgot many details about how my Jekyll code works about a hour after I wrote it. These explanations require a lot of mental work for developers to perform (their reluctance reinforces why they're often not writing the documentation in the first place).

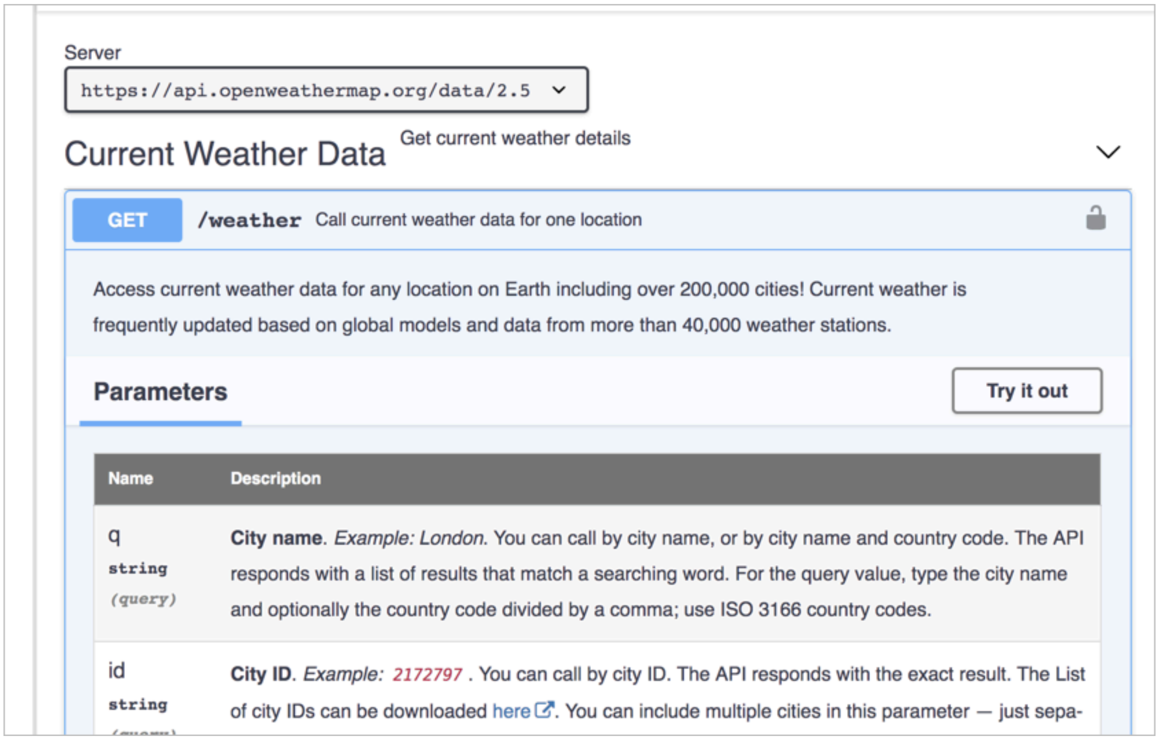
In these conversations, I highly recommend recording your interviews with developers. This will allow you to go back and listen to the explanations in slow motion, hitting stop and rewind as much as you want. If the engineer mentions an unfamiliar concept, you can use that as a springboard for your own study. This will give you a relevant list of topics to learn (rather than following some generic course that might never get around to addressing the specific code you actually need to know). At the very least, you could simply parrot out what the developer said from the recording — you'll have all the lingo and phrasings ready to pull from.

As you decompile the logic of the code to the simplest pattern, you will face another challenge: where to draw the line about what to explain and what not to explain. This again is nearly impossible to answer without a more thorough understanding of your audience, and chances are the engineer won't have any more sense of the audience than you, so the engineer will likely make the same assumption that technical writers often do — imagining a user not too unlike ourselves.

Approach 5: Interactive browser experiences

Teaching core patterns shifts the documentation more into the domain of a tutorial. Related to this type of learning are interactive browser experiences that combine action with learning. These browser-executable interfaces have as their goal the aim to help users better understand the results of inputs and outputs so that users can see for themselves how the code works through a more hands-on, try-it-for-yourself approach.

The most common example of interactive documentation for APIs is with Swagger UI, which I have already covered at length in OpenAPI tutorial using Swagger Editor and Swagger UI: Overview (p. 272) and showed in the Swagger UI Demo (p. 240):


Server
https://api.openweathermap.org/data/2.5
Current Weather Data
Get current weather details
GET
/weather
Call current weather data for one location
Parameters
Try it out

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

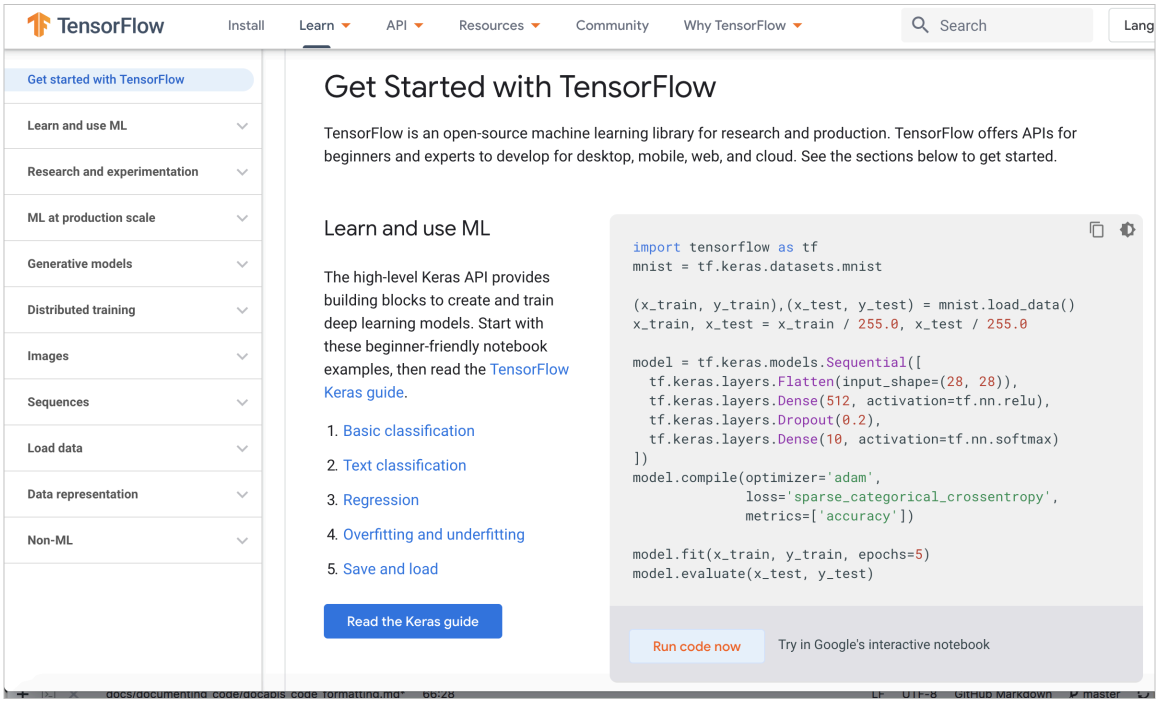
Name	Description
q `string` *(query)*	**City name**. *Example: London*. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by a comma; use ISO 3166 country codes.
id `string`	**City ID**. *Example:* `2172797` . You can call by city ID. The API responds with the exact result. The List of city IDs can be downloaded here. You can include multiple cities in this parameter — just sepa-

Try it out button in Swagger's interactive REST API interface

Swagger provides an ingenious blending of documentation and try-it-out interactions that help users learn your API (by both reading and doing). But making requests with REST API endpoints tends to be somewhat simple. More extensive code tutorials will be harder to make interactive in the browser. Even so, some “Notebooks” (as they’re often called) allow you to run code, specifically Jupyter Notebooks. Jupyter explains:

> The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

Google has some collaborative notebook options with TensorFlow documentation, which has operations that you can execute on web pages. In the following screenshot, you can see an option to “Run code now”:



Get Started with TensorFlow

TensorFlow is an open-source machine learning library for research and production. TensorFlow offers APIs for beginners and experts to develop for desktop, mobile, web, and cloud. See the sections below to get started.

Learn and use ML

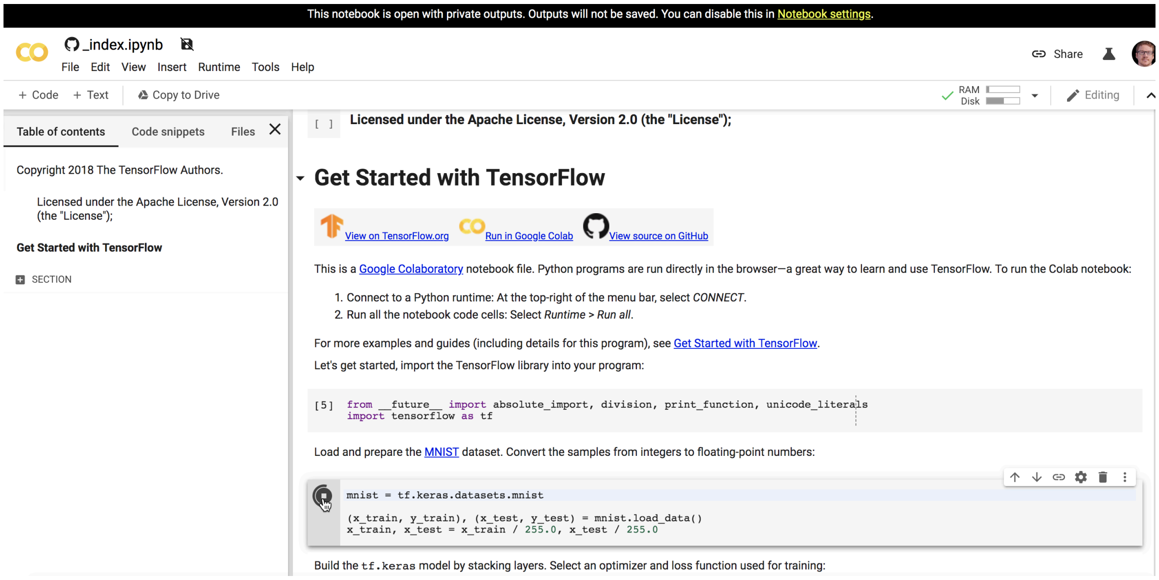
The high-level Keras API provides building blocks to create and train deep learning models. Start with these beginner-friendly notebook examples, then read the TensorFlow Keras guide.

1. Basic classification
2. Text classification
3. Regression
4. Overfitting and underfitting
5. Save and load

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Interactive code examples from TensorFlow

Clicking “Run code now” takes you to Google’s interactive notebooks that actually run the code in the browser:



This notebook is open with private outputs. Outputs will not be saved. You can disable this in Notebook settings.

Copyright 2018 The TensorFlow Authors.

Licensed under the Apache License, Version 2.0 (the "License");

Get Started with TensorFlow

[ ] **Licensed under the Apache License, Version 2.0 (the "License");**

Get Started with TensorFlow

This is a Google Colaboratory notebook file. Python programs are run directly in the browser—a great way to learn and use TensorFlow. To run the Colab notebook:

1. Connect to a Python runtime: At the top-right of the menu bar, select *CONNECT*.
2. Run all the notebook code cells: Select *Runtime* > *Run all*.

For more examples and guides (including details for this program), see Get Started with TensorFlow.

Let's get started, import the TensorFlow library into your program:

```
[5] from __future__ import absolute_import, division, print_function, unicode_literals
    import tensorflow as tf
```

Load and prepare the MNIST dataset. Convert the samples from integers to floating-point numbers:

```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Build the `tf.keras` model by stacking layers. Select an optimizer and loss function used for training:

Google's interactive notebooks let you run the code in the browser

Although interactive notebooks look cool, they seem like a lot of work for something that could more easily be accomplished with a sample app. Instead of figuring out how you can compile Python code or some other language in the browser, why not just provide a sample app that users can download and then proceed through locally, using their own compiling tools and setup?

Sure, users might need to have some utilities and frameworks installed on their computer to make the sample apps run, as well as an IDE, but making code run in the browser might not fully inform users about all the required setup and preparation that will eventually be necessarily for them to make the code run locally. Browsers tend to be somewhat stiff and formal in what they can do — users might have more freedom to experiment (and learn from those experiments) with a sample app.

Conclusion

In this section, we looked at five approaches to documenting code:

- Approach 1: Separate the "how" into inline comments, the "why" into external docs
- Approach 2: Juxtaposed commentary in a third column
- Approach 3: The Lego approach
- Approach 4: The Nautilus approach
- Approach 5: Interactive browser experiences

Overall, use the approach that helps empower users with the knowledge for creating their own code.

Activity: Classify code documentation into one of the five types

Find a code tutorial within one of the API documentation sites you've been exploring. Or look at one of the API doc sites in this post by Nordic APIs on 5 Examples of Excellent API Documentation (and Why We Think So) ↗. This post lists five API doc sets: Stripe ↗, Twilio ↗, Dropbox ↗, GitHub ↗, and Twitter ↗.

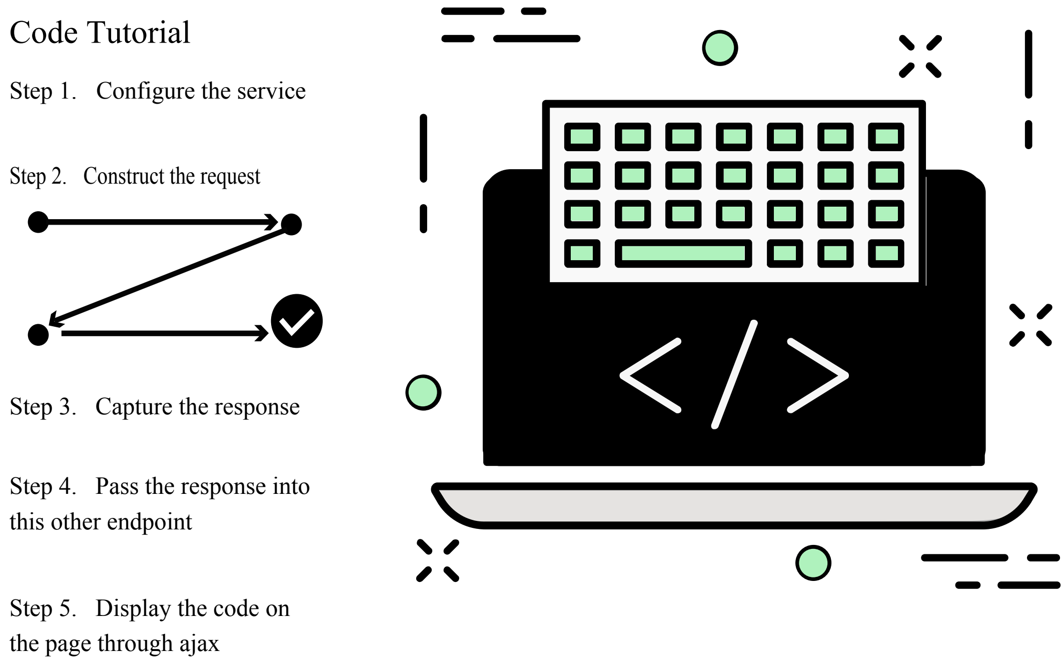
Find a code tutorial in the documentation. Figure out which approach the code documentation best aligns with.

Code samples

Developer documentation tends to include a lot of code samples. These code samples might not be included with the endpoints you document, but as you create tasks and more sophisticated workflows about how to use the API to accomplish a variety of goals, you'll end up leveraging different endpoints and showing how to address different scenarios. Code tutorials are a crucial part of your user guide.

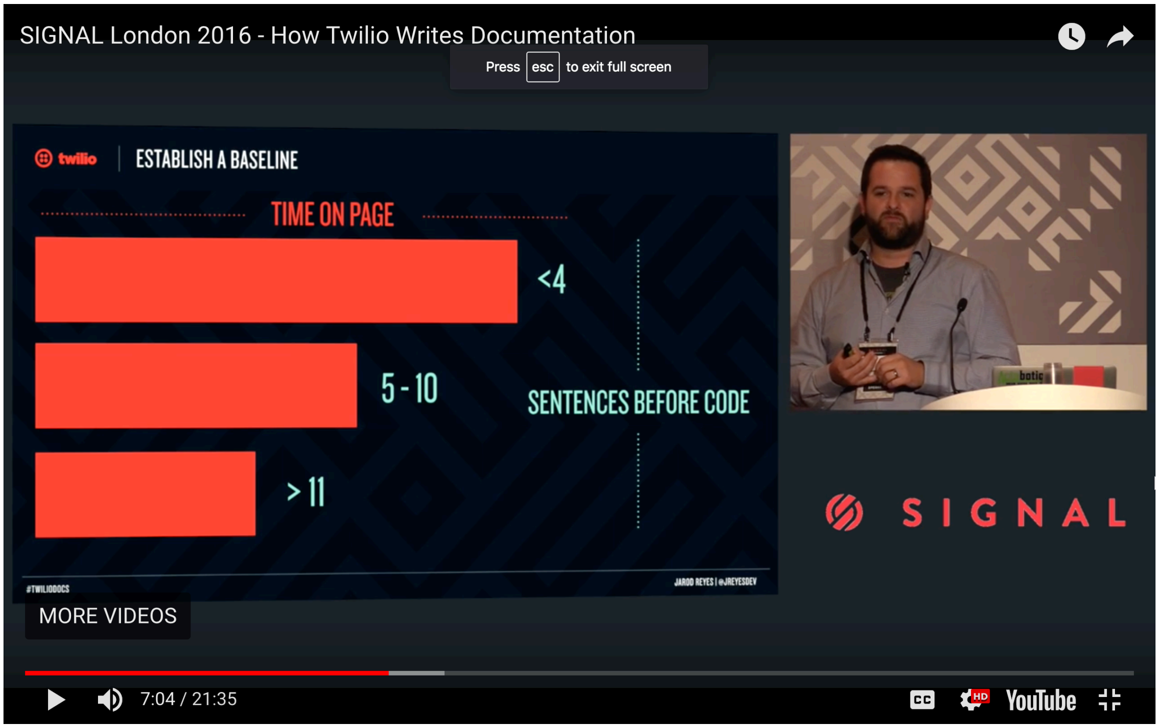
Code samples are like candy for developers

Code samples play an essential role in helping developers use an API. Code is literally another language, and when users who speak that language see it, the code communicates with them in powerful ways that non-code text (however descriptive it is) can't achieve.


Code Tutorial
Step 1. Configure the service
Step 2. Construct the request
Step 3. Capture the response
Step 4. Pass the response into this other endpoint
Step 5. Display the code on the page through ajax
</ >

Code is in another language, so as much as you might try to describe the communication in this other language through text, it often falls short. When developers see code, they can often read the code and understand it natively.

In user testing that Twilio did with their documentation, they found that pages that started more quickly with code samples performed better with users.



Twilio found that when pages had fewer sentences before code samples, the pages engaged users more.

Specifically, pages with less than 4 sentences before code samples performed twice as well as pages with 11 sentences before code samples. Jarod Reyes explains:

> It’s a mental block more than it is not being able to see code. It tells a developer that this page has a lot to say, and that they have a lot to do. They don’t want to necessarily want to spend the time to read what you want to say. We saw this across section length; we saw this across page depth. Any time that there is a lot of prose on a page and not a lot of code, that page didn’t perform well. (How Twilio Writes Documentation)

In other words, when developers see code, it’s the equivalent of seeing a task-based topic with a user guide — the code indicates a concrete action for the developer to take. This attracts the developer’s attention.

Don’t just provide reference docs

Sometimes engineers want to avoid including code samples in API docs because they feel the endpoint reference documentation contains all the information developers need and stands on its own. However, this view is often shortsighted. In an article on the Programmable Web called “The Six Pillars of Complete Developer Documentation,” the authors explain:

> While a developer's guide should walk a developer through the basic usage of an API and its functionality, it can't cover every possible use of that API in a coherent way. That is where articles and tutorials come in, to teach developers tangential or specialized uses of an API, like combining it with another service, framework, or API. (Note: Programmable Web has since shut down.)

In other words, the articles and tutorials complement the reference documentation to provide complete instruction. Code samples that show how to use the various endpoints to achieve a goal occupy an important space in your user guide.

Additionally, even if including code samples, the level of detail and explanation is also somewhat debatable. Many developers assume that the developer audience has a skill set similar to their own, without recognizing different developer specializations. And so they will add a code sample but not give any explanation about it. Internal developers will often say, "If the user doesn't understand this code, he or she shouldn't be using our API."

If you encounter this attitude, remind developers that users often have technical talent in different areas. For example, a user might be an expert in Java but only mildly familiar with JavaScript. Someone who is a database programmer will have a different skill set than someone who is a Python programmer, who will have a different skill set from a front-end web development engineer, and so on. Given these differences and the likely possibility that you will have many novice (or unfamiliar) users, more extensive code tutorials and explanations are warranted.

Focus on the why, not the what

Once you have code samples in your documentation, the next question is how to document them. User interfaces have clear tasks — buttons to click, linear workflows, etc. But documenting code can be more of a conundrum. Remember this basic principle: In any code sample, focus your explanation on the *why*, not the *what*. Explain why you're doing what you're doing, not the detailed play-by-play of what's going on, especially when the *what* refers more to standard programming mechanics that aren't unique to your API.

Here's an example of the difference:

- **what**: "In this code, several arguments are passed to jQuery's `ajax` method. The response is assigned to the data argument of the callback function, which in this case is `success`."
- **why**: "Use the `ajax` method from jQuery because it allows for asynchronous responses that won't block the loading of your page.

Developers unfamiliar with common code not related to your company (for example, the `.ajax()` method from jQuery) should consult outside sources for tutorials about that code. Don't write your own version of documentation for another programming language or service. Instead, focus on the parts of the code unique to your company. Let the developer rely on other sources for the rest (feel free to link to other sites).

Add both code comments and before-and-after explanations

Your documentation regarding the code should mix code comments with some explanation either before or after the code sample. Different languages have different conventions for comments, but generally brief code comments are set off with forward slashes `//` in the code; longer comments are set off between slashes and asterisks, like this: `/* .... */`.

Comments within the code are usually short one-line notes that appear after every 5-10 lines of code. You can follow up this code with more robust explanations in your documentation, but it's ideal to pepper code samples with comments because it puts the explanation next to the code doing the action. This approach of adding brief comments within the code, followed by more robust explanations after the code, aligns with principles of progressive information disclosure that help align with both advanced and novice user types. In this case, progressive information disclosure means you provide some detail in the context of an activity, and then add links or references for more information if the user needs it.

If you have comments interspersed in code as well as in conceptual sections before or after the code, won't that be somewhat redundant? Not really. Some research about how developers use documentation found that there are two common behaviors: developers who start in code and read higher-level conceptual documentation only as needed (called "opportunistic" behavior). And developers who start in higher-level conceptual documentation before working their way down to code (called "systematic" behavior). Michael Meng, Stephanie Steinhardt, and Andreas Schubert explain:

> Once a high-level understanding of the API purpose and features has been formed, two different pathways seem to emerge that closely resemble the "systematic" and the "opportunistic" developer personas described by Clarke (2007) (see also Stylos, 2009). According to Clarke (2007), developers represented by the systematic developer persona work top down in the sense that they try to get a deeper understanding of the system as a whole before turning to individual components. On the other hand, the learning goals of opportunistic developers are more narrowly focussed on solving a particular problem and dependent on the specific issues and blockers they encounter while working toward a solution ("Application Programming Interface Documentation: What Do Software Developers Want?" *Journal of Technical Writing and Communication*. 2018, Vol. 48(3) 295–330. ResearchGate)

For the opportunistic developer that first starts in the code, comments in the code can provide helpful documentation to get the developer oriented quickly. But not everyone starts in the code. Some prefer to read the conceptual overviews first.

For more research on how to document code, see "When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects" by Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight (2018 *ACM/IEEE 40th International Conference on Software Engineering.* ResearchGate). The researchers explore whether developers are more apt to look in header files (where more formal descriptions of the class and methods appear) or implementation code for documentation (they focused on C++). In some cases, reading the implementation code directly provides a clearer path to understanding for developers. Additionally, some developers distrust that documentation is up to date and so prefer to look at the code directly. For more complex code, however, learning from more elaborate documentation in header files was helpful.

Overall, not every programmer reads code the same way. But based on the research, it's a good idea to include comments directly in the code as well as more formal explanations outside the code. If developers give you a chunk of code that has comments peppered throughout, don't assume that the code comments are somehow separate from documentation or outside your stewardship as a technical writer. Think of comments in code as the equivalent of context-sensitive help in a user interface — in many ways, this might be the most read content of all.

Keep code samples simple

Code samples should usually be stripped down to their simplest possible form. Providing code for an entire HTML page is probably unnecessary. But including some surrounding code doesn't hurt anyone, and for newbies, it can help them see the big picture. (It's also easier to copy and paste.)

Additionally, avoid including a lot of styling or other details in the code that will potentially distract the audience from the main point. The more minimalist the code sample, the better. For example, if you're showing a simple JavaScript function, you might be tempted to support it with elaborate CSS styling so that the demo looks sharp. However, all the extra CSS will only introduce more complexity and confusion that competes with the original principle you're trying to show with the code sample.

When developers take the code and integrate it into a production environment, they'll probably make a lot of changes to account for scaling, threading, and efficiency, and other production-level factors. But don't start out this way just to have a polished and professional looking demo.

Make code samples copy-and-paste friendly

Many times developers will copy and paste code directly from the documentation into their application. Then they will usually tweak it a little bit for their specific parameters or methods.

If you intend for users to copy and paste the code, make sure it works. When I first used some sample `ajax` code from a code tutorial on an API site, the `dataType` parameter was spelled `datatype`. As a result, the code didn't work (it returned the response as text, not JSON). It took me about 30 minutes of troubleshooting before I consulted the `ajax` method ↗ and realized that it should be `dataType` with a capital `T`.

Ideally, test out all the code samples yourself (or implement a more robust process for testing code). Testing allows you to spot errors, understand whether all the parameters are complete and valid, and more. In the earlier video from Twilio, the authors say they wanted to treat code samples in documentation like their other engineering code, so they stored their code in a separate container (also pushed to GitHub) to run regular tests on the code. They pulled the code into their documentation where appropriate. For lengthy code samples, consider storing the code on GitHub. This way engineers can more easily test it as part of their test cases for each release. Sometimes when code blocks are buried in documentation, they're overlooked with new releases.

Provide a sample in your target language

With REST APIs, developers can use pretty much any programming language to make the request. One question will inevitably arise: Should you show code samples that span across several languages? If so, how many languages?

Providing code samples is almost always a good thing, so if you have the bandwidth to show code samples in various languages, go for it. However, providing just one code example in your audience's target language is probably enough. If there isn't a standard language for most users, you could also just provide the curl examples in your docs, and then provide users with a Postman collection (p. 71) or an OpenAPI specification file (p. 181) — both of these approaches will allow developers to generate code samples in many different languages.

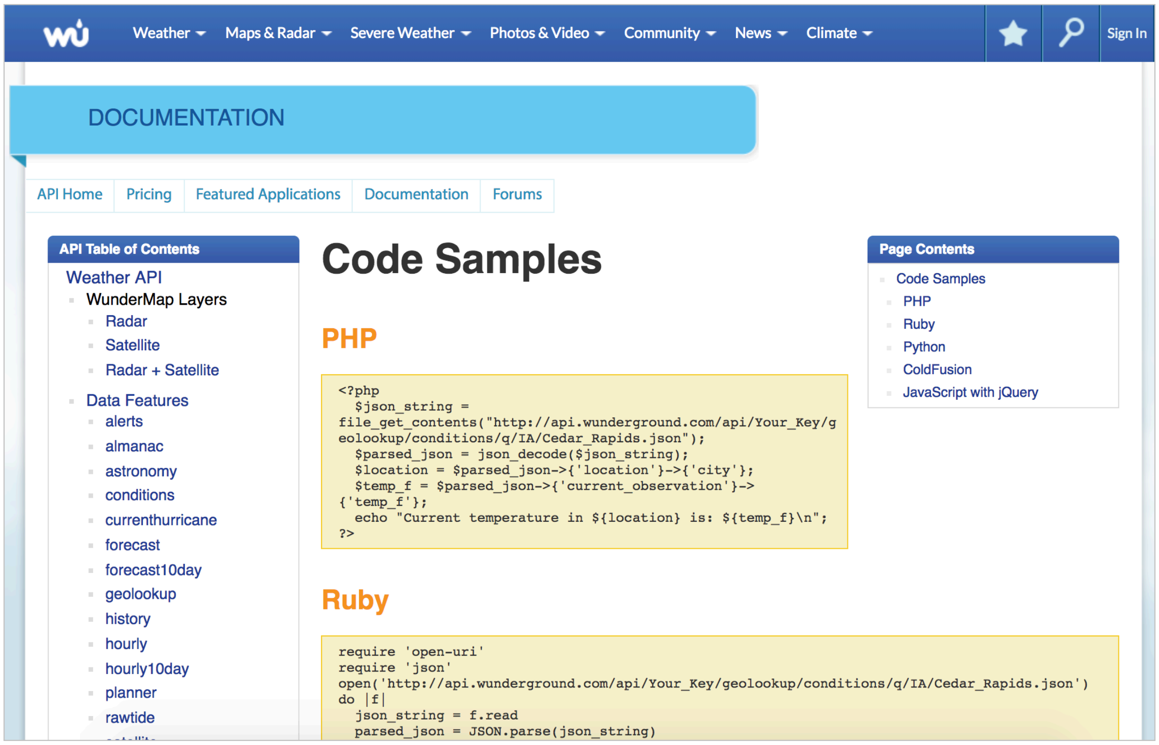
Remember that each code sample you provide needs to be tested and maintained. When you make updates to your API, you'll need to update each of the code samples across all the different languages. When your API pushes out a new release, you'll need to check all the code samples to make sure the code doesn't break with the changes in the new release (this is called "regression testing" in QA lingo).

Including a lot of code samples increases the amount of testing and maintenance, but this is the most helpful type of content for users. Take an approach that you can support and maintain.

Sample code tutorials

The following are a few samples of code tutorials in API documentation.

Weather Underground



```
<?php
  $json_string =
file_get_contents("http://api.wunderground.com/api/Your_Key/g
eolookup/conditions/q/IA/Cedar_Rapids.json");
  $parsed_json = json_decode($json_string);
  $location = $parsed_json->{'location'}->{'city'};
  $temp_f = $parsed_json->{'current_observation'}->
{'temp_f'};
  echo "Current temperature in ${location} is: ${temp_f}\n";
?>
```

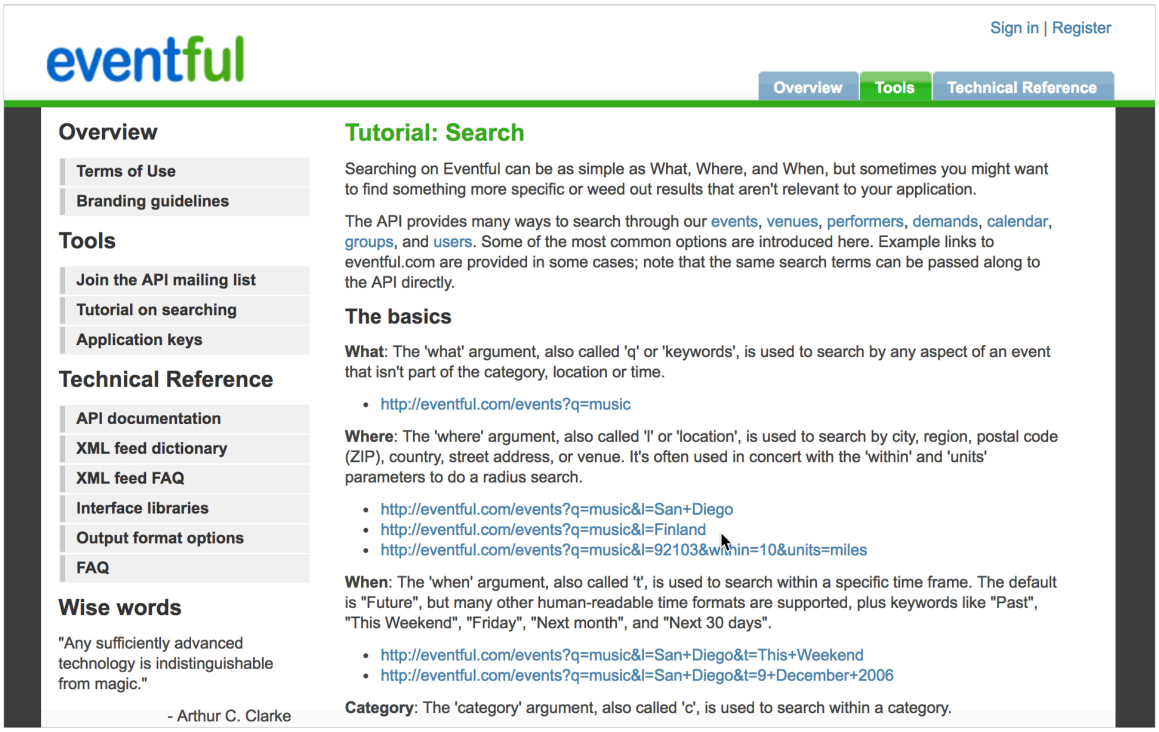
```
require 'open-uri'
require 'json'
open('http://api.wunderground.com/api/Your_Key/geolookup/conditions/q/IA/Cedar_Rapids.json')
do |f|
  json_string = f.read
  parsed_json = JSON.parse(json_string)
```

Weather Underground code samples

In this Weather Underground example, there are various code samples across half a dozen languages, but no explanation about what the code sample returns. In this case, the code is probably simple enough that developers can look at it and understand from the code itself what's going on. Still, some explanation is usually warranted, especially if there are multiple ways to make the call.

Sometimes developers will tell you that code is "self-documenting," meaning it's evident from the code itself what's going on. Without a knowledge of the programming language, it's hard to evaluate this statement. If you encounter this question, consider checking this assertion with some other engineers, especially outside the product team (or with users, if you have access to them).

Eventful



"Any sufficiently advanced technology is indistinguishable from magic."

- Arthur C. Clarke

Tutorial: Search

Searching on Eventful can be as simple as What, Where, and When, but sometimes you might want to find something more specific or weed out results that aren't relevant to your application.

The API provides many ways to search through our events, venues, performers, demands, calendar, groups, and users. Some of the most common options are introduced here. Example links to eventful.com are provided in some cases; note that the same search terms can be passed along to the API directly.

The basics

What: The 'what' argument, also called 'q' or 'keywords', is used to search by any aspect of an event that isn't part of the category, location or time.

- http://eventful.com/events?q=music

Where: The 'where' argument, also called 'l' or 'location', is used to search by city, region, postal code (ZIP), country, street address, or venue. It's often used in concert with the 'within' and 'units' parameters to do a radius search.

- http://eventful.com/events?q=music&l=San+Diego
- http://eventful.com/events?q=music&l=Finland
- http://eventful.com/events?q=music&l=92103&within=10&units=miles

When: The 'when' argument, also called 't', is used to search within a specific time frame. The default is "Future", but many other human-readable time formats are supported, plus keywords like "Past", "This Weekend", "Friday", "Next month", and "Next 30 days".

- http://eventful.com/events?q=music&l=San+Diego&t=This+Weekend
- http://eventful.com/events?q=music&l=San+Diego&t=9+December+2006

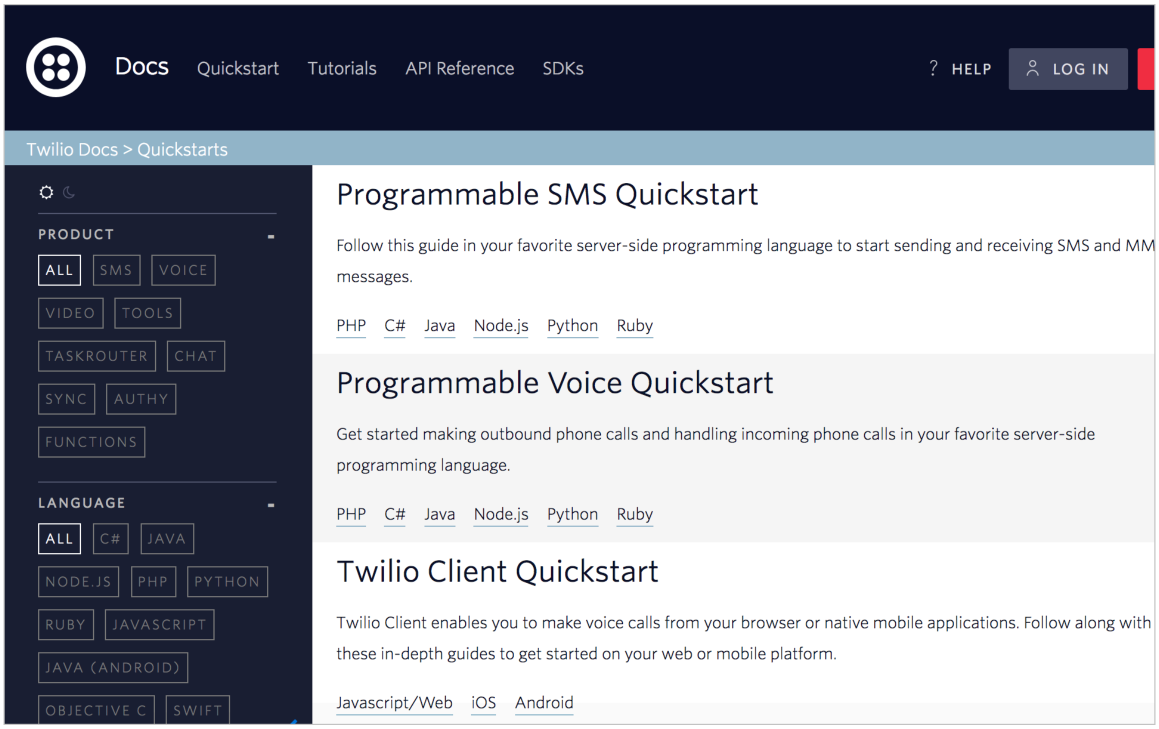
Category: The 'category' argument, also called 'c', is used to search within a category.

Eventful code samples

You won't see chunks of code here, but the Eventful docs include various examples about query string parameters for the endpoints. Although these parameters are also defined in their reference documentation for the search endpoint, the tutorial here expands on how to use the parameters in a more friendly, detailed way.

I like the Eventful tutorial because it shows how documentation that is usually contained in reference material can be pulled out and explained more narratively with examples. It shows more of the difference between reference and tutorial information.

Twilio



Programmable SMS Quickstart

Follow this guide in your favorite server-side programming language to start sending and receiving SMS and MM messages.

PHP C# Java Node.js Python Ruby

Programmable Voice Quickstart

Get started making outbound phone calls and handling incoming phone calls in your favorite server-side programming language.

PHP C# Java Node.js Python Ruby

Twilio Client Quickstart

Twilio Client enables you to make voice calls from your browser or native mobile applications. Follow along with these in-depth guides to get started on your web or mobile platform.

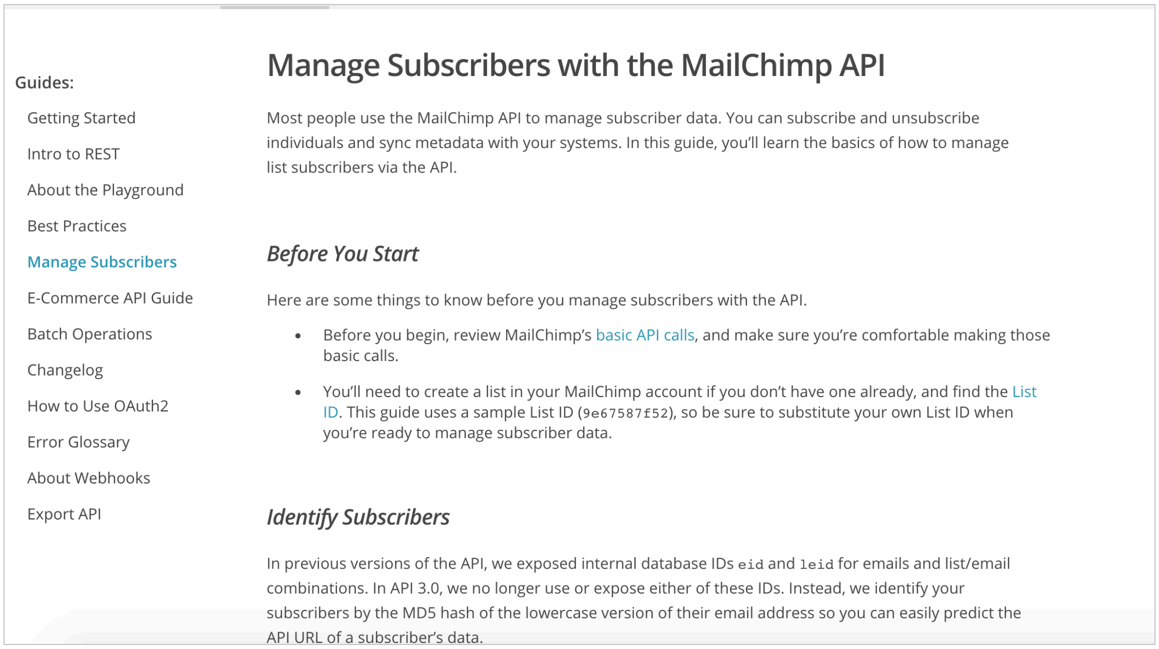
Javascript/Web iOS Android

Twilio code samples

Twilio’s tutorials are probably the most impressive and fully detailed tutorials in the examples here. Not only do they walk users through a task from beginning to end, they do so in half a dozen languages. The specific code examples have been extracted out into the right-column, while the narrative of the tutorial occupies in the middle column. All the steps in the tutorial aren’t shown at once. When you reach the end of one step, you click a button to show the next step. This progressive disclosure of information might reduce any sense of intimidation users might feel when beginning the tutorial.

Although the middle column is narrow and the right-column larger, actually this middle column just contains narrative text to annotate and explain the code. When you click a button in the tutorial, it brings the code on the right in focus and blurs the other code. Their implementation is a technical feat that I haven’t seen anywhere else.

Mailchimp



Manage Subscribers with the MailChimp API

Most people use the MailChimp API to manage subscriber data. You can subscribe and unsubscribe individuals and sync metadata with your systems. In this guide, you'll learn the basics of how to manage list subscribers via the API.

Before You Start

Here are some things to know before you manage subscribers with the API.

- Before you begin, review MailChimp's basic API calls, and make sure you're comfortable making those basic calls.
- You'll need to create a list in your MailChimp account if you don't have one already, and find the List ID. This guide uses a sample List ID (`9e67587f52`), so be sure to substitute your own List ID when you're ready to manage subscriber data.

Identify Subscribers

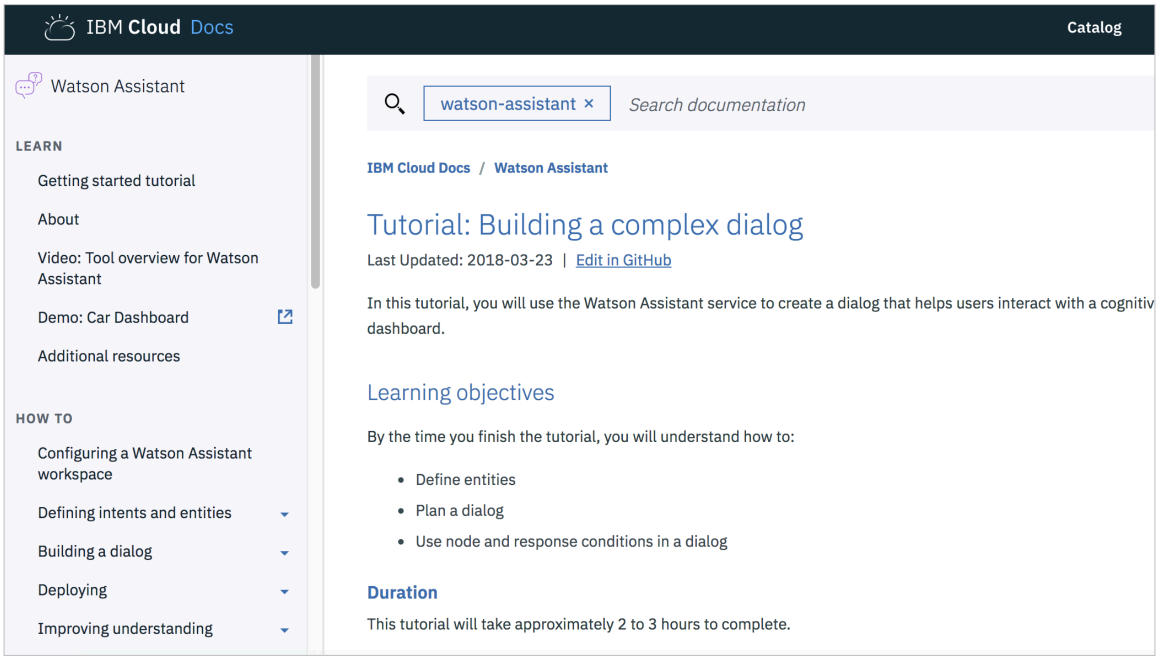
In previous versions of the API, we exposed internal database IDs `eid` and `leid` for emails and list/email combinations. In API 3.0, we no longer use or expose either of these IDs. Instead, we identify your subscribers by the MD5 hash of the lowercase version of their email address so you can easily predict the API URL of a subscriber's data.

Mailchimp code samples

As usual, Mailchimp provides solid tutorials for their products. The "Before You Start" section lists any necessary prerequisites before starting the tutorial. Each part of the tutorial is set off with section headings.

The section heading style (rather than numbered steps) is worth considering. Most technical writers have numbered steps as a habit for tech docs, so when they start writing a code tutorial, the first inclination is to begin a sequence of steps. But with a code tutorial, you might have lengthy code samples that are followed by detailed explanations, and so on. Maintaining the list numbers across steps can become onerous. The section headings provide less problematic formatting, and you can still preface each section heading with "Step 1", "Step 2", and so on.

IBM Watson



Tutorial: Building a complex dialog

Last Updated: 2018-03-23 | Edit in GitHub

In this tutorial, you will use the Watson Assistant service to create a dialog that helps users interact with a cognitiv dashboard.

Learning objectives

By the time you finish the tutorial, you will understand how to:

- Define entities
- Plan a dialog
- Use node and response conditions in a dialog

Duration

This tutorial will take approximately 2 to 3 hours to complete.

IBM Watson code samples

The IBM Watson tutorial does an excellent job breaking up the tutorial steps into different sections, with easy-to-follow steps in each section. Up front, it lists the learning objectives, duration, and prerequisites. There's nothing particularly challenging about the formatting or the display — the emphasis focuses on the content.

Code samples for sample weather API

Earlier in the course, we walked through each element of reference documentation (p. 115) for a fictitious new endpoint called `surfreport` in the weather API we were working with. Let's return briefly to that scenario and assume that we also want to add a code tutorial for showing the surfreport on a web page. What might that tutorial look like? Here's an example:

Code tutorial for surfreport endpoint

The following code samples show how to use the `surfreport` endpoint to get the surf height for a specific beach.

```
<!DOCTYPE html>
<head>
<script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
<script>
// limit the result through days parameter to keep the returned data set light
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://api.openweathermap.org/surfreport/25&days=1",
  "method": "GET"
}

// use ajax method to allow for asynchronous calls that won't block page loading
$.ajax(settings).done(function (response) {
  console.log(response);
  $("#surfheight").append(response.surfreport.conditions);
});
</script>
</head>
<body>
<h2>Surf Height</h2>
<div id="surfheight"></div>
</body>
</html>
```

In this example, the `ajax` method from jQuery is used because it allows us to load a remote resource asynchronously.

In the request, you submit the authorization through a query string URL. The endpoint limits the days returned to 1 to increase the download speed.

For demonstration purposes, the response is assigned to the `response` argument of the `done` method, and then written out to the `surfheight` tag on the page.

We're just getting the surf height, but there's a lot of other data you could choose to display.

One could go into a lot more detail with the explanation, even going line by line through the code, but here the commentary is already about half the length of the code. And there are some comments interspersed in the code. The comments address more the question of “why” rather than “what.”

Documenting code can be one of the most challenging aspects of developer documentation. Part of the challenge is that code isn’t organized such that a line-by-line (or block-by-block) description makes sense. Variables are often defined first, functions are called that are defined elsewhere, and other aspects are non-linear as well. As you explain the logic, you might find that you’re jumping around to different places in the code, not necessarily moving from top to bottom.

For a deeper dive into how to document code samples, see my presentation on Creating code samples for API/SDK documentation.

Activity with code samples

With the open-source project you identified (p. 686), identify code samples in the API documentation. Answer the following questions:

1. Are there code samples provided? In which languages?
2. How many code samples are there? Lots? Just a few? None?
3. Are there comments within the blocks of code?
4. How do the conceptual explanations point to specific lines of code? Is the explanation given before, during, or after the blocks of code?
5. Do the code explanations focus more on the “why” (the decisions behind the code) or the “what” (the mechanics of the code)?

Sample apps

Sample apps aren't just a convenience for developers, nor even an illuminating demonstration of abstract concepts in code. Sample apps are critical for technical writers to write good documentation. Without the ability to test content (p. 343) through a sample app, it's nearly impossible to know if your code instructions in your docs work.

Whether a sample app is relevant and available depends on the nature of the product, but in general you need sample apps or sample code to make sure the steps in the documentation work correctly. Additionally, sample apps tend to be powerful learning tools for developers because they allow them to experiment and see code in context from end to end.

Sample apps are critical to writing good docs

Let me start the discussion by sharing an experience related to sample apps. At one company, I was working on documentation for a project called Video Skills for Fire TV Apps ↗. This product involves implementing Alexa video skills into Fire TV apps so that customers can say phrases like "Play Bosch" and the app will automatically play the content, without the need to use a remote control.

The implementation process was pretty involved and required a series of steps listed here ↗. I inherited an initial set of docs (written by a PM + engineers) and sample app. I spent a month just getting the steps to work with the sample app. Then I spent another month fixing and updating the documentation. Using a sample app was critical for determining which steps in the initial docs were missing or incomplete. After I was confident that the steps worked well (because by following said steps, the sample app actually worked), I published the sample app as a companion to the docs ↗.

Without testing the docs through this sample app, I wouldn't have had any idea if they actually worked, and I'd basically be an engineer's secretary ↗. What I love most about sample apps is that they provide an end-to-end sample of working code. If you're working in any particular language, a simple code snippet might not run on its own (unless it's a standalone web call to an endpoint). But a chunk of Java or PHP or C++ code will usually require a sample app to run, or an IDE to compile and run the code.

Having little disconnected pieces of code that aren't associated within a sample app often isn't a testable scenario. And if you have a bunch of code that you can't easily test, how do you even know that it works? If you don't know whether the code you're documenting works, how can you feel good about the technical documentation you're writing? This is why sample apps are critical — they let you know that your instructions probably work.

Getting approval to publish sample apps

I've discussed the importance of sample apps in writing documentation and testing that documentation. Should you also seek to publish the sample app as a companion to the documentation? For example, with each disconnected code snippet, you could point the user to the code in the sample app for context and more details.

Unfortunately, publishing a sample app alongside the docs is not an easy task. Most sample apps are distributed via GitHub, so you might need to get a company-approved GitHub repo for your sample app. To do that, there's usually an approval process that involves Legal and other stakeholders. You'll have to figure out the right license for the code, make sure you have approval to even use the libraries and other third-party components in the code, and more. You'll also need to tag the code with the license in the proper way, remove any confidential files or information, and follow other security protocols for releasing the code. You're kind of in engineering territory at this point.

Code simplicity versus more robust functionality

Another hurdle in releasing a sample app is code quality. Assuming you get approval to put the sample app in a GitHub repo, is the code even worth making available to engineers? Will the code provide a *clarifying* example of how to implement a technique in an app, or will the code just muddle the waters even more?

Ideally, you want to make the sample app as simple as possible, elucidating the technique you are trying to demonstrate in the docs and nothing more. But isolating the code at that level of simplicity is tough because most apps usually require some other code to load and run.

With one project I worked on, developers didn't want to provide a sample app because they felt that any app that implemented Technique A but not Techniques B, C, D, E, and F would mislead developers. Developers implementing the feature might use the sample app as a starting point (copying the code wholesale, or using the sample app as a starting point for their app) and not realize that Techniques B, C, D, E, and F were also required.

But coding Techniques B, C, D, and E were beyond the scope of this sample app and accompanying documentation, and they made everything much more complex. Was it better to not provide any sample app at all rather than provide one that was either overly complex or which was rudimentary/incomplete?

With the video skills project I referenced earlier, the sample app was originally a fork of another Android streaming media template called Fire App Builder that had a lot of interchangeable modules. All of these modules added a lot of confusion and bulk to what should otherwise be much simpler code. But the app looked and functioned like a real TV app, with a landing page and video thumbnails. Project teams often want to demonstrate code that yields an attractive output, but they also want to keep everything as simple as possible. Later on, the team replaced the sample app with a much simpler one that didn't have as many polished features as the original Fire App Builder app.

Overall, figuring out the right balance in a sample app is a tradeoff. A simpler Fire TV app might not even include a landing page or thumbnail view of videos, but it might be more illustrative from a code perspective. Users wouldn't be distracted by all the other code that isn't essential to implement for the feature you're describing.

Functional but poorly written code

Sometimes the only sample app you can get your hands on is too embarrassing for engineers to feel comfortable sharing, so you might have to pitch hard for its value in the docs. Getting engineers to build beautiful, illustrative sample apps to accompany the docs can be a tall order, especially if these same engineers are struggling to complete their bi-weekly sprint tasks. In a best case scenario, you might be able to leverage a sample app from the QA team. Almost invariably, the QA team is vetting a list of features through some kind of test code, which is likely packaged into a simple sample app.

Whether that sample app can be extrapolated into a companion piece for the documentation is another matter. Usually, test apps are rudimentary and not necessarily best practice showcases for code. For example, maybe the code works/compiles, but there are many other aspects that engineers look for in code. In The Best Code is No Code At All, Jeff Atwood (quoting another author) explains what amounts to good code:

> - Brevity of code
> - Featurefulness
> - Speed of execution
> - Time spent coding
> - Robustness
> - Flexibility

Atwood then explains:

> It's painful for most software developers to acknowledge this, because they love code so much, but the best code is no code at all. Every new line of code you willingly bring into the world is code that has to be debugged, code that has to be read and understood, code that has to be supported. Every time you write new code, you should do so reluctantly, under duress, because you completely exhausted all your other options. Code is only our enemy because there are so many of us programmers writing so damn much of it. If you can't get away with no code, the next best thing is to start with brevity.

In other words, all code that works is not the same. Code should be as brief and efficient as possible. Just because the code in a sample app used by QA executes the feature, it doesn't mean the technique used in the code runs efficiently (executes with speed), uses a low amount of memory (reduces space), is elegant (implements re-use), and so on. It might be like an old car that putters along the road — it will get you to your destination, eventually. But you wouldn't want other car builders to pattern their cars after it.

As such, internal engineers are usually a lot more cautious about sharing shoddy code. There is a danger of potentially doing more harm than good, especially if the code in the demo implements poor practices that developers then copy and duplicate in their own apps. Getting devs to create exemplary code is usually an effort beyond their initial mandate in building the feature.

Is it better to require third-party developers to work out the implementation code on their own (probably misinterpreting and also implementing it incorrectly) rather than potentially leading them astray with bad sample code? I think most developers would say yes. If you push for sample apps as companion pieces to documentation, you might have to find a compromise here between simple, half-baked but functional code and more robust, "showcase" code that would scale in a production environment. Even if you can't release the sample app, you can probably use the sample app for your own testing and validation of the documentation.

Who owns and maintains the sample app?

If you do manage to publish the sample app, questions about ownership will need to be clarified. Who owns the sample app, especially as a companion piece for the documentation? If the apps are integrated into the documentation and intended to highlight the docs, then wouldn't it make sense for the doc team to have some stake in the code's ownership? At the same time, who will respond to the issues logged on GitHub, the pull requests, and other comments around the code there? Ownership and responsibility matter.

I’ve seen repos abandoned by engineering teams who simply move on to other projects. Engineers transition to other teams/projects/initiatives and don’t want to be supporting the same GitHub repos for the rest of their lives. Other times, projects get de-prioritized or sunset, but the GitHub repo stays in the public space, and the initial docs remain, still supported by the doc team. In these cases, you might end up with a sample app that is seemingly inactive or unsupported (with issues piling up), and docs that you still continue support in a “keep the lights on” kind of way. You might have to keep tabs on the GitHub issues logs and champion internal sources to provide support.

SDKs (software development kits)

SDKs (software development kits) and sample apps are similar to code samples and tutorials (p. 464) but are much more extensive and usually involve a whole collection of files that work together as a package or sample app. The SDK might include libraries that you download and incorporate into your application, and can consist of tools, sample apps, and other code.

What is an SDK?

The terms API and SDK are often used together, but they aren't synonyms. SDKs implement the language-agnostic REST API in a specific language, such as Java or C++. REST APIs by themselves aren't tied to any particular language; usually, you demonstrate the APIs by making calls using cURL (p. 80), a command-line tool for submitting web requests and getting responses. But developers won't use cURL requests when they implement your API. Instead, they will implement the API requests using the language their application is coded in.

For example, Python, C++, or Node applications make API requests in different ways. Each language has its own way of constructing requests to a web API. You can use Postman or Paw to auto-generate a simple request in a specific language (see Auto-generating code samples (p. 146)). However, the SDK takes the implementation to another level. SDKs might involve many more files or libraries as part of the implementation.

In What is the Difference Between an API and an SDK? ↗, Kristopher Sandoval explains an SDK as follows:

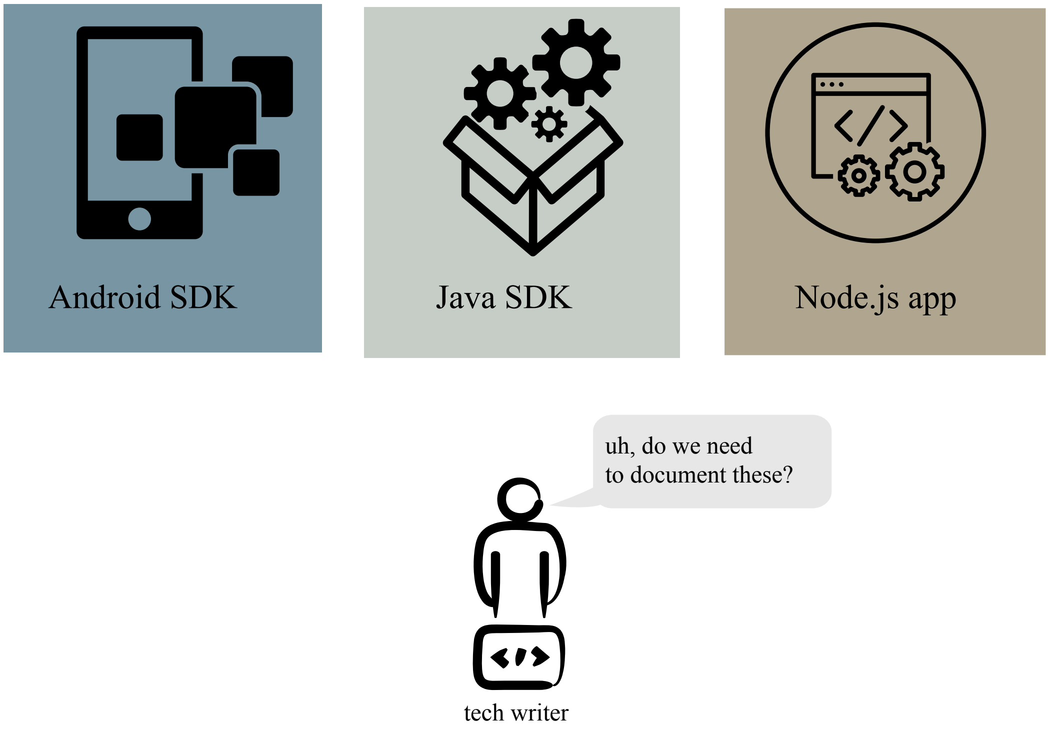
> SDK stands for "Software Development Kit", which is a great way to think about it — a kit. Think about putting together a model car or plane. When constructing this model, a whole kit of items is needed, including the kit pieces themselves, the tools needed to put them together, assembly instructions, and so forth.
>
> An SDK or devkit functions in much the same way, providing a set of tools, libraries, relevant documentation, code samples, processes, and or guides that allow developers to create software applications on a specific platform. If an API is a set of building blocks that allows for the creation of something, an SDK is a full-fledged workshop, facilitating creation far outside the scopes of what an API would allow.

Sandoval compares examples from both Facebook APIs and SDKs to clarify the difference. He sums up the difference as follows: "The SDK is the building blocks of the application, whereas the API is the language of its requests." In other words, the SDK provides all the necessary code you would need to build an application that uses the API.

What is your role in documenting the SDK and sample app

In the SwaggerHub tutorial, I showed how to auto-generate client SDKs (p. 234) through SwaggerHub's interface. But usually rather than relying on auto-generated SDKs, if your development team offers a client SDK, it will be code that the development team prepares and tests. The development team often provides the SDK in a few target languages based on their user's main language, making it easier for users to implement the API.

As an API technical writer, documenting SDKs and sample apps presents a tough challenge because SDKs require you to be familiar with one or more programming languages. I explore the question of how much code you need to know (p. 664) in the Jobs section (p. 659), so I won't get into too much detail here. Usually, engineers don't expect you to know multiple programming languages in depth, but some familiarity with them will be required in order to both write and review the documentation. When deciding whether to call a block of code a function, class, method, or another name, you need to have a basic understanding of the terms used in that language.



SDKs and sample apps

If you're unfamiliar with the language, you can just take what engineers write, clean it up a bit, try to walk through the steps to get any sample apps working and see what feedback you get from users. Usually, if you can get a sample app installed and working, and make sure that the basic documentation for running the app works, as well as what the app does, that might be sufficient. But of course, making any significant contributions to SDK documentation will require you to be familiar with that programming language.

As I mentioned in the Code samples and tutorials (p. 464), you don't need to document how a particular language works, just how your own company's SDK works. Presumably, if an engineer downloads the Java SDK for an API, it's because the engineer is already familiar with Java. However, if your API was implemented in a particular way in Java, you should explain *why* that approach was taken. (Granted, understanding the difference between documenting Java and documenting a particular approach in the Java implementation also requires you to understand Java (p. 701).)

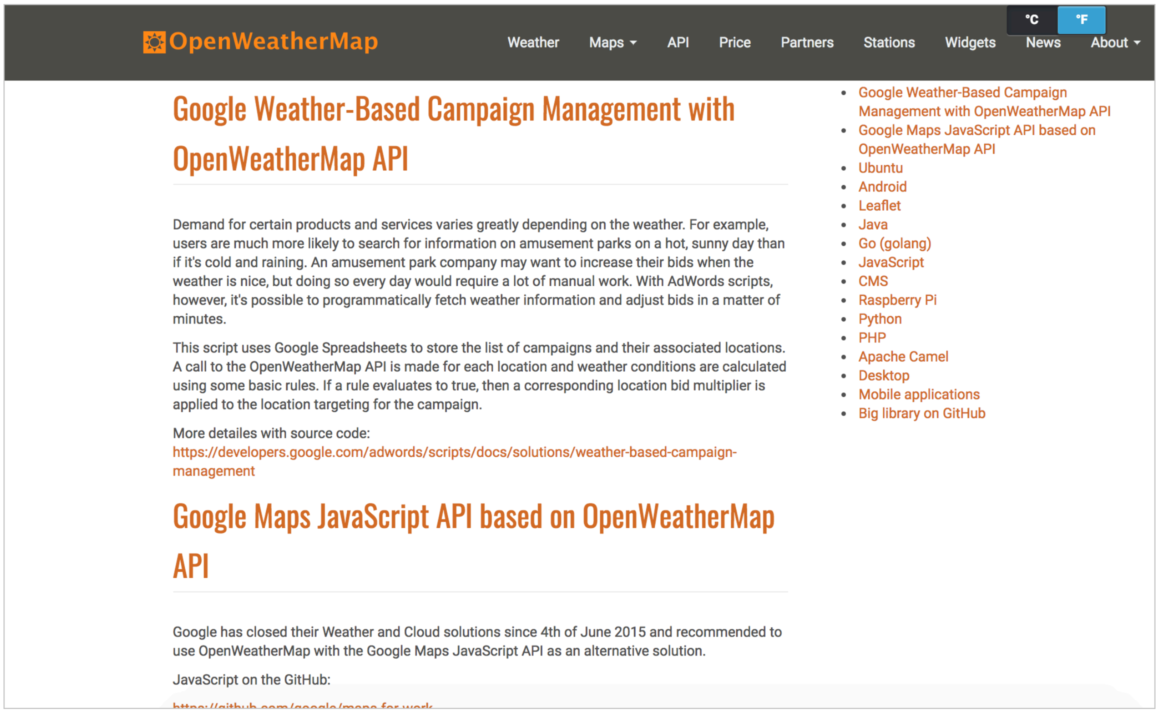
Releasing your SDK

When you release the SDK, although engineers might handle the release, they will probably look to you for input on SDK readiness, including preparation of the Readme, documentation, licensing, and other details. See Processes for managing SDK release processes (p. 755) for more information about these details.

Sample SDKs and sample apps

The following examples show documentation for some sample SDKs and sample apps.

OpenWeatherMap API



Google Weather-Based Campaign Management with OpenWeatherMap API

Demand for certain products and services varies greatly depending on the weather. For example, users are much more likely to search for information on amusement parks on a hot, sunny day than if it's cold and raining. An amusement park company may want to increase their bids when the weather is nice, but doing so every day would require a lot of manual work. With AdWords scripts, however, it's possible to programmatically fetch weather information and adjust bids in a matter of minutes.

This script uses Google Spreadsheets to store the list of campaigns and their associated locations. A call to the OpenWeatherMap API is made for each location and weather conditions are calculated using some basic rules. If a rule evaluates to true, then a corresponding location bid multiplier is applied to the location targeting for the campaign.

More detailes with source code: https://developers.google.com/adwords/scripts/docs/solutions/weather-based-campaign-management

Google Maps JavaScript API based on OpenWeatherMap API

Google has closed their Weather and Cloud solutions since 4th of June 2015 and recommended to use OpenWeatherMap with the Google Maps JavaScript API as an alternative solution.

JavaScript on the GitHub:

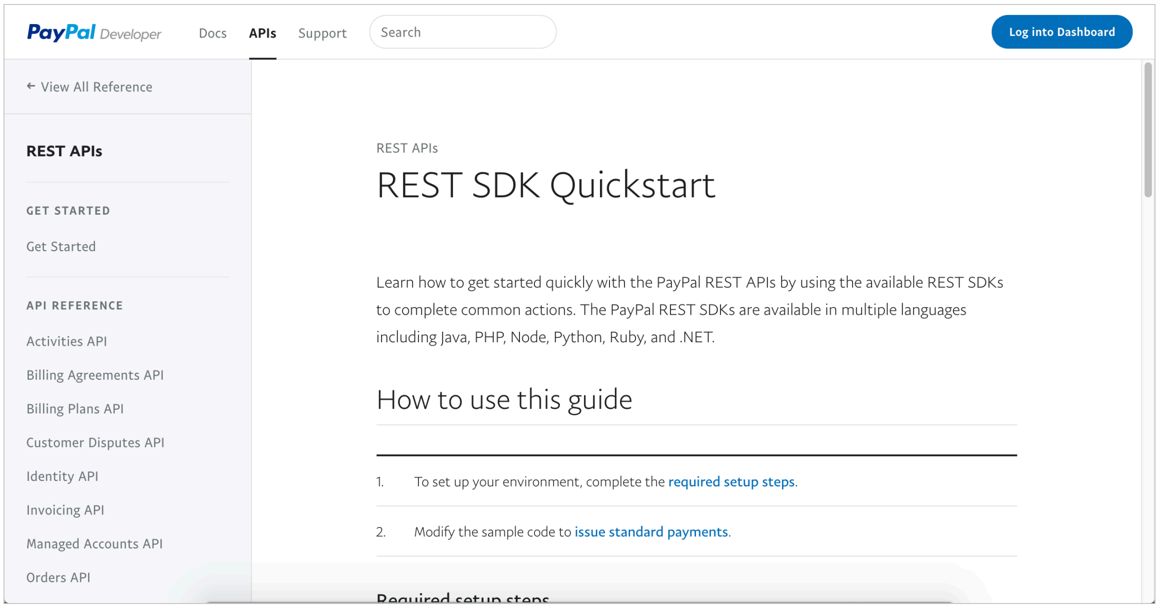
- Google Weather-Based Campaign Management with OpenWeatherMap API
- Google Maps JavaScript API based on OpenWeatherMap API
- Ubuntu
- Android
- Leaflet
- Java
- Go (golang)
- JavaScript
- CMS
- Raspberry Pi
- Python
- PHP
- Apache Camel
- Desktop
- Mobile applications
- Big library on GitHub

Code samples for OpenWeatherMap API

The example integrations for the OpenWeatherMap API aren't just short code snippets that show how to call an endpoint. Instead, they are full-fledged, sophisticated integrations across a variety of platforms. As such, many of the code samples are stored in GitHub. Each scenario has a detailed explanation.

If you can put your sample apps and SDKs on GitHub, it's usually a good idea to do so. Storing code on GitHub accomplishes two purposes: First, it usually puts the burden on engineering to maintain and test the code samples as well as respond to issues users might log against the project. Second, it makes it easier to provide fully functional code, since users can clone the project and start working with it immediately. The development team can also push out updates easily.

Paypal REST SDK



REST APIs

REST SDK Quickstart

Learn how to get started quickly with the PayPal REST APIs by using the available REST SDKs to complete common actions. The PayPal REST SDKs are available in multiple languages including Java, PHP, Node, Python, Ruby, and .NET.

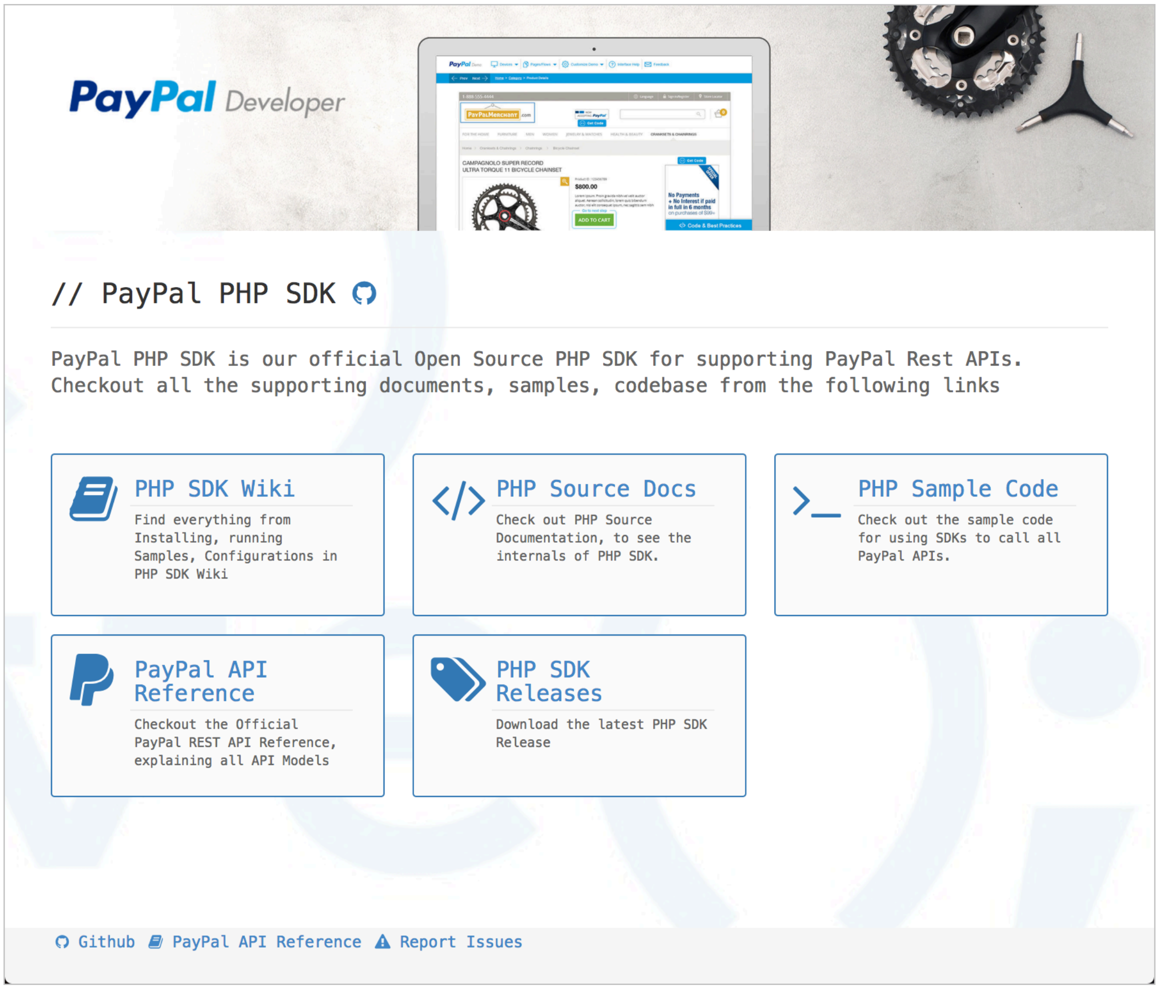
How to use this guide

1. To set up your environment, complete the required setup steps.
2. Modify the sample code to issue standard payments.

Required setup steps

Paypal REST SDKs

The SDKs in the Paypal’s Additional information section include Node JS, PHP, Python, Ruby, Java, and .NET SDKs. Each implementation has its own GitHub site, with its own wiki, sample code, source docs, and more. If you browse some of these GitHub pages (such as the site for PHP), you can see the whole collection of language-specific files for this SDK. These sites show how SDKs include a variety of file types.



// PayPal PHP SDK

PayPal PHP SDK is our official Open Source PHP SDK for supporting PayPal Rest APIs. Checkout all the supporting documents, samples, codebase from the following links

PHP SDK Wiki
Find everything from Installing, running Samples, Configurations in PHP SDK Wiki

PHP Source Docs
Check out PHP Source Documentation, to see the internals of PHP SDK.

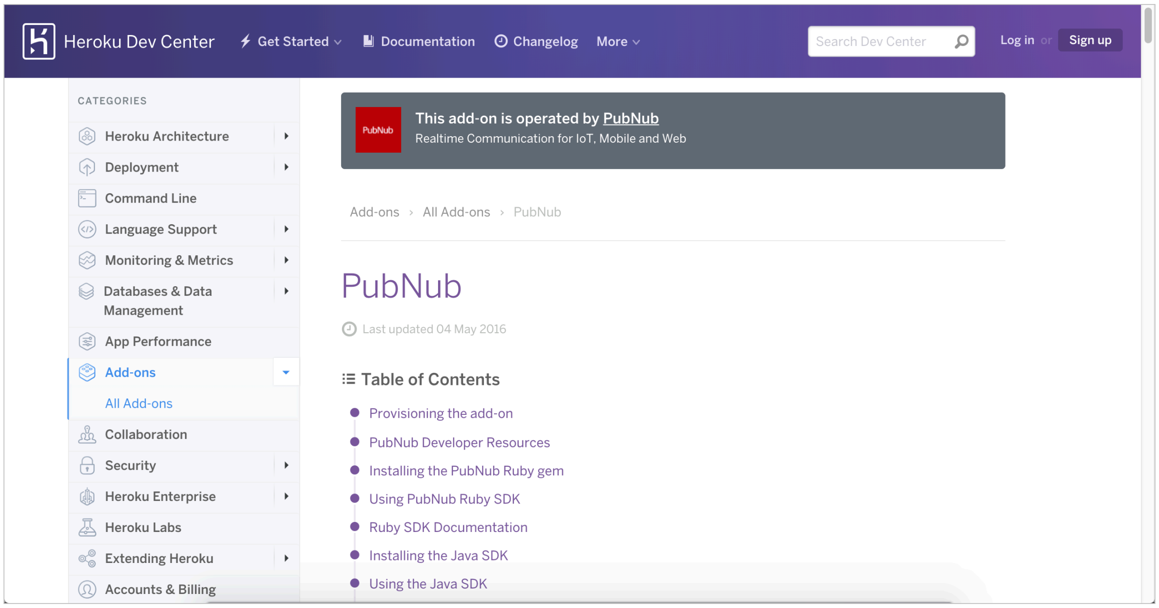
PHP Sample Code
Check out the sample code for using SDKs to call all PayPal APIs.

PayPal API Reference
Checkout the Official PayPal REST API Reference, explaining all API Models

PHP SDK Releases
Download the latest PHP SDK Release

Paypal PHP SDK

Heroku SDK



This add-on is operated by PubNub
Realtime Communication for IoT, Mobile and Web

Add-ons › All Add-ons › PubNub

PubNub

Last updated 04 May 2016

Table of Contents

- Provisioning the add-on
- PubNub Developer Resources
- Installing the PubNub Ruby gem
- Using PubNub Ruby SDK
- Ruby SDK Documentation
- Installing the Java SDK
- Using the Java SDK

Heroku SDK

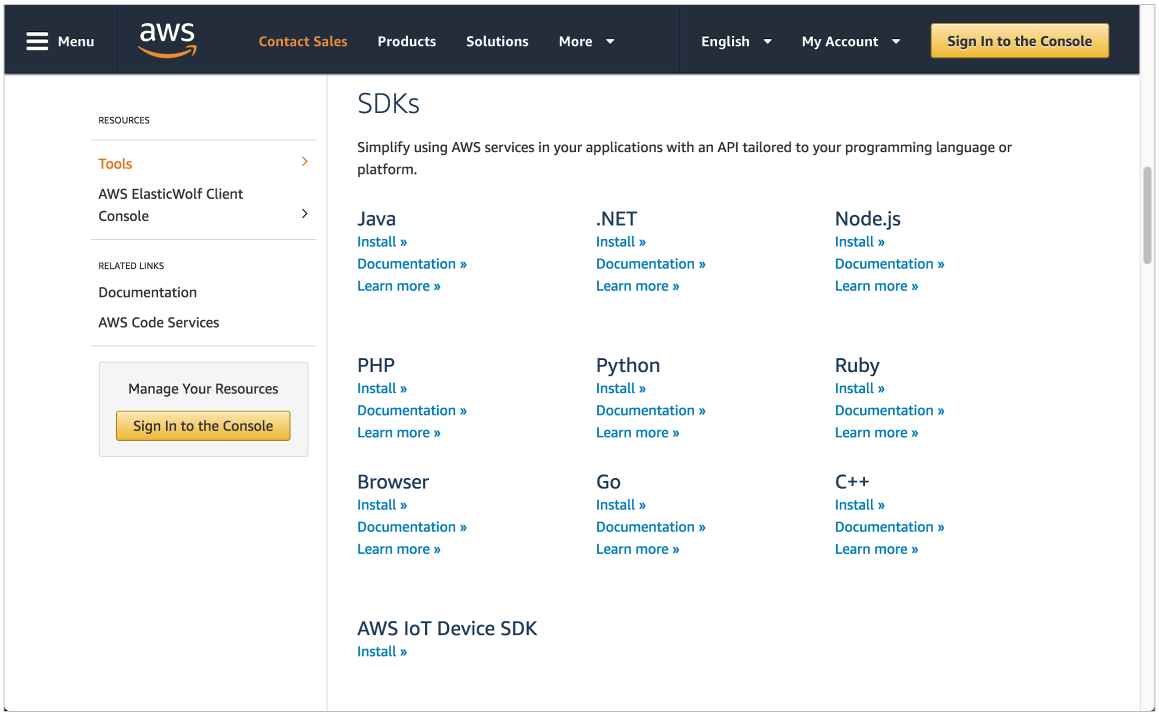
The Heroku SDK is actually operated by PubNub and includes a Ruby, Java, Node JS, Python, and PHP SDK. If you look at, for example, the Python SDK documentation, you see links to Getting Started, Tutorials, and API reference.

As I mentioned earlier, it's unlikely that you'll be able to contribute significantly to either writing or reviewing the SDK documentation unless you're somewhat familiar with the language. Development groups usually don't expect technical writers to be conversant in half a dozen languages. More likely, you'll be reliant on engineers who are conversant in these languages and frameworks to author this content. But doing so will require you to interact skillfully with engineers and be somewhat familiar with programming jargon and concepts.

If engineers tell you that users should know X, don't just submit to their judgment out of ignorance with the language. Instead, find some developers in that language (even internal engineers in other groups) to test the documentation against. If those users push back and say they need more detail, you can interface with the engineering team to provide it.

Without more familiarity with the language of the SDK, technical writers act more as mediators between the engineering authors and the engineering users. Technical writers identify and fill gaps in the documentation, and they often manage the publishing and distribution of the docs. But the content itself might be too technical for most technical writers to play a content authoring role. (I explored this topic in depth in an essay in my Simplifying Complexity series called Be both a generalist and specialist through your technical acuity.)

Amazon SDK



SDKs

Simplify using AWS services in your applications with an API tailored to your programming language or platform.

Java	.NET	Node.js
Install »	Install »	Install »
Documentation »	Documentation »	Documentation »
Learn more »	Learn more »	Learn more »
PHP	**Python**	**Ruby**
Install »	Install »	Install »
Documentation »	Documentation »	Documentation »
Learn more »	Learn more »	Learn more »
Browser	**Go**	**C++**
Install »	Install »	Install »
Documentation »	Documentation »	Documentation »
Learn more »	Learn more »	Learn more »

AWS IoT Device SDK
Install »

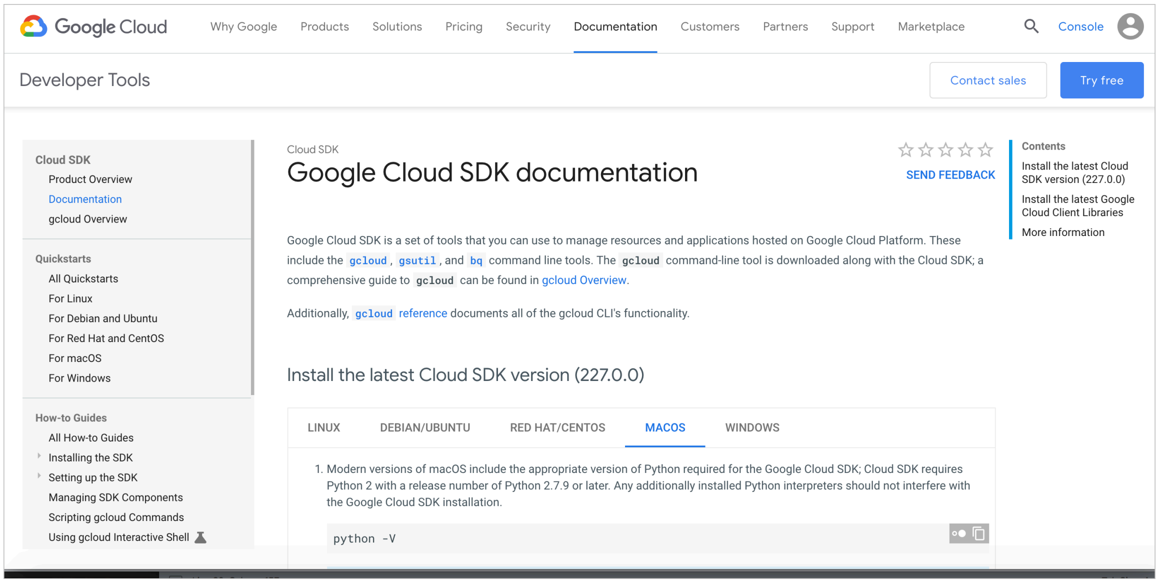
Amazon AWS SDKs

One notable characteristic of the AWS docs is their consistency from doc set to doc set. The consistency leads to predictability and hence usability. However, in the SDK docs, you can see that different document generators are used to generate the docs for the various libraries. If you look at the API references for each of these SDK libraries, you'll see a C++ document generator for C++ SDK docs, a Ruby document generator for Ruby SDK docs, a PHP document generator for PHP SDK, a .NET document generator for .NET SDK docs, a Java document generator for Java SDK docs and so on.

Each programming language typically has its own unique annotation syntax and document generation tools. The annotation syntax (which programmers use directly in the code — see Javadoc tags (p. 712) for an example of Javadoc tags) differs by language and tool but is mostly similar. Because the documentation is generated from annotations in the code, engineers usually write and maintain this documentation. (Having engineers write and maintain it also reduces documentation drift.)

Even so, there is probably quite a bit of variability from one library to the next. How do engineers ensure they use the same description for a class in Java that they do for Ruby and PHP? These document generator tools aren't usually smart enough to leverage snippets or includes stored in a shared online repository. You also can't usually use variables or other single-sourcing techniques. As a result, there might be a lot of variation from one SDK's documentation to another for mostly the same concepts.

Google Cloud SDK



Cloud SDK

Google Cloud SDK documentation

Google Cloud SDK is a set of tools that you can use to manage resources and applications hosted on Google Cloud Platform. These include the `gcloud`, `gsutil`, and `bq` command line tools. The `gcloud` command-line tool is downloaded along with the Cloud SDK; a comprehensive guide to `gcloud` can be found in gcloud Overview.

Additionally, `gcloud` reference documents all of the gcloud CLI's functionality.

Install the latest Cloud SDK version (227.0.0)

1. Modern versions of macOS include the appropriate version of Python required for the Google Cloud SDK; Cloud SDK requires Python 2 with a release number of Python 2.7.9 or later. Any additionally installed Python interpreters should not interfere with the Google Cloud SDK installation.

```
python -V
```

Google Cloud SDK documentation

The Google Cloud SDK provides quickstart guides for Linux, Debian, Ubuntu, and other operating systems. The guides explain how to install, set up, and manage the SDK commands. An API reference for the commands is also included.

Looking at the Google Cloud SDK versus the Amazon SDK shows some of the breadth and variety of technologies you might have to document in SDK territory. These SDKs are specific to a particular programming language, operating system, or another framework, and as such, it can be daunting to try to ramp up to document this category of tools. For SDK documentation, you'll need to work closely with engineers and listen to feedback from users.

Activity with SDKs

With the open-source project you identified (p. 686), identify the information about any SDKs for the API. Answer the following questions:

1. Does the API project include any SDKs?
2. In what languages are the SDKs provided?
3. Why did the developers choose to make the SDK available in that particular language?
4. How extensive are the instructions for working with that SDK?
5. Where is the code for the SDK stored and delivered? In GitHub? In a separate downloadable zip file?

API design and usability

Technical writing intersects closely with usability like no other domain on technical projects. This close connection with usability especially surfaces as you're writing tutorials on how to use the API in a step-by-step way. Because of this, in this section I've decided to add some more tips and insights about how to assess the usability and design of an API.

Documenting APIs tests the API design

In *The Design of Web APIs*, Arnaud Lauret, aka the API Handyman, says,

> Exhaustively documenting APIs has a nice side effect—it is testing the design. If you are unable to document how to use the API or how to implement it, it can be a sign of improper design.

In other words, as you write out the specific steps users need to follow to perform their goals, you are making your way through the user's footsteps and workflows. If you find these paths to be fuzzy and full of treacherous pits and gotchas, inconsistent parameters or unintelligible responses, chances are the API's usability is poor.

Bob Watson has a post that echoes similar sentiments. In the post "If your API is hard to document, be warned", Bob explains that one of the first signs that your API is in trouble is when the technical writers find it difficult to write the documentation. Bob compares technical writers to the canaries that coal miners once brought with them to detect poisonous gases.

Bob says, "If your API is hard to document, it could be a sign of more fundamental problems." If technical writers find it "difficult to articulate market needs and pain points" (because the market needs don't exist), if technical writers find themselves "adding the verbal duct tape to make the product ready to ship," your API is in trouble. The first signs of an API's failure are likely signposted right there in the documentation.

Since technical writers are uniquely suited to provide feedback on usability, especially as we step through the user tasks in detail, it makes sense to expand here on what usability means, how to evaluate the usability of an API, and other details to look for. In presenting this information, I'll rely extensively on Arnaud Lauret's *The Design of Web APIs* since it provides an in-depth treatment of APIs and design/usability.

What is usability

Usability has the same meaning whether you're referring to end-user applications with graphical user interfaces or developer code with an API. Usability refers to how easy it is for users to accomplish their desired tasks with the tool.

To evaluate usability, you must first understand what tasks users want to perform with the API. Since the API user's guide that technical writers create covers these tasks in detail, technical writers are perfectly situated to provide feedback on an API's usability.

Challenges in evaluating developer experience usability

Despite the seemingly perfect fit between tech comm and usability, it's a lot more challenging than it might seem for tech writers to provide usability feedback.

The first issue is that tech writers are often too far downstream to have any actionable feedback on the design, so the motivation to influence design might not be strong. Questions about the API's design have likely been decided on long before the documentation task usually arrives in the tech writer's inbox. Unless you're extremely proactive in identifying APIs at their inception point, you probably don't even hear about the API until it's been approved by leadership and half-way coded. At this point, if you raise your hand and say, this endpoint doesn't make any sense to me, the feedback is too radical. Developers aren't going to drop everything and go back to the drawing board.

But supposing that the developers are listening intently for your feedback — the design is often above the heads of technical writers. On one project I worked on, the team was finalizing their security design and solicited feedback on the authentication workflows. I dug into these security workflows with good intentions, but honestly I didn't have the engineering knowledge to evaluate whether the security mechanism and workflow they had chosen was the right one for the audience, or whether the process for authorizing requests was appropriate and secure, especially in context of the larger company policies around privacy and authorization.

Much of developer docs have the same knowledge weight. As technical writers, we are often straining to keep our heads above water, learning as we go. We aren't often in a position to critique an approach taken. And even when we create an end-to-end tutorial for using the API toward some business goal, it might be hard to decide whether the workflow is easy or hard for a developer to implement — this is where lack of an engineering background can be challenging. Much of the code might seem difficult to implement. And assessing whether it's the norm for the space is hard to determine without working with other APIs across the industry.

Despite these challenges, with a little understanding of what to look for, you can learn how to critique an API's usability just as with any other tool. All you need is to understand what to look for. So let's jump into Arnaud's book for a more focused discussion of these points.

Evaluating API usability

Arnaud says that "an inconsistent design introduces variations or contradictions that make an interface harder to understand and use." In other words, an API should be consistent in ways that developers can predict without always consulting the documentation. The following sections cover a few questions to ask related to consistency.

Straightforward endpoints

Are endpoints intuitive in what they return? Or do you look at an endpoint and not have any idea what it returns? Keep in mind the list of tasks that users want to accomplish with the API, along with the terms they use to describe those tasks. Could you draw lines between the users' tasks and the endpoints they might need to use to accomplish such tasks? If so, then your endpoints are probably straightforward. If the two don't have any resemblance to each other, users will likely be stumped as well.

Consistent endpoint patterns

Do the endpoints follow consistent patterns? A common pattern is to use plurals in the paths. Arnaud explains:

> Although there are no official REST rules regarding resource path design (apart from uniqueness), the most widely adopted format is `/{plural name reflecting collection’s item type}/{item id}`. Using resource paths exposing resource hierarchy and using plural names for collections to show the collection’s item type has become a de facto REST standard. In our example, a catalog should therefore be identified by `/products` and a product by `/products/{productId}`.
>
> … Once consumers are familiar with the use of plural names for collections, they expect to see all collections with plural names. You can use a singular for collections if you want, but whatever your choice, stick to it! And this doesn’t only apply to URLs: it concerns every single name and value you choose…

Look at the collection of endpoints in your API. Do they follow consistent formats as Arnaud explains? Or are the endpoints a hodgepodge of different formats and tenses (e.g., `/products/{productId}` and `/transfer/{transferId}`)? Does it seem like different teams worked independently on different parts of the API, without communicating with each other? Technical writers can usually spot parallelism quite easily in lists. So list out your endpoints and see if they follow a consistent pattern, preferably with a plural for the collection and a singular for the specific item.

When you get into the nuts and bolts of an API (as you’re likely to do by documenting the API), you’ll start to spot inconsistencies. One API I worked on had a different implementation for two different device types. Unfortunately, the APIs used for one device type didn’t work similarly for the other device type. Names followed different conventions, and it was obvious that the designs originated from different teams. One reason for slow adoption from users was due to the inconsistency between APIs. Pointing out this inconsistency didn’t create any “Aha” moments for the project team, as they were already aware of the embarrassing inconsistency for many months, but it was good to have some validation for my feedback anyway.

Nouns in endpoints, verbs in operations

Another common pattern with endpoints is to use nouns with the resources, and to let the operations/methods supply the verb. Arnaud covers this in his book, but other sites such as Stoplight also recommend the practice. Stoplight.io says:

> **Use nouns for resources** and avoid anything that looks like procedure calls. There’s no need for endpoints like `/getContacts` when we’re using HTTP methods. Instead, your resource would be named `/contacts` and you could perform the `GET` action (and any others that are relevant) against that resource. (Basic API Design Guide & Best Practices ↗)

At the most basic level, REST API endpoints and their supported operations have a noun + verb grammar (e.g., `contacts` with a `READ` operation). If the endpoint name blends the resource with the operation, as in `getContacts`, it can both be redundant as well as misleading. Separate nouns from verbs when it comes to endpoint names.

Note that this same convention might not be the case with native library APIs (p. 692), such as a Java API. With native library APIs, the same resource might not have different operations that you can apply to it; instead, the APIs are often functions that perform specific actions, so a `getContacts` function might make sense.

Data formats in responses

Another point to consider is the data formats returned in responses. Just how usable is the data format for the audience? Are you returning JSON or XML? Do you give users the ability to specify the type of units they want? Arnaud writes,

> Data is at the core of APIs—resources, parameters, responses, and their properties shape an API. And all of their meanings, names, types, formats, and organization must be consistent in order to help consumers understand these easily.

For example, will developers prefer to work in metric or imperial systems? Celsius or Fahrenheit? I've seen some APIs fail to even indicate what units are used. What time format are you returning, and is this format consistent across all your endpoints? For example, if you return Unix Epoch time for one endpoint but ISO 8601 for another endpoint, that's inconsistent and potentially confusing.

Arnaud explains,

> People seek global uniformity in design. Once consumers have seen one date and time property represented by an ISO 8601 string, they expect all date and time properties to be ISO 8601 strings. Once a data format has been chosen for a type of data, it should be used for all representations of the same data type.

As you define the parameters (p. 132) in your requests, as well as the responses in your models or schemas (p. 154), you will be describing the data types for each element. As such, you'll be pretty familiar with common patterns for data types. Sometimes developers will mistakenly identify numbers as strings, or specify much more specific numbers (e.g., floats, doubles) than is usually done in REST API docs, or they might have other inconsistencies. Look closely at these data types and see if they conform to consistent patterns across all endpoints. Again, inconsistency is easy to spot and hard to argue against when providing usability feedback.

Casing with parameter names

Another area of high inconsistency is the casing of parameter names. Do all parameter names follow similar casing, or are there variations among upper camel case (`ResourceId`), lower camel case (`resourceId`), underscores (`resource_id`), or other conventions?

Enums (a list of allowed values) are often written in full caps (`RESOURCEID`), so they might differ from the lower camel case convention, which is most commonly used. But generally, parameter names should have similar casing conventions. Identify which conventions are used in your API. Then locate any inconsistencies.

For example, in the Microsoft API Guidelines, the guide notes that "JSON property names SHOULD be camelCased." Additionally, "Services SHOULD provide JSON as the default encoding."

Note that it might not always be feasible to implement changes you recommend if your API is already live. If developers push out endpoints with inconsistent parameter names, and developers adopt and implement these APIs, changing the parameter names can cause breaking changes for the implementations.

For example, if someone has already implemented code that looks for the `ResourceId`, and a later update changes this to `resourceId`, code that parses JSON won't recognize this value because JSON is case sensitive. As such, if developers are reluctant to fix inconsistent code because making it consistent would introduce breaking changes, it might be a good argument for keeping it as is.

Industry conventions

Beyond consistency within your own API's world, how do your conventions align with the API industry in general? Once after I gave an API workshop, a participant asked me to look over her company's API doc site and provide some feedback. The doc site was a list of a seemingly infinite collapsed sections on a single page, without any other sidebar navigation. Though the site was perhaps novel, I found it hard to navigate and recommended that they align more with industry conventions — by adding a sidebar with navigation.

Awareness and alignment with industry conventions is why I so frequently include activities in this course that involve reviewing approaches on other API doc sites. Following common industry conventions leads to predictability when new users navigate your site. If you organize your endpoints under a section called "API Reference," most users will intuitively understand what that section contains because so many other sites likewise call it "API Reference." Arnaud also emphasizes the importance of aligning with industry conventions:

> … APIs have to be consistent with the rest of the world. There are common practices—standards, if you will—that you can use. Not only does following these make your APIs predictable for people who have never used any of your APIs before, thereby enhancing your APIs interoperability with the rest of the world, but it also makes your API designer's job easier.

Look for top APIs from the big tech companies for examples. Stripe, Twilio, Twitter, GitHub, Microsoft, Facebook, Google, and other sites provide good examples of how to approach and organize API docs. Compare your company's approach to that in several of these common API doc sites.

Consult Your API style guide

Just as there are style guides for tech docs and marketing material, companies often have API design style guides as well. Consider the scenario of a large company with many different engineering teams each building APIs. To ensure consistency across the APIs they build, these companies usually have an API style guide they adhere to. Arnaud says that best practices for API development usually involve "formally defin[ing] your design with rules in a document called the 'API Design Guidelines' or the 'API Design Style Guide.'"

In fact, Arnaud has collected more than 15 of these style guides in a site called API Stylebook ↗. In the site, he pulls from each of these style guides to show their guidance for common sections. You can also click links directly into the style guide sites. For example, check out the Cisco API Design Guide ↗ and Microsoft REST API Guidelines ↗.

Just as documentation style guides go in-depth about more minutia than most people care about, API design style guides do the same about every aspect of an API. Before providing feedback about the API you're documenting, find out if your company has an API style guide. If they do, you'll have direct support to back your claims about potential inconsistencies or lack of adherence to standards. If your company doesn't have a design style guide, you could compare the approach in other design style guides.

Style guides for APIs are just as helpful with API design as they are with documentation. With documentation, style guides make it much easier to write. You no longer have to decide what to make bold and what to italicize, what to call certain terms, whether to use hyphens in "drop-down" or how to refer to required/optional items, and so on. Becoming familiar with your API design style guide can have similar effects as you work with an API. When you encounter multiple-word parameters, you won't have to guess which case to use, what convention to follow for endpoint paths, what to call common properties in responses, and so on.

Workflows

Now let's look at another aspect of API usability: workflows. Presumably, this is where the technical writer's moment shines. Developers might write the reference documentation, but the use cases and other tutorials in the user guide are usually handled by technical writers.

As you describe the workflows required to use the API, you are uniquely suited to critique the usability. Start by assessing how many APIs you have to chain together to arrive at your result. For example, in my Flickr API gallery activity (p. 908), I walk through a use case of getting photos from a specific gallery. Presumably, one would use the `getPhotos` endpoint for this, but this task requires you to chain endpoints together. In the activity I explain:

> One of the arguments we need for the `getPhotos` endpoint is the `gallery_id`. Before we can get the `gallery_id`, however, we have to use another endpoint to retrieve it. Somewhat unintuitively, the `gallery_id` is not the ID that appears in the URL of the gallery. Instead, we use the `flickr.urls.lookupGallery` endpoint listed in the URLs resource section to get the `gallery_id` from a gallery URL

In your API, do users have to use one API to retrieve a specific value that they then plug into the parameters of another API? How many endpoints does a user need to use together before they can achieve their objective? Arnaud explains that "usability heavily depends on the simplicity of this flow of interactions."

The Illusion of Explanatory Depth

As developers are designing an API, presumably they think through each scenario and design the list of endpoints required. However, despite the forethought and planning that goes into the design of an API, despite all the upfront discussions and ideation before coding even begins, there is often a gap of detail that gets overlooked. When you get down to describing all the steps that users must really perform, the nitty-gritty detail, another reality often surfaces: implementation is much more complex than anyone realized.

There is a phenomenon to describe the gap between how users think something works and how it actually works. It's called The Illusion of Explanatory Depth ↗ and refers to the idea that people often don't realize the complexity of things they think they understand. Psychologist Adam Waytz explains,

> If you asked one hundred people on the street if they understand how a refrigerator works, most would respond, yes, they do. But ask them to then produce a detailed, step-by-step explanation of how exactly a refrigerator works and you would likely hear silence or stammering. This powerful but inaccurate feeling of knowing is what Leonid Rozenblit and Frank Keil in 2002 termed, the illusion of explanatory depth (IOED), stating, “Most people feel they understand the world with far greater detail, coherence, and depth than they really do.”

When you write an end-to-end tutorial about how to use an API for a user goal, you expose all of the detail in a meticulously granular fashion required to achieve the end. As Waytz says, you “produce a detailed, step-by-step explanation of exactly how” this goal is achieved using the system. In so doing, you surface many steps “with far greater detail, coherence, and depth” than people who have been thinking conceptually about design in perhaps a philosophical level have thought out. This is why Arnaud says that writing the documentation *tests* the design. The very act of writing the documentation reduces the illusion in the The Illusion of Explanatory Depth phenomenon.

This is why writing the tutorial is one of the best ways to assess the usability of the API. Unfortunately, by the time you’re actually writing this tutorial, the API’s design is often finalized (at least for the upcoming release, but maybe not for the next release).

Error Messages

As you’re writing tutorials, testing the API as you go, you will no doubt run into error messages (p. 402). These little messages pop up here and there in often subtle, hard-to-catch ways. Always note error messages that you see, as error messages are a key factor in usability.

Ideally, error messages should be descriptive enough to let users know what’s wrong and to help them correct the issue. Arnaud explains,

> Error feedback must explain what the problem is and should help the consumers to solve it themselves.

If you simply give users errors that say, `500 Internal Server` error, this isn’t very helpful. Not only should the documentation seek to clarify the conditions that trigger errors but the error messages themselves should contain the helpful information.

One could say a lot about error messages beyond the general guidance here. Take a look at the section titled Error condition responses in Microsoft’s REST API Guidelines to get a sense of the potential level of detail around error messages:

> For nonsuccess conditions, developers SHOULD be able to write one piece of code that handles errors consistently across different Microsoft REST API Guidelines services. This allows the building of simple and reliable infrastructure to handle exceptions as a separate flow from successful responses. …
>
> The error response MUST be a single JSON object. This object MUST have a name/value pair named “error.” The value MUST be a JSON object.
>
> This object MUST contain name/value pairs with the names “code” and “message,” and it MAY contain name/value pairs with the names “target,” “details” and “innererror.”
>
> The value for the “code” name/value pair is a language-independent string. Its value is a service-defined error code that SHOULD be human-readable. This code serves as a more specific indicator of the error than the HTTP error code specified in the response. Services SHOULD have a relatively small number (about 20) of possible values for “code,” and all clients MUST be capable of handling all of them….

And on it goes. In other words, there’s a lot to consider even with something like error messages.

Other usability matters

If you’d like to dive deeper into API design and usability, there are plenty of other aspects to cover. Arnaud’s book *The Design of Web APIs* goes into depth here. For example, he mentions some other aspects to consider:

- **Pagination, filtering, sorting**. When you return a large number of items in the response, how easy is it for users to page to the next item, to reduce the scope through filtering, or to apply other sorting options to determine their ordering?
- **Response groupings**. In the responses provided, how is the information grouped and structured? Does this structure clarify the meaning of the elements? Is there too much nesting?
- **Grouping and arrangement of endpoints**. How are the endpoints tagged and sorted in the API? Do the grouping make sense?
- **Right-sizing of endpoints**. Are the endpoints scoped appropriately to return the right information the user needs, or do the endpoints return truckloads of data that might not all be meaningful or useful to users?

Conclusion and additional resources

There’s a lot to say about the design and usability of a REST API. The intent here is not to be comprehensive but to touch on those aspects most relevant and accessible to technical writers. Hopefully, the sections covered here will get you started in assessing and evaluating the design of the API you document.

For a podcast I recorded with Arnaud Lauret on usability, see Podcast: API Design and Usability with Arnaud Lauret (API Handyman).

For an additional resource, see Andrew Johnston’s presentation The Trials and Tribulations of the API Style Guide at API the Docs London 2018.

For more on developer usability, be sure to see the next article in this course: Developer Experience (DevX) Usability (p. 497).

Developer experience (DevX) usability

In API design and usability (p. 488), I covered different aspects of usability related to API design. In this topic, I'll explore usability related to developer processes and technologies in general, independent of API design.

Three types of usability

Usability can be roughly divided into at least three different areas:

- Usability with physical products (p. 497)
- Usability with code products (p. 498)
- Usability with documentation (p. 499)

(Many other divisions could likely be made, for but the discussion here, this division is enough.)

Usability with physical products

Usability with physical products or graphical user interfaces (GUIs) is what people traditionally think of with usability. When you have a physical product to use, assessing its usability is more straightforward.

For example, suppose you're testing the usability of a new headphone design. A usability researcher might bring in a group of beta testers to try the headphones on, ask them to adjust the headphone controls, use the headphones in their daily workflows, and so on. Users might evaluate the headphones based on comfort, familiarity of buttons, how easy they might be to pair to bluetooth, the placement of the microphone, and so on.

Usually, technical writers are too far down the development line to influence product design. By the time the product arrives at your door, most product teams just ask technical writers to provide documentation for the product, not to provide input on design itself. Usually, it would not be expected for you to push back on the design and try to get the product to team to reconsider fundamental aspects such as the placement of the microphone or the fit over your ears. (Of course, you can always try, and different scenarios might be more appropriate for rethinking product design.) You might be able to influence *button names* at this point, and maybe catch glitches in the software (e.g., maybe the bluetooth pairing is wonky), but don't expect too much. Changing product design is usually required before the specs are sent to the factory. After factory lockdown, it's hard to change a product. And documentation is usually something that begins after factory lockdown.

That's not to say that you shouldn't venture forth to comment and critique product design. Just don't get your hopes up if the timeline doesn't allow for drastic changes. By the way, some technical writers get tired of documenting poorly designed products, so they move into usability and product design instead. Once you make this move, you're usually operating outside the standard tech writer domain. For more on playing a product design role, see Playing a product design role as a content designer – podcast with Jonathon Colman ↗.

Product design has a rich history of research and best practices. See this book for many good ideas on how to assess the usability of a product: *Universal Principles of Design: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design* ↗.

Usability with code products

In the developer documentation domain, usability with code products is more common. Few usability researchers venture into the code domain because it's much less clear how to assess the usability of code. But make no mistake, usability is just as much in play with code products for developers as with physical products or products with GUIs.

Code usability asks questions such as the following: How easy is it for developers to use the code? Do the APIs provide the information that developers need? How long does it take developers to figure the code out and implement it? Are you using a language or framework that will be familiar to your developers? Are you following general patterns in the industry for tackling common problems?

Let me illustrate with an example. One product I recently documented was the Video Skills Kit for Fire TV ↗ . The original design involved a cloudside implementation ↗ that had about a dozen different technologies and steps. The implementation required devs to use several AWS services (Identity Access Management, CloudWatch, Lambda), several Appstore services (Login with Amazon and security profiles, Alexa console, Amazon Device Messaging, Live App Testing), a client library (JAR), Fire TV device, Alexa app, and ADB. These services were on top of Android services that developers were expected to know. (Altogether, this is about a dozen different services or tools.)

Unsurprisingly, users were unfamiliar with all of these services, so not only was there an initial learning curve, when they did run into problems, the problems were hard to troubleshoot. Additionally, troubleshooting was also complex not just due to the unfamiliar toolset, but because there were so many different places where something might be misconfigured. For example, after integrating everything, you could say, "Alexa, watch Interstellar," and your app would play a video. But if it didn't work, you had to figure out where the problem was. The logs could provide a clue, but if you had some invalid formatting in your Lambda function, Alexa might just respond, "Something went wrong," with no other logs. Alexa might respond similarly for a lot of different issues, as there were many backend services as well (invisible to the user) that processed the information. For example, one issue might be with the natural language processing logic occurring behind the scenes.

The cloudside implementation took a long time to complete (multiple months, usually). We had a sample app, and each time I would go through the steps to set up the services around the app, it would take the entire afternoon or more. So many pieces had to be in place, and many times it felt like a house of cards — if one little piece was misconfigured (e.g., signing your app with the wrong key), the whole thing didn't work.

Fast-forward two years later. The product team released a new implementation design to the Video Skills Kit called app-only integrations ↗. The app-only integration grounded most of the configuration within Android itself. Developers didn't have to bother with configuring any AWS services, no Lambda functions, no authorization through ADM. Developers instead would need only configure a security profile, capability reporter, and BroadcastReceiver in their Android app to handle incoming messages sent from an on-device routing agent built into Fire TV.

Partners welcomed this approach much more wholeheartedly. Not only did the implementation stay within the technology realm they were familiar with (Android), it didn't cobble together a dozen different services that all had to be working in harmony for success. Troubleshooting was also simplified because most issues were contained within the same Android platform. The new design shortened the implementation time from a couple of months to a couple of weeks.

Finally, the new design also improved the speed of communication among the different services. With the first solution (cloudside), messages had to travel from one service to another (Alexa in the cloud to a Lambda function in AWS, then to the partner's app on Fire TV). With the app-only integration, most of the communication took place on the device itself. Alexa sent a message to an on-device routing agent that communicated with the app (also installed on the device). The communication between components had much less latency.

In general, when evaluating code usability, consider these questions:

- Does the implementation use a technology that your developers are familiar with?
- Does the implementation rely on a small number of core services/components rather than a dozen separate services cobbled together?
- How easy will it be to troubleshoot errors when things go wrong?
- How long will it take developers to complete the implementation?
- How much latency exists for communication among the different components?

Evaluating developer usability can be hard because many times, there are technology constraints that make more streamlined approaches problematic. No one sets out to build a house of cards, but as you encounter one issue after another, with fast deadlines to finish the code, you might adopt solutions that work in the short term but which require a lot of hacks. String together too many hacks and accrue too much technical debt (shortcuts that provide fixes while leaving larger problems unaddressed), and you head down the path to a house-of-cards solution.

In general, product teams building developer tools strive to provide APIs or other tools that simplify the third-party developer's implementation. You want to abstract complexity behind an API, so that the developer need only provide the specified inputs to get the desired outputs in a response. The developer probably doesn't care (or need to know) how the sausage is made on the backend to transform the inputs into outputs. If you can abstract more of this complexity behind the scenes, it will result in a better developer experience.

Assessing the architectural design of a developer solution isn't easy for most technical writers. Unless you have an engineering background and can assess the feasibility of a code solution, you might not have a lot of input. A lot of factors go into an engineering solution, including security, memory usage, infrastructure, efficiency, latency, databases, and more. In most scenarios, tech writers struggle just to understand the workflow and describe it. Going the extra mile to suggest a more efficient and streamlined architectural design is often something that would be "next-level" for tech writers.

However, if you keep an eye out for the questions I mentioned above, this can provide you with a starting point for evaluation and feedback. You're often the first user of the code (outside the product team), so teams value your feedback, even if you're not an engineer.

Usability withy documentation products

You can also incorporate usability with documentation products. How easy is the documentation to understand and follow? Are the topics something that users can search for and find in the documentation? Does the documentation assume that users have deep background knowledge about all the services, or does it help even novice users complete complicated tasks?

As an example, consider the car manual in your glovebox. Suppose your headlight goes out and you want to replace it. Or suppose a light flashes ("LDW") on your dashboard and you need to look up what it means. Does the user manual make it easy to find this information, such as through a search or index? Are the terms used in the manual similar to the terms you would use to describe the features? When you do locate the right section, are the steps easy to follow? Are there visuals that help clarify the information? Can the documentation help even someone who isn't mechanically inclined follow a sequence of steps successfully?

The main usability task of technical writing is to simplify complexity. I wrote an entire series on simplifying complexity. One strategy is to provide users with a map to guide them through a complex number of steps spanning multiple topics. See Principle 1: Let users switch between macro and micro views.

How do you know if your documentation is easy to follow? Just as you can observe users interact with products or interfaces in a usability lab, you can also observe users with documentation. Ask users to do a task they're unfamiliar with, and see how they interact in your help system to locate the information. I outlined some notes from my observations in a usability lab some years ago. See A Few Notes from Usability Testing: Video Tutorials Get Watched, Text Gets Skipped. I observed two basic patterns. Novice users welcomed video tutorials and read the documentation carefully, sometimes multiple times. In contrast, advanced users skimmed and scanned, not reading carefully but rather trying to find a specific piece of information.

In the usability lab, time and time again we observed how something that looks simple usually turns out to be much more complex and difficult for users. Remember this and make your instructions as simple as possible, assuming that users will be reading while looking elsewhere, dealing with interruptions, pausing and resuming the task, and more. Although this documentation scenario was for a GUI-based product, you could do the same with code.

Few of us get opportunities to regularly observe users, so probably most of the feedback will come through asynchronous channels (e.g., email, Slack, tickets.) That's fine, and you will probably get more feedback than you can handle anyway.

Many of the same principles in *Universal Principles of Design: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design* can also be used in documentation. See Introduction to User-Centered Documentation for some ideas.

Video

I recently participated in a short interview about usability for a class on Usability & User Experience. The presentation doesn't cover all the details from above, but it does touch upon usability in different scenarios (e.g., professional writing versus a blog, academic contexts, and other). Kirk St.Amant, a distinguished professor in the tech comm field, conducted the interview with me. You can see his usability course online at https://usability-ux.weebly.com/. Here's the video:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/devx-usability-examples.html] to view the content.

You can view other interviews Kirk has done with usability professionals here:

- Interview with Dr. Katie Panciera of the Milwaukee School of Engineering
- Interview with Dr. Joy Robinson of University of Alabama in Huntsville and Northrop Grumman
- Interview with Maiken Blok-Wahlgreen of TimeLog
- Interview with Dr. Quan Zhou of Metro State University
- Interview with Joe Welinske of WritersUA/ Welinske & Associates
- Featured Lecture with Eva Snee of Google

For more, see Kirk's Guest Lectures.

Chapter 9: The writing process

The writing process refers to the way you write and develop content. There are five main steps to the writing process: planning, information gathering, writing, reviewing, and publishing. Although these aspects of the writing process are general to almost every type of writing, in this section I tailor the information to be specific to writing technical documentation.

Chapter sections:

Overview of the writing process 502

1. Planning 504

2. Information gathering 507

3. Writing 513

4. Reviewing 519

5. Publishing 527

Overview of the writing process

If there's one topic in this course that is glossed over, it's the writing process itself. This is ironic because, more than anything else, it's the content that matters. And this content is likely your primary stewardship.

If you're an engineer, the task of writing might be unfamiliar territory and the blank page intimidating. You're likely burdened by the curse of knowledge, unaware of what jargon might be familiar or unfamiliar, unsure of the assumptions about the user's setup and knowledge. The blank page might give you writer's block.

On the flip side, if you're coming from a humanities background, you might be intimidated by all the programming-specific terminology and concepts, unaware of the technical level and familiarity of the users, unsure of how things work on a code level, confused about what to call various elements, and more.

So whether you're an engineer or a professional writer, writing API documentation can be tough. It is extremely challenging, and most of the how-to texts on writing give you very plain, bare-bones writing instruction.

For example, most texts will explain that you first need to understand your audience's needs, gather your sources, write an outline, draft some content, review and edit the content, and finally publish. These *are* the phases of the writing process, but these general descriptions don't tailor the tasks specifically to the tech writer's scenario nor give you insider tips on how to succeed.

Rather than rehash the same writing steps and phases, I'll instead paint more of an inside, hands-on guide to the documentation writing process, with the most practical tips I can give, specifically tailored to technical writers creating documentation.

The writing process involves these five general steps:

1. Planning (p. 504)
2. Information gathering (p. 507)
3. Writing (p. 513)
4. Reviewing (p. 519)
5. Publishing (p. 527)

Entire books could be written about each of these phases. I tried to pick out a few salient tips and practical advice for each section. But by no means is my advice comprehensive. If you're a seasoned pro, I invite you to add your own tips and insights in the comments for each page.

The writing scenario

Before we jump into this process, let's start by presenting a writing scenario. A product team (consisting of a handful of engineers plus a product manager and QA) is creating a new API. In contrast to the more language-agnostic territory of REST APIs, this happens to be a Java API (p. 691). Your target audience involves engineers who know Java and, for the sake of having a concrete use case, they are implementing the API to pull location data about coffee shops. They're using the API to allow coffee vendors to build map-based Android apps that identify which coffee shops in the city sell their brand of coffee.

The API is slated for release next month, so even though engineers have been cracking away at this API for half a year, they're mostly done and ready for you to create the documentation. You have about three weeks to learn the product, write the docs, get the docs reviewed, and publish the docs in time for release.

This is a very typical scenario and timeline, by the way — being brought at the last minute, often unaware of the product’s development cycles that have been going on for months previously. Product teams are long past the stage where they debated the best implementation, the feature set, how the API matches use cases, and so on. By the time you’re brought in, they’ve already entered QA testing stages and just need to put a bow on the product with some docs before shipping it.

Next steps

Continue on to 1. Planning (p. 504).

1. Planning (the writing process)

First things first — you need to figure out what the writing project involves by doing some basic planning. In this step, you'll create a doc plan, establish roles, gather source material, and view a demo.

1.1. Create a doc plan

Create a documentation plan (p. 740), even if it's brief or incomplete. The doc plan will help you get organized and set some expectations about the deliverables and timelines. Answer some basic questions, such as these:

- When is the release for the new API (and hence when is the documentation deadline)?
- What content resources are already available? Any drafts? Product design documents? QA testing documents?
- Where will we publish the docs? Most likely the developer portal where the other docs live, but are there special pre-release needs for early access partners? Is there a need for PDFs or gated access?
- What outputs and formats will we need? Tutorials, API reference, other how-to docs? Documentation for a sample app? GitHub readme's?
- Who are my contacts? Who needs to review and sign off on the content? Identify the product manager, tech lead, engineers, QA, and other stakeholders.

There are a host of other details you could gather, but these are the basics. All writing moves into action when you establish a deadline, so this date should set your wheels into motion.

1.2 Establish who's going to write the docs

The next step to ask is who will do the writing. Although you might think that being a technical *writer* means you should do the writing, this might not be the best approach. This Java API (part of our fictitious writing scenario (p. 502)) might be complicated. Are you really going to articulate how to use the API in a series of detailed steps, with code examples and other observations, if you don't really know Java beyond the one-week course you took two years ago?

You will find that with extremely technical topics, writing might not come as easily as with topics you know well. Some technology can be a tough nut to crack, and if you think you can pull some writing prowess out of your pocket and just hack it out, let me warn you, get ready for engineers to pick the heck out of each phrase and term used in your content as they explain that what you've written is wrong. Engineers aren't shy at all about telling tech writers each nitty-gritty detail of incorrectness.

Depending on your workload and expectations, think about starting with this approach: ask the engineers to write a first draft, even if it's just a brain dump (maybe this content already exists on an internal wiki somewhere). If they want more guidance, you could give them a template with a lot of fill-in-the-blank type sections. See if they can write some initial content.

Many tech writers are spread so thin, they act more like editors, curators, and writing coaches more than actual writers developing content from scratch. It just depends on how your organization has resourced tech writers and what their expectations are. In some ways, it's kind of crazy that the very people who possess all the knowledge about the product aren't the same ones writing the documentation (at least the first draft). So try your hand to push engineers toward this route first.

If you do get engineers to write, you might find that they write crap (like a one-hour stream-of-consciousness style that has never been edited or reviewed after they typed it out). Then you have to spend more time fixing the content and trying to understand whether it's even coherent for the audience, filling in the gaps, and so on. It might be uncomfortable to rework their writing, especially if you're making changes to terminology and processes you're unfamiliar with.

At any rate, just because engineers have "written" content, it doesn't mean you somehow escaped a lot of work — it just gives you a different starting point. Rewriting content can take just as long as writing from scratch. So maybe you won't want to go the path of having engineers write anyway. Either way, whether you have the first "draft" to begin with or need to author it yourself, you still have a lot of work to do.

I'm going to assume that, in this scenario, engineers aren't inclined to write the docs. They do, however, have a lot of source material they've been holding out on.

1.3 Scrounge up all the source material

For any writing tasks, gathering source material is key, but for tech docs, there are a few important documents you should track down. These documents often exist, but unless you ask for them by name, product teams might never mention them or send them to you.

To provide some background first, here's how most software development groups work. Before engineers started working on the feature, someone had to pitch the feature to a program management office (and executive) to green-light the effort. This means they had to write a description of the feature at a high level. There might be a short document (nicknamed something like a "two-pager") that acts as the product pitch, describing the problem and the need for this API. Or there could be a pitch slide deck used to persuade an executive to fund the project. Locate this **two-pager or pitch deck**. If you can identify the nomenclature for this company document, all the better.

Once a product is green-lit, the feature gets planned for a specific timeframe (for example, 2021 Quarter 4), and the team who will develop the feature gets identified. The product team then sizes up the feature and writes a **product design document** that acts as the blueprint for the feature's design. The engineers will debate the proposed design long before any actual code development begins.

During the code development phase, some names used in the product design document will probably change, and some other details as well. The first release is usually a "minimum viable product," or MVP, so it will lack some features described in the document.

Even so, the product design document is usually a goldmine of information for documentation, and you should be able to leverage some information here, such as the product description and a high-level description of the API, the data it provides, important elements, and how to work with it. So make sure you locate this product design document.

The engineers usually populate the **reference documentation** themselves, building Javadoc from annotations in the source code. They may want you to review the reference content, but most comments you'll have will come later in the writing process, after you've written some documentation and become familiar with the product/API.

At any rate, the Javadoc output will give you a good sense of what classes and methods are available in the API. It will communicate the scope and the possible use cases. Ask the engineers to generate a Javadoc output for you to review.

Besides the pitch materials, product design, and Javadoc, there might be other documents as well to gather. Each company tends to call its documents by unique names. Understand what the key documents are for projects in your company, and this will usually unlock people finding and giving them to you. Gather up as many documents as you can find, list them in your doc plan in a resources section, and then browse through these documents to (a) learn about the product, and (b) see what material you can use either directly or indirectly in your docs.

1.4 Ask for a demo of the product

The next step is to ask the product team for a demo of some kind. Perhaps one of the engineers has written a sample app that demonstrates how to call the API in some scenario and what to do with the response. The response might simply be to log data to the console, or to see it rendered in the terminal. That’s fine, and this simplicity is actually what you want at this point. You don’t want an enterprise-scale application that has so many different dimensions and files that the API is obscured. You want a simple sample app that shows how to call the API and what data comes back.

If you can, try to play around with the demo yourself. Try to get the product to work in some way, even if it’s just superficial. For example, you could ask this same engineer who created the sample app to share the sample app with you and provide details on how to run it. Get it working and play around with it a little bit so you get a better sense of what you’re documenting. (See Set up a test environment (p. 345) for more details.)

Almost no engineer develops an API without also developing a sample app that calls the API and gets some data back, though frequently the sample app is just a bare-bones type of code project that logs data. Having some initial familiarity with the product will pay dividends later as you start writing about the feature.

Most engineers, by the way, are usually happy to demo the product they’ve built. This is a point of pride for them, and it’s what they do best. Later, you will be asking these same engineers to review the documentation you’ve written, and then you will see another side to these engineers. They will likely hate reading and reviewing written content. You will have to constantly pull them back on track to avoid sliding into discussions about bugs and future plans rather than continue their document review. But to show off their product and talk through what it’s doing in a demo, they love it.

Next steps

Continue on to 2. Information gathering (p. 507).

2. Information gathering (the writing process)

Now that you've created a doc plan (p. 504), gathered up source material (and presumably read it as well), and observed a demo, you're ready to move on to phase two of the writing process: information gathering through interviews.

How you see yourself: writer versus journalist

As you gather information, a paradigm shift about the tech writing role would help out during this phase. Rather than thinking of yourself as a technical *writer*, think of yourself as a technical *journalist*. As a journalist, you gather information for your story by interviewing people. There are several different groups of people to gather information from:

- **Engineering** — the group that coded the feature
- **Product management** — the group that oversees the feature's planning and strategy
- **Partner engineers** — the group that helps users implement the feature
- **Quality assurance** — the group that tests the feature

These groups will each provide a different input on the feature. Note that groups and titles will vary (especially the term for "partner engineers"), but the four domains represented here are common across companies.

General tips for interviews

The following are several tips for interviews that apply across all groups.

Prepare lists of questions

With each interview, it's good to give the person a list of questions ahead of time. This not only keeps you on track but also reassures the interviewees that the meeting will have a focus and purpose. Some people despise meetings that consume time they could have otherwise been spent coding. A simple list of questions sends the message that the meeting will be worthwhile. For more on this technique, see A tip for doc reviews — bring a list of questions ↗.

Be aware of your role as a product aligner

There's another reason for reaching out to these four distinct groups: any group alone might present a skewed perspective, and without more variety, it's easy to mistake their perspective as the *Truth*. When you write documentation, you often raise awareness about differing points of view that need to be addressed. You might find disagreements between engineering and product management, or between product management and business development, and so on.

For example, in our scenario of creating an API for coffee shops, perhaps the product management group envisions partners being able to add their own coffee shops and supplementary data, including ratings and reviews — essentially enabling partners to both read and write to the API. In contrast, Engineering might have more conservative views about what's possible with the data and might want to promote a more simplified model of updates only going one way — read-only. Writing documentation raises awareness of these differences and will force these groups to align as they review the documentation.

When these differences of opinion surface, don't be surprised. You will occasionally run into a product manager who freaks out about potential content almost going out to partners before it can be edited/tweaked in a certain way, which actually changes the direction provided by some other group. In these scenarios, your role pivots from tech writer to something entirely different: *product aligner*. During these moments, your role is to help define and shape the product's story and messaging.

Rather than inserting yourself into any disputes between groups, let the different groups do their own sparring and other hand-to-hand combat. Your job is mostly just to bring them into the ring together and see who's left standing at the end.

Be aware of engineering myopia

A strategic advantage to the tech writer role is your broad perspective across products. As you interview different groups, you might find that, surprisingly, they lack awareness of other products. Some aspects of the product you're documenting might actually duplicate other products or have other redundancy and conflict. This phenomenon happens most frequently in large tech companies, where many teams work independently across different org lines.

Be aware of your cross-product perspective and recognize that it's a unique advantage you bring to the table. When a PM or engineer is unaware of how the API they're building differs from one that a neighboring team built, don't be shocked, and don't belittle the PM or engineers. These groups are often so specialized, they know only their sliver of the technology. Part of the value you bring to the table is your awareness of other products. Don't be afraid to surface the issues you observe and bring other groups into the conversation.

2.1 Interview the engineers

Now let's get into some strategies for interviews, starting with the engineers. Most likely the engineering team has already designated someone to be your contact point — the main engineer who coded the feature. (If you don't have a contact point, start with the tech lead or team lead for the API; they might designate someone else down the line.)

Set up time with this engineer and prepare a list of questions (the technique I described earlier). Now, this is extremely important: *Record your interview with the engineer.* Even if you're meeting in person, create a conference session in the room on your computer and record the meeting. The engineer is going to dive into gobbledygook and other technical speak, and you'll want to go back and re-listen to this again, even if it's just to get the terminology and phrasing correct. You won't be able to take enough notes in real-time to keep up (unless you're part-engineer already), and you might be too busy taking notes to catch all the details of the conversation.

Some questions to ask the engineer might be as follows:

- Are there any requirements to use this API?
- How do you set it up?
- How do you initialize it?
- Can you talk through the code in the sample app?
- How do you use this API for some common use cases?
- Are there limitations that partners will want to know about?

Pitfall warning: The engineer might give you a Javadoc and say that all the details are contained within the Javadoc, and it's self-explanatory to any Java engineer — as such, there's not much more to say. *All a user needs is a Javadoc to understand the API and implementation details*. For tech writers without an engineering background, this attitude can be difficult to counter because the Javadoc is challenging to read without a background in Java. In fact, just navigating the Javadoc output can be confusing, and you might feel up against a technology barrier, unsure if your lack of understanding will make it impossible to write intelligently about the feature. What does the user know or not know? What is common knowledge in the programming domain and what isn't? Are engineers even using industry standard terms here? You're at a real disadvantage in this situation, but press the engineer to provide some details.

Recognize this attitude ("All the developer needs is this Javadoc!") as a red flag to push back on. You can later get more ammunition for a counter argument when you meet with partner engineers, but for now, press on.

One strategy to get the engineer to unravel more detail is to have him or her talk through the code in the sample app, which was presumably used in the demo. The demo provides an instance in which the API was implemented toward some end. The engineer might have all kinds of disclaimers and notes about how the demo doesn't represent a true implementation, or how it's likely not how users will actually implement the API, but at least it's something to go on. You could also loop in the partner engineers to better gauge the knowledge levels and needs that users will have.

As you talk with the engineer, you'll find that a lot of times, the engineer who codes the feature doesn't know how the feature should be messaged or pitched, and doesn't have the larger picture about workflows and product implementation. Just as you might be unfamiliar with how to actually work with the API, the engineers who created it might not know how partners/developers will implement the API in their particular business domain and use cases. Your project engineers know software, but they often don't know the domain or industry where the partners will be implementing the API.

This is a key point and one that throws a monkey wrench in the writing process. Ideally, you might think the engineers who created the feature could walk you through how third-party developers will use it. After all, the API was designed to support various business use cases. Companies don't create APIs without a strong knowledge of who they're for and what scenarios they envision the APIs being used. They're providing a stream of data for a specific business use case. The company using the API will pay large amounts of money for this data.

And yet, does the engineer creating the API actually know how to implement the API in such business scenarios? Not always. As such, the information you get from the engineer will be somewhat general and not specifically tailored to how the APIs are used for key business tasks.

Think of the API and its various methods or endpoints like providing a cupboard of ingredients to a chef. Project teams don't know exactly how the chef (partner) will use the eggs, flour, sugar, butter, molasses, paprika, and cornmeal — because the ingredients could be used in a variety of recipes and intents. From the engineer's perspective, the requirements doc only specified that they create the molasses and paprika, not how they concoct these ingredients together toward some coherent recipe.

Engineers are so specialized, their view of the system often is much narrower than you assume. They might have specialized technical knowledge while lacking a broader understanding of the technology in the context of business domains. As a result, your interview with this development engineer will only take you so far. To gather the other information you need, you need to set up time with several other people.

2.2 Interview the product manager

The product manager (PM) might not be as technical as the engineers, but he or she will have a better handle on product messaging, use cases, audience analysis, and concepts related to the product. The product manager also likes to have input on the docs because the docs are part of the product experience.

Some topics will absolutely require the PM's input. If you start listing known limitations and issues with the feature, the PM will want to have input. If you drop hints about future, related features on the roadmap, you'll need to clear this with the PM as well.

Some questions to ask the PM might be as follows:

- Who is the audience for this API?
- What are some use cases for this API?
- What are some important concepts to know when working with this API?
- When is the release date for the feature?

The PM is sometimes easier to speak with because PMs aren't as steeped in the technology as the engineers, and so you might find this interview more comfortable. However, while the PM is your friend, the PM often has an agenda: to present the product in an extremely positive light, reflecting the successes of the product team (led by the PM). The PM will squirm against listing any warts, limitations, and issues with the product. PMs generally dislike transparency.

Whereas the engineers might note some important issues/limitations, the PM might want you to spin these issues in a positive way. Or the PM might try to persuade you that the omission of limitations is an indirect way of addressing them. For example, the PM might say, *We don't need to say that you can't do [X] with the feature; we just don't mention [X] at all. So by not saying it, we don't have to note this limitation about our product.*

While this might be reasonable in many situations, if X is a feature most users will want or expect to do, the documentation should mention it. Tech writers are supposed to be *user* champions, not product champions. For example, in our coffee API scenario, suppose the coffee API works only for coffee produced in the U.S., not coffee imported from Columbia, Guatemala, or elsewhere. It would be a pretty big gap not to mention this limitation explicitly in the documentation.

For more on the topic of transparency, see Transparency in documentation: dealing with limits about what you can and cannot say ↗.

Keep the PM in the loop of the doc review even if they have little input on the technical aspects of the product. The PM is aware of all the stakeholders that you might not know about, and they can easily pull people into conversations to answer questions. For example, the PM likely connects the dots up the executive chain and can include others in the doc review several levels above you, or across teams that you didn't even know existed. The PM can be your connection point to all the other teams and interested parties related to the feature.

2.3 Interview the partner engineers

Another group to interview is the partner engineers. The title might differ by company — for example, "solutions engineers" or "developer relations." This is the group that interfaces with partners/users to help them implement the company's products. This group is one of the most important to have frequent communication with because they give you the closest feedback from users.

Roles and companies vary of course, but in my experience, I rarely have direct interactions with the users. Instead, the partner engineers are the contact points with the users, and most feedback about users comes from the partner engineers. The partner engineers' familiarity with the partner's experience, friction points, and feature needs/requests gives them their primary value.

Partner engineers appreciate being in the loop about feature development and the documentation efforts, though they might not have much input on the documentation until you show them a draft. Partner engineers tend to work with existing documentation more than future documentation. They might be reactive (responding to current partner issues) rather than proactive (planning for future features and roadmaps, as PMs do).

Whether you choose to interview the partner engineers now or later depends on how much they know about the feature/API and the partner needs. Ideally, meet with the partner engineer who specializes in this feature ahead of time and get a better understanding of the target users and their use cases.

This information might be redundant with the use cases that the PM describes, but maybe not. The partner engineer can describe the business cases, implementation workflows, and other partner details in more depth. You could, at the very least, check the information you gathered from the PM against the partner engineer's knowledge.

Some sample questions to ask the partner engineer include the following:

- Which partners are we developing this feature for/with?
- How is this target partner planning to use the API?
- How does this partner implement the feature within their company? What's their workflow and setup?
- What issues or questions do you anticipate partners will have related to this API?

Usually, when companies develop a new feature, they select a target partner to develop the feature for, and this target partner becomes the beta partner that product teams work closely with. The partner provides constant input about features of the product and helps steer the product's development, feature set, and roadmap against their own needs and requirements. Partner engineers will often be the contact points with this development partner (more so than the PM), so it's key that you plug into this partner relationship — your access to the partner/user often comes directly through the partner engineer.

Most partner engineers are strong champions of documentation because documentation provides a tool that partner engineers need to do their jobs. At some point, partner engineers will be handing partners a link to the docs. As such, partner engineers are invested in having great docs because it makes their job easier. If the docs are poor, the partner engineers will have to hold the partner's hand in the implementation and answer a lot of questions, troubleshoot issues, and more. In contrast, good docs might free them up from many partner meetings and bug threads.

Building rapport with your partner engineers is one of the main strategies you can implement for success with your doc group. Touch base with a partner engineer now to gather more information. Later you'll loop back around with them for documentation reviews.

2.4 Interview the Quality Assurance (QA) group

Another group to interview is the QA group. This group won't have much information about the users and use cases. Instead, QA should have a group of test cases that they are testing the feature against. This list of test cases might help flesh out common tasks with the API. For example, the QA group might verify that the API returns the correct data for a variety of use cases. If you can get a hold of these test cases, they could inform the various how-tos and other technical details that are described in your documentation.

QA teams aren't used to working with tech writers, so you'll need to steer this conversation towards the paths you want. Some questions to ask QA include the following:

- Do you have a list of test cases that you're testing the API against?
- How are you testing the API? Is there a specific test environment you're using?
- Are there any test devices or setup I can use?
- Is there a sample app you're using to do the testing? How do you use this sample app?
- Do you have any documents that describe testing steps?
- Are there any bugs or limitations to be aware of? Where are the bugs tracked?
- Do you have a list of exactly what data the API should return in various scenarios?
- Are there any resources related to this API that might be helpful?

You can later involve QA in the review of the docs, but they tend to have minimal input on the documentation quality. They only seem to be interested in two questions: what do we promise, and does the feature return what we promised? For example, if your docs say that the API provides data for a specific use case, QA might chime in to say that the API doesn't actually provide that data. They rarely comment on clarity, organization, or style of the documentation.

The main benefit from QA will be information about how to test the API. They often have sample apps, test environments, streamlined procedures, and other resources set up to test an API. This can help facilitate how you can explore the API yourself because, as I mentioned earlier, playing around with the API is essential to better understand the product and inform your ability to write. I don't know about you, but I find it hard to write about something that I can't experience and explore for myself. If QA can help unlock that hands-on exploration, it's a huge win.

Next steps

Continue on to 3. Writing (p. 513).

3. Writing (the writing process)

In this section, I'll dive into practical tips and strategies for writing the content. Before you get to this point, I assume you've already done some planning (p. 504) and information gathering (p. 507).

3.1 Separate content into similar groups

Now that you've gathered information and interviewed a handful of people related to the feature, you probably have a large pile of content. For example, you might have a single doc where you've been collecting all of this information (copying and pasting nuggets of information from various docs). That doc might be 20+ pages long.

You must now begin organizing the information. I think of this task like sorting laundry. Start by sorting the content into different piles — put the socks together, the shirts together, the pants together, etc. (See Writing is like sorting laundry — practical advice for tackling documentation projects for a full elaboration of this analogy.) Just group like information together. Each pile will probably become a separate page in your final content output (but maybe not).

3.2 Create an outline for each group

Now that you have different groups of content, create outlines for each group. I described the general shape of different documentation types in the Conceptual topics section (p. 358). For more documentation projects, you'll need several sections:

- Product overview (p. 360) — high-level description of the feature, requirements, prerequisites, geographic availability, components involved, workflow implementation, how to get help, etc.
- Concepts (p. 359) — foundational concepts related to using the feature. These topics might include diagrams or other visuals to assist with understanding.
- Reference (p. 114) — the Javadoc or other reference documentation.
- How-tos — descriptions of how to implement the API and handle various scenarios.

An outline is a list of section headings and bullet points to cover in each section. Creating an outline precedes the actual writing of content. A good outline can reduce the fear of the blank page.

For inspiration about the right sections to include in different content types, check out the Good Docs Project for templates.

You can even review the outline with other groups you interviewed before you begin writing, if you want. It depends on how confident you are and how extensive the documentation is. The outline is basically your content plan.

It might be worth calling out some other fundamentals to tech writing here. In a creative writing project, you might have only a general sense of how you'll get from your starting point to the conflict to the resolution. As creative writers start writing, they get into a flow state, and sometimes the story takes them in directions they didn't intend.

Some novelists talk about how the characters dictate the shape of the story, and a kind of muse takes over and they just start describing the story that the characters themselves write. The story takes them in directions they can't anticipate or don't expect. This flow state might last hours and take them along unfamiliar paths and directions.

Writing documentation is a much different experience. You're operating in an *explanatory* mode, not a *discovery* mode. It's unlikely that your content will deviate much from the outline. Sure, to some degree, as you begin writing, the flow of concepts and sentences might veer from the outline or list of bullet points, but not like a creative writing flow.

(Despite all the differences from creative writing, at the heart of most types of writing is conflict. Story is the path taken to overcome conflict. So at this foundational level, there is still some parallel between creative writing and technical writing, but not a ton.)

No matter the type of writing, an outline is your friend. In an outline, the initial bullet points don't need to be polished or have strong coherence. You're just sketching out the plan, and you can rearrange sections and bullet points as much as you want. If the documentation is extensive and you're unsure about the approach, you can set up meetings with the same people previously described to talk them through the outline.

Your outline should call out major sections. Note that unlike other forms of writing, documentation uses a lot of section headings. An academic essay might not have any subheadings, but documentation uses section headings every few paragraphs. The idea is that users skim and jump around rather than reading linearly, so this section-dominant style fits the user behavior about how people read documentation.

Section headings also reduce the need for a more formal order and flow to the content, so in some ways writing documentation is easier than other forms of writing. You can always just tack on information in another section heading.

3.3 Set up meetings with reviewers far in advance

Before you start writing, take a few minutes to set up some review meetings. Get on the calendars of your reviewers now. Most reviewers like to have a few others reviewing the document at the same time, and if you wait until you actually have a draft ready, chances are there won't be any open time slots when those reviewers are all free. But if you schedule it 2-3 weeks in advance, you can ensure their availability and a room if necessary. Scheduling a review meeting also gives you a deadline.

In the review meeting details, indicate that you'll later update the event with a link to the draft. However, as I'll explain in Reviewing (p. 519), most reviewers won't read anything until the actual meeting anyway.

3.4 Write the first draft

Now that you have an outline, you're ready to start writing. This is one area that's difficult to give instruction for, but I'll do my best, sharing what works for me.

When I have writing to do, I often find myself looking around for anything else to do — check email, Twitter, Slack, browse Amazon, and so on. It's easy to suddenly think of all the other things I have to do and to focus my energy there because the task of *writing requires a lot of mental energy*. Writing isn't something I can sustain for long periods of time, especially if I'm already fatigued. As a result, to find the time and energy for writing, I maximize the part of my day when I have the most energy for this task: mornings.

After I wake up and have some coffee and breakfast, check email and news, I devote this time to writing. As the day wears on, my energy gets lower and lower, and by evening, I'm usually spent. Most people have an optimal time where their energy is at max productivity. When I was in college and didn't have kids or a full-time job, my writing energy peaked from about 8 to 11 pm. But now, during those same evening hours, I'm usually cruising in escape mode. Figure out when your mental energy is at full capacity, and then plan to write during that time.

The second strategy I use is to set a four-hour timer using a Focus app. This is just an app that keeps track of time on task, following the strategy of the Pomodoro technique. I find that four hours of writing is the most that I can squeeze out during a day. I set 4 one-hour sessions, and if I find myself getting off task during the session, I pause the timer. (For more on this technique, see Writing productivity tip: Focus sessions.)

With the timer set, I look at the outline, and then I just dive right into it. Even if the words don't flow at first, I just start typing and usually after a few paragraphs, I'm in a rhythm. (If you really struggle to start, just follow a stream-of-consciousness approach until your brain gets in gear.)

I allow myself to veer from the specific outline if the flow of the content dictates it. As a best practice, I try to connect the ideas from one sentence to another in a flow-like way. The general pattern is that a sentence ends on an idea that the next one begins on. If you keep that flow, from old-to-new information, it will provide more of a coherence and readability to your sentences.

I also keep my paragraphs relatively short, usually several sentences each and focusing on a single main idea. After several paragraphs, I start a new subheading that encapsulates the focus of the next paragraphs.

I like to listen to music while I write because it reduces my sense of hyper-correctness and allows me to access the part of my brain that is more focused on creating than on grammar and accuracy.

It's good to get the first draft of content on the page. After that, I can begin shaping it and refining it in a much less demanding way.

Even though I've been a professional writer for many years, I don't have a lot of writing stamina. After I get several hours of writing in, I often tap out. Writing is somewhat like exercising — after a couple of good hours max, I'm done. I can't just sit down for hours on end writing documentation. Very few can, especially given how much preparation is required to get the knowledge to write technical documentation. Instead of spending long hours writing, I try to chip away at larger content projects a little bit each day over a series of weeks. This approach (writing a little each day) usually ends up being more successful.

3.5. Make a running list of questions you have

While I'm writing, I usually have plenty of questions regarding things I'm unsure about or which are unclear. I make a list of these questions and use them later as I proceed through the review process. It's painful for reviewers to read docs from beginning to end, and reviewers will much prefer to be asked questions, as it's much easier. This also lets me zero in on those gray areas that I'm unsure about, maximizing the review time.

But for now, as I write, I just throw the questions into the questions doc. (I also sometimes put them in brackets directly in the documentation draft, though I find this clutters things up more than I like.) Later, when I'm done with the draft and ready to tackle the questions, I sort and organize the questions into logical groups.

3.6 Editing

Once I have a first draft of content (no matter how bad), I'm ready to start editing. I try to space out my edits across multiple days. Prior to writing new content, I spend time reading over and revising the previous chunk of content I wrote. I find this helps reorient me about the content I've already written and allows me to fix or polish it with semi-fresh eyes.

Additionally, many days I don't feel like writing, but if I rev up my brain by reading and editing the previous chunk of content I wrote (for 30 minutes to an hour), I'm usually ready to push forward and write new content. This time for editing also helps reconstruct the context of what I was previously writing, bringing many of the details into focus so they're accessible when I begin writing again. (Writing is easier when you have this context fresh in your mind.)

Editing requires a fresh perspective, so I allow myself some space between edits — usually about a day. If I just keep editing the same content over and over all day long, especially content that I've only recently written, I become blind to so many aspects of it. I can't see all the problems. Somehow, allowing myself 24 hours focusing on other things enlivens my ability to see issues and fix errors in the content.

Ideally, if I can space out edits each day over the course of a week, that gives me 5 separate editing sessions. I'm usually amazed at how much I can fix and polish during this time period. It's really the secret to good writing. In contrast, if I crank out the content the day before and release it, the content isn't nearly as good as putting it through a cycle of edits spaced out like this. Think of it like bread that you have to let rise, followed by a session of kneading. When you make bread from scratch, you might need to let it rise a few times and punch it down to get it into the right shape.

3.7 Supplement confusing concepts with visuals and code examples

The nature of writing technical documentation is to simplify and communicate complex concepts and techniques. You might find that words alone fall short of this task. To simplify complexity, leverage these two tools:

- Visuals (p. 516)
- Code examples (p. 517)

Visuals

Simple conceptual diagrams are easy to create. You don't need to be a graphic artist to draw boxes and arrows in a simple diagram to illustrate a concept. Download a program like Sketch ↗ and create some basic visuals to illustrate technical concepts. Keep the diagrams simple, focusing on the aspect you're trying to explain.

If you keep the graphics simple, you'll maintain control over the graphic assets and be empowered to create them as needed. In contrast, if you ask a graphic designer to create the graphics, you might find that it's impossible for you to create more graphics matching their same style and brand. I highly recommend reading The Back of the Napkin ↗ by Dan Roam (and my related post Drawing as a Tool for Thinking: The Back of the Napkin ↗. See also Creating professional looking graphics in the easiest, simplest way possible ↗.

Overall, Roam argues that you should get rid of the idea that graphics can only be rendered by professional artists. Use graphics as a tool for thinking and for articulating complex ideas. The hard part about creating the graphic is coming up with an idea for depicting a concept. When that's your focus, simple diagrams are okay. Little boxes with text below them and some arrows to other boxes showing relationships between the boxes might be all you need to get your point across. If the graphic accomplishes its purpose — to help bring a confusing concept into focus — then it succeeds.

Here are some general techniques to follow when creating visuals:

- To show a difference between two things, consider showing graphics side by side with a slight difference between them.
- To keep the minimalist focus, try to communicate your ideas by showing as few lines as possible. Check out Jean-luc Doumont's Trees, maps, and theorems ↗ for inspiration. Or read about how Edward Tufte despises chart junk ↗.

- Use a change in color to draw attention to something; one subtle color change in an element can bring the reader's focus to what you're trying to highlight. (This is a tip from Doumont's book.)

Overall, creating visuals might be a welcome change from writing. You're still working but not taxing the same part of the brain.

Code examples

Code examples are another way that confusing concepts take on more clarity. I explored code samples more fully in Code samples (p. 464), so I won't repeat everything there except to summarize the main reasons why code helps clarify details. You can write a ton of content trying to describe technical concepts and techniques, but sometimes a simple code sample can speak for itself because it's in a language that the reader understands.

For example, suppose you're trying to explain a technique for getting certain data using a combination of APIs. You could list a few bullets explaining the overall process, but users might get a lot more out of viewing a code sample showing the technique. Code presents the idea in ways that natural language (e.g., English) simply can't do, no matter how you describe it. Code brings with it a story in itself with classes, methods, and other logic. Code examples take a lot of pressure off of you to explain concepts. It's a win-win — easier for you, and better for the reader.

In fact, in nearly every survey about APIs from SmartBear, "examples" ranks number one in terms of the type of docs users want. So don't be shy. Ask engineers for code samples demonstrating how to approach various tasks. This might be something QA already has, and you just need to request it. I've found that engineers are usually more than happy to supply code samples here and there. They sometimes underestimate the power of code examples in conceptual documentation, so let them know you want them. As you're writing, include little placeholders here and there for code examples.

I chose to highlight the top two techniques for clarifying complex information: visuals and code samples. My larger point is to not rely entirely on writing to do the work. Use these other tools to help simplify complexity.

Don't try to eat an elephant in one day

As you write content, don't get frustrated if you tap out long before the task is done. As I'm writing the content for this section, I've written it little by little over the course of two to three weeks. I try to squeeze in some writing time at the beginning and end of my day. I might spend one day writing a new section, another day reviewing a different section, another day adding new details to the previous section, another day sketching out section headings for an unwritten section, and so on.

I don't try to write everything at once. In fact, as I explained above, I don't think I really could. Writing is kind of like a nitro boost — it requires massive amounts of energy, and I can only tap into this energy periodically for short bursts of time.

If I get exhausted, I take a break. I watch Netflix for a while or walk around a bit. When my energy rebuilds, I start writing again. Getting that first draft is the hard part. If I absolutely can't get going, I just spend 15 minutes typing anything. Eventually, my writing brain usually resolves into focus and the words will take shape. (I might have to delete the first page, but that's okay.) If after 15 minutes I'm still not feeling it, then I bag the effort for the day and try again in the morning.

Once you get that first draft, you can iterate endlessly afterward as you edit and review the content for weeks following. But without some starting point, you won't have anything to shape. NaNoWriMo (National Novel Writing Month) is a good model for getting the first draft out. The goal of NaNoWriMo is to generate an initial draft of a novel in one month. It might be crap, but you can then spend the next year editing and fixing it. You at least have a starting point, a draft to read and evaluate.

Run content through a spell checker

At some point when the content is more or less finalized, run it through a spell checker. You might prefer to do this later, after the reviews, but using a spellchecker now can help eliminate superficial errors so that reviewers aren't distracted by the typos and misspellings but can instead focus on substance.

My favorite spell checker is Grammarly ↗, though this service tends to be blocked in most companies due to the way Grammarly stores data on its servers. See Is Premium Grammarly worth it for identifying style and grammar issues in tech docs? ↗ for more details.

Other options could be the spell checker in Microsoft Word, Google Docs, HyperSTE plugin ↗ (to check for simplified technical English), or Acrolinx (if your company has it).

You might be amazed by how many errors these services catch, especially in lengthy content. I personally prefer to leverage as many of these services as possible. I almost always use Grammarly to check new blog posts and articles on my site. One writing teacher explained that a small grammar or spelling error calls attention to itself and can be like having your fly down while you're wearing a tuxedo. Imagine finishing writing 2,000 words of content, sending it to your entire department, and getting feedback from a higher-up executive that you misspelled a word. That kind of superficial feedback can be deflating.

Next steps

Continue on to 4. Reviewing (p. 519).

4. Reviewing (the writing process)

Now that you've written a draft (p. 513) and have edited the content a few times, you're ready to begin the review process. Unlike other forms of writing, reviews are essential for documentation because you're usually writing about a technical domain that you're not an expert in.

General strategies for reviewing content

This section lists general strategies for the review process. Before we jump into this, hopefully you've already set up some review session times with various stakeholders. If not, try to get time on their calendars now rather than later. It doesn't matter whether you have a draft to review or not. Book the reviews to reserve a spot on the reviewers' calendars.

Review strategies that don't work

I've tried different processes for reviewing content, and I have more or less realized what works and doesn't. Here's what *doesn't work*:

- Sending a general email to a group of people asking them to review content.
- Asking people to review large amounts of content at once.
- Asking people to come to a review meeting having read the content beforehand.
- Scheduling meetings with all the necessary reviewers together at the same time.
- Sending a link to docs and hoping that others will review it without being forced to do so in a scheduled meeting.

Unfortunately, the review process isn't as easy as sending out links and getting detailed feedback soon after. The process I'll suggest below works, based on years of experience. This process would not be how you'd expect to review a sample chapter or essay in a creative writing workshop. In a creative writing workshop, usually you send out about 10-20 pages of material a week ahead of time to the group. They read it during the week, prepare comments, and then share their comments in a round-robin format during the workshop. When I was getting my MFA in literary nonfiction, this was the pattern for how we reviewed content.

So when I began my career as a technical writer, I more or less expected the same workflow. It took years of disappointment and frustration with the review process to realize that this isn't how the corporate world works, and it's not how documentation reviews work.

In some ways, you don't even need to have much technical depth or familiarity with the content you're reviewing. You're *facilitating a process*, bringing people together to collect their feedback, making changes, and aligning different parties. You're arbitrating, aligning, clarifying, etc. By facilitating the review among different groups, you are moving content through this phase of its development.

Giving time during the meeting for reviewers to read the content

What works best is to carve out time during the actual meeting for reviewers to read the content. If you have a 45-minute meeting (an ideal length for review meetings), allow reviewers to either spend the first 20 minutes reading the content before launching into a discussion, or allow them to give feedback in real-time while they're reading. The former technique (giving time to read for 20 minutes) is the basic *modus operandi* of meetings at Amazon ↗, and it works great elsewhere too. It's a little odd to have this quiet time during the meeting, but it's also somewhat reassuring for the reviewers to know that you've given them the necessary time to read the docs during the meeting.

You might think, oh, we'll just ask everyone to read the doc beforehand and come to the meeting prepared with comments. When you do this, usually one person has read the content and others haven't. This frustrates the person who did read (he or she now just has to sit there waiting for others), and makes others feel irresponsible for not having done the preparation. The others end up skimming the content quickly so they can get started with the review. Then when they share their feedback, they make suggestions about adding content that's already there (because they didn't read it carefully enough).

So in general, bake time into the review meeting to allow people to read the content during the meeting. While you'll occasionally encounter reviewers who can do the reviews outside of meetings, these types are rare. Also, I've found that the more executive the reviewer, the more they expect you to present them with content to read on the spot and then take notes as they provide feedback.

Keep the meetings 45 minutes or less

A second technique is to keep the review session somewhat short and painless. Reading, analyzing, and providing feedback on content takes a lot of analytical bandwidth, and most people tap out after about 45 minutes. I've rarely seen any review session stay on track for a full hour. As such, don't try to review more than a handful of pages at a time. The rule of thumb is this: don't review more than the reviewer can read in 20 minutes. That works out to about 6 pages, no more.

People don't have time to review more than this, and you'll observe that after about a half-hour, their attention span wanes. It's at this point that the reviewers try to steer the discussion into easier waters, talking about the feature roadmap or current bugs rather than the documentation. To ensure you stay on track, keep the meeting short and focused.

This means you might need to set up multiple doc review sessions if you have a lot of content to review. That's all right. You could set up several 30-minute review sessions with people, if that's what it takes. It's also a lot easier to book 30 minutes on someone's calendar rather than a full hour, especially if you're including multiple reviewers.

If you have 12 pages for people to read, and only 30 minutes for the review session, consider setting up three review sessions. Give people the first 10 minutes to read about 4 pages and then provide a response. It might seem like you're filling their calendar with lots of doc reviews, but it might be the only way to finish the reviews.

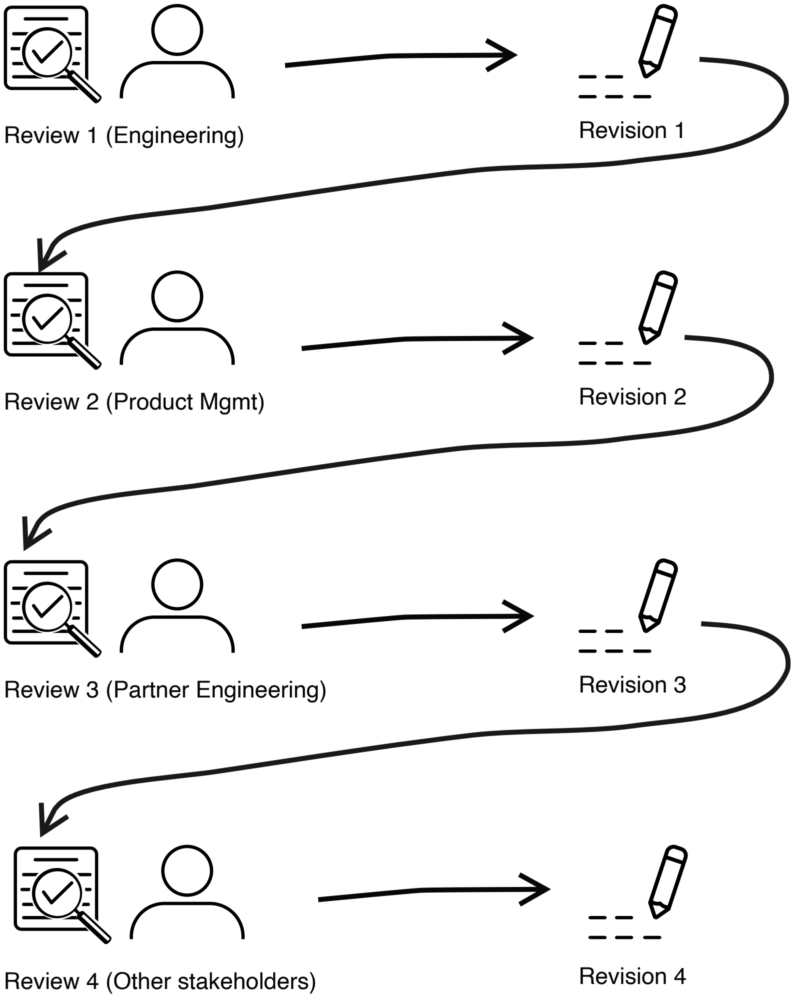
Stagger the review process

Another strategy is to layer the review process. It doesn't make sense to have all reviewers analyze the content at the same time. Ideally, you want to incorporate the edits from each review before the next reviewer sees the content. This way you can iteratively improve the content each time and get a fresh review of the newly edited content.

Without staggering the reviews, if you have five reviewers all look at the content at the same time, they might all point out the same issues. After you fix the issues, you won't have any more reviewers to look at the revised copy because you've already consumed all their reviewing time.

Reviewers do actually have a finite amount of time they're willing to meet with you to review content, especially people in higher-level roles. For example, if you're pulling in a Level 8 or 9 person, don't expect to get more than 30 minutes of their review time total for the entire project. You basically get one shot with them and that's it. Because of this, you want to stagger your reviews so that the higher-ups review content only after the lower-levels have ironed out all of the issues.

To layer this review process, start the review at the product team and expand out from there. After you get the engineers to review the content, review it with the PM, then the partner engineers, then any other stakeholders (e.g, support groups), then business development, and finally any executives if the content requires it. You might not need to include all these groups, but include at least several as appropriate. The review process should look somewhat like this:



The review process is iterative. After each review, incorporate the edits before the next review.

I expand more on this staggered approach in Processes for reviewing documentation (p. 764).

Bring a list of questions

During each review, bring a list of questions that you can ask the reviewer. At first, you can let the reviewers read the content and provide open feedback on any aspect, but as they run out of comments, bring up the questions in your list. You might find the reviewer has more thoughts about topics from your question list than they originally led on. During later reviews, such as follow-up reviews, the list of questions can be instrumental in providing purpose and structure to the review meetings. You can even link the questions to the calendar meeting invite.

During any review session, don't worry about asking dumb questions. One thing I've observed as I get older is that younger people are anxious about asking dumb questions, while older people simply don't care about what others think. Back when I was in grad school and teaching writing composition 101, I observed this difference between young students and returning adult students. The returning adults weren't afraid to stop me mid-sentence and have me re-explain something that they didn't get, even if it seemed somewhat basic. The same adult student could ask five questions in a row without thinking twice about it. Bring that same attitude to doc reviews.

Everyone doesn't need to review all the content

Another principle to remember is that everyone doesn't need to review all the content. You'll have a mix of technical and conceptual material in the docs. A product manager might not have any feedback on the code samples but could have a lot to say about the product overview, messaging, and concepts. In contrast, engineers might not have much to say about the product overview and messaging, but they could have a lot of comments on the code samples.

If you can highlight different parts of the content and ask reviewers to specifically look at those sections, your review meetings will be shorter and more focused. You'll have better results and won't waste your time having people read content that they will have little input on. This suits the idea of shorter, more targeted review meetings.

Best days and times to review content

It might look like you have a full week to schedule people for reviews, but there are optimal days and times to review content. At all costs, avoid Monday mornings. During this time, people are still recovering from the weekend and are inundated with email from people working in other time zones. People in other time zones (India, Australia, Europe) have already been operating for a full day (sending emails, raising issues, and generally working), so if you're meeting with someone in the Pacific time zone, on Monday morning they're slogging through email, issues, and other tasks surfaced by these other groups. Also, people are generally grumpy on Monday mornings (still not in the rhythms of work), and if they already dislike reading and reviewing documentation, by combining the Monday morning with this doc review meeting, they might come across as cold and disinterested.

Another time to avoid is Friday afternoon. By this time, people are usually checked out mentally for the week. From about 2 pm onward on Fridays, it gets quiet at companies. Some people have already completed the week (especially people in time zones that are a day ahead). Additionally, some people leave early for weekend trips or other engagements, or they're just tired from a week of work. The absolute worst thing you can do is schedule a 4 pm Friday meeting for an hour. People will hate you for it.

Another time to avoid is lunchtime. Scheduling a meeting from 12-1 pm is considered somewhat rude, even if people don't eat lunch with anyone. Many people do have lunch engagements and prefer to spend this time interacting socially, but even for loners or workaholics who prefer to eat at their desks, they usually want a little personal time to relax and be idle. By booking their lunchtime, you incur their displeasure (which you'll sense from their tone and attitude during the review).

That leaves you Monday afternoons to Friday mornings. I've found that engineers typically prefer to review docs in the afternoons rather than the mornings. During the mornings, engineers tend to be more focused on coding and have more energy for work; they don't want to be sidetracked with doc reviews during their most optimal coding time. Instead, the early afternoon works much better, when they've accomplished some coding work and are ready to decompress a bit.

Also, be considerate of people in other time zones. If someone is in Australia, you might only have a three-hour window of overlap to catch both you and the reviewer during business hours. Avoid the temptation to conduct a meeting after hours. You might think that to get time on a person's schedule in India, you'll need to join a 7 pm meeting. After you do this several times, wait for burnout to set in. I've seen it happen multiple times. A writer starts accommodating India time zones for doc reviews (because it seems to be the only way to make progress in the doc reviews), and a few months later, this same writer transitions to another team.

Likewise, if someone is in another time zone, don't schedule them outside of their regular business hours. People rarely shift their schedules to review documentation. In a worst-case scenario, the only overlap might be one hour during the early morning, from 8 am to 9 am. If so, this is better than meeting outside of work. However, if this is the only window of time, rest assured that every other team who wants to meet with this person will also be scheduling meetings during that window. In short, if you have to review docs across problematic time zones, plan far ahead for this.

For impossibly busy schedules, ask the reviewer to indicate the time

Another challenge is dealing with reviewers who have impossibly busy schedules (especially common for higher-level people). If the reviewers don't have any gaps in their schedule, ask them when a good time would be. A lot of people fill their schedules with recurring meetings that make them appear to be much busier than they are. They might have meetings on their schedule that they never attend. Some will carve out big chunks of time and label them DNS (for "Do Not Schedule") because they need to have some time to actually work. Politely ask someone when a good time would be to review a doc, letting them know your publishing deadlines and timeframes. They might allocate time during their DNS block.

4.1 Schedule the reviews in a staggered timeframe

Now that you're familiar with the general strategies for reviewing docs, it's time to act. See if you can set up all meetings at once, building toward your deadline. For example, suppose the deadline to publish is in two weeks, and you want to get input from four groups: Engineering, Product Management, Partner Engineering, and other stakeholders. This means you need to schedule 4 meetings. Give yourself time in between to make the edits. Set up meetings like this:

- Monday afternoon - Engineering doc review
- [incorporate edits]
- Thursday - Product Mgmt doc review
- Monday afternoon - Partner Engineering doc review
- [incorporate edits]
- Thursday - Other stakeholders doc review
- [incorporate edits]
- Publish docs!!!

Let them know that they'll have time to read the doc during the meeting.

How to conduct the review session

During the review session, start by telling everyone that you'll give them 20 minutes to review the doc. Then after 20 minutes, ask for general feedback. For example, ask, "Does anyone have any general feedback?" Usually, a few people will start out. Don't steer the discussion just yet. Take notes and allow the reviewers to direct the focus and flow of their feedback. You don't want to bias their perspective or fill them with ideas that they didn't have.

As reviewers list out some issues, don't attempt to defend or counter any of their points (at least not at first). You are in listening mode. Nod your head, periodically summarize what they're saying to show that you understand, and take notes. If you have multiple reviewers, they can check each other's assertions and provide pushback if they disagree.

If reviewers do disagree with each other, you don't need to enter the fray unless they're missing some key piece of background information or detail. For example, maybe you included a certain perspective because the business development (BD) group wanted to pitch the product that way. You could point this out so that reviewers better understand why you took the approach you did. But overall, adopt an attitude of listening. You asked for their feedback. You're not defending a dissertation.

During the meetings, keep on the lookout for quiet people, and ask them questions to give them an opportunity to share their perspectives. Oftentimes, meetings are dominated by more extroverted, vocal types who have a lot of opinions. Introverts might stay quiet because that's just how they roll, even if they have more insightful things to say than the extroverts filling the air with their sound waves.

When you spot a quiet person, ask them politely for their input. For example, "Sam, what's your opinion on this?" You'll find that either (1) they're quiet because they actually don't have an opinion, in which case they'll usually just agree with what everyone else is saying. Or (2) they may start to express some feedback that they were holding back, for whatever reason, and it's actually refreshing and insightful. Look to balance out the voices in the room.

(By the way, this tip about balancing out the voices in a room applies for all types of meetings, not just doc review meetings. See Quiet: The Power of Introverts in a World That Can't Stop Talking ↗ for more on introverts in the workplace.)

If the reviewers run out of questions, bring out your list of questions and start asking your questions one by one. It's at this point that you start to control the focus and flow of the meeting. You know the weak spots and gray areas of your docs better than most reviewers do. Spend time addressing these gray areas, but only after all reviewers have given their feedback on issues that jumped out to them.

4.2 Incorporate edits

After the meeting, incorporate edits into the documentation. Make a tally of all the changes you made, and list these changes in the revision so that others can see that their voices were heard and their feedback was incorporated. When people see that their feedback has been incorporated into the next revision, it helps them feel that their input and participation in the doc review was worthwhile, and they will be more likely to provide feedback in the future. If you don't incorporate their feedback, consider indicating reasons why in your revision notes. You could always lay the blame at another group. For example, "re combining the two topics, the partner engineering group said they needed them separate due to access control issues."

It's worth noting that not all feedback is worth incorporating. Sometimes reviewers are wrong, or the notion they describe during the meeting might have sounded all right when vocalized in the air (and without more context), but it really doesn't work when you try to incorporate it. Or you might simply disagree because you have more product information and a better understanding of partners and use cases than the reviewer does. You are, after all, the writer. Trust your writer's instincts.

Tech writers as product aligners

Occasionally you will find that different groups have conflicting perspectives about what the documentation should say. For example, Engineering might want to be much more hesitant about supporting a feature that Product Management wants to advertise in a more visible way. Or BD might want you to avoid including too many technical details, instead of focusing more on a decision-maker audience than an engineering integrator audience. In contrast, your partner engineers might want you to include more technical details, even including min and max values that no one knows and which BD doesn't care about.

Realizations about misalignment between groups is expected during the review process. Reviewing documentation serves two hidden purposes: (1) it raises awareness about conflicting perspectives on products, and (2) it forces people to come to an agreement. As a writer, because you're the one defining and describing the product, you're the driving force to create the alignment. Just don't be surprised when your doc suddenly spawns a sense of controversy and heated exchange between groups. Sometimes, that's what it's supposed to do. This disagreement has been hiding in the shadows, and now by putting ideas into writing, you've forced the disagreement into the sun — people will have to deal with it.

If you're really caught in a pickle, don't try to take sides or promote your own view. Instead, bring the two parties together in a meeting to hash it out. You might have your own thoughts about the right direction, and since you control the docs, you can choose the direction you want to go. But it usually doesn't go well when tech writers face off against PMs or Engineering in battles like these. You might need to function as a broker between the two sides, trying to find a compromise that appeases both parties.

Tracking down information

During the reviews, some reviewers might ask you for information that neither you nor the reviewers have. For example, they might ask you for min/max values of a certain parameter. Part of your job is to track down this information. If you can't find it, try this blunt force tactic. Identify a wiki page or code file related to the question. Then look for the last 5 authors to work on this content (as noted in the version history for the file). Send an email blast to all five of these people with the question.

They might not have the answer either, but they might suggest where you should go to find the answer. Perhaps there's a special group or email list that supports the technology, and you can ask your question there. This technique works pretty well. As a tech writer, you work across many different groups and roles, so you should be good at navigating the org chart and using different groups, email lists, office hours, or other tools to find information.

Reviewing docs is fun and social

Finally, I want to point out that reviewing docs can be one of the more fun and interesting aspects of the tech writer's job. You might have been working alone on this doc for weeks, but the review period allows you to dip your toes into more social waters. You get to meet people, interact, engage in discussions, and they get to know you and read your work.

It can be exhilarating if they like the docs, and also exhausting. You might get feedback that discourages you because you'll need to track down information, rework sections you thought were fine, or address other large gaps that will require more time and effort than you wanted to spend. Even so, it's worth it. Other times, reviewers might have very little to say except praise for a job well done.

When does the review cycle end

The review cycle ends when you release the product, basically. If there's no release timeframe, it ends when your number of edits approach zero with each revision. With each review session, there should be fewer edits to make from subsequent reviewers. At some point, as you're reading and rereading the docs, you'll end up changing only minor cosmetic style and grammar details. When you hit this point (accompanied by doc nausea), recognize that your changes are no longer moving the needle anymore. You've addressed content accuracy, information about user journeys, and other higher-level concerns.

If you ever find yourself changing a period to a semicolon and then back to a period, stop. You've hit the editing wall. No one cares at this point. Also, if your readers are engineers, they probably won't appreciate the fine stylistic details you might be tweaking anyway. To move the content to the next level, you need partner feedback from the actual people using the product.

Next steps

Continue on to 5. Publishing (p. 527), the final step in the writing process.

5. Publishing (the writing process)

After you've finished the reviews (p. 519) and your deadline is imminent, it's time to make plans to publish your docs. Hitting that publish button will fill you with a rush of exhilaration, but before that moment arrives, you have a few final considerations.

5.1 Understand the larger context where you're publishing

Hopefully, you already know exactly where the content should be published, as this is something you planned for when writing the content. But in case not, make sure you don't just throw another article over the wall onto your developer portal without *integrating* the content into the larger context.

Notice that I used *integrate* instead of *add* — the former implies more work to make the new content fit seamlessly into existing content, whereas the latter implies just tacking on the new content without looking closely at how it fits into the existing content.

Where is the most logical context to integrate the content? Make sure you're well-read enough in your content landscape to see the whole and understand the big picture of your docs. One mistake many writers and engineers make when contributing docs is to focus only on their contribution, without reading more broadly to understand if their contribution conflicts, repeats, or otherwise clashes with what's already published.

If you have a humanities background, you're probably used to reading long, difficult texts and performing critical analyses on the content. Don't abandon those college reading skills in your corporate job. Instead, use your reading superpower to read through the docs on your developer portal that have a similar theme to what you're publishing. For more on this, see Principle 3: Ensure information harmony in the larger landscape ↗.

5.2 Cross-reference the new content

As you read through the larger context on your developer portal, take note of good places to insert cross-references to the new content. If there are similar topics, consider adding cross-references in places where themes overlap. Many times, users search for information and don't always land on the right topic. They might have land close to the needed information, based on the keywords they're searching for, but not on the right page. (This idea aligns with the whole information scent theory ↗ that is part of information foraging.) Include links to help point them to your new content. Conversely, insert cross-references in your new content to other pages.

Some systems have tags that define how related articles surface. If you have a system of tags (defined through a larger taxonomy), make sure your new content has the right taxonomy tags so that it's visible and discoverable in the larger system.

5.3 Include a link in your release notes and what's new

Most developer portals have two sections new content is publicized:

- **Release notes.** The release notes highlight new features in the APIs and other products that have been released, often with a link to the accompanying documentation for the feature.
- **What's new in docs.** The What's new in docs section indicates new articles or updates in the documentation. This list usually covers doc updates only, as these updates can happen independently of feature releases.

Make sure you list and describe your new content using whatever patterns make sense for your developer portal. Many times, program managers literally copy and paste the content from release notes and what's new pages into newsletters that they send out to partners.

Alternatively, you might have a developer relations group or developer marketing that actually has a newsletter. If so, reach out to the group and let them know you have content they could include in their next newsletter.

5.4 Add metadata to pages about authors and last-updated

Another task before publishing your new content is to add metadata to the pages about authorship and date last updated. How you do this depends on your authoring system, but hopefully your authoring/publishing system supports two essential fields: author and last-updated. These fields might be in YAML frontmatter if you're using a static site generator (p. 591), or they could be in more structured fields if using a CCMS ↗.

Don't skip adding metadata. Recognize that right now, at the point of release, everyone is excited about the new content. But it will only be a matter of time before most people forget about this page, before you move on, before the teams that support the content transition to other projects and parts of the org, or before the feature itself is sunset, then deprecated.

Others who follow you will need to know who wrote the content and when it was last updated. (There's nothing worse than arriving at a new job and not having any history about the content's accuracy, freshness, or authorship.) Sometimes the content metadata is automatically tracked through version control — if so, great. But the version control metadata often supplements more explicit metadata tagging within the content itself.

5.5 Coordinate the timing of the release

One final consideration is timing. When should you hit that publish button? Usually, you want to coordinate your timing with the release. After your release goes out, then you publish the documentation. Don't push your new documentation out before the release. It sounds like common sense, but this might be harder to coordinate than it seems.

When I worked at Amazon, for large releases (such as the launch of a new product), there was an elaborate release schedule and a "war room" set up. The release schedule involved a long series of steps required to release the feature, such as turning on services, dialing up the visibility of new features in the UI, launching a general press release and announcement, and more. These release meetings would often start at about 5 am and continue until 9 am. Somewhere in the list of release events was a line item for publishing the documentation. At that planned time, the release manager would say, "Go ahead and publish the docs now."

Our doc publishing process took about 15 minutes to complete, so one time I anticipated my slot a bit early and jumped the gun by publishing about a half-hour early. Due to some other hiccup, the release manager actually wanted me to pull back the docs until some other items had been released. I explained that rolling back the doc release would create all kinds of hassle and havoc, and it wasn't worth it. But after this experience, I was careful to wait my turn to publish and to respect the timing and sequence, even if the publishing task took a long time of just waiting around.

Most doc releases aren't so formal or coordinated, but you should plug into the release process, understand the timing and the expectations about when the docs should go live, and follow it.

5.6 Celebrate the exhilaration of publishing

When you actually hit the publish button (or type `git push` and then press Enter), it can be exhilarating. After months of work, you execute the final step that makes it live. There's a rush in publishing. If you're an engineer and have not published before, this rush of adrenaline might be a new experience. I feel it every time I publish a new blog post or article as well. There's a high associated with sending something out into the world, contributing to a body of knowledge in a new way. Take time to appreciate this moment. Go celebrate in some way. Maybe this means getting a fancy caramel latte at a nearby coffee shop or taking a leisurely stroll around the city at lunch.

After you publish, spend the rest of the afternoon doing something you find rewarding. You'll be giddy and feel like the world is full of rainbows and ponies. It's a natural high not only to publish but to fling that albatross from your neck. No more review cycles and edits for that content — it's done and live. You can move on to other tasks now. You might have been postponing a host of smaller tasks or errands while focusing on this larger writing project. Now you can take care of those little tasks, or focus on frivolous ones. Go chat with someone in the breakroom for a while, or post a few tweets on Twitter. Celebrate the doc release. At most, this high lasts a day or two. Then you return to earth.

5.7 Publicize what you published

After the content is live, you have one more step: publicize what you published. In a documentation newsletter or report, describe what you've recently published or updated in your developer portal, and send this email to the relevant internal email lists who might be interested.

A good cadence for sending this report is monthly, but you could also send it right after publishing if it's a huge release. In a monthly newsletter, I also recommend including some other sections, such as discussions of doc strategies, roadmap, and metrics. For more information on what to include, see Sending doc status reports — a tool for visibility and relationship building (p. 792).

If there's one mistake most tech writers commit, it's being too quiet about the documentation they work on. If others don't know you wrote and published the new content, they might not understand your role, your contribution to the product release, or your value. Being overlooked leads tech writers to feeling that they're second-class citizens, that their contributions are marginal, and other negative attitudes.

The fix is simple: publicize what you publish. Really, you're a writer. Just write, only instead of docs, write the meta-docs that explain the details about what you just published. Share your meta-docs in the form of newsletters, doc reports, meeting notes, or other emails. Repeat this process regularly, and soon everyone will know you and your contributions.

Next steps

This section covered the writing process, including considerations when publishing. The next section, Publishing API Docs (p. 530), will go into publishing tools and workflows, focusing less on writing and more on publishing technology.

Chapter 10: Publishing your API documentation

API documentation often follows a docs-as-code workflow, where the tools to author and publish documentation align closely with the same tools developers use to write, manage, build, and deploy code. Docs-as-code involves using lightweight formats such as Markdown, collaborating through Git or other version control, building your doc site with a static site generator, and deploying it through a continuous build model, where the build happens on the server when you push commits to a particular branch.

Chapter sections:

Overview for publishing API docs 531
Survey of API doc sites 536
Design patterns with API doc sites 540
Docs-as-code tools 551
More about Markdown 556
Version control systems (e.g., Git) 562
Activity: Manage content in a GitHub wiki 569
Activity: Use the GitHub Desktop Client 577
Activity: Pull request workflows through GitHub 586
Static site generators 591
Hosting and deployment options 602
Hybrid documentation systems 607
Using Oxygen XML with docs-as-code workflows 612
Blobr: An API portal that arranges your API's use cases as individual products 626
Which tool to choose for API docs — my recommendations 633
Jekyll and CloudCannon continuous deployment tutorial 637
Case study: Switching tools to docs-as-code 645
Tools FAQ 657

Chapter X: Overview for publishing API docs

As you look for ways to provide value as a technical writer in a highly technical organization, you might find that you do less direct authoring of technical content and more editing/publishing. You might be guiding and directing the publishing of technical content that engineers mainly develop. For this reason, I have a lengthy focus on publishing in this course about documenting APIs.

Why focus on publishing API docs?

The first question about a focus on publishing API documentation might be, *why?* What makes publishing API documentation so different from publishing other kinds of documentation such that it would merit its own section? How and why does the approach with publishing API docs need to differ from the approach for publishing regular documentation?

With API documentation, you're no longer in the realm of GUI (graphical user interface) documentation, usually intended for mainstream end users. A lot of the content for developers is complex and requires a background not just in programming, but in a specific programming language or framework.

As such, you may find that as a technical writer, you're in over your head in complexity and as such, you're reliant on engineers to write more of the content. You end up playing of a doc tooling and workflow role.

In How API Documentation Fails (published in *IEEE Software*), Martin Robillard and Gias Uddin surveyed developers to find out why API docs failed for them. They found that most of the shortcomings were related to content, whether it was incomplete, inaccurate, missing, ambiguous, fragmented, etc. They summarized their findings here:

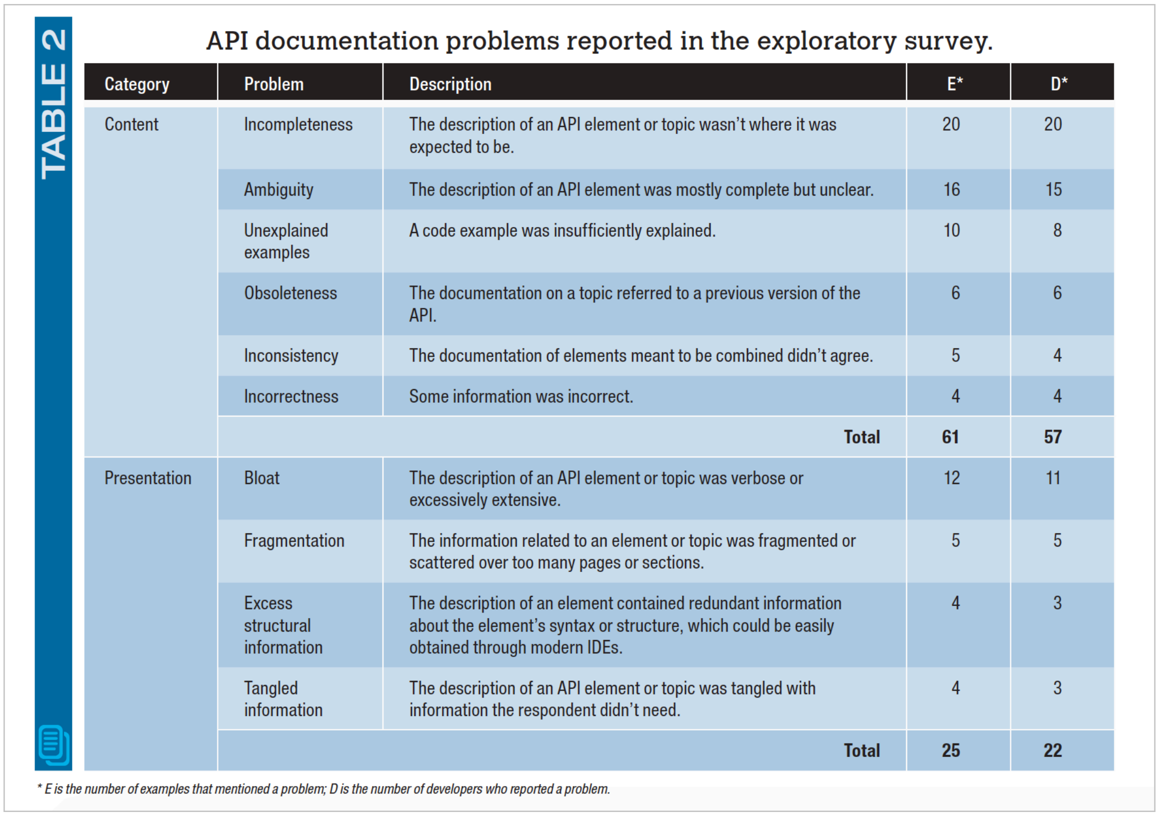


TABLE 2

API documentation problems reported in the exploratory survey.

Category	Problem	Description	E*	D*
Content	Incompleteness	The description of an API element or topic wasn't where it was expected to be.	20	20
	Ambiguity	The description of an API element was mostly complete but unclear.	16	15
	Unexplained examples	A code example was insufficiently explained.	10	8
	Obsoleteness	The documentation on a topic referred to a previous version of the API.	6	6
	Inconsistency	The documentation of elements meant to be combined didn't agree.	5	4
	Incorrectness	Some information was incorrect.	4	4
		Total	**61**	**57**
Presentation	Bloat	The description of an API element or topic was verbose or excessively extensive.	12	11
	Fragmentation	The information related to an element or topic was fragmented or scattered over too many pages or sections.	5	5
	Excess structural information	The description of an element contained redundant information about the element's syntax or structure, which could be easily obtained through modern IDEs.	4	3
	Tangled information	The description of an API element or topic was tangled with information the respondent didn't need.	4	3
		Total	**25**	**22**

** E is the number of examples that mentioned a problem; D is the number of developers who reported a problem.*

Reasons why docs fail for developers

The problem is that the very people who can fix this content are usually fully engaged in development work. Robillard and Uddin write,

> Perhaps unsurprisingly, the biggest problems with API documentation were also the ones requiring the most technical expertise to solve. Completing, clarifying, and correcting documentation require deep, authoritative knowledge of the API's implementation. This makes accomplishing these tasks difficult for non-developers or recent contributors to a project.
>
> So, how can we improve API documentation if the only people who can accomplish this task are too busy to do it or are working on tasks that have been given a higher priority? One potential way forward is to develop recommendation systems that can reduce as much of the administrative overhead of documentation writing as possible, letting experts focus exclusively on the value-producing part of the task. As Barthélemy Dagenais and Martin Robillard discovered, a main challenge for evolving API documentation is identifying where a document needs to be updated.

For example, suppose you identify a high point of developer friction related to poor documentation. Fixing it might not just be a matter of converting the content into plain language or adding some details about missing parameters. The required fixes might involve explaining how the parameters interact in the code, how one value gets used by another and how they get mapped into variables that the code iterates through, etc. Maybe the only person who truly understands the crazy syntax users have to write is the lead developer.

But guess what? What lead developer is going to have time to figure out docs? He or she is usually heads-down deep in a complex programming scenario. So the very person who has the knowledge to decompile and excogitate the needed concepts in the documentation usually isn't available to do so. But if the content is beyond the comprehension of generalists, at some point, these SMEs will need to devote some time to docs. In these scenarios, Robillard and Uddin say the best help would be to reduce the overhead of the documentation process.

As an editor/publisher, you can help the SME author by accurately identifying the point of confusion, the area of the doc that needs updating, and provide easy tools for the SME to make the updates. The engineers can't be bothered to figure out static site generators or publishing workflows, PDFs, or other doc publishing tools. By playing a role as an editor/publisher, you can be a valuable contributor to the product team. This publishing role is why being a doc tools expert is particularly relevant in API documentation contexts.

Using tools your SME authors want to use to collaborate

If engineers and SMEs will be collaborating on some of the doc content, consider using engineering-centric tools rather than writing-centric tools. When I first transitioned to API documentation, I had my mind set on using DITA, and I converted a large portion of my content over to it.

However, as I started looking more at API documentation sites, I didn't find many DITA-based API doc sites. It turns out that almost none of the API doc sites listed on Programmable Web even use traditional tech comm authoring tools.

Despite many advances with single sourcing, content re-use, conditional filtering, and other features in help authoring tools and content management systems, almost no API documentation sites use them. Why is that? Why has the development community implicitly rejected tech comm tools and their many years of evolution?

Granted, there is the occasional HAT, but they're rare. And it's even rarer to find an API doc site that structures the content in DITA.

The short answer is that in API documentation scenarios, more engineers are writing. The content is so technical, they're the only ones who understand it. And when engineers write, they'll naturally gravitate towards tools and workflows they're familiar with.

Andrew Davis ↗, a recruiter who specializes in API documentation jobs in the Bay area, told me that specializing in docs-as-code tools is 100% more advantageous than becoming adept with DITA or some other traditionally technical-writer-oriented tooling.

Davis knows the market, especially the Silicon Valley market, extremely well. Without hesitation, he urged me to pursue the static site generator route (instead of DITA). He said many small companies, especially startups, are looking for writers who can publish documentation that looks beautiful, like the many modern web outputs.

His response, and my subsequent emphasis on static site generators, led me to understand why traditional help authoring tools aren't used often in the API doc space. To make the case even stronger, let me dive into five main reasons why tech writers use docs-as-code tools in developer documentation spaces:

1. The HAT tooling doesn't match developer workflows and environments

If devs are going to contribute to docs (or write docs entirely themselves), the tools need to fit their own processes and workflows. Their tooling is to treat doc as code (p. 551), committing it to version control (p. 562), building outputs from the server, etc. They want to package the documentation in with their other code, checking it into their repos, and automating it as part of their build process. If you're hoping for developers to contribute to the documentation, it's going to be hard to get buy-in if you're using a HAT.

Additionally, almost no HAT runs on a Mac. Many developers and designers prefer Macs because they have a much better development platform (the command line is much friendlier and functional, for example). If you're using a PC, you might struggle to install developer tools or to follow internal tutorials to get set up and test out content.

Even if you could get developers to use a HAT, you'd likely need to buy a license for each contributing developer. In contrast, docs-as-code tools are often open source and can, therefore, scale across the company without budgetary funding and approval.

2. HATs won't generate docs from source

Ideally, engineers want to add annotations in their code and then generate the doc from those annotations. They've been doing this with Java and C++ code through Javadoc (p. 707) and Doxygen (p. 723) for the past 25+ years (for a comprehensive list of these tools, see Comparison of document generators in Wikipedia).

Even for REST APIs, there are tools/libraries that will auto-generate documentation from source code annotations (such as from Java to an OpenAPI spec through Swagger Codegen), but it's not something that HATs can do. For more on auto-generating from source, see Auto-generating the OpenAPI file from code annotations (p. 183).

3. API doc follows a specific structure and pattern not modeled in any HAT

Engineers often want to push the reference documentation for APIs into well-defined templates that accommodate sections such as endpoint parameters, sample requests, sample responses, and so forth. (I discuss these reference sections in Documenting API endpoints (p. 114).)

If you have a lot of endpoints, you need a system for pushing the content into standard templates. Ideally, you should separate the various sections (description, parameters, responses, etc.) and then compile the information through your template when you build your site. Or you can use a specification such as OpenAPI (p. 181) to populate the information into a template. You can also incorporate custom scripts. However, you don't often have these options in HATs, since you're mostly limited to what workflows and templates are supported out of the box.

4. Many APIs have interactive API consoles, allowing you to try out the calls

You won't find an interactive API console (p. 549) in a HAT. By interactive API console, I mean you enter your own API key and values and then run the call directly from the web pages in the documentation. (Flickr's API explorer provides one such example of this interactivity, as does Swagger UI (p. 218).) The response you see from this explorers is from your own data in the API.

5. With APIs, the doc *is* the product's interface, so it has to be attractive enough to sell the product.

Most outputs from HATs look dated and old. They look like a relic of the pre-2000 Internet era. (For more on the dated display, see Tripane help and PDF files: past their prime? from Robert Desprez.)

With API documentation, often the documentation *is* the product's interface — there isn't a separate product GUI (graphical user interface) that clients interact with. Because the product's GUI is the documentation, it has to be sexy and attractive.

Most tripane help doesn't make that cut. If the help looks old and frame-based, it doesn't instill much confidence with developers evaluating it.

In Flare's latest release, you *can* customize the display in pretty significant ways, so maybe it will help end the dated tripane output's appearance. Even so, the effort and process of skinning a HAT's output is usually drastically different from customizing the output from a static site generator. Web developers will be much more comfortable with the latter.

A new direction: Static site generators

Based on all of these factors, I decided to put DITA authoring on pause and try a new tool with my documentation: Jekyll (p. 637). I've come to love using Jekyll, which allows you to work primarily in Markdown, leverage Liquid for conditional logic, and initiate builds directly from a repository.

I realize that not everyone has the luxury of switching authoring tools, but when I made the switch, my company was a startup, and we had only three authors and a minimal amount of legacy content. I wasn't burdened by a ton of documentation debt or cumbersome processes, so I could innovate.

Jekyll is just one documentation publishing option in the API doc space. I enjoy working with Jekyll's code-based approach (p. 551), but there are many different tools (p. 591) and publishing options (p. 602) to explore. That's what we'll dive into in this section.

Now that I've hopefully established that traditional HATs aren't the go-to tools with API docs, let's explore various ways to publish API documentation. Most of these routes will take you away from traditional tech comm tools more toward more developer-centric tools.

Video about publishing tools for API docs

If you'd like to view a presentation I gave to the Write the Docs South Bay chapter on this topic, you can view it here:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_course_videos.html] to view the content.

Survey of API doc sites

Rather than approach the topic of publishing prescriptively, let's begin with some concrete examples and move towards the formulation of general principles. The following are more than 100 openly accessible REST APIs that you can browse as a way to look at patterns and examples.

100+ API doc sites

Browse a few of these documentation sites to get a sense of the variety, but also try to identify common patterns. In this list, I include not only impressively designed docs but also docs that look like they were created by a department intern just learning HTML. The variety in the list demonstrates the many options for publishing tools and approaches, as well as terminology. It seems that almost everyone does their API docs their own way, with their own site, branding, organization, and style.

1. Shopgate API docs
2. Google Places API docs
3. Twitter API docs
4. Flickr API docs
5. Facebook's Graph API docs
6. Youtube API docs
7. eBay API docs
8. Amazon EC2 API docs
9. Twilio API docs
10. Last.fm API docs
11. Bing Maps docs
12. gpodder.net Web Service docs
13. Google Cloud API docs
14. Foursquare Places API docs
15. Walmart API docs
16. Dropbox API docs
17. Splunk API docs
18. Revit API docs
19. Docusign API docs
20. Geonames docs
21. Adsense API docs
22. Box API docs
23. Amazon API docs
24. Linkedin REST API docs
25. Instagram API docs
26. Google Analytics Management API docs
27. Yelp API docs
28. Lyft API docs
29. Facebook API docs
30. Eventful API docs
31. Concur API docs
32. Paypal API docs
33. Bitly API docs
34. Callfire API docs
35. Reddit API docs
36. Netvibes API docs

37. Rhapsody API docs ↗
38. Donors Choose docs ↗
39. Sendgrid API docs ↗
40. Mailchimp docs ↗
41. Basecamp API docs ↗
42. Smugmug API docs ↗
43. NYTimes API docs ↗
44. USPS API docs ↗
45. NWS API docs ↗
46. Evernote API docs ↗
47. Stripe API docs ↗
48. Parse API docs ↗
49. Opensecrets API docs ↗
50. News API docs ↗
51. CTA Train Tracker API ↗
52. Amazon API docs ↗
53. Revit API docs ↗
54. Citygrid API docs ↗
55. Mapbox API docs ↗
56. Groupon API docs ↗
57. AddThis Data API docs ↗
58. Yahoo Weather API docs ↗
59. Glassdoor Jobs API docs ↗
60. Crunchbase API docs ↗
61. Zendesk API docs ↗
62. Validic API docs ↗
63. Ninja Blocks API docs ↗
64. Pushover API docs ↗
65. Pusher Client API docs ↗
66. Pingdom API docs ↗
67. Daily Mile API docs ↗
68. Jive docs ↗
69. IBM Watson docs ↗
70. HipChat API docs ↗
71. Stores API docs ↗
72. Alchemy API docs ↗
73. Indivo API 2.0 docs ↗
74. Socrata API docs ↗
75. Github API docs ↗
76. Mailgun API docs ↗
77. RiotGames API docs ↗
78. Basecamp API docs ↗
79. ESPN API docs ↗
80. Snap API docs ↗
81. SwiftType API docs ↗
82. Snipcart API docs ↗
83. VHX API docs ↗
84. Polldaddy API docs ↗
85. Gumroad API docs ↗
86. Formstack API docs ↗
87. Livefyre API docs ↗
88. Salesforce Chatter RESt API docs ↗

89. The Movie Database API docs
90. SoundCloud API docs
91. Context.io docs
92. CouchDB docs
93. Smart Home API (Amazon Alexa) docs
94. Coinbase docs
95. Shopify API docs
96. Authorize.net docs
97. Trip Advisor docs
98. Pinterest docs
99. Uber docs
100. Spotify API
101. Trello API
102. Yext API
103. Threat Stack API docs
104. Strava API
105. Plaid API
106. Paymill API
107. Rundown API
108. Zype API
109. Yummyly API
110. Edamam Food APIs
111. Veracode API
112. SAP
113. Sunbird API
114. Dwolla API

Tip: If any of the links fail, just type *{product} + api docs* into Google's search, you will likely find the company's developer doc site. Most commonly, the API docs are at developer.{company}.com.

Activity: Look for common patterns in API doc sites

In this activity, identify common patterns in API documentation sites.

1. Go the list of 100+ API documentation sites (p. 536).
2. Select about three different APIs (choose any of those listed on the page).
3. Look for several patterns or commonalities among the API doc sites. For example, you might look for any of the following patterns:

    - Structure and templates
    - Seamless branding (between docs and the marketing site)
    - Abundant code samples and syntax highlighting
    - Lengthy pages
    - API Interactivity (such as an API Explorer)
    - Docs as code tooling

4. Note any non-patterns, such as the following:

    - PDF
    - Translation
    - Video tutorials
    - Commenting features

    - Multiple outputs by role

5. Make some notes in an API log or journal (or share them in the comments below).

In the next section, we'll look at Design patterns with API doc sites (p. 540). From your notes, look to see if the patterns I highlight match the ones you observed in the API doc sites you analyzed.

Design patterns with API doc sites

In the previous topic, we browsed through a long survey of API doc sites (p. 536) and looked for similar patterns in their designs. “Design patterns” are common approaches or techniques in the way something is designed. Although one could probably identify many more patterns, the following design patterns are common with API doc sites: structure and templates, single seamless website, abundant code examples, lengthy pages, and interactive API explorers. I explore each of these elements in the following sections.

Pattern 1: Structure and templates

One overriding commonality with API documentation is that they share a common structure, particularly with the reference documentation around the endpoints. In an earlier section, we explored the common sections in API endpoint documentation (p. 114).

From a tool perspective, if you have common sections to cover with each endpoint, it makes sense to formalize a template to accommodate the publishing of that content. The template can provide consistency, automate publishing and styles, and allow you to more easily change the design without manually reformatting each section. (Without a template, you could just remember to add the same sections on each page, but this requires more effort to be consistent.) With a template, you can insert various values (descriptions, methods, parameters, etc.) into a highly stylized output, complete with sophisticated styling.

Different authoring tools have different ways of processing templates. With Jekyll (p. 637), a static site generator, you can create values in a YAML file (p. 276) and loop through them using Liquid to access the values.

Here’s how you might go about it. In the frontmatter of a page (for example, endpoints.yml), you could list out the key-value pairs for each section.

```
resource_name: surfreport
resource_description: Gets the surf conditions for a specific beach.
endpoint: /surfreport
```

And so on.

You could then use a `for` loop to cycle through each of the items and insert them into your template:

```
{% for p in site.endpoints %}
<div class="resName">{{p.resource_name}}</div>
<div class="resDesc">{{p.resource_description}}</div>
<div class="endpointDef">{{p.endpoint}}</div>
{% endfor %}
```

This approach makes it easy to change your template without reformatting all of your pages. If you decide to change the order of the elements on the page, or if you want to add new classes or some other value, you just alter the template. The values remain the same, since they can be processed in any order.

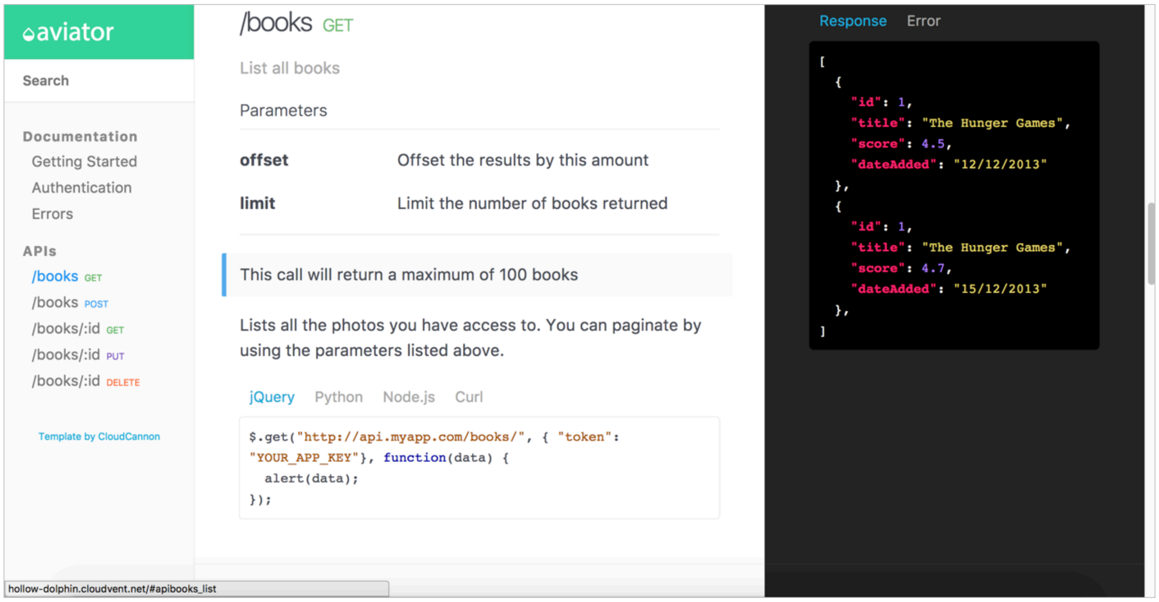
For a more full-fledged example of API templating, see the Aviator theme from CloudCannon. The sample endpoint for adding books in the Aviator theme looks as follows:

```
---
title: /books
position: 1.1
type: post
description: Create Book
right_code: |
  ~~~ json
  {
    "id": 3,
    "title": "The Book Thief",
    "score": 4.3,
    "dateAdded": "5/1/2015"
  }
  ~~~
  {: title="Response" }

  ~~~ json
  {
    "error": true,
    "message": "Invalid score"
  }
  ~~~
  {: title="Error" }
---
title
: The title for the book

score
: The book's score between 0 and 5

The book will automatically be added to your reading list
{: .success }

Adds a book to your collection.

~~~ javascript
$.post("http://api.myapp.com/books/", {
  "token": "YOUR_APP_KEY",
  "title": "The Book Thief",
  "score": 4.3
}, function(data) {
  alert(data);
});
~~~
{: title="jQuery" }
```

(The `~~~` are alternate markup for backticks ` ``` `. The notation `{: .success }` is kramdown syntax for custom classes.) The theme author created a layout that iterates through these values and pushes the content into HTML formatting. If you look in the Aviator's index.html file, you'll see this code:

```
{% assign sorted_collections = site.collections | sort: "position" %}
{% for collection in sorted_collections %}
  {% assign sorted_docs = collection.docs | sort: "position" %}
  {% for doc in sorted_docs %}
     <section class="doc-content">
       <section class="left-docs">
         <h3>
           <a id="{{ doc.id | replace: '/', '' | replace: '.', '' }}">
             {{ doc.title }}
             {% if doc.type %}
             <span class="endpoint {{ doc.type }}"></span>
             {% endif %}
           </a>
         </h3>
         {% if doc.description %}
         <p class="description">{{doc.description}}</p>
         {% endif %}
         {{ doc.content | replace: "<dl>", "<h6>Parameters</h6><dl>" }}
         </section>
         {% if doc.right_code %}
         <section class="right-code">
           {{ doc.right_code | markdownify }}
         </section>
         {% endif %}
       </section>
       {% endfor %}
{% endfor %}
```

This code uses `for` loops in Liquid scripting to iterate through the items in the `docs` collection and pushes the content into the HTML styles of the template. The result looks like this:



/books GET

List all books

Parameters

offset	Offset the results by this amount
limit	Limit the number of books returned

This call will return a maximum of 100 books

Lists all the photos you have access to. You can paginate by using the parameters listed above.

```
$.get("http://api.myapp.com/books/", { "token":
"YOUR_APP_KEY"}, function(data) {
  alert(data);
});
```

```
[
  {
    "id": 1,
    "title": "The Hunger Games",
    "score": 4.5,
    "dateAdded": "12/12/2013"
  },
  {
    "id": 1,
    "title": "The Hunger Games",
    "score": 4.7,
    "dateAdded": "15/12/2013"
  },
]
```

Note that this kind of structure is really only necessary if you have a lot of different endpoints. If you only have a handful, there's no need to automate the template process.

I provided details with Jekyll only as an example. Many of the web platforms and technologies used for API documentation implement a similar templating approach.

When I worked at Badgeville, a gamification startup, we published using Drupal. We had a design agency construct a highly designed template in Drupal. To publish the API reference documentation, engineers wrote a custom script that generated the content from a database into a JSON file that we then imported into Drupal. The import process populated various fields in the Drupal template.

The resulting output was an eye-popping, visually appealing design. To achieve that kind of style in the UX, it would have certainly required a lot of custom div tags to apply classes and other scripts on the page. By separating the content from the template format, we could work with the content without worrying about the right style tags and other formatting. As you look for documentation tools, keep in mind the need to templatize your API reference documentation.

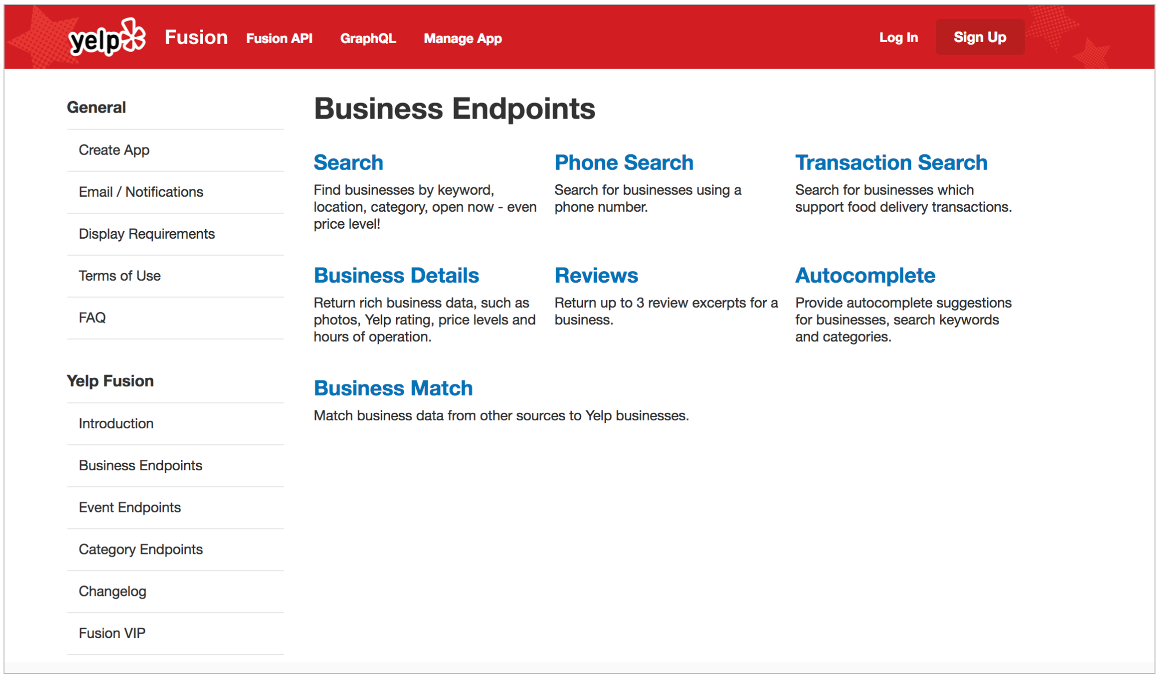
Although it can be fun to create your own templates (if you like to tinker with doc tools), you can probably already see problems related to custom templates. The templates are entirely arbitrary, with terms and structure based on the designer's perceived needs and styles. If you write documentation to fit a specific template, what happens when you want to switch themes or products? You'd have to create new templates that parse through the same custom frontmatter. It's a lot of custom coding.

Given that REST APIs follow similar characteristics and sections, wouldn't it make sense to create a standard in the way APIs are described, and then leverage tools that parse through these standard descriptions? *Yes!* That's what the OpenAPI specification is all about. Earlier in this course, I explained several REST API description formats (p. 180) and then launched into an extensive tutorial for the OpenAPI specification (p. 272). I provided a tutorial for reading the OpenAPI specification using Swagger UI (p. 218), along with an activity to create your own Swagger UI (p. 222).

My point here is that you shouldn't be overwhelmed by the coding challenges around creating your own API templates. The Aviator theme shows one custom approach, and I highlight it here with code samples to demonstrate the complexity and custom-nature of defining your own templates. But this isn't the only approach nor is it even the recommended approach.

Pattern 2: A single seamless website

Many API doc sites provide *one integrated website* to present all of the information. You usually aren't opening help in a new window, separate from the other content. The website is branded with the same look and feel as the product. Here's an example from Yelp:



General

- Create App
- Email / Notifications
- Display Requirements
- Terms of Use
- FAQ

Yelp Fusion

- Introduction
- Business Endpoints
- Event Endpoints
- Category Endpoints
- Changelog
- Fusion VIP

Business Endpoints

Search
Find businesses by keyword, location, category, open now - even price level!

Phone Search
Search for businesses using a phone number.

Transaction Search
Search for businesses which support food delivery transactions.

Business Details
Return rich business data, such as photos, Yelp rating, price levels and hours of operation.

Reviews
Return up to 3 review excerpts for a business.

Autocomplete
Provide autocomplete suggestions for businesses, search keywords and categories.

Business Match
Match business data from other sources to Yelp businesses.

I hinted at this earlier (in Introduction to REST API documentation (p. 22)), but with API documentation, there usually isn't a GUI (graphical user interface) that the documentation complements. In most cases, the API documentation itself is the interface that users navigate to use your product. As such, users will expect more from it.

One of the challenges in using documentation generated from OpenAPI and Swagger (p. 181) or some other document generation process is figuring out how to integrate it with the rest of the site. Ideally, you want users to have a seamless experience across the entire website. If your endpoints are rendered into their own separate view, how do you integrate the endpoint reference into the rest of the documentation?

If you can integrate the branding and search, users may not care. But if it feels like users are navigating several sites that are poorly cobbled together, the UX experience will be somewhat fragmented.

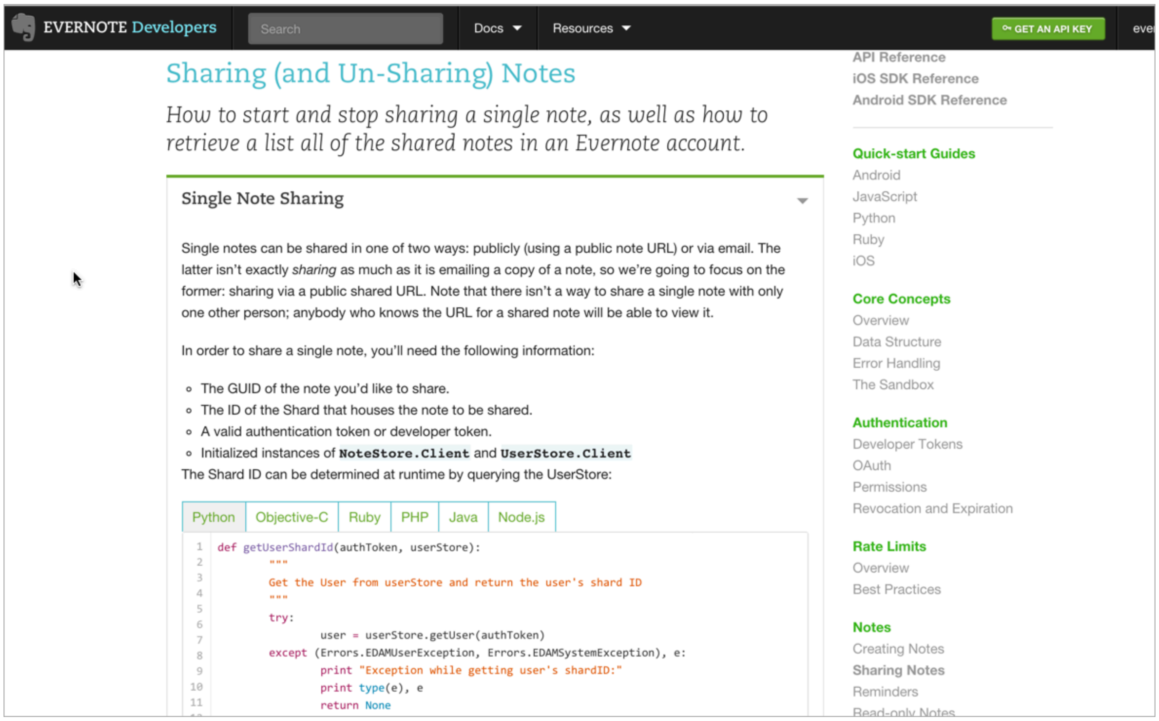
Think about other content that users will interact with, such as marketing content, terms of service, support, and so on. How do you pull together all of this information into a single site experience without resorting to an overbloated CMS or some other web framework?

The reality is that most API documentation sites are custom-designed websites that blend seamlessly with the other marketing content on the site because your API doc must sell your API. As a website platform (rather than a tripane help output), you can leverage all the HTML, CSS, and JS techniques available in building websites. You aren't limited to a small subset of available tools that are allowed by a particular help authoring tool; instead, you have the whole web landscape and toolset at your disposal.

This open invitation to use the tools of the web to construct your API doc site is both a benefit and a challenge. A benefit because, for the most part, there's nothing you can't do with your docs. You're only limited by your lack of knowledge about front-end coding. But it's also a challenge because many of the needs you may have with docs (single sourcing, PDF, variables, and more) might not be readily available with most website tooling.

Pattern 3: Abundant code samples

More than anything else, developers love code examples (p. 464), and the abundance of syntax-highlighted, properly formatted code samples on API doc sites constitutes a design pattern. Usually, the more code you can add to your documentation, the better. Here's an example from Evernote's API:



Sharing (and Un-Sharing) Notes

How to start and stop sharing a single note, as well as how to retrieve a list all of the shared notes in an Evernote account.

Single Note Sharing

Single notes can be shared in one of two ways: publicly (using a public note URL) or via email. The latter isn't exactly *sharing* as much as it is emailing a copy of a note, so we're going to focus on the former: sharing via a public shared URL. Note that there isn't a way to share a single note with only one other person; anybody who knows the URL for a shared note will be able to view it.

In order to share a single note, you'll need the following information:

- The GUID of the note you'd like to share.
- The ID of the Shard that houses the note to be shared.
- A valid authentication token or developer token.
- Initialized instances of `NoteStore.Client` and `UserStore.Client`

The Shard ID can be determined at runtime by querying the UserStore:

```
def getUserShardId(authToken, userStore):
        """
        Get the User from userStore and return the user's shard ID
        """
        try:
                user = userStore.getUser(authToken)
        except (Errors.EDAMUserException, Errors.EDAMSystemException), e:
                print "Exception while getting user's shardID:"
                print type(e), e
                return None
```

James Yu at Parse gives the following advice:

> Liberally sprinkle real world examples throughout your documentation. No developer will ever complain that there are too many examples. They dramatically reduce the time for developers to understand your product. In fact, we even have example code right on our homepage. (Designing Great API Docs)

For code samples, you'll want to incorporate syntax highlighting. The syntax highlighter colors different elements of the code sample appropriately based on the programming language. There are many syntax highlighters that you can usually incorporate into your platform. For example, Jekyll uses rouge by default. Another common highlighter is pygments. These highlighters have stylesheets prepared to highlight languages based on specific syntax.

Usually, tools that you use for authoring will incorporate highlighting utilities (based on Ruby or Python) into their HTML generation process. You don't normally implement the syntax highlighter as a standalone tool. If you don't have access to a syntax highlighter for your platform, you can manually add a highlighting using syntax highlighter library.

Another important element in code samples is to use consistent white space. Use a tool to format the code with the appropriate spacing and line breaks. You'll need to format the code based on the conventions of the programming language. Fortunately, there are many code beautifier tools online to automate that (such as Code Beautify ↗).

Sometimes development shops have an official style guide for formatting code samples. This style guide for code might prescribe details such as the following:

- Spaces inside of parentheses
- Line breaks
- Inline code comment styles

For example, here's the Google JavaScript Style Guide ↗. If developers don't have an official style guide, ask them to recommend one online, and then compare the code samples against the guidelines in it. I dive more into code samples in Code samples and tutorials (p. 464).

Pattern 4: Lengthy pages

One of the starkest differences between regular end-user documentation and developer documentation is that developer doc pages tend to be much longer. Why are the pages longer? In How API Documentation Fails ↗ (published in *IEEE Software* ↗, Martin Robillard and Gias Uddin looked at common reasons why developers had severe problems with API documentation. In addition to incompleteness, ambiguity, and other reasons, the researchers also found that "fragmentation" was cited as a common issue (which is more related to the presentation of content than the content itself).

Robillard and Udin explain:

> When the respondents had to click through multiple pages of an API document to learn the functionality and use of an API element, they found the separation of the descriptions at such a micro level to be unnecessary.

Developers in their survey said they "had difficulty navigating through the myriad pages in an API document to find information," with one respondent explaining:

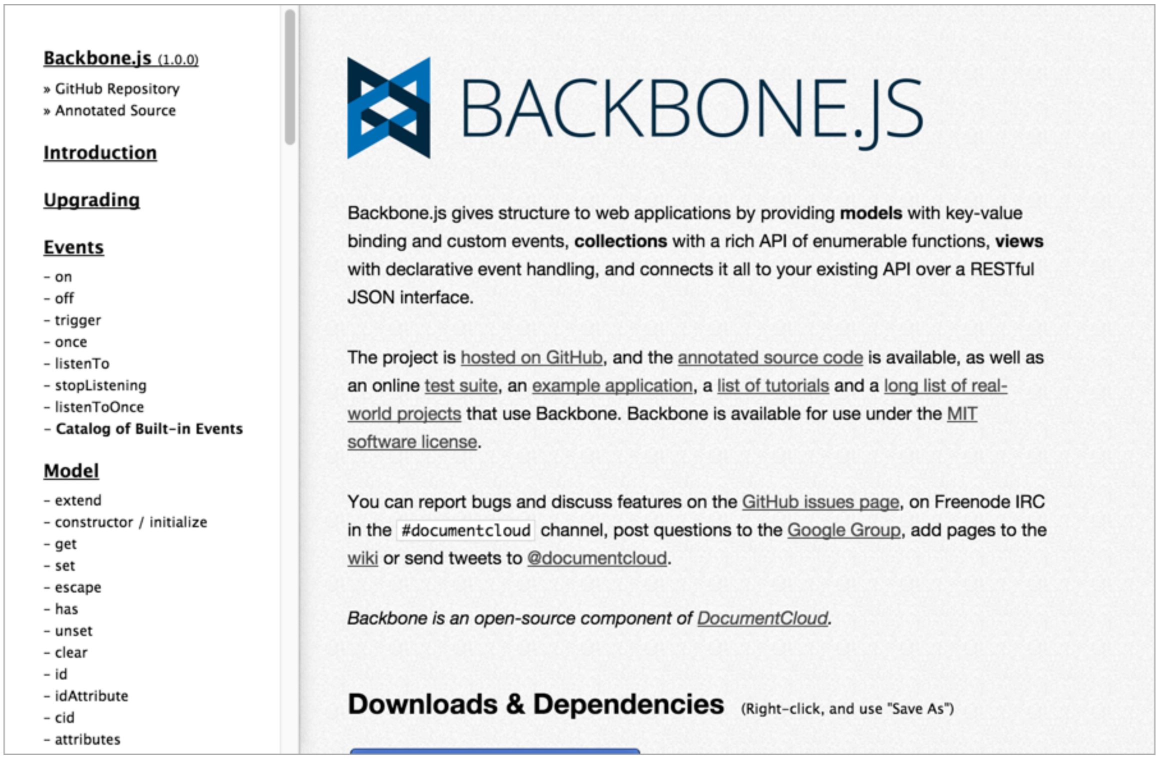
> Fragmented documentation I find really difficult to use, where you have to have 10s of clicks through links to find the information you need, and page after page to read.

If you're using an information model in your documentation that chunks information, make sure your presentation to the user doesn't fragment the content into too many discrete pieces (as I wrote about in Does DITA Encourage Authors to Fragment Information into a Million Little Pieces? ↗).

James Yu at Parse also echoes similar feedback about fragmentation. He says,

> It’s no secret that developers hate to click. Don’t spread your documentation onto a million different pages. Keep related topics close to each other on the same page.
>
> We’re big fans of long single page guides that let users see the big picture with the ability to easily zoom into the details with a persistent navigation bar. This has the great side effect that users can search all the content with an in-page browser search.
>
> A great example of this is the Backbone.js documentation, which has everything at your fingertips. (Designing Great API Docs)

The Backbone.js documentation takes this length to an extreme, publishing everything on one page:


Backbone.js (1.0.0)
» GitHub Repository
» Annotated Source

Introduction

Upgrading

Events
- on
- off
- trigger
- once
- listenTo
- stopListening
- listenToOnce
- **Catalog of Built-in Events**

Model
- extend
- constructor / initialize
- get
- set
- escape
- has
- unset
- clear
- id
- idAttribute
- cid
- attributes

BACKBONE.JS

Backbone.js gives structure to web applications by providing **models** with key-value binding and custom events, **collections** with a rich API of enumerable functions, **views** with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

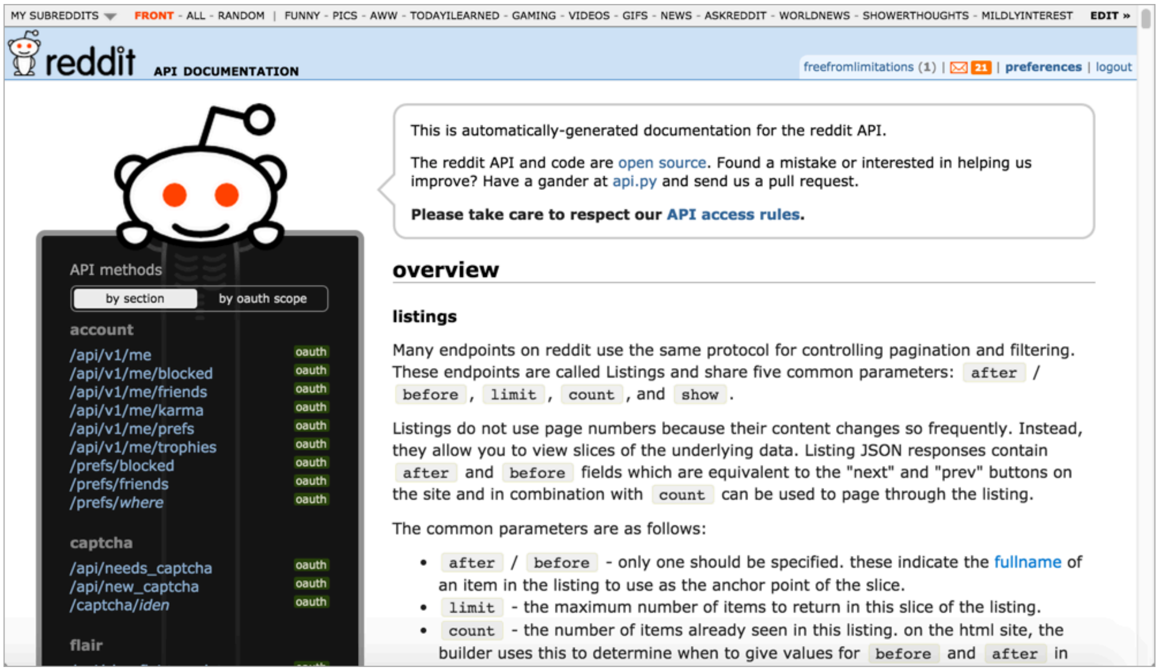
The project is hosted on GitHub, and the annotated source code is available, as well as an online test suite, an example application, a list of tutorials and a long list of real-world projects that use Backbone. Backbone is available for use under the MIT software license.

You can report bugs and discuss features on the GitHub issues page, on Freenode IRC in the `#documentcloud` channel, post questions to the Google Group, add pages to the wiki or send tweets to @documentcloud.

Backbone is an open-source component of DocumentCloud.

Downloads & Dependencies (Right-click, and use "Save As")

For another example of a long page, see the Reddit API:



This is automatically-generated documentation for the reddit API.

The reddit API and code are open source. Found a mistake or interested in helping us improve? Have a gander at api.py and send us a pull request.

Please take care to respect our API access rules.

overview

listings

Many endpoints on reddit use the same protocol for controlling pagination and filtering. These endpoints are called Listings and share five common parameters: `after` / `before`, `limit`, `count`, and `show`.

Listings do not use page numbers because their content changes so frequently. Instead, they allow you to view slices of the underlying data. Listing JSON responses contain `after` and `before` fields which are equivalent to the "next" and "prev" buttons on the site and in combination with `count` can be used to page through the listing.

The common parameters are as follows:

- `after` / `before` - only one should be specified. these indicate the fullname of an item in the listing to use as the anchor point of the slice.
- `limit` - the maximum number of items to return in this slice of the listing.
- `count` - the number of items already seen in this listing. on the html site, the builder uses this to determine when to give values for `before` and `after` in

Why do API doc sites tend to have such lengthy pages? Here are a few reasons:

- **Provides the big picture**: As Yu indicates, single-page docs allow users to zoom out or in depending on the information they need. A new developer might zoom out to get the big picture, learning the base REST path and how to submit calls. But a more advanced developer already familiar with the API might need only to check the parameters allowed for a specific endpoint. The single-page doc model allows developers to jump to the right page and use Ctrl+F to locate the information.
- **Many platforms lack search**: A lot of the API doc sites don't have good search engines. Many lack built-in search features altogether. This is partly because Google does such a better job at search, the in-site search feature of any website is often meager by comparison. Also, some of the other document generator and static site generator tools don't have search (neither did Javadoc). Without search, you can find information by creating long pages and using Ctrl+F.
- **Everything is at your fingertips**: If the information is chunked up into little pieces here and there, requiring users to click around constantly to find anything (as is often the case with DITA's information model ↗), the experience can be like playing information pinball. As a general strategy, you want to include complete information on a page. If an API resource has several different methods, splitting them out into separate pages can create findability issues. See Single Page Docs: Stop the Click Insanity ↗ by Brandon Philips for more. Sometimes it makes sense to keep all related information in one place, with "everything at your fingertips." This is the approach the authors of the OpenAPI spec on GitHub ↗ took — one endless page, with the ability to use Ctrl+F to jump around easily.
- **Today's navigation controls are sophisticated**: Today there are better navigation controls today for moving around on long pages than there were in the past. For example, Bootstrap's Scrollspy feature ↗ dynamically highlights your place in the sidebar as you're scrolling down the page. Other solutions allow collapsing or expanding of sections to show content only if users need it.

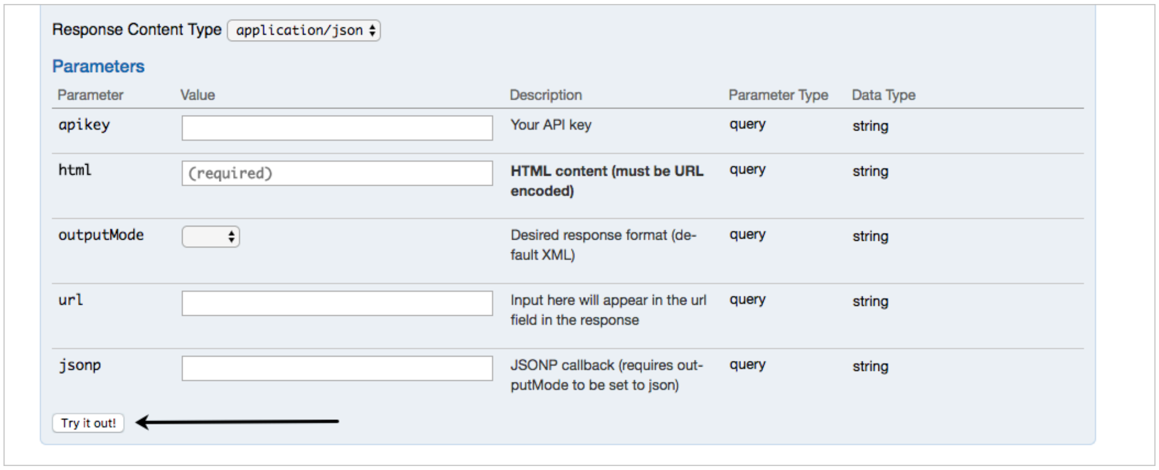
Usually the long pages on a site are the reference pages. Personally, I'm not a fan of listing every endpoint on the same long page. Long pages also present challenges with linking as well. However, I do tend to create lengthier pages in API doc sites than I typically see in other types of documentation.

Pattern 5: API Interactivity

A recurring feature in many API doc publishing sites is interactivity with API calls. Swagger UI, Readme.com, Apiary, and many other platforms allow you to try out calls and see responses directly in the browser.

For APIs not on these platforms, wiring up an API Explorer is often done by engineers. Since you already have the API functionality to make calls and receive responses, creating an API Explorer is not usually a difficult task for a UI developer. You're just creating a form to populate the endpoint's parameters and printing the response to the page.

Here's a sample API explorer from Watson's AlchemyLanguage API that uses Swagger or OpenAPI (p. 181) to provide the interactivity.



Response Content Type `application/json`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
`apikey`		Your API key	query	string
`html`	(required)	**HTML content (must be URL encoded)**	query	string
`outputMode`		Desired response format (default XML)	query	string
`url`		Input here will appear in the url field in the response	query	string
`jsonp`		JSONP callback (requires outputMode to be set to json)	query	string

Are API explorers novel, or instructive? If you're going to be making a lot of calls, there's no reason why you couldn't just use curl (p. 80) or Postman (p. 71) (particularly the Postman Run Button (p. 382)) to quickly make the request and see the response. However, the API Explorer embedded directly in your documentation provides more of a graphical user interface that makes the endpoints accessible to more people. You don't have to worry about entering the right syntax in your call — you just have to fill in the blanks.

However, API Explorers tend to work better with simpler APIs. If your API requires you to retrieve data before you can use a certain endpoint, or if the data you submit is a JSON object in the body of the post, or you have some other complicated interdependency with the endpoints, the API Explorer might not be as helpful. Nevertheless, it is a design pattern to provide this kind of interactivity in API documentation.

If your users log in, you can store their API keys and dynamically populate the calls and code samples with API keys. The API key can most likely be a variable that stores the user's API key. This API-key feature is provided with sites like Readme.com.

However, if you store customer API keys on your site, this might create authentication and login requirements that make your site more complex to create. If you're creating this experience in a custom way, you'll probably need the help of a front-end designer and web developer.

Some non-patterns in API doc sites

Finally, I'd like to briefly mention some non-patterns in API documentation. With many API documentation sites, you rarely see any of the following:

- Video tutorials
- PDFs
- Commenting features
- Localization
- Single sourced outputs for different roles

These might even be considered "non-patterns" in API doc sites. By non-patterns, I'm not saying these elements aren't a good idea, but generally they aren't emphasized as primary requirements in API documentation. If you get pressure to provide these outputs as part of your documentation requirements, look around to see how other API doc sites deliver it. Their frequent omission might inform your own decision and provide some support to make a case for or against the requirement.

Docs-as-code tools

One of the first considerations to make when you think about API doc tooling is who will be doing the writing. If technical writers will create all the documentation, the choice of tools may not matter as much. But if developers will be contributing to the docs, it's generally advantageous to integrate your authoring and publishing tools into the developer's toolchain and workflow. Developer-centric tools for documentation are often referred to as docs-as-code tools. Docs-as-code tools are much more common than traditional help authoring tools (HATs) with API documentation.

Integrating into engineering tools and workflows

Riona Macnamara, a technical writer at Google, says that several years ago, internal documentation at Google was scattered across wikis, Google Sites, Google Docs, and other places. In internal surveys at Google, many employees said the inability to find accurate, up-to-date documentation was one of their most significant pain points. Despite Google's excellence in organizing the world's external information online, organizing it internally proved to be difficult.

Riona says they helped solve the problem by integrating documentation into the engineer's workflow. Rather than trying to force-fit writer-centric tools onto engineers, they fit the documentation into developer-centric tools. Developers now write documentation in Markdown files in the same repository as their code. The developers also have a script to display these Markdown files in a browser directly from the code repository.

The method quickly gained traction, with hundreds of developer projects adopting the new method. Now instead of authoring documentation in a separate system (using writer-centric tools), developers simply add the doc in the same repository as the code. This location ensures that anyone who is using the code can also find the documentation. Engineers can either read the documentation directly in the Markdown source, or they can read it displayed in a browser.

If you plan to have developers write, definitely check out Riona Macnamara's Write the Docs 2015 presentation: Documentation, Disrupted: How two technical writers changed Google engineering culture ↗ .

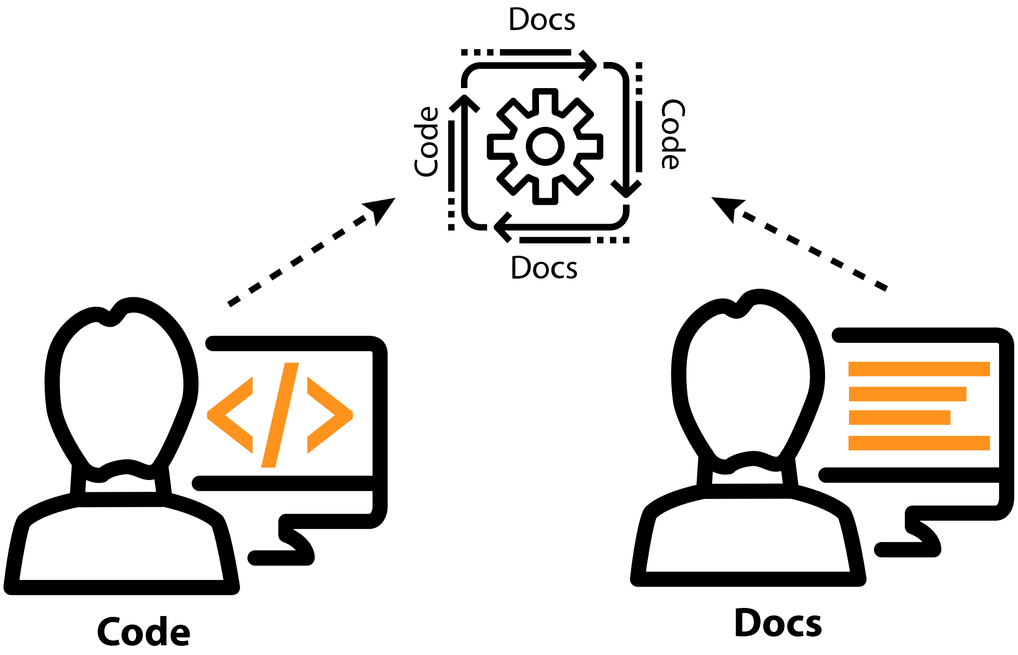
What docs-as-code tools means

Having developers write or contribute to documentation should inform your tool choice with API documentation. If you plan to involve developers in writing and editing, you'll naturally choose more of a docs-as-code tools approach. Docs-as-code means to treat docs just like developers treat code. To treat docs like code generally means doing some of the following:

- **Working in plain text files** (rather than binary file formats like Adobe FrameMaker or Microsoft Word).
- **Using an open-source static site generator** like Sphinx (p. 596), Jekyll (p. 592), or Hugo (p. 594) to build the files locally through the command line (rather than using a commercial program such as FrameMaker or Word).
- **Working with files through a text editor** such as Atom or Sublime Text (rather than relying on commercial tools with proprietary, closed systems that function like black boxes).
- **Storing docs in a version control repository** (usually a Git repo) similar to how programming code is stored (rather than keeping docs in another space like SharePoint or a shared drive); also if appropriate, potentially storing the docs in the same repository as the code itself.
- **Collaborating with other writers using version control** such as Git to branch, merge, push, and pull updates (rather than collaborating through large content management systems or

SharePoint-like check-in/check-out sites).
- **Automating the site build process with continuous delivery** to build the web output from the server when you update a particular branch (rather than manually publishing and transferring files from one place to another).
- **Running validation checks** using custom scripts to check for broken links, improper terms/styles, and formatting errors (rather than spot checking the content manually).
- **Managing docs using processes similar to engineers (e.g., agile scrum)**, such as chunking out doc work in an issue manager (such as JIRA), assigning the issues to bi-weekly sprints, and reporting to stakeholders on the doc work completed (showing demos). (For more on this point, see Following agile scrum with documentation projects (p. 733).)

In short, treating docs like code means to use the same systems, processes, and workflows with docs as you do with programming code.



Treating docs like code

Advantages to docs-as-code approaches for docs

Just because you *can* manage docs like code, should you? What exactly are the advantages of treating docs like code? Here are a few reasons to embrace docs-as-code tools for documentation.

Collaboration with developers

If you work with developer documentation, chances are you'll be working on a wide variety of deeply technical topics and will be reliant on engineers to contribute and review the docs. Many times developer documentation is so complex, only developers can really write and review it. Unless you have a background in engineering, understanding all the details in programming, server configuration, or other technical platforms may be beyond your ability to document (without a lot of research, interviewing, and careful note taking).

See my post What technical writing trends will we see in 2018? for a description of how specialization is forcing technical writers to play more of a generalist role with content.

Additionally, some developers prefer just to write the doc themselves. If a developer is the audience, and another developer is the writer, chances are they can cut through some of the guesswork about assumptions, prerequisite knowledge, and accuracy. In short, in some API documentation contexts, developers write the content. This fact alone is enough to make the case for adopting a docs-as-code approach.

Most developers are comfortable with Markdown, enjoy being able to work in their existing text editor or IDE (integrated development environment) to edit content, prefer to collaborate in a Git repo using branching, merging, and code review tools, and are generally comfortable with the whole code-based process and environment. By using tooling that is familiar to them, you empower them to contribute and participate more fully with the documentation authoring and publishing.

Granted, engineers who write documentation often fall prey to the curse of knowledge. That is, the more they know about a topic, the more assumptions and background information they have getting in the way of clear communication. Even so, technical writers may not always have the time to write documentation for engineering topics. In many cases, a development group that has an API might not even have a technical writer available. Developers might handle everything, from coding to docs.

If tech writers are available, API documentation is usually a collaborative effort between developers and technical writers. Developers tend to focus more on writing the reference documentation (p. 114), while technical writers focus more on the conceptual documentation (p. 358). Regardless of the division of labor, both technical writers and developers tend to work with each other in a close way. As such, docs-as-code tools become essential.

Continuous delivery

Continuous delivery with docs means rebuilding your output by simply committing and pushing content into a Git repository, which then detects and change and triggers a build and publishing job. Continuous delivery greatly simplifies the act of publishing. You can make edits across dozens of pages and commit your code into your doc repo. When you merge your branch into a gamma or production environment, a server process automatically starts building and deploying the content to your server. You don’t have to FTP files to a server or follow some other manual deployment process.

At first, learning the right Git commands might take some time. But after working this way for a few weeks, these commands become second-nature and almost built into your typing memory. Eliminating the hassle of publishing and deploying docs allows you to focus more on content, and you can push out updates quickly and easily. Publishing and deploying the output is no longer a step you have to devote time towards. Continuous delivery is the killer feature that makes docs-as-code so much more effortless (when it comes to publishing) compared to other solutions.

Increased collaboration with other contributors

When your tech writing team collaborates in the same Git repository on content, you’ll find a much greater awareness around what your teammates are doing. Before committing your updates into the repo, you run a `git pull` to get any updates from the remote repository. You see the files your teammates are working on, the changes they’ve made, and you can also more easily work on each other’s content. You can also use the diffs and commits for metrics.

By working out of the same repository, you aren’t siloed in separate projects that exist in different spaces. Docs-as-code tools encourage collaboration.

Flexibility and control

Docs-as-code tools give you incredible flexibility and control to adjust to your particular environment or company's infrastructure. For example, suppose the localized version of your website requires you to output the content with a particular URL pattern, or you want to deliver the content with a special layout in some environments, or you want to include custom metadata to process your files in a particular way with your company's authentication or whitelisting mechanisms. With docs-as-code tools, the files are open and can be customized to incorporate the logic you want. This openness and flexibility can be especially important if you're integrating your docs into a website rather than generating a standalone output (see Pattern 2: A single seamless website (p. 543)).

The docs-as-code tools are as robust as your coding skills allow. At a base level, almost all docs-as-code tools use HTML, CSS, and JavaScript. So if you are a master with these web technologies, there's almost nothing you can't do.

Further, many static site generators allow you to use scripting logic such as Liquid that simplifies JavaScript and makes it easier to perform complex operations (like iterating through files and inserting particular fields into templates). The scripting logic gives you the ability to handle complex scenarios. You can use variables, re-use content, abstract away complex code through templates, and more.

To read details about switching to docs as code tools, see Case study: Switching tools to docs-as-code (p. 645).

Not just tools, but processes too

Historically, discussions about docs-as-code have centered on tools rather than processes. But I think a good case can be made for expanding the definition of docs-as-code to involve implementing engineering processes for managing docs as well. By far the most common approach for software development is agile scrum. I expand on this topic at length in Following agile scrum with documentation projects (p. 733).

Dealing with more challenging factors

A lot of the docs-as-code solutions aren't built with robust technical documentation needs in mind. Suppose you have some of the following requirements:

- Localization
- Content re-use
- Versioning
- Authentication
- PDF

You can often find ways to handle these challenges with non-traditional tools, but it's not going to be a push-button experience. It might require some creativity or a higher degree of technical skill and coding.

At one company where I used Jekyll, we had requirements around both PDF output and versioning. We singled sourced the content into about 8 different outputs (for different product lines and programming languages). It was double that number if you included PDF output for the same content.

Jekyll provides a templating language called Liquid that allows you to do conditional filtering, content re-use, variables, and more, so you can fill these more robust requirements. I used this advanced logic to single source the output without duplicating the content. Other static site generators (like Hugo or Sphinx) have similar templating and scripting logic that lets you accomplish advanced tasks.

To handle PDF with Jekyll, I integrated a tool called Prince ↗, which converts a list of HTML pages into a PDF document, complete with running headers and footers, page numbering, and other print styling (it even uses CSS for the styling) You could also use Pandoc ↗ to fill simpler PDF requirements. PDF is possible, just not usually an out-of-the-box feature (except with Sphinx ↗).

You can handle these more challenging factors with non-traditional tools, but it might require more expertise and creativity.

Conclusion

In the developer documentation space, static site generators dominate the authoring and publishing landscape. HATs and other traditional technical writing tools aren’t used nearly as much. This focus on unique, non-traditional publishing tools is why I’ve dedicated an entire section to publishing in this documentation course.

More about Markdown

Most of the docs-as-code solutions use lightweight markup syntax, often Markdown. So let's learn a bit more about Markdown. Markdown is a shorthand syntax for HTML. Instead of using `ul` and `li` tags, for example, you use asterisks (`*`). Instead of using `h2` tags, you use hashes (`##`). There's a Markdown tag for most of the commonly used HTML elements, but not all of them.

Sample Markdown syntax

To get a sense of the Markdown syntax, look at this sample:

```
## Heading 2

This is a bulleted list:

* fireStructuredText item
* second item
* third item

This is a numbered list:

1. Click this **button**.
2. Go to [this site](http://www.example.com).
3. See this image:

![My alt tag](myimagefile.png)
```

Markdown is meant to be kept simple, so there isn't a comprehensive Markdown tag for each HTML tag. For example, if you need `figure` elements and `figcaption` elements, you'll need to use HTML. What's nice about Markdown is that if the Markdown syntax doesn't provide the tag you need, you can just use HTML.

Development by popular demand versus by committee

John Gruber, a blogger, created Markdown as a way to simplify HTML (see his Markdown documentation here). Others adopted it, and many made modifications to include the syntax they needed. As a result, there are various "flavors" of Markdown, such as Github-flavored Markdown, Multimarkdown, kramdown, CommonMark, and more.

In contrast, DITA is a committee-based XML architecture derived from a committee. There aren't lots of different flavors and spinoffs of DITA based on how people customized the tags. There's an official DITA spec that is agreed upon by the DITA OASIS committee. Markdown doesn't have that kind of committee, so it evolves on its own as people choose to implement it.

Why developers love Markdown

In many development tools (particularly static site generators) that you use for publishing documentation, many of them will use Markdown. For example, Github uses Markdown. If you upload files containing Markdown and use an md file extension, Github will automatically render the Markdown into HTML.

Markdown has appeal (especially with developers) for various reasons:

- You can work in text-file formats using your favorite code editor.
- You can treat the Markdown files with the same workflow and routing as code.
- Markdown is easy to learn, so you can focus on the content instead of the formatting.

Why not use a more semantically rich markup?

Why not use a more semantically rich markup language like DITA? Although you can also work with DITA in a text editor, it's a lot harder to read the code with all the XML tag syntax. For example, look at the tags required by DITA for a simple instruction about printing a page:

```
<task id="task_mhs_zjk_pp">
    <title>Printing a page</title>
    <taskbody>
<steps>
        <stepsection>To print a page:</stepsection>
    <step>
        <cmd>Go to <menucascade>
            <uicontrol>File</uicontrol><uicontrol>Print</uicontrol>
        </menucascade></cmd>
    </step>
    <step>
        <cmd>Click the <uicontrol>Print</uicontrol> button.</cmd>
    </step>
</steps>
    </taskbody>
</task>
```

Now compare the same syntax with Markdown:

```
## Print a page

1. Go to **File > Print**.
2. Click the **Print** button.
```

I wrote about this difference in Why developers will never adopt DITA. Granted, the XML example has a lot more semantic information packed into it, which the Markdown version lacks. So, in theory, the two don't have the same content, even though the output is the same. However, unless you're taking advantage of the additional semantic information DITA includes, the result is the same.

Although you can read the XML and get used to it, most people who write in XML use specialized XML editors (like OxygenXML) that make the raw text more readable. Simply by working in XML all day, you get used to working with all the tags.

But if you send a developer an XML file, they probably won't be familiar with all the tags, nor the nesting schema of the tags. Developers tend to be allergic to XML for at least these reasons:

- Most developers usually don't want to expend energy learning an XML documentation format. Their brains already hurt from all the programming they have to figure out, so with docs, they don't want to add to this technical burden.
- Most developers don't want to spend a lot of time in documentation in the first place, so when they do review content, the simpler the format, the better.

In contrast, Markdown allows you to easily read it and work with it in a text editor.

Most text editors (for example, Sublime Text or Webstorm or Atom) have Markdown plugins/extensions that will create syntax highlighting based on Markdown tags.

Another benefit to Markdown (as opposed to binary tools like Word or Framemaker) is that you can package up the Markdown files and run them through the same workflow as code. You can run diffs to see what changed, insert comments, and exert the same control as you do with regular code files. Working with Markdown files comes naturally to developers. (Granted, you can also work with DITA in plain text files and manage the content in Git repositories.)

Drawbacks of Markdown

Markdown does have a few drawbacks:

- **Limited to HTML tags**: You're pretty much limited to HTML tags. For the times when Markdown doesn't offer a shortcut for the HTML, you just use HTML directly. XML advocates like to emphasize how XML offers semantic tagging (and avoids the div soup that HTML can become). However, HTML5 offers many semantic tags (such as `section`, `header`, `footer`, etc.), and even for those times in which there aren't any unique HTML elements, all XML structures that transform into HTML become bound by the limits of HTML anyway.
- **Non-standard**: Because Markdown is non-standard, it can be a bit of a guessing game as to what is supported by the Markdown processor you may be using. By and large, the Github flavor of Markdown is the most common dialect used, as it allows you to add syntax classes to code samples and use tables. Whatever system you adopt, if it uses Markdown, make sure you understand what type of Markdown it supports.
- **White space sensitivity**: Markdown is white-space sensitive, which can be frustrating at times. If you have spaces where there shouldn't be, the extra spaces can cause formatting issues. For example, if you don't indent blank spaces in a list, it will restart the list. As a result, with Markdown formatting, it's easy to make errors. Documents usually still render broken Markdown as valid when they convert it to HTML, making it hard to catch formatting errors.

Markdown and complexity

If you need more complexity than what Markdown or HTML offers, a lot of tools will leverage other templating languages, such as Liquid or CoffeeScript. Many times these other processing languages (often like a lightweight JavaScript) will fill in the gaps for Markdown and provide you with the ability to create includes, conditional attributes, conditional text, and more.

For example, if you're using Jekyll, you have access to a lot of advanced scripting functionality. You can use variables, for loops, sorting, and a host of other functionality. For a detailed comparison of how to achieve the same DITA functionality within Jekyll, see my series Jekyll versus DITA. In this series, I cover the following:

- Variables and conditional processing
- Creating re-usable chunks (conref)
- Building a table of contents
- Reviewing content
- Producing PDFs
- Creating links

Activity: Get practice with Markdown

To get a sense of how Markdown works, let’s practice a bit with Markdown.

1. Go to an online Markdown editor (such as Dillinger.io).
2. Create the following:
    - Numbered list
    - Bulleted list
    - Word with bold formatting
    - Code sample
    - Level 2 heading
    - code formatted text
3. If desired, copy over the Markdown content from this surfreportendpoint.md file and look at the various Markdown tags.

Markdown handles most of the syntax I normally use, but for tables, I recommend simply using HTML syntax. HTML syntax gives you more control over column widths, which can be important when customizing tables, especially if the tables have code tags.

Markdown and static site generators

If you’re using a static site generator, see the specific Markdown syntax used. With Jekyll, the default Markdown is kramdown.

kramdown gives you more capabilities than the basic Markdown. For example, in kramdown, you can add a class to an element like this:

```
{: .note}
This is a note.
```

The HTML will be rendered like this:

```
<p class="note">This is a note.</p>
```

kramdown also lets you use Markdown inside of HTML elements (which is usually not allowed). If you add `markdown="span"` or `markdown="block"` attribute to an HTML element, the Markdown content will be processed as either an inline `span` or a block `div` element. For example:

```
<p markdown="span">This is a **bold** tag</p>.
```

Normally, the Markdown tags `**` wouldn’t be processed because they appear inside of HTML tags. But through the `markdown="span"` attribute, kramdown will process the Markdown. The content is processed as a `span`, which means no line breaks will appear. If you want to process the content as a `div`, you could also do that. For example:

```
<ul markdown="span">
first section

second section
</ul>
```

The content will be processed as two paragraphs rather than one line. See Syntax in the kramdown documentation for more details.

What about reStructuredText and Asciidoc?

If you're using lightweight markup, you might also be interested in exploring reStructured Text (rST) or Asciidoc. reStructuredText is similar to Markdown, in that it offers lightweight wiki-like syntax for more complex HTML. However, reStructuredText is more semantically rich than Markdown (for example, there's a syntax for notes or warnings, and Python classes).

reStructuredText can be extended, follows a standard (rather than having many variants), and gives you more features specific to writing technical documentation, such as cross-references. See reStructuredText vs Markdown for documentation for a more detailed comparison. If you're using Sphinx, you'll want to use reStructuredText.

Asciidoc also offers more semantic richness and standardization. Asciidoc provides syntax for tables, footnotes, cross-references, videos, and more. In fact, Asciidoc "was initially designed as a plain-text alternative to the DocBook XML schema" (asciidoc-vs-markdown.adoc). As with reStructuredText, you don't have the variety of flavors with Markdown, so you can process it more consistently. Asciidoctor is one static site generator that uses Asciidoc as the syntax. Both reStructuredText and Asciidoc (and other syntaxes) are supported on GitHub.

Objections to Markdown

Some people object to Markdown due to the inconsistency across Markdown flavors. Given that there are so many varieties of Markdown (CommonMark, kramdown, Gruber's original Markdown, Github-flavored Markdown, Multimarkdown, and more), it's hard to create a tool to process Markdown consistently.

Eric Holscher, co-founder of Write the Docs and the Read the Docs platform, argues that Markdown creates lock-in to a specific tool. He says many authors think Markdown is a good choice because many tools support it, and they think they can always migrate their Markdown content to another tool later. However, the differing Markdown flavors make this migration problematic. Eric writes:

> The explosion of flavors and lack of semantic meaning leads to lock in. Once you've built out a large set of Markdown documents, it's quite hard to migrate them to another tool, even if that tool claims to support Markdown! You have a large set of custom HTML classes and weird flavor extensions that won't work anywhere but the current set of tools and designs.
>
> You also can't migrate Markdown easily to another markup language (Asciidoc or reStructuredText), because Pandoc and other conversion tools won't support your flavor's extensions. (Why You Shouldn't Use "Markdown" for Documentation)

There's merit to the argument, for sure. You might be able to switch Markdown flavors using a tool such as Pandoc, or by converting the Markdown to HTML, and then converting the HTML to another Markdown flavor. However, switching tools will likely lead to a headache in updating the syntax in your content.

Here's an example. For many years, Jekyll used redcarpet and pygments to process Markdown and apply code syntax highlighting. However, to increase Windows support, Jekyll switched to kramdown and rouge at version 3.0. It was supposed to be a seamless backend switch that wouldn't require any adjustment to your existing Markdown. However, I found that kramdown imposed different requirements around spacing that broke a lot of my content, particularly with lists. I wrote about this issue here: Updating from redcarpet and Pygments to Kramdown and Rouge on Github Pages.

Despite the variations within Markdown, I'm reluctant to switch to a more semantic lightweight syntax because tool support for Markdown in general, following GitHub-flavored Markdown, is still much more widespread than support for reStructuredText or Asciidoc. kramdown is largely compatible with GitHub-flavored Markdown — it wouldn't be that difficult to migrate.

Additionally, developers tend to be familiar with Markdown but not reStructuredText or Asciidoc. If you want to encourage collaboration with developers, you might encounter more resistance by forcing them to write in reStructuredText or Asciidoc. Simplicity tends to win out in the end, and Markdown has clear momentum in the lightweight syntax arena. I imagine that in 10 years, reStructuredText and Asciidoc will be dwarfed in the same way that RAML (p. 924) and API Blueprint (p. 938) were dwarfed by the OpenAPI spec (p. 181).

Further, the OpenAPI spec lets you use CommonMark Markdown in `description` elements, which might make Markdown a better choice for API documentation. As long as you use the Markdown elements that are common across most flavors, migration (should it be required at some point) might not be as painful.

Overall, debates between Markdown, reStructuredText, and Asciidoc are pretty heated. You will find many for-and-against arguments for each lightweight syntax, as well as debates between XML and lightweight syntax.

One problem with lightweight syntax is its incompatibility with larger content management systems. Component content management systems (CCMSs) typically require more structured content such as DITA. The DITA committee recently approved Lightweight DITA, which will allow you to use GitHub-flavored Markdown and HTML in your DITA projects (assuming tool vendors support it). So we may see more convergence of XML tools and Markdown in the years to come. For more details about Lightweight DITA (LwDITA), see the interview with Carlos Evia on my blog. Carlos is co-chair of the OASIS committee for LwDITA.

Version control systems (such as Git)

Almost every IT shop uses some form of version control as they develop software code. Version control is how developers collaborate and manage their work. When you use docs-as-code tools, you'll probably also use version control such as Git. Because version control is such an essential element to learn, we'll dive more deeply into it here and in upcoming activities. In many ways, mastering Git is more challenging than learning a particular static site generator such as Jekyll or Hugo.

Plugging into version control

If you're working in API documentation, you'll most likely need to plug into your developer's version control system to get code. Or you may be creating branches and adding or editing documentation there.

Many developers are extremely familiar with version control, but typically these systems aren't used much by technical writers because technical writers have traditionally worked with binary file formats, such as Microsoft Word and Adobe Framemaker. Binary file formats are readable only by computers, and version control systems do a poor job in managing binary files because you can't easily see changes from one version to the next.

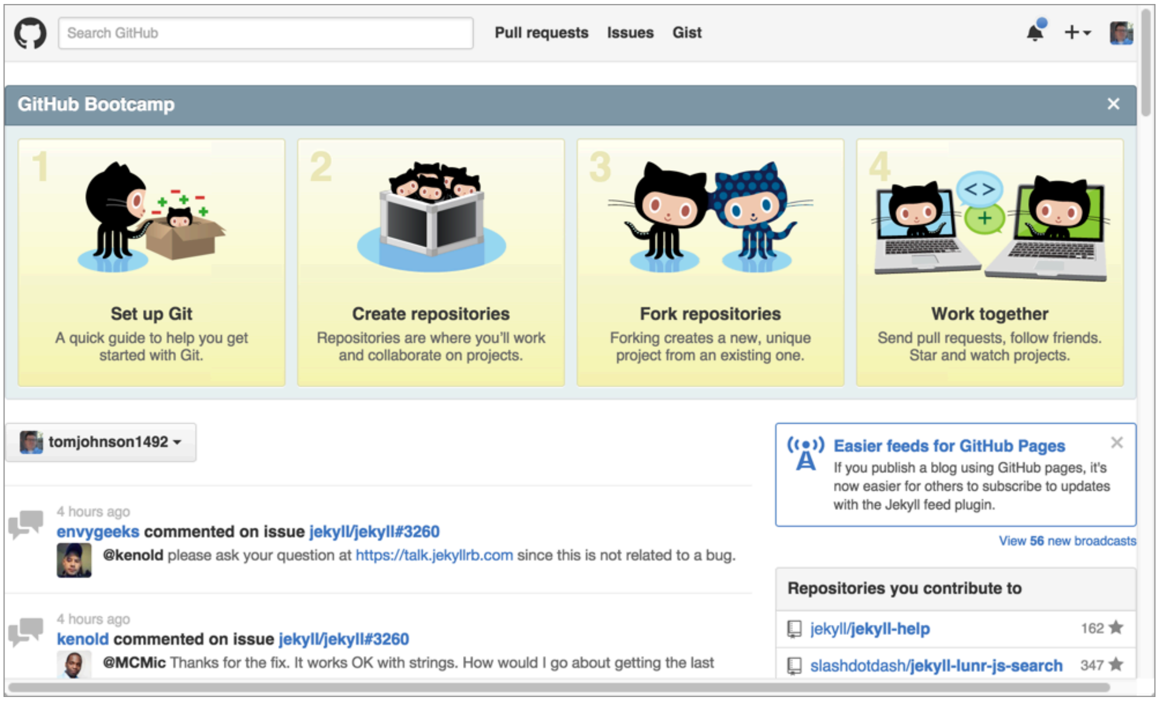
If you're working in a text file format, you can integrate your doc authoring and workflow into a version control system. When you do, a whole new world will open up.

Different types of version control systems

There are different types of version control systems. A *centralized* version control system requires everyone to check out or synchronize files with a central repository when editing them. This setup isn't so typical anymore because working with files on a central server tends to be slow.

More commonly, software shops use *distributed* version control systems. The most common system is Git (probably because GitHub provides Git repositories for free on the web) so we'll be focusing on it here. Other version control systems include Mercurial, Subversion (SVN), and Perforce. Due to Git's popularity, we'll focus on it exclusively here.

Note that Git is different from GitHub. GitHub provides online repositories and tools for Git. GitHub is a platform for managing Git projects, with a nice GUI for performing some Git tasks such as pull requests.



Set up Git
A quick guide to help you get started with Git.

Create repositories
Repositories are where you'll work and collaborate on projects.

Fork repositories
Forking creates a new, unique project from an existing one.

Work together
Send pull requests, follow friends. Star and watch projects.

4 hours ago
envygeeks commented on issue jekyll/jekyll#3260
@kenold please ask your question at https://talk.jekyllrb.com since this is not related to a bug.

4 hours ago
kenold commented on issue jekyll/jekyll#3260
@MCMic Thanks for the fix. It works OK with strings. How would I go about getting the last

Easier feeds for GitHub Pages
If you publish a blog using GitHub pages, it's now easier for others to subscribe to updates with the Jekyll feed plugin.

Github is a site that built tooling around Git.

Bitbucket is Altassian's version of GitHub. Bitbucket lets you use either Git or Mercurial, but most of the Bitbucket projects use Git. Bitbucket tends to only be used by companies already heavily using Atlassian.

The idea of version control

When you install version control software such as Git and initialize a repository in a folder, an invisible folder gets added to the repository. This invisible folder handles the versioning of the content in that folder. (If you want to move the Git tracking to another folder, you can move the invisible git folder to that other folder.)

When you add files to Git and commit them, Git takes a snapshot of the committed files at that point in time. When you commit another change, Git creates another snapshot. If you decide to revert to an earlier version of the file, you just revert to the particular snapshot. These snapshots are the basic idea behind versioning content.

Basic workflow with version control

There are many excellent tutorials on version control on the web, so I'll defer to those tutorials for more details (for example, see Peter Gruenbaum's Git and GitHub for Writers course on Udemy, with coupon code `IDRATHER` for a discount). In short, Git provides several stages for your files.

Here's the general workflow:

1. You must first add any files that you want Git to track. Just because the files are in the initialized Git repository doesn't mean that Git is actually tracking and versioning their changes. Only when you officially "add" files to your Git project does Git start tracking changes to that file.
2. Any modified files that Git is tracking are said to be in a "staging" area.
3. When you "commit" your files, Git creates a snapshot of the file at that point in time. You can always revert to this snapshot.

4. After you commit your changes, you can “push” your changes to the master. Once you push your changes to the master, your own working copy and the master branch are back in sync.

Branching

Git’s default repository is the `master` branch. When collaborating with others on the same project, usually people branch the master, make edits in the branch, and then merge the branch back into the master.

If you’re editing doc annotations in code files, you’ll probably follow this same workflow — making edits in a particular doc branch. When you’re done, you’ll create a pull request to have developers merge the doc branch back into the master.

Can you collaborate on doc projects in Git with other writers?

My discussion with Git here has been introductory and mostly directed as if you’ll be interfacing with developers for all your Git interactions. However, if you’re working in a docs-as-code environment, most likely your entire doc team (e.g., 5-10 writers) might be using a tool such as Jekyll, Hugo, Sphinx, or some other static site generator, and you’ll be collaborating in the same repo on docs using Git.

Collaborating on docs in Git can be a lot more challenging than it initially seems, especially if you’re working on the same files. Exactly how much you use Git when collaborating with docs, and in which scenarios tends to be somewhat fuzzy. For example, I recently received a question from a former colleague that hints at some of the challenges in collaborating with Git. My colleague writes,

> Hi Tom, do you have a recommended article for how a team of writers collaborate in git? We’ve made the big switch to our new docs pipeline and the biggest challenge is that we are unable to collaborate in git. A lot of changes and suggestion to technical documentation is hard to resolve in the git workflow. We are now talking about keeping the doc in Google docs and then porting to git after approval from the team and stakeholders. Does that make sense?

It seems that in their switch to Git, they tried to manage all aspects of content development and collaboration through Git. Note that even if you’re using Git, there is a good reason for initially developing content in some other platform such as Google Docs or Salesforce Quip. If you’re working together in a wiki-like way on the initial version of the content, making edits back and forth while the content is in a high degree of flux, then yes you need a more fluid and flexible platform that won’t involve playing tennis with Git with various commits and merges. I consider this wiki stage more of a sandbox platform for early content development, when you’re making radical edits in a highly collaborative way. If you try to do this purely in Git, you’ll find it stiff and awkward.

When working with developers on content, the developers often prefer to draft it on a wiki (“wiki” tends to be the universal name for any collaborative platform, such as Google Docs, Quip, Workdocs, or any other non-published web page platform, regardless of whether it’s actually a wiki). These platforms are much faster when you want to develop content with comments and replies, and lots of other back and forth. Developers really aren’t accustomed to managing documentation in Git anyway, and will find dumping and editing knowledge on a wiki the most common method.

But as soon as the content is finalized (for a first draft) and ready to be published as a web page with the rest of the documentation, you can migrate it from the wiki into your doc system. At this point in the process, the writer takes full control of the content. When you want to gather input from others to make changes, you might render the page as a PDF and put it into something like Workdocs for easy annotations (comments and replies and replies to comments, etc.).

As the review finishes and the content is merged in with the other sea of documentation topics, you'll probably make your edits directly in the repo, without always pushing the content out for review. At this point, your doc team will need to decide on the right Git workflow for your team.

Note that there are various workflows that teams might adopt (see 5 types of Git workflow that will help you deliver better code for some examples). Different teams adopt processes that fit their needs, which is what's great about Git. Git is flexible enough that it can be used in a dozen different ways all to achieve a similar end.

For example, some teams are strong proponents of following a "rebasing" workflow. Rebasing changes the way Git stores the history of commits by (fictionally) representing the commits as if in chronological order (rather than merged in from various timelines). This makes it easy to roll back catastrophes. (I'm not a fan of rebasing, though, as I think it introduces too much complexity and has some problematic scenarios.)

My personal approach is to leave content in the `master` branch as much as possible. I don't like branching every time I have an update. If I'm working on a new page, I'll set the page as unpublished. If I'm adding a new section, I might surround the new section with comment tags until ready to publish it.

Only if I'm developing an extensive edit to existing content do I create a branch for the new content. I store the content in the branch so that I can switch back to `master` and continue making other updates until I'm ready to go live with the branch content. Otherwise, if I start overhauling a topic (e.g., Topic A) in preparation for an upcoming release two weeks away, but then someone says I need to fix some typos in Topic B immediately, I wouldn't be able to push Topic B's edits live without also pushing the unfinished overhaul of Topic A live as well. To accommodate these scenarios, I store the edits to Topic A in a branch.

With most doc teams, we split the work into different stewardships. For example, John owns Project A, Sally owns Project B, and Kim owns Project C. It's rare that we're each making updates on content that the other owns. When everyone works on different files, merge conflicts rarely happen. In these scenarios of divided stewardship, it's fairly easy to collaborate on docs with your team in Git.

But suppose that you are in fact collaborating with others on the same topics. If John and Sally start making edits to the same topics in Project A on their local machines, as soon as they push their local versions to the remote, one of them (the later one) is bound to receive merge conflicts. Merge conflicts are Git's way of raising a red flag about the merge — you're asking Git to make contradictory merges. Two writers made conflicting updates to the same lines between commits, and now Git wants you to tell it which changes it should use. You might need to reach out to the other writer to see which updates should be committed.

When merge conflicts happen, Git removes the conflicted files from the staging area and inserts conflict markers (`<<<<<<< HEAD` and `>>>>>>> local commit msg`) in the problematic places in the files. In addition to the conflict markers, Atom editor will also highlight these blocks in visibly distinct ways (blue and purple).

If you run `git pull` and get a merge conflict, the message from Git looks like this:

```
CONFLICT (content): Merge conflict in somefile.md
Automatic merge failed; fix conflicts and then commit the result.
```

(If you want to get familiar with merge conflicts and scenarios, it's easy to simulate merge conflicts. Make edits in the GitHub browser while logged in as one user, and make different edits locally with Git configured as another user. Then push the local to origin and observe the conflicts.)

When you get a merge conflict, the conflicted files are removed from the merge process and taken out of the staging area. To see which files have conflicts, run the following:

```
git status
```

At this point you have two options for addressing the merge conflicts.

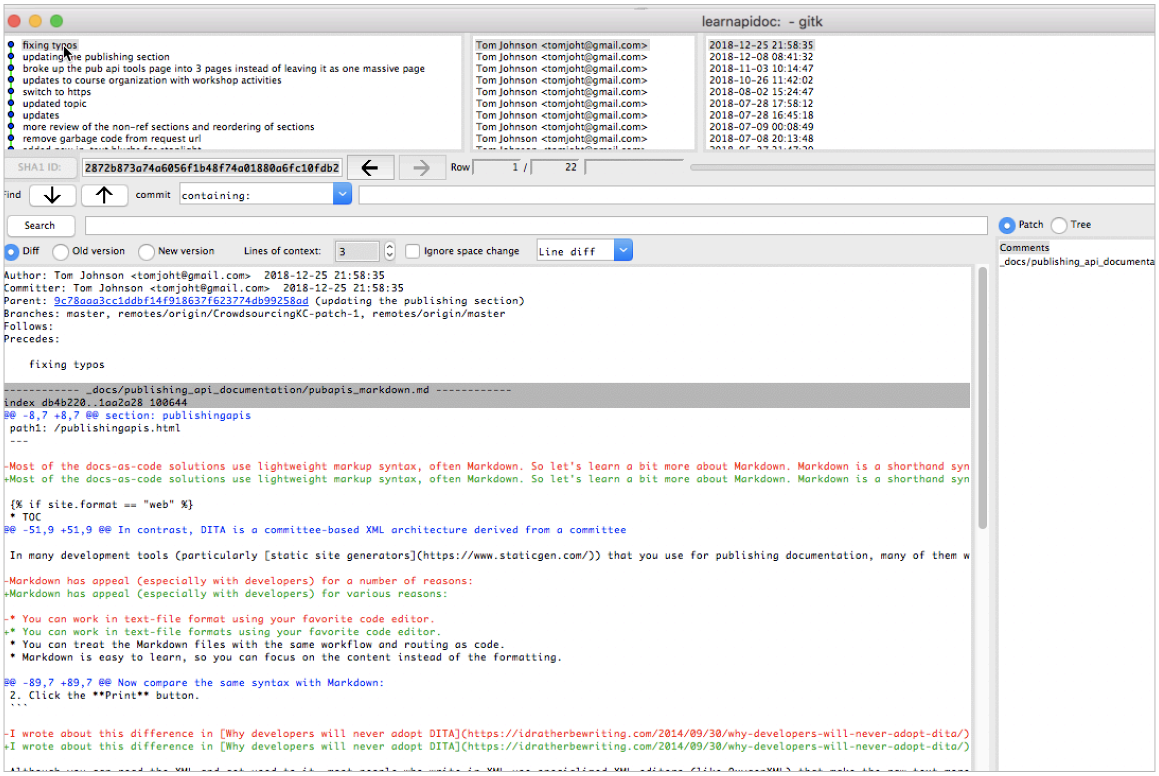
Option 1: Ask the responsible author to resolve the merge conflicts (recommended)

The first option is to identify the author responsible for the merge conflicts and ask him or her to fix them. Use this method if you didn't make changes to any of the problematic files and aren't sure why they are being flagged as having merge conflicts.

You can look at the change history of the file using a utility called gitk. After you run `git status` to identify the files with conflicts, right-click one of the files with conflicts in Atom's side menu and choose **Copy Project Path**. Then run a command such as this:

```
gitk some-path/myfile.md
```

In the gitk window that opens up, look to see who made the most recent changes to the file. Ping the author and ask them to address the conflicts and then push the updates out to the remote. Here's an example of the gitk window:



fixing typos	Tom Johnson <tomjoht@gmail.com>	2018-12-25 21:58:35
updating the publishing section	Tom Johnson <tomjoht@gmail.com>	2018-12-08 08:41:32
broke up the pub api tools page into 3 pages instead of leaving it as one massive page	Tom Johnson <tomjoht@gmail.com>	2018-11-03 10:14:47
updates to course organization with workshop activities	Tom Johnson <tomjoht@gmail.com>	2018-10-26 11:42:02
switch to https	Tom Johnson <tomjoht@gmail.com>	2018-08-02 15:24:47
updated topic	Tom Johnson <tomjoht@gmail.com>	2018-07-28 17:58:12
updates	Tom Johnson <tomjoht@gmail.com>	2018-07-28 16:45:18
more review of the non-ref sections and reordering of sections	Tom Johnson <tomjoht@gmail.com>	2018-07-09 00:08:49
remove garbage code from request url	Tom Johnson <tomjoht@gmail.com>	2018-07-08 20:13:48

SHA1 ID: 2872b873a74a6056f1b48f74a01880a6fc10fdb2

Row 1 / 22

Lines of context: 3

```
Author: Tom Johnson <tomjoht@gmail.com>  2018-12-25 21:58:35
Committer: Tom Johnson <tomjoht@gmail.com>  2018-12-25 21:58:35
Parent: 9c78aaa3cc1ddbf14f918637f623774db99258ad (updating the publishing section)
Branches: master, remotes/origin/CrowdsourcingKC-patch-1, remotes/origin/master
Follows:
Precedes:

    fixing typos

------------ _docs/publishing_api_documentation/pubapis_markdown.md ------------
index db4b220..1aa2a28 100644
@@ -8,7 +8,7 @@ section: publishingapis
 path1: /publishingapis.html
 ---

-Most of the docs-as-code solutions use lightweight markup syntax, often Markdown. So let's learn a bit more about Markdown. Markdown is a shorthand syn
+Most of the docs-as-code solutions use lightweight markup syntax, often Markdown. So let's learn a bit more about Markdown. Markdown is a shorthand syn

 {% if site.format == "web" %}
 * TOC
@@ -51,9 +51,9 @@ In contrast, DITA is a committee-based XML architecture derived from a committee

 In many development tools (particularly [static site generators](https://www.staticgen.com/)) that you use for publishing documentation, many of them w

-Markdown has appeal (especially with developers) for a number of reasons:
+Markdown has appeal (especially with developers) for various reasons:

-* You can work in text-file format using your favorite code editor.
+* You can work in text-file formats using your favorite code editor.
 * You can treat the Markdown files with the same workflow and routing as code.
 * Markdown is easy to learn, so you can focus on the content instead of the formatting.

@@ -89,7 +89,7 @@ Now compare the same syntax with Markdown:
 2. Click the **Print** button.
 ```

-I wrote about this difference in [Why developers will never adopt DITA](https://idratherbewriting.com/2014/09/30/why-developers-will-never-adopt-dita/)
+I wrote about this difference in [Why developers will never adopt DITA](https://idratherbewriting.com/2014/09/30/why-developers-will-never-adopt-dita/)
```

*Gitk example window. The commit history of the file is shown in the upper-left corner, along with the author*
```

for each commit. The main window shows the changes made for each commit (green is content added; red is content removed).

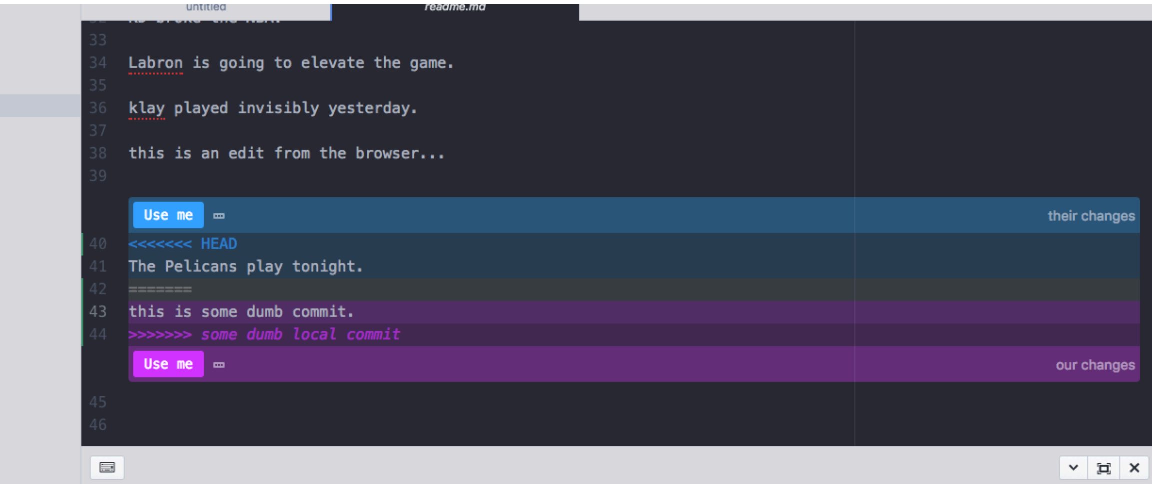
While waiting for the author to fix the merge conflict, you can abort your attempted merge with this command:

```
git merge --abort
```

This rolls back the merge. After the author fixes the merge conflict, push in your updates.

Option 2: Resolve the merge conflicts yourself

You can also resolve the merge conflict yourself. Choose this method if you're implicated in the file's history and might be the one responsible for the conflicts. Open up the conflicted files (cmd + T in Atom) that have the conflict and look for the blue and purple highlighted blocks:



```

Labron is going to elevate the game.

klay played invisibly yesterday.

this is an edit from the browser...

<<<<<<< HEAD
The Pelicans play tonight.
=======
this is some dumb commit.
>>>>>>> some dumb local commit


```

How Atom highlights conflicting commit messages

On the right, Atom notes the blue block that is "their changes" and the purple block that is "our changes." In this case, the text "The Pelicans play tonight" is from a commit that someone else made (it's the commit made via the GUI). It's a change we're pulling into our local version; hence, Atom flags it as "HEAD." The "HEAD" is the last commit on the branch that is currently checked out. In contrast, the purple "our changes" block is our local commit. It's the change we made in our commit.

If you own the file's content, simply select which "Use me" block you want to keep (probably "our changes"). If this isn't a file you own, choose the "Use me" that relates to "their changes" (the HEAD). You might also reach out to the author to confirm that you're selecting the right commit. (You can use gitk to easily see commits on the file.)

When merge conflicts occur, Git takes the files out of the staging index and requires you to fix the conflict and re-add the file to the staging area. Then you will need to commit the file:

```
### add specific files with conflicts that you resolved
git add somefile.html

### or add all files
git add .

### commit the change
git commit -m 'commit the change'

### push your changes
git push
```

For more information on resolving merge conflicts, see Resolving a merge conflict using the command line ↗.

Also, note that sometimes Git throws merge conflicts for no apparent reason. For example, change a file from title case to lower case, and you might throw Git into a bit of confusion. Other times you might see merge conflicts for reasons no one can figure out — maybe you didn't make any changes to the files that it's flagging as having conflicts. In Git scenarios, there are lots of actors making different commands in different environments. It can be hard to track down the reasons for conflicts. Just resolve them and move on.

For more info learning Git overall, see Learning Git ↗. See also this online course on Udemy: Git Learning Journey - Guide to Learn Git (Version Control) ↗, created by Anna Skoulikari, who you can also find on Write the Docs Slack ↗.

Activities

With that brief introduction to docs-as-code and version control, let's move on to two activities that use Git:

- Activity: Manage content in a GitHub wiki (p. 569)
- Activity: Use the GitHub Desktop client (p. 577)

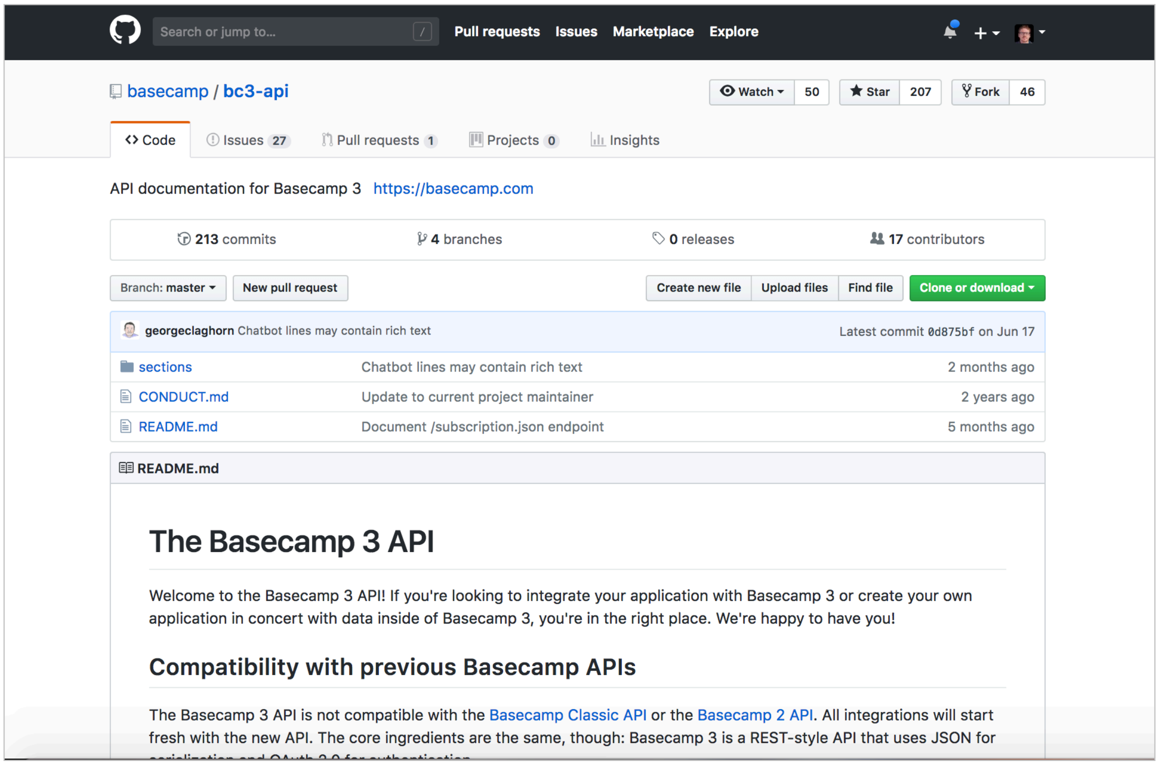
Activity: Manage content in a GitHub wiki

In this tutorial, we'll walk through a publishing workflow with one of the most common developer platforms: GitHub. When you create a repository on GitHub, the repository comes with a wiki that you can add pages to. This wiki can be convenient if your source code is stored on GitHub. Although GitHub might not be a platform where you publish your docs, understanding how to interact with it can be important for understanding version control (p. 562).

Learning GitHub will allow you to become familiar with the version control workflows that are common with many docs-as-code tools (p. 551). For this reason, I have a detailed tutorial for using GitHub in this course. Regardless of whether you use GitHub as a publishing tool, this tutorial will introduce you to Git workflows with content.

About GitHub Wikis

You could actually use the default GitHub wiki as your doc site. Here's an example of the Basecamp API, which is housed on GitHub.



API documentation for Basecamp 3 https://basecamp.com

georgeclaghorn Chatbot lines may contain rich text Latest commit 0d875bf on Jun 17

sections	Chatbot lines may contain rich text	2 months ago
CONDUCT.md	Update to current project maintainer	2 years ago
README.md	Document /subscription.json endpoint	5 months ago

README.md

The Basecamp 3 API

Welcome to the Basecamp 3 API! If you're looking to integrate your application with Basecamp 3 or create your own application in concert with data inside of Basecamp 3, you're in the right place. We're happy to have you!

Compatibility with previous Basecamp APIs

The Basecamp 3 API is not compatible with the Basecamp Classic API or the Basecamp 2 API. All integrations will start fresh with the new API. The core ingredients are the same, though: Basecamp 3 is a REST-style API that uses JSON for

Basecamp API

Unlike other wikis, the GitHub wiki you create is its own repository that you can clone and work on locally. (If you look at the "Clone this wiki locally" link, you'll see that it's a separate repo from your main code repository.) You can work on files locally and then commit them to the wiki repository when you're ready to publish. You can also arrange the wiki pages into a sidebar.

With GitHub, you write wiki pages in Markdown syntax. There's a special flavor of Markdown syntax for GitHub wikis called Github-flavored Markdown, or GFM. The GitHub Flavored Markdown allows you to create tables, add classes to code blocks (for proper syntax highlighting), and more.

Because you can work with the wiki files locally, you can leverage other tools (such as static site generators, or even DITA) to generate the Markdown files if desired. Working locally, you can handle all the re-use, conditional filtering, and other logic outside of the GitHub wiki. You can then output your content as Markdown files and commit them to your GitHub repository.

Use Git only to track text files. Don't start tracking large binary files, such as audio files, video files, Microsoft Word files, or Adobe PDF files. Version control systems really can't handle that kind of format well, and your repo size will increase exponentially. If you use Git to manage your documentation, exclude these files through your .gitignore file. You might also consider excluding images, as they bloat your repo size as well.

Limitations with GitHub wikis

GitHub wikis have some limitations:

- **Limited branding**. All GitHub wikis pretty much look the same.
- **Open access on the web**. If your docs need to be private, GitHub probably isn't the place to store them (private repos, however, are an option).
- **No structure**. The GitHub wiki pages give you a blank page and allow you to add sections. You won't be able to do any advanced styling or more attractive-looking interactive features.

I'm specifically talking about the built-in wiki feature with GitHub, not GitHub Pages. You can use tools such as Jekyll to brand and auto-build your content with whatever look and feel you want. I explore GitHub Pages with more depth in the tutorial on Jekyll (p. 637).

Install Git

Before you start working with GitHub, you need to set up Git and install any necessary tools and credentials to work with GitHub (especially if you're on Windows).

Mac:

To install Git on a Mac, see Installing on Mac. Once installed, you can use Git in several ways:

- Open the default Terminal application by doing to **Applications > Utilities > Terminal**.
- Install iTerm, a separate terminal app.
- Use PlatformIO IDE Terminal in Atom (this is my preferred method when working with projects).

Windows:

On Windows, install Git using the installer here: Git for Windows.

This installer includes a Git BASH terminal emulator that will allow you to use Git and Unix commands from the terminal.

You can check to see if you have Git already installed by opening a terminal and typing the following:

```
git --version
```

Set up automatic GitHub authentication

You can configure Git so that when you push changes to GitHub, you won't need to type your username and password each time. See the following topics to set this up:

- Set up Git.
- Generating a new SSH key and adding it to the ssh-agent
- Adding a new SSH key to your GitHub account
- Associating text editors with Git

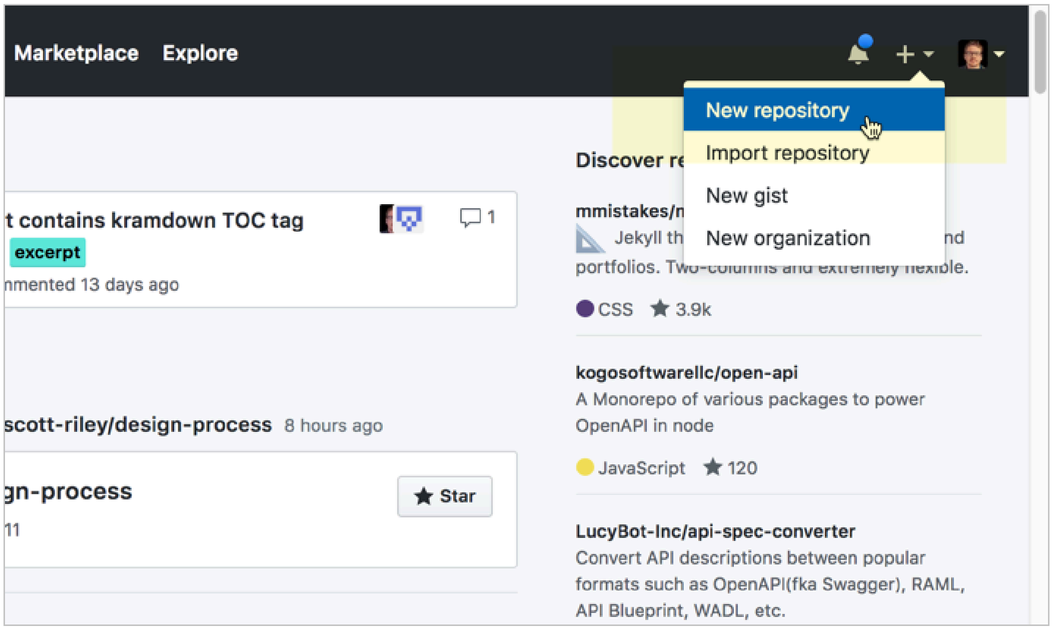
After you make these configurations, close and re-open your terminal.

GitHub and Git are not the same. Git provides distributed version control (p. 562). GitHub is a platform that helps you manage Git projects. GitHub also provides a GUI interface that allows you to execute a lot of Git commands, such as pull requests.

Activity: Create a GitHub wiki and publish content on a sample page

In this section, you will create a new GitHub repo and publish a sample file there.

1. Create a GitHub account at GitHub.com.
2. Go to GitHub and sign in. After you're signed in, click the **+** button in the upper-right corner and select **New repository**.



portfolios. Two-columns and extremely flexible.

A Monorepo of various packages to power OpenAPI in node

Convert API descriptions between popular formats such as OpenAPI(fka Swagger), RAML, API Blueprint, WADL, etc.

Creating a new GitHub repository

3. Give the repo a **Repository name**, a short **Description**, select **Public**, select **Initialize the repo with a README**, and then click **Create repository**. (Don't worry about selecting the license or gitignore settings for this activity.)
4. Click the **Wiki** tab on the top navigation bar of your new repository.
5. Click **Create the first page**.
6. In the default page ("Home"), insert your own sample documentation content, preferably using Markdown syntax. Or grab the sample Markdown page of a fake endpoint called surfreport here

and insert it into the page.

7. In the **Edit message** box, type a description of what you updated (your commit message).
8. Click **Save Page**.

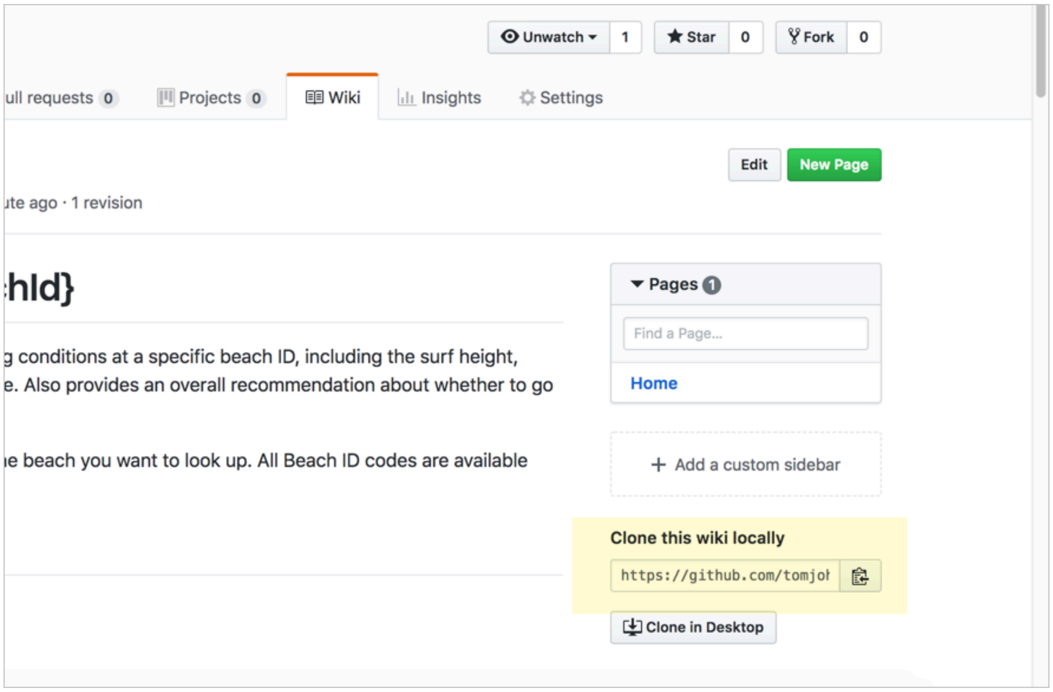
Notice how GitHub automatically converts the Markdown syntax into HTML and styles it in a readable way. You could work with this GitHub wiki entirely in the browser as a way for multiple people to collaborate and edit content. However, unlike other wikis, with GitHub you can also take all the content offline and edit locally, and then commit your changes and push the changes back online.

Activity: Clone your GitHub repo locally

So far you've been working with GitHub in the browser. Now we'll take the same content and work with it locally. This is what makes the GitHub wiki unique from other wikis — it's a Git repo, so you can manipulate the content the same way as any other Git repo (working locally, pushing, pulling, merging, branching, etc.).

To clone the GitHub repo locally:

1. If you don't already have Git installed, set it up on your computer. (You can check whether Git is installed by typing `git --version` in your terminal or command prompt. See Install Git (p. 570) for more information on installation.)
2. While viewing your the GitHub wiki in your browser, look for the section that says **Clone this wiki locally** (highlighted below). Click the clipboard button. (This copies the clone URL to your clipboard.)



ıte ago · 1 revision

:hId}

g conditions at a specific beach ID, including the surf height,
e. Also provides an overall recommendation about whether to go

ıe beach you want to look up. All Beach ID codes are available

Clone this wiki locally

`https://github.com/tomjoh`

Clone this wiki locally

The wiki is a separate clone URL than the project's repository. Make sure you're viewing your wiki and not your project. The clone URL will include `.wiki`.

In contrast to the “Clone this wiki locally” section, the “Clone in Desktop” button launches the GitHub Desktop client and allows you to manage the repository and your modified files, commits, pushes, and pull through the GitHub Desktop client. If you’re interested in using the GitHub Client of the command line, see this other activity: Activity: Use the GitHub Desktop client (p. 577).

3. Open your terminal emulator:
    - If you’re a Windows user, open the **Git BASH** terminal emulator, which was installed when you installed Git.
    - If you’re a Mac user, go to **Applications > Utilities > Terminal** (or launch iTerm, if you installed it instead).
4. In your terminal, either use the default directory or browse (`cd`) to a directory where you want to download your repository.
5. Type the following, but replace the git URL with your own git URL that you copied earlier (it should be on your clipboard). The command should look something like this:

```
git clone https://github.com/tomjoht/weatherapi.wiki.git
```

When you clone a repo, Git will show something like the following:

```
Cloning into 'weatherapi.wiki'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 9 (delta 0), reused 0 (delta 0), pack-reused 6
Unpacking objects: 100% (9/9), done.
```

The folder Git creates in the above example is called `weatherapi.wiki`.

Cloning the wiki gives you a copy of the content on your local machine. Git is distributed version control software, so everyone has their own copy. When you clone the repo, you create a copy on your local machine; the version in the cloud on GitHub is referred to as “origin.” Thus, you have two instances of the content.

More than just copying the files, though, when you clone a repo, you initialize Git in the folder where you clone the repo. Initializing Git means Git will create an invisible Git folder in that directory, and Git can start tracking your edits to the files, providing version control. With Git initialized, you can run `pull` commands to get updates from the online repository (origin) pulled down to your local copy. You can also `commit` your changes and then `push` your changes back up to origin.

6. Navigate to the directory where you cloned the repo (either using standard ways of browsing for files on your computer or via the terminal with `cd`) to see the files you downloaded. For example, type `cd weatherapi.wiki` and then `ls` (Mac) or `dir` (Windows) to see the files.

You don’t need to type the full directory name. Just start typing the first few letters and then press your **Tab** key to autocomplete the rest.

You might also want to browse to this folder via Finder (Mac) or Explorer (Windows). The folder also continas an invisible folder called `.git`. For instructions on making hidden files visible, see one of the following: Windows or Mac).

Activity: Push local changes to the remote

1. In a text editor, open the Markdown file you downloaded in the GitHub repository.
2. Make a small change to the content and save it. For example, type your name below the page title.
3. In your terminal, make sure you're in the directory where you downloaded the GitHub project.

   To look at the directories under your current path, type `ls` (Mac) or `dir` (Windows). Then use `cd {directory name}` to drill into the folder, or `cd ../` to move up a level.

4. See what files have changed:

   ```
   git status
   ```

   Git shows the files it's tracking but which haven't been added to the commit's staging area:

   ```
   Changes not staged for commit:
   (use "git add <file>..." to update what will be committed)
   (use "git checkout -- <file>..." to discard changes in working direct
   ory)
   modified:   Home.md
   ```

5. Type `git add .` to add all the files to your staging area. The staging area contains all files that you want added to your next commit:

   ```
     git add .
   ```

   Git doesn't automatically track all files in the same folder where Git has been initialized. Git tracks modifications only for the files that have been "added" to Git. By typing `git add .` or `git add --all`, you're telling Git to start tracking modifications to all files in this directory. You could also type a specific file name here instead, such as `git add Home.md`, to just add a specific file (rather than all files changed) to Git's tracking.

   After you run the `git add` command, Git adds the files into what's called the staging area. These files will be committed when you run `git commit`.

6. See the changes set in your staging area:

   ```
     git status
   ```

   Git responds with a message indicating which files are on-deck to be committed.

   ```
   Changes to be committed:
   (use "git reset HEAD <file>..." to unstage)

       modified:   Home.md
   ```

   The staging area lists all the files that have been added to Git that you have modified in some way. It's a good practice to always type `git status` before committing files because you might realize that you accidentally added some files you didn't intend to track (such as large binary files). If you want to remove this file from the staging area, you can type `git reset HEAD Home.md` to unstage it.

7. Commit the changes:

   ```
   git commit -m "updated some content"
   ```

   When you commit the changes, you're creating a snapshot of the files at a specific point in time for versioning.

   The `git commit -m` command is a shortcut for committing and typing a commit message in the same step. It's much easier to commit updates this way.

   If you type `git commit` only, you'll be prompted with another window to describe the change. On Windows, this new window will probably be a Notepad window. Describe the change on the top line, and then save and close the Windows file.

   On a Mac, a new window doesn't open. Instead, the Vim editor mode opens up within the terminal. ("vi" stands for visual and "m" for mode, but it's not a very visual editor.) I don't recommend using Vim. If you get stuck in this mode and need to escape, press your **Escape** key. Then type **q** to quit. (See Vim commands here.) Normally, you want an external editor such as Sublime Text to open from your terminal. See Associating text editors with Git for details.

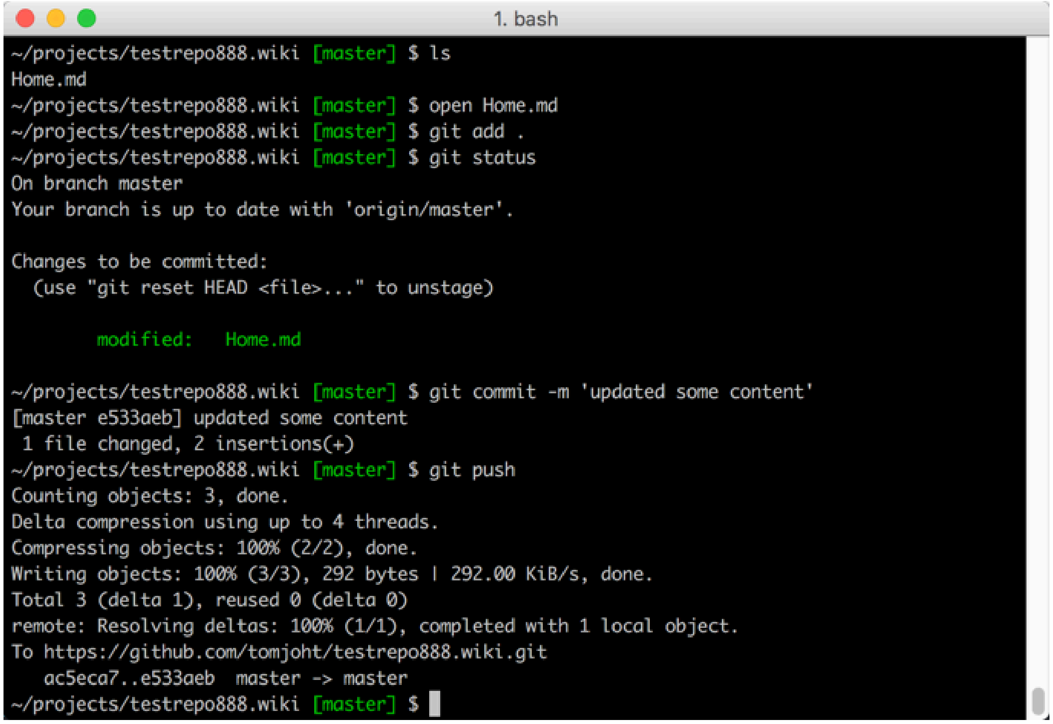
8. Push the changes to your repository:

   ```
   git push
   ```

   Unless you set up automatic GitHub authentication (p. 571), you will be prompted for your GitHub username and password. (Note that your username is your GitHub login ID, such as "jdoe," not your friendly name, such as "John Doe.")

   When you type `git push` or `git pull` and don't specify the branch, GitHub uses the default branch from origin. The default branch on GitHub is called `main`. Thus the command actually passed is `git push origin main` (which means "push these changes to the remote origin[al] repository, in the `main` branch"). Some developers prefer to specify the repository and branch to ensure they are interacting with the right repositories and branches.

   Your terminal window probably looks something like this:



```
~/projects/testrepo888.wiki [master] $ ls
Home.md
~/projects/testrepo888.wiki [master] $ open Home.md
~/projects/testrepo888.wiki [master] $ git add .
~/projects/testrepo888.wiki [master] $ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Home.md

~/projects/testrepo888.wiki [master] $ git commit -m 'updated some content'
[master e533aeb] updated some content
 1 file changed, 2 insertions(+)
~/projects/testrepo888.wiki [master] $ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 292 bytes | 292.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/tomjoht/testrepo888.wiki.git
   ac5eca7..e533aeb  master -> master
~/projects/testrepo888.wiki [master] $ 
```

Terminal window with git commands

9. Now verify that your changes took effect. Browse to your GitHub wiki repository, refresh the page, and look to see the changes.

Although there are many options for authoring and publishing tools with developer docs (see staticgen.com for common docs-as-code tools), at the core of these tools is usually a Git workflow. The Git workflow can be more powerful and complex than any authoring tool. Interacting with Git might also be key for interacting with engineering repos to make edits to documentation that appears inside code.

Avoiding merge conflicts when editing online and locally

The visual editor on GitHub.com might be an easy way for subject matter experts to contribute, whereas tech writers will probably want to clone the repo and work locally. If some people make edits in the browser while others edit locally, you might encounter merge conflicts. To avoid merge conflicts, always run `git pull` before running `git push`. If two people edit the same content simultaneously between commits, you will likely need to resolve merge conflicts.

Activity: Use the GitHub Desktop client

Although most developers use the command line when working with version control systems, there are many GUI clients available that can potentially simplify the process. GUI clients might be especially helpful when you're trying to see what has changed in a file since the GUI can quickly highlight and indicate the changes taking place.

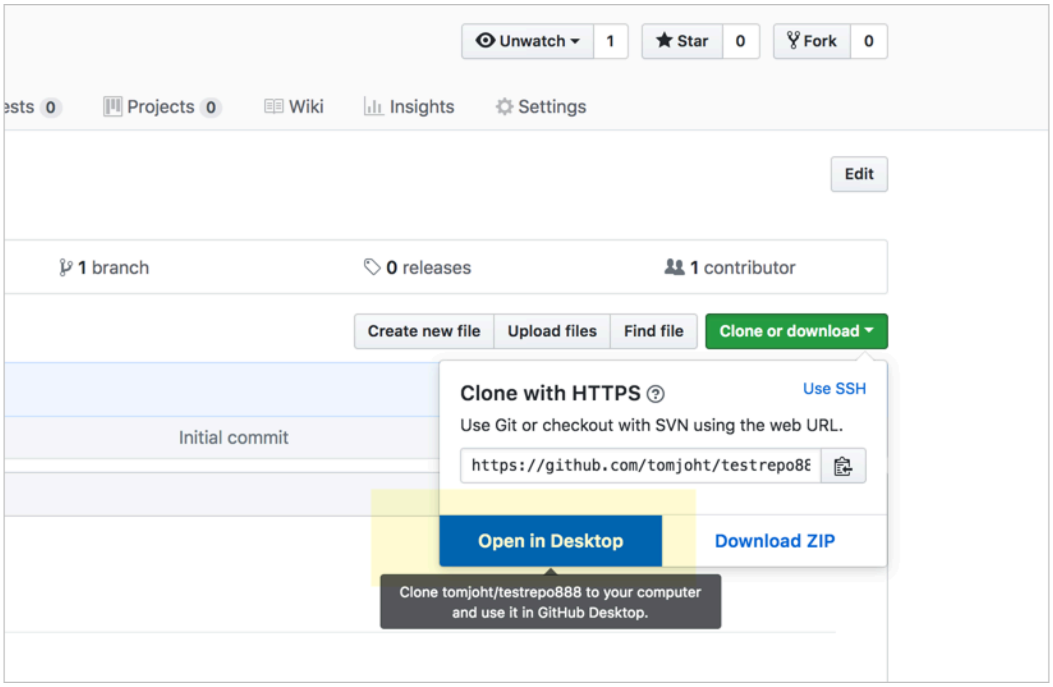
Follow a typical workflow with a GitHub project using GitHub Desktop

In this tutorial, you'll use GitHub Desktop to manage the Git workflow.

Rather than working in a GitHub wiki (as you did in the previous GitHub tutorial (p. 569)), you'll work in a regular Git repository. This is because GitHub wikis have some limitations when it comes to making pull requests.

To set up your Git repo using the GitHub Desktop client:

1. First, download and install GitHub Desktop. Launch the app and sign in. (You should already have a GitHub account from previous tutorials (p. 569), but if not, create one.)
2. Go to Github.com and browse to the repository you created in the GitHub tutorial (p. 569), but not the wiki. Just go to the main repo. (If you didn't do the previous activity, just create a new repository.)
3. While viewing your GitHub repo in the browser, click **Clone or download** and select **Open in Desktop**.



Use Git or checkout with SVN using the web URL.

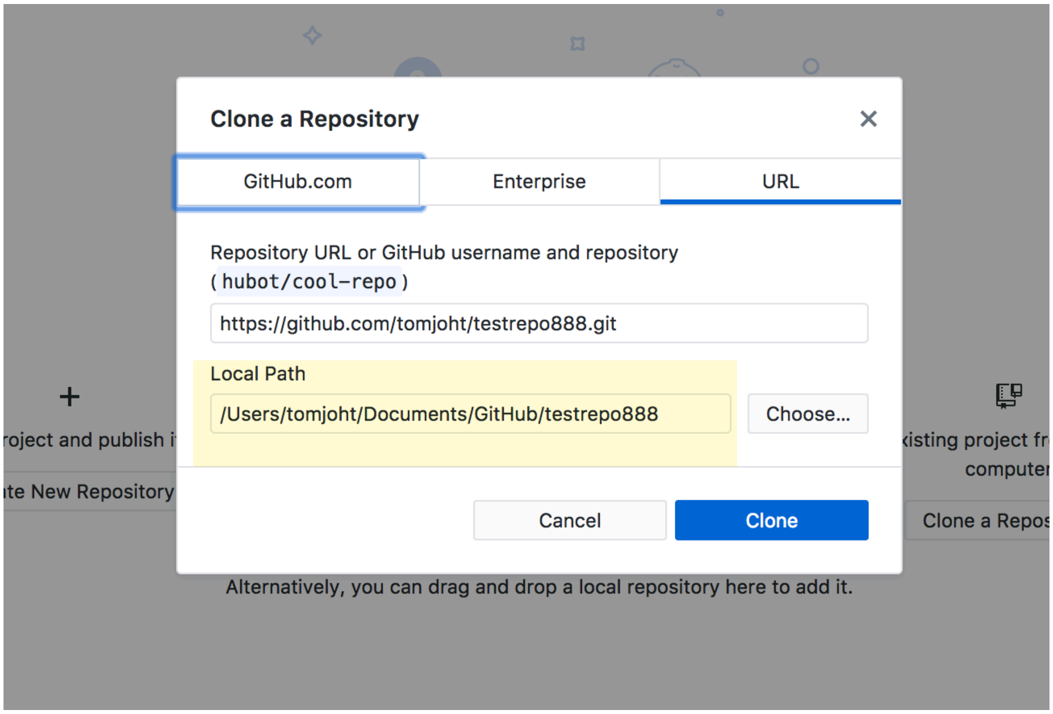
`https://github.com/tomjoht/testrepo8E`

Clone tomjoht/testrepo888 to your computer and use it in GitHub Desktop.

Open in GitHub Desktop

4. Open GitHub Desktop client and go to **File > Clone Repository**.

5. In the confirmation dialog, select **Open GitHub Desktop.app**. GitHub Desktop should launch with a “Clone a Repository” dialog box about where to clone the repository. If desired, you can change the Local Path.
6. Click the **URL** tab, and then paste in the clone URL. In the **Local Path** field, select where you want the repo cloned. For example:



Repository URL or GitHub username and repository (`hubot/cool-repo`)

https://github.com/tomjoht/testrepo888.git

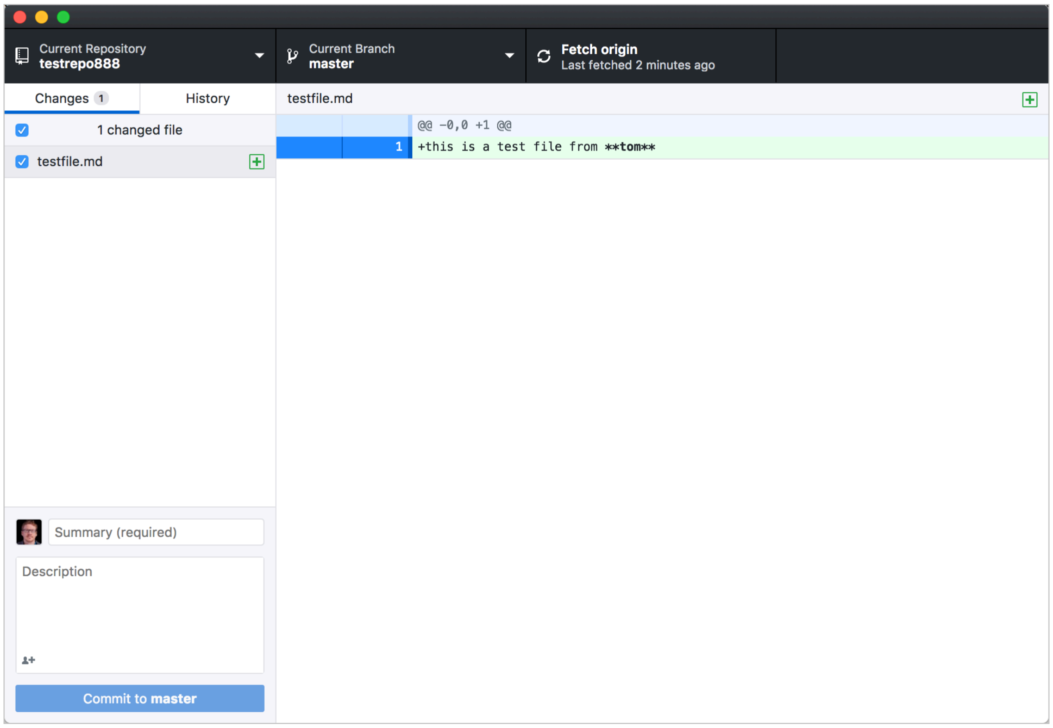
Local Path

/Users/tomjoht/Documents/GitHub/testrepo888

Alternatively, you can drag and drop a local repository here to add it.

Selecting paths for the repo in GitHub Desktop

7. Click **Clone**.
8. Go into the repository where GitHub Desktop cloned the repo (use your Finder or browse the folders with Finder or Explorer) and either add a simple text file with some content or make a change to an existing file.
9. Go back to GitHub Desktop. You’ll see the new file you added in the list of uncommitted changes on the left.



```
@@ -0,0 +1 @@
1 +this is a test file from **tom**
```

Uncommitted changes shown

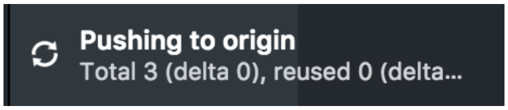
In the list of changed files, the green + means you've added a new file. A yellow circle means you've modified an existing file.

10. In the lower-left corner of the GitHub Desktop client (where it says "Summary" and "Description"), type a commit message, and then click **Commit to master**.

    When you commit the changes, the left pane no longer shows the list of uncommitted changes. However, you've committed the changes only locally. You still have to push the commit to the remote (origin) repository. ("origin" is the alias name that refers to the remote repository.)

11. Click **Push origin** at the top.

    You'll see GitHub Desktop show that it's "Pushing to origin."



```
Total 3 (delta 0), reused 0 (delta...
```

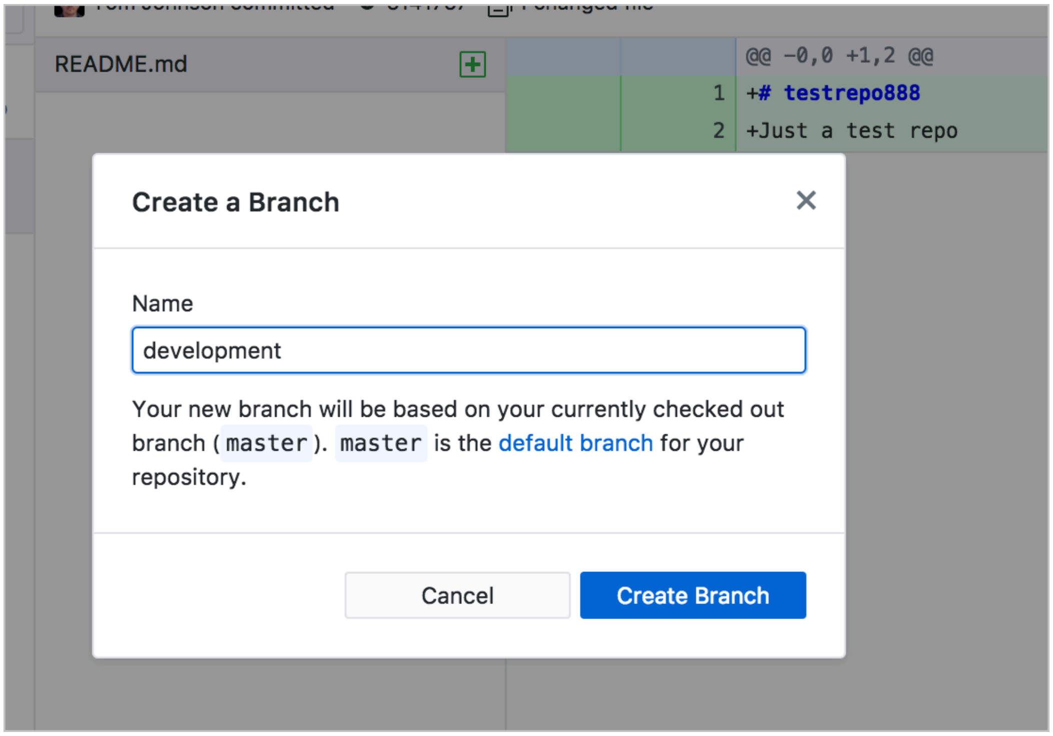
If you view your repository online, you'll see that the change you made has been pushed to the master branch on origin. You can also click the **History** tab in the GitHub Desktop client (instead of the **Changes** tab), or go to **View > Show History** to see the changes you previously committed.

Although I prefer to use the terminal instead of the GitHub Desktop GUI, the GUI gives you an easier visual experience to see the changes made to a repository. You can use both the command line and Desktop client in combination, if you want.

Create a branch

Now let's create a branch, make some changes, and see how the changes are specific to that branch.

1. In the GitHub Desktop client, go to **Branch > New Branch** and create a new branch. Call it "development" branch, and click **Create Branch**.

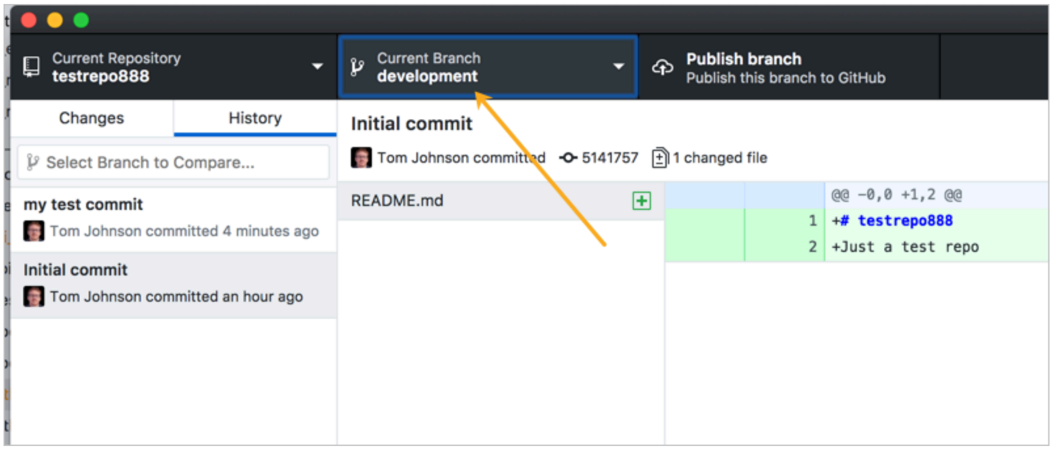


Name: development

Your new branch will be based on your currently checked out branch (`master`). `master` is the default branch for your repository.

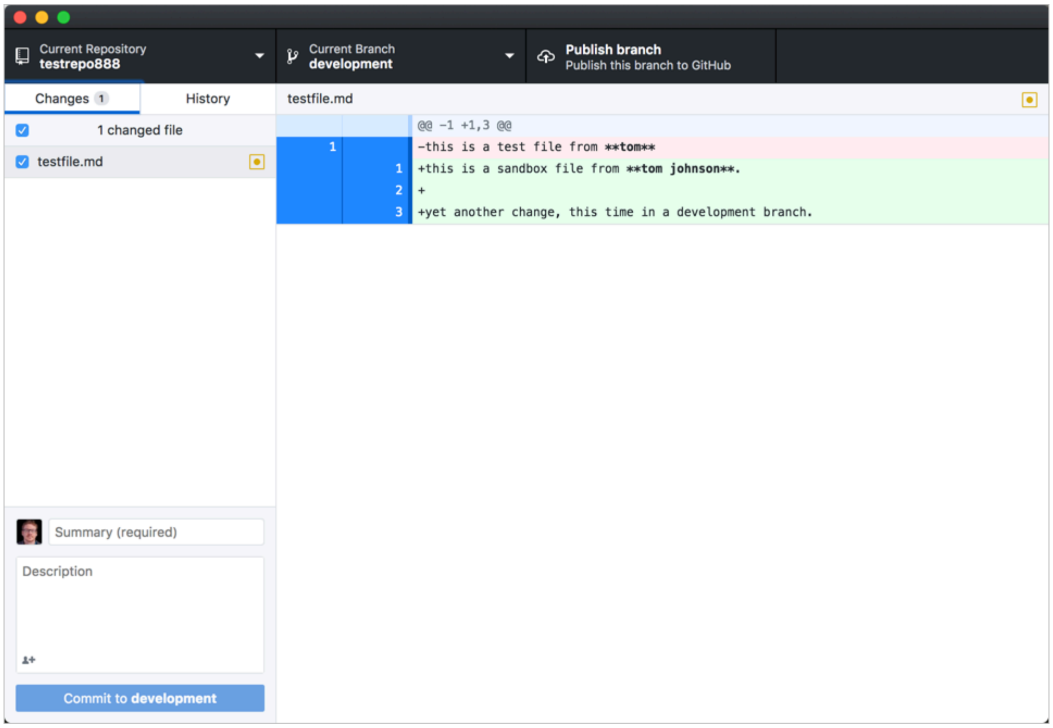
Creating a new branch

When you create the branch, you'll see the Current branch drop-down menu indicate that you're working in that branch. Creating a branch copies the existing content (from the branch you're currently in, *master*) into the new branch (*development*).



Working in a branch

2. Using Finder or Explorer, browse to the file you created earlier and make a change to it, such as adding a new line with some text. Save the changes.
3. Return to GitHub Desktop and notice that on the Changes tab, you have new modified files.



```
@@ -1 +1,3 @@
-this is a test file from **tom**
+this is a sandbox file from **tom johnson**.
+
+yet another change, this time in a development branch.
```

New files modified

The file changes show deleted lines in red and new lines in green. The colors help you see what changed.

4. Commit the changes using the options in the lower-left corner, and click **Commit to development**.
5. Click **Publish branch** (on the top of the GitHub Desktop window) to make the local branch also available on origin (GitHub). (Remember, there are usually two versions of a branch — the local version and the remote version.)
6. Switch back to your master branch (using the Branch drop-down option at the top of the GitHub Desktop client). Then look at your file (in your text editor, such as Sublime text). Note how the file changes you made while editing in the development branch don't appear in your master branch.

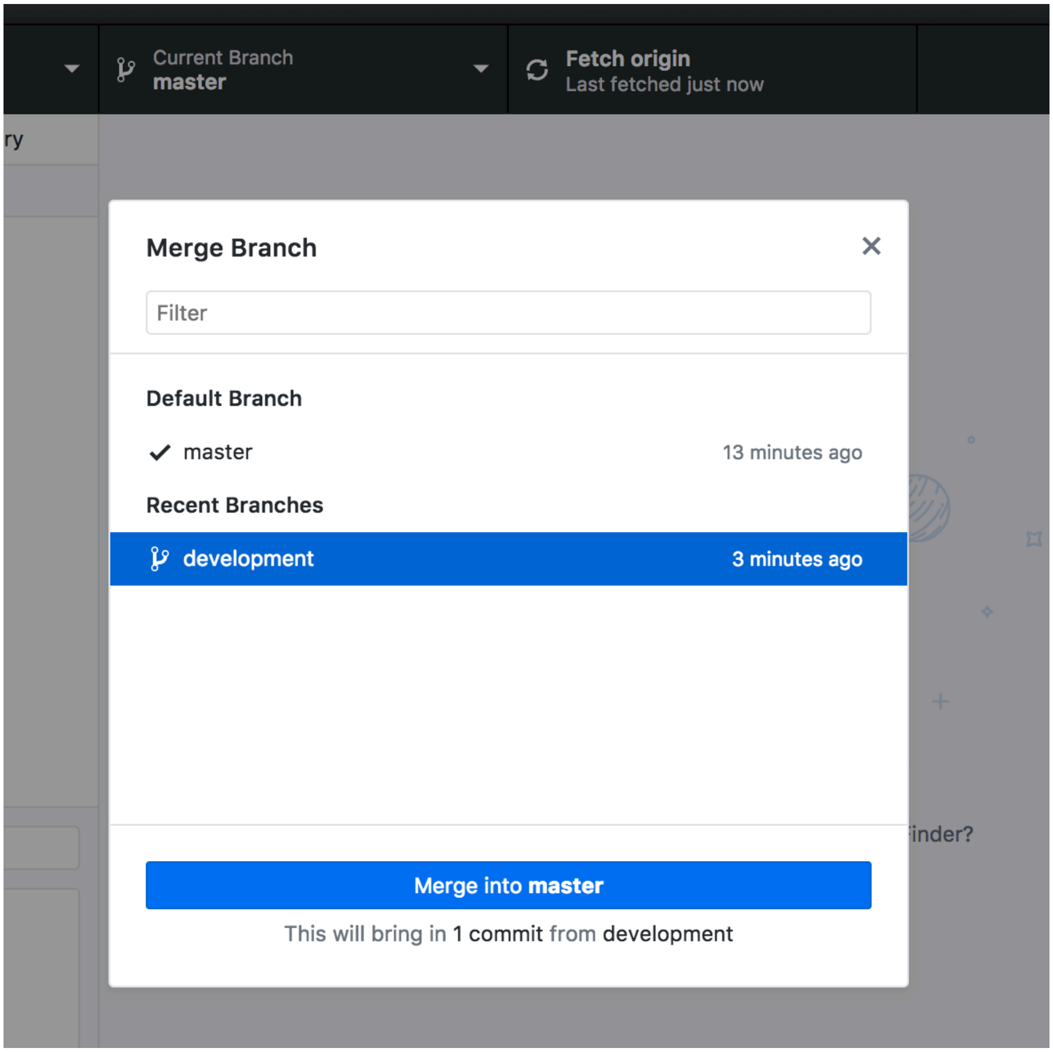
You usually create a new branch when you're making extensive changes to your content. For example, suppose you want to revamp a section ("Section X") in your docs. However, you might want to publish other updates before publishing the extensive changes in Section X. If you were working in the same branch, it would be difficult to selectively push updates on a few files outside of Section X without pushing updates you've made to files in Section X as well.

Through branching, you can confine your changes to a specific version that you don't push live until you're ready to merge the changes into your master branch.

Merge the development branch into master

Now let's merge the development branch into the master branch.

1. In the GitHub Desktop client, switch to the branch you want to merge the development branch into. From the branch selector, select the **master** branch.
2. Go to **Branch > Merge into Current Branch**.
3. In the merge window, select the **development** branch, and then click **Merge development into master**.



This will bring in 1 commit from development

Merging into master

If you look at your changed file, you should see the changes in the master branch.

4. Then click **Push origin** to push the changes to origin.

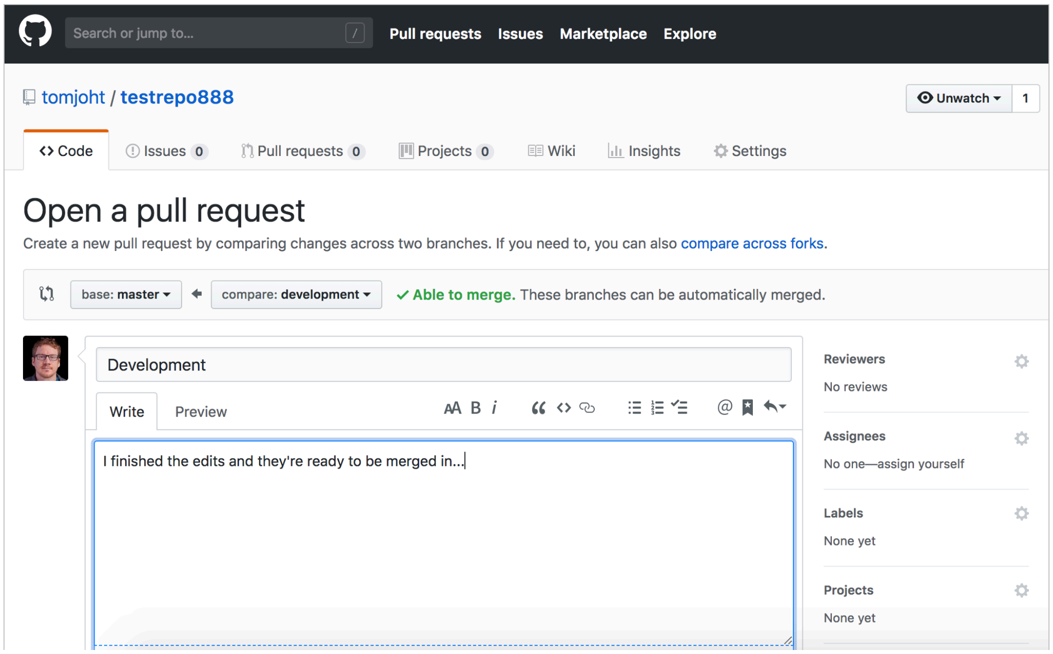
You will now see the changes reflected on the file on GitHub.

Merge the branch through a pull request

Now let's merge the development branch into the master using a pull request workflow. We'll pretend that we've cloned a repo managed by engineers, and we want to have the engineers merge in the development branch. (In other words, we might not have rights to merge branches into the master.) To do this, we'll create a pull request.

1. Just as you did in the previous section, switch to the development branch, make some updates to the content in a file, and then save and commit the changes. After committing the changes, click **Push origin** to push your changes to the remote version of the development branch on GitHub.
2. In the GitHub Desktop client, while you have **development** branch selected, go to **Branch > Create Pull Request**.

   GitHub opens in the browser with the Pull Request form opened.



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

✓ Able to merge. These branches can be automatically merged.

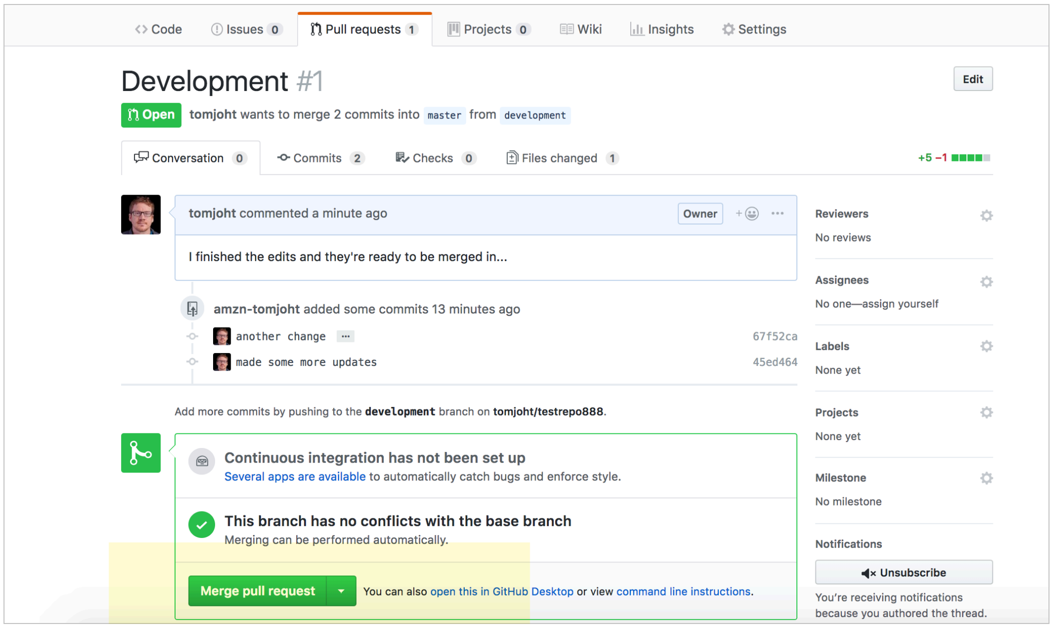
Development

I finished the edits and they're ready to be merged in...

Pull request

The left-facing arrow (pointing from the development branch towards the master) indicates that the pull request ("PR") wants to merge the development branch into the master branch.

3. Describe the pull request, and then click **Create pull request**.
4. At this point, engineers would get an email request asking for them to merge in the edits. Play the part of the engineer by going to the **Pull requests** tab (on GitHub) to examine and confirm the merge request. As long as the merge request doesn't pose any conflicts, you'll see a **Merge pull request** button.



tomjoht wants to merge 2 commits into `master` from `development`

tomjoht commented a minute ago

I finished the edits and they're ready to be merged in...

amzn-tomjoht added some commits 13 minutes ago

```
another change        67f52ca
made some more updates 45ed464
```

Add more commits by pushing to the **development** branch on **tomjoht/testrepo888**.

Continuous integration has not been set up

Several apps are available to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch

Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

Reviewers: No reviews

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet

Milestone: No milestone

Notifications: You're receiving notifications because you authored the thread.

Confirm merge request

5. To see what changes you're merging into master, you can click the **Files changed** tab (which appears on the secondary navigation bar near the top). Then click **Merge pull request** to merge in the branch, and click **Confirm merge** to complete the merge.
6. Now let's get the updates you merged into the `master` branch online into your local copy. In your GitHub Desktop GUI client, select the **master** branch, and then click the **Fetch origin** button. Fetch gets the latest updates from origin but doesn't update your local working copy with the changes.

   After you click **Fetch origin**, the button changes to **Pull Origin**.
7. Click **Pull Origin** to update your local working copy with the fetched updates.

   Now check your files and notice that the updates that were originally in the development branch now appear in master.

For a more detailed tutorial in making pull requests using the GitHub interface, see Pull request workflows through GitHub (p. 586). I include an extensive tutorial with pull requests because it will likely be a common workflow if you are contributing to an open-source project (p. 686).

Managing merge conflicts

Suppose you make a change on your local copy of a file in the repository, and someone else changes the same file using the online GitHub.com browser interface. The changes conflict with each other. What happens?

When you click **Push origin** from the GitHub Desktop client, you'll see a message saying that the repository has been updated since you last pulled:

> "The repository has been updated since you last pulled. Try pulling before pushing."

The button that previously said “Push origin” now says “Pull origin.” Click **Pull origin**. You now get another error message that says,

> “We found some conflicts while trying to merge. Please resolve the conflicts and commit the changes.”

If you decide to commit your changes, you’ll see a message that says:

> “Please resolve all conflicted files, commit, and then try syncing again.”

The GitHub Desktop client displays an exclamation mark next to files with merge conflicts. Open up a conflict file and look for the conflict markers (`<<<<<<<` and `>>>>>>>`). For example, you might see this:

```
<<<<<<< HEAD
This is an edit I made locally.
=======
This is an edit I made from the browser.
>>>>>>> c163ead57f6793678dd41b5efeef372d9bd81035
```

(From the command line, you can also run `git status` to see which files have conflicts.) The content in `HEAD` shows your local changes. The content below the `=======` shows changes made by elsewhere.

Fix all the conflicts by adjusting the content between the content markers and then deleting the content markers. For example, update the content to this:

```
This is an edit I made locally.
```

Now you need to re-add the file to Git again. In the GitHub Desktop client, commit your changes to the updated files. Then click **Push origin.** The updates on your local get pushed to the remote without any more conflicts.

Conclusion

The more you use GitHub, the more familiar you’ll become with the workflows you need. Git is a robust, powerful collaboration platform, and there are many commands and workflows and features that you could adopt for a variety of scenarios. Despite Git’s variety of commands and workflows, most likely the scenarios you’ll actually use are somewhat limited in scope and learnable without too much effort. Pretty soon, these workflows will become automatic.

Although we’ve been using the GitHub Desktop client for this exercise, you can do all of this through the command line, and you’ll probably find it preferable that way. However, the GitHub Desktop client can be a good starting point as you transition into becoming more of a Git power user.

Pull request workflows through GitHub

In the previous step, Activity: Use the GitHub Desktop Client (p. 577), you used Github Desktop to manage the workflow of committing files, branching, and merging. In this tutorial, you'll do a similar activity but using the browser-based interface that Github provides rather than using a terminal or Github Desktop.

Understanding the pull request workflow is essential for reviewing changes in a collaborative project, such as an open-source project with many contributors. Using GitHub's interface is also handy if you have non-technical reviewers.

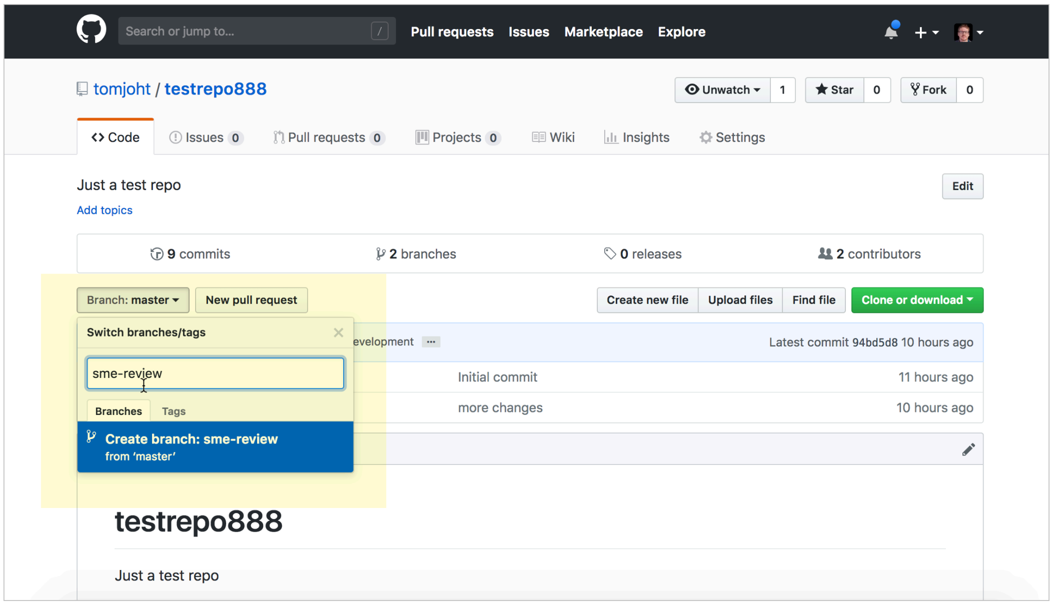
Make edits in a separate branch

By default, your new repository has one branch called "Master." Usually, when you're making changes or reviews/edits, you create a new branch and make all the changes in the branch. Then when finished, the repo owner merges edits from the branch into the master through a "pull request."

Although you can perform these operations using Git commands from your terminal, you can also perform the actions through the browser interface. The browser interface might be helpful if you have less technical people making edits to your content.

To make edits in a separate branch on GitHub:

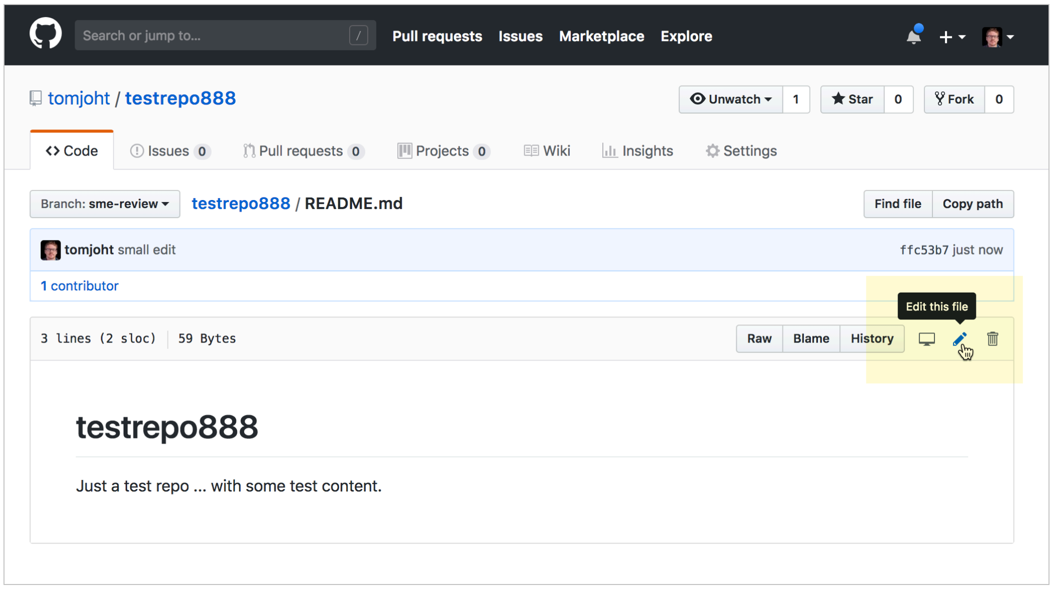
1. Pretend you're a SME reviewer. Go to the same GitHub repo you created in an earlier tutorial (p. 577) (or create a new repo). Create a new branch by selecting the branch drop-down menu and typing a new branch name, such as "sme-review." Then press your **Enter** key.



Creating a new branch

When you create a new branch, the content from the master (or whatever branch you're currently viewing) is copied over into the new branch. The branch is like doing a "Save as" with an existing document.

2. Click a file, and then click the pencil icon (“Edit this file”) to edit the file.



testrepo888

Just a test repo ... with some test content.

Making an edit

3. Make some changes to the content, and then scroll down to the Commit changes area. Explain the reason for the changes and commit the changes to your sme-review branch by clicking **Commit changes**.

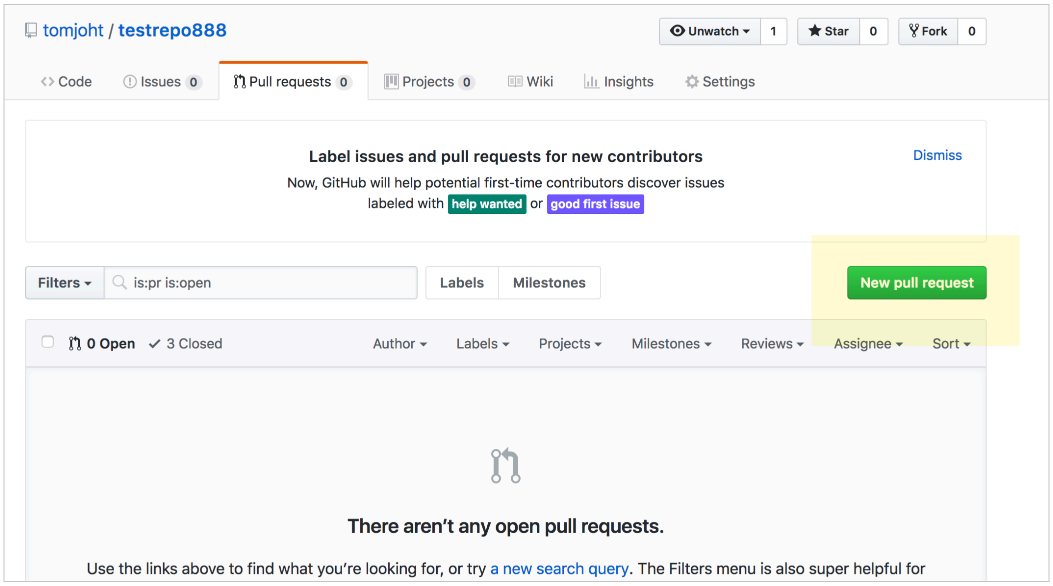
   Reviewers could continue making edits this way until they have finished reviewing all of the documentation. All of the changes are made on a branch, not the master.

Create a pull request

Now let’s suppose that the review process is complete, and it’s time to merge the branch into the master. You merge the branch into the master through a pull request. Any “collaborator” on the team with write access can initiate and complete the pull request (you can add collaborators through Settings > Collaborators).

To create a pull request:

1. View the repository and click the **Pull requests** tab.
2. Click the **New pull request** button.



Label issues and pull requests for new contributors

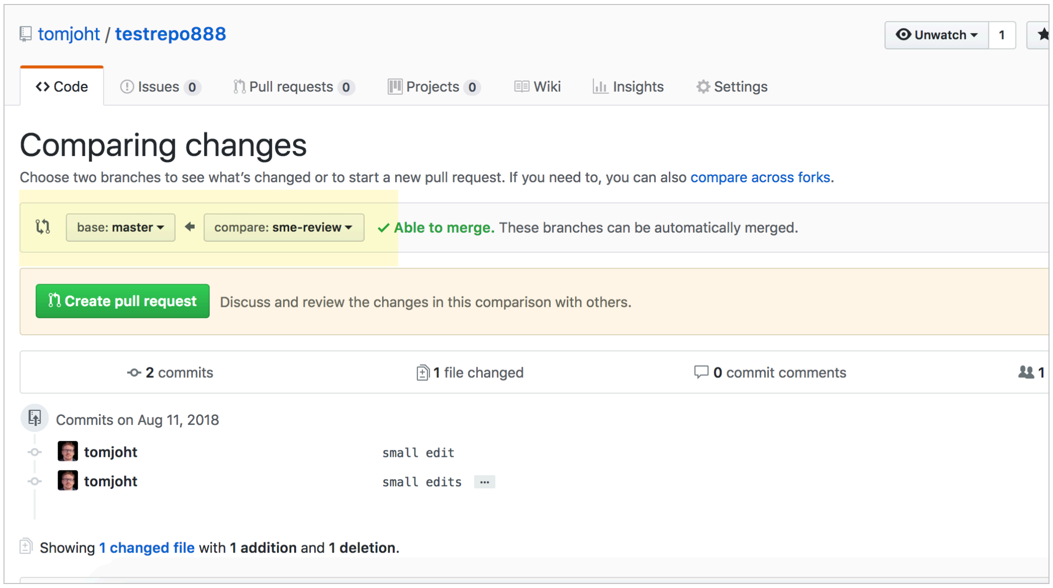
Now, GitHub will help potential first-time contributors discover issues labeled with help wanted or good first issue

There aren't any open pull requests.

Use the links above to find what you're looking for, or try a new search query. The Filters menu is also super helpful for

New Pull Request

3. Select the branch ("sme-review") that you want to compare against the master.



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

✓ **Able to merge.** These branches can be automatically merged.

Discuss and review the changes in this comparison with others.

Commits on Aug 11, 2018

tomjoht	`small edit`
tomjoht	`small edits`

Showing 1 changed file with **1 addition** and **1 deletion**.

Compare to option looks to see if the branches can be merged

When you compare the branch against the master, you can see a list of all the changes. You can view the changes through two viewing modes: Unified or Split (these are tabs shown on the right of the content). Unified shows the edits together in the same content area, whereas split shows the two files side by side.

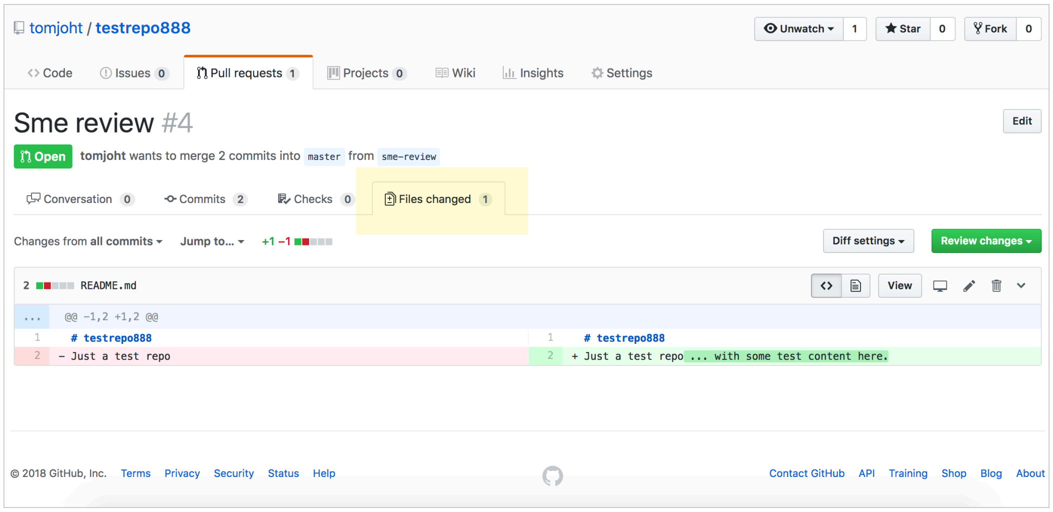
4. Click **Create pull request**.
5. Describe the pull request, and then click **Create pull request**.

The maintainers will then see the Pull Request and can take action to merge it.

Process the pull request

Now pretend you are the project owner, and you see that you received a new pull request. You want to process the pull request and merge the sme-review branch into the master.

1. Click the **Pull requests** tab to see the pending pull requests.
2. Click the pull request and view the changes by clicking the **Files changed** tab.



Sme review #4

tomjoht wants to merge 2 commits into `master` from `sme-review`

```
2 README.md
@@ -1,2 +1,2 @@
1   # testrepo888                 1   # testrepo888
2 - Just a test repo              2 + Just a test repo ... with some test content here.
```

Github files changed

If you only want to implement some of the edits, go into the sme-review branch and make the updates before processing the pull request. The pull request doesn't give you a line-by-line option about which changes you want to accept or reject (like in Microsoft Word's Track Changes). Merging pull requests is an all-or-nothing process. You can also click **Review changes**, add some comments, and then select the **Request changes** radio button, asking the reviewer to make the changes.

Note also that if the pull request is made against an older version of the master, such that the master's original content no longer exists or has moved elsewhere, the merges will be more difficult to make.

3. Click the **Conversation** tab, and then click the **Merge pull request** button.
4. Click **Confirm merge**.

   The sme-review branch gets merged into the master. Now the master and the sme-review branch are the same.

5. Click the **Delete branch** button to delete the sme-review branch.

   If you don't want to delete the branch here, you can always remove old branches by clicking the **branches** link while viewing your Github repository, and then click the **Delete** (trash can) button next to the branch.

   If you look at your list of branches, you'll see that the deleted branch no longer appears.

Add collaborators to your project

You might need to add collaborators to your Github project so they can commit edits to a branch. If other project members aren't collaborators and they want to make edits, they will receive an error. (See Inviting collaborators to a personal repository ↗.)

If people don't have write access, they can fork the repo ↗ instead of making edits on a branch on the same project. Forking a project clones the entire repository, though, rather than creating a branch within the same repository. The fork (copy) then exists in the user's personal GitHub account. You can merge a forked repository (this is the typical model for open-source projects with many outside contributors), but this scenario probably is less common for technical writers working with developers on the same projects.

To add collaborators to your Github project:

1. While viewing your Github repository, click the **Settings** tab.
2. Click the **Collaborators** tab on the left.
3. Type the Github usernames of those you want to have access in the Collaborator area.
4. Click **Add collaborator**.

Static site generators

Static site generators, such as Jekyll, Hugo, and Sphinx, are one of the most common authoring and publishing tools used in docs-as-code scenarios (p. 551). Static site generators build all the files for your website, pushing Markdown files into the layouts you define, running scripts to automate logic you need and more as they generate out HTML files. This section focuses exclusively on static site generators. In upcoming topics, I'll also explore hosting and deployment options (p. 602) and hybrid documentation systems (p. 607).

What are static site generators

Static site generators are applications that you can run on the command line (or potentially through some other UI) to compile a website from simpler source files. For example, you might have various files defining a layout, some "include" files (containing re-usable content), a configuration file, and your Markdown content files.

The static site generator reads your configuration file and pushes your content into the layout files, adds whatever includes are referenced (such as a sidebar or footer or re-used snippets), and writes out the HTML pages from the Markdown sources. Each page usually has the sidebar and other navigation included directly into the page (pre-built), as well as all the other layout code you've defined, ready for viewing online.

Additionally, static site generators can be used programmatically in build scripts that are run as part of a process on a server. This allows them to be leveraged in continuous delivery processes that are triggered by a particular event, such as a commit to a particular branch in a version control repository, or as part of a script.

With a regular content management system (CMS) like WordPress, content is stored in a separate database and dynamically pulled from the database to the web page on each user visit. Static site generators don't have databases — all the content is on the page already, and nothing is dynamically assembled on the fly through PHP or other server-side scripting. All the pages on a static site were built prior to the browser's request, enabling an instantaneous response; nothing changes dynamically based on the user's profile (unless done with client-side JS).

Freedom from the database model makes static site generators much more portable and platform independent. You simply have a collection of text files. In contrast, moving from one CMS to another usually involves database migration, and the many database fields from one CMS don't usually map cleanly to other databases, not to mention the unique server configurations and other infrastructure required for each solution. Static site generators remove that database and infrastructure complexity, making the text files lighter, more portable, and less prone to error from database and server issues.

Before I had my blog idratherbewriting.com in Jekyll, I used WordPress (and was even a WordPress consultant for five years as a side job). I can't count how many times my WordPress blog went down or had other issues. I routinely had to contact Bluehost (my web host) to find out why my site was suddenly down. I religiously made backups of the database, applied security patches and hardening techniques, optimized the database through other tools, and more. And with all of this maintenance hassle, the site was extremely slow, delivering pages in 2+ seconds instead of 0.5 seconds with Jekyll. For my many WordPress clients, I often had to troubleshoot hacked databases.

With static site generators, when you're developing content on your local machine, you usually have web server preview (such as `http://127.0.0.1:4000/`) provided through the static site generator. Many static site generators rebuild your site continuously in the preview server each time you make a change. The time to rebuild your site could take less than a second, or if you have thousands of pages, several minutes.

Because everything is compiled locally from text files, you don't need to worry about security hacks into a database. Everything is a human-readable plain text file, from the content files you write to the application code. It's also incredibly easy to work with custom code, such as special JavaScript libraries, advanced HTML, or other complex code you want to use on a page. You can author your content in Markdown or HTML, add code samples inside code blocks that are processed with a code-syntax highlighter, and more. The openness and flexibility of static site generators let you do what you want with them.

Most static site generators allow you to use a templating and scripting languages, such as Liquid or Go, inside your content. You can use if-else statements, run loops, insert variables, and do a lot more sophisticated processing of your content through this templating language.

Because you're working with text files, you usually store your project files (but not the built site output) in a code repository such as GitHub. You treat your content files with the same workflow as programming code — committing to the repository, pushing and pulling for updates, branching and merging, and more.

When you're ready to publish your site, you can usually build the site directly from your Git repository, rather than building it locally and then uploading the files to a web server. This means your code repository becomes the starting point for your publishing and deployment pipeline. "Continuous delivery," as it's called, eliminates the need to manually build your site and deploy the build. Instead, you just push a commit to your repository, and the continuous delivery mechanism builds and deploys it for you.

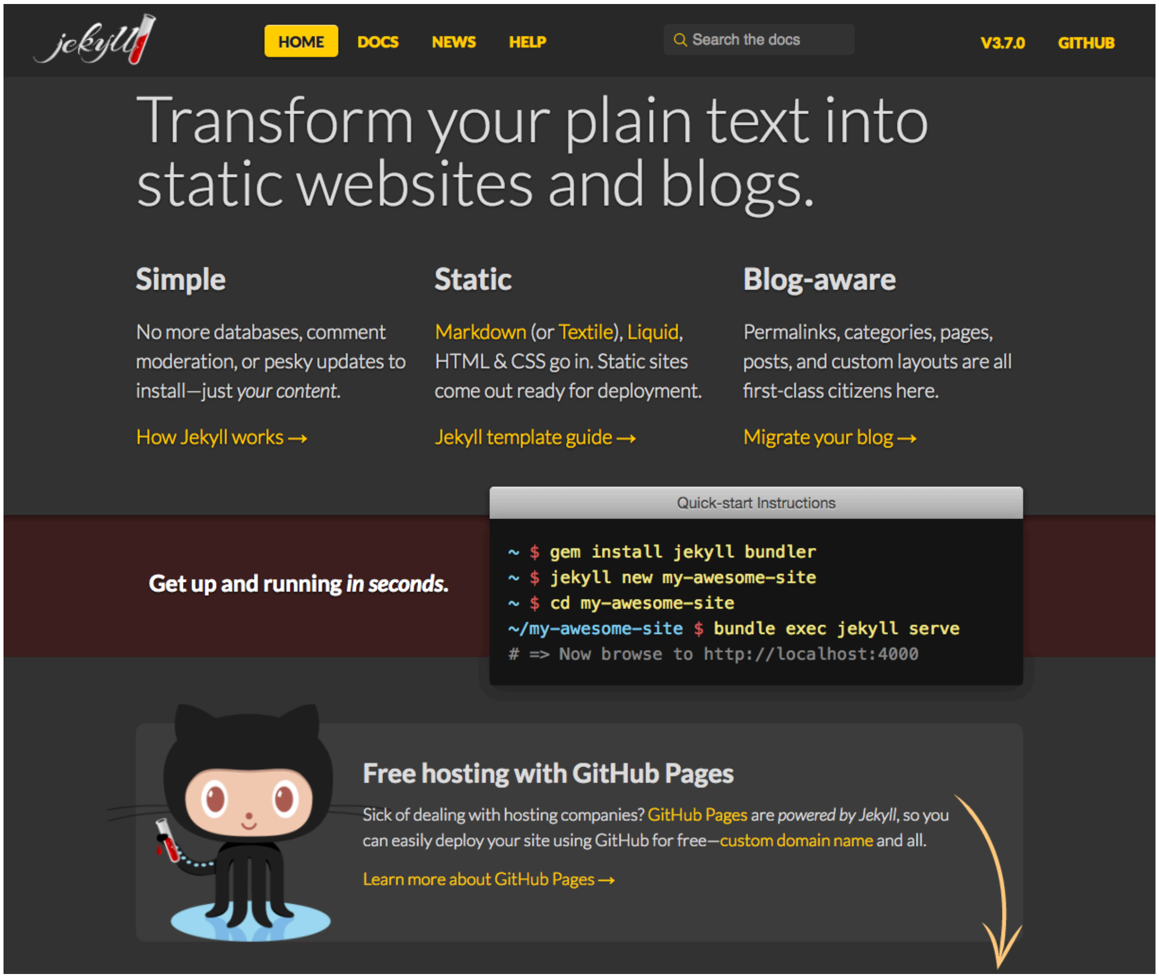
Although there are hundreds of static site generators (you can view a comprehensive list at Staticgen.com ↗), only a handful of are probably relevant for documentation. I'll consider these several here:

- Jekyll (p. 592)
- Hugo (p. 594)
- Sphinx (p. 596)
- MkDocs (p. 599)
- Docusaurus (p. 600)

One could discuss many more — Hexo, Vue, Middleman, Gitbook, Pelican, and so on. But the reality is that only a handful of static site generators are commonly used for documentation projects.

Jekyll

I devote an entire topic to Jekyll (p. 637) in this course, complete with example Git workflows, so I won't go as deep in detail here. Jekyll is a Ruby-based static site generator originally built by the co-founder of GitHub. Jekyll builds your website by converting Markdown to HTML, inserting pages into layouts you define, running any Liquid scripting and logic, compressing styles, and writing the output to a site folder that you can deploy on a web server.



Transform your plain text into static websites and blogs.

Simple

No more databases, comment moderation, or pesky updates to install—just *your content*.

How Jekyll works →

Static

Markdown (or Textile), Liquid, HTML & CSS go in. Static sites come out ready for deployment.

Jekyll template guide →

Blog-aware

Permalinks, categories, pages, posts, and custom layouts are all first-class citizens here.

Migrate your blog →

Get up and running *in seconds*.

```
~ $ gem install jekyll bundler
~ $ jekyll new my-awesome-site
~ $ cd my-awesome-site
~/my-awesome-site $ bundle exec jekyll serve
# => Now browse to http://localhost:4000
```

Free hosting with GitHub Pages

Sick of dealing with hosting companies? GitHub Pages are *powered by Jekyll*, so you can easily deploy your site using GitHub for free—custom domain name and all.

Learn more about GitHub Pages →

There are several compelling reasons to use Jekyll:

- **Large community**. The Jekyll community, arguably the largest and longest-running among static site generator communities, includes web developers, not just documentation-oriented groups. This broader focus attracts more developer attention and helps propel greater usage.
- **Control.** Jekyll provides a lot of powerful features (often through Liquid, a scripting language) that allow you to do almost anything with the platform. This scripting capability gives you an incredible amount of control to abstract complex code from users through simple templates and layouts. Because of this, you probably won't outgrow Jekyll. Jekyll will match whatever web development skills or other JS, HTML, or CSS frameworks you want to throw at it. Even without a development background, it's fairly easy to figure out and code the scripts you need. (See my series Jekyll versus DITA for details on how to do in Jekyll what you're probably used to doing in DITA.)
- **Integration with GitHub and AWS S3.** Tightly coupling Jekyll with the most popular version control repository on the planet (GitHub) almost guarantees its success. The more GitHub is used, the more Jekyll is also used, and vice versa. GitHub Pages (p. 602) will auto-build your Jekyll site (continuous delivery), allowing you to automate the publishing workflow without effort. If GitHub isn't appropriate for your project, you can also publish to AWS S3 bucket using the s3_website plugin, which syncs your Jekyll output with an S3 bucket by only adding or removing the files that changed.

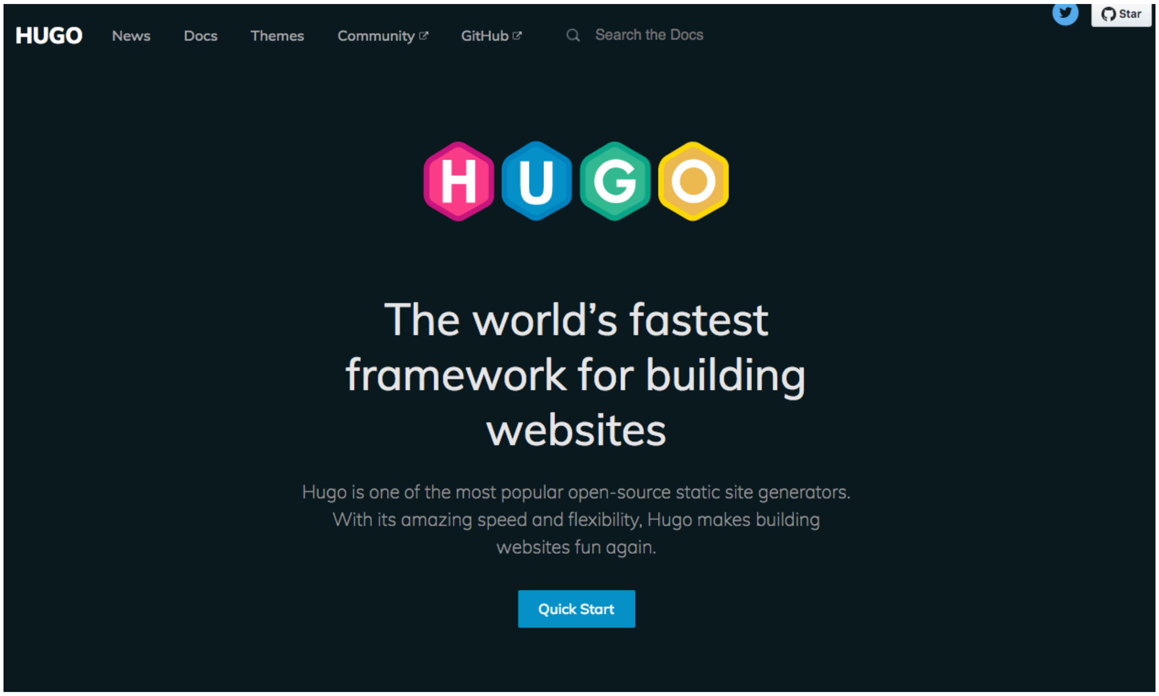
For theming, Jekyll offers the ability to package your theme as a Rubygem and distribute the gem across multiple Jekyll projects. Rubygems is a package manager, which means it's a repository for plugins. You pull the latest gems (plugins) you need from Rubygems through the command line, often using Bundler. Distributing your theme as a Rubygem is one approach you could use for breaking up your project into smaller projects to ensure faster build times.

Although Jekyll was one of the first major static site generators, its popularity has waned, in part due to the lack of leadership and contributors in the open-source project (see Jared White's controversial post, Jekyll and the Genesis of the Jamstack). Additionally, Jekyll's Ruby architecture gives us slow build times (compared to Hugo). Finally, even though Jekyll is supported by GitHub, GitHub is slow to roll in version updates, so even though Jekyll is up to version 4.x+, GitHub supports only version 3.9.0.

Although I use Jekyll for all my sites, if starting out today, I probably wouldn't choose Jekyll, as I think it's on the way out. That said, GitHub could up their support game, and Jekyll could continue for many years forward. The platform, especially Liquid syntax, is one of the easier ones to learn and work with.

Hugo

Hugo is a static site generator that is rapidly growing in popularity. Based on the Go language, Hugo builds your site significantly faster than most other static site generators, especially Jekyll. There's an impressive number of themes, including some designed for documentation. Specifically, see the Docsy theme, the Learn theme and this Multilingual API documentation theme.



The world's fastest framework for building websites

Hugo is one of the most popular open-source static site generators. With its amazing speed and flexibility, Hugo makes building websites fun again.

As with Jekyll, Hugo allows you to write in Markdown, add frontmatter content in YAML (or TOML or JSON) at the top of your Markdown pages, and more. In this sense, Hugo shares a lot of similarity with Jekyll and other static site generators.

Hugo has a robust and flexible templating language (Golang) that makes it appealing to designers, who can build more sophisticated websites based on the depth of the platform (see Hugo's docs here). Go templating has more of a learning curve than templating with Liquid in Jekyll, and the docs might assume more technical familiarity than many users have. Still, the main selling point behind Hugo is that it builds your site quickly. This speed factor might be enough to compensate for the steeper learning curve.

It's also worth noting that Go is a language developed and supported by Google, and the Docsy theme theme also has a group of enthusiastic supporters, many of whom are Googlers who needed a publishing framework for many of the open-source tools like Kubernetes, Kube Flow, and gRPC. Because of this support from Google, Hugo might have more longevity than other static site generators that seem to be hobby projects from solo developers.

Comparing speed with Hugo with Jekyll

Build times may not be immediately apparent when you first start evaluating static site generators (often using small projects as tests). You probably won't realize how important speed is until you have thousands of pages in your site and are waiting for it to build.

Speed here refers to the time to compile your web output, not the time your site takes to load when visitors view the content in a browser. Most static site generators load the pre-built pages quickly (less than 0.5 seconds), but the time it takes for the files to compile into a website before they're deployed depends on the platform, the number of pages, and the complexity of the code on the pages.

Although it depends on how you've coded your site (e.g., the number of `for` loops that iterate through pages), in general, I've noticed that with Jekyll projects, if you have, say, 1,000 pages in your project, it might take about a minute or two to build the site. Thus, if you have 5,000 pages, you could be waiting 5 minutes or more for the site to build. The whole automatic re-building feature becomes almost irrelevant, and it can be challenging to identify formatting or other errors until the build finishes. (There are workarounds, though, and I'll discuss them later on.)

If Hugo can build a site much faster, it offers an advantage in the choice of static site generators. Smashing Magazine recently chose Hugo and built a variety of complementary tools for managing their site.

For a detailed comparison of Hugo versus Jekyll, see Hugo vs. Jekyll: Comparing the leading static website generators. In one of the comments, a reader makes some interesting comments about speed:

> Our documentation is about 2700 pages…. Generating the whole site takes about 90 seconds. That's kind of annoying when you're iterating over small changes. I did a basic test in Hugo, it does it in about 500ms.

This build time is a serious speed advantage that will allow you to scale your documentation site in robust ways. The author (whose docs are at https://docs.mendix.com) made the switch from Jekyll to Hugo (see the doc overview in GitHub). His switch suggests that speed is perhaps a primary characteristic to evaluate in static site generators.

The deliberation between Hugo and Jekyll will require you to think about project size — how big should your project be? Should you have one giant project, with content for all documentation/products stored in the same repo? Or should you have multiple smaller repos? These are some of the considerations I wrestled with when implementing docs-as-code tooling (p. 645). I concluded that having a single, massive project is preferable because it allows easier content re-use, onboarding, validation, and error checking, deployment management, and more.

Regarding build speed, there are workarounds in Jekyll to enabling faster builds. In your build commands, you can limit the builds to one particular doc directory. For example, you can have one configuration file (e.g., _config.yml, the default) that sets all content as `publish: true`, and another configuration file (e.g., config-acme.yml) that sets all content as `publish: false` except for a particular doc directory (the one you're working with, e.g., acme). When you're working with that acme doc directory, you can build Jekyll like this:

```
jekyll serve --config _config.yml,config-acme.yml
```

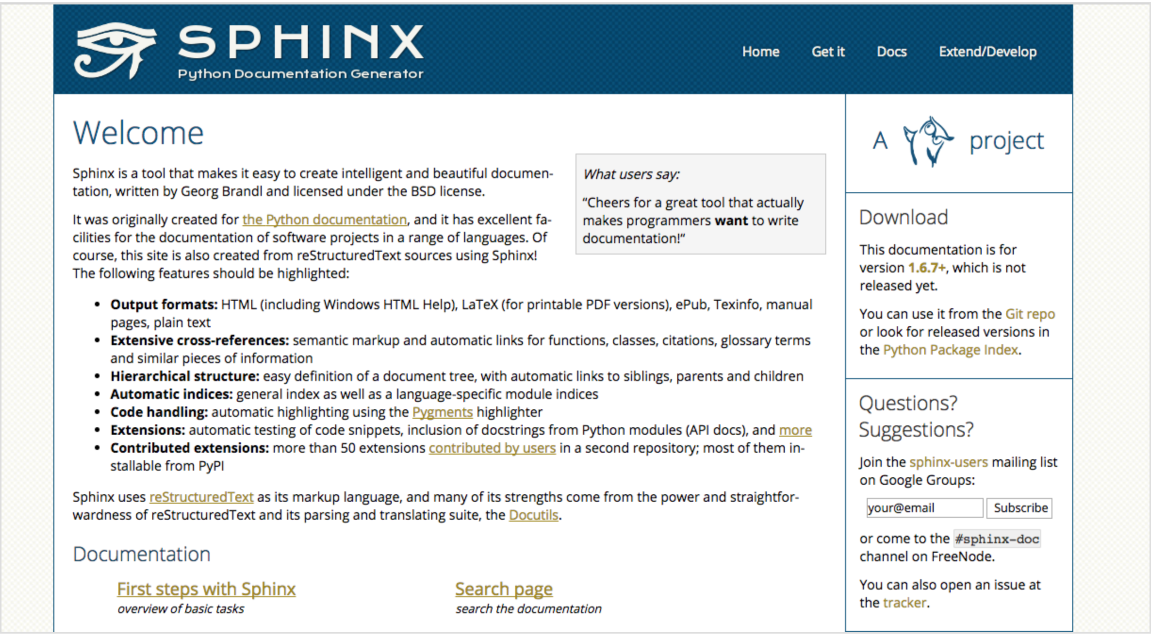
The `config-acme.yml` will overwrite the default `_config.yml` to enable one specific doc directory as `publish: true` while disabling all others. Using this method, Jekyll builds lightning fast. This method tends to work quite well and is used by others with large Jekyll projects as well. Writers usually focus on one documentation directory at a time. If you have continuous delivery configured with the server, when it's time to push out the full build (where `publish: true` is applied to all directories and no config-acme.yml file is used), the full build process takes place on the server, not the local machine. (The server might have other pipeline logic that validates, ingests, and deploys files as well, adding to the time.)

Although static site generators seem to change quickly, it's harder for one tool, like Hugo, to overtake another, like Jekyll, because of the custom coding developers usually do with the platform. If you're just using someone's theme with general Markdown pages, great, switching will be easy. But if you've built custom layouts and frontmatter in your Markdown pages that gets processed in unique ways, as well as other custom scripts or code that you created in your theme specifically for your content, changing platforms will be more challenging. You'll have to change all your custom Liquid scripting to Go. Or if working with another platform, you might need to change your Go scripts to Jinja templating, and so forth.

For this reason, unless you're using themes built by others, you don't often jump from one platform to the next as you might do with DITA projects, where all content usually conforms to the same specification.

Sphinx

Sphinx ↗ is a popular static site generator based on Python. It was originally developed by the Python community to document the Python programming language (and it has some direct capability to document Python classes), but Sphinx is now commonly used for many documentation projects unrelated to Python. Part of Sphinx's popularity is due to its Python foundation since Python works well for many documentation-related scripting scenarios.



Welcome

Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license.

It was originally created for the Python documentation, and it has excellent facilities for the documentation of software projects in a range of languages. Of course, this site is also created from reStructuredText sources using Sphinx! The following features should be highlighted:

- **Output formats:** HTML (including Windows HTML Help), LaTeX (for printable PDF versions), ePub, Texinfo, manual pages, plain text
- **Extensive cross-references:** semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information
- **Hierarchical structure:** easy definition of a document tree, with automatic links to siblings, parents and children
- **Automatic indices:** general index as well as a language-specific module indices
- **Code handling:** automatic highlighting using the Pygments highlighter
- **Extensions:** automatic testing of code snippets, inclusion of docstrings from Python modules (API docs), and more
- **Contributed extensions:** more than 50 extensions contributed by users in a second repository; most of them installable from PyPI

Sphinx uses reStructuredText as its markup language, and many of its strengths come from the power and straightforwardness of reStructuredText and its parsing and translating suite, the Docutils.

What users say:

"Cheers for a great tool that actually makes programmers **want** to write documentation!"

Documentation

First steps with Sphinx
overview of basic tasks

Search page
search the documentation

Download

This documentation is for version **1.6.7+**, which is not released yet.

You can use it from the Git repo or look for released versions in the Python Package Index.

Questions? Suggestions?

Join the sphinx-users mailing list on Google Groups:

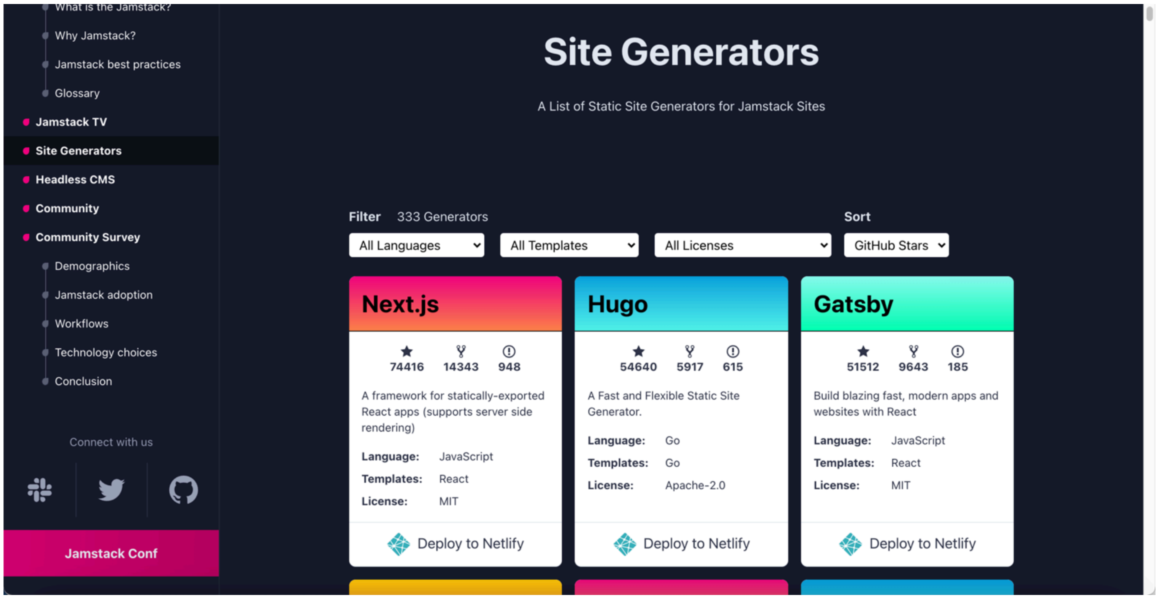
or come to the `#sphinx-doc` channel on FreeNode.

You can also open an issue at the tracker.

Because Sphinx was designed from the ground up as a documentation tool, not just as a tool for building websites (like Jekyll and Hugo), Sphinx has more documentation-specific functionality that is often absent from other static site generator tools. Some of these documentation-specific features include robust search, more advanced linking (linking to sections, automating titles based on links, cross-references, and more), and use of reStructuredText (rST), which is more semantically rich, standard, and extensible than Markdown. (See What about reStructuredText and Asciidoc? (p. 560) for more details around rST compared to Markdown.)

For continuous deployment with your hosting, Sphinx can be used with the readthedocs.com ↗ platform. Overall, Sphinx has a passionate fan base among those who use it, especially among the Python community. However, because Sphinx was specifically designed as a documentation tool, the community might not be as large as some of the other static site generator communities (which use the static site generators for building general websites, not just documentation sites).

As of October 2021, Staticgen.com ↗ shows the number of stars, forks, and issues as follows:



What is the Jamstack?
Why Jamstack?
Jamstack best practices
Glossary
Jamstack TV
Site Generators
Headless CMS
Community
Community Survey
Demographics
Jamstack adoption
Workflows
Technology choices
Conclusion

Connect with us

Jamstack Conf

Site Generators

A List of Static Site Generators for Jamstack Sites

Filter 333 Generators
All Languages | All Templates | All Licenses
Sort GitHub Stars

Next.js
★ 74416 14343 948
A framework for statically-exported React apps (supports server side rendering)
Language: JavaScript
Templates: React
License: MIT
Deploy to Netlify

Hugo
★ 54640 5917 615
A Fast and Flexible Static Site Generator.
Language: Go
Templates: Go
License: Apache-2.0
Deploy to Netlify

Gatsby
★ 51512 9643 185
Build blazing fast, modern apps and websites with React
Language: JavaScript
Templates: React
License: MIT
Deploy to Netlify

Top static site generators

On the Staticgen.com site, the star icon represents the number of users who have "starred" the project (basically followed its activity). The forked icon represents the number of repo clones that exist registered on their platform (GitHub, etc.). The bug icon represents the number of open issues logged against the project. To gauge how active the project is, browse the GitHub source and look to see how regular the commits are.

Next, Hugo, and Gatsby are the most common static site generators. If you look at Staticgen.com, you'll see that between Hugo and Sphinx, there are many other more popular static site generators. But I called out Sphinx here because of its popularity *among documentation groups* and for its integration with Read the Docs (p. 603). That's what you have to keep in mind with static site generators — a particular option might be more popular, but for what purpose? Building general websites, or building documentation websites? Documentation websites tend to have unique needs and requirements (like a robust sidebar), so don't be tricked by merely choosing the most popular options listed in staticgen.com.

Gatsby

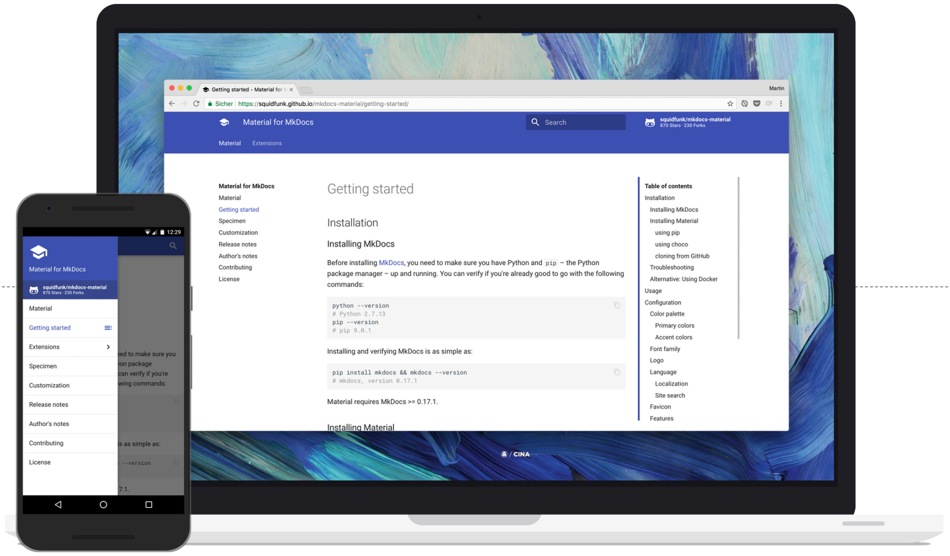
Gatsby is a popular static site generator, and even offers a cloud-hosted version. However, Gatsby is more of an *application static site generator* than a pure static site generator that lets you write in Markdown and publish HTML. Gatsby appeals to React developers who are building more extensive web applications; the architecture and code in Gatsby tend to be more complex and confusing than the other simpler static site generators. I've met a number of people who experimented with Gatsby and then abandoned it (even though tech docs are still a primary use case for Gatsby). Other application static site generators similar to Gatsby include Next.js and Nuxt.js.

In selecting a static site generator, I don't recommend using an application-based static site generator. You're writing docs and primarily working in Markdown and HTML. You're probably not building more extensive web app logic. You mainly want a tool that will take your Markdown and publish the HTML. Your company might want to create custom pipelines that push the published HTML into some other framework or system, and having a simpler static site generator will be advantageous in these scenarios. In general, stick with simple.

However, if your larger goal is to build a developer portal, with all the internal application logic (for example, API keys and developer profiles) that goes along with a portal, Gatsby might be the system your UX team chooses to use. Leveraging React components can speed development time, and some tools such as Redoc (p. 249) offer developer portals build on Gatsby. If the rest of your developer portal is built on Gatsby, it might be a sound choice for your docs as well.

MkDocs

MkDocs is a static site generator based on Python and designed for documentation projects. Similar to Jekyll, with MkDocs you write in Markdown and store page navigation in YAML files. You can adjust the CSS and other code (or create your own theme). Notably, the MkDocs provides some themes that are more specific to documentation, such as the Material theme. MkDocs also offers a theme ("ReadtheDocs") that resembles the Read the Docs platform.



Getting started

Installation

Installing MkDocs

Before installing MkDocs, you need to make sure you have Python and `pip` – the Python package manager – up and running. You can verify if you're already good to go with the following commands:

```
python --version
# Python 2.7.13
pip --version
# pip 9.0.1
```

Installing and verifying MkDocs is as simple as:

```
pip install mkdocs && mkdocs --version
# mkdocs, version 0.17.1
```

Material requires MkDocs >= 0.17.1.

Installing Material

Some other themes are also available. MkDocs uses Jinja templating, provides template variables for custom theming, and more.

Although there are many static site generators with similar features, MkDocs is one more specifically oriented towards documentation. For example, it does include search. (You can incorporate Algolia search into any of these platforms, though, so built-in search — unless it's really phenomenal — probably shouldn't be a distinguishing factor.)

While it seems like orienting the platform towards documentation would be advantageous for tech writers, this approach might actually backfire, because it shrinks the community. The number of general web designers versus documentation designers is probably a ratio of 100:1. As such, MkDocs remains a small, niche platform that probably won't see much growth and long-term development beyond the original designer's needs.

This is the constant tradeoff with tools — the tools and platforms with the most community and usage aren't usually the doc tools. The doc tools have more features designed for tech writers, but they lack the momentum and depth of the more popular website building tools.

Docusaurus

Docusaurus ↗, built by Facebook, is also a popular static site generator oriented towards documentation needs. Docusaurus includes integration with Algolia for search, supports document versioning, translation, and more. The React foundation will also make it popular if you have front-end developers who prefer to work with React.

With support from Facebook, Docusaurus looks like a great option for a documentation website, and although I haven't experimented with it myself, feedback from others has been positive. You can see how Docusaurus compares with other tools ↗ as well as look over their showcase ↗. The showcase, which includes many doc sites such as Algolia ↗, GraphQL ↗, Redis ↗, can give you an idea of how your doc site will look. Most themes have an attractive left sticky sidebar with collapse/expand toggles along with a right sidebar showing a floating table of contents.

What about this or that tool?

Right now there are probably many readers who are clenching their first and lowering their eyebrows in anger at the omission of their tool. *What about … Docpad!!??? What about Nikola??!! What about Slate!! And Docsify?*

Hey, there are *a lot* of tool options out there, and you might have found the perfect match between your content needs and the tool. Additionally, the tools landscape for developer docs is robust, complex, and seemingly endless. Most people don't have deep exposure to that many tools. If you start with one and become familiar with it, changing course becomes harder. There often isn't a killer feature that prompts you to refactor and recode your entire website.

And the winds seem to change from year to year. What may be relevant one day might be passé the next. Docs-as-code tooling is a difficult space to navigate, and selecting the right tool for your needs is a tough question. I offer more specific advice and recommendations in Which tool to choose for API docs — my recommendations (p. 633). The tool you choose will affect both your productivity and capability, so it tends to be an important choice.

For more doc tools, see the Generating Docs ↗ list in Beautiful Docs. Additionally, DocBuilds ↗ tries to index some of more popular documentation-specific static site generators.

If you want to explore different API doc sites (p. 536) and analyze what tools they use, check out a Chrome extension called Wappalyzer ↗. With this extension, you can easily see the underlying technologies for a site. It can help you detect trends and patterns with API tooling.

One pattern you'll observe, as I noted in the overview (p. 531) to this section, is that few API documentation sites tend to use traditional help authoring tools. Yes, common documentation generators like Swagger (p. 218), Redoc (p. 249), and Stoplight (p. 201) are used, but for the bulk of the conceptual documentation, you'll find a variety of tools used, with no dominant static site generator or architecture.

Tools for generating the OpenAPI reference

In this discussion on tools, I have purposely avoided diving deeper into tools that auto-generate out reference documentation from the OpenAPI specification (p. 181). I cover these tools more in the OpenAPI spec and generated reference docs (p. 179) section. These tools include Stoplight (p. 195), SwaggerHub (p. 230), Redoc (p. 249) (all site sponsors, by the way), as well as Readme.com ↗ and DeveloperHub.io ↗. For more of these tools, see Ultimate Guide to 30+ API Documentation Solutions ↗ from Nordic APIs and Tools and Integrations ↗ from Smartbear.

One drawback with tools that auto-generate out the OpenAPI reference is that their support for more conceptual documentation will likely be limited. Tech writers tend to spend the bulk of their time writing more conceptual docs; reference docs are often written and generated by engineering teams themselves. Even so, you will likely run into challenges with integrating the conceptual and reference docs into a seamless experience (p. 241).

Hosting and deployment options

Static site generators (p. 591) handle content development, but not hosting and deployment. For this, you need another category of tools. I call this category of tools “hosting and deployment options.”

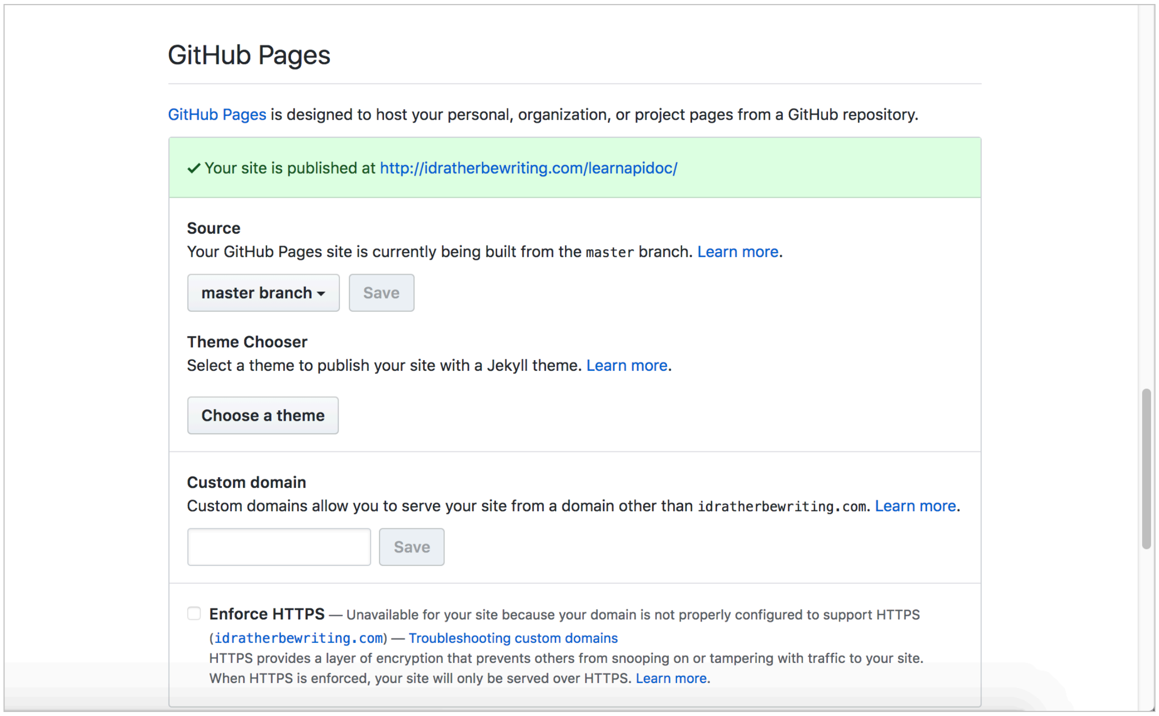
Theoretically, you could publish a static website on any web server (e.g., AWS S3, Bluehost, and more). But continuous delivery hosting platforms do something more: They automatically build your output when you commit a change to a repo. The whole build process takes place entirely on the server.

Hosting and deployment platforms usually offer a number of additional features beyond simple web hosting, such as SSL, CDNs, minification, authentication, backup/redundancy, and more. These platforms often integrate with specific static site generators as well (which is one reason I limited my earlier discussions to Jekyll, Hugo, and Sphinx).

GitHub Pages

GitHub Pages provides a free hosting and deployment option for Jekyll projects. If you upload a Jekyll project to a GitHub repository, you can indicate that it’s a Jekyll project in your GitHub repo’s Settings, and GitHub will automatically build it when you commit to your repo. This feature — building Jekyll projects directly from your GitHub repo — is referred to as GitHub Pages.

In your GitHub repo, click **Settings** and scroll down to **GitHub Pages**. This section is where you activate GitHub Pages for your project.



GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

✓ Your site is published at http://idratherbewriting.com/learnapidoc/

Source

Your GitHub Pages site is currently being built from the `master` branch. Learn more.

Theme Chooser

Select a theme to publish your site with a Jekyll theme. Learn more.

Custom domain

Custom domains allow you to serve your site from a domain other than `idratherbewriting.com`. Learn more.

☐ **Enforce HTTPS** — Unavailable for your site because your domain is not properly configured to support HTTPS (`idratherbewriting.com`) — Troubleshooting custom domains

HTTPS provides a layer of encryption that prevents others from snooping on or tampering with traffic to your site. When HTTPS is enforced, your site will only be served over HTTPS. Learn more.

Every GitHub repository is potentially a Jekyll project that you can auto-build when you commit to it.

The tight integration of Jekyll with GitHub makes for a compelling argument to use a Jekyll-GitHub solution. For the most part, GitHub is the dominant platform for open-source projects. If you're already using GitHub, it makes sense to choose a static site generator that integrates into the same platform to build your docs.

GitHub Pages is free but does have some limitations in scope:

> - GitHub Pages source repositories have a recommended limit of 1GB.
> - Published GitHub Pages sites may be no larger than 1 GB.
> - GitHub Pages sites have a soft bandwidth limit of 100GB per month.
> - GitHub Pages sites have a soft limit of 10 builds per hour. (See Usage Limits ↗)

Unlike with other hosting and deployment platforms, GitHub Pages doesn't offer a commercial version that expands these limits. You can learn more about GitHub Pages here ↗.

I build this site and my blog ↗ using Jekyll and GitHub Pages. They are actually separate Jekyll projects and repos. My blog is in a GitHub repo called `tomjoht.github.io` ↗, named after my GitHub username but published using a custom domain `idratherbewriting.com`. (Without the custom domain, it would be available at `http://tomjoht.github.com`.) The API doc site is in a repo called learnapidoc ↗. It's available by default at `https://idratherbewriting.com/learnapidoc`. They seem like the same site, but they are really separate projects in separate repos. The fact that each repo in addition to the primary repository (`tomjoht.github.io`) appears as a subdirectory to the primary domain (`idratherbewriting.com`) is pretty cool.

CloudCannon

Suppose you want to use Jekyll and GitHub, but you're frustrated by GitHub's limitations and you need a more robust platform for your Jekyll project. If so, CloudCannon ↗ is your solution. CloudCannon gives you a host of additional features ↗ that GitHub lacks, such as:

- Authentication of users
- Multiple environments for different branches
- Visual online interface for editing
- Jekyll plugins
- SSL for custom domains
- Automatic minification, and more

The founders of CloudCannon are experts with Jekyll and have designed the platform specifically for Jekyll projects. They also created a host of Jekyll tutorials ↗ to enrich developer knowledge.

For a tutorial on setting up Jekyll on CloudCannon, see Jekyll and CloudCannon continuous deployment tutorial (p. 637).

Read the Docs

Read the Docs is an online hosting and deployment platform that can read Sphinx projects (from a public repository such as GitHub or Bitbucket) and automatically build the web output. In other words, it is a "continuous documentation platform for Sphinx" (see An introduction to Sphinx and Read the Docs for Technical Writers ↗).

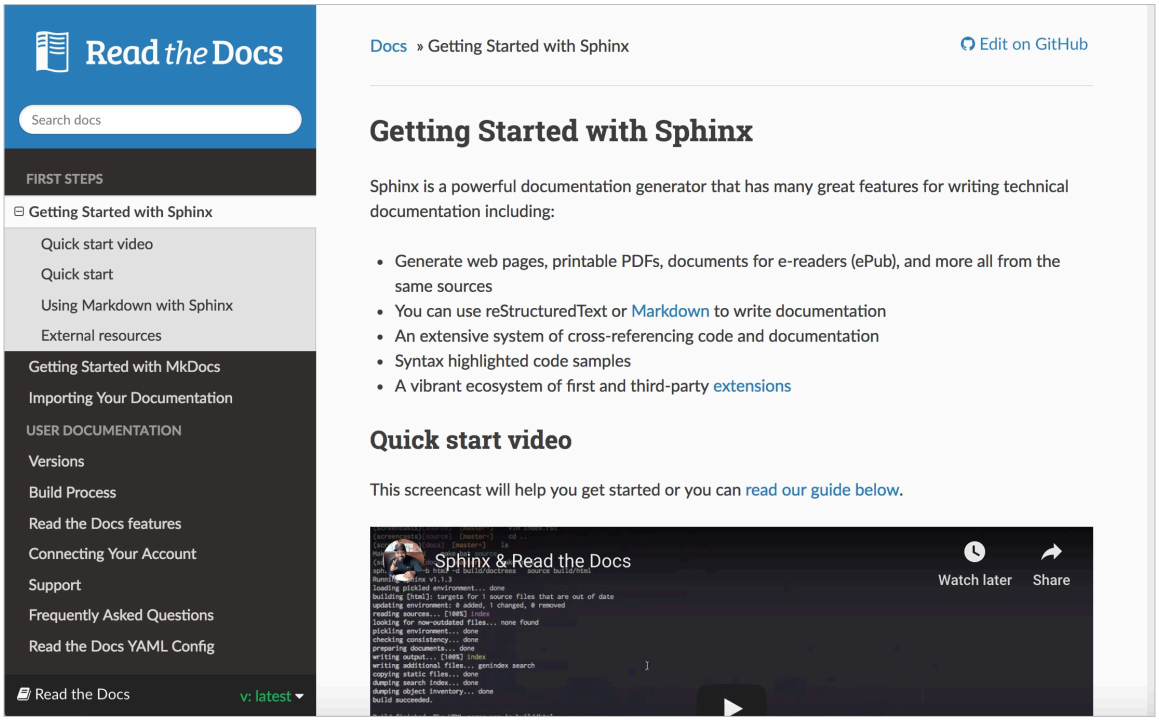
The introduction on the Read the Docs homepage ↗ describes the platform as follows:

> We will host your documentation for free forever. There are no tricks. We help 94,898 open-source projects share their docs. … Whenever you push code to your favorite version control system, whether that is Git, Mercurial, Bazaar, or Subversion, we will automatically build your docs so your code and documentation are never out of sync.

Read the Docs provides both an open-source, free version (readthedocs.org) and a commercial version (readthedocs.com). These tiers allow you to level-up your project when your needs mature but also don't lock you into a paid solution when you're not ready for it.

Read the Docs provides themes specific for documentation websites, and also lets you author in reStructuredText (or Markdown, if you prefer that instead). reStructuredText provides more documentation-specific features and semantics — see my discussion in What about reStructuredText and Asciidoc? (p. 560) for more details, or see Why You Shouldn't Use "Markdown" for Documentation for a more passionate argument for rST.

The Read the Docs documentation shows a sample output.



Getting Started with Sphinx

Sphinx is a powerful documentation generator that has many great features for writing technical documentation including:

- Generate web pages, printable PDFs, documents for e-readers (ePub), and more all from the same sources
- You can use reStructuredText or Markdown to write documentation
- An extensive system of cross-referencing code and documentation
- Syntax highlighted code samples
- A vibrant ecosystem of first and third-party extensions

Quick start video

This screencast will help you get started or you can read our guide below.

Some key features include a robust sidebar with expand/collapse functionality, search, versioning, output to PDF and ePub, and more.

To learn more about the platform, read through the Read the Docs guide. Read the Docs includes most of the features technical writers would expect, especially related to single-source publishing. Some of these features, noted in An introduction to Sphinx and Read the Docs for Technical Writers, include the following:

- Output HTML, PDF, ePub, and more
- Content reuse through includes
- Conditional includes based on content type and tags
- Multiple mature HTML themes that provide great user experience on mobile and desktop
- Referencing across pages, documents, and projects
- Index and Glossary support
- Internationalization support.

The Read the Docs platform was co-founded by Eric Holscher, the same co-founder of Write the Docs. Write the Docs was originally intended as a conference for the Read the Docs community but evolved into a more general conference focused on technical communication for software projects. If you go to a Write the Docs conference, you'll find that sessions focus more on best practices for documentation rather than discussions about tools. (You can read my post, Impressions from the Write the Docs Conference or listen to this Write the Docs podcast with the co-founders for more details.)

Read the Docs has an impressive number of users. The platform has thousands of projects and receives millions of page views a month across these projects. In 2016, Read the Docs had more than 77,000 projects and received 338 million page views and 75 million unique visitors). You can view their 2017 stats here. Read the Docs is one of the most visited sites on the web and continues to grow at an impressive rate.

Netlify

Netlify is a popular hosting and deployment service for static site projects. Unlike with other hosting platforms, Netlify works with almost any static site generator, not just with Jekyll or Sphinx.

Netlify offers continuous delivery for your project. You can store your content on GitHub, GitLab, or Bitbucket, then link it to Netlify, and Netlify will build whenever you push changes.

Netlify not only offers a free plan with features similar to GitHub Pages but also lets you scale up to Pro, Business, or Enterprise plans for more robust needs. With Netlify, you can get deploy previews, rollbacks, form handling, distributed content delivery network (CDN), infinite scalability, SSL, a programmable API, CLI, and more.

The most impressive example of a Netlify-hosted site is Smashing Magazine. Previously hosted on WordPress, Smashing Magazine made the switch to Netlify, with Hugo as the static site generator engine. See Smashing Magazine just got 10x faster for details.

Other notable doc sites using Netlify include Docker, Kubernetes, React, Yarn, Lodash, Gatsby, and Hugo.

Complementing Netlify is Netlify CMS (p. 608), a headless CMS for your content (which I discuss in more detail in Hybrid documentation platforms (p. 607)).

Aerobatic

Aerobatic is similar to Netlify in that it builds and publishes your static site. Aerobatic gives you a robust publishing engine that includes a CDN, SSL, continuous delivery, a deployment CLI, password protection, and more. Aerobatic can build your site using various static site generators, including Jekyll, Hugo, Hexo, and more. Aerobatic says,

> Aerobatic is a specialized platform for efficient delivery of static webpages and website assets. We take care of the configuration details for you that provide the best balance of performance and maintainability. Stop fiddling with CDNs and web server configs and focus on coding great front-end experiences. — Static website serving

Overall, there are many options for hosting and deploying your site. GitHub Pages, CloudCannon, Read the Docs, Netlify, and Aerobatic are just a few. You can also probably explore custom-built hosting and deployment options available through your company's existing infrastructure.

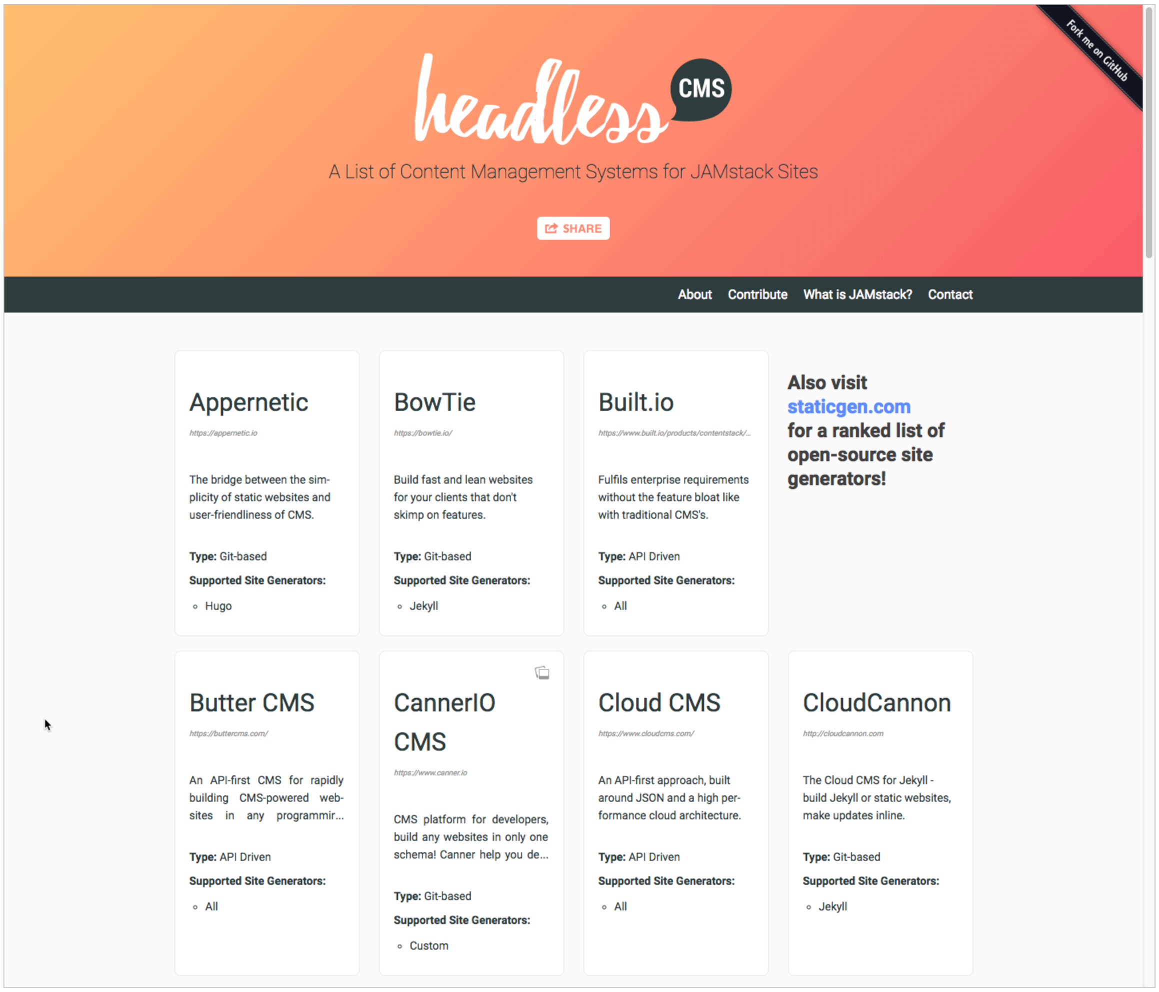
There's another category of tools that I'm calling Hybrid documentation platforms (p. 607). Some of these solutions combine the authoring and deployment solutions together in different ways. These solutions might offer a "headless" authoring interface with a body that is stores content in another place, such as GitHub.

Hybrid documentation systems

Rounding out the publishing tool options, there is a class of developer doc tools that provide online GUIs for authoring and publishing, but they still store your content as flat files in repositories such as GitHub and Bitbucket. In other words, they provide a WordPress.com-like experience for your content (giving you a user interface to browse your posts, pages, layouts, and other content) but allow your content to live in plain text files in version control repositories.

Headless CMS solutions

Just as we have staticgen.com that lists common static site generators, there's a similar index of headless content management systems, this one arranged in alphabetical order (rather than ranked by popularity).



Apprenetic
https://appernetic.io
The bridge between the simplicity of static websites and user-friendliness of CMS.
Type: Git-based
Supported Site Generators:
- Hugo

BowTie
https://bowtie.io/
Build fast and lean websites for your clients that don't skimp on features.
Type: Git-based
Supported Site Generators:
- Jekyll

Built.io
https://www.built.io/products/contentstack/...
Fulfils enterprise requirements without the feature bloat like with traditional CMS's.
Type: API Driven
Supported Site Generators:
- All

Also visit staticgen.com for a ranked list of open-source site generators!

Butter CMS
https://buttercms.com/
An API-first CMS for rapidly building CMS-powered websites in any programmir...
Type: API Driven
Supported Site Generators:
- All

CannerIO CMS
https://www.canner.io
CMS platform for developers, build any websites in only one schema! Canner help you de...
Type: Git-based
Supported Site Generators:
- Custom

Cloud CMS
https://www.cloudcms.com/
An API-first approach, built around JSON and a high performance cloud architecture.
Type: API Driven
Supported Site Generators:
- All

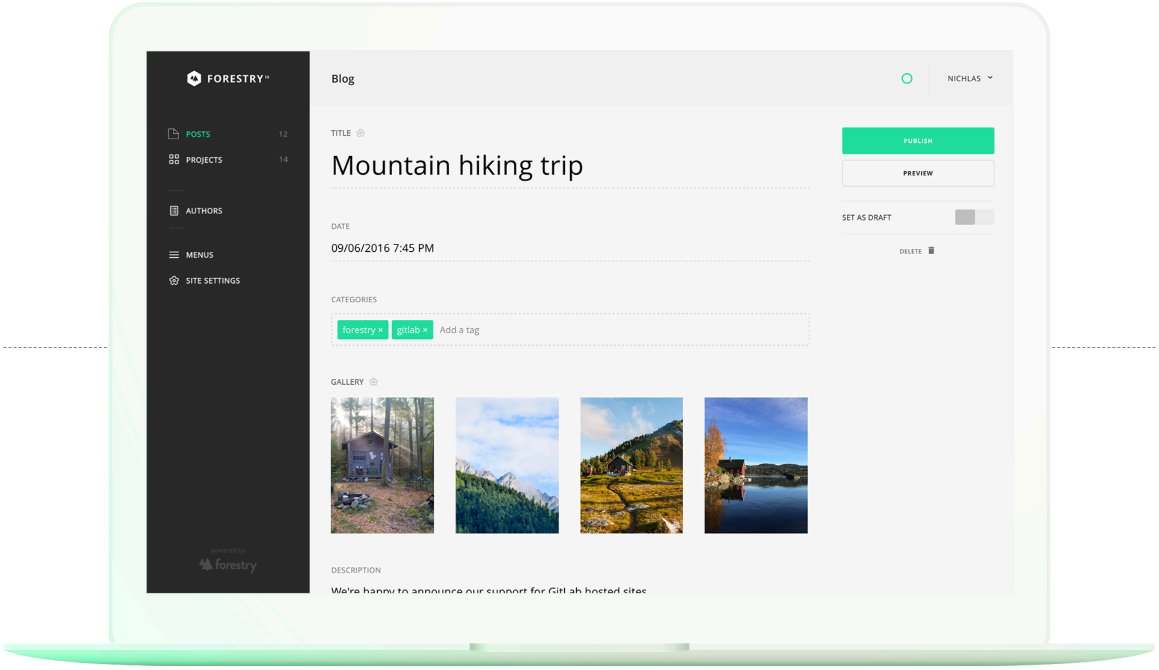
CloudCannon
http://cloudcannon.com
The Cloud CMS for Jekyll - build Jekyll or static websites, make updates inline.
Type: Git-based
Supported Site Generators:
- Jekyll

For an in-depth introdution to headless CMSs, see Headless CMS vs Decoupled CMS: The Ultimate Guide. There are nuances here that I won't go into, but the basic metaphor is that the head refers to the authoring interface, and the body refers to the storage database or file system. If a CMS is headless, it means the authoring interface is connected to the data storage through an API, and the data storage can live in a completely different system, such as GitHub.

Not all tools listed here are headless CMSs, so I have called them more generally “hybrid” solutions, since many of these doc tools don’t fit into neat categories.

Forestry.io

Forestry.io is similar to CloudCannon in that it offers online hosting for Jekyll projects, but it also provides hosting for Hugo and Git. Forestry’s emphasis is on providing an online CMS interface for static site generators. The CMS interface gives you a WordPress-like GUI for seeing and managing your content.



TITLE
Mountain hiking trip

DATE
09/06/2016 7:45 PM

CATEGORIES
forestry × gitlab ×

DESCRIPTION
We're happy to announce our support for GitLab hosted sites

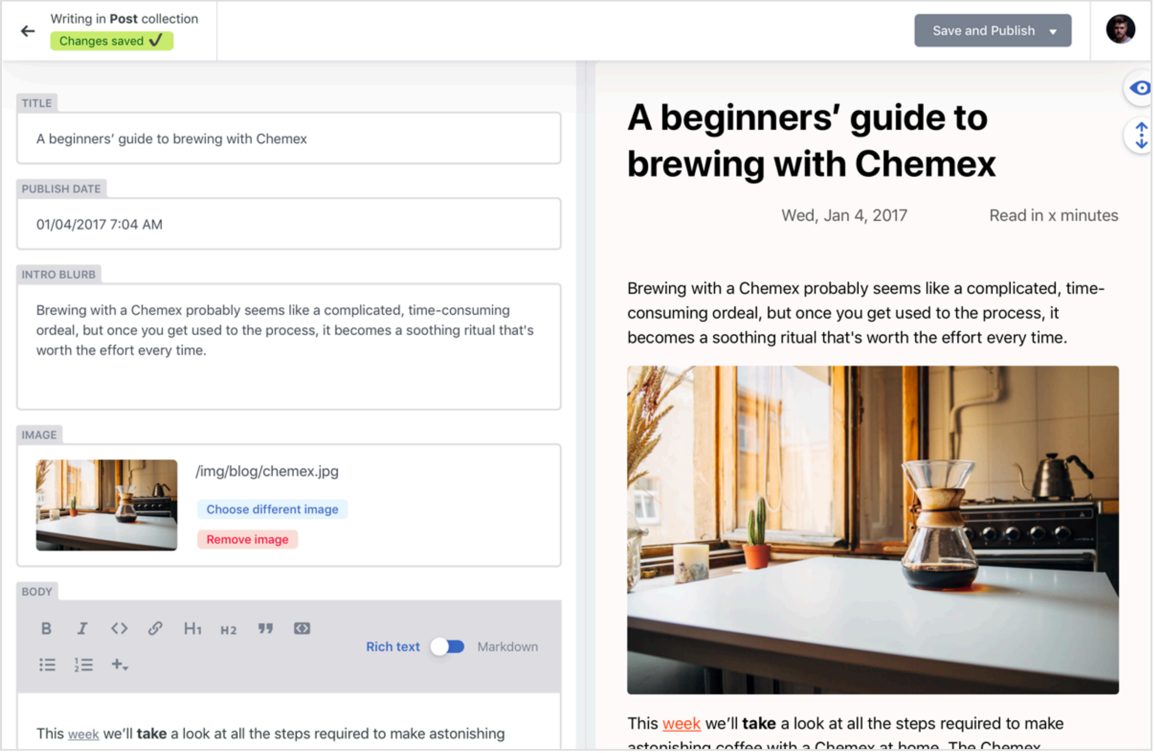
The idea behind the CMS interface is that most static site generators ostracize less technical users by forcing them into the code. (For example, when I write a post in Jekyll, usually others who look over my shoulder think I’m actually programming, even though I’m just writing posts in Markdown.) The CMS removes this by making the experience much more user-friendly to non-technical people while also still leveraging the openness and flexibility of the static site generator platform.

Forestry.io (unlike CloudCannon) also offers an on-premise enterprise installation so you can host and manage the entire platform behind your company’s firewall.

Netlify CMS

Netlify CMS is similar to Forestry in its offering of a content management system for static site generators. But rather than limiting the static site generators you can use, it provides a more open platform wrapper (built with React but using Git to manage the content) that integrates with any static site generator.

One of Netlify CMS’s key advantages is in simplifying the content development experience for less technical users. But you can also standardize your authoring through the interface. Netlify CMS lets you map the custom fields in your theme to a GUI template, as shown in the image below. These custom fields reduce the chance that authors might use the wrong frontmatter tag in their pages (for example, `intro_blurb` or `IntroBlurb` or `introBlurb`). Here’s a sample interface:



TITLE

A beginners' guide to brewing with Chemex

PUBLISH DATE

01/04/2017 7:04 AM

INTRO BLURB

Brewing with a Chemex probably seems like a complicated, time-consuming ordeal, but once you get used to the process, it becomes a soothing ritual that's worth the effort every time.

IMAGE

/img/blog/chemex.jpg

BODY

This week we'll **take** a look at all the steps required to make astonishing

A beginners' guide to brewing with Chemex

Wed, Jan 4, 2017 Read in x minutes

Brewing with a Chemex probably seems like a complicated, time-consuming ordeal, but once you get used to the process, it becomes a soothing ritual that's worth the effort every time.

This week we'll **take** a look at all the steps required to make astonishing coffee with a Chemex at home. The Chemex

Netlify lets you create a user interface for your custom fields.

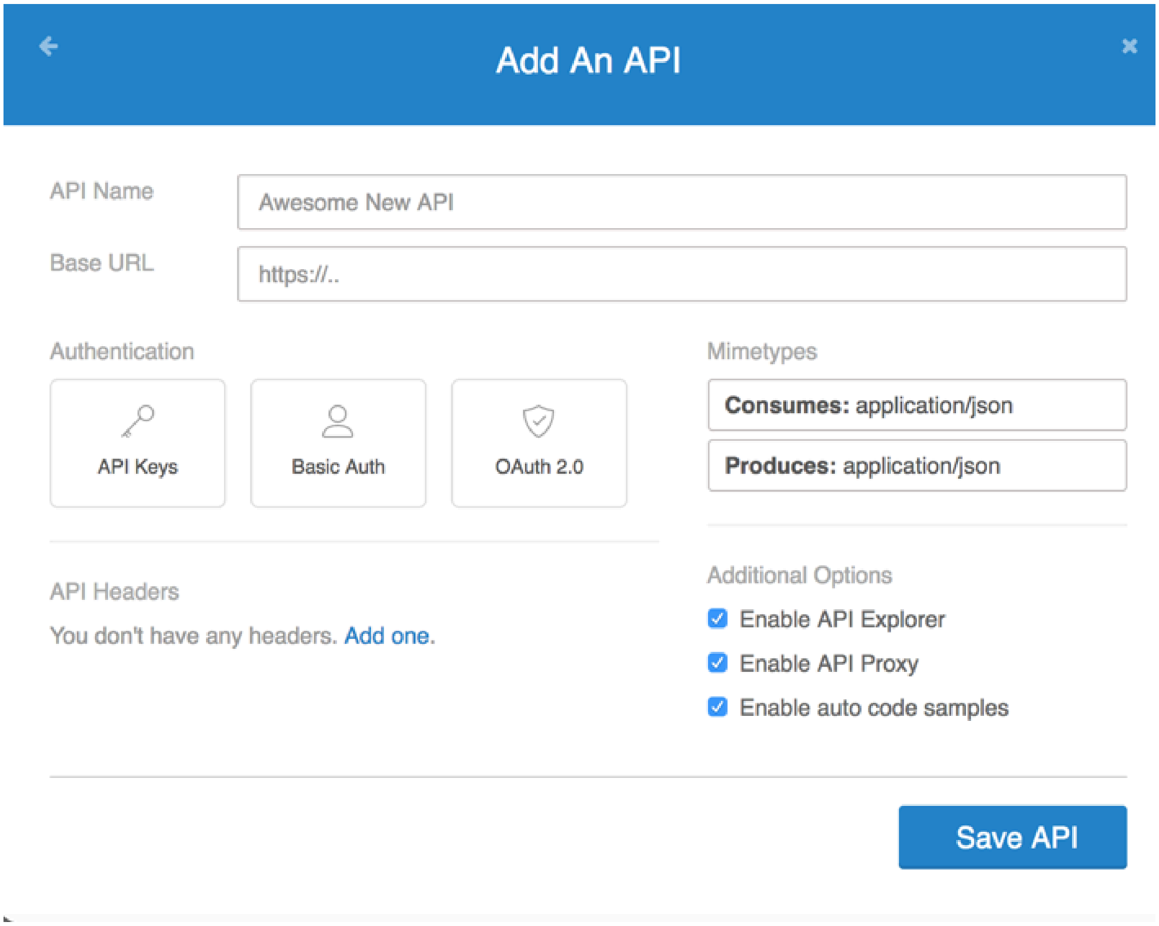
Your content source can be stored in GitHub, GitLab, or BitBucket. Netlify CMS also integrates with Netlify (p. 605), which is a popular hosting and deployment service for static site projects.

For a tutorial on integrating Jekyll with Netlify CMS, see Adding a CMS to Your Static Site With Netlify CMS. Or just start with the Netlify CMS documentation.

Readme.com

Readme.com is an online CMS for docs that offers one of the most robust, full-featured interfaces for developer docs available. Readme.com isn't a headless CMS, meaning you don't just point to your GitHub repo to pull in the content. Instead, I believe Readme.com stores content in a database (though this detail isn't mentioned on their site). So strictly speaking, I probably shouldn't include Readme.com here, but I didn't know where else to describe it, and the CMS-like GUI makes it the most similar to the headless CMS.

Readme.com's emphasis is on providing an interface that helps you more easily write documentation based on best practices and designs. Readme.com provides various wizard-like screens to move you through documentation processes, prompting you with forms to complete. For example:



API Name: Awesome New API

Base URL: https://..

Consumes: application/json

Produces: application/json

You don't have any headers. Add one.

☑ Enable API Explorer

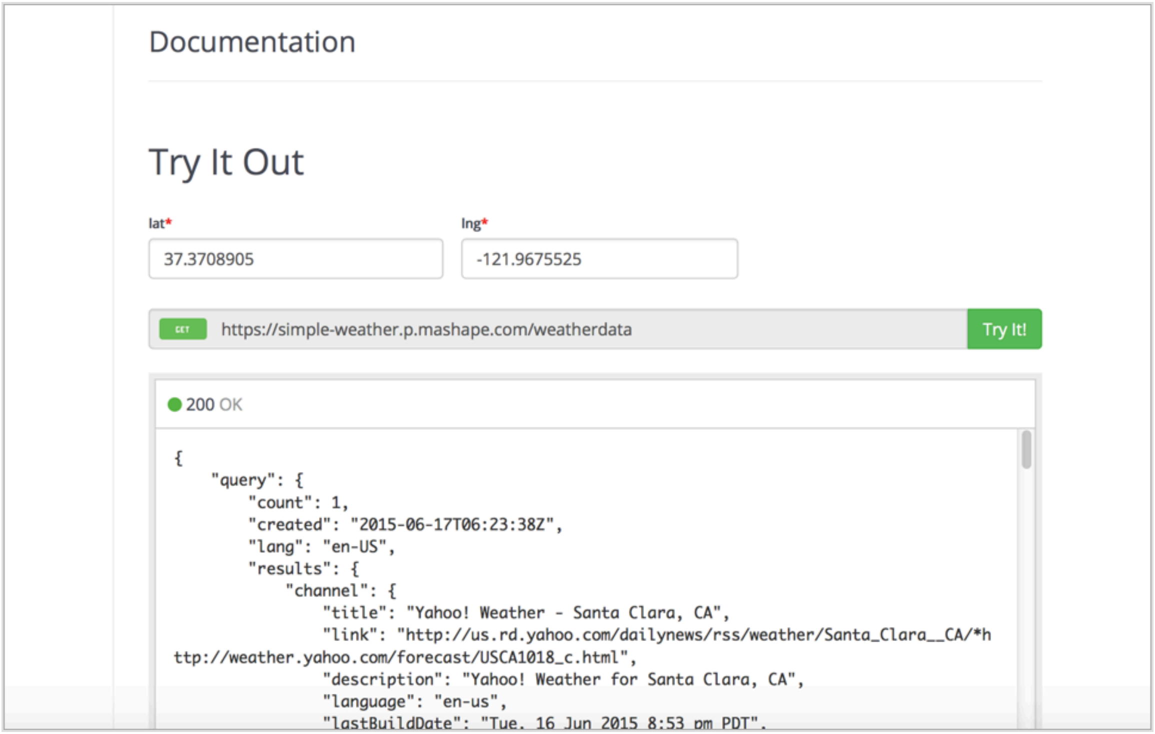
☑ Enable API Proxy

☑ Enable auto code samples

Readme.com

Most importantly, Readme.com includes specific features for displaying API documentation content, which puts it into a class of its own. Although you can add your API information manually, you can also import an OpenAPI specification file (p. 181). You can experiment by choosing one from the OpenAPI examples, such as this one. Readme.com's integration of OpenAPI along with other doc content helps integrate outputs that are often separated. (This fragmentation is a problem I explore later in Integrating Swagger UI with the rest of your docs (p. 241).)

Overall, Readme.com provides a robust GUI for creating API documentation in a way that is more extensive and well-designed than virtually any other platform available. The output includes an interactive, try-it-out experience with endpoints:



Documentation

Try It Out

lat*: 37.3708905

lng*: -121.9675525

https://simple-weather.p.mashape.com/weatherdata

```
{
    "query": {
        "count": 1,
        "created": "2015-06-17T06:23:38Z",
        "lang": "en-US",
        "results": {
            "channel": {
                "title": "Yahoo! Weather - Santa Clara, CA",
                "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*h
ttp://weather.yahoo.com/forecast/USCA1018_c.html",
                "description": "Yahoo! Weather for Santa Clara, CA",
                "language": "en-us",
                "lastBuildDate": "Tue, 16 Jun 2015 8:53 pm PDT",
```

The experience is similar to Swagger in that the response appears directly in the documentation. This API Explorer gives you a sense of the data returned by the API.

There are some challenges with Readme.com. It isn't free, so you'll need licenses per author. Additionally, there isn't any content re-use functionality (currently), so if you have multiple outputs for your documentation that you're single sourcing, Readme.com may not be for you. Finally, if you want to customize your own design or implement a feature not supported, you can't just hack the code (though you can adjust the stylesheet). Overall, with a hosted solution like Readme.com, you're stuck within the platform's constraints.

Even so, the output is sharp, and the talent behind this site is top-notch. The platform is continuously growing with new features, and there are many high-profile companies with their docs on Readme. If you consider how much time it takes to build and deploy your own doc solution, going with a site like Readme.com will save you a lot of time. It will let you focus on your content while also adhering to best practices with site design.

Here are a few sample API doc sites built with Readme.com:

- Validic
- Box API
- Coinbase API
- Farmbase Software

Which tool should you use? I provide some more concrete recommendations in Which tool to choose for API docs – my recommendations (p. 633).

Using Oxygen XML with docs-as-code workflows

Oxygen XML is a robust authoring and publishing tool for technical content that allows you to author in multiple formats (Markdown, HTML, or XML) as well as publish to multiple outputs (HTML-based webhelp, PDF, and more). Although traditionally used for XML authoring and publishing, Oxygen XML has expanded its support with Markdown files, especially with the DITA's recent support for Lightweight Markdown. In this tutorial, you'll learn more about Oxygen XML, different workflows you can use to publish in a docs-as-code model, Git integration with Oxygen XML, supported Markdown formats, how to get started, and more.

Note: Oxygen XML is one of the sponsors of my site.

Publishing workflows with DITA and Markdown in Oxygen XML

Although Oxygen XML is most well-known for XML authoring and publishing, Oxygen XML also supports Markdown authoring and publishing. Some of the Markdown support in Oxygen XML is entirely unique among Markdown tools. You can use at least three distinct authoring and publishing workflows with Oxygen XML:

- Workflow #1: Combine DITA and Markdown in the same project (p. 612)
- Workflow #2: Convert Markdown to DITA and publish (p. 612)
- Workflow #3: Convert DITA to Markdown and publish with a static site generator (p. 613)

Workflow #1: Combine DITA and Markdown in the same project

With Oxygen XML, you can reference Markdown files alongside DITA files in your DITA maps. (A DITA map lists the files included in your output and defines the sidebar navigation.) This means you can use Markdown for some content, and DITA for other content, all within the same project. For example, the DITA map might include these references to topics:

```
<topicref href="files/topic1.dita"/>
<topicref href="files/topic2.md" format="markdown"/>
```

Why would you want to mix and match format types within the same project? You might have engineers write in Markdown and technical writers write in DITA. Technical writers often have to handle more complex publishing challenges (for which DITA might be more suitable), while external contributors might want simpler needs that can be handled with Markdown alone.

Workflow #2: Convert Markdown to DITA and publish

Another workflow would be to convert Markdown files to DITA syntax. With this workflow, contributors could submit content in Markdown, and tech writers could convert the content into DITA as they pull the content into their projects and then use more advanced reuse features.

This workflow would require tech writers to convert the Markdown files to DITA each time. If the contributor has a later edit and wants to work in Markdown, tech writers could convert the DITA file back to Markdown for the contributor.

Workflow #3: Convert DITA to Markdown and publish with a static site generator

Another workflow could be to publish DITA to Markdown using the DITA Open Toolkit. After obtaining Markdown, you would publish the files using a static site generator (e.g., Hugo, Jekyll, Elevanty, Docusaurus, Gatsby) as the publishing engine.

This publishing engine might be preferable if you have a UX team who wants an extremely customized doc website to match other branding, or perhaps if you have customized templates and Swagger integration for your API reference documentation. Some static site generators like Gatsby offer a robust React-based framework that some UX front-end engineers might prefer to work in, rather than customizing the Oxygen XML webhelp output. Then again, if you don't have UX and engineering support, creating your own custom doc-themed website might not be appealing to you.

Even within XML projects, many teams use OxygenXML as their editor only (not to also publish). They use the editor to author and edit content, but then connect with another system to handle the publishing.

Markdown support

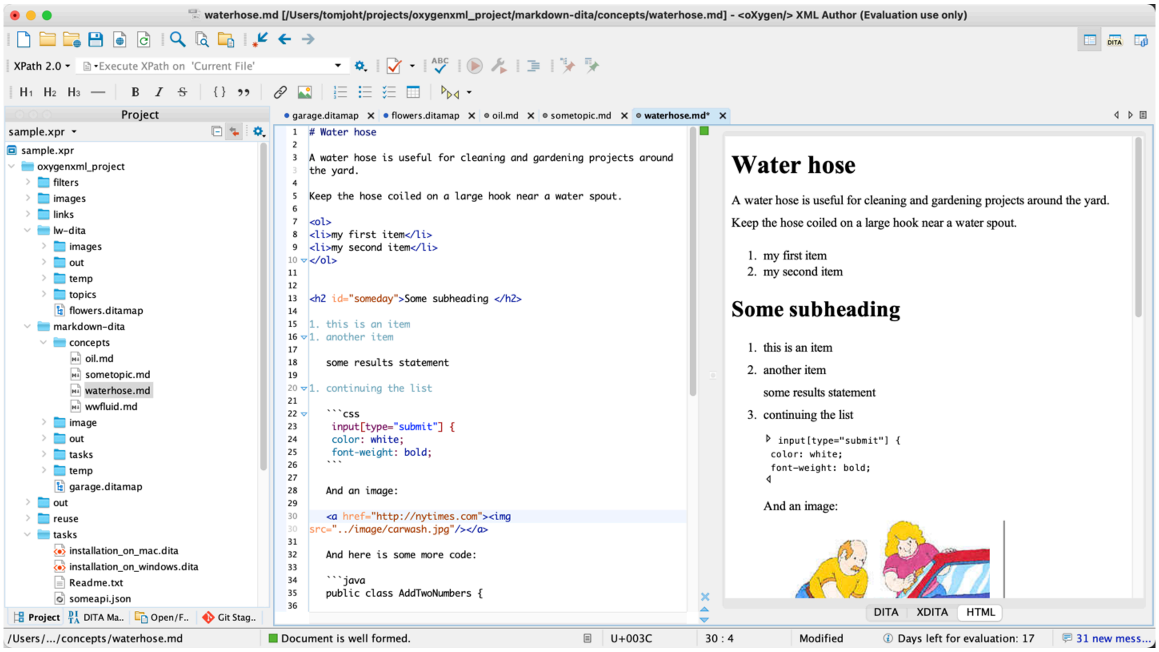
Because Markdown is the most common format for docs-as-code tools, let's look in depth at the Markdown authoring experience with Oxygen XML.

Markdown formats supported in Oxygen XML

Oxygen XML provides support for CommonMark Markdown (the same Markdown standard used by the OpenAPI spec) as well MDITA files (Markdown DITA) from the Lightweight DITA (LwDITA) spec. The MDITA format follows GitHub-Flavored Markdown (for example, fenced code blocks with a language identifier) and also incorporates some re-use features from DITA (primarily `keyrefs` and `conrefs`) — see the supported MDITA syntax here. The `keyref` element allows you to use variables in your content, and `conref` allows for content re-use (also called transcludes, where you can insert the content from one file into another).

Markdown preview in Oxygen XML's editor

As you're writing Markdown in Oxygen XML, a preview pane appears to the right of the code showing how the content will appear in HTML, including image rendering. The preview allows you to view the content you're working on without rebuilding your entire project to check format rendering. This preview also provides real-time validation checking to make sure your Markdown syntax is correct. Here's what the Oxygen XML editor looks like:



~~~
# Water hose

A water hose is useful for cleaning and gardening projects around the yard.

Keep the hose coiled on a large hook near a water spout.

<ol>
<li>my first item</li>
<li>my second item</li>
</ol>


<h2 id="someday">Some subheading </h2>

1. this is an item
1. another item

   some results statement

1. continuing the list

   ```css
    input[type="submit"] {
    color: white;
    font-weight: bold;
   ~~~

   And an image:

   <a href="http://nytimes.com"><img src="../image/carwash.jpg"/></a>

   And here is some more code:

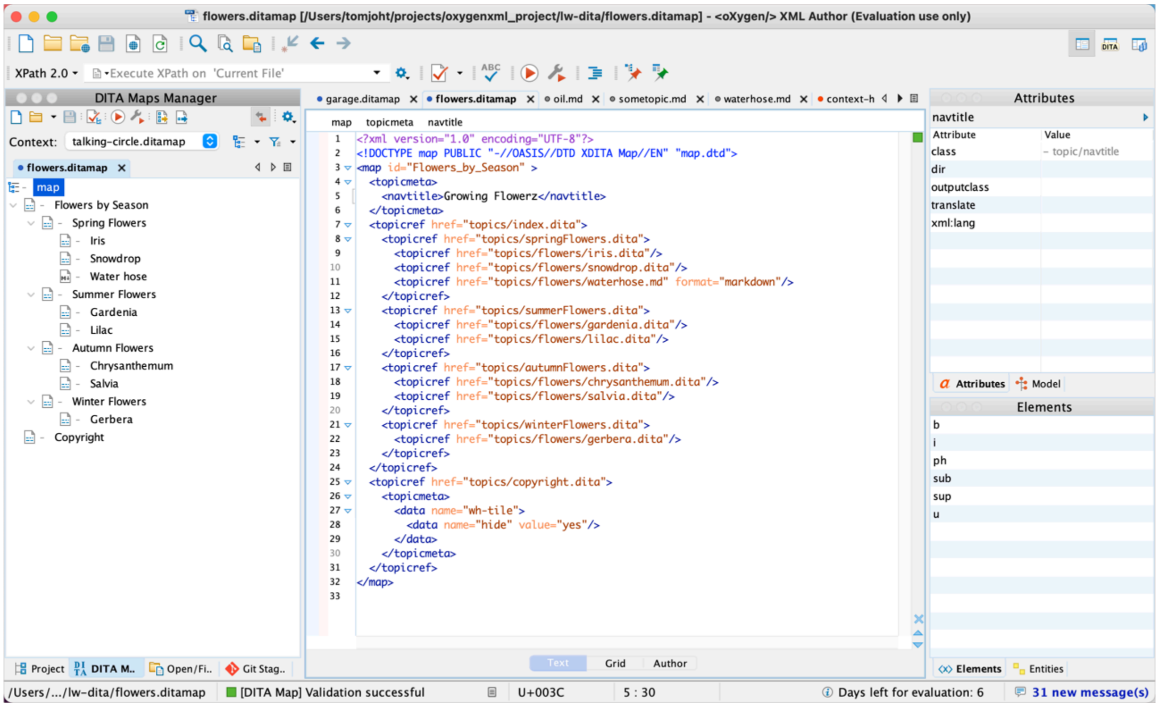
   ```java
   public class AddTwoNumbers {
```

__Figure 1.__ The left pane shows your Project files. The middle pane shows the text editor. The right pane shows the Markdown preview.

If you want, you can turn off the HTML preview by right-clicking the preview pane and choosing Hide preview. (You toggle it back on by right-clicking the main window and choosing Show Preview.) You might disable the preview to give you more space to work in the editor, and also to speed up file loading times. Oxygen XML's editor has a lot of panels that you can undock and rearrange as desired.

In the code view, syntax highlighting colors different parts of the Markdown text to make it more readable. Even some code languages have syntax highlighting directly in the Markdown editor. (The output has full syntax highlighting across code languages.)

In the Project pane on the left, you can also switch to the DITA Map Manager to visually arrange your DITA map, which determines your sidebar navigation. The following screenshot shows how you can either work directly with the DITA map using XML tags or by dragging and dropping the topics in the hierarchical arrangement using the DITA Maps Manager. The Manager gives you a more visual tool to see how topics are nested.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD XDITA Map//EN" "map.dtd">
<map id="Flowers_by_Season" >
  <topicmeta>
    <navtitle>Growing Flowerz</navtitle>
  </topicmeta>
  <topicref href="topics/index.dita">
    <topicref href="topics/springFlowers.dita">
      <topicref href="topics/flowers/iris.dita"/>
      <topicref href="topics/flowers/snowdrop.dita"/>
      <topicref href="topics/flowers/waterhose.md" format="markdown"/>
    </topicref>
    <topicref href="topics/summerFlowers.dita">
      <topicref href="topics/flowers/gardenia.dita"/>
      <topicref href="topics/flowers/lilac.dita"/>
    </topicref>
    <topicref href="topics/autumnFlowers.dita">
      <topicref href="topics/flowers/chrysanthemum.dita"/>
      <topicref href="topics/flowers/salvia.dita"/>
    </topicref>
    <topicref href="topics/winterFlowers.dita">
      <topicref href="topics/flowers/gerbera.dita"/>
    </topicref>
  </topicref>
  <topicref href="topics/copyright.dita">
    <topicmeta>
      <data name="wh-tile">
        <data name="hide" value="yes"/>
      </data>
    </topicmeta>
  </topicref>
</map>
```

Figure 2. *The DITA Maps Manager on the left lets you more easily arrange your topics into the sidebar hierarchy you want.*

The ability to visually arrange the topics in the DITA Maps Manager is a huge convenience because you can see at a glance how the sidebar navigation will appear, without building your output. Some other docs-as-code tools rely on YAML frontmatter tags or a YAML file itself to manage sidebar navigation, which can be prone to error and extremely fussy with syntax, especially when you have hundreds of topics in your project. (For example, one extra space in a YAML file can cause the required indentation to be malformed.)

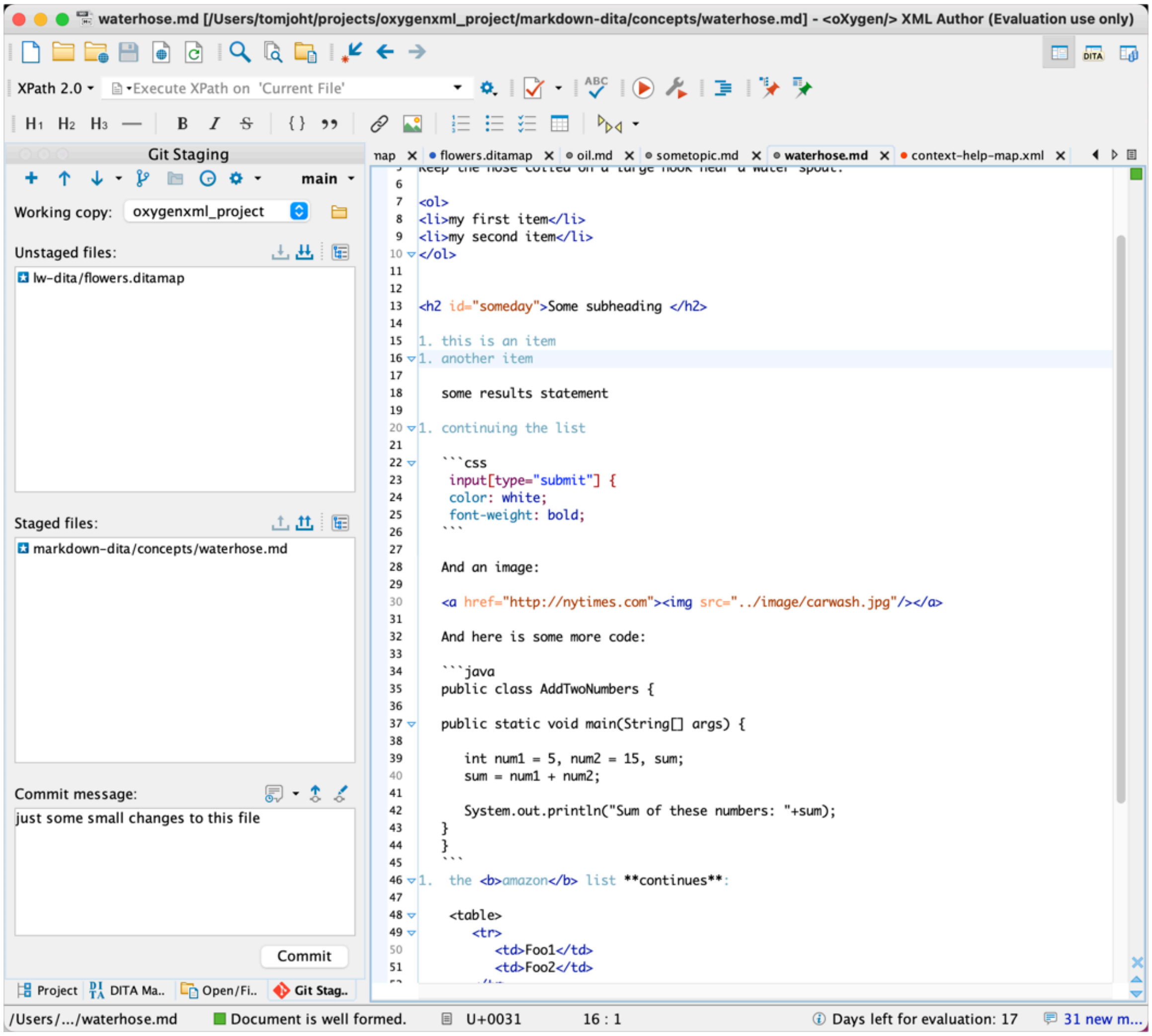
Overall, Oxygen XML lets you choose either a direct text editing experience (working in the code) or a visual editor experience, depending on your preference. You can switch back and forth between modes seamlessly, saving a file to see the updates populate into the other view.

Git client integration

Because the Oxygen XML files are all text files, you can manage the content in Git. (In contrast, binary files that are machine-readable can't be managed in Git.) Although you could manage your Oxygen XML files with Git using your external terminal, Oxygen XML also provides integration with Git directly in the Oxygen XML editor through the Git Client Add-on ↗.

Staging and committing files

When you add the Git Client Add-on, a new Git Staging panel appears in the left pane:



Working copy: oxygenxml_project

Unstaged files:
lw-dita/flowers.ditamap

Staged files:
markdown-dita/concepts/waterhose.md

Commit message:
just some small changes to this file

~~~
Keep the hose coiled on a large hook near a water spout.

<ol>
<li>my first item</li>
<li>my second item</li>
</ol>


<h2 id="someday">Some subheading </h2>

1. this is an item
1. another item

   some results statement

1. continuing the list

   ```css
    input[type="submit"] {
    color: white;
    font-weight: bold;
   ~~~

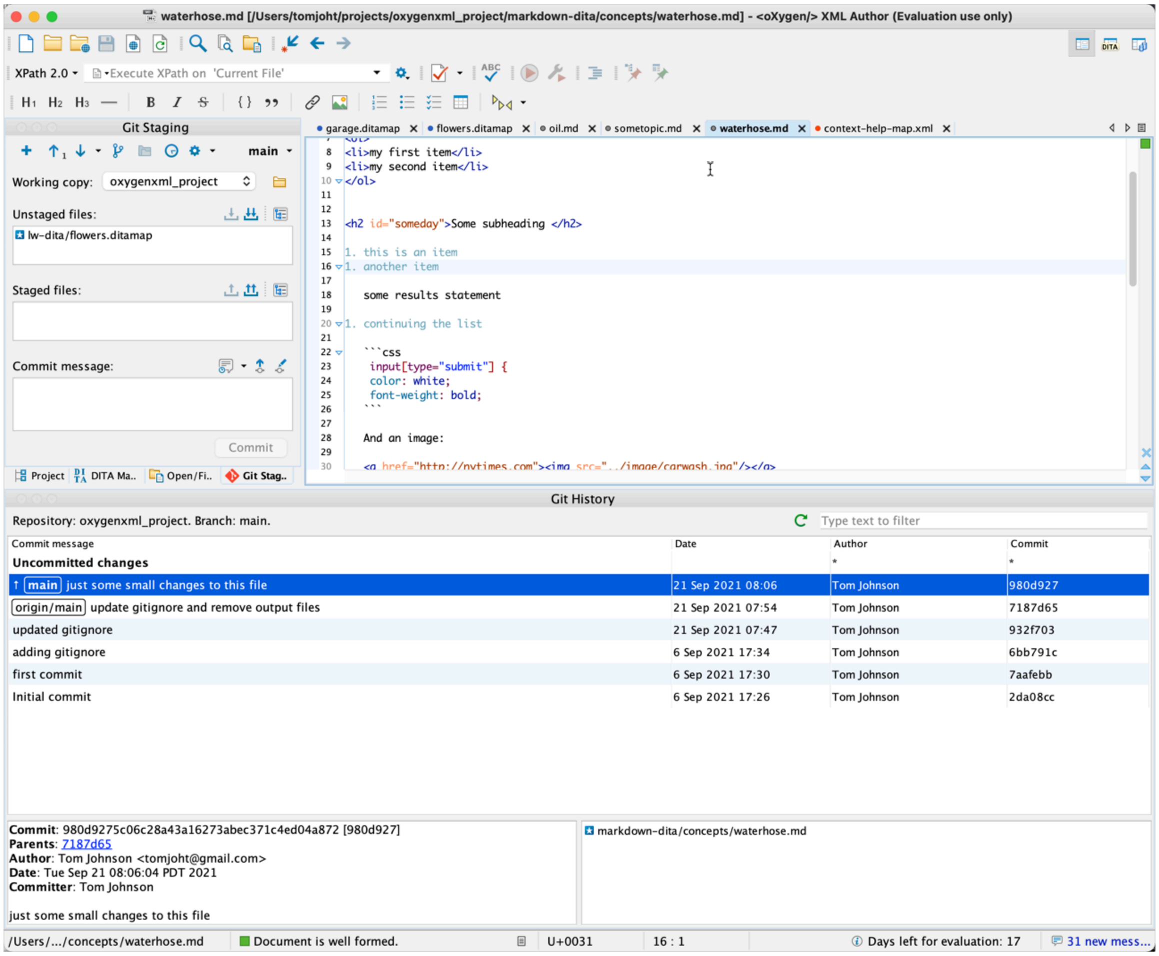
   And an image:

   <a href="http://nytimes.com"><img src="../image/carwash.jpg"/></a>

   And here is some more code:

   ```java
   public class AddTwoNumbers {

   public static void main(String[] args) {

      int num1 = 5, num2 = 15, sum;
      sum = num1 + num2;

      System.out.println("Sum of these numbers: "+sum);
   }
   }
   ```
1. the <b>amazon</b> list **continues**:

    <table>
       <tr>
          <td>Foo1</td>
          <td>Foo2</td>
```

*Figure 3.* *A Git client add-on directly integrates into Oxygen XML so you can more easily manage your files with Git.*

The “Unstaged files” section shows changes you’ve made to files that haven’t been added to the staging area in Git. After selecting the files to stage, click the Stage selected arrow ⤓ to move it into your staging directory. (Moving these files into the staging area is the equivalent of running `git add <filename>` in the terminal.) The file then appears under Staged files and will be pushed to your Git repository on the next commit.

To commit the files, type a commit message and click Commit. (This is the equivalent of typing `git commit -m 'some message'` in the terminal.)

## Viewing Git history

Oxygen XML also has additional views for seeing Git history, managing Git branches, and more. For example, to look at the history of commits, go to Window > Show View > Git History. A list of commits then appears in a commit log in the bottom panel:
```



Working copy: oxygenxml_project

Unstaged files:
lw-dita/flowers.ditamap

Staged files:

Commit message:

~~~
<li>my first item</li>
<li>my second item</li>
</ol>


<h2 id="someday">Some subheading </h2>

1. this is an item
1. another item

   some results statement

1. continuing the list

   ```css
    input[type="submit"] {
    color: white;
    font-weight: bold;
   ```

   And an image:

   <a href="http://nytimes.com"><img src="../image/carwash.jpg"/></a>
~~~

Repository: oxygenxml_project. Branch: main.

Commit message	Date	Author	Commit
Uncommitted changes		*	*
↑ main just some small changes to this file	21 Sep 2021 08:06	Tom Johnson	980d927
origin/main update gitignore and remove output files	21 Sep 2021 07:54	Tom Johnson	7187d65
updated gitignore	21 Sep 2021 07:47	Tom Johnson	932f703
adding gitignore	6 Sep 2021 17:34	Tom Johnson	6bb791c
first commit	6 Sep 2021 17:30	Tom Johnson	7aafebb
Initial commit	6 Sep 2021 17:26	Tom Johnson	2da08cc

Commit: 980d9275c06c28a43a16273abec371c4ed04a872 [980d927]
Parents: 7187d65
Author: Tom Johnson <tomjoht@gmail.com>
Date: Tue Sep 21 08:06:04 PDT 2021
Committer: Tom Johnson

just some small changes to this file

markdown-dita/concepts/waterhose.md

/Users/.../concepts/waterhose.md | Document is well formed. | U+0031 | 16 : 1 | Days left for evaluation: 17 | 31 new mess...

Figure 4. *Git history shown in the bottom panel. Here you can see the commits that have been made to the project. For example, if something looks unfamiliar in the content, look back through the commits to see who made the change.*

Managing Git branches

There's also a Git Branch Manager view that lets you see all the branches and check out a specific branch. Note that the Git functionality within the Git add-on client in Oxygen XML isn't comprehensive, especially with branch management. For example, while you can create and delete branches, to push these local branches upstream to your remote repository, you'd need to run a command from a terminal (for example, `git push --set-upstream origin branchname`). Also, Oxygen XML doesn't open a terminal directly in the editor. But for the most common Git tasks, the integration is convenient and easy to understand.

Overall, the Git functionality within Oxygen XML brings the most common Git workflows right into your editor, making the authoring experience more seamless and integrated.

More docs-as-code features with Oxygen XML

We've covered only a few aspects of docs-as-code tools, namely, Markdown format with text files, and content management with Git. For more details about using Oxygen XML as a docs-as-code tool, see this webinar: Docs as Code - Documentation Management Inspired by Software Development" presented by Alex Jitianu.

Note: This content doesn’t embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/pubapis_oxygenxml.html] to view the content.

The webinar covers ways to store and manage issues with GitHub, how to automate quality checks, how to set up continuous delivery and publishing with Travis CI, ways to collaborate with editing and reviews, and more.

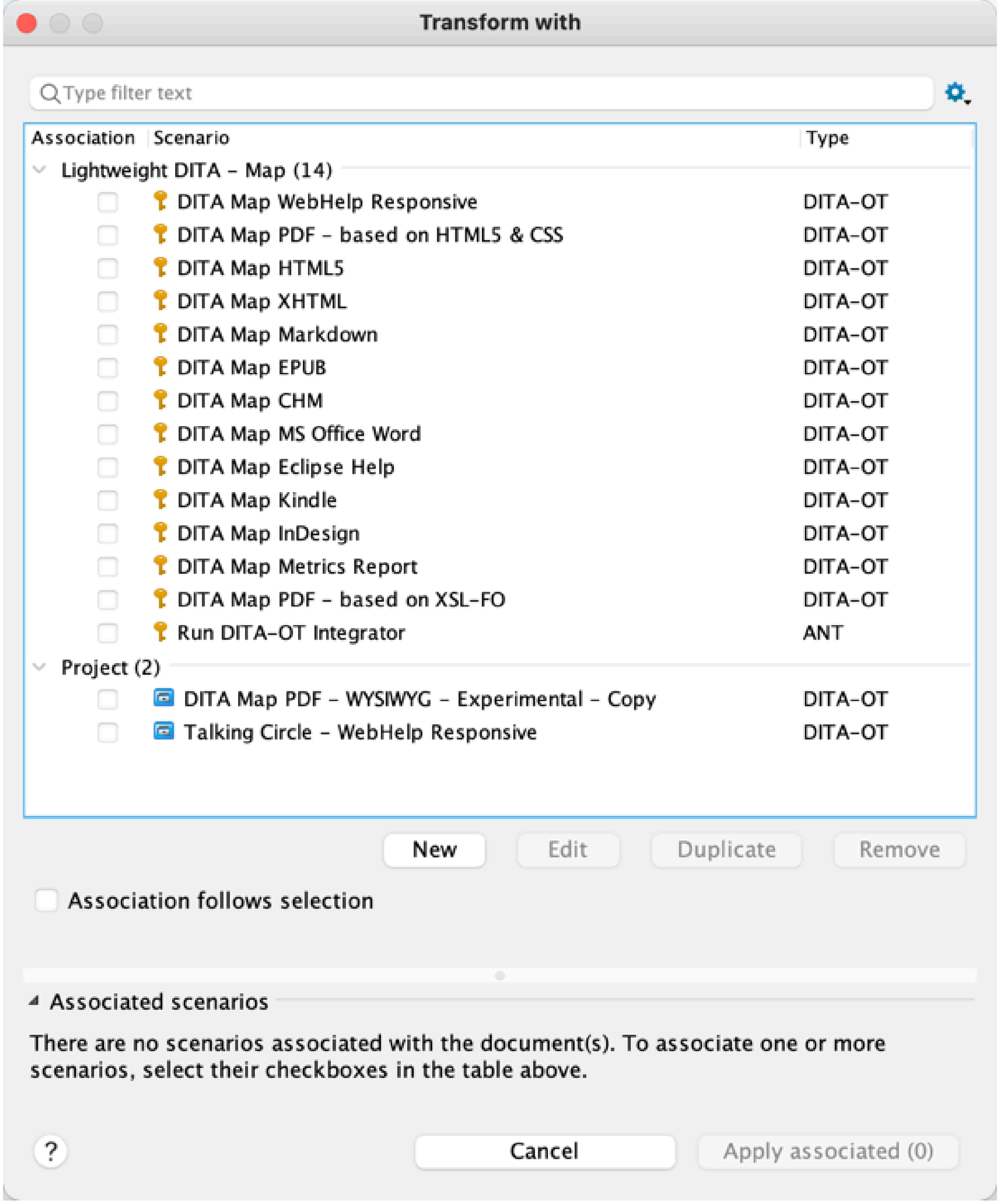
If you’ve never heard the phrase “docs as code,” see Doc-as-code tools (p. 551) for a full description of what this entails. In short, you use the same general tools and workflows to author, manage, and publish documentation as developers use to write code.

Other reasons to explore Oxygen XML

Beyond the docs-as-code features described above, there are other reasons to explore Oxygen XML. Some of these reasons I selected because of my experience in implementing Jekyll while at Amazon (p. 645). Some features didn’t seem like a big deal at the time (I told myself, *we’ll figure it out later*) but they proved to be extremely challenging. These features include customizing the theme, generating PDFs, round-tripping translation workflows, and avoiding the temptation to script everything.

Responsive HTML5 webhelp

One advantage in using Oxygen XML is the many readily available outputs for your content, especially the webhelp output and PDF. If you select a ditamap file in your project and then click the Apply Transformation Scenario(s) button ▶, you can see the available outputs:



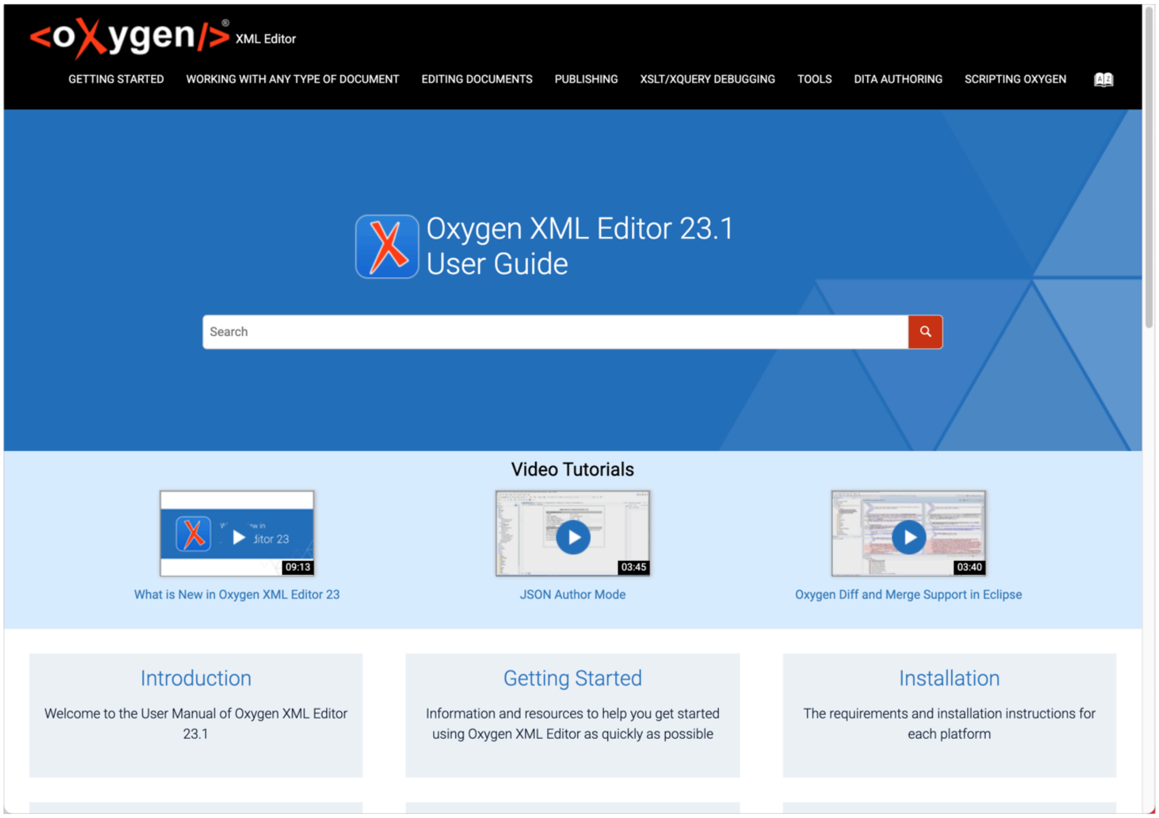
Association	Scenario	Type
Lightweight DITA - Map (14)		
☐	DITA Map WebHelp Responsive	DITA-OT
☐	DITA Map PDF - based on HTML5 & CSS	DITA-OT
☐	DITA Map HTML5	DITA-OT
☐	DITA Map XHTML	DITA-OT
☐	DITA Map Markdown	DITA-OT
☐	DITA Map EPUB	DITA-OT
☐	DITA Map CHM	DITA-OT
☐	DITA Map MS Office Word	DITA-OT
☐	DITA Map Eclipse Help	DITA-OT
☐	DITA Map Kindle	DITA-OT
☐	DITA Map InDesign	DITA-OT
☐	DITA Map Metrics Report	DITA-OT
☐	DITA Map PDF - based on XSL-FO	DITA-OT
☐	Run DITA-OT Integrator	ANT
Project (2)		
☐	DITA Map PDF - WYSIWYG - Experimental - Copy	DITA-OT
☐	Talking Circle - WebHelp Responsive	DITA-OT

Associated scenarios

There are no scenarios associated with the document(s). To associate one or more scenarios, select their checkboxes in the table above.

Figure 5.*Outputs available from Oxygen XML include virtually every type of output you can think of, including webhelp, PDF, Kindle, Word, and more. The two most common outputs are webhelp and PDF.*

The default webhelp output for Oxygen looks pretty good, and you can see an example with Oxygen XML's own documentation.



Oxygen XML Editor 23.1
User Guide

Video Tutorials

What is New in Oxygen XML Editor 23

JSON Author Mode

Oxygen Diff and Merge Support in Eclipse

Introduction

Welcome to the User Manual of Oxygen XML Editor 23.1

Getting Started

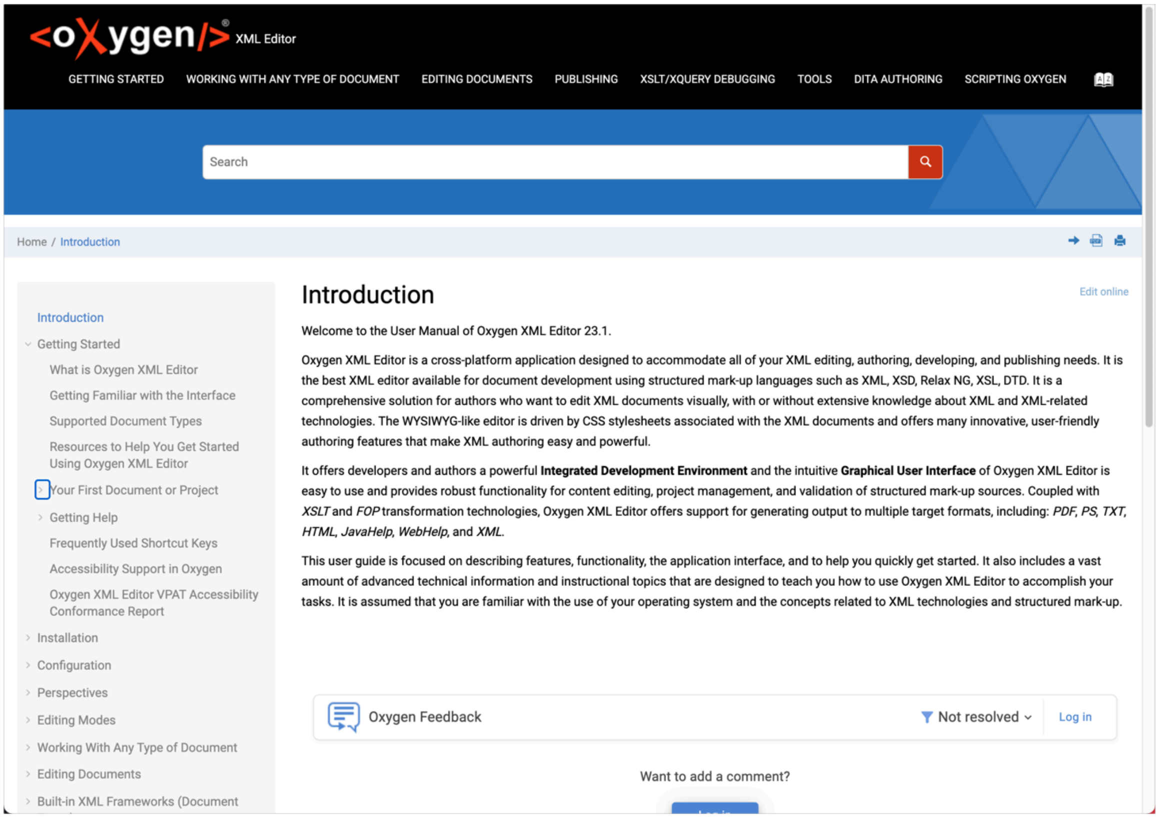
Information and resources to help you get started using Oxygen XML Editor as quickly as possible

Installation

The requirements and installation instructions for each platform

Figure 6.*Oxygen's HTML5 responsive webhelp output. This view shows the landing page, which can showcase different sections of your documentation.*

You can click into a topic area to see its own navigation. Breadcrumbs are included, and if you resize the browser smaller, you can see the responsive elements kick in.



<oXygen/> XML Editor
GETTING STARTED
WORKING WITH ANY TYPE OF DOCUMENT
EDITING DOCUMENTS
PUBLISHING
XSLT/XQUERY DEBUGGING
TOOLS
DITA AUTHORING
SCRIPTING OXYGEN
Search
Home / Introduction
Introduction
Getting Started
What is Oxygen XML Editor
Getting Familiar with the Interface
Supported Document Types
Resources to Help You Get Started Using Oxygen XML Editor
Your First Document or Project
Getting Help
Frequently Used Shortcut Keys
Accessibility Support in Oxygen
Oxygen XML Editor VPAT Accessibility Conformance Report
Installation
Configuration
Perspectives
Editing Modes
Working With Any Type of Document
Editing Documents
Built-in XML Frameworks (Document
Edit online
Oxygen Feedback
Not resolved
Log in

Introduction

Welcome to the User Manual of Oxygen XML Editor 23.1.

Oxygen XML Editor is a cross-platform application designed to accommodate all of your XML editing, authoring, developing, and publishing needs. It is the best XML editor available for document development using structured mark-up languages such as XML, XSD, Relax NG, XSL, DTD. It is a comprehensive solution for authors who want to edit XML documents visually, with or without extensive knowledge about XML and XML-related technologies. The WYSIWYG-like editor is driven by CSS stylesheets associated with the XML documents and offers many innovative, user-friendly authoring features that make XML authoring easy and powerful.

It offers developers and authors a powerful **Integrated Development Environment** and the intuitive **Graphical User Interface** of Oxygen XML Editor is easy to use and provides robust functionality for content editing, project management, and validation of structured mark-up sources. Coupled with *XSLT* and *FOP* transformation technologies, Oxygen XML Editor offers support for generating output to multiple target formats, including: *PDF, PS, TXT, HTML, JavaHelp, WebHelp*, and *XML*.

This user guide is focused on describing features, functionality, the application interface, and to help you quickly get started. It also includes a vast amount of advanced technical information and instructional topics that are designed to teach you how to use Oxygen XML Editor to accomplish your tasks. It is assumed that you are familiar with the use of your operating system and the concepts related to XML technologies and structured mark-up.

Want to add a comment?

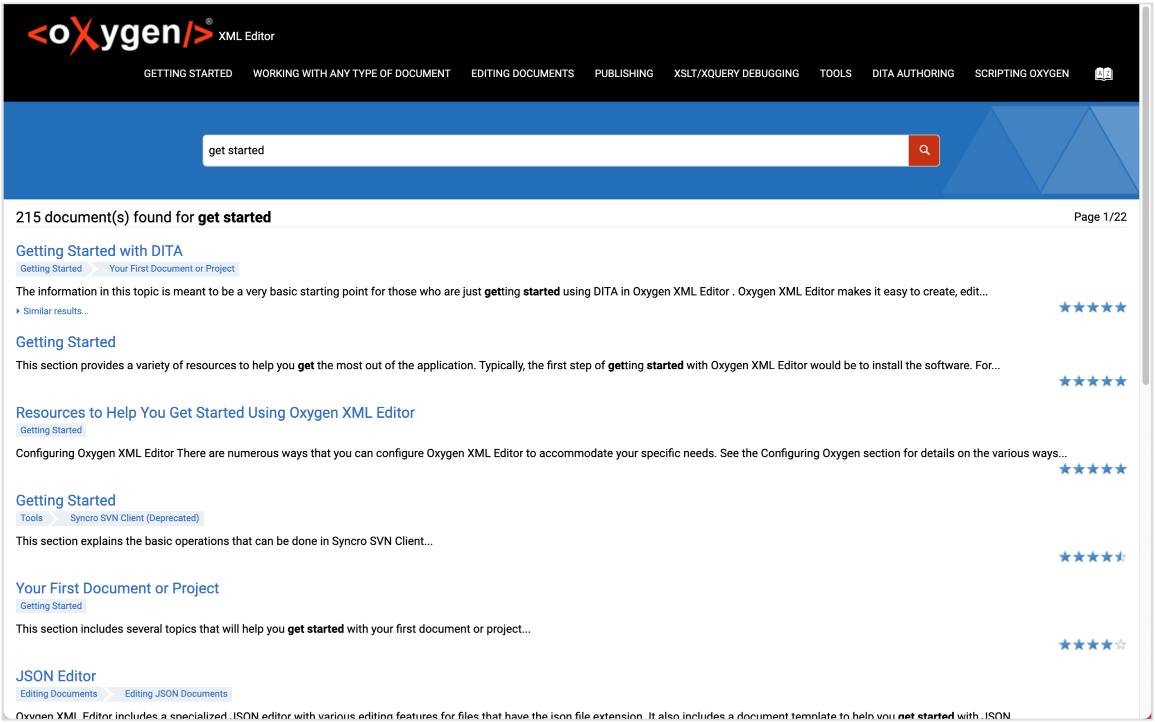
Figure 7. *The webhelp output has a robust sidebar with expand/collapse functionality, breadcrumbs, and more.*

In contrast, if you go with another static site generator, you'll likely have to do a lot of custom UX theming on your own to make the output more documentation-oriented rather than a general website. This customization will require you to be familiar not only with front-end technologies like JavaScript, CSS, and HTML, but the specific templating language of the static site generator. Usually, for companies that decide to create a custom theme, an engineering team and UX designer create this output. (If tech writers do it, their productivity with writing docs tends to plummet.)

Search

Search can also be a huge challenge with docs-as-code tools. Most static site generators don't include search by default but rather require you to integrate Algolia, Swiftype, or another third-party search service. Some free search engines like Lunr are available, but they are often problematic and don't scale.

With Oxygen XML, you get robust search right out of the box. The search results even include breadcrumbs below topic titles so you can see the context. For example, suppose you search for “get started,” and the results show half a dozen getting started tutorials in your dev portal. In this case, you'd want to see more context about the getting started topics so you can select the right tutorial.



215 document(s) found for **get started** Page 1/22

Getting Started with DITA
Getting Started › Your First Document or Project
The information in this topic is meant to be a very basic starting point for those who are just **get**ting **started** using DITA in Oxygen XML Editor . Oxygen XML Editor makes it easy to create, edit...
Similar results...

Getting Started
This section provides a variety of resources to help you **get** the most out of the application. Typically, the first step of **get**ting **started** with Oxygen XML Editor would be to install the software. For...

Resources to Help You Get Started Using Oxygen XML Editor
Getting Started
Configuring Oxygen XML Editor There are numerous ways that you can configure Oxygen XML Editor to accommodate your specific needs. See the Configuring Oxygen section for details on the various ways...

Getting Started
Tools › Syncro SVN Client (Deprecated)
This section explains the basic operations that can be done in Syncro SVN Client...

Your First Document or Project
Getting Started
This section includes several topics that will help you **get started** with your first document or project...

JSON Editor
Editing Documents › Editing JSON Documents
Oxygen XML Editor includes a specialized JSON editor with various editing features for files that have the json file extension. It also includes a document template to help you **get started** with JSON

Figure 8.*Sample search results page from Oxygen XML's search engine. The search results include breadcrumbs for context.*

If you want to swap in your own search engine, you can replace the default search with a custom search engine.

PDF output

Although it might seem like PDF is an outdated mode for delivering docs, PDF is still needed in many documentation scenarios, especially for providing docs to early access partners. Many times when you're writing docs, you need to send the topic to an early access partner, excluding the topic from the rest of your docs until the feature is ready. Other times, PDFs are needed to send to partners who haven't signed contracts yet (and hence can't get access to your developer portal) but who want to preview the docs to gauge the level of integration complexity.

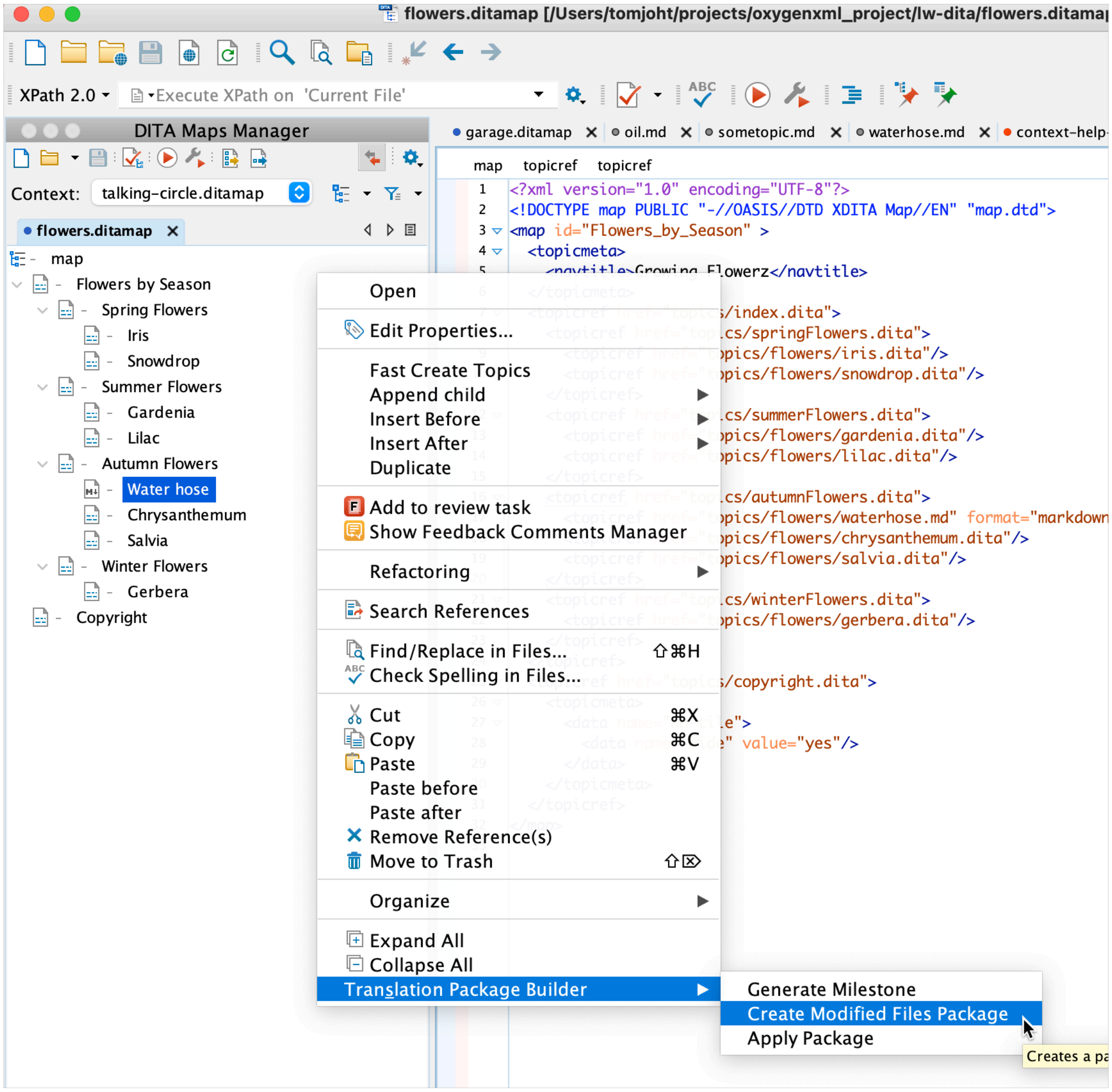
Integrating PDF output in other docs-as-code tools typically involves using external tools such as Prince or open-source PDF generators (which require their own styling and configuration). This can be a huge headache and hassle to deal with. With Oxygen XML, the PDF is included already and attractively styled.

If you want to customize the PDF, you can do so using CSS, which is more accessible and familiar, rather than XSLT. The Oxygen Styles Basket allows you to build your own CSS skin used for both PDF and webhelp output.

Translation support

Finally, one often overlooked but significant feature is translation. If you anticipate the need to translate your content, consider how you will handle this workflow long before you decide on your authoring tool. A cumbersome translation workflow can be a huge time drain, especially if you have goals to keep your translated content in sync with your primary language. Translation is usually where static site generators hit their first major stumbling block.

Translation processes and requirements vary depending on the content and vendor, but the general process for translating content involves bundling up the files you want translated (either XML, HTML, or Markdown format), sending them off to the vendor, and then getting these same files back, but with translated content. You then need to reimport the translated content back into your project. With Oxygen XML, you can manage translation tasks using the DITA Translation Package Builder Add-on.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD XDITA Map//EN" "map.dtd">
<map id="Flowers_by_Season" >
  <topicmeta>
    <navtitle>Growing Flowerz</navtitle>
                      /index.dita">
                      cs/springFlowers.dita">
                      pics/flowers/iris.dita"/>
                      pics/flowers/snowdrop.dita"/>
                      cs/summerFlowers.dita">
                      pics/flowers/gardenia.dita"/>
                      pics/flowers/lilac.dita"/>
                      cs/autumnFlowers.dita">
                      pics/flowers/waterhose.md" format="markdown
                      pics/flowers/chrysanthemum.dita"/>
                      pics/flowers/salvia.dita"/>
                      cs/winterFlowers.dita">
                      pics/flowers/gerbera.dita"/>
                      /copyright.dita">
                      e">
                      " value="yes"/>
```

Figure 9. Translation Manager contextual menu in Oxygen XML

When you install the add-on in Oxygen XML, you can create “milestones” that are snapshots in time for your project that future changes are compared against. After creating a milestone, you can package up only the files changed after that milestone, or you can package up all files. Most translation vendors have translation memory in their databases that will only detect changes. (However, practices and support for translation memory vary, and you might want to just send the changed files.)

If you make use of the `conref` (content reuse) and `keyref` (naming variables) in your content, you might reduce translation costs. But also, because Oxygen XML enforces the language standards, when you submit the content to the vendor, they won't require handholding with custom code. The ability to quickly package up all content files can streamline your translation workflows.

Standard Markdown and HTML formats

Many static site generators (p. 591) (Jekyll, Hugo, Vuepress, Eleventy, Gatsby, Docusaurus, and more) use a scripting language (for example, Liquid, Jinja) that you can also use in your Markdown or HTML pages to create custom functions or templates. For example, with Jekyll, you can store data in YAML files and then build Liquid functions to parse through the data.

Using custom functions in your docs might seem attractive at first, as this gives you more flexibility to build custom solutions around your content. However, with time, these projects become filled with custom code by many different writers, with convoluted logic that becomes forgotten, and pretty soon you're locked into that tool and theme. For example, when I left Amazon after 5 years, I had to make more than a dozen videos where I attempted to explain what this or that script was doing, why I created it, how to use it, etc.

In contrast, Oxygen XML is more standards-based. (This is part of the whole selling point of XML, in fact.) You write in Markdown, DITA, HTML, or some other standard. And then Oxygen XML validates your content before publishing.

With Oxygen XML, you can do some additional validation on the Markdown formatting through Schematron. Schematron allows you to check your docs against a list of formatting patterns and rules. Oxygen XML includes some basic Schematron rules out of the box (see `/samples/markdown/schematron-validation/` in the sample project), and you can add more as well.

But the larger point here is that your Oxygen XML projects will stick more closely to the standards of the language you're writing in. This can keep your content more consistent and structured, and if you ever decide to use another tool or theme, your standards-based content can easily migrate.

First steps with Oxygen XML

To get started with Oxygen XML, download a 30-day trial. When you open Oxygen XML for the first time, a sample project will load. Explore the files in the Markdown folders. Click the `.ditamap` file and then click the Apply Transformation Scenarios button ▶ and build the output. Explore editing topics within the text editor and previewing the Markdown. If you need help, there are Oxygen XML forums that are actively monitored.

Conclusion

Oxygen XML provides robust support for nearly every documentation scenario and is built from the ground up for technical writers and others creating documentation. Because of that design, you'll feel at home with many of the features and functionality.

The ability to scale and connect into larger systems for managing content is one of the reasons tech pubs groups choose Oxygen XML. Now with the support for Markdown, integration with Git, and other docs-as-code features, Oxygen XML can be a great choice for writing developer docs as well.

Additional resources

Here are some additional resources that might be useful, especially if you’re new to content re-use with Oxygen XML and DITA:

- Getting Started with Oxygen and DITA ↗
- Basic DITA Profiling and Reuse ↗
- Advanced DITA Profiling and Reuse ↗

See also the Oxygen Content Fusion add-on ↗, which allows writers using Oxygen XML Author to create review tasks. The content can be reviewed by subject matter experts directly from a web browser (see a video here ↗).

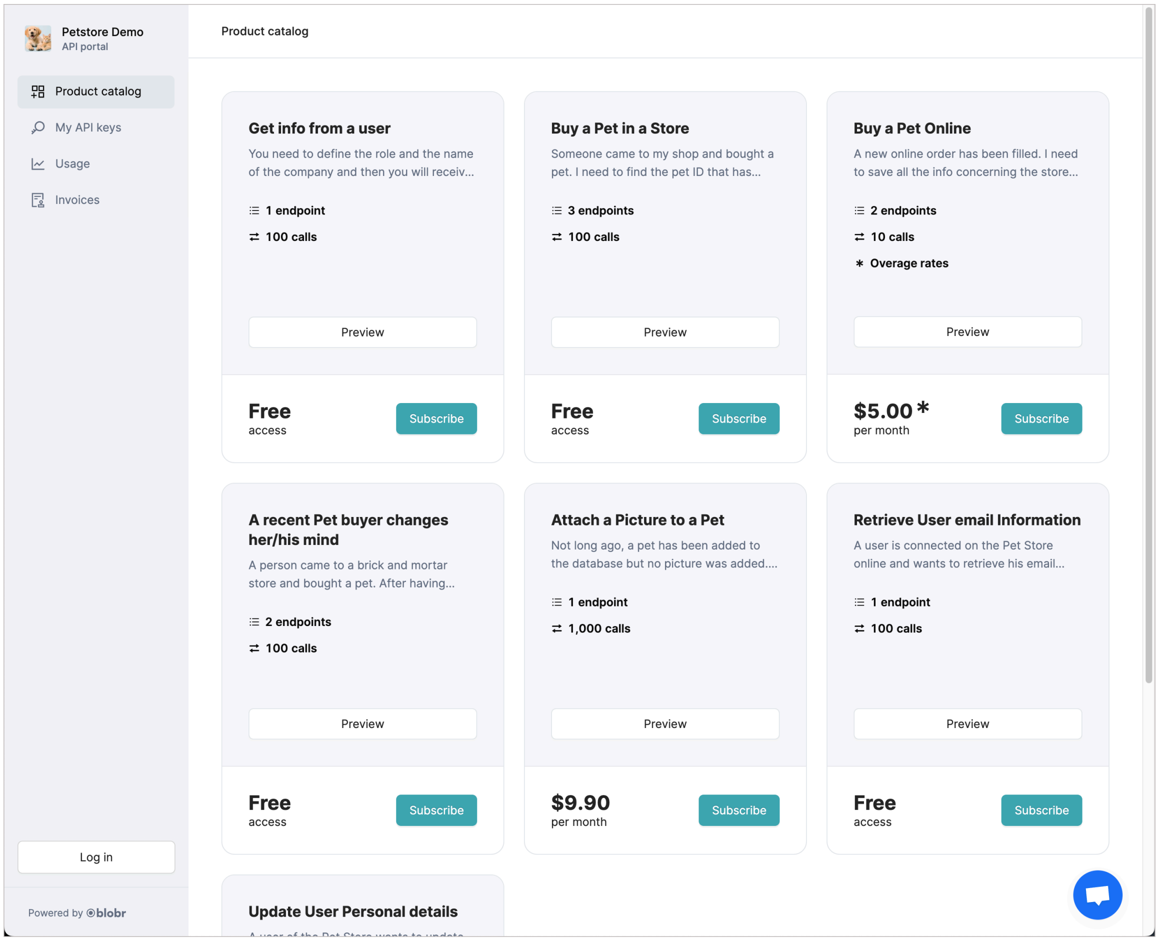
Blobr: An API portal that arranges your API's use cases as individual products

With Blobr, you can create an API store to launch and grow an API business with different monetization models. In the same Blobr portal, you can also include documentation that describes the precise workflow for each use case, helping API consumers easily onboard with your API. You can also customize your portal's name, logo, colors, domain name, add support chat integration, and more.

Note: Blobr is one of the sponsors of my site.

Blobr example

Let's walk through a Blobr example. Here's a product catalog for a Petstore API on Blobr:



Get info from a user
You need to define the role and the name of the company and then you will receiv...
1 endpoint
100 calls
Free access

Buy a Pet in a Store
Someone came to my shop and bought a pet. I need to find the pet ID that has...
3 endpoints
100 calls
Free access

Buy a Pet Online
A new online order has been filled. I need to save all the info concerning the store...
2 endpoints
10 calls
Overage rates
$5.00 * per month

A recent Pet buyer changes her/his mind
A person came to a brick and mortar store and bought a pet. After having...
2 endpoints
100 calls
Free access

Attach a Picture to a Pet
Not long ago, a pet has been added to the database but no picture was added....
1 endpoint
1,000 calls
$9.90 per month

Retrieve User email Information
A user is connected on the Pet Store online and wants to retrieve his email...
1 endpoint
100 calls
Free access

Update User Personal details

Figure 1. *Product catalog overview on the Blobr API portal. Developers can browse the various ways the API can be used ("products") and then preview or purchase the product they want.*

In this example, the use cases for the Petstore API have been packaged into seven different "products":

- Get info from a user

- Buy a pet in a store
- Buy a pet online
- A recent pet buy changes his/her mind
- Attach a picture to a pet
- Retrieve user email information
- Update user personal details

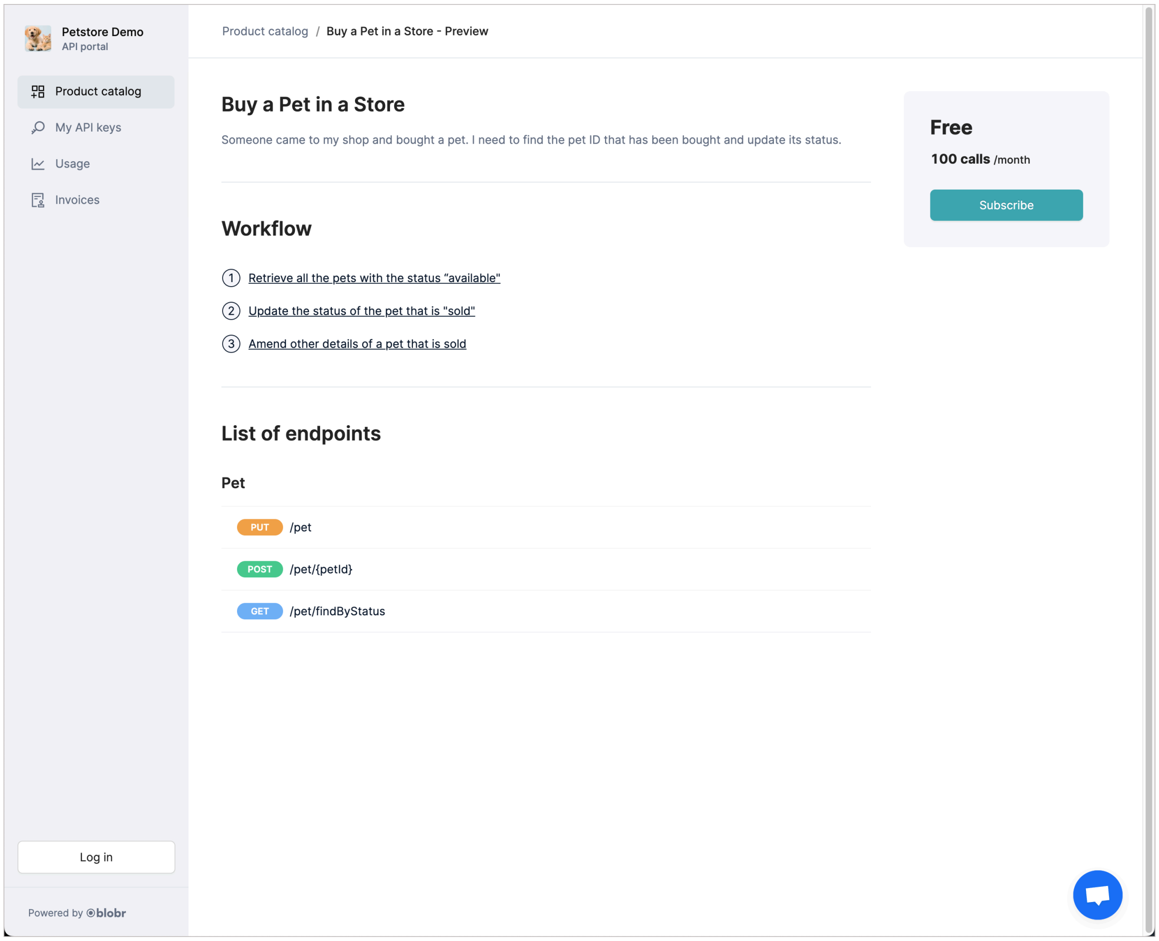
This is what’s ingenious about Blobr — it allows you to take each use case for your API and package it as an individual product, which can then be monetized or made available for free. Each product usually involves using several different API endpoints in a specific workflow.

API consumers (developers or other integrators) choose the products they want to consume, and then they become authorized to use the workflow’s endpoints. For example, a typical authorization might allow developers to make a certain number of calls per month.

Blobr’s new approach to organizing, presenting, and monetizing APIs not only simplifies API consumption for developers, it helps API providers better communicate to consumers how to use the API.

Product and workflow in detail

Let’s look at a product and its workflow in more detail. Suppose you want to implement the use case “Buy a Pet in a Store.” You would click this product to see details on the implementation. The product landing page shows an overview of the product, the cost, and the workflow.



Product catalog / Buy a Pet in a Store - Preview

Buy a Pet in a Store

Someone came to my shop and bought a pet. I need to find the pet ID that has been bought and update its status.

Workflow

1. Retrieve all the pets with the status “available"
2. Update the status of the pet that is "sold"
3. Amend other details of a pet that is sold

List of endpoints

Pet

PUT /pet
POST /pet/{petId}
GET /pet/findByStatus

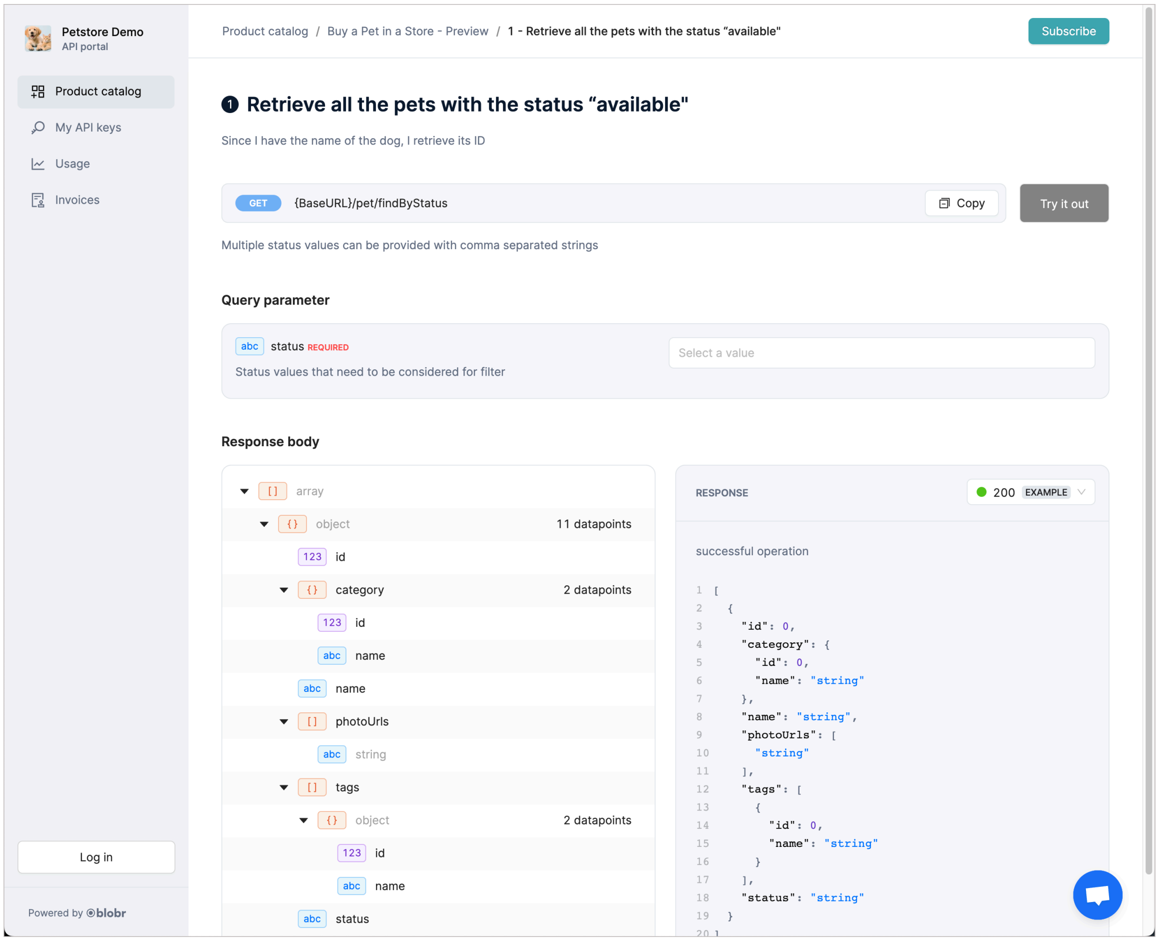
Free

100 calls /month

Figure 2. *Product detail page.This view describes the product and the workflow to implement it. The details also lists the endpoints used for the product.*

In this example, the product description is brief, but a more detailed product could include screenshots and other formatting (see an example).

In addition to a product overview page, documentation appears as workflow steps. The first step in this workflow is to “Retrieve all the pets with the status ‘available’.” Clicking this workflow step reveals more detail, as shown below:



1 Retrieve all the pets with the status "available"

Since I have the name of the dog, I retrieve its ID

GET {BaseURL}/pet/findByStatus

Multiple status values can be provided with comma separated strings

Query parameter

abc status REQUIRED — Select a value

Status values that need to be considered for filter

Response body

- [] array
  - {} object — 11 datapoints
    - 123 id
    - {} category — 2 datapoints
      - 123 id
      - abc name
    - abc name
    - [] photoUrls
      - abc string
    - [] tags
      - {} object — 2 datapoints
        - 123 id
        - abc name
    - abc status

RESPONSE 200 EXAMPLE

successful operation

```
[
  {
    "id": 0,
    "category": {
      "id": 0,
      "name": "string"
    },
    "name": "string",
    "photoUrls": [
      "string"
    ],
    "tags": [
      {
        "id": 0,
        "name": "string"
      }
    ],
    "status": "string"
  }
```

Figure 3. *Workflow step detail page in Blobr. This view describes a workflow step, including the endpoint used, its parameters, and example response.*

The workflow steps usually reference an API endpoint. How do you get your API reference details in there? Although you can enter your API details manually, a more common approach is to upload an OpenAPI document (p. 181). After uploading your OpenAPI document, you can select the endpoints that apply to the workflow step. (The same endpoint might apply across multiple products in an API, and this approach allows you to single-source your reference documentation.)

For each workflow step, Blobr also includes a "Try it out" explorer so that developers can execute the endpoint directly on the page and see the result.

API tracking and monitoring

With Blobr, endpoint usage is tracked in a granular way. API providers can price each product as desired, such as 1,000 calls for $5. As calls get made to the endpoint, a counter shows how many calls remain in the purchased plan.

This granular call tracking allows API providers and consumers to measure API usage. Further, if the workflow involves calling multiple endpoints, you can see how far along the users get in the workflow funnel. In this example, users will most likely make a lot of calls to see the available pets in the petstore and fewer calls to actually buy the pets.

Blobr also allows you to monitor API activity through API logs. Monitoring helps both API providers and consumers to debug, access, and filter call errors, as well as troubleshoot issues in real time. API consumers can also replay their calls directly from the portal.

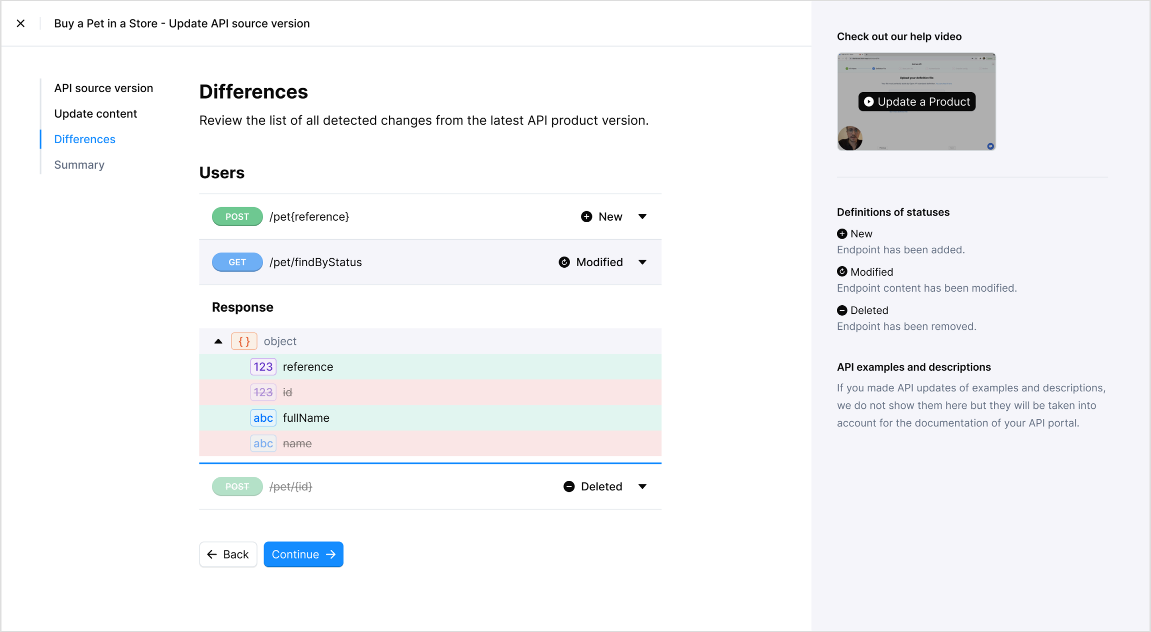
Monetization models

With Blobr, you can choose from different monetization models: subscription based, tiered, or metered models. For each model, the monetization is either based on the number of API calls made or, with custom units, defined by the API provider. The custom units can include the number of users, the amount of data processed, or other options. Flexibility with custom units provides a way to bring the price and value to the user closer together.

If your API isn't monetized at all, you can use Blobr for free with all the features needed to replace your documentation solution or create your own store.

API changelogs and personalized release notes

When there are changes to the API, rather than broadcasting all changes to a general release notes page, the changelogs are delivered on a product-by-product basis so that changes only go to the relevant consumers who have purchased those products. The following screens demonstrate how Blobr shows changes to users:



Differences

Review the list of all detected changes from the latest API product version.

Definitions of statuses

New
Endpoint has been added.

Modified
Endpoint content has been modified.

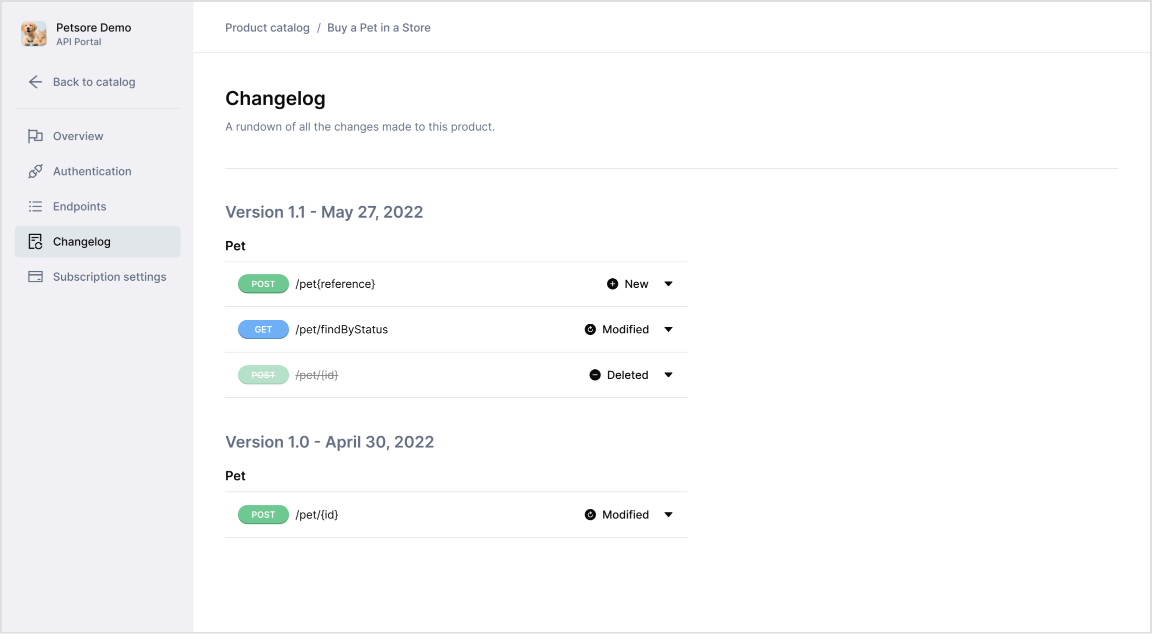
Deleted
Endpoint has been removed.

API examples and descriptions

If you made API updates of examples and descriptions, we do not show them here but they will be taken into account for the documentation of your API portal.

Figure 4. *Product update — differences. This screen shows the differences to API providers when changing the version of an API in a product. Note the tags that read "New," "Modified," and "Deleted." API providers can review everything that has changed from version to version.*

The following screenshot shows the changelog that API consumers see.



Changelog

A rundown of all the changes made to this product.

Version 1.1 - May 27, 2022

Pet

POST /pet{reference} New

GET /pet/findByStatus Modified

~~POST /pet/{id}~~ Deleted

Version 1.0 - April 30, 2022

Pet

POST /pet/{id} Modified

Figure 5. *Product changelog. Those who consuming the product (and who will be impacted by the API changes) can see what's new, modified, or deleted through these changelogs.*

If desired, Blobr can automatically send personalized changelogs to API consumers. These personalized changelogs avoid broader communications in which only a small subset of information is actually relevant to the API consumer.

Reasons for Blobr’s unique approach to API portals

What’s the story behind Blobr’s unique approach to API portals? Alexander Airvault, co-founder and CEO of Blobr, said that previously, when he worked with API customers, they often struggled to understand how to implement the APIs. The APIs were usually more technical than the customers could follow.

Compounding the difficulty, the traditional model of API portals mostly provides a long list of endpoints for the user to figure out on their own. To reduce some of the complexity, Alexandre started creating “subscriptions” to different parts of the API for customers. That’s how he got the idea of Blobr.

Blobr’s approach *productizes* the use cases of your API, making it clear how to use your API to achieve various goals. There’s no ambiguity about what you can do with the API or which endpoints you need to accomplish each goal. This granularity also allows API consumers to focus only on the use cases of the API they actually need.

Dividing the API into distinctly packaged products not only eliminates complexity but avoids information overload for API consumers. The consumers see only the endpoints and workflows related to their product use cases.

One platform that brings it all together

Although we’ve talked about various pieces of the Blobr API portal, let’s step back and consider the platform as a whole. Blobr provides multiple elements that are critical to distribute and monetize APIs. For example, Blobr combines building blocks that include API authentication, limits, monitoring, documentation, monetization, and analytics *all in the same platform*. Blobr gives you workflow API documentation and testing, API monetization linked to real-time usage and authentication, personalized changelogs with each API update, API logs and error replay, and more.

The combination of all of these necessary elements, working together on the same platform, provides a seamless experience for both API providers and consumers through the whole API lifecycle — from pre-sales to implementation, post-sales, and beyond. It’s a case where the overall platform is more valuable than the sum of each of the parts.

The story behind Blobr

Blobr is named after the blob — a unicellular organism best described as slime mold. Neither plant, animal, or fungus, the blob has some unique properties. If you cut a blob in half, each part continues to grow as a discrete organism. Conversely, if you push two blobs together, they merge into a single blob. The blob’s division or unification provides a living example of how APIs can be chunked into separate products, or how the same API endpoint can be merged into multiple products. (Blob’s have other unique properties that also make them an appealing metaphor.)

Blobr is in the startup phase. Based in France, Blobr received VC funding and began commercializing operations in January of 2022. As of May 2022, the company already has more than 100 paying customers. To learn more about Blobr, see https://www.blobr.io/.

Which tool to choose for API docs — my recommendations

I described a smattering of tools in static site generators (p. 591), hosting and deployment options (p. 602), and hybrid solutions (p. 607). Which solution should you choose? It's a complicated decision that will invariably involve tradeoffs. The decision depends on your skill set, product, environment, and requirements. But here's my general recommendation. First, identify what authoring requirements you have. Then decide on a static site generator, and then consider a hosting and deployment platform.

Also, note that I don't have total familiarity with all of these tools and solutions. My core experience with docs-as-code tools involves Jekyll, GitHub Pages, and internally developed publishing pipelines. I have only dabbled or experimented with a lot of these other tools and platforms, so I can't speak authoritatively about them.

Define your requirements

The first step to selecting a tool is to define your authoring requirements. Start by answering the following questions:

- Will engineers be heavily authoring and collaborating on the content?
- Does your security group restrict you from using third-party platforms to host documentation, such as GitHub?
- Do you have existing internal infrastructure that you want to hook into for storing and automatically building your site?
- Do you have engineering resources available to implement your own continuous delivery publishing workflow?
- Do you have a strong familiarity with a particular programming language?
- Approximately how many documentation pages do you have in your project?
- Do you have some web development skills (or access to UX resources) to design or modify your theme?
- Do you have an available budget to pay for a third-party hosting and deployment option?
- How many authors will be authoring directly as contributors in the system?
- Do you need to authenticate documentation for specific users? Is there an existing authentication system already in place at your company?
- Do you need to integrate your docs directly into your larger company site, with the same branding and appearance?
- Do you need to localize the content? If so, how many other languages? Are there formatting requirements imposed by your translation vendor and system?
- Do you need to create PDF deliverables for the content (in addition to web output)?
- How will you review the content with SMEs?
- Do you want a lot of control and flexibility to extend or customize the solution with your specific doc's needs, which might involve time-intensive custom scripting or integration with another system?
- Can you use an external search service such as Swiftype, Algolia, or Google Custom Search?
- To what extent do you need to re-use the same content in multiple instances or outputs?
- Do you have to version your content with each new release?

Now that you've gathered some data about requirements, understand that you're probably not going to find a single system that does all of what you need. There are tradeoffs with every tool choice. The question is which features you want to *prioritize*.

For example, maybe it's more important to reduce the time spent with tools than it is to have custom branding and output. Or perhaps a modern web output is more important than the ability to build PDFs. Or perhaps you must have authentication for your docs, but you also don't have a budget. There are going to be some hard decisions to make.

1. Select a static site generator

If you want power and control to create the complex features you need (maybe you want to build a custom theme or build your doc site with unique branding), then use a static site generator such as Hugo (p. 594), Sphinx (p. 596), or Jekyll (p. 592). If you have serious doc needs (maybe you migrated from the world of DITA and are used to more robust tooling), you're going to want a platform that can go as deep as you want to take it. Jekyll, Sphinx, and Hugo offer this depth in the platform.

Granted, this power and control will require a more complex platform and learning curve, but you can start out easy with a ready-made theme and later work your way into custom development as desired.

If you don't have web development skills and don't want to tinker with the theme or other code development, choose a solution such as Readme.com (p. 609) or Netlify CMS (p. 608) (though, with Netlify CMS, you'd still have to select a theme). Readme provides a ready-made design for your API doc site, removing the need for both designing a theme and figuring out hosting/deployment. That can save you a lot of time and effort.

Realize that when implementing a solution, you might spend a quarter of your time (over a period of months between time on projects) customizing your theme and working on doc tooling. If you don't want to devote that much time to your tooling, Readme is a good option. However, I personally want more control and flexibility over the information design and theme. I like to experiment, and I want the power to code whatever feature I want, such as an embedded navigation map ↗, JS features to collapse/expand elements ↗, custom metadata ↗, or whatever. I think many tech writers and developers want similar flexibility and control. What is important to you? Are flexibility and control so crucial that you're willing to sink weeks/months of time into the solution?

Additionally, if you have a large number of contributing authors who will need direct access to the system, consider whether you have the budget for a hosted solution like Readme that charges per author.

If you want to use a static site generator, which should you choose — Jekyll, Hugo, Sphinx, or some other? Sphinx has the most documentation-oriented features, such as search, PDF, cross-reference linking, and semantic markup. If those features are important, consider Sphinx.

However, choosing Jekyll or Hugo rather than Sphinx does have rationale because their communities extend beyond documentation groups. Sphinx was designed as a documentation platform, so its audience tends to be more niche. Documentation tools almost never have the community size that more general web development tools have. So the tradeoff with Jekyll or Hugo is that although they lack some better documentation features (cross-references, search, PDF, semantic markup), they might have more community and momentum in the long-term. Still, this may leave you in a tight spot if you have to figure out a solution for search, PDF, and translation (which are do-able, just not out-of-the-box).

Markup is also a consideration. If you've narrowed the choice down to Sphinx with reStructuredText or Jekyll/Hugo with Markdown, then one question to ask is whether engineers at your company will write in reStructuredText (assuming engineers will write at all). If they'll write in reStructuredText, great, Sphinx is probably superior for documentation projects due to the semantic advantages of reStructuredText (p. 560). But if engineers insist on Markdown, then maybe Jekyll or Hugo will be a better choice.

Also recognize that there's flexibility even within the static site generator you choose. You can also use Markdown with Sphinx, but when you do, some other Sphinx features become limited. Also, you can use Asciidoc with Jekyll through the jekyll-asciidoc plugin ↗.

If deciding between Jekyll and Hugo, consider your project size. Do you have thousands of pages, all in the same project? Will each author be building the project locally? If so, how much does speed (how fast the project compiles the output) matter? If speed is an important consideration, Hugo will probably be better. But if you prefer a community and a platform that integrates tightly with GitHub, then Jekyll might be better. Coding your own Liquid scripts in Jekyll is also easier than with Go in Hugo.

2. Select a hosting and deployment platform

After you've narrowed down which static site generator you want to use, next think about hosting and deployment options (which offer continuous delivery). If you've decided on Sphinx, consider using Read the Docs (p. 603). If you've decided on Jekyll, then explore GitHub Pages (p. 602), CloudCannon (p. 603), Netlify (p. 605), or Aerobatic (p. 605). If you've decided on Hugo, then explore Netlify (p. 605) or Aerobatic (p. 605). By using one of these platforms, you offload a tremendous burden in maintaining a server and deploying your site.

Usually, within a company, engineering groups manage and control the server infrastructure. Setting up and maintaining your own server for documentation using internal resources only can be a considerable expense and headache. It can take months (if not years) to get engineering to give you space on a server, and even if they do, it likely will not provide half of the features you need (like continuous delivery and a CLI). That's why I recommend these third-party hosting and deployment options if at all possible.

Maintaining your own server is not the business you want to be in, and these third-party platforms enable you to be much more efficient. Removing the hassle of publishing through continuous delivery from the server will simplify your life. On the other hand, if you have an engineering tools support group, and they have bandwidth and interest in supporting tech docs, using internal tools can facilitate integration into the other tools (such as validation testing) available at your work.

If your company prefers to build its own publishing pipeline, before you go down this road, find out what features the internal solution will provide. Explore some of the benefits of these third-party host and deployment options and examine whether the internal solution will have enough parity and long-term support. If you have strong engineering backing, then great, you're probably in a good spot. But if engineers will barely give you the time of day, consider a third-party solution. See Case study: Switching tools to docs-as-code (p. 645) for my experience going down this route.

If you don't have the budget for a third-party host and deployment option, nor do you have internal engineering resources, consider deploying to an AWS S3 bucket ↗. Jekyll has a plugin called S3_website ↗ that easily deploys to S3. It's not a continuous delivery model, but neither does it involve uploading your entire site output every time you want to publish. The S3_website plugin looks at what changed in your output and synchronizes those changes with the files on S3. (However, I admit that once you get used to continuous delivery publishing by simply committing to your repo, it's hard to consider publishing any other way.)

Also, note that even if you're not using Jekyll, you can use GitHub Pages ↗ as a free publishing host for any static site generator output. You simply build your files locally and then push your built files into the GitHub-Pages-enabled repository. With this approach, you wouldn't have the server perform the build process, but you can still handle the process through the command line. Free hosting for your docs on GitHub, regardless of the tool, can be especially convenient.

3. Decide how you'll parse the OpenAPI specification

The OpenAPI specification (p. 181) could also be an important factor in your consideration of tools. How will you display all the endpoint reference documentation (p. 114)? Rather than creating your own template (p. 540) or manually defining these reference sections, I recommend using a tool that can read and parse the OpenAPI for your reference documentation. Not many standalone doc tools easily incorporate and display the OpenAPI specification (p. 241) (yet), so perhaps the best alternative might be to either link to or embed Swagger UI (p. 218) with your docs (assuming you don't have UX resources for a deeper integration).

I've seen some deeper integrations of Swagger UI into existing websites, and someday I hope to do this with a Jekyll theme, but I haven't yet. You could also create a theme using the Swagger UI theme itself. Static site generators can convert any HTML website into a theme where content is powered by the static site generator — see Convert an HTML site to Jekyll ↗ (a tutorial I wrote).

Tools versus content

Although this section has focused heavily on tools, I want to emphasize that content always trumps tooling. The content should be your primary focus, not the tools you use to publish the content. After you get the tooling infrastructure in place, it should mostly take a back seat to the daily tasks of content development.

For a great article on the importance of content over tools, see Good API Documentation Is Not About Choosing the Right Tool ↗ from the Algolia blog. The author explains that "a quality README.md stored on GitHub can be far more efficient than over-engineered documentation that is well displayed but has issues with content."

In some ways, tools are the basketball player's shoes. They matter, for sure. But Michael Jordan wasn't a great basketball player because he wore Nikes, nor was Kobe Bryant great due to his Adidas. You can probably write incredible documentation despite your tooling and platform. Don't let tooling derail your focus on what really matters in your role: the content.

I've changed my doc platforms numerous times, and rarely does anyone seem to care or notice. As long as it looks decent, most project managers and users will focus on the content much more than the design or platform. In some ways, the design should be invisible and unobtrusive, not foregrounding the focus on the content. The user shouldn't be distracted by the tooling.

Also, users and reviewers won't notice (or appreciate) all the effort behind the tools. Even when you've managed to single source content, loop through a custom collection to generate out a special display, incorporate language switchers to jump from platform to platform, etc., the feedback you'll get is "There's a typo here." Or "What's the data type for this parameter?"

On the other hand, the tools you choose do make a huge difference in your productivity, capabilities, and general happiness as a technical writer. Choosing the wrong tool can set back your ability to deliver documentation that your users need.

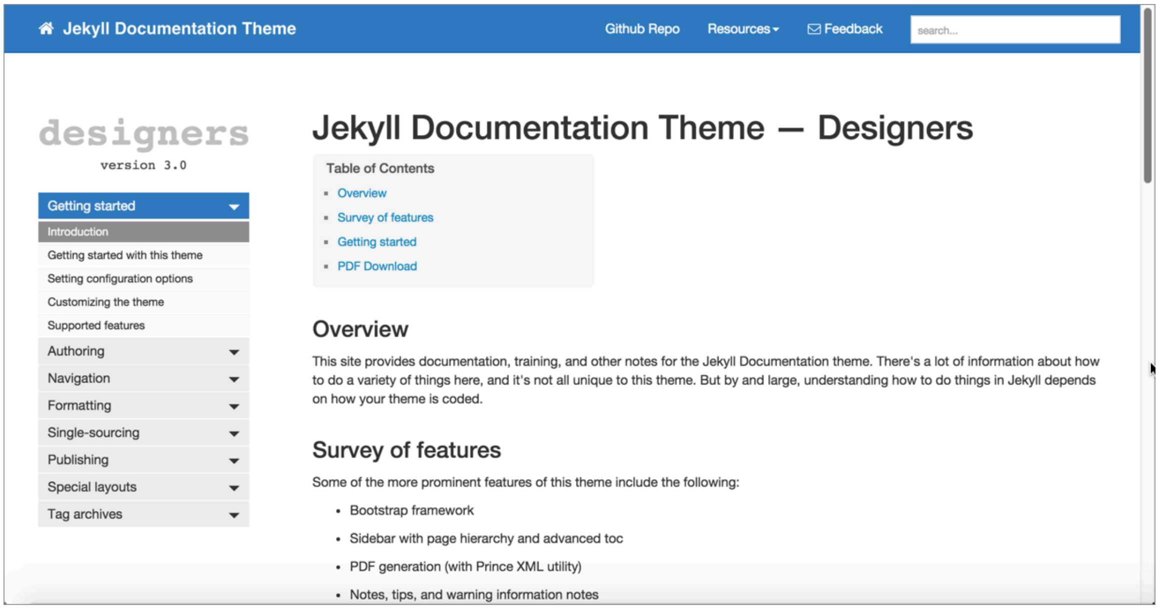
Jekyll and CloudCannon continuous deployment

Static site generators (p. 591) are a breed of website compilers that package up a group of files (usually written in Markdown (p. 556)) and make them into a fully deployable website. One of the most popular static site generators (based on the number of downloads, usage, and community) is Jekyll, and it's the one I have the most experience with, so I'll be focusing on Jekyll here, particularly how you can integrate Jekyll with CloudCannon for a continuous delivery publishing solution.

About Jekyll

Jekyll is a Ruby-based static site generator (p. 592), meaning it uses Ruby as the underlying programming language to compile the website. This site and my blog use Jekyll, and I used Jekyll for the documentation at my past two jobs. For example, all the docs here use Jekyll. With Jekyll, you can publish a fully functional tech comm website that includes content re-use, conditional filtering, variables, PDF output, and everything else you might need as a technical writer.

Here's a documentation theme that I developed for Jekyll:



Jekyll Documentation Theme — Designers

Table of Contents

- Overview
- Survey of features
- Getting started
- PDF Download

Overview

This site provides documentation, training, and other notes for the Jekyll Documentation theme. There's a lot of information about how to do a variety of things here, and it's not all unique to this theme. But by and large, understanding how to do things in Jekyll depends on how your theme is coded.

Survey of features

Some of the more prominent features of this theme include the following:

- Bootstrap framework
- Sidebar with page hierarchy and advanced toc
- PDF generation (with Prince XML utility)
- Notes, tips, and warning information notes

There isn't any kind of special API reference endpoint formatting here, but the platform is so flexible, you can do anything with it as long as you know HTML, CSS, and JavaScript (the fundamental language of the web). With a static site generator, you have a tool for building a full-fledged website using pretty much any style or JavaScript framework you want. With the Jekyll website, you can include complex navigation, content re-use, variables, and more.

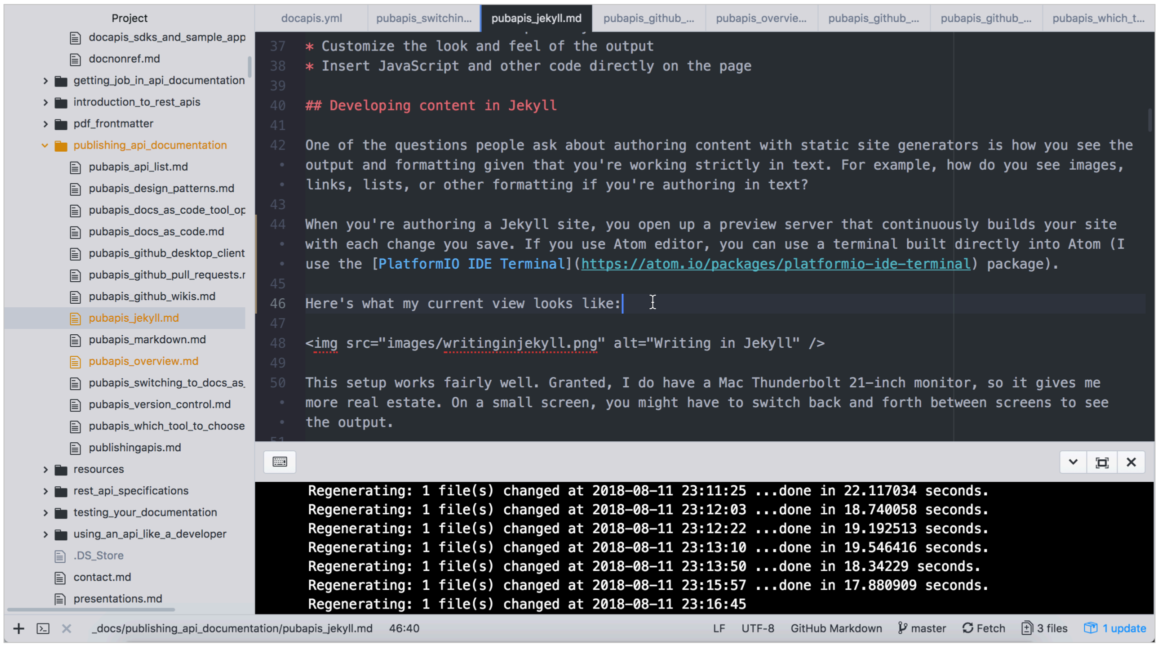
Static site generators give you a lot of flexibility. They're a good choice if you need a lot of control and customization with your site. You're not locked into rigid templates or styles. You define your own templates and structure things however you want. For example, with static site generators, you can do the following:

- Write in a text editor working with Markdown
- Create custom templates for documentation
- Use a revision control repository workflow
- Customize the look and feel of the output
- Insert JavaScript and other code directly on the page

Developing content in Jekyll

One of the questions people ask about authoring content with static site generators is how you see the output and formatting given that you're working strictly in text. For example, how do you see images, links, lists, or other formatting if you're authoring in text?

Here's what the current view of my Jekyll project in Atom editor looks like:



```
* Customize the look and feel of the output
* Insert JavaScript and other code directly on the page

## Developing content in Jekyll

One of the questions people ask about authoring content with static site generators is how you see the
output and formatting given that you're working strictly in text. For example, how do you see images,
links, lists, or other formatting if you're authoring in text?

When you're authoring a Jekyll site, you open up a preview server that continuously builds your site
with each change you save. If you use Atom editor, you can use a terminal built directly into Atom (I
use the [PlatformIO IDE Terminal](https://atom.io/packages/platformio-ide-terminal) package).

Here's what my current view looks like:

<img src="images/writinginjekyll.png" alt="Writing in Jekyll" />

This setup works fairly well. Granted, I do have a Mac Thunderbolt 21-inch monitor, so it gives me
more real estate. On a small screen, you might have to switch back and forth between screens to see
the output.
```

```
Regenerating: 1 file(s) changed at 2018-08-11 23:11:25 ...done in 22.117034 seconds.
Regenerating: 1 file(s) changed at 2018-08-11 23:12:03 ...done in 18.740058 seconds.
Regenerating: 1 file(s) changed at 2018-08-11 23:12:22 ...done in 19.192513 seconds.
Regenerating: 1 file(s) changed at 2018-08-11 23:13:10 ...done in 19.546416 seconds.
Regenerating: 1 file(s) changed at 2018-08-11 23:13:50 ...done in 18.34229 seconds.
Regenerating: 1 file(s) changed at 2018-08-11 23:15:57 ...done in 17.880909 seconds.
Regenerating: 1 file(s) changed at 2018-08-11 23:16:45
```

Atom text editor view while working in Jekyll

When you're authoring a Jekyll site, you first generate a local preview by running this command:

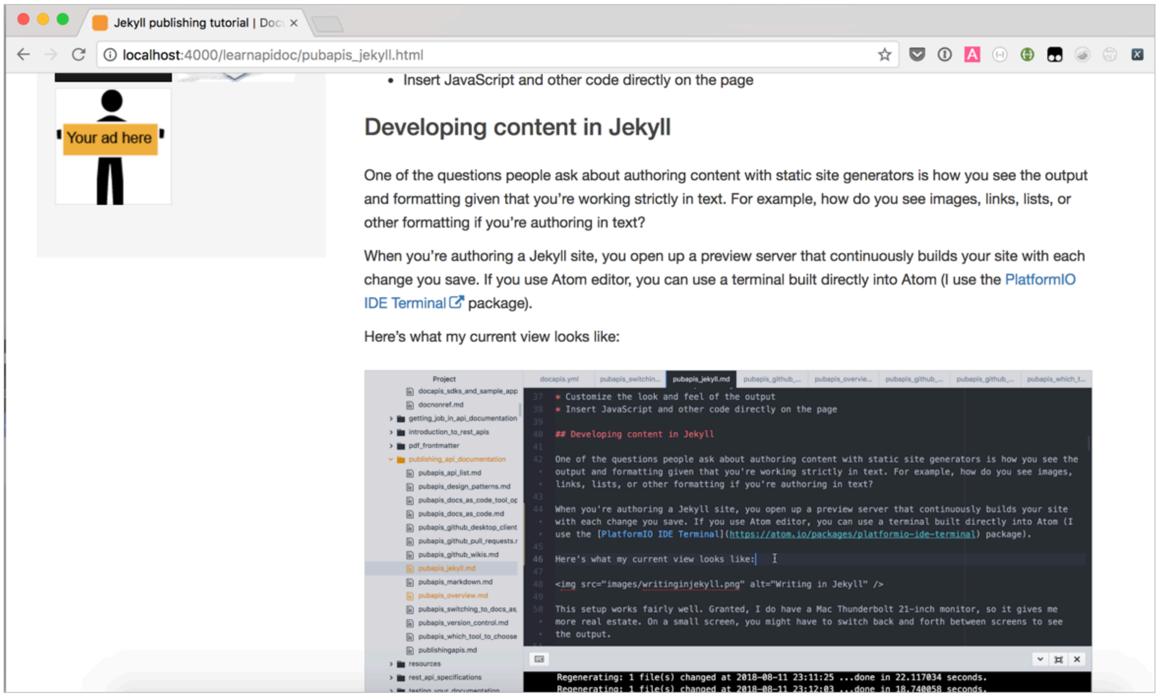
```
bundle exec jekyll serve
```

The response looks something like this:

```
~/projects/learnapidoc [master] $ bundle exec jekyll serve
Configuration file: /Users/tomjoht/projects/learnapidoc/_config.yml
            Source: /Users/tomjoht/projects/learnapidoc
       Destination: _site
 Incremental build: disabled. Enable with --incremental
      Generating...
                    done in 6.457 seconds.
 Auto-regeneration: enabled for '/Users/tomjoht/projects/learnapidoc'
    Server address: http://127.0.0.1:4000/learnapidoc/
  Server running... press ctrl-c to stop.
```

If you use Atom editor, you can use a terminal built directly into Atom (I use the PlatformIO IDE Terminal package).

You then copy the preview server address (in this case, `http://127.0.0.1:4000/learnapidoc/`) and paste it into your browser. This preview server continuously builds your site with each file change you save. I usually view this browser preview side by side with my Atom editor to make sure the formatting and images look right:


Jekyll publishing tutorial | Doc ×
localhost:4000/learnapidoc/pubapis_jekyll.html
Your ad here
Project
docapis_sdks_and_sample_app
docnonref.md
getting_job_in_api_documentation
introduction_to_rest_apis
pdf_frontmatter
publishing_api_documentation
pubapis_api_list.md
pubapis_design_patterns.md
pubapis_docs_as_code_tool_op
pubapis_docs_as_code.md
pubapis_github_desktop_client
pubapis_github_pull_requests.r
pubapis_github_wikis.md
pubapis_jekyll.md
pubapis_markdown.md
pubapis_overview.md
pubapis_switching_to_docs_as_
pubapis_version_control.md
pubapis_which_tool_to_choose
publishingapis.md
resources
rest_api_specifications
testing_your_documentation
docapis.yml
pubapis_switchin...
pubapis_jekyll.md
pubapis_github_...
pubapis_overvie...
pubapis_github_...
pubapis_github_...
pubapis_which_t...

- Insert JavaScript and other code directly on the page

Developing content in Jekyll

One of the questions people ask about authoring content with static site generators is how you see the output and formatting given that you're working strictly in text. For example, how do you see images, links, lists, or other formatting if you're authoring in text?

When you're authoring a Jekyll site, you open up a preview server that continuously builds your site with each change you save. If you use Atom editor, you can use a terminal built directly into Atom (I use the PlatformIO IDE Terminal package).

Here's what my current view looks like:

```
* Customize the look and feel of the output
* Insert JavaScript and other code directly on the page

## Developing content in Jekyll

One of the questions people ask about authoring content with static site generators is how you see the
output and formatting given that you're working strictly in text. For example, how do you see images,
links, lists, or other formatting if you're authoring in text?

When you're authoring a Jekyll site, you open up a preview server that continuously builds your site
with each change you save. If you use Atom editor, you can use a terminal built directly into Atom (I
use the [PlatformIO IDE Terminal](https://atom.io/packages/platformio-ide-terminal) package).

Here's what my current view looks like:

<img src="images/writinginjekyll.png" alt="Writing in Jekyll" />

This setup works fairly well. Granted, I do have a Mac Thunderbolt 21-inch monitor, so it gives me
more real estate. On a small screen, you might have to switch back and forth between screens to see
the output.
```

```
Regenerating: 1 file(s) changed at 2018-08-11 23:11:25 ...done in 22.117034 seconds.
Regenerating: 1 file(s) changed at 2018-08-11 23:12:03 ...done in 18.740058 seconds.
```

Jekyll preview server

This setup works reasonably well. Granted, I do have a large monitor, so it gives me more real estate. On a small screen, you might have to switch back and forth between screens to see the output.

Admittedly, the Markdown format is easy to use but also susceptible to error, especially if you have complicated list formatting. But the majority of the time, writing in Markdown is a joy. You can focus on the content without getting wrapped up in tags. If you do need complex tags, anything you can write in HTML or JavaScript you can include on your page.

Automating builds from Github

You can integrate Jekyll into platforms such as GitHub Pages or CloudCannon to create continuous delivery publishing. Continuous delivery means that when you commit a change to your Git repo, the server automatically rebuilds your Jekyll site.

GitHub Pages (p. 602) is free and is what I use for my blog and this API docs site. But CloudCannon provides more features that might be needed by the enterprise. So let's follow an example in publishing in CloudCannon, which describes itself as the "The Cloud CMS for Jekyll."

In this activity, we'll publish to CloudCanon using the Documentation Theme for Jekyll (the theme I built). You don't need to have a Windows machine to facilitate the building and publishing — you'll do that via CloudCannon and Github. (Of course, being able to build locally is important if you're developing with Jekyll, but I want to avoid Jekyll installation issues here and just demonstrate the continuous deployment features that a host like CloudCannon provides.) This tutorial will show you how to plug into a robust hosting platform that reads content stored and managed on GitHub.

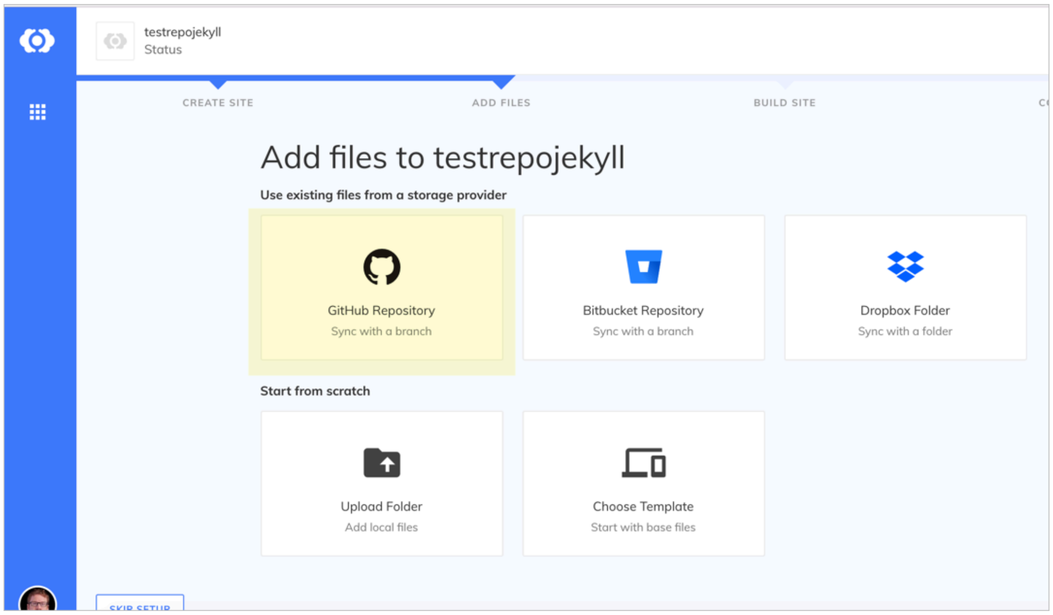
Set up your doc theme on Github

1. Go to the Github page for the Documentation theme for Jekyll and click **Fork** in the upper-right corner.

   When you fork a project, a copy of the project (using the same name) gets added to your own Github repository. You'll see the project at `https://github.com/{your github username}/documentation-theme-jekyll`.

   Sometimes people fork repositories to make changes and then propose pull requests of the fork to the original repo. Other times people fork repositories to create a starting point for a splinter project from the original. Github is all about social coding — one person's ending point is another person's starting point, and multiple projects can be merged into each other. You can learn more about forking here.

2. Sign up for a free account at CloudCannon; sign in using your GitHub credentials.
3. Once you sign in, click **Create Site** and then give the new site a name. Then press **Enter** to create the site.
4. Click the **GitHub Repository** box.

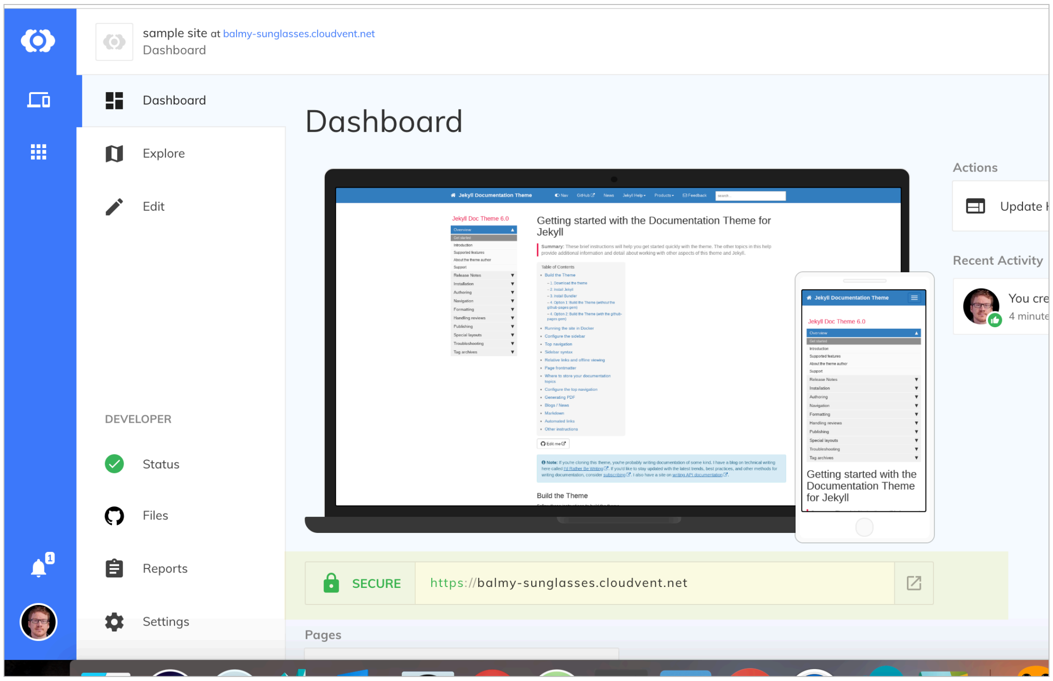


Sync with GitHub

5. Choose the GitHub repo that you forked — **documentation-theme-jekyll**. Select the **gh-pages** branch.
6. CloudCannon pulls the files from the synced repo over to CloudCannon. When the sync finishes, click **Start Site Build** to have CloudCannon build Jekyll from the server.

   When it finishes building, you see a green check mark in the upper-right corner.

7. Go to the **Dashboard.**
8. From the Dashboard, click the URL that appears below your site image:



Viewing your live site

The preview URL will be something random, such as `https://balmy-sunglasses.cloudvent.net/`. You'll need to click **Continue to sample site** because you're on the free plan. When you visit the URL, the theme should look just like the Documentation theme for Jekyll here.

Using the CloudCannon user interface, you can make updates to files, and the updates will sync back to GitHub. Likewise, if you push updates to GitHub, CloudCannon will be notified, pull the changes, and rebuild the output.

The integration between CloudCannon and GitHub is pretty mind-blowing. Through CloudCannon, you can offload all the hassle of hosting and maintaining your website, but you aren't locked into the system in a proprietary way. Your content lives in a custom Jekyll theme on GitHub.

CloudCannon automatically builds the site when you commit new updates to your GitHub repo, entirely removing the publishing and deployment step with a website. CloudCannon also provides additional features for authentication, metrics, suggested improvements, and more.

The only drawback with CloudCannon is that your company must allow you to host documentation content on GitHub. Also, CloudCannon charges a monthly fee (see their pricing). If you need to make a case for third-party hosting, I recommend doing so by analyzing the costs of internal hosting and maintenance.

If cost is an issue and you don't have any privacy restrictions around your docs, consider using GitHub Pages instead. GitHub Pages also gives you continuous integration delivery for GitHub projects, and it's free.

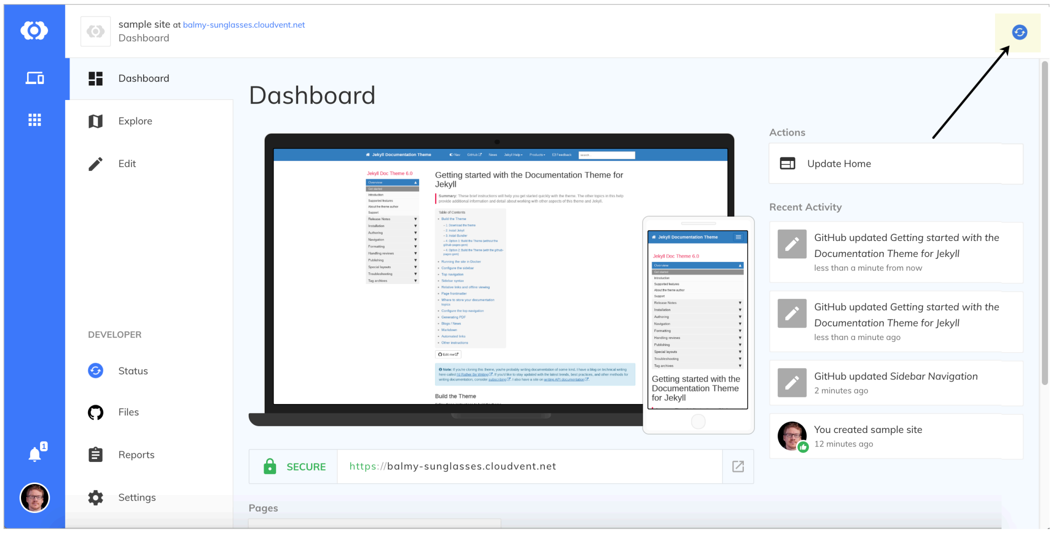
Make an update to your Github repo

When you connect a GitHub repo with CloudCannon, the two sites sync the files. Let's see that workflow in action.

1. In your browser, on GitHub, go to your Github repository that you forked and make a change to a file.

   For example, browse to the index.md file, click the pencil icon ("Edit this file"), make an update, and then commit the update.

2. Return to CloudCannon and observe that your site automatically starts rebuilding.



GitHub updated *Getting started with the Documentation Theme for Jekyll*
less than a minute from now

GitHub updated *Getting started with the Documentation Theme for Jekyll*
less than a minute ago

GitHub updated *Sidebar Navigation*
2 minutes ago

You created sample site
12 minutes ago

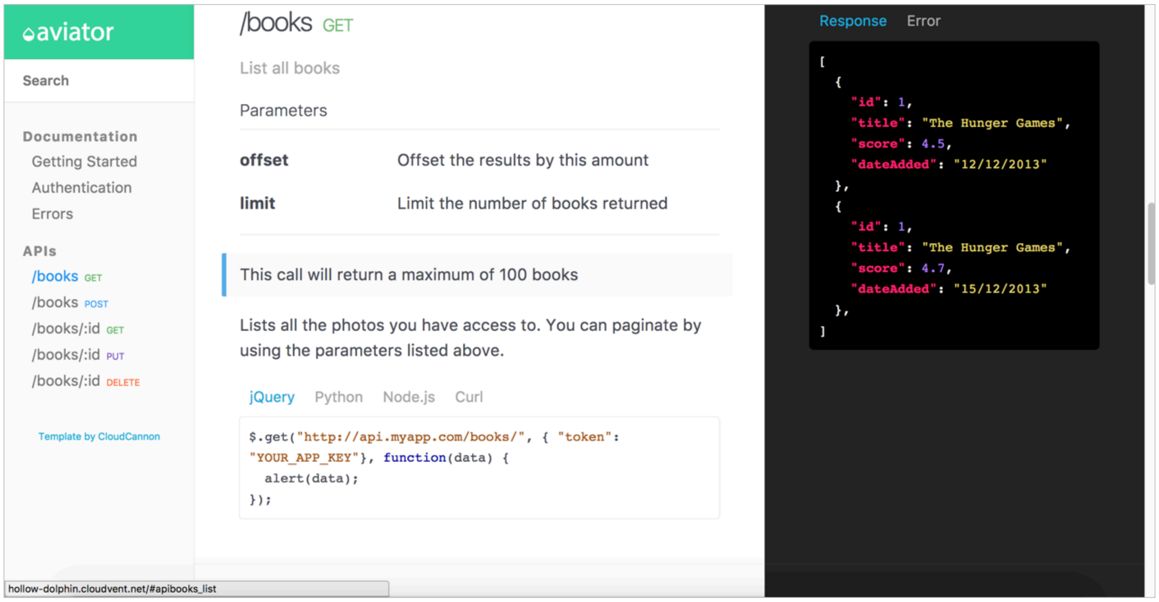
CloudCannon detects the change and automatically rebuilds Jekyll

Wait until the build finishes (the blue circling arrows change into a green check mark), and then look for the change at the preview URL. The change should be reflected.

You've now got a workflow that involves Github as the storage provider syncing to a Jekyll theme hosted on CloudCannon. You're publishing on the fly, based on commits to a repo. This continuous publishing based on Git commits is the essential characteristic of a docs-as-code publishing workflow (p. 551).

The Jekyll Aviator theme

My Jekyll documentation theme is just one possibility for a Jekyll site. (Most people who use it end up heavily customizing it for their needs.) You could also use this Aviator API documentation theme, which has some API templating built-in, or you could choose from many other Jekyll themes. I also wrote a tutorial for creating your own Jekyll theme, likely using your company's site as the template.



/books GET

List all books

Parameters

offset	Offset the results by this amount
limit	Limit the number of books returned

This call will return a maximum of 100 books

Lists all the photos you have access to. You can paginate by using the parameters listed above.

```
$.get("http://api.myapp.com/books/", { "token":
"YOUR_APP_KEY"}, function(data) {
  alert(data);
});
```

```
[
  {
    "id": 1,
    "title": "The Hunger Games",
    "score": 4.5,
    "dateAdded": "12/12/2013"
  },
  {
    "id": 1,
    "title": "The Hunger Games",
    "score": 4.7,
    "dateAdded": "15/12/2013"
  },
]
```

CloudCannon Aviator theme

For inspiration, here are some websites using Jekyll:

- Bootstrap
- Stack Overflow blog
- RethinkDB
- Github docs
- Basekit
- Jekyllrb docs
- SendGrid docs
- Atlassian Design
- CloudCannon docs
- Wistia help center
- Liquid (Shopify)
- devo.ps documentation
- healthcare.gov

Case study: Switching tools to docs-as-code

Changing any documentation tooling at a company can be a massive undertaking. Depending on the amount of legacy content to convert, the number of writers to train, the restrictions and processes you have to work against in your corporate environment and more, it can require an immense amount of time and effort to switch tools from the status quo to docs-as-code.

Additionally, you will likely need to make this change outside your regular documentation work, and you'll probably need to develop the new system *while still updating and publishing content in the old system*. Essentially, this means you'll be laying down a new highway while simultaneously driving down it.

For an overview of the docs-as-code approach, see Docs-as-code tools (p. 551). In this article, I describe the challenges we faced in implementing a docs-as-code approach within a tech writing group at a large company.

Previous processes

Previously, our team published content through a content management system called Hippo (by Bloomreach). Hippo is similar to WordPress or Drupal but is Java-based rather than PHP-based (which made it attractive to a Java-centric enterprise that restricted PHP but still needed a CMS solution for publishing).

To publish a page of documentation, tech writers had to create a new page in the Hippo CMS and then paste in the HTML for the page (or try to use the WYSIWYG editor in the Hippo CMS). If you had 50 pages of documentation to publish, you would need to paste the HTML into each CMS page one by one. Originally, many writers would use tools such as Pandoc to convert their content to HTML and then paste it into the Hippo CMS. This copy-and-paste approach was tedious, prone to error, and primitive.

When I started, I championed using Jekyll to generate and manage the HTML, and I started storing the Jekyll projects in internal Git repositories. I also created a layout in Jekyll that was explicitly designed for Hippo publishing. The layout included a documentation-specific sidebar (previously absent in Hippo on a granular level) to navigate all the content in a particular set of documentation. This Jekyll layout included several styles and scripts to override settings in the CMS.

Despite this innovation, our publishing process still involved pasting the generated HTML (after building Jekyll) page by page into the CMS. Thus, we were only partially using docs-as-code and still had a long way to go. One of the tenets of docs-as-code is to build your output directly from the server (called "continuous deployment"). In other words, you incorporate the publishing logic on the server rather than running the publishing process from your local computer.

This last step, publishing directly from the server, was difficult because another engineering group was responsible for the website and server, and we couldn't just rip Hippo out and start uploading the Jekyll-generated files onto a web server ourselves. It would take another year or more before the engineering team had the bandwidth for the project. Once it started, the project was a wild ride of mismatched expectations and assumptions. But in the end, we succeeded.

Most of the lessons learned here are about this process, specifically how we transitioned to building Jekyll directly from an internal Git repo, the decisions we made and the reasoning behind those decisions, the compromises and other changes of direction, and so on. My purpose here is to share lessons learned so that other writers embarking on similar endeavors can benefit from understanding what might be on the road ahead.

Advantages of integrating into a larger system

Why did we want to move to docs as code in the first place? At most large companies, there are plenty of robust, internally developed tools that tech writers can take advantage of. The docs-as-code approach would allow us to integrate into this robust enterprise infrastructure that developers had already created.

Documentation tools are often independent, standalone tools that offer complete functionality (such as version control, search, and deployment) within their own system. But these systems are often a black box, meaning, you can't really open them up and integrate them into another process or system. With the docs-as-code approach, we had the flexibility to adapt our process to fully integrate within the company's infrastructure and website deployment process. Some of this infrastructure we wanted to hook into included the following:

- Internal test environments (a gamma environment separate from production)
- Authentication for specific pages based on account profiles
- Search and indexing
- Website templating (primarily a complex header and footer)
- Robust analytics
- Secure servers to satisfy Information Security policies with the corporate domain
- Media CDN for distributing images
- Git repositories and GUI for managing code
- Build pipelines and a build management system

All we really needed to do was to generate out the body HTML along with the sidebar and make it available for the existing infrastructure to consume. The engineering team that supported the website already had a process in place for managing and deploying content on the site. We wanted to use similar processes rather than coming up with an entirely different approach.

End solution

In the end, here's the solution we implemented. We stored our Jekyll project in an internal Git repository — the same farm of Git repositories other engineers used for nearly every software project, and which connected into a build management system. After we pushed our Jekyll doc content to the master branch of the Git repository, a build pipeline would kick off and build the Jekyll project directly from the server (similar to GitHub Pages ↗).

Our Jekyll layout omitted any header or footer in the theme. The built HTML pages were then pulled into an S3 bucket in AWS through an ingestion tool (which would check for titles, descriptions, and unique permalinks in the HTML). This bucket acted as a flat-file database for storing content. Our website would make calls to the content in S3 based on permalink values in the HTML to pull the content into another website template that included the header and footer.

The build process from the Git repo to the deployed website took about 10 minutes, but tech writers didn't need to do anything during that time. After you typed a few commands in your terminal (merging with the `gamma` or `production` branch locally and then pushing out the update to origin), the deployment process kicked off and ran all by itself.

The first day in launching our new system, a team had to publish 40 new pages of documentation. Had we still been in Hippo, this would have taken several hours to publish. Even more painful, their release timeframe was an early morning, pre-dawn hour, so the team would have had to publish 40 pages in Hippo CMS at around 4 am to 6 am, copying and pasting the HTML frantically to meet the release push and hoping they didn't screw anything up.

Instead, with the new process, the writer just merged her `development` branch into the `production` branch and pushed the update to the repo. Ten minutes later, all 40 pages were live on the site. She was floored! We knew this was the beginning of a new chapter in the team's processes. We all felt like a considerable publishing burden had been lifted off our shoulders.

Challenges we faced

I've summarized the success and overall approach, but there were a lot of questions and hurdles in developing the process. I'll detail these main challenges in the following sections.

Inability to do it ourselves

The biggest challenge, ironically, was probably with myself — dealing with my own perfectionist, controlling tendencies to do everything on my own, just how I wanted. (This is probably both my biggest weakness and strength as a technical writer.) It's hard for me to relinquish control and have another team do the work. We had to wait *about a year* for the overworked engineering team's schedule to clear up so they would have the bandwidth to do the project.

During this wait time, we refined our Jekyll theme and process, ramped up on our Git skills, and migrated all of the content out of the old CMS into kramdown Markdown. Even so, as project timelines kept getting delayed and pushed out, we weren't sure if the engineering team's bandwidth would ever lighten up. I wanted to jump ship and just deploy everything myself through the S3_website plugin on AWS S3.

But as I researched domain policies, server requirements, and other corporate standards and workflows, I realized that a do-it-myself approach wouldn't work (unless I possessed a lot more engineering knowledge than I currently did). Given our corporate domain, security policies required us to host the content on an internal tier 1 server, which had to pass security requirements and other standards. It became clear that this would involve a lot more engineering and server knowledge and time than I had, as well as maintenance time if I managed the server post-release. So we had to wait.

We wanted to get this right because we probably wouldn't get bandwidth from the engineering team again for a few years. In the end, waiting turned out to be the right approach.

Understanding each other

When we did finally begin the project and start working with the engineering team, another challenge was in understanding each other. The engineering team (the ones implementing the server build pipeline and workflow) didn't understand our Jekyll authoring process and needs.

Conversely, we didn't understand the engineer's world well either. To me, it seemed all they needed to do was upload HTML files to a web server, which seemed a simple task. I felt they were overcomplicating the process with unnecessary workflows and layouts. And what was the deal with storing content in S3 and doing dynamic lookups based on matching permalinks? But they already had a process for publishing content to the existing site, so they were more or less following the same build flow.

Despite the common goal, we lived in seemingly different worlds, and we had to continually communicate about what each other needed. It didn't help that we were located in different states and had to interact virtually, often through chat and email.

Figuring out repo size

Probably the primary challenge was to figure out the correct size for the documentation repos. Across our teams, we had 30 different products, each with their doc navigation and content. Was it better to store each product in its own repo, or to store all products in one giant repo? I flipped my thinking on this several times.

Storing content in multiple repos led to quick build times, reduced visual clutter, resulted in fewer merge conflicts, didn't introduce warnings about repo sizes, and had other benefits with autonomy.

On the other hand, storing all content in one repo simplified content re-use, made link management and validation easier, reduced maintenance efforts, and more. Most of all, it made it easier to update the theme in a single place rather than duplicating theme file updates across multiple repos.

Originally, our team started out storing content in separate repos. When I had updates to the Jekyll theme, I thought I could simply explain what files needed to be modified, and each tech writer would make the update to their theme's files. This turned out not to work — tech writers didn't like making updates to theme files. The Jekyll projects became out of date, and then when someone experienced an issue, I had no idea what version of the theme they were on.

I then championed consolidating all content in the same repo. We migrated all of these separate, autonomous repos into one master repo. This worked well for making theme updates. But soon the long build times (1-2 minutes for each build) became painful. We also ran into size warnings in our repo (images and other binary files such as Word docs were included in the repos). Sometimes merge conflicts happened.

The long build times were so annoying, we decided to switch back to individual repos. There's nothing worse than waiting 2 minutes for your project to build, and I didn't want the other tech writers to hate Jekyll like they did Hippo. The lightning-fast auto-regenerating build time with Jekyll is part of its magic.

Creative solutions for theme distribution across repos

I came up with several creative ways to push the theme files out to multiple small repos in a semi-automated way. My first solution was to distribute the theme through RubyGems ↗, which is Jekyll's official solution for theming ↗. I created a theme gem, open-sourced it and the theme (see Jekyll Doc Project ↗), and practiced the workflow to push out updates to the theme gem and pull them into each repo.

It worked well (just as designed). However, it turns out our build management system (an engineering system used to build outputs or other artifacts from code repositories) couldn't build Jekyll from the server using Bundler ↗, which is what RubyGems required. (Bundler is a tool that automatically gets the right gems for your Jekyll project based on the Jekyll version you are using. Without Bundler, each writer installs the jekyll gem ↗ locally and builds the Jekyll project based on that gem version.)

My understanding of the build management system was limited, so I had to rely on engineers for their assessment. Ultimately, we had to scrap using Bundler and just build using `jekyll serve` because the engineers couldn't make Bundler work with the build system. So I still had the problem of distributing the same theme across multiple repos.

My second attempt was to distribute the theme through Git submodules ↗. This approach involved storing the theme in its own Git repo that other Git repos would pull in. However, our build management system couldn't support Git submodules either, it turned out.

I then came up with a way to distribute the theme through Git subtrees. Git subtrees worked in our build system (although the commands were strange), and it preserved the short build times. However, when the engineering team started counting up all the separate build pipelines they'd have to create and maintain for each of these separate repos (around 30), they said this wasn't a good idea from a maintenance point of view.

Not understanding all the work involved around building publishing pipelines for each Git repo, there was quite a bit of frustration here. It seemed like I was going out of my way to accommodate engineering limitations, and I wasn't sure if they were modifying any of their processes to accommodate us. But eventually, we settled on two Git repos and two pipelines. We had to reconsolidate all of our separate repos back into two repos. You can probably guess that moving around all of this content, splitting it out into separate repos and then re-integrating it back into consolidated repos, etc., wasn't a task that the writers welcomed.

There was a lot of content and repo adjustment, but in the end, an architecture consisting of two large repos was the right decision. In fact, in retrospect, I wouldn't have minded just having one repo for everything.

Each repo had its own Jekyll project. If I had an update to any theme files (e.g., layouts or includes), I copied the update manually into both repos. Copying the files was easier than trying to devise an automated method. It also allowed me to test updates in one repo before rolling them out to the other repo. To reduce the slow build times, I created project-specific config files that would cascade with the default configuration file and build only one directory rather than all of them. This project-specific config technique reduced the build time to the normal lightning-fast times of less than 5 seconds.

Let me provide a little more details here on how we shortened the build times, because this is a reason many adopt Hugo instead of Jekyll. To reduce the build times, we created a project-specific configuration file (e.g., acme-config.yml) that sets, through the `defaults`, all the directories to `publish: false` but lists one particular directory (the one with content you're working on) as `publish: true`. Then to build Jekyll, you cascade the config files like this:

```
jekyll serve --config _config.yml,acme-config.yml
```

The config files on the right overwrite the config files on the left. It works quite well.

Also, although at the time I grumbled about having to consolidate all content into two repos, as the engineers required, I eventually came to agree with the engineers' decision. Recognizing this, my respect and trust in the engineering team's judgment grew considerably. In the future, I started to treat the engineers' recommendations and advice about various processes with much more respect. I didn't assume they misunderstood our authoring needs and requirements so much, and instead followed their direction more readily.

Ensuring everyone builds with the same version of Jekyll

Another challenge was ensuring everyone built the project using the same version of Jekyll. Normally, you include a Gemfile in your Jekyll project that specifies the version of Jekyll you're using, and then everyone who builds the project with this Gemfile runs Bundler to make sure the project executes with this version of Jekyll. However, since our build pipeline had trouble running Bundler, we couldn't ensure that everyone was running the same version of Jekyll.

Ideally, you want everyone on the team using the same version of Jekyll to build their projects, and you want this version to match the version of Jekyll used on the server. Otherwise, Jekyll might not build the same way. You don't want to later discover that some lists don't render correctly or that some code samples don't highlight correctly because of a mismatch of gems. Without Bundler, everyone's version of

Jekyll probably differed. Additionally, the latest supported version of Jekyll in the build management system was an older version of Jekyll (at the time, it was 3.4.3, which had a dependency on an earlier version of Liquid that was considerably slower in building out the Jekyll site).

The engineers finally upgraded to Jekyll 3.5.2, which allowed us to leverage Liquid 4.0. This version upgrade reduced the build time from about 5 minutes to 1.5 minutes. Still, Jekyll 3.5.2 had a dependency on an older version of the rouge gem ↗, which was giving us issues with some code syntax highlighting for JSON. The process of updating the gem within the build management system was foreign territory to me, and it was also a new process for the engineers.

To keep everyone in sync, we asked that each writer check their version of Jekyll and manually upgrade to the latest version. This request (which some ignored) turned out not to be much of an issue since there wasn't much of a difference from one Jekyll gem version to the next (at least not for the features we were using).

Ultimately, I learned that it's one thing to update all the Jekyll gems and other dependencies on your own machine, but it's an entirely different effort to update these gems within a build management server in an engineering environment you don't own. We relied on the engineering team to make these updates (but often had to plead and beg them to do it).

Figuring out translation workflows

Figuring out the right process for translation was also tricky. We started out translating the Markdown source. Our translation vendor affirmed that their ingestion process could handle Markdown as a source format, and we did tests to confirm it. However, after a few translation projects, it turned out that they couldn't handle content that *mixed* Markdown with HTML, such as a Markdown document with an HTML table (and we almost always used HTML tables in Markdown). The vendors would count each HTML element as a Markdown entity, which would balloon the cost estimates.

Further, the number of translation vendors that could handle Markdown was limited, which created risks around the vendors that we could use. For example, our localization managers often wanted to work with translation agencies in their own time zones. But if we were reliant on a particular vendor for their ability to process Markdown, we restricted our flexibility with vendors. If we wanted to scale across engineering, we couldn't force every team to use the same translation vendors, which might not be available in the right time zones. Eventually, we decided to revert to sending only HTML to vendors.

However, if we sent only the HTML output from Jekyll to vendors, it made it difficult to apply updates. With Jekyll (and most static site generators), your sidebar and layout are packaged into *each* of your doc pages. Assuming that you're just working with the HTML output (not the Markdown source), if you have to add a new page to your sidebar, or update any aspect of your layout, you would need to edit each individual HTML file instance to make those updates across the documentation. That wasn't something we wanted to do.

In the end, the process we developed for handling translation content involved manually inserting the translated HTML into pages in the Jekyll project and then having these pages build into the output like the other Markdown pages. We later evolved the process to create container files that provided the needed frontmatter metadata but which used includes to pull the body content from the returned HTML file supplied by the translation vendors. It was a bit of manual labor, but acceptable given that we didn't route content through translation all that often.

The URLs for translated content also needed to have a different `baseurl`. Rather than outputting content in the `/docs/` folder, translated content needed to be output into `/ja/docs/` (for Japanese) or `/de/docs/` (for German). However, a single Jekyll project can have only one `baseurl` value as defined in the default _config.yml file. I had this `baseurl` value automated in a number of places in the theme.

To account for the new `baseurl`, I had to incorporate a number of hacks to prepend language prefixes into this path and adjust the permalink settings in each translated sidebar to build the file into the right `ja` or `de` directory in the output. It was confusing, and if something breaks in the future, it will take me a while to unravel the logic I implemented.

Overall, translation remains one of the trickier aspects to handle with static site generators, as these tools are rarely designed with translation in mind. But we made it work. (Another challenge with translation was how to handle partially translated doc sets — I won't even get into this here.)

Overall, given the extreme flexibility and open nature of static site generators, we were able to adapt to the translation requirements and needs on the site.

Other challenges

There were a handful of other challenges worth mentioning (but not worth full development as in the previous sections). I'll briefly list them here so you know what you might be getting into when adopting a docs-as-code approach.

Moving content out of the legacy CMS

We probably had about 1,500 pages of documentation between our 10 writers. Moving all of this content out of the old CMS was challenging. Additionally, we decided to leave some deprecated content in the CMS, as it wasn't worth migrating. Creating redirect scripts that would correctly re-route all the content to the new URLs (especially with changed file names) while not routing away from the deprecated CMS pages was challenging. Engineers wanted to handle these redirects at the server level, but they needed a list of old URLs and new URLs.

To programmatically create redirect entries for all the pages, I created a script that iterated throughout each doc sidebar and generated out a list of old and new URLs in a JSON format that the engineering team could incorporate into their redirect tool. It worked pretty well, but migrating the URLs through comprehensive redirects required more analysis and work.

Implementing new processes while still supporting the old

While our new process was in development (and not yet rolled out), we had to continue supporting the ability for writers to generate outputs for the old system (pasting content page by page into the legacy Hippo CMS). Any change we made had to also include the older logic and layouts to support the older system. Accommodating multiple templates and scripts was particularly difficult with translation content since it required such a different workflow. Being able to migrate our content into a new system while continuing to publish in the older system, without making updates in both places, was a testament to the flexibility of Jekyll. We created separate layouts and configuration files in Jekyll to facilitate these needs.

One of the biggest hacks was with links. Hippo CMS required links to be absolute links if pasting HTML directly into the code view rather than using the WYSIWYG editor (insane as this sounds, it's true). We created a script in our Jekyll project to populate links with either absolute or relative URLs based on the publishing targets. It was a non-standard way of doing links (essentially we treated them as variables whose values we defined through properties in the config file). It worked. Again, Jekyll's flexibility allowed us to engineer the needed solution.

Constantly changing the processes for documentation

We had to constantly change the processes for documentation to fit what did or did not work with the engineering processes and environment. For example, git submodules, subtrees, small repos, large repos, frontmatter, file names, translation processes, etc., all fluctuated as we finalized the process and worked around issues or incompatibilities.

Each change created some frustration and stress for the tech writers, who felt that processes were changing too much and didn't like to hear about updates they would need to make or learn. And yet, it was hard to know the end from the beginning, especially when working with unknowns around engineering constraints and requirements. Knowing that the processes we were laying down now would likely be cemented into the pipeline build and workflow for long into the distant future was stressful.

I wanted to make sure we got things right, which might mean adjusting our process, but I didn't want to make too many adjustments because each time there was a change, it weakened the confidence among the other tech writers about our direction and expertise about what we were doing.

During one meeting, I somewhat whimsically mentioned that updating our permalink path wouldn't be a bad idea (to have hierarchy in the URLs). One of the tech writers noted that she was already under the gun to meet deadlines for four separate projects and wasn't inclined to update all the permalinks for each page in these projects. After that, I was cautious about introducing any change without having an extremely compelling reason for it.

The experience made me realize that the majority of tech writers don't like to tinker around with tools or experiment with new authoring approaches. They've learned a way to write and publish content, and they resent it when you modify that process. It creates an extreme amount of stress in their lives. And yet, I kind of like to try new approaches and techniques. How do you know, without experimenting, if there isn't a better way of doing something?

In the engineering camp, I also took some flak for changing directions too frequently, particularly with the repo sizes. But from my perspective, I had to change directions to try to match the obscure engineering requirements. In retrospect, it would have helped if I had visited the engineers for a week to learn their workflow and infrastructure in depth.

Styling the tech docs within a larger site

Another challenge was with tech doc styles. The engineering team didn't have resources to handle our tech doc styling, so I ended up creating a stylesheet (3,000 lines long) with all CSS namespaced to a class of `docs` (for example, `.docs p, .docs ul`, etc.). I implemented namespacing to ensure the styles wouldn't alter other components of the site. Much of this CSS I simply copied from Bootstrap ↗. The engineers pretty much incorporated this stylesheet into their other styles for the website.

With JavaScript, however, we ran into namespace collisions and had to wrap our jQuery functions in a unique name to avoid conflicts (the conflicts would end up breaking the initialization of some jQuery scripts). These namespace collisions with the scripts weren't apparent locally and were only visible after deploying on the server, so the test environment constantly flipped between breaking or not breaking the sidebar (which used jQuery). As a result, seeing broken components created a sense of panic from the engineers and dread among the tech writers.

The engineers weren't happy that we had the ability to break the display of content with our layout code in Jekyll. At the same time, we wanted the ability to push out content that relied on jQuery or other scripts. In the end, we got it to work, and the returned stability calmed down the writers.

Transitioning to a Git-based workflow

While it may seem like Jekyll was the authoring tool to learn, the more taxing challenge was becoming familiar with Git-based workflows for doc content. Git required some learning and familiarity with the command line and version control workflows.

Some writers already had a background with Git, while others had to learn it. Although we all ended up learning the Git commands, I'm not sure everyone actually used the same processes for pulling, pushing, and merging content (there's a lot of ways to do similar tasks).

There were plenty of times where someone accidentally merged a development branch into the master or found that two branches wouldn't merge, or they had to remove content from the master and put it back into development, etc. Figuring out the right process in Git is not a trivial undertaking. Even now, I'll occasionally find a formatting error because Git's conflict markers `>>>>>>>` and `<<<<<<<` find their way into the content, presumably from a merge gone wrong. We don't have any validation scripts (yet) that look for marker stubs like this, so it's a bit disheartening to suddenly come across them.

Striking a balance between simplicity and robustness in doc tooling

Overall, we had to support a nearly impossible requirement in accommodating less technical contributors (such as project managers or administrators outside our team) as well as advanced authors. The requirement was to keep doc processes simple enough for non-technical people to make updates (similar to how they did in the old CMS), while also providing enough robustness in the doc tooling to satisfy the needs of tech writers, who often need to single-source content, implement variables, re-use snippets, output to PDF, and more.

In the end, given that our primary audience and contributors were developers, we favored tools and workflows that developers would be familiar with. To contribute substantially in the docs, we decided that you would have to understand, to some extent, Git, Markdown, and Jekyll. For non-technical users, we directed them to a GUI (similar to GitHub's GUI) they could interact with to make edits in the repository. Then we would merge in and deploy their changes.

However, even the less technical users eventually learned to clone the project and push their updates into a development branch using the command line. It seems that editing via the GUI is rarely workable as a long-term solution.

Building a system that scales

Although we were using open source tools, our solution scaled for the enterprise. Because the content used Markdown as the format, anyone could quickly learn it. And because we used standard Git processes and tooling, engineers can more easily plug into the system.

We already had some engineering teams interacting in the repo. Our goal was to empower lots of engineering teams with the ability to plug into this system and begin authoring. Ideally, we could have dozens of different engineering groups owning and contributing content, with the tech writers acting more like facilitators and editors.

No licenses or seats were required to scale out the authoring. A writer just used Atom editor (or another IDE). The writer would open up the project and work with the text, treating docs like code.

Within the first few weeks of launching our system, we found that engineers liked to contribute updates using the same code review tools they used with software projects. This use of code review tools simplified the editing workflow. But it also created more learning on our part, because it meant we would need to learn these code review tools, how to push to the code review system, how to merge updates from the reviews, and so forth. Trying to evaluate a doc contribution by looking at a diff file in a code review tool is more annoying than helpful. I prefer to see the content in its whole context, but engineers typically just want to focus in on what has changed.

Additionally, empowering these other groups to author required us to create extensive instructions, which was an entire documentation project in itself. I created around 30+ topics in our guide that explained everything from setting up a new project to publishing from the command line using Git to creating PDFs, navtabs, inserting tooltips and more. Given that this documentation was used internally only and wasn't documentation consumed externally, business executives didn't see a high value in creating it, nor did they want us devoting much bandwidth to it. Yet it consumed *a lot* of time. Making good documentation is hard, and given the questions and onboarding challenges, I realized just how much the content needed to be simplified and fine-tuned. (If writers couldn't follow it, they pinged me for help, draining even more of my time.)

Unfortunately, when we began the project, we didn't secure resourcing and funding for its ongoing maintenance and support. In many ways, working on the project was like working on an open-source project. Although much work still needed to be done in documenting our toolset and workflows, our official priorities always focused on externally facing documentation content. Almost no one wanted to acknowledge the time and energy required to support internal tooling process and documentation. While much of this hassle could have been eliminated through third-party hosting and deployment solutions (like CloudCannon), the company preferred to build its own tools but not fully dedicate resourcing for tool maintenance and documentation.

Conclusion

Almost everyone on the team was happy about the way our doc solution turned out. Of course, there are always areas for improvement, but the existing solution was head and shoulders above the previous processes. Perhaps most importantly, Jekyll gave us an incredible degree of flexibility to create and adapt to our needs. It was a solution we could build on and make fit our infrastructure and requirements.

I outlined the challenges here to reinforce the fact that implementing docs-as-code is no small undertaking. It doesn't have to be an endeavor that takes months, but at a large company, if you're integrating with engineering infrastructure and building out a process that will scale and grow, it can require a decent amount of engineering expertise and effort.

If you're implementing docs-as-code at a small company, you can simplify processes and use a system that meets your needs. For example, you could use GitHub Pages, or use the S3_website plugin to publish on AWS S3, or better yet, use a continuous deployment platform like CloudCannon or Netlify. (I explore these tools in more depth in Hosting and deployment options (p. 602).) I might have opted for either of these approaches if allowed and if we didn't have an engineering support team to implement the workflow I described.

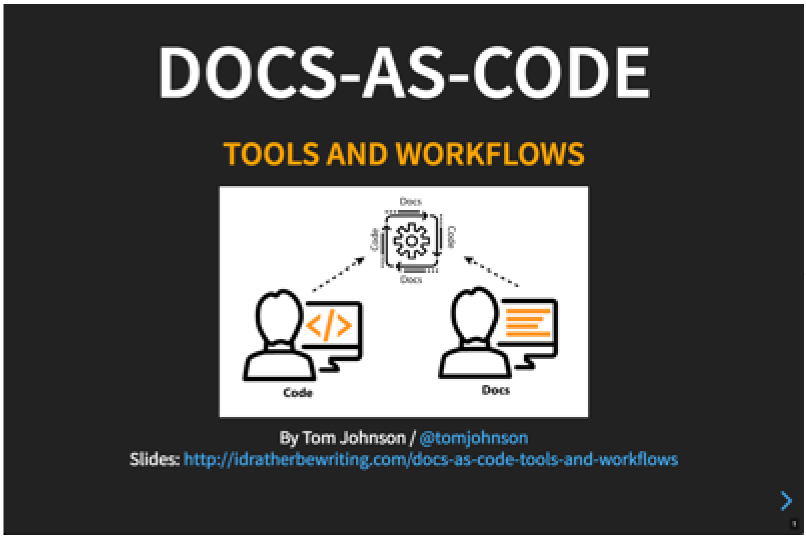
Also, tools implementation is somewhat of a mixed experience for me. Intimate knowledge of doc tools is critical *when you're implementing your solution.* After you're finished, you no longer need that depth of knowledge, and I find it somewhat fading from my awareness. The ephemeral need for tool knowledge is probably why so many consultants specialize in tools — they ride into town, set things up, and then drive their wagons to the next town to repeat the show. But if you're a full-time employee, and your primary job is developing content, not tools, then how do you find the time and support to develop the needed tool knowledge for the temporary period when you're implementing a system, only to abandon the knowledge later, after everything is implemented and running smoothly?

I enjoy getting my hands in the code of docs-as-code tools, but I'm pretty sure both the other tech writers and engineering teams are happy to see the sense of stability and normalcy return. They don't like it when I continually experiment and develop on the platform because it inevitably means change. It means occasionally things break. Or I discover that a particular approach wasn't optimal. In some way, it causes a bit of stress.

And yet, platforms and tools are rarely static for any duration of time. Even Jekyll continually releases new versions, responding to changes in the Internet landscape and trending technology needs. So maybe in a few years, we'll go through this whole process again. Even so, I have a propensity and facility with doc tools, and I like getting my hands dirty in the code.

Slides and links to republished content

For a slide presentation that covers the topics listed in this article, see the following:



DOCS-AS-CODE

TOOLS AND WORKFLOWS

Docs
Code
Code
Docs
Code
Docs

By Tom Johnson / @tomjohnson
Slides: http://idratherbewriting.com/docs-as-code-tools-and-workflows

Additionally, note that this content was also republished in the Developer Portals e-Magazine Winter 2018, by Pronovix:

Developer Portals
WINTER 2018
A selection of articles
about developer portals
and API documentation

It was also republished in Anne Gentle's Docs Like Code: Case Studies:

Image credit: Flickr laurencehorton

ENGINEERING • SECURITY • GITHUB • GIT • JEKYLL • STATIC SITE • CHANGE • CULTURE • TEAMWORK • COLLABORATION • DEVELOPMENT

Case study: Switching tools to docs-as-code

Originally published in the Documenting APIs: A Guide for Technical Writers on Tom Johnson's site, I'd Rather Be Writing. Thanks, Tom, for sharing your story in detail for others to learn.

For an overview of the docs-as-code approach, see Docs-as-code tools. In this article, I describe the challenges we faced in implementing a docs-as-code approach within a tech writing group at a large company.

Changing any documentation tooling at a company can be a huge undertaking. Depending on the amount of legacy content to convert, the number of writers to train, the restrictions and processes you have to work against in your corporate environment and more, it can require an immense amount of time and effort to switch tools from the status quo to docs-as-code.

BY TOM JOHNSON

FEBRUARY 12, 2018

FEBRUARY 12, 2018

TWEET

I also gave a presentation on Docs-as-code tools and workflows to the STC Rocky Mountain and WTD Denver group.

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/pubapis_overview.html] to view the content.

To read some other docs-as-code posts on my blog, see the following:

- Discoveries and realizations while walking down the Docs-as-Code path
- Limits to the idea of treating docs as code
- Will the docs-as-code approach scale? Responding to comments on my Review of Modern Technical Writing

To Learn more about docs as code in general, see Anne Gentle's book Docs Like Code.

Tools FAQ

Q: Is it possible to author API documentation using DITA/XML or an XML editor (like Oxygen)? I saw your recommended list of tools in the Publishing API docs (p. 530) section, but it would be great if we could leverage existing tools.

> Sure, you can use OxygenXML or similar tools and write your dev docs in DITA. However, for the reference content, if you're working with REST APIs, I recommend documenting the endpoints using the OpenAPI spec (p. 181). If documenting class-based libraries (p. 691), then use in-code annotations (p. 712) for that language.
>
> In the dev doc survey I conducted (p. 44) about dev doc trends, about 11% of people writing docs for devs use XML-based solutions, so you wouldn't be alone. However, if you're intending to solicit contributions from engineers, and you have a lively and engaged developer community who will *actually be contributing* to docs, then I do not recommend XML solutions. Use Markdown (p. 556) instead.

Q: We use Madcap Flare but document our APIs in Swagger. We're considering dropping Flare and using Readme.com instead, which can import Swagger, but should we be looking at other tools? We also don't document just APIs, but have protocols we document outside of the APIs too.

> MadCap Flare ↗ is a much more robust tool for handling documentation than Readme.com ↗. For example, last time I checked, Readme.com doesn't let you store content for re-use. There was a recent discussion in the #documenting-apis channel in WTD Slack ↗ about Readme.com. More people seem to be abandoning Readme.com to pursue a more flexible platform. I asked for more details. One person said, "Outages, organizing content. I didn't use it much myself. … After your reference specs and examples are in readme.com, you're limited in what you can get out. If nothing else, that's a mark against them." In short, sure Readme.com can import and display reference API content well (if you like the design), but if you're stuck using a simple editor for everything else and can't leverage more powerful tools for other doc management, life might get more challenging. That said, many top companies use Readme.com and the output looks professional and industry-aligned.
>
> Personally, I recommend going with a static site generator ↗ (SSG) for dev docs instead of a third-party online hosted platform. SSGs are the trend for a reason — they're way more fun, flexible, and engineering-oriented. If you aren't into developing your own site, though, you can also look at some alternatives in this same hosted doc category – Developerhub.io ↗, Readthedocs.com ↗. If you're doing heavy localization and generating lots of PDF, probably stick with your existing tool. Generate out your reference using Redoc (p. 249), Swagger (p. 218), or other specialized generators.

Q: We're considering using GitHub as our documentation CMS, but we're unsure what it can support. Does it provide a documentation portal? HTML generation? An editor to add/update content? Themes? bi-directional sync with Git? What is the process flow if it allows documentation?

> GitHub is a solid platform for storing content, especially for open-source projects. It's not a CMS, and it will require you to work out your Git workflows (p. 562) for development, branching, and collaboration. Most people use GitHub as a storage source, not for the final presentation of the docs. GitHub will render Markdown into HTML, but most don't use the default HTML rendered display from GitHub wikis, as the display isn't customizable. However, note that you can store your branded

content and higly customized theme for any static site generator ↗ in GitHub, as well as for any XML project. The only requirement is that your content be text files rather than binary files (e.g., Word, FrameMaker).

If your project is a Jekyll project, GitHub Pages (p. 602) will auto-build your project when you commit to a branch you tell it to watch. If it's not a Jekyll project (for example, maybe it's a Hugo or Gatsby project), you can use Netlify ↗ to get your content from GitHub and build out other outputs. Forestry.io ↗ can also pull out your GitHub content (into more of a headless CMS client) and build out Jekyll and Hugo outputs. CloudCannon ↗, which is also CMS-like, can do so for Jekyll with GitHub. Readthedocs.com ↗ uses Sphinx but stores content in GitHub. There are many ways to use GitHub, including GitHub wikis (p. 569), but most people just use GitHub to store content that gets pulled into other platforms.

Chapter 11: Thriving in the API doc space

Getting an API documentation job and thriving requires you to demonstrate your technical aptitude through a writing portfolio. The portfolio should include samples of documentation written for developers. One way to build this portfolio is by working on an open-source project. You also need to live in a tech hub where API documentation jobs are available, such as California, Texas, New York, or Virginia. Overall, thriving in the developer documentation space requires you to continually learn a healthy dose of code, which can be challenging.

Chapter sections:

The job market for API technical writers 660
How much code do you need to know? 664
Best locations for API documentation jobs 674
Activity: Find an Open-Source Project 686
Activity: Create or fix an API reference documentation topic 690

The job market for API technical writers

Technical writers who can write developer documentation are in high demand, especially in the Silicon Valley area. There are plenty of technical writers who can write documentation for graphical user interfaces but not many who can navigate the developer landscape to provide highly technical documentation for developers working in code.

Although I briefly mentioned jobs in Introduction to REST API documentation (p. 22), in this section of my API documentation course, I'll dive deeper into the job market for API documentation.

Basic qualifications you must have

Breaking into your first API documentation role can be challenging. Employers will usually have three requirements to hire you:

1. Familiarity with 1-2 programming languages or other technical foundations
2. Experience writing docs for a developer audience
3. A portfolio with writing samples demonstrating the above two points

These requirements are why I've focused this course on activities that will help you break into the field. Although I could create more quizzes in this course, and at the end, you could earn a "certificate" (which wouldn't be a bad idea, actually) it would be virtually meaningless in your job search and larger goals. There's no way around it: if you're serious about breaking into API documentation, you need to fulfill the above requirements. Completing the activities in this course will help you do that.

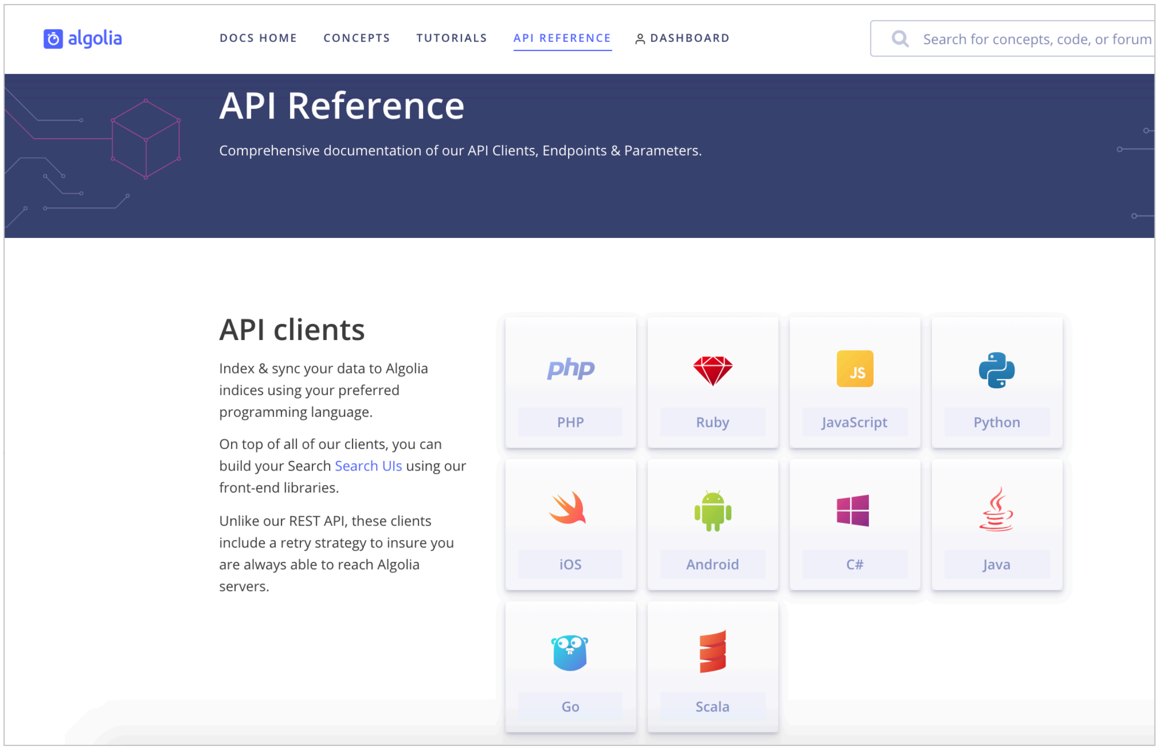
Why employers look for candidates who can read programming languages

In nearly every job description for technical writers in developer documentation, you'll see requirements like this:

> Ability to read code in one or more programming languages, such as Java, C++, or Python.

You may wonder what the motivation is behind these requirements, especially if the core APIs are RESTful. After all, they can't expect you to *do* a programmer's job. No, but here's the most common scenario. The company has a REST API for interacting with their services. To make it easy for developers, the company provides SDKs and client implementations (p. 480) in various languages for the REST API.

For an example, take a look at Algolia's API. You can view the documentation for their REST API here. However, when you implement Algolia (which provides a search feature for your site), you'll probably follow the documentation for your specific platform or language.



API Reference

Comprehensive documentation of our API Clients, Endpoints & Parameters.

API clients

Index & sync your data to Algolia indices using your preferred programming language.

On top of all of our clients, you can build your Search Search UIs using our front-end libraries.

Unlike our REST API, these clients include a retry strategy to insure you are always able to reach Algolia servers.

Although users could construct their own code when using the REST endpoints, most developers would rather leverage existing code and copy and paste what they need.

When I worked at Badgeville, we developed a collection of JavaScript widgets that developers could easily copy and paste into their web pages, making a few adjustments as needed. Developers could also create their own JavaScript widget code (from scratch) based on calls to the REST endpoints, but sometimes it can be tricky to know how to retrieve all the right information and then manipulate it in the right way in your chosen language. It's easier to use the pre-built JavaScript widgets.

Remember that developers are typically using a REST API as a *third-party* service. The developers' primary focus is their own company's code; they're just leveraging your REST API as an additional, extra service. As such, developers want to just get in, get the code, and get out. This get-in-and-get-out mentality is why companies need to provide multiple client SDKs in as many languages as possible — these client implementations allow developers to implement the API quickly and efficiently.

If you were recruiting for a technical writer to document Algolia, how would you word the job requirements? Can you now see why even though the core work involves documenting the REST API, it would also be good to have an "ability to read code in one or more programming languages, such as Java, C++, or Python."

The number of SDKs a company distributes can vary considerably. You might not have six SDKs in multiple languages and frameworks for your API. You might be in a C++ only shop where all you need to know is C++ and nothing more. If that's the case, you'll need to develop a deeper knowledge of C++ so you can provide more value in your writing role.

Although the proliferation of code and platforms creates pressure on the multi-lingual capabilities of technical writers, if you can understand what's going on in one programming language, your description of the reference implementations in other programming languages will follow highly similar patterns.

What mainly changes across languages are the code snippets and some of the terms. You might refer to "functions" instead of "classes," and so on. Even so, getting all the language right can be a challenge, which is why it's so hard to find technical writers who have skills for producing developer documentation, especially for the SDKs and sample apps.

Providing value without in-depth technical knowledge

The degree to which you can provide value in your role as a technical writer is often directly proportional to your level of technical knowledge. For example, if you land (or inherit) a job that involves working with several API projects involving languages you don't know, you can still facilitate the documentation for the projects. However, you'll play more of an editing/publishing role rather than an authoring role.

In many highly technical shops, this editor/publisher role is becoming increasingly common. Engineers will write the technical material, especially the reference documentation, and technical writers will focus more on making sure the content checks all the boxes — that it has an overview, release notes, addresses user tasks, follows the style guide, integrates with the other docs, and so on. You can shape and organize the content, and identify areas where it's deficient or needs expansion, but the ability to add deeper value requires a deeper knowledge of the subject matter.

A lack of more technical knowledge will likely remove some of the value from your role. In How API Documentation Fails, Martin Robillard and Gias Uddin explain:

> Perhaps unsurprisingly, the biggest problems with API documentation were also the ones requiring the most technical expertise to solve. Completing, clarifying, and correcting documentation require deep, authoritative knowledge of the API's implementation. This makes accomplishing these tasks difficult for non-developers or recent contributors to a project.

Without in-depth, authoritative knowledge of the API, it will be challenging to complete, clarify, and correct errors in the content.

The balance between generalist and specialist roles is an ongoing challenge that I'll explore more in the next topic: How much code do you need to know? (p. 664) In short, if you want to solve the biggest problem with API documentation, you'll need to develop more technical expertise in the subject domain.

Consolations for technical writers

As a consolation to this stress of having to navigate multiple programming domains, you can take comfort in the fact that REST APIs (which remember are language agnostic) are becoming more common and are replacing native-library APIs (p. 692). The advantages of providing a universally accessible API using any language platform usually outweigh the specifics you get from a native library API.

For example, when I worked at 41st Parameter (a startup acquired by Experian), the company had a Java, .NET, and C++ API — each implementation did the same thing but in different languages. We also had an SDK for Android and iOS.

Maintaining the same functionality across three separate API platforms was a serious challenge for the company's developers. Not only was it difficult to find skill sets for developers across these three platforms, having multiple code bases made it harder to test and maintain the code. It was three times the amount of work, not to mention three times the amount of documentation.

Additionally, since native library APIs are implemented locally in the developer's code, it was almost impossible to get users to upgrade to the latest version of the API. We had to send out new library files and explain how to upgrade versions, licenses, and other deployment code. If you've ever tried to get a big company with a lengthy deployment process on board with making updates every couple of months to the code they've deployed, you realize how impractical it is. Rolling out a simple update could take 6-12 months or more. During that time, the company is often struggling with a load of bugs and other issues that are setting them back.

It's much more feasible for API development shops to move to a SaaS model using REST, and then create various client implementations that briefly demonstrate how to call the REST API using the different languages. With a REST API, you can update it at any time (hopefully maintaining backward compatibility), and developers can continue using their same deployment code.

As such, you won't be hopelessly lost if you can't navigate these other domains in the programming languages. Your core function will hopefully involve documenting the REST API, with brief docs on the client SDKs mostly authored by the engineers.

That said, one area where REST APIs can be problematic is with devices (for example, smartphones and tablets, devices in cars, streaming media devices). In these cases, calls to REST APIs tend to be slow, so a native library API (such as Android ↗) is used instead.

In the next topic, How much code do you need to know? (p. 664), I'll explore the topic of how much code you need to know and strategies for learning it.

How much code do you need to know?

With developer documentation roles, some level of coding is required. But you don't need to know as much as developers, and acquiring that deep technical knowledge will usually cost you expertise in other areas.

The ideal hybrid: programmer + writer

When faced with these multi-language documentation challenges, hiring managers often search for technical writers who are former programmers to do the tasks. There are a good number of technical writers who were once programmers, and they can command more respect and competition for these developer documentation jobs.

But even developers will not know more than a few languages. Finding a technical writer who commands a high degree of English language fluency in addition to possessing in-depth technical knowledge of Java, Python, C++, .NET, Ruby, in addition to mastering docs tools (p. 551) to facilitate the authoring/publishing process from beginning to end is like finding a unicorn. (In other words, these technical writers don't really exist.)

If you find one of these technical writers, the person is likely making a small fortune in contracting rates and has a near limitless choice of jobs. Companies often list knowledge of multiple programming languages as a requirement, but they realize they'll never find a candidate who is both a William Shakespeare and a Steve Wozniak.

Why does this hybrid individual not exist? In part, it's because the more a person enters into the worldview of computer programming, the more they begin thinking in computer terms and processes. Computers by definition are non-human. The more you develop code, the more your brain's language starts thinking and expressing itself with these non-human, computer-driven gears. Ultimately, you begin communicating less and less to humans using natural speech and fall more into the non-human, mechanical lingo. (I explored this concept more in Reducing the complexity of technical language.)

This mental transformation is both good and bad — good because other engineers in the same computer mindset may better understand you, but bad because anyone who doesn't inhabit that perspective and terminology will already be somewhat lost.

Writers who learned to program

When looking for candidates, would you rather hire a writer who learned programming, or a programmer who learned writing? There are pros and cons to each approach. Let's first examine writers who learn programming, and then in the next section look at the reverse: programmers who learned writing.

In Enough to Be Dangerous: The Joy of Bad Python, Adam Wood argues that tech writers don't need to be expert coders, on par with developers. Learning to code badly (such as is usually the case with writers who learn to code) is often enough to perform the tasks needed for documentation. As such, Wood aligns more with the camp of writers who learned programming. Wood writes:

> You already know how hard it is to go from zero (or even 1) to actually-qualified developer. And you've met too many not-actually-qualified developers to have any interest in that path.
>
> So how do you get started?
>
> By deciding you are not ever going to write any application code. You are not going to be a developer. You are not even going to be a "coder."
>
> You are going to be a technical writer with bad coding skills. (Enough to Be Dangerous: The Joy of Bad Python)

Wood says tech writers who are learning to code frequently underestimate the degree of difficulty in learning code. To reach developer proficiency with production-ready code, tech writers will need to sink much more time than they feasibly can. As such, tech writers shouldn't aspire to the same level as a developer. Instead, they should be content to develop minimal coding ability, or "enough to be dangerous."

James Rhea, in response to my post on Generalist versus Specialist, also says that "adequate" technical knowledge is usually enough to get the job done, and acquiring deeper technical knowledge has somewhat diminishing returns since it means other aspects of documentation will likely be neglected. Rhea writes:

> I wouldn't aim for deep technical knowledge. I would aim for adequate technical knowledge, recognizing that what constitutes adequacy may vary by project, and that technical knowledge ought to grow over time due to immersion in the documentation and exposure to the technology and the industry.
>
> I speculate that the need for writers to have deep technical knowledge diminishes as Tech Comm teams grow in size and as other skills become more important than they are for smaller Tech Comm teams. I'm not claiming that deep technical knowledge is useless. I'm suggesting that (to frame it negatively) neglecting deep technical knowledge has less severe consequences than neglecting content curation, doc tool set, or workflow considerations. (Adding Value as a Technical Writer)

In other words, if you spend excessive amounts of time learning to code, at the expense of tending to other documentation tasks such as shaping information architecture, analyzing user metrics, overseeing translation workflows, developing user personas, ensuring clear navigation, and more, your doc's technical content might improve a bit, but the overall doc site will go downhill.

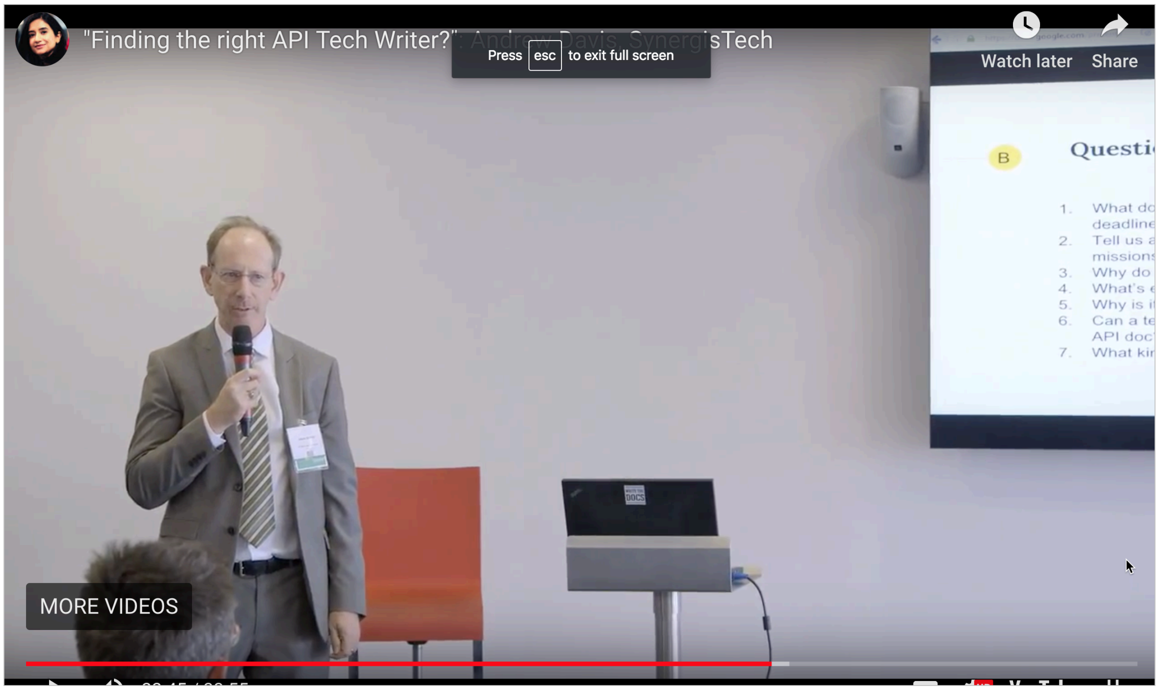
Additionally, while engineers can fill in the deep technical knowledge needed, no one will provide the tech comm tasks in place of a tech writer. As evidence, look at any corporate wiki. Corporate wikis are prime examples of what happens when engineers (or other non-tech writers) write and publish documentation. Some pages might be rich with technical detail, but the degree of ROT (redundant, outdated, trivial content) gets compounded, navigation suffers, clarity gets muddled, and almost no one can find anything.

Programmers who learned to write

Now let's flip to the other side of the argument. What are the advantages of hiring programmers who learned writing? In contrast to Wood and Rhea, James Neiman, an experienced API technical writer, says that tech writers need engineering backgrounds, such as a computer science degree or previous experience as an engineer, to excel in API documentation roles.

Neiman says tech writers often need to look over a developer's shoulder, watching the developer code, or listen to an engineer's brief 15-minute explanation, and then return to their desks to create the documentation. You might need to take the code examples in Java and produce equivalent samples in another language, such as C++, all on your own. In Neiman's view, API technical writers need significant technical depth to excel in this role.

Neiman and Andrew Davis (a recruiter for API tech writers in the Bay area) recently gave a presentation titled Finding the right API Technical Writer at an API conference in London. Their presentation format includes a Q&A exchange between the two. Scrub to around the 22-minute mark for the relevant part:


"Finding the right API Tech Writer?": Andrew Davis, SynergisTech
Press esc to exit full screen
Watch later
Share
B
Questi
MORE VIDEOS

James Neiman on the required technical baseline for API docs" caption="James Neiman on the required technical baseline for API docs

Here's a transcript of two questions in their exchange (cleaned up a bit for readability):

Andrew: What is essential to your relationship with each new client?

James: Being part of the product team, what's essential is communication within the team. Communication is essential to keep up with what is changing (and I expect things to change very rapidly, especially in a disordered environment where people are trying to stand up a product). I also need to earn and retain trust.

Why should I say that? If I'm going to be sitting with an engineering team, I'll need them to let me into their source code so I can modify their source code comments. I'll need to be able to pick an engineer's brain for fifteen minutes and fifteen minutes only — and get meaningful information out of that interview so that I can go and produce the documentation they need and get it right *the first time*. If I don't get it right the first time, I've wasted the engineer's time, and I've wasted the company's money.

Andrew: Can a tech writer without a development background write great API documentation?

James: Absolutely not. There is no way that a busy engineering team has time to train a person without a computer science degree. That's just the reality of it. Engineers at best can speak to you in some version of English, which may or may not be their native language. They don't have a lot of time, and they expect you to finish their thoughts for them. That means that you need to be able to sit next to them and look at how they're coding, and then be able to replicate that and extend it and even create examples.

They may say, "Here's an example. You can extend it, add on these other APIs, work out this use case for us. We haven't had time to finish this." They can say, "Well, let me show you how this works in Objective C; we also support this on Java. Can you create something similar on Java?"

If you don't have that kind of development background, it's unrealistic that you could expect to train, for example, somebody with a masters degree in English (and who is a very intelligent person but otherwise not technical) to do such a thing.

Keep in mind that Davis and Neiman are trying to persuade more European countries to use Synergistech as their recruiting agency to find and hire API tech writers, so they're presenting the need for engineering-savvy tech writers. These super tech-savvy writers are harder to find — hence the need for expert recruiters. Regardless of the agenda, Neiman and Davis argue for a higher level of coding proficiency than Wood or Rhea.

The level of coding knowledge required no doubt depends on the position, environment, and expectations at your company. Perhaps if the tech writer doesn't have more of an engineering background, engineers will just send the tech writer code snippets (p. 464) to paste into the docs. But without the technical acumen to fully understand, test, and integrate the code in meaningful ways, the tech writer will be at the mercy of engineers and their terse explanations or cryptic inline comments. The tech writer's role will be reduced to being an editor/publisher instead of a writer.

In my experience, Neiman's explanation about developers instructing tech writers to create similar code in other languages (based on a 15-minute over-the-shoulder conversation at the engineer's desk) goes too far. Although I've created simple JavaScript code samples (based on a pattern the engineers showed me), I've never been asked to create code samples across other languages. I could auto-generate code snippets for web API requests (using Postman), but to develop code across multiple languages tends to be more of the programmer's responsibility, not the tech writer's.

Neiman goes on to say that in one company, he tested out the code from engineers and found that much of it relied on programs, utilities, or other configurations already set up on the developers' computers. As such, the engineers were blind to the initial setup requirements that users would need to run the code properly. Neiman says this is one danger of simply copying and pasting the code from engineers into documentation. While it may work on the developer's machine, it will often fail for users.

This comment from Neiman does ring more true to me. As I argued for extensively in Testing your API documentation (p. 342), you have to be able to test the endpoints, code samples, and SDKs in order to write and evaluate the documentation. It is usually true that programmers (who set up their machines months ago) have long forgotten or can't even identify all the frameworks, configurations, and other utilities they installed to get something working. The more technical you are, the more powerful of a role you can play in shaping the information.

Neiman is a former engineer and says that during his career, he has probably worked with 20-25 different programming languages. Being able to learn a new language quickly and get up to speed is a key characteristic of his tech comm consulting success, he says.

But in this celebration of technical knowledge, companies make a mistake and assume that these programmers-turned-technical writers can easily handle writing tasks, because c'mon, everyone can write ↗, right? However, without a stronger writing background, these programmers who are now writing might be a lot less proficient in areas where it really matters.

For example, recently I was working with an engineering team on a new voice feature for our product. The engineering team was partly based in India and other places, and they frequently met (during India business hours) to shape a document about the new voice feature. This documentation and the feature were in constant flux, so the team kept iterating on the content over the course of about two weeks after meetings with stakeholders, solutions architects, and other reviewers. After each review, the team sent me the document to edit and publish it for stakeholder reviews.

I wasn't directly embedded with the team, nor was I a dedicated resource for the team. In this role, I simply acted as editor and publisher. But I had to turn around the gibberish they wrote at a rapid rate, usually in 1-2 hours. As this project was one of many I was juggling, I had to quickly restructure and rewrite the content (sometimes touching every sentence) to make it read like a native speaker had written it rather than engineers in another country. During this same time, I was working on rewriting our team website and other writing projects.

I have a tech writing colleague who is a former engineer, and I often wonder if he has the same writing skills to edit this content with the same speed and efficiency that I do. Of course, I shouldn't make judgments, but I'm pretty good at both writing and editing. After all, look at the output on my blog. In just a couple of hours during the evening, I can write a post that is worth reading in the morning. Can engineers who lack writing backgrounds do this? If tech writers are increasingly playing publishing/editing roles instead of developing content directly (because the content is so highly technical that only specialists can create it), then shouldn't companies prioritize writing abilities over technical abilities (to an extent)?

Further, companies who assume that "everyone can write" fail to distinguish the different levels of writing. It's one thing to write coherent sentences in a paragraph or even a single topic, but can the writer read over 20 pages in a documentation system and ensure consistency across all the topics? Can the writer weave together workflows and journeys across these larger systems? Can the writer distill information from a long, complicated process into an intelligible quick reference guide? Writing skills fall along a spectrum, and while most professionals appear somewhere on the spectrum, their skills might not be enough to excel in ways that provide deeper value for documentation.

Overall, technical writers of all stripes are playing generalist roles in increasing ways, and in these generalist roles, strong writing skills rather than specialized knowledge might be more important. For sure, a combination of the two skills — writing and technical expertise — tends to be a knockout punch in the job market.

For an in-depth analysis of the dilemma between being a generalist or specialist, see my essay Be both a generalist and specialist through your technical acuity ↗ in my Simplifying Complexity series.

Wide, not deep understanding of programming

Let's settle the question about the best candidate to hire by finding some middle ground between the two extremes. Clearly, tech writers need to understand code, but they probably don't need to be engineers to the extent that Neiman argues (writing their own code in other languages).

Although you might have client implementations in a variety of programming languages at your company, the implementations will be brief. The core documentation needed will most likely be for the REST API, and you will have a variety of reference implementations or demo apps in these other languages.

You don't need to have *deep* technical knowledge of each of the platforms to provide documentation. You're probably just scratching the surface with each of these client SDKs. As such, your knowledge of programming languages has to be more wide than deep. It will probably be helpful to have a grounding in fundamental programming concepts and familiarity across a smattering of languages instead of in-depth technical knowledge of just one language.

Having broad technical knowledge of multiple programming languages isn't easy to pull off. As soon as you throw yourself into learning one language, the concepts will likely start blending together.

And unless you're immersed in the language on a regular basis, the details may never fully sink in. You'll be like Sisyphus, forever rolling a boulder up a hill (learning a programming language), only to have the boulder roll back down (forgetting what you learned) the following month.

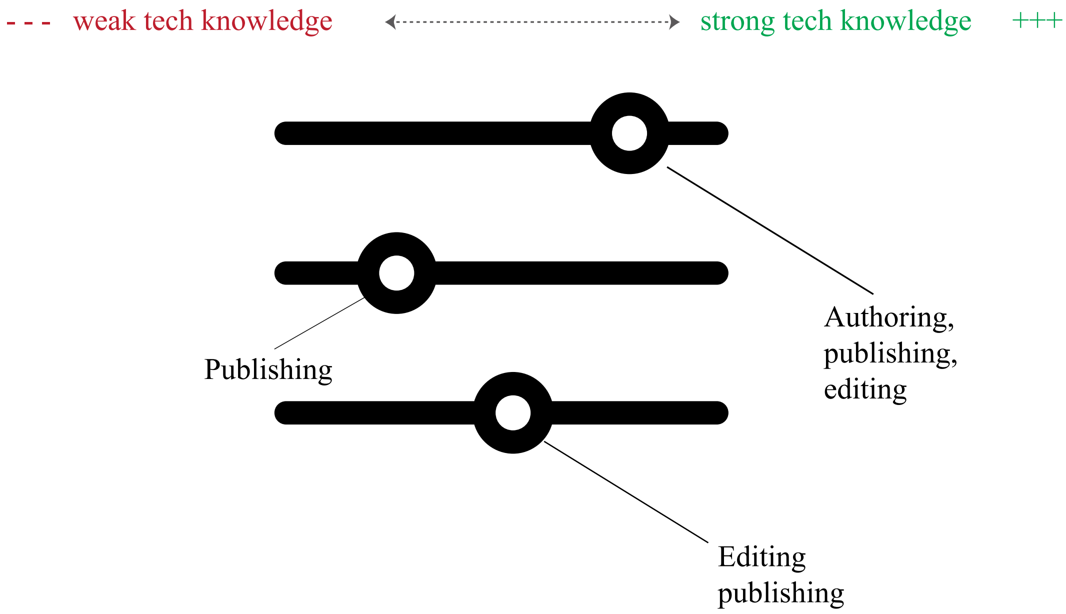
Full immersion is the only way to become fluent in a language, whether referring to programming languages or spoken languages. As such, technical writers are at a disadvantage when it comes to learning programming. To get fully immersed, you might consider focusing on one core programming language (like Java) and only briefly playing around in other languages (like Python, C++, .NET, Ruby, Objective C, and JavaScript).

Of course, you'll need to find a lot of time for this as well. Don't expect to have much time on the job for actually learning a programming language. It's best if you can make learning programming one of your "hobbies."

Strategies to get by in deeply technical situations

Suppose you find yourself deep in APIs that require you to know a lot more technical detail than you currently do (despite your programs of study to learn more)? How can you get by without a deeper knowledge of programming?

Keep in mind that your level of involvement with editing, publishing, and authoring depends on your level of tech knowledge. If you have a strong knowledge of the tech, you can author, edit, and publish. If you have weak tech knowledge, your role might involve publishing only. The following spectrum diagram illustrates this range of involvement:


- - - weak tech knowledge
strong tech knowledge +++
Authoring, publishing, editing
Publishing
Editing publishing

The degree to which you can publish, edit, and author depends on your level of tech knowledge.

If you're stuck in the publishing/editing area, you can interview engineers at length about what's going on in the code (record these discussions — Evernote has a nifty recording feature built-in that I've used multiple times for just this purpose), and then try your best to describe the actions in as clear speech as possible. You can always fall back on the idea that for those users who need Python, the Python code should look somewhat familiar to them. Well-written code should be, in some sense, self-descriptive in what it's doing. Unless there's something odd or non-standard in the approach, engineers fluent in code should be able to get a sense of how the code works.

In your documentation, you'll need to focus on the higher-level information, the "why" behind the approach, highlighting of any non-standard techniques, and the general strategies behind the code. You can get this *why* by asking developers for the information in informational interviews. The details of *what* will either be apparent in the code or can be minimized. (See Code samples and tutorials (p. 464) for details.)

As you decide how much detail to include, remember that even though your audience consists of developers, it doesn't mean they're all experts with every language. For example, the developer may be a Java programmer who knows just enough iOS to implement something on iOS, but for more detailed knowledge, the developer may be depending on code samples in the documentation. Conversely, a developer who has real expertise in iOS might be winging it in Java-land and relying on your documentation to pull off a basic implementation.

More detail in the documentation is always welcome, but you can use a progressive-disclosure approach so that expert users aren't bogged down with novice-level detail. Expandable sections, additional pages, or other ways of grouping the more basic detail (if you can provide it) might be a good approach.

There's a reason developer documentation jobs pay more — the job involves a lot more difficulty and challenges, in addition to technical expertise. At the same time, it's just these challenges that make the job more interesting and rewarding.

Techniques for learning code

The diversity and complexity of programming languages is not an easy problem to solve. To be a successful API technical writer, you'll need to incorporate a regular regiment of technical study. You always have to be learning to survive in this field.

Fortunately, there are many helpful resources (my favorite being O'Reilly's Safari Books Online). If you can work in a couple of hours a day, you'll be surprised at the progress you can make.

The difficulty of learning programming is probably the most strenuous aspect of API documentation. How much programming do you need to know? How much time do you spend learning to code? How much priority should you spend on learning technology?

For example, do you dedicate two hours a day to learning to code in the particular language of the product you're documenting? Should you carve this time out of your employer's time, or your own, or both? How do you get other doc work done, given that meetings and miscellaneous tasks usually eat up another 2 hours of work time? What strategies should you implement to learn code in a way that sticks? What if what your learning has little connection or relevance with the code you're documenting?

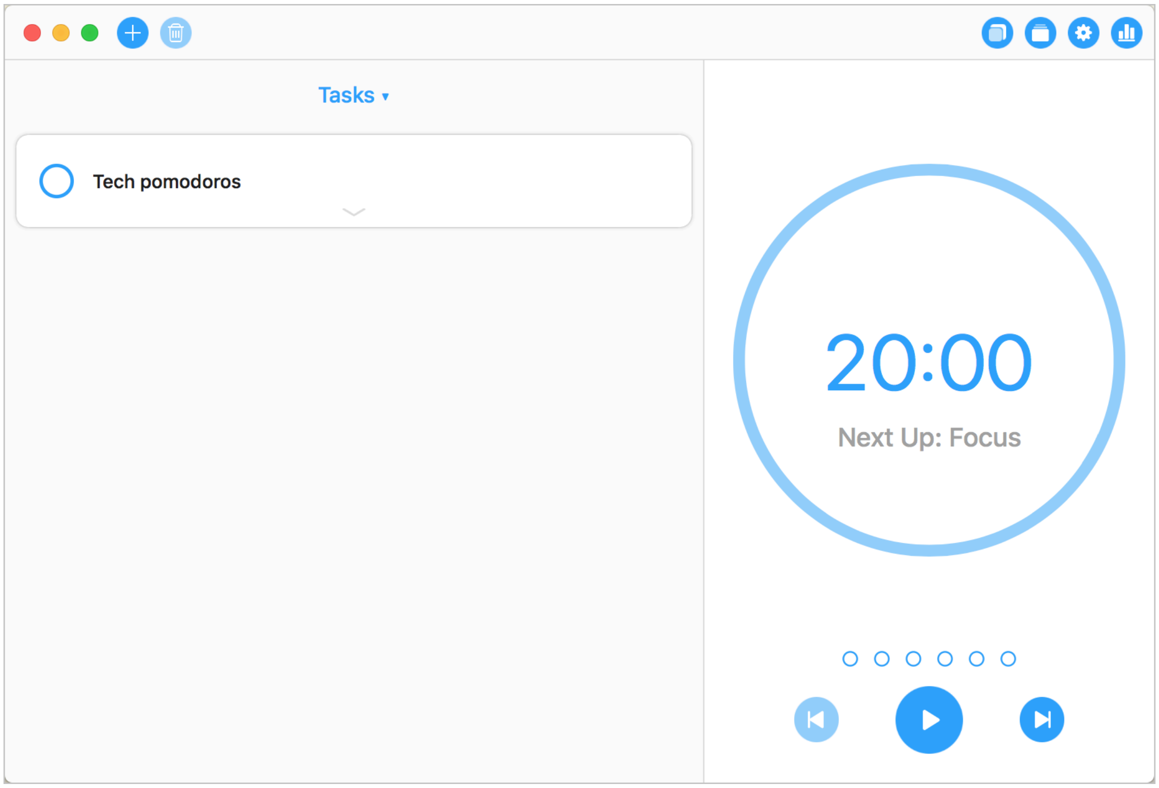
In a post called Strategies for learning technology – podcast recommendation and a poll, I linked to a 10-minute Tech Comm podcast with Amruta Ranade on Learning New Technology and then polled readers to learn a little about their tech learning habits. In the reader responses, most indicated that they *should* spend 30-60 minutes each day learning technology, but most spend between 0-20 min actually doing so. To learn, they use general Google searches. They mostly devote this time to learning tech at work, though some split the time between work at home.

Personally, I think spending 20 minutes a day isn't enough to keep up with the knowledge needs. Sixty minutes is more appropriate, but really, if you want to make progress, you'll need to devote about twice that time. Finding 1-2 hours of time (and motivation) at work to learn it is unlikely. I always feel like I'm not getting enough done as is during work hours — learning technology often feels like a side activity taking me away from my real duties.

Also, the information I need to document in the present moment is usually too advanced to simply learn from watching tutorials on Safari Books or other sources. But I can't just start out consuming advanced material. I have to ramp up through the foundational topics first, and that slow ramp-up feels like a tangent to the real work that needs to get done.

For example, you might need to document the equivalent of Advanced Calculus concepts. But to ramp up on Advanced Calculus, you need to build a foundation with Trigonometry and Algebra. When you spend time studying Trigonometry and Algebra instead of the Advanced Calculus concepts that you need to document, it can feel like you're not making any progress on your documentation.

One strategy I've found to work well is to divide the learning into "pomodoros" (a technique named after tomato kitchen timers). With the Pomodoro Technique, you set a timer for 20 minutes and focus on your learning task for that chunk of time. You can set a goal to complete as many pomodoros a day as you want. After about 1-2 months of these regular pomodoros, you'll be surprised at your progress. I use the Focus app for my pomodoro timer:



The Focus app lets you define and track pomodoros

Additionally, I recommend keeping a list of notes about technical details you struggle with while you're trying to document something, and then during your pomodoros, focus on what's listed in your technical notes.

Even so, this pomodoro technique for focus doesn't solve the problem. It's still hard to squeeze time in for the pomodoros. Whenever I squeeze these into my life, I end up squeezing other activities out.

There are a lot of questions about just how to learn code, and I don't have all the answers. But here's what I know:

- Developer documentation requires familiarity with code, though exactly how much expertise you need is debatable.
- You have to understand explanations from engineers, including the terms used (p. 418). The explanations in your documentation should focus on the *why* more than the *how*.
- You should be able to test code (p. 343) from engineers so that you can identify assumptions that engineers are often blind to.
- To thrive in an API documentation career, you have to incorporate a regiment of continual learning.
- Completing several pomodoros a day over the course of weeks and months can result in significant progress in building your technical understanding.

Is being a generalist a career disappointment?

Technical writers will likely be generalists with the code, not good at developing it themselves but knowing enough to get by, often getting code samples from engineers and explaining the basic functions of the code at a high level.

Some might consider the tech writer's bad coding ability and superficial technical knowledge somewhat disappointing. After all, if you want to excel in your career, usually this means mastering something thoroughly, right? You want to be an expert in your field.

It might seem depressing to realize that your coding knowledge will usually be kindergartner-like in comparison to developers. This disparity positions tech writers more like second-class citizens in the corporation. In a university setting, it's the equivalent of having an associates degree where others have PhDs.

However, take consolation in the fact that your job is not to code but rather to create helpful *documentation*. Creating helpful documentation isn't just about knowing code. There are a hundred other details that factor into the creation of good documentation. As long as you set your goals on creating great documentation, not just on learning to code, you won't feel entirely disappointed in being a bad coder. This perspective doesn't address all the issues, but it does provide some consolation at the end of the day.

For more information about working with code, see these two topics:

- SDKs (p. 480)
- Code samples and tutorials (p. 464)

Let's look at one more topic in this jobs section: Locations for API doc writer jobs (p. 674).

Best locations for API documentation jobs

You're taking this course most likely because you want to break into API documentation. However, if you live in a place where there aren't many API documentation jobs, it will be difficult to find work. In this topic, I'll explain where the API documentation jobs are.

Where are the API jobs

Where are the best locations for API documentation jobs? The 2017-2018 STC Salary Database ↗ (which is based on Bureau of Labor Statistics ↗ data) provides a lot of research about technical writer employment in different parts of the U.S., but nothing that breaks down the numbers by API documentation opportunities.

As such, we'll have to triangulate a bit and make some best guesses about where the API documentation jobs are. The STC Salary Database information gives us a starting point. Here are the "Ten Largest MSAs [Metropolitan Statistical Areas] by Number of Technical Writers" (based on 2017 data, which is the latest data set available):

Metropolitan Statistical Area (MSA)	Technical writers employed
New York-Newark-Jersey City, NY-NJ-PA	3,390
Los Angeles-Long Beach-Anaheim, CA	1,730
San Jose-Sunnyvale-Santa Clara, CA	1,560
Chicago-Naperville-Elgin, IL-IN-WI	1,220
Houston-The Woodlands-Sugar Land, TX	1,040
San Diego-Carlsbad, CA	970
Atlanta-Sandy Springs-Roswell, GA	930
Baltimore-Columbia-Towson, MD	910
Minneapolis-St. Paul-Bloomington, MN-WI	810

Metropolitan Statistical Area (MSA)	Technical writers employed
Austin-Round Rock, TX / Columbus, OH	740

“MSA” stands for “metropolitan statistical area.” Most of the research in the STC Salary Database focuses on MSAs rather than states. Why look at MSAs rather than states as a whole? While California might rank high in technical writer employment, you would have a tough time finding employment in some remote areas, such as Lone Pine or Corning. Employment measurements are more accurate when they’re focused on a specific metropolitan area rather than the state as a whole.

However, it’s hard to filter API data by MSAs, so I’ll use states anyway. The top 10 states (rather than MSAs) for technical writer employment (again, according to the STC Salary Database) are as follows:

State	Number of Employed Technical Writers
California	6,340
Texas	4,360
Virginia	2,840
Massachusetts	2,450
New York	2,390
Florida	2,360
Maryland	2,180
New Jersey	1,980
Ohio	1,770
Pennsylvania	1,540

They aren’t too different from the list of MSAs — California, New York, Texas, New Jersey, Maryland, and Ohio still feature prominently on both lists, so focusing on states probably reflects the MSAs within that state as well.

To try to determine which states have the most density for API-specific documentation jobs, I searched Indeed.com for the term "API" and filtered by state (it wasn't possible to filter by MSA). I performed one set of searches on August 12, 2018, and another on December 26, 2018. Then I averaged the two.

I didn't meticulously examine each hit in the Indeed results for "API" to see if the hit actually related to *documentation jobs* with APIs, or even that it related to APIs (though I did spot check). I assume that the number of documentation jobs will be a relative percentage of the API-related jobs overall. For example, suppose a state has 1,000 hits for "API." If we were to examine each job and filter the results down to "technical writer" jobs that mention APIs, we would probably see a relative percentage (e.g., 5 percent) of jobs related to technical writing. Error rates will probably be the same overall as well.

Admittedly, I'm by no means versed in statistics, and I admit that this method has many flaws for our overall question (*where are the best locations for API documentation jobs?*). For example, a high number of jobs doesn't necessarily mean more jobs will be available if there are just more technical writers competing for those jobs.

To illustrate some of these issues, suppose the state is small and has only about 500 technical writers in the area, but the technical writer growth rate in the area is high as well as the number of "API" hits on Indeed. Is that a better area in which to find a job? Maybe. It depends on how many writers are competing for the jobs — that information would be even harder to gather.

On the other hand, living in a larger tech hub with more variation in the job opportunities and required technical skills might provide more selection and fit for your particular skills. API doc hiring managers usually want candidates familiar with the languages they're targeting, so with more jobs available, you might have a better shot at matching up with one of the roles.

Additionally, just because states have a high employment rate of technical writers doesn't mean it's the best location to find a job. Job growth rates might be a better factor to consider, as it means more new jobs will be available in the area. In the STC Salary Database, high-growth-rate areas are listed in the "Fastest-Growing States for Technical Writers and Export and GDP Performance in 2017" table.

Alternatively, we could have examined the top areas based on industry sectors. The Computer Systems Design and Related Services Industries employs the most technical writers, but if you're focused on another industry, such as medical, different locations might be more optimal for these other industry sectors. California has a high density of computer-focused companies, while New Jersey has more pharmaceutical companies.

Thus, it's hard to say for sure where the best locations are for API doc jobs, especially as they fit your unique profile. At most, in this topic I indicate where some promising areas appear to be, based on the technical writer employment in the area and the prevalence of "API" as a keyword in job postings in the area.

API keyword density by state

Here's the data from my search for the keyword "API" in job searches on Indeed.com, contextualized with the technical writer employment for the state. I sorted by most API hits (because API frequency is what we're optimizing for). In measuring hits, I took the average of two searches for "API" spread out over about six months. Also, I divided the API hits by the employment to give a percentage hinting at the density of API documentation jobs.

State (U.S. only)	Technical writers employed in the state	API average (Aug hits + Dec hits ÷ 2)	# of "API" hits on Indeed (Aug 2018)	# of "API" hits on Indeed (Dec 2018)	% of API hits ÷ employment
California	6,340	5,262	4,983	5540	83%
Texas	4,360	2,569	2,450	2688	59%
New York	2,390	1,864	1,735	1992	78%
Washington State	1,240	1,257	1,283	1230	101%
Virginia	2,840	1,216	1,092	1340	43%
Massachusetts	2,450	1,128	1,080	1176	46%
Illinois	1,520	1047	949	1145	69%
Georgia	1,180	897	879	915	76%
New Jersey	1,980	846.5	800	893	43%
Pennsylvania	1,540	842.5	800	885	55%
Florida	2,360	750.5	722	779	32%
North Carolina	1,480	704	631	777	48%
Colorado	1,220	673	635	711	55%
Ohio	1,770	512.5	466	559	29%
Maryland	2,180	491.5	567	416	23%

State (U.S. only)	Technical writers employed in the state	API average (Aug hits + Dec hits ÷ 2)	# of "API" hits on Indeed (Aug 2018)	# of "API" hits on Indeed (Dec 2018)	% of API hits ÷ employment
Arizona	820	446	404	488	54%
Minnesota	1,030	407.5	389	426	40%
Michigan	1,280	367	351	383	29%
District of Columbia	700	355	355	355	51%
Missouri	1,120	313.5	322	305	28%
Utah	740	287	275	299	39%
Oregon	530	280.5	271	290	53%
Tennessee	480	255.5	212	299	53%
Connecticut	690	248	229	267	36%
Wisconsin	850	245	225	265	29%
Indiana	740	199.5	204	195	27%
Louisiana	180	182.5	167	198	101%
Oklahoma	790	159.5	169	150	20%
Iowa	470	138	130	146	29%
South Carolina	530	133.5	110	157	25%

State (U.S. only)	Technical writers employed in the state	API average (Aug hits + Dec hits ÷ 2)	# of "API" hits on Indeed (Aug 2018)	# of "API" hits on Indeed (Dec 2018)	% of API hits ÷ employment
Alabama	700	109.5	115	104	16%
Nebraska	300	103.5	102	105	35%
Kentucky	250	100	95	105	40%
Kansas	550	99.5	101	98	18%
Delaware	90	83	65	101	92%
Nevada	250	76.5	69	84	31%
Arkansas	180	69.5	56	83	39%
New Hampshire	180	64.5	51	78	36%
Rhode Island	210	62.5	73	52	30%
North Dakota	100	50	46	54	50%
Idaho	250	39.5	35	44	16%
New Mexico	370	38	36	40	10%
West Virginia	90	30	37	22	33%
Maine	70	29	20	38	41%
Alaska	70	27	31	23	39%
South Dakota	100	19	25	13	19%

State (U.S. only)	**Technical writers employed in the state**	**API average (Aug hits + Dec hits ÷ 2)**	**# of "API" hits on Indeed (Aug 2018)**	**# of "API" hits on Indeed (Dec 2018)**	**% of API hits ÷ employment**
Montana	60	18	17	19	30%
Mississippi	120	17.5	7	28	15%
Wyoming	30	15	12	17	50%
Vermont	160	13.5	8	19	8%
Hawaii	70	10.5	15	6	15%
Puerto Rico	110	7	4	10	6%

If “API” is a more common term for the area, I assume more technical writer jobs will involve working with APIs in those areas. Based on this assumption, California, Texas, Virginia, Massachusetts, New York, Florida, Maryland, New Jersey, Ohio, and Pennsylvania are promising areas for API documentation jobs for technical writers.

This superficial query doesn’t indicate whether there are more tech comm jobs overall for these states. For example, Michigan might have a lot more general tech writing jobs than API doc jobs, even though it ranks far down in the above table.

To get more granular with details, such as job losses or gains, the hottest MSAs for each state, the best industry sectors, the highest salaries, etc., consult the STC Salary Database and mine the data you want.

I want to note some issues. In my initial searches for District of Columbia, I failed to specify “exact location” and as a result, the original hits (which allow a 25-mile radius by default) included many surrounding states such as Virginia, Maryland, and others. I revised the search in January 2019.

Also, I’m not sure why Washington State, Louisiana, and Delaware are getting so many hits for APIs. I think “API” might have another meaning in some of these states (e.g., “API CNC Machinist,” “Familiar with OSHA, API, NFPA and other regulations and standards,” “Basic knowledge of API RP 53 and 30 CFR Part 250,” etc.). Like I said, I didn’t sort through hundreds of results to make sure API referred to “Application Programming Interface” in every counted hit.

At any rate, the top 10 states with the highest percentages (dividing API hits by employment) are Washington State, Louisiana, Delaware, California, New York, Georgia, Illinois, Texas, Colorado, and Arizona.

Should you move to California?

Since I’m currently in California (the “San Jose-Sunnyvale-Santa Clara” MSA), and California frequently appears at the top of these employment and API lists, I’ll comment a bit on API jobs in this location. California (specifically Silicon Valley, which roughly spans from San Francisco to San Jose) continues to be a hub (if not *the* hub) for API technical writer jobs.

The number of tech writers in the area fluctuates quite a bit. Here’s the data from the last several Salary Database for San Jose-Sunnyvale-Santa Clara, CA, starting in 2007 (the first year the STC started publishing the report):

Year	Tech writers employed	
2006	1,370	
2007	1,350	
2008	1,350	
2009	1,510	
2010	1,340	
2011	1,340	
2012	1,200	
2013	no data	
2014	1,360	
2015	1,360	
2016	1,160	
2017	1,560	

Overall, San Jose-Sunnyvale-Santa Clara is a hot area for tech writer jobs. The employment levels have been relatively flat for the past decade but jumped up 34% this past year. I can certainly see the effects of the job growth in this area. I get emails from recruiters pinging me several times a week, trying to fill positions. At my own company, I've seen how difficult it is to find qualified candidates for developer doc positions. At times, it seems like the candidates just aren't here.

Should you move to California (or stay here if you're already in California)?

However, California has some drawbacks, mainly with housing and traffic. To live in Silicon Valley, you likely need the dual income of two working professionals to survive financially. Alternatively, you probably need to be single and willing to share a house with roommates. Or just be willing to live in an older, smaller house in an ugly part of town. Traffic along the 101 can also crawl at a snail's pace during rush hour.

That said, given the increased exodus of tech professionals in the Silicon Valley area (some estimate that 46% are leaving ↗ because housing is so unaffordable), the need and opportunity for tech writers will be even greater in Silicon Valley.

So if you're willing to sacrifice a few comforts related to housing — like being willing to live in a 1,000 square-foot house for $3k/month rent instead of owning a $3,000 square-foot house for $1,500/month mortgage — your future could be bright in California. California has many other perks as well — beautiful beaches, mountains with awe-inspiring Sequoias and Redwoods, free city-wide wifi, bike-friendly paths, abundant parks and good schools, year-round good weather, laid-back attitudes, and more.

But living here is a tradeoff. In Tech Comm and The State of Urbanization, Danielle Villegas, frustrated by the lack of tech writer jobs in the New Jersey area but not ready to give up her nice home and garden for the urban life, writes:

> Why would I want to try to get a studio apartment in San Francisco or Silicon Valley or New York City for USD $1-2 million when I can get a three-to-four bedroom house in a nice neighborhood, have some green space/a garden, a good school district for my child, for a fraction of that? Why should I have to sacrifice my time with my family and other obligations I have to my community by commuting four hours round trip every day, and sacrificing my physical and mental well-being at the same time?

It's a tradeoff that might not make sense for the lifestyle you want. For me, I moved to California (from Utah) after our tech writing team at a Utah company was laid off. I received a generous severance package and decided to move to Silicon Valley to be in an environment that has an abundance of jobs. I have four daughters and am the primary breadwinner in my family, so it was important to be located in a place where jobs were more abundant, stable, and where salaries were more lucrative. I was willing to trade my spacious Utah house and the green garden (okay, I actually hate gardening) for the chance to work in a thriving tech space.

Having been here for five years, I honestly love it. For starters, there are a plethora of tech writing jobs here if you're qualified. If you have a few years of experience writing developer docs and are familiar with some technical languages or frameworks, many companies will gladly open their doors. Startups offer opportunities to build doc departments and tool workflows from the ground up as well. It might be tough to land your *first* tech writing job here, but once you get some experience, doors open.

Working in large companies like Amazon, Google, Facebook, Linkedin, Microsoft, and more can be invigorating. I love working with mainstream commercial products like Fire TV — literally, it seems like every day I see an article talking about Amazon products or the company in some way. It's interesting to be working in a space that has influence, which is frequently in the news (for good or bad), and which is at the forefront of technology.

These companies are shaping culture and so many other details of our lives. For example, even Fire TV may seem like a simple device, but it's one of the technology products fueling the cord-cutter revolution that is disrupting the cable industry. Amazon is now taking TV to another category of experience by integrating Alexa into the experience (providing more of a hands-free interaction using natural language).

Alternatives to California

Before I moved to Santa Clara, I considered two alternatives besides the Bay area: Austin and Seattle. Texas offers a much more affordable housing market and many tech jobs. For 2017, the "Austin-Round Rock, TX" MSA employment numbers jumped up 34.5% (for a total of 740 tech writers) and in "Houston-The Woodlands-Sugar Land, TX," the employment jumped up 9.5% (for a total of 1,040 tech writers).

Seattle (specifically "Seattle-Bellevue-Everett, WA") also has an abundance of jobs (an employment increase of 5.8%, for a total of 910 tech writers). And the real estate market is slightly better than the Bay area but still expensive (food seems to cost more Seattle too).

Traffic in all of these cities is pretty horrible, but you might be able to avoid this by becoming a bicycle commuter (as I have). In the end, I chose California because it seemed like the Ivy League of technology spaces, so I wanted to experience it for myself.

Will I stay here forever? Probably not. I'll probably never be able to afford a home here, but for the time being, accruing experience at some well-known companies seems like a worthwhile investment.

Overall, tech companies will continue to grow and expand outside of Silicon Valley. There's not enough space in Silicon Valley, and the number of engineers becomes fewer and fewer as companies grow and expand. Many say that Silicon Valley's days are over because it's no longer practical for a startup to launch in the conditions here. So if you want to come to California to work as a technical writer, great. But many other locations might be more ideal.

For ideas on alternative areas, see the "Fastest-Growing States for Technical Writers and Export and GDP Performance in 2017" section of the Salary Database. Michigan had a 28.0% increase in tech writer employment. Georgia had a 19.2% increase. New Mexico had a 23.3% increase.

Without looking at averages over several years, it's hard to know whether these boosts are temporary anomalies or indicative of a continuous upward trajectory. To give better context to any growth percentages, the Salary Database says to look at the state's overall economic growth, not just their tech comm job market. The report notes, "State employment gains are often correlated with the state's economy. A growing economy will typically prompt new hires in nearly all industries."

Overall, if you're serious about choosing locations for jobs, consider these factors:

- Number of tech writers employed
- Abundance of API doc opportunities
- The industry
- Cost of living
- Health of the state's economy
- Companies in the area you want to work for

How to conduct a simple test

To spot-check the job opportunities in an area, select a few cities where you might like to live, set up some job alerts on Indeed.com for those areas, and then monitor the frequency of jobs there over the next few months.

Indeed will send you a daily job alert whenever there are new jobs in that area. Over a period of time, you can consistently see how many new tech writing jobs are popping up in a city. I did this when I was initially considering where to live, and without question, there were about 3-4 times more technical writing jobs appearing in San Jose than Portland, and 2-3 times more than in Seattle.

In these searches, I didn't run comparisons for more cities, nor did I add "API" into the job title. But you can set up multiple searches and assess the results.

If you're relocating, let companies know when you'll be in the area. Set up as many interviews as you can during the same week. When companies see you arrive on site in person, it can allay fears about hiring someone out of state.

Conclusion

Much more could be written about where to live for API tech jobs. Overall, if you want to find a job in API documentation, target these popular areas: California, Texas, Virginia, Massachusetts, New York, Florida, Maryland, New Jersey, Ohio, and Pennsylvania. On the other hand, if you want to start a farm or ranch, move to Wyoming.

Activity: Look at API documentation jobs and requirements

In this activity, you'll get a sense of the skills needed for the jobs in your location, and then draw up a plan.

1. Go to indeed.com ↗.
2. In the **Where** field, type your desired location.
3. Search for "API technical writer" or some combination of API + technical writer + developer documentation jobs.
4. Read the descriptions of 5 jobs.
5. Note a few of the salient requirements for these jobs.
6. Assess where you're currently at with the following:
   - Portfolio with writing samples that include developer documentation
   - Technical knowledge related to developer domain
   - Experience writing developer documentation
7. Make a plan for how you'll match up your portfolio, tech knowledge, and experience related to what these job descriptions are asking for. You might need to dedicate more time to an open-source documentation project (p. 686) to build up the needed skills.

Activity: Find an open-source project

To break into API documentation, you need to start thinking about API documentation samples in your portfolio. Your portfolio is key to Getting an API documentation job and thriving (p. 659). Without a portfolio that contains compelling API documentation samples, it will be tough to get a job in API documentation.

Avoiding a catch-22

Let’s assume you don’t have any experience in API documentation, but you’re trying to get an API documentation job. Employers will be willing to overlook experience if you can demonstrate API documentation writing samples. But how will you get API doc writing samples without an API doc job? And without API doc samples, how can you get an API doc job? This can seem like an impossible situation.

Getting around this catch-22 is simple: you create these API doc samples through open-source projects that you contribute to. This is where the activities in this course become essential.

Rather than merely completing modules and tracking your progress toward the course’s completion, the activities you do will help build up your portfolio with API documentation samples, helping you progress to the goal of either obtaining an API documentation job or hitting a home run on an API doc project in your current role.

Finding an open-source API project

If you’ve already got an API project through your work, or if you’re an engineer working on an API project, great, just select your existing API for the course activities. However, if you’re breaking into API doc or building your API doc skills from the ground up, you’ll need to find an open-source API documentation project to contribute to.

Finding the right project can be challenging, but it is critical to your portfolio and your success in breaking into API documentation. Fortunately, almost all open-source projects use GitHub, and GitHub provides various tags for documentation and “help wanted” in order to attract volunteers. (The task is so common, GitHub provides advice for finding open-source projects.)

The ideal open-source API project should meet the following criteria. The project should:

- Involve a REST API (not a library-based API (p. 692) or some other developer tool that isn’t an API).
- Have some documentation needs.
- Not be so technical that it’s beyond your ability to learn it. (If you already have familiarity with a programming language, you might target projects that focus on that language.)
- Be active, with a somewhat recent commit.

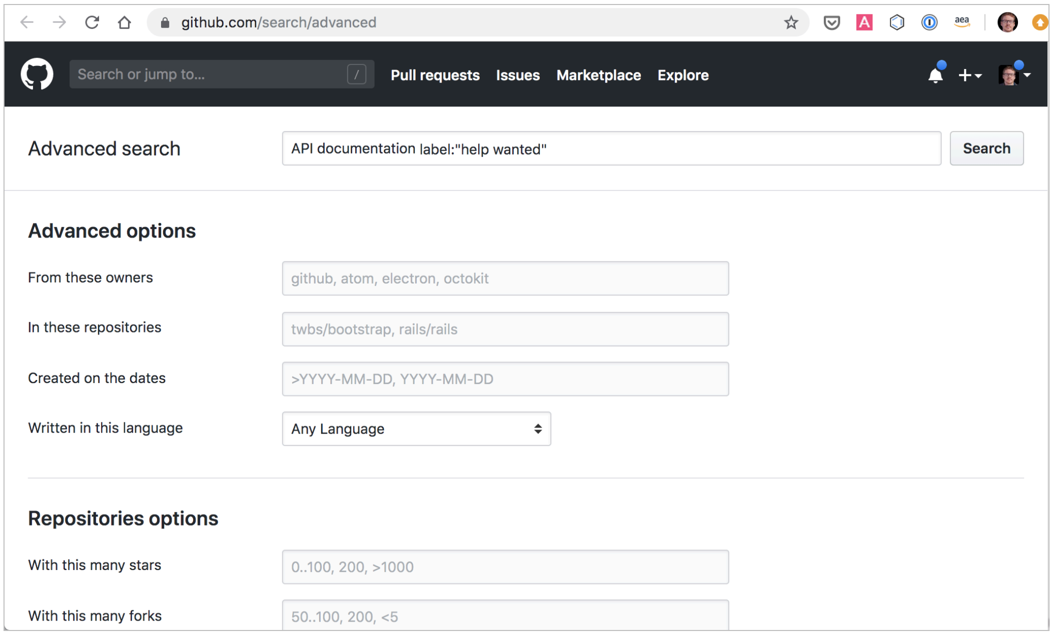
Activity: Find an open-source project with API doc needs

To find an open-source project with API doc needs:

1. Go to the GitHub Advanced Search.
2. Under the **Issues Options** section, in the **With the labels** row, type `help wanted`. This is a standard tag that teams use to attract volunteers to their project (but some teams that need help might not use it).

   Scroll to the top and notice that `label: "help wanted"` automatically populates in the field.

3. In this Advanced Search box at the top, add some additional keywords (such as `API documentation`) as well:



Advanced search: API documentation label:"help wanted"

Written in this language: Any Language

4. Click **Search** and browse the results.

   In the results, you might want to look for a *REST API* project (rather than a native-library API (p. 692) such as a Java API). Developers don't often distinguish between the two with requests related to API documentation. If you see doc requests related to Java, C++, JavaScript, or some other programming-specific framework (with no identifiable endpoints), it's probably not a REST API. However, working on such a project could be equally fulfilling as a learning opportunity. It depends on what you're interested in.

   As you browse the results, are there any projects that look interesting or promising? If so, great. If not, adjust some of the keywords and keep looking.

5. If searching GitHub doesn't yield any appropriate projects, try the following resources:

   - Trending GitHub projects
   - Crowdforge
   - Up for Grabs
   - Bus Factor
   - Code Triage
   - Changelog
   - 24-hour Pull Requests

   Note: You could spend a long time evaluating and deciding on open-source projects. For this activity, it's okay if you focus on a project that looks only mildly interesting. You don't need to commit to it. You can always change it later.

6. After selecting a project, make notes on the following:

- Does the project involve a REST API?
- How does the project tag documentation-related issues? For example, does it use the “documentation” label?
- Identify the current state of the project’s documentation. Are the docs robust, skimpy, nonexistent, extensive?
- How active is the project? (What is the frequency of commits?)
- How many contributors does the project have?

You don’t have to actually reach out or interact with the team yet. You’re just gathering information and analyzing documentation needs here.

Recognizing the type of API used in the project

When you look for API projects, recognize that there are many different types of APIs. Many of the APIs you run across might be native library APIs (p. 692), which don’t use web protocols to make requests and responses (as REST APIs do) but rather involve incorporating a language-specific library into the project. If the API seems to focus on a particular language, and the API documentation looks auto-generated, it’s probably a native-library API.

On the other hand, if the project’s documentation contains these core reference sections for their endpoints, it’s probably a REST API.

Contributing will require Git skills

When you later contribute to the open-source project, you will need to understand the basic Pull request Git workflow (p. 586). Understanding the Git workflow might require you to ramp up on Git tutorials a bit first, but there’s no better way to learn Git than by actively using it in a real project scenario.

Don’t worry so much about Git now. You can learn these skills later when you have content you’re ready to contribute. For now, just find a project.

Don’t undervalue your doc skills

You may think that it’s too early to even think about joining let alone contributing to an API documentation project, especially when you’re learning. When you interact with the open-source team, you might feel intimidated that you don’t have any programming skills.

However, don’t undervalue your role as a contributor to documentation (regardless of the contribution). Open-source projects suffer greatly from lousy documentation. In GitHub Survey: Open Source Is Popular, Plagued by Poor Docs and Rude People, David Ramel summarizes findings from the 2017 GitHub Survey:

> Incomplete or outdated documentation is a pervasive problem, observed by 93 percent of respondents, yet 60 percent of contributors say they rarely or never contribute to documentation.

Also check out Open source documentation is bad, but proprietary software is worse by Matt Asay as well. Asay highlights the documentation results from the same GitHub survey:

> 93% of respondents gnashed their teeth over shoddy documentation but also admitted to doing virtually nothing to improve the situation. … If you think this deeply felt need for documentation would motivate more developers to pitch in and help, you'd be wrong: 60% of developers can't be bothered to contribute documentation.

So yeah, as a technical writer, you may not be fixing bugs in the code or developing new features, but your documentation role is still highly needed and valued. You are a rare bird in the forest here.

I know the value of the doc role intimately from my own experience in contributing to open source doc projects. At one point, before focusing my energy on this API doc course, I contributed several tutorials to the Jekyll docs ↗. I added instructions that included a lot of new content and even added a Tutorials section ↗.

I thought other developers would continue creating new tutorials in a steady stream, but they didn't. Developers tend to add little snippets of documentation to pages — a sentence here, a paragraph there, an update here, a correction there. You will rarely find someone who writes a new article or tutorial from scratch. When there's a new release, there often aren't release notes — there are merely links to (cryptic) GitHub issue logs.

As such, you should feel confident about the value you can bring to an open-source project. You're creating much-needed documentation for the project.

More reading

See the following for more information on finding an open-source project:

- How to choose (and contribute to) your first open-source project ↗
- Contribute to open-source projects through documentation ↗

For a tutorial on pull requests workflows with GitHub projects, see Pull request workflows through GitHub (p. 586).

Next steps

After you find an open-source project, go to the next activity: Evaluate API reference docs for core elements (p. 177).

Activity: Create or fix an API reference documentation topic

This part of the activity might be challenging, but here is where you'll start building some examples for your portfolio.

1. In the same project as before, identify one of the API reference topics that needs help. (If the API has a new reference endpoint to document, focus on this endpoint.)
2. Edit the topic to improve it. (If it's a new endpoint, write the documentation for it.)
3. Create a pull request (p. 586) and contribute your edits to the project.

Chapter 12: Native library APIs

Native library APIs refer to Java, C++, or other programming-specific APIs. In this model, rather than making requests across the web for the information, you download a library of code and integrate it into your project. The library is compiled directly into your application's build (rather than accessed through web protocols as with REST APIs). Although this type of API is less common, I include it here in part to clarify what makes REST APIs so different from native library APIs.

Chapter sections:

Overview of native library APIs 692
Get the sample Java project 695
Java crash course 701
Activity: Generate a Javadoc from a sample project 707
Javadoc tags 712
Explore the Javadoc output 720
Make edits to Javadoc tags 722
Doxygen, a document generator mainly for C++ 723
Create non-ref docs with native library APIs 726

Overview of native library APIs

In most of this course, we focused on REST APIs (p. 36). In this section, we'll explore native library APIs, which are more common when building native apps that you install on devices (such as Android or iOS devices). Understanding native library APIs will also help clarify what makes REST APIs so different.

Characteristics of native library APIs

Native library APIs (also called class-based APIs or just APIs) are notably different in the following ways:

- **Installed locally**. Native library APIs are installed locally, compiled into the programmer's code as an additional library. The programmer can then use the classes, methods, or other functions available in the library. (The API part refers to the *public* classes the developers use to access the functions in the library. There are probably lots of helper and utility classes in the Java library that aren't public, but users will only interact with the public classes.)
- **No requests and responses**. The classes in native library APIs don't use HTTP protocol, nor are there requests and responses sent across the web. Native library APIs consist of a collection of classes or functions that enhance your existing project with more capabilities. The APIs are entirely local, with no web communication.
- **Language specific**. Native library APIs are language specific. There are as many different types of APIs as there are programming languages, more or less. You can have a Java API, Python API, C++ API, C# or .NET API, JavaScript API, and so on.
- **Requires some programming knowledge to document**. To understand how the API works, you need to have a general understanding of the programming language of the API. You don't need to be a programmer, but you should be familiar with the nuts and bolts of the programming language, the correct terms, how the different parts fit together, and how developers will use the API.

We will focus this section on Java APIs because they're probably one of the most common. However, many of the concepts and code conventions mentioned here will apply to the other languages, with minor differences.

Do you have to be a programmer to document native library APIs?

Because native library APIs are so dependent on a specific programming language, the documentation is usually written or driven by engineers rather than generalist technical writers. Native library APIs are one area where it helps to be a former software engineer when writing documentation.

Even so, you don't need to be a programmer. You just need a minimal understanding of the language. Technical writers can contribute a lot here concerning style, consistency, clarity, tagging, and overall professionalism.

My approach to teaching native library API doc

There are many books and online resources you can consult to learn a specific programming language. This section of the course will not try to teach you Java (p. 701) in any comprehensive way. However, to understand a bit about Java API documentation (which uses a document generator called Javadoc ↗), you will need some understanding of Java.

To keep the focus on API documentation, we'll take a documentation-centric approach as we explore Java. You'll learn the various parts of Java by looking at a specific Javadoc file and sorting through the main components.

What you need to install

For this part of the course, you need to install the Java Development Kit (JDK) (p. 693) and the Eclipse IDE for Java Developers (p. 693).

Java Development Kit (JDK)

To check whether you have the Java Development Kit installed, do the following:

- On Mac: Open Terminal and type `java -version`.
- On Windows: Open a Command Prompt and type `where java`.

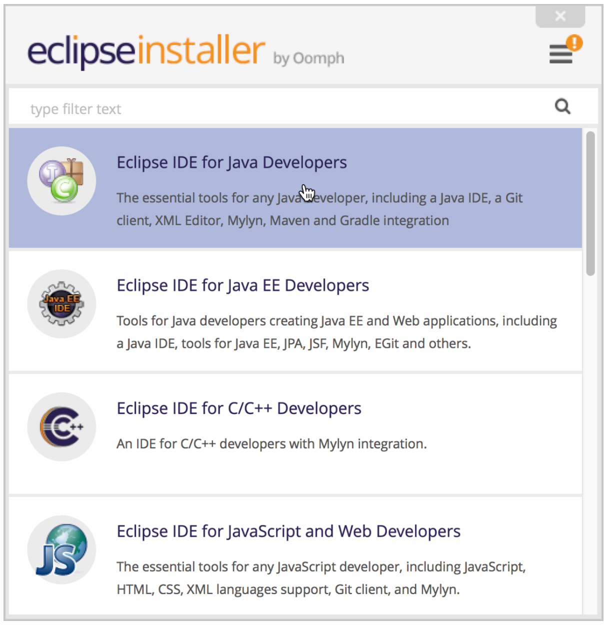
If it's installed, you should see a message somewhat like this:

```
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)
```

If it's not installed, you can download the JDK here. Click the Java button and then select the appropriate download for your machine.

Eclipse IDE for Java Developers

Download and install the Eclipse IDE for Java Developers.



Eclipse IDE for Java Developers

The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Mylyn, Maven and Gradle integration

Eclipse IDE for Java EE Developers

Tools for Java developers creating Java EE and Web applications, including a Java IDE, tools for Java EE, JPA, JSF, Mylyn, EGit and others.

Eclipse IDE for C/C++ Developers

An IDE for C/C++ developers with Mylyn integration.

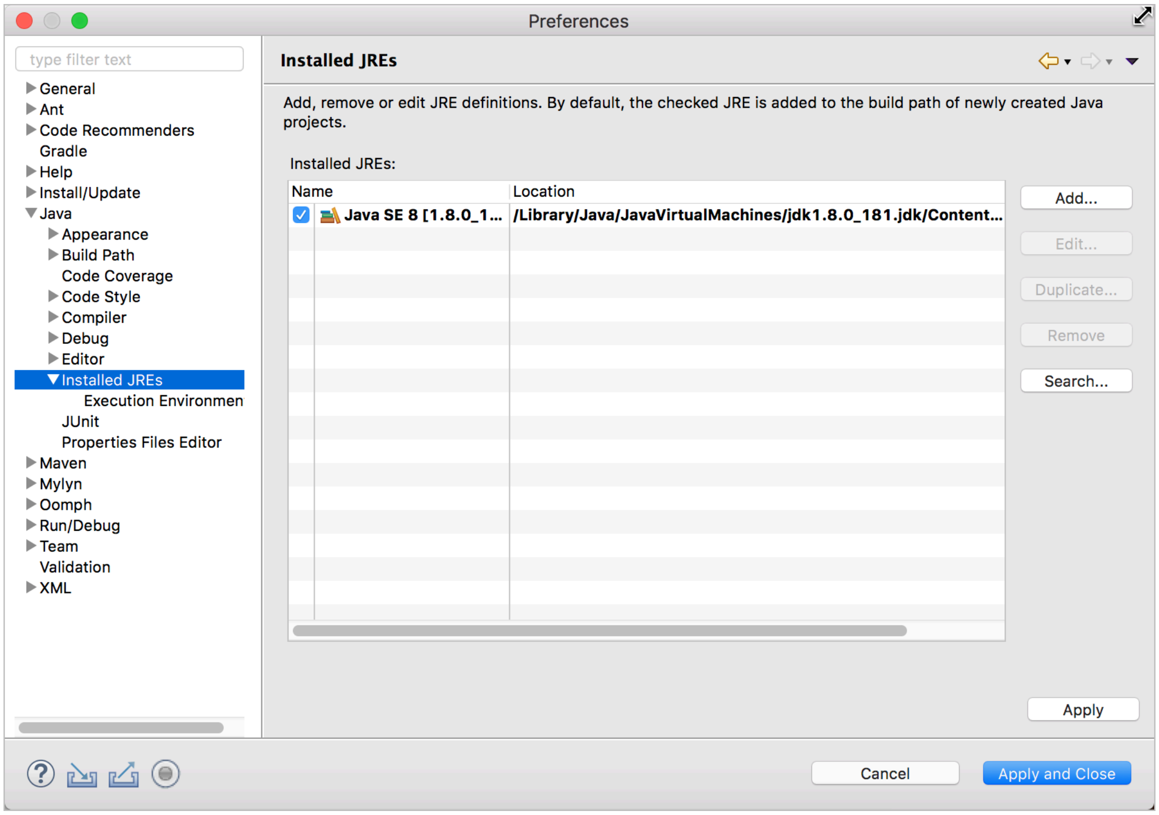
Eclipse IDE for JavaScript and Web Developers

The essential tools for any JavaScript developer, including JavaScript, HTML, CSS, XML languages support, Git client, and Mylyn.

Eclipse installer

Start Eclipse. When prompted to select a workspace, select the default location and click **Launch**. Close the welcome tab.

To make sure Eclipse is configured to use version 1.8, go to **Eclipse > Preferences** and then **Java > Installed JREs**.



Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path of newly created Java projects.

Installed JREs:

Name	Location
☑ Java SE 8 [1.8.0_1...	/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Content...

Installed JRE settings

If 1.8 isn't selected, browse to your install directory (on Mac, it's `/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home`) and select it.

(Since we'll just be using Java within the context of Eclipse, Windows users don't need to add Java to their classpath. But if you want to be able to compile Java from the command line, you would also need to do this.)

Get the sample Java project

To understand the documentation for Java APIs, it helps to see the context of what the documentation describes. As such, I created a simple little Java application (called sample-java-project) to demonstrate how the various tags get rendered into the Javadoc.

Sample Java Project

The sample Java project (available at github.com/tomjoht/sample-java-project) is a little Java application about different tools that a coyote will use to capture a roadrunner. There are two classes (`ACMESmartphone` and `Dynamite`) and another class file called `App` that references the classes.

This program only prints little messages to the console, but it's simple enough to be instructive in its purpose. The purpose of the app is to demonstrate different doc tags, their placement, and how they get rendered in the Javadoc.

Download and open the project in Eclipse

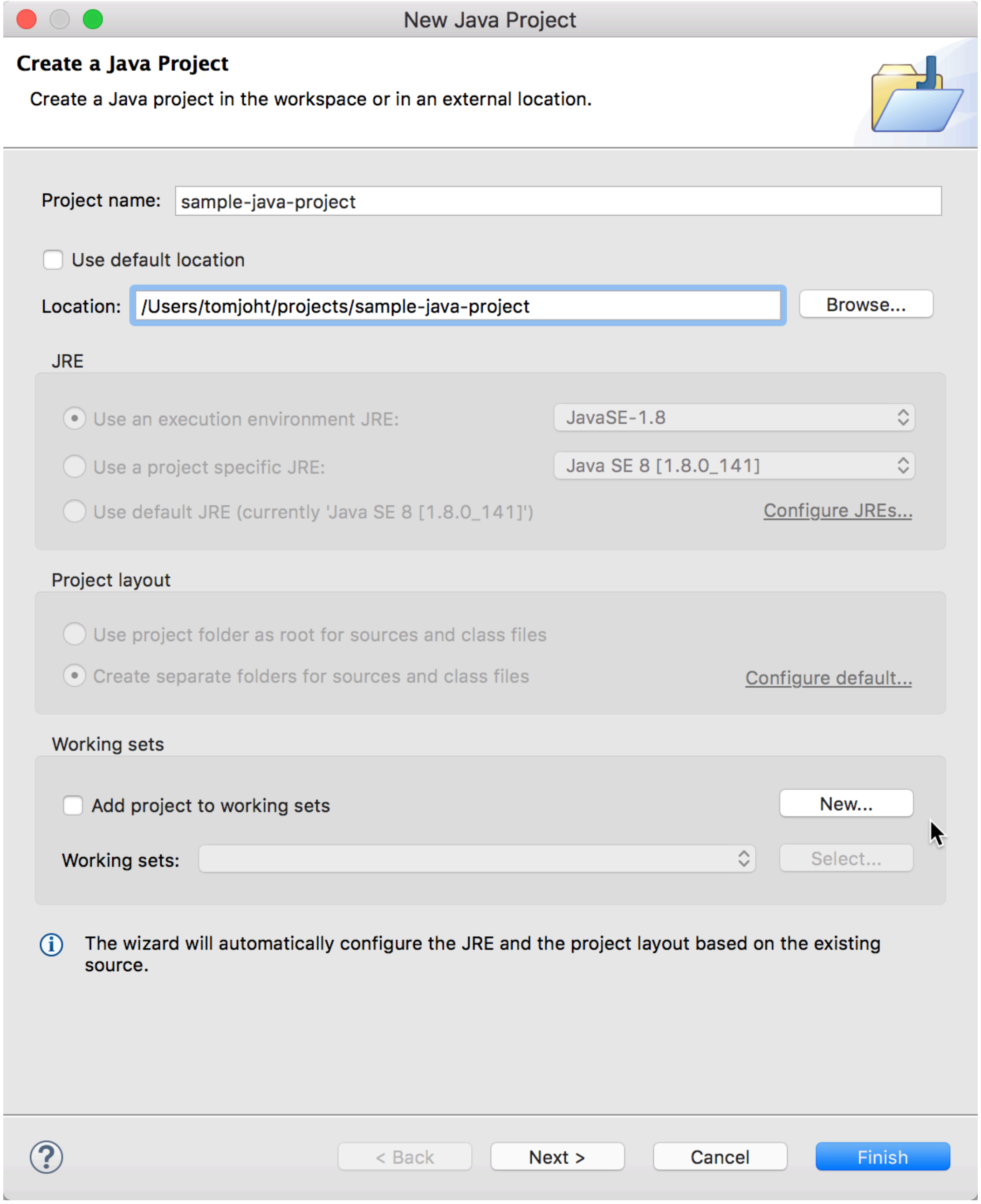
One of your immediate challenges to editing Javadoc will be to get the source code into your IDE and open the project.

1. Go to the sample-java-project and clone the project using version control.

   ```
   git clone https://github.com/tomjoht/sample-java-project
   ```

   (If you need to review version control basics, see Version control systems (such as Git) (p. 562).)

2. If you haven't already downloaded and installed Eclipse (p. 693), do so.
3. Open Eclipse and go to **File > New > Java Project**.
4. Clear the **Use default location** check box, and then click **Browse** and select the folder where you cloned the Github project. Click **Open**.


New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: sample-java-project

☐ Use default location

Location: /Users/tomjoht/projects/sample-java-project

JRE

● Use an execution environment JRE: JavaSE-1.8

○ Use a project specific JRE: Java SE 8 [1.8.0_141]

○ Use default JRE (currently 'Java SE 8 [1.8.0_141]')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

Working sets

☐ Add project to working sets

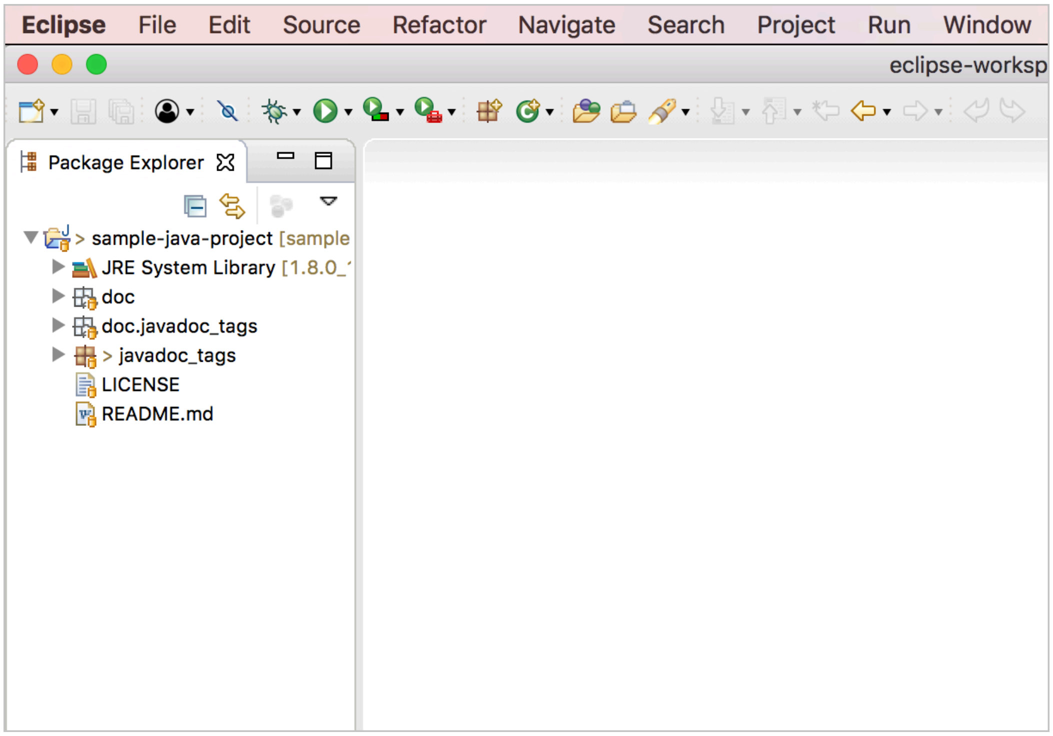
Working sets:

The wizard will automatically configure the JRE and the project layout based on the existing source.

Import existing Java project

5. Click **Finish** to close the New Java Project dialog box.

   The Java files should be visible in the left pane (Package Explorer) within your Eclipse IDE.

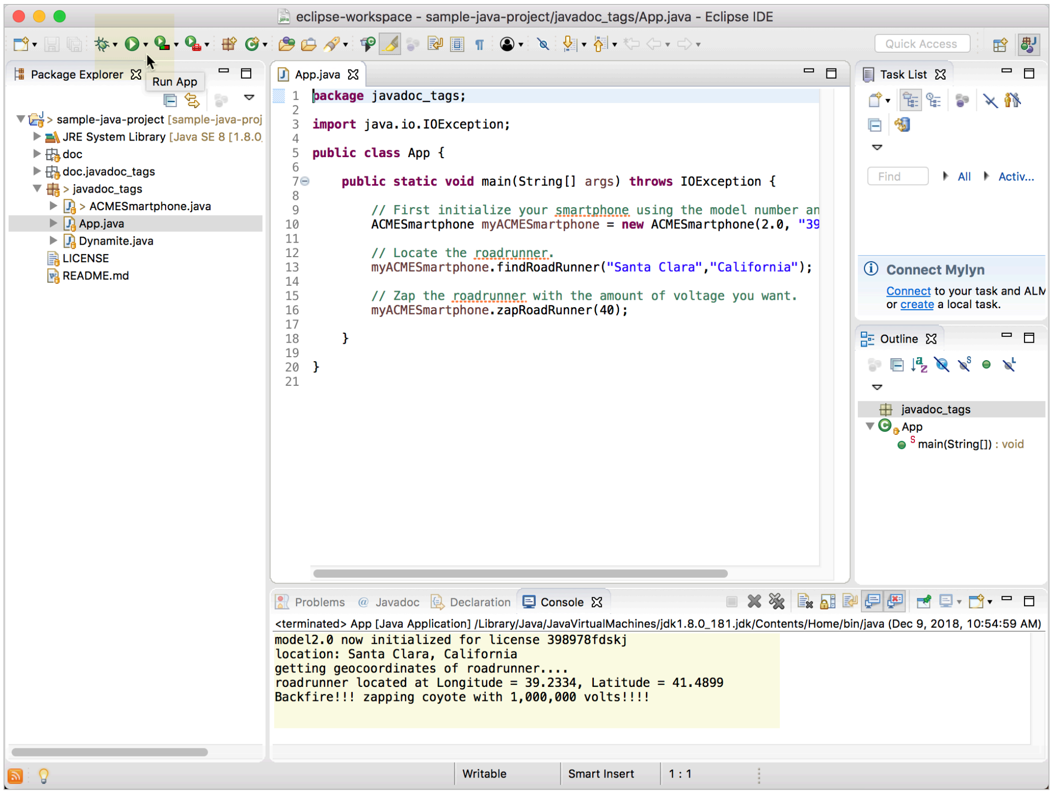


Eclipse File View

Play with the sample Java project

This Java app doesn’t do much. Its only purpose is to create some classes that demonstrate some Javadoc annotations. But for fun, run the app.

1. In the Package Explorer, expand **javadoc_tags**.
2. Double-click the **App.java** file to open it.
3. Click the **Run App** button :



```java
package javadoc_tags;

import java.io.IOException;

public class App {

    public static void main(String[] args) throws IOException {

        // First initialize your smartphone using the model number an
        ACMESmartphone myACMESmartphone = new ACMESmartphone(2.0, "39

        // Locate the roadrunner.
        myACMESmartphone.findRoadRunner("Santa Clara","California");

        // Zap the roadrunner with the amount of voltage you want.
        myACMESmartphone.zapRoadRunner(40);

    }

}
```

```
<terminated> App [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java (Dec 9, 2018, 10:54:59 AM)
model2.0 now initialized for license 398978fdskj
location: Santa Clara, California
getting geocoordinates of roadrunner....
roadrunner located at Longitude = 39.2334, Latitude = 41.4899
Backfire!!! zapping coyote with 1,000,000 volts!!!!
```

Running the app in Eclipse

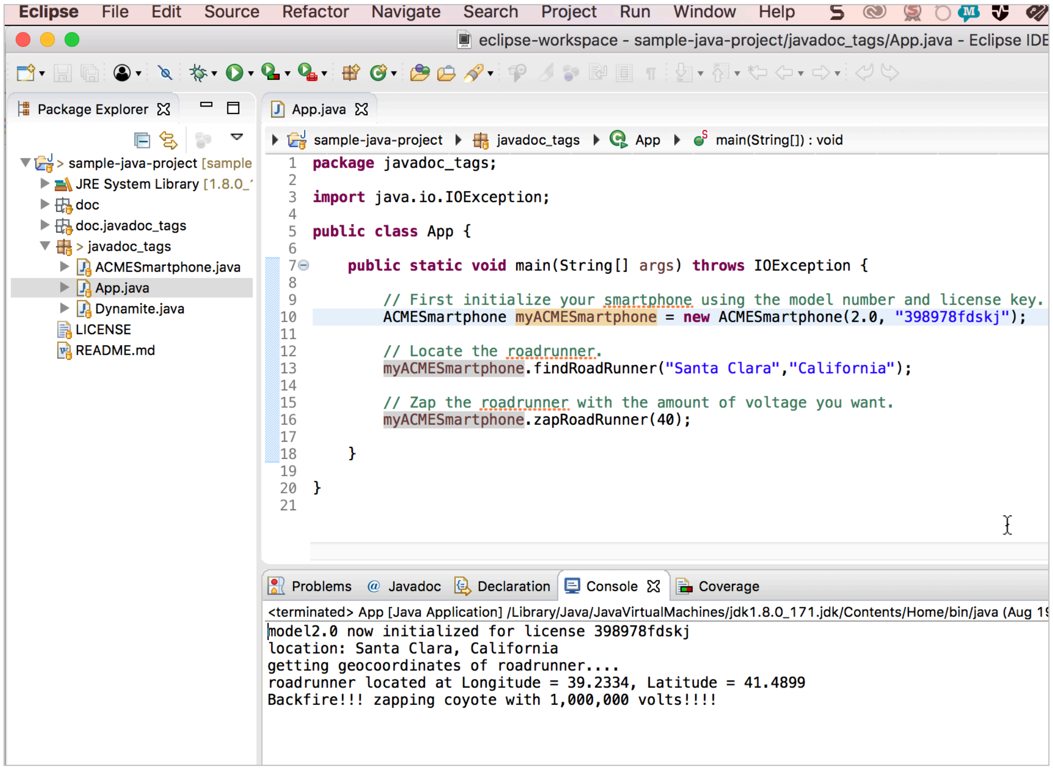
The main method (`public static void main(String[] args) throws IOException`), which appears in the **App.java** file, runs the functions defined in the other package's files (`ACMESmartphone.java` and `Dynamite.java`).

```java
public static void main(String[] args) throws IOException {

  // First initialize your smartphone using the model number and lice
nse key.
  ACMESmartphone myACMESmartphone = new ACMESmartphone(2.0, "398978fd
skj");

  // Locate the roadrunner.
  myACMESmartphone.findRoadRunner("Santa Clara","California");

  // Zap the roadrunner with the amount of voltage you want.
  myACMESmartphone.zapRoadRunner(40);

}
```

You can view the details of each function by clicking the `ACMESmartphone.java` and `Dynamite.java` class files.

Then app prints this text to the console:

```
model2.0 now initialized for license 398978fdskj
location: Santa Clara, California
getting geocoordinates of roadrunner....
roadrunner located at Longitude = 39.2334, Latitude = 41.4899
Backfire!!! zapping coyote with 1,000,000 volts!!!!
```


Eclipse File Edit Source Refactor Navigate Search Project Run Window Help
eclipse-workspace - sample-java-project/javadoc_tags/App.java - Eclipse IDE
Package Explorer
sample-java-project [sample
JRE System Library [1.8.0_
doc
doc.javadoc_tags
javadoc_tags
ACMESmartphone.java
App.java
Dynamite.java
LICENSE
README.md
App.java
sample-java-project › javadoc_tags › App › main(String[]) : void
Problems @ Javadoc Declaration Console Coverage

```java
package javadoc_tags;

import java.io.IOException;

public class App {

    public static void main(String[] args) throws IOException {

        // First initialize your smartphone using the model number and license key.
        ACMESmartphone myACMESmartphone = new ACMESmartphone(2.0, "398978fdskj");

        // Locate the roadrunner.
        myACMESmartphone.findRoadRunner("Santa Clara","California");

        // Zap the roadrunner with the amount of voltage you want.
        myACMESmartphone.zapRoadRunner(40);

    }

}
```

```
<terminated> App [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin/java (Aug 1
model2.0 now initialized for license 398978fdskj
location: Santa Clara, California
getting geocoordinates of roadrunner....
roadrunner located at Longitude = 39.2334, Latitude = 41.4899
Backfire!!! zapping coyote with 1,000,000 volts!!!!
```

Running the program

Side Note about Maven projects

Before leaving this intro topic on getting started with a Java project, I'd like to make a note about Maven. Java projects often have a lot of dependencies on packages that are third-party libraries or at least non-standard Java utilities. Rather than requiring users to download these additional packages and add them to their class manually, developers frequently use Maven to manage the packages. (Maven is to Java as Gradle is to Android.)

Maven projects use a pom.xml file that defines the dependencies. Eclipse ships with Maven already installed, so when you import a Maven project and install it, the Eclipse Maven plugin will retrieve all of the project dependencies and add them to your project.

The sample project doesn't use Maven, but chances are if you're getting a Java project from developers, you won't import it in the way I previously described. Instead, you'll import it as an existing Maven project.

(To import a Maven project into Eclipse, you would go to **File > Import > Maven > Existing Maven Projects** and click **Next**. In the Root Directory field, you would click **Browse** and browse to the Java project folder (which contains the Maven pom.xml file) and then click **Open**. Then you would click **Finish** in the dialog box. In the Project Explorer pane in Eclipse, you would right-click the Java folder and select **Run**

as Maven Install. Maven retrieves the necessary packages and builds the project. If the build is successful, you will see a “BUILD SUCCESS” message in the console. You would then use the source code in the built project.)

Java crash course

To understand the different components of a Javadoc, first you have to understand a bit about Java. Becoming familiar with the names of the different components of Java will allow you to enter conversations and understand the code at a high level. When you describe various aspects of sample code, knowing when to call something a class, method, parameter, or enum can be critical to your documentation's credibility.

I'll run you through a brief crash course in the basics. Don't worry if this seems like a blur. For more detail about learning Java, I recommend consulting lynda.com and safaribooksonline. Below I'll focus on some basic concepts in Java that will be important in understanding the Javadoc tags and elements (p. 712).

About Java

Java's flexibility and widespread use make it one of the best programming languages to learn for technical writers. Java isn't tied to a specific language platform. Instead, Java code compiles into byte code. The platform you deploy your code on contains a Java Virtual Machine (JVM) that interprets the byte code. Hence through JVMs, different platforms can interpret and run Java code — this gives Java more flexibility with different platforms.

Classes

Classes are templates or blueprints that drive pretty much everything in Java. It's easiest to understand classes through an example. Think of a class like a general blueprint of a "bicycle." There are many different types of bicycles (Trek bikes, Specialized bikes, Giant bikes, Raleigh bikes, etc.). But they're all just different instances of the general class of a bicycle.

In Java, you start by defining classes. Each class is its own file and begins with a capital letter. The file name matches the class name, which means you have just one class per file.

Each class can contain some fields (variables for the class) and methods (subroutines the class can do).

Before the class name, an access modifier indicates how the class can be accessed. Several options for access modifiers are as follows:

- `public` : Anyone can access
- `private` : Only other packages can access
- `static` : No one can change the class
- `abstract` : The class can't be instantiated, only sub-classed.

Here's an example of a class:

```
public class Bicycle{

//code...

}
```

You mostly need to focus on `public` classes, since these are the classes that will be used by external developers. The `public` classes are the API of the library.

Methods

Methods are subroutines or actions that the class can do. For example, with a bicycle you can pedal, brake, and turn. A class can have as many methods as it needs.

Methods can take arguments, so there are parentheses `()` after the method name. The arguments are variables that are used within the code for that method. For example:

```
add(a, b) {
  sum = a + b;
}
```

Methods can return values. When a method finishes, the value can be returned to the caller of the method.

Before the method name, the method indicates what type of data it returns. If the method doesn't return anything, `void` is listed. Other options are `String` or `int`.

Here's an example of some methods for our Bicycle class:

```
class Bicycle {

        void turn() {
        // code ...
        }
        void pedal(int rotations) {
           System.out.println("Your speed is " + rotations + " per minut
e".);
        }

        int brake(int force, int weight) {
          torque == force * weight;
          return torque;
        }
}
```

See how the `brake` method accepts two arguments — `force` and `weight`. These arguments are integers, so Java expects whole numbers here. (You must specify the data type before the parameters in the method.) The arguments passed into this method are used to calculate the `torque`. The `torque` is then returned to the caller.

Java has different types of methods. *Instance* methods can only be used from objects instantiated from the class. *Static* methods can be used directly from the class without instantiating an object first. *Abstract* methods can only be used through subclasses.

Somewhere in your Java application, users will have something called a `main` method that looks like this:

```
public static void main(String[] args) {
}
```

Inside the `main` method is where you add your code to make your program run. As a result, the code inside the `main` method references other objects (instantiated from classes). The Java Virtual Machine will look inside the `main` method to run the application code.

Fields

Fields are variables available within the class. A variable is a placeholder that is populated with a different value depending on what the user wants.

Fields indicate their data types because all data in Java is “statically typed” (meaning, its format/length is defined) so that the data doesn’t take up more space than it needs. Some data types include `byte`, `short`, `long`, `int`, `float`, or `double`. These are numbers or decimals of different sizes. You can also specify a `char`, `string`, or `boolean`.

Here’s an example of some fields in class:

```
class Bicycle {
  String brand;
  int size;
}
```

Many times fields are “encapsulated” with getter-setter methods, which means their values are set in a protected way. Users call one method to set the field’s value, and another method to get the field’s value. This way you can avoid having users set improper values or incorrect data types for the fields.

Fields that are constant throughout the Java project are called ENUMS. Alternatively, the fields are given `public static final` modifiers.

Objects

Objects are instances of classes. They are the Treks, Raleighs, Specialized, etc., of the Bicycle class.

If I wanted to use the `Bicycle` class, I would create an instance of the class. The instance of the class is called an object. Here’s what it looks like when you “instantiate” the class:

```
Bicycle myBicycle = new Bicycle();
```

You write the class name followed by the object name for the class. Then assign the object to be a new instance of the class. Now you’ve got `myBicycle` to work with.

The object inherits all of the fields and methods available to the class.

You can access fields and methods for the object using dot notation, like this:

```
myBicycle Bicycle = new Bicycle();

myBicycle.brand = "Trek";
myBicycle.pedal();
```

You probably won’t see many objects in the native library. Instead, the developers who implement the API will create objects. However, if you have a reference implementation or sample code on how to implement the API, you will see a lot of objects.

Constructors

Constructors are methods used to create new instances (objects) of the class. The default constructor for the class looks like the one above, with `new Bicycle()`.

The constructor uses the same name as the class and is followed by parentheses (because constructors are methods).

Often classes have constructors that initialize the object with specific values passed in to the constructor.

For example, suppose we had a constructor that initialized the object with the brand and size:

```
public class Bicycle{

  public Bicycle(String brand, int size) {
          this.brand = model;
          this.size = size;
  }

}
```

Now you can use this constructor when creating a new `Bicycle` object:

```
Bicycle myBicycle = new Bicycle ("Trek", 22);
```

It's a best practice to include a constructor even if it's just the default.

Packages

Classes are organized into different packages in Java projects. Packages are like folders or directories where the classes are stored. Putting classes into packages helps avoid naming conflicts.

When you create your class, if it's in a package called `vehicles`, you list this package at the top of the class:

```
package vehicles

public class Bicycle {
//
}
```

Classes also set boundaries on access based on the package. If the access modifier does not say `public`, the class would only be accessible to members of the same package. If the access modifier is `protected`, the class is only accessible to the class, package, and subclasses.

When you want to instantiate the class (and your file is outside the package), you need to import the package into your class, like this:

```
import vehicles

        public static void main(String[] args) {

        }
```

When packages are contained inside other packages, you access the inner packages with a dot, like this:

```
import transportation.motorless.vehicles
```

Here you would have a transportation package containing a package called motorless containing a package called vehicles. Package naming conventions are like URLs in reverse (com > yoursite > subdomain).

Maven handles package management for Java projects. Maven will automatically go out and get all the package dependencies for a project when you install a Maven project.

Exceptions

To avoid broken code, developers anticipate potential problems through exception handling. Exceptions say, if there’s an issue here, flag the error with this exception and then continue through the code.

Different types of errors throw different exceptions. By identifying the type of exception thrown, you can more easily troubleshoot problems when code breaks because you know the specific error that’s happening.

You can identify a specific exception the class throws in the class name after the keyword `throws`:

```
public class Bicycle throws IOException {

}
```

When you indicate the exception here, you list the type of exception using a specific Javadoc tag.

Inheritance

Some classes can extend other classes. Extending a class means a class inherits the properties of another class. When one class extends another class, you’ll see a note like this:

```
public class Bicycle extends Vehicle {

}
```

This code means that `Bicycle` inherits all of the properties of `Vehicle` and then can add to them.

Interfaces

An interface is a class that has methods with no code inside the method. Interfaces are intended to be implemented by another class that will insert their own values for the methods. Interfaces are a way of formalizing a class that will have a lot of subclasses. Interfaces force the subclasses to standardize on common strings and methods.

JAR files and WAR files

The file extension for a class is `.java`, but when compiled by the Java Development Kit into the Java program, the file becomes `.class`. The `.class` file is binary code, which means only computers (in this case, the Java Virtual Machine, or JVM) can read it.

Developers often package up java files into a JAR file, which is like a zip file for Java projects. When you distribute your Java files, you'll likely provide a JAR file that the developer audience will add to their Java projects.

Developers will add their JAR to their class path to make the classes available to their project. To do this, they right-click their project and select **Properties**. In the dialog box, they select **Java Build Path** and then click the **Libraries** tab. They then click **Add JARs** and browse to the JAR.

When you deliver a JAR file, developers can use the classes and methods available in the JAR. However, the JAR will not show them the source code, that is, the raw Java files. For this, users will consult the Javadoc.

If you're distributing a reference implementation that consists of a collection of Java source files (so that developers can see how to integrate your product in Java), you'll probably just send them a zip file containing the project.

A WAR file is a web application archive. A WAR is a compiled application that developers deploy on a server to run an application. Whereas the JAR is integrated into a Java project while the developers are actively building the application, the WAR is the deployed program that you run from your server.

That's probably enough Java to understand the different components of a Javadoc.

Summary

Here's a quick summary of the concepts we talked about:

- **Class**: Blueprints for something
- **Object**: An instance of a class
- **Methods**: What the object/class can do
- **Fields**: Variables in the object/class
- **Constructor**: A method to create an object for a class
- **Package**: A folder that groups classes
- **Access modifier** (e.g., `public`): The scope at which a thing can be accessed
- **Interface**: A skeleton class with empty methods (used for standardizing)
- **Enum**: A data type offering predefined constants
- **Subclass**: A class that inherits the fields + methods of another class
- **JAR file**: A zip-like file containing Java classes
- **WAR file**: A compiled Java web application to be deployed on a server

The point is that now you can use these terms intelligently in documentation and have at least a sense of what's going on.

Comparing native library APIs to REST

Now that you see the specific programming language and elements in Java, you can probably get a sense of how different native library APIs are from REST APIs. To document a native library API, you need to become familiar with the programming language. In contrast, REST APIs are language agnostic, so they tend to be more accessible to generalists.

Activity: Generate a Javadoc from a sample project

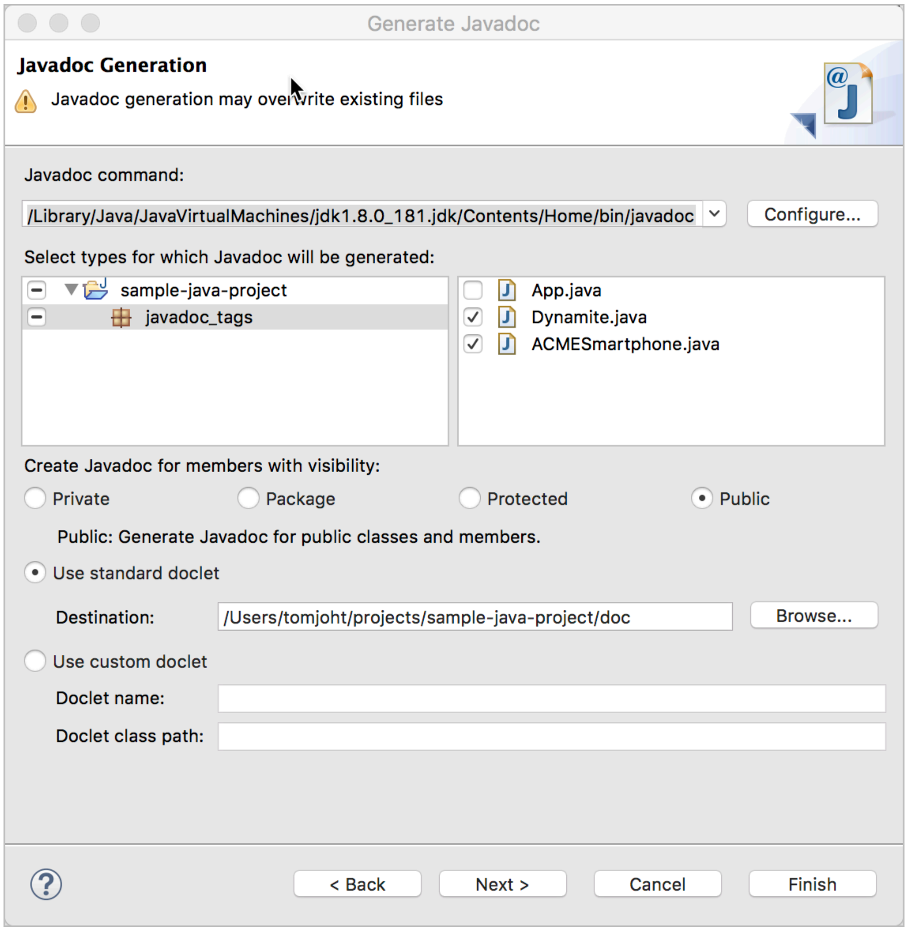
Javadoc is the standard output for Java APIs, and it's somewhat effortless to build a Javadoc. The Javadoc is generated through something called a "doclet." Different doclets can parse the Java annotations in different ways and produce different outputs. But by and large, almost every Java documentation uses the standard doclet. The Javadoc output will be familiar and welcome to Java developers.

Javadoc is supported by Oracle. Developers can integrate the Javadoc output directly into their IDE, which makes the documentation convenient and easily accessible. In fact, Javadoc is often delivered this way rather than deployed and uploaded to a server. Javadoc contains reference documentation only. You can't add any conceptual help files into the output, nor change the layout.

Generate a Javadoc

In this activity, you'll generate a Javadoc from the sample java project (p. 695) you downloaded earlier. If you want to preview the Javadoc that you're going to build, you can view the output here.

1. In Eclipse, go to **File > Export**.
2. Expand **Java** and select **Javadoc.** Then click **Next**.
3. Select your project and package. Then in the right pane, select the classes you want to be included in the Javadoc — in this case, **Dynamite.java** and **ACMESmartphone.java.**



Generate Javadoc

Javadoc Generation

Javadoc generation may overwrite existing files

Javadoc command: /Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/javadoc

Select types for which Javadoc will be generated:

- sample-java-project
  - javadoc_tags

☐ App.java
☑ Dynamite.java
☑ ACMESmartphone.java

Create Javadoc for members with visibility:

○ Private ○ Package ○ Protected ● Public

Public: Generate Javadoc for public classes and members.

● Use standard doclet

Destination: /Users/tomjoht/projects/sample-java-project/doc

○ Use custom doclet

Doclet name:

Doclet class path:

Generating a Javadoc

Don’t select the `App.java` class that contains your `main` method because you don’t need to provide Java documentation for your program; your Javadoc will provide documentation for the classes that developers might choose to use in their programs. In this sample project, the `main` method is included in `App.java`, but it could have been included in a file named otherwise.

4. Select which visibility option you want: Private, Package, Protected, or Public. Generally, you select **Public**.

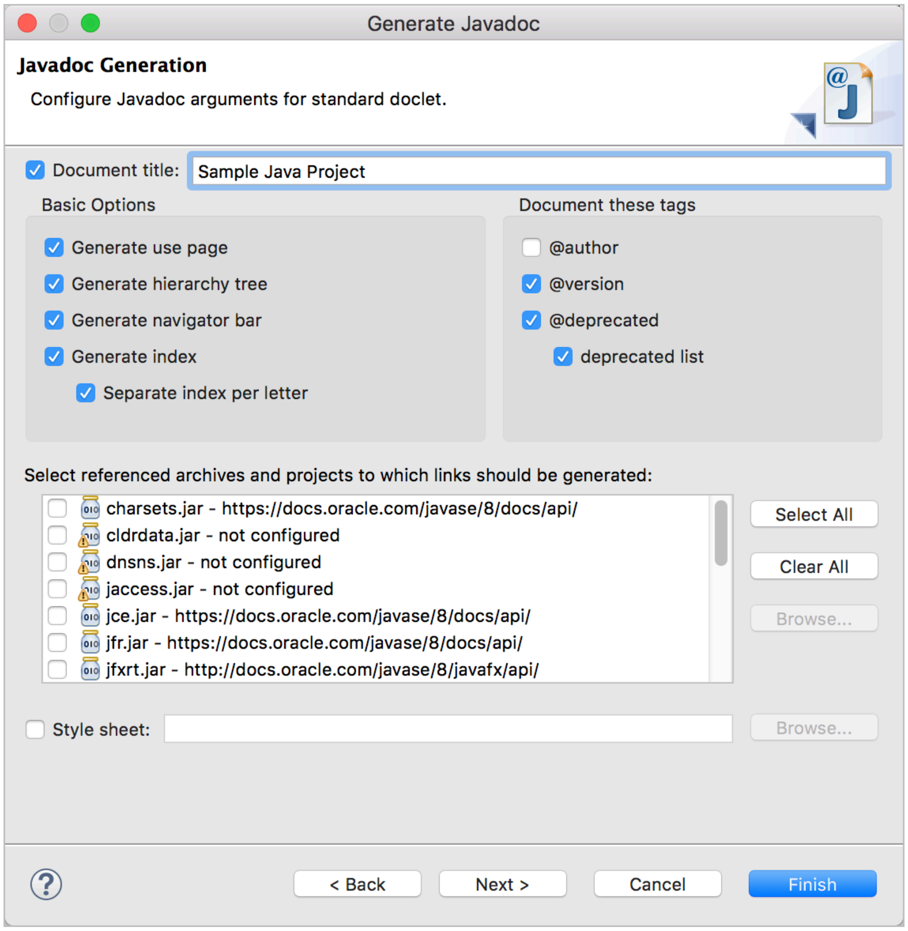
   Your API probably has a lot of helper or utility classes used on the backend, but only a select number of classes will be used by your developer audience. These classes are made public. It’s the public classes that your developer audience will use that form the API aspect of the class library.

5. Make sure the **Use standard doclet** radio button is selected. (It’s selected by default.)
6. Click the **Browse** button and select the output location where you want the Javadoc generated. By default, it will be generated in the same project folder as your code, but in a subfolder called `doc`. This way you can browse the Javadoc directly within your Eclipse IDE.

   For this activity, choose a different output location (such as a folder on your desktop or in your documents) other than the default. Reason being, the project already has the generated Javadoc in a docs folder, so you might not even realize that you’ve generated a Javadoc file because your new output will just overwrite the existing doc files.

When developers deliver Java libraries, they often include the doc directly in a distributed zip file that contains that JAR. Users are accustomed to looking in the doc folder for documentation.

7. Click **Next** and see additional configuration options.



Generate Javadoc

Javadoc Generation

Configure Javadoc arguments for standard doclet.

☑ Document title: Sample Java Project

Basic Options

- ☑ Generate use page
- ☑ Generate hierarchy tree
- ☑ Generate navigator bar
- ☑ Generate index
  - ☑ Separate index per letter

Document these tags

- ☐ @author
- ☑ @version
- ☑ @deprecated
  - ☑ deprecated list

Select referenced archives and projects to which links should be generated:

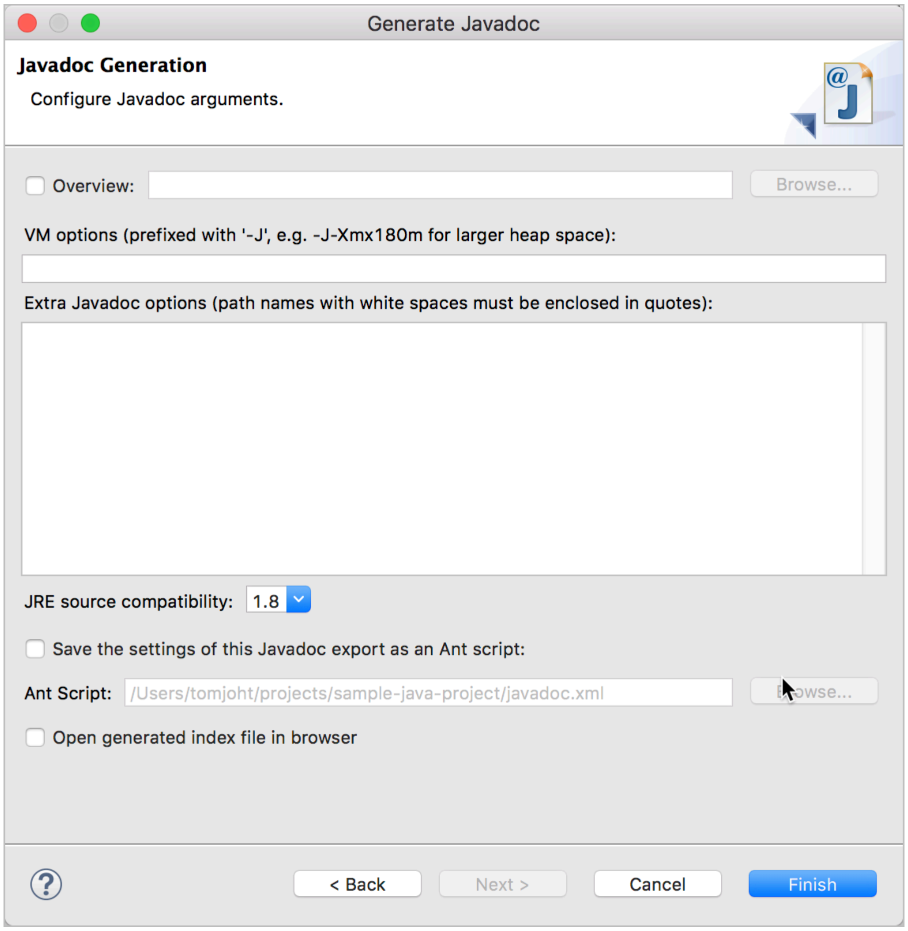
- ☐ charsets.jar - https://docs.oracle.com/javase/8/docs/api/
- ☐ cldrdata.jar - not configured
- ☐ dnsns.jar - not configured
- ☐ jaccess.jar - not configured
- ☐ jce.jar - https://docs.oracle.com/javase/8/docs/api/
- ☐ jfr.jar - https://docs.oracle.com/javase/8/docs/api/
- ☐ jfxrt.jar - http://docs.oracle.com/javase/8/javafx/api/

☐ Style sheet:

Javadoc next screen

Here you can select if you want to omit some tags, such as @author and @deprecated. Generally, you don’t include the @author tag, since it may only be important internally, not externally. You can also select different options in the Javadoc frame. If you have a custom stylesheet, you can select it here. Most likely you would only make superficial style changes such as with colors.

8. Click **Next**.



Generate Javadoc

Javadoc Generation

Configure Javadoc arguments.

☐ Overview:

VM options (prefixed with '-J', e.g. -J-Xmx180m for larger heap space):

Extra Javadoc options (path names with white spaces must be enclosed in quotes):

JRE source compatibility: 1.8

☐ Save the settings of this Javadoc export as an Ant script:

Ant Script: /Users/tomjoht/projects/sample-java-project/javadoc.xml

☐ Open generated index file in browser

Overview page

Here you can select an HTML page that you want to be your overview page in the Javadoc. You can select any HTML page, and it will be included in the index.

9. Click **Finish**.

   If the Finish button isn't available, it might be because Eclipse can't find the right Javadoc executable. You can click **Configure** and browse to the file manually. On a MacBook Pro, the Javadoc file is at `/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin/javadoc`.

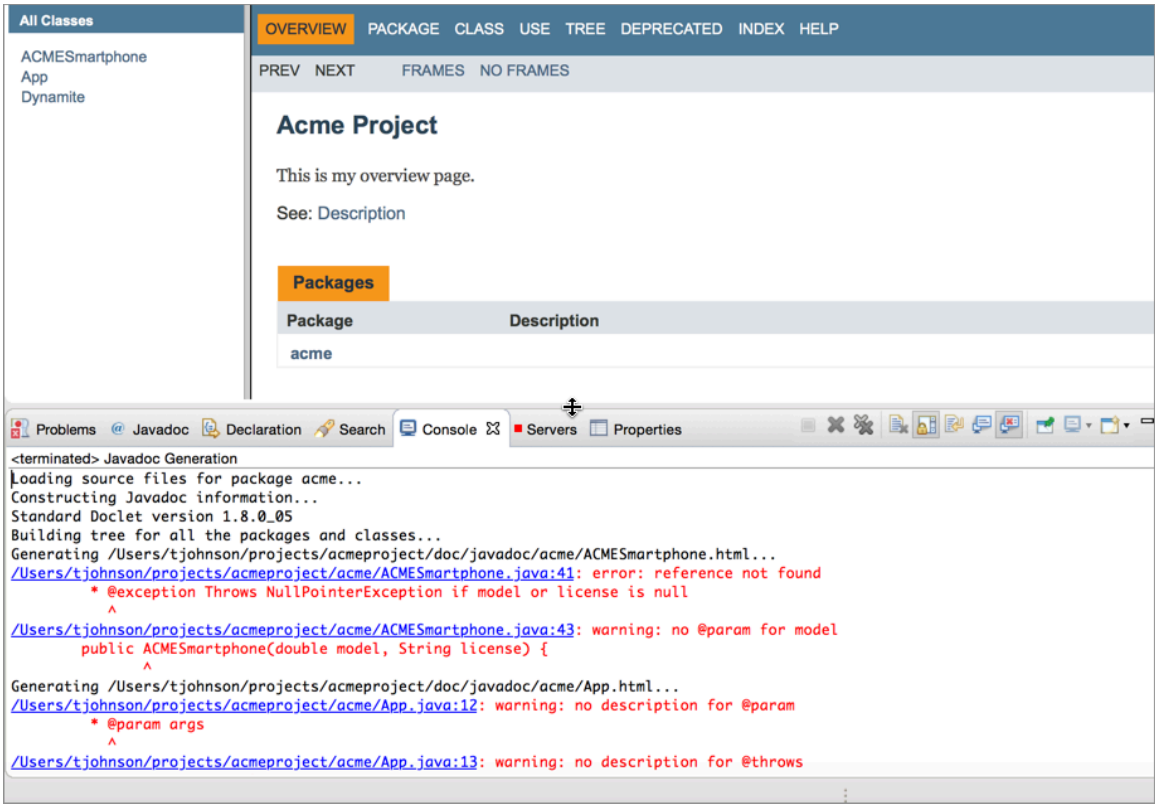
   If prompted to update the Javadoc location (which likely differs from your Eclipse workspace location), do so by clicking **Yes to all**.

   Browse to the destination location and open the `index.html` file in your browser to view the files.

   If you had trouble building your Javadoc, you could view the output here.

Javadoc and error checking

Javadoc also checks your tags against the actual code. If you have parameters, exceptions, or returns that don't match up with the parameters, exceptions, or returns in your actual code, then Javadoc will show some warnings.



Acme Project

This is my overview page.

See: Description

Packages

Package	Description
acme	

<terminated> Javadoc Generation

```
Loading source files for package acme...
Constructing Javadoc information...
Standard Doclet version 1.8.0_05
Building tree for all the packages and classes...
Generating /Users/tjohnson/projects/acmeproject/doc/javadoc/acme/ACMESmartphone.html...
/Users/tjohnson/projects/acmeproject/acme/ACMESmartphone.java:41: error: reference not found
         * @exception Throws NullPointerException if model or license is null
           ^
/Users/tjohnson/projects/acmeproject/acme/ACMESmartphone.java:43: warning: no @param for model
        public ACMESmartphone(double model, String license) {
               ^
Generating /Users/tjohnson/projects/acmeproject/doc/javadoc/acme/App.html...
/Users/tjohnson/projects/acmeproject/acme/App.java:12: warning: no description for @param
         * @param args
           ^
/Users/tjohnson/projects/acmeproject/acme/App.java:13: warning: no description for @throws
```

Javadoc error checking

Play with Javadoc parameters

Try removing a parameter from a method and generate the Javadoc again. Make sure the console window is open so you can see the error warnings.

Auto-building the Javadoc

In most development shops, you'll never manually create a Javadoc file as I've explained here. Javadoc allows you to build it from the command line by feeding it a configuration file. Most developers have a build management tool where they can configure outputs from their code, including Javadoc. If developers ask you to build the Javadoc, chances are their build management system might be somewhat primitive. However, it can be helpful to build the Javadoc output yourself (as I've explained here) to test and review the output before generating it through the build management tool.

Javadoc tags

Javadoc (p. 707) is a document generator that looks through your Java source files for specific annotations. It parses out the annotations into the Javadoc output. Knowing the annotations is essential because this is how the Javadoc gets created.

Comments versus Javadoc tags

A general comment in Java code is signaled like this:

```
// sample comment...

/*
sample comment
*/
```

Javadoc does nothing with these comments.

To include content in Javadoc, you add *two* asterisks at the start, before the class or method:

```
/**
*
*
*
*
*/
```

(In Eclipse, if you type `/**` and hit return, it autofills the rest of the syntax automatically.)

The format for adding the various elements is like this:

```
/**
* [short description]
* <p>
* [long description]
*
* [author, version, params, returns, throws, see, other tags]
* [see also]
*/
```

Here's a real example of Javadoc comments for a method.

```
/**
* Zaps the roadrunner with the number of volts you specify.
* <p>
* Do not exceed more than 30 volts or the zap function will backfire.
* For another way to kill a roadrunner, see the {@link Dynamite#blowDynamite()} method.
*
* @exception IOException if you don't enter a data type amount for the voltage
* @param voltage the number of volts you want to send into the roadrunner's body
* @see #findRoadRunner
* @see Dynamite#blowDynamite
*/
public void zapRoadRunner(int voltage) throws IOException {
   if (voltage < 31) {
       System.out.println("Zapping roadrunner with " + voltage + " volts!!!!");
   }
   else {
    System.out.println("Backfire!!! zapping coyote with 1,000,000 volts!!!!");
   }
}
```

You put the Javadoc description and tags *before* the class or method (no need for any space between the description and class or method).

Common Javadoc tags

The following are the most common tags used in Javadoc. Each tag has a word that follows it. For example, `@param latitude` means the parameter is “latitude.”

To view a lengthy example of Javadoc tags, see this example from Oracle.</div>

The following are some common Javadoc tags:

- **`@author`** A person who made significant contributions to the code. Applied only at the class, package, or overview level. Not included in the Javadoc output. It’s not recommended to include this tag since authorship changes often.
- **`@param`** A parameter that the method or constructor accepts.
- **`@deprecated`** Lets users know the class or method is no longer used. This tag will be positioned prominently in the Javadoc. Accompany it with a `@see` or `{@link}` tag as well.
- **`@return`** What the method returns.
- **`@see`** Creates a “see also” list. Use `{@link}` for the content to be linked.
- **`{@link}`** Used to create links to other classes or methods. Example: `{@link Foo#bar}` links to the method `bar` that belongs to the class `Foo`. To link to the method in the same class, just include `#bar`.
- **`@since 2.0`** The version since the feature was added.
- **`@throws`** The kind of exception the method throws. Note that your code must indicate an exception thrown in order for this tag to validate. Otherwise, Javadoc will produce an error. `@exception` is an alternative tag.

- `@Override` Used with interfaces and abstract classes. Performs a check to see if the method is an override.

What elements you add Javadoc tags to

You add Javadoc tags to classes, methods, and fields.

- For the `@author` and `@version`, add these tags only to classes and interfaces.
- The `@param` tags get added only to methods and constructors.
- The `@return` tag gets added only to methods.
- The `@throws` tag can be added to classes or methods.

Public versus private modifiers and Javadoc

Javadoc includes only classes, methods, etc., marked as `public`. Elements marked as `private` are not included in Javadoc unless you specifically select **private** when generating the Javadoc. If you omit `public` from your source code, the default is that the class or method is available to the package only. In this case, it is not included in Javadoc.

The description

Javadoc provides both a short and long description. Here's an example showing how the description part is formatted:

```
/**
* Short one line description.
* <p>
* Longer description. If there were any, it would be
* here.
* <p>
* And even more explanations to follow in consecutive
* paragraphs separated by HTML paragraph breaks.
*
* @param variable Description text text text.
* @return Description text text text.
*/
public int methodName (...) {
// method body with a return statement
}
```

(This example comes from Wikipedia entry.)

The short description is the first sentence and gets shortened as a summary for the class or method in the Javadoc. After a period, the parser moves the rest of the description into a long description. Use `<p>` to signal the start of a new paragraph. You don't need to surround the paragraphs with opening and closing `<p>` tags because the Javadoc compiler automatically adds them. Also, you can use HTML in your descriptions, such as an unordered list, code tags, bold tags, or others.

After the descriptions, enter a blank line (for readability) and then start the tags. You can't add any more description content below the tags. Note that only methods and classes can have tags, not fields. Fields (variables) have descriptions only.

Note that the first sentence is much like the `shortdesc` element in DITA. This first sentence is supposed to be a summary of the entire class or method. If one of your words has a period in it (like `Dr. Jones`), then you must remove the space following the period by adding `Dr.&nbsp;Jones` to connect it.

Avoid using links in the first sentence. After the period, the next sentence shifts to the long paragraph, so you have to load up that first sentence to be descriptive. The verb tense should be present tense, such as *gets, puts, displays, calculates…*

What if the method is so obvious (for example, printPage) that your description ("prints a page") becomes redundant and looks useless? Oracle says in these cases, you can omit saying "prints a page" and instead try to offer some other insight. Oracle suggests:

> Add description beyond the API name. The best API names are "self-documenting", meaning they tell you basically what the API does. If the doc comment merely repeats the API name in sentence form, it is not providing more information. For example, if method description uses only the words that appear in the method name, then it is adding nothing at all to what you could infer. The ideal comment goes beyond those words and should always reward you with some bit of information that was not immediately obvious from the API name. – How to write javadoc comments

Avoid `@author`

Commenting on Javadoc best practices, some people recommend using `@author` because the author value easily slips out of date, and the source control provides better indication of the last author. (See Javadoc coding standards for more detail.)

Order of tags

Oracle says the order of the tags should be as follows:

```
@author (classes and interfaces)
@version (classes and interfaces)
@param (methods and constructors)
@return (methods)
@throws (@exception is an older synonym)
@see
@since
@serial
@deprecated
```

`@param` tags

@param tags apply only to methods and constructors, both of which take parameters. After the @param tag, add the parameter name, and then a description of the parameter, in lowercase, with no period, like this:

```
@param url the web address of the site
```

The parameter description is a phrase, not a full sentence. The order of multiple @param tags should mirror their order in the method or constructor.

Stephen Colebourne recommends adding an extra space after the parameter name to increase readability (and I agree).

As far as including the data type in the parameter description, Oracle says:

> By convention, the first noun in the description is the data type of the parameter. (Articles like “a”, “an”, and “the” can precede the noun.) An exception is made for the primitive int, where the data type is usually omitted. (See How to write doc comments using Javadoc.)

The example Oracle gives is as follows:

```
@param ch the character to be tested
```

However, the data type is visible from the parameters in the method. So even if you don’t include the data types, it will be easy for users to see what they are.

Note that you can have multiple spaces after the parameter name so that your parameter definitions all line up.

`@param` tags must be provided for every parameter in a method or constructor. Failure to do so will create an error and warning when you render Javadoc.

Note that usually, classes don’t have parameters. There is one exception: Generics. Generic classes are classes that work with different type of objects. The object is specified as a parameter in the class in diamond brackets: `<>`. Although the Javadoc guidance from Oracle doesn’t mention them, you can add a `@param` tag for a generic class to note the parameters for the generic class. See this StackOverflow post for details. Here’s an example from that page:

```
/**
* @param <T> This describes my type parameter
    */
    class MyClass<T>{

        }
```

`@return` tags

Only methods return values, so only methods would receive a `@return` tag. If a method has `void` as a modifier, it doesn’t return anything. If it doesn’t say `void`, then you must include a `@return` tag to avoid an error when you compile Javadoc.

`@throws` tags

You add `@throws` tags to methods or classes only if the method or class throws a particular kind of error. Here’s an example:

```
@throws IOException if your input format is invalid
```

Stephen Colebourne recommends starting the description of the throws tag with an “if” clause for readability. He says,

> The `@throws` feature should normally be followed by `if ...` and the rest of the phrase describing the condition. For example, `@throws if the file could not be found`. This aids readability in source code and when generated.

If you have multiple throws tag, arrange them alphabetically.

Doc comments for constructors

It’s a best practice to include a constructor in a class. However, if the constructor is omitted, Javadoc automatically creates a constructor in the Javadoc but omits any description of the constructor.

Constructors have `@param` tags but not `@return` tags. Everything else is similar to methods.

Doc comments for fields

Fields have descriptions only. You would only add doc comments to a field if the field were something a user would use.

Cases where you don’t need to add doc comments

Oracle says there are three scenarios where the doc comments get inherited, so you don’t need to include comments in these scenarios:

> - When a method in a class overrides a method in a superclass
> - When a method in an interface overrides a method in a superinterface
> - When a method in a class implements a method in an interface
>   (See How to write Javadoc comments ↗.)

`@see` tags

The @see tag provides a see also reference. There are various ways to denote what you’re linking to in order to create the link. If you’re linking to a field, constructor, or method within the same field, use `#`.

If you’re linking to another class, put that class name first followed by the `#` and the constructor, method, or field name.

If you’re linking to a class in another package, put the package name first, then the class, and so on. See this sample from Oracle:

```
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id)
@see Class#method(Type, Type,...)
@see Class#method(Type id, Type id,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id)
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type id, Type, id)
```

For more information, see How to write Javadoc comments.

Links

You can create links to other classes and methods using the `{@link}` tag.

Here's an example from Javadoc coding standards on making links:

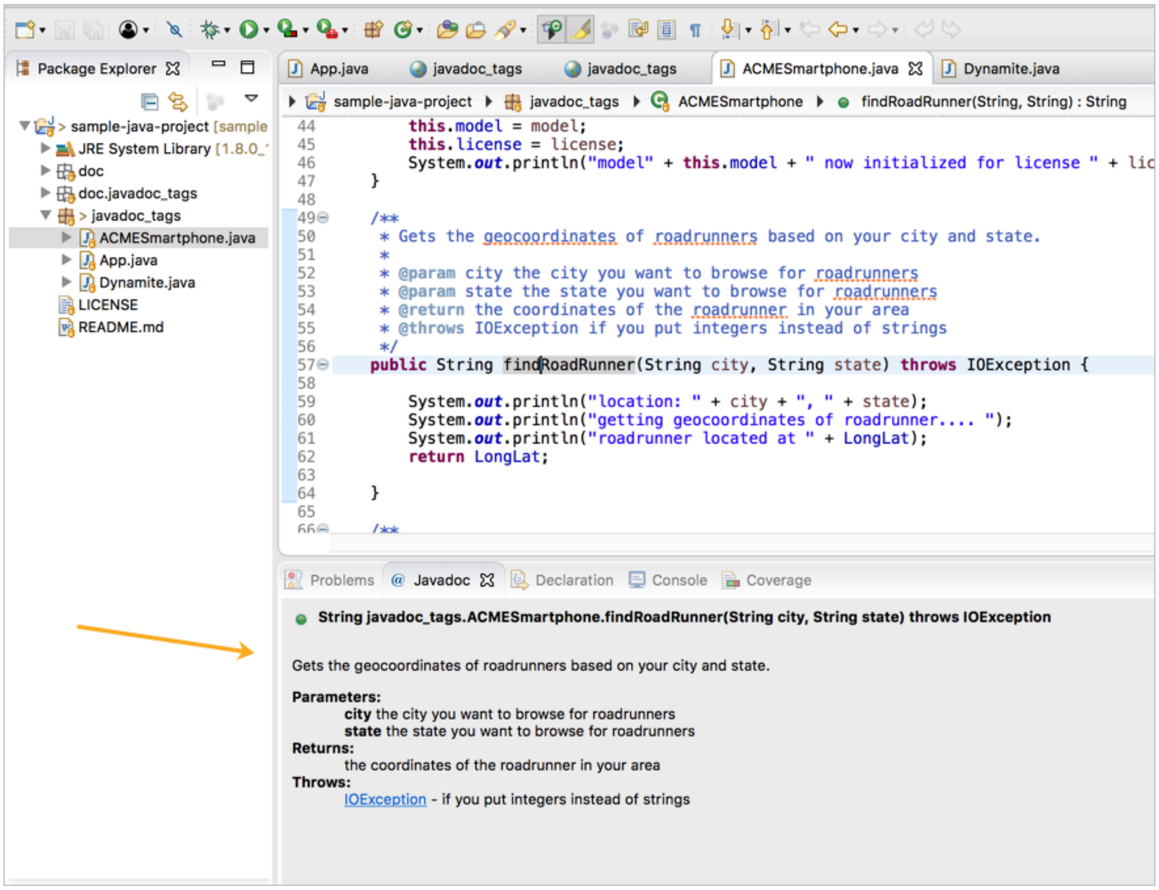
```
/**
* First paragraph.
* <p>
* Link to a class named 'Foo': {@link Foo}.
* Link to a method 'bar' on a class named 'Foo': {@link Foo#bar}.
* Link to a method 'baz' on this class: {@link #baz}.
* Link specifying text of the hyperlink after a space: {@link Foo the Foo cl
ass}.
* Link to a method handling method overload {@link Foo#bar(String,int)}.
*/
public ...
```

To link to another method within the same class, use this format: `{@link #baz}`. To link to a method in another class, use this format: `{@link Foo#baz}`. However, don't over hyperlink. When referring to other classes, you can use `<code>` tags.

To change the linked text, put a word after `#baz` like this: `@see #baz Baz method`.

Previewing Javadoc comments

In Eclipse, you can use the Javadoc tab at the bottom of the screen to preview the Javadoc information included for the class you're viewing.



```
        this.model = model;
        this.license = license;
        System.out.println("model" + this.model + " now initialized for license " + lic
    }

    /**
     * Gets the geocoordinates of roadrunners based on your city and state.
     *
     * @param city the city you want to browse for roadrunners
     * @param state the state you want to browse for roadrunners
     * @return the coordinates of the roadrunner in your area
     * @throws IOException if you put integers instead of strings
     */
    public String findRoadRunner(String city, String state) throws IOException {

        System.out.println("location: " + city + ", " + state);
        System.out.println("getting geocoordinates of roadrunner.... ");
        System.out.println("roadrunner located at " + LongLat);
        return LongLat;

    }

    /**
```

String javadoc_tags.ACMESmartphone.findRoadRunner(String city, String state) throws IOException

Gets the geocoordinates of roadrunners based on your city and state.

Parameters:
  city the city you want to browse for roadrunners
  state the state you want to browse for roadrunners
Returns:
  the coordinates of the roadrunner in your area
Throws:
  IOException - if you put integers instead of strings

Javadoc pane

Why all the detail about Javadoc tags?

I've added a lot of specific detail and style guidelines about Javadoc tags here. Why? I wanted you to realize that annotations for the Javadoc follow a lot of style conventions and best practices. These conventions and best practices might not always be apparent or followed in Java files that you might be working on. You can add a lot of value just by making sure the content aligns with these style conventions.

More information about Javadoc

- Oracle's explanation of Javadoc tags
- Javadoc

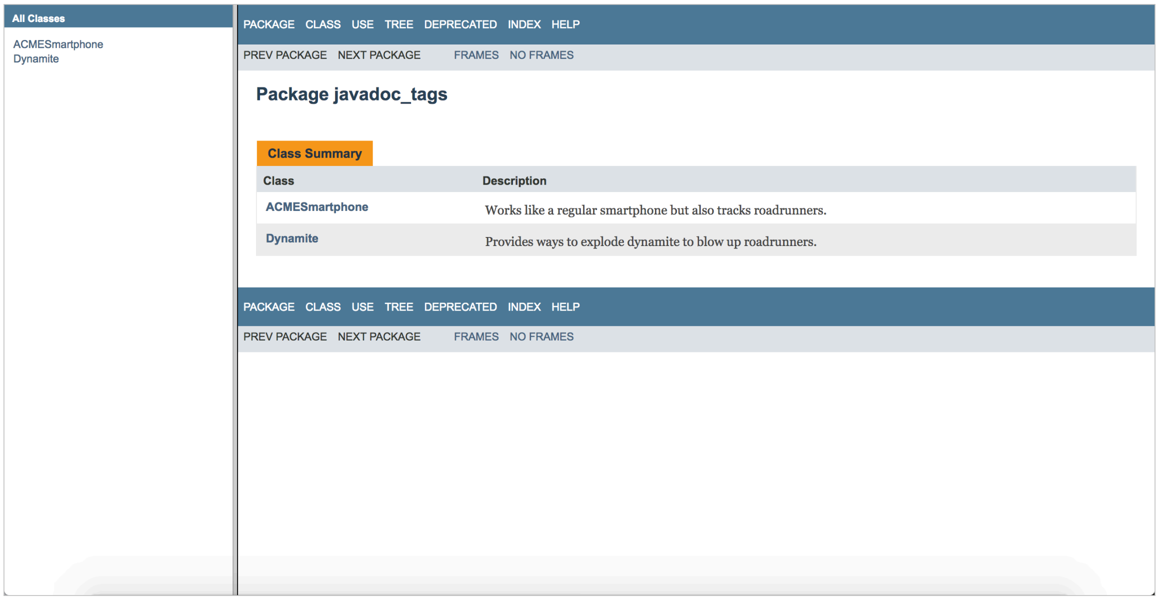
Explore the Javadoc output

The Javadoc output hasn't changed much in the past 20 years, so in some sense, it's predictable and familiar. On the other hand, the output is dated and lacks some critical features, like search, or the ability to add more pages. In this topic, we'll explore how the Javadoc is organized.

Class summary

Open the index.html file in the Javadoc folder you generated in Activity: Generate a Javadoc from a sample project (p. 707).

The class summary page shows a short version of each of the classes. The description you write for each class (up to the period) appears here. It's kind of like a quick reference guide for the API.


All Classes
ACMESmartphone
Dynamite
PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES
PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package javadoc_tags

Class Summary

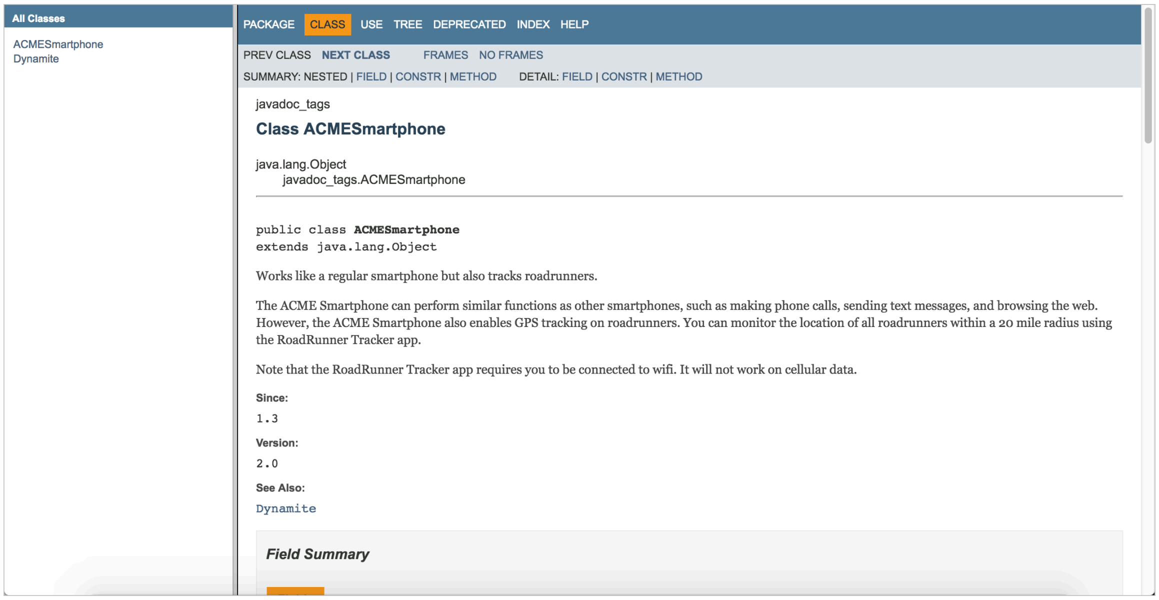
Class	Description
ACMESmartphone	Works like a regular smartphone but also tracks roadrunners.
Dynamite	Provides ways to explode dynamite to blow up roadrunners.

Class summary

You click a class name (in this case, `ACMESmartphone` or `Dynamite`) to dive into the details.

Class details

When you view a class page, you're presented with a summary of the fields, constructors, and methods for the class. Again, this is just an overview. When you scroll down, you can see the full details about each of these elements.



javadoc_tags

Class ACMESmartphone

java.lang.Object
    javadoc_tags.ACMESmartphone

```
public class ACMESmartphone
extends java.lang.Object
```

Works like a regular smartphone but also tracks roadrunners.

The ACME Smartphone can perform similar functions as other smartphones, such as making phone calls, sending text messages, and browsing the web. However, the ACME Smartphone also enables GPS tracking on roadrunners. You can monitor the location of all roadrunners within a 20 mile radius using the RoadRunner Tracker app.

Note that the RoadRunner Tracker app requires you to be connected to wifi. It will not work on cellular data.

Since:
`1.3`

Version:
`2.0`

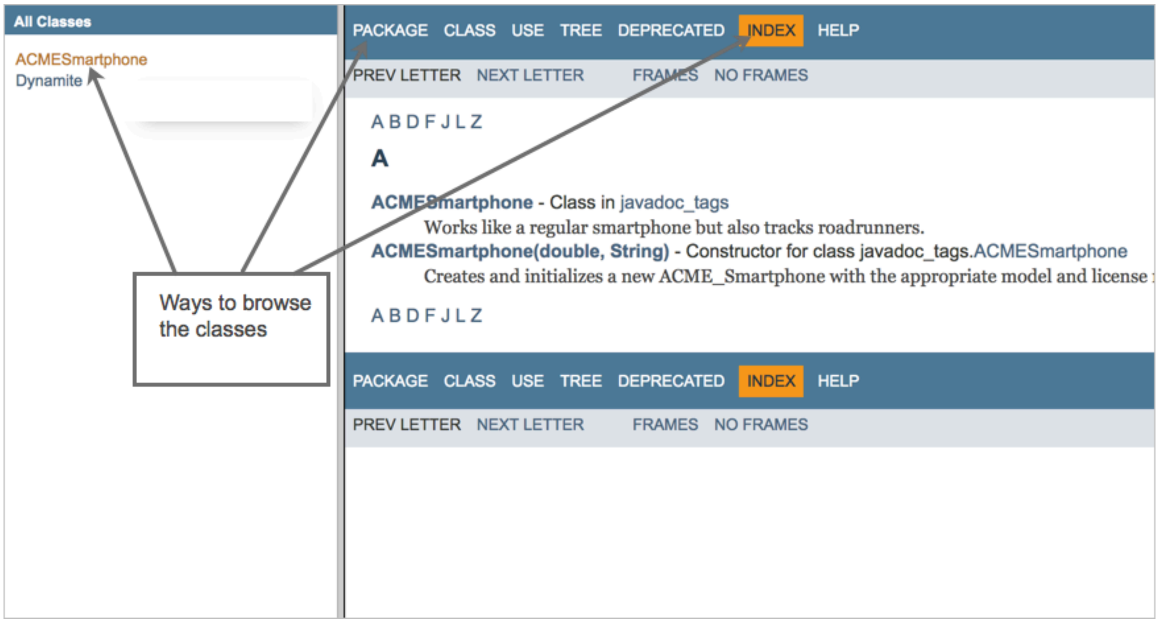
See Also:
`Dynamite`

Field Summary

Full class details

Other navigation

If you click **Package** at the top, you can also browse the classes by package. Or you can go to the classes by clicking the class name in the left column. You can also browse everything by clicking the **Index** link.



ACMESmartphone - Class in javadoc_tags
    Works like a regular smartphone but also tracks roadrunners.
ACMESmartphone(double, String) - Constructor for class javadoc_tags.ACMESmartphone
    Creates and initializes a new ACME_Smartphone with the appropriate model and license

All the ways to browse the classes

For more information about how the Javadoc is organized, click the **Help** button in the Javadoc top navigation bar.

Make edits to Javadoc tags

It's pretty common for developers to add Javadoc tags and brief comments as they're creating Java code. In fact, if they don't add some annotations, the IDE will usually produce a warning error.

However, the comments that developers add might be poor, incomplete, or incomprehensible. A tech writer's job with Javadoc is often to edit the content that's already there, providing more clarity, structure, inserting the right tags, and more.

What to look for when editing Javadoc content

When you make edits to Javadoc content, look for the following:

- **Missing doc**. Lots of Javadoc is incomplete. Look for missing documentation.
- **Consistent style**. See if the existing tags follow Java's style conventions with annotations (p. 712).
- **Clarity**. Some descriptions are unintelligible due to the curse of knowledge (it might be hard to tell without a stronger grasp of Java)

Make some edits to the Javadoc

Make some edits to a class and method. Then regenerate the Javadoc and find your changes. See how they get rendered in the output.

Tip: If you need to review how to export to Javadoc, see Activity: Generate a Javadoc from a sample project (p. 707). Each time you export to Javadoc, you need to select the classes you want to be included.

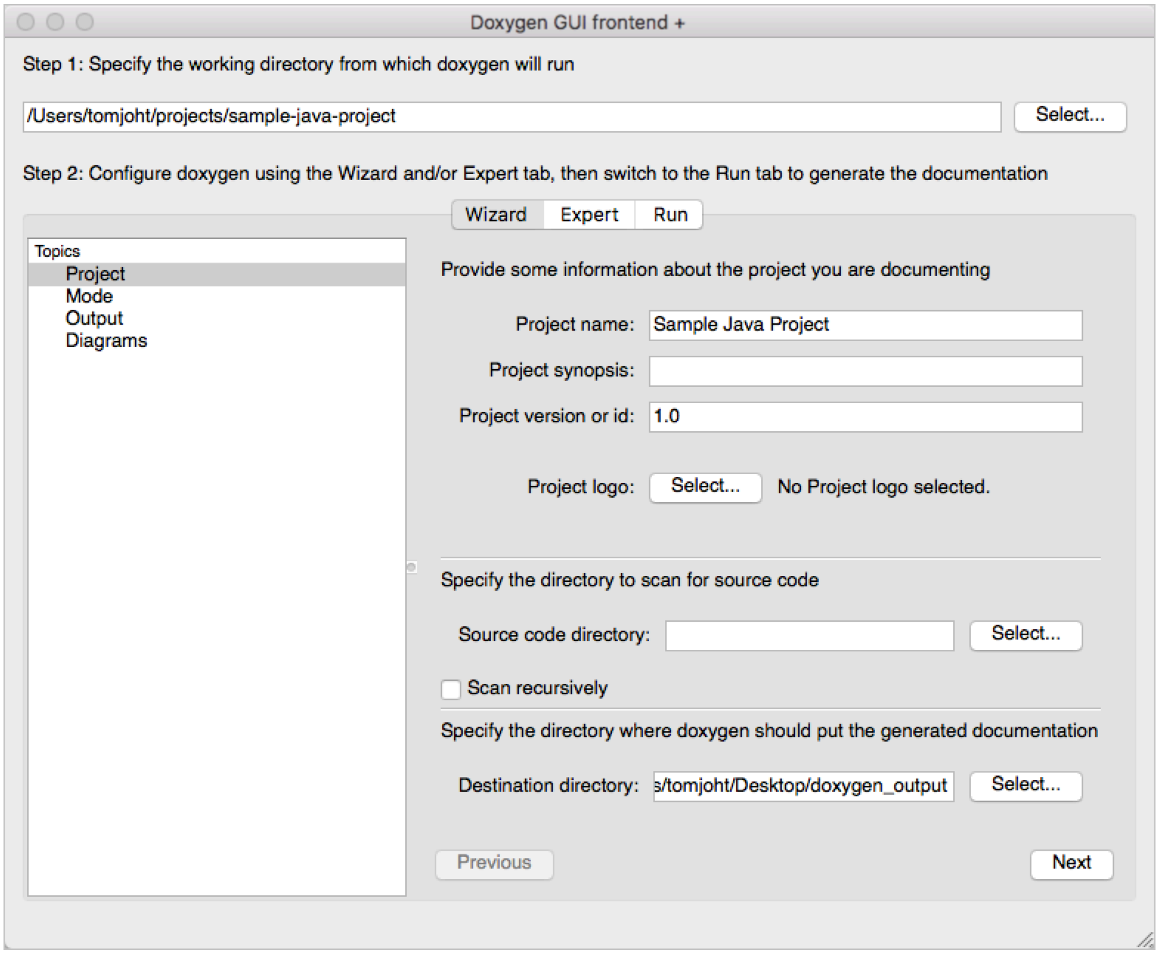
Doxygen, a document generator mainly for C++

An alternative to Javadoc is Doxygen. Doxygen works highly similarly to Javadoc, except that it can process more languages (Java, C++, C#, and more). Doxygen is most commonly used with C++. Additionally, there's a GUI tool called Doxywizard that makes it easy to generate the file.

Download Doxywizard

The Doxygen download also includes Doxywizard. See the Doxygen Downloads page for more download links.

Here's the Doxywizard front-end GUI generator:



Step 1: Specify the working directory from which doxygen will run

/Users/tomjoht/projects/sample-java-project

Step 2: Configure doxygen using the Wizard and/or Expert tab, then switch to the Run tab to generate the documentation

Provide some information about the project you are documenting

Project name: Sample Java Project

Project synopsis:

Project version or id: 1.0

Project logo: No Project logo selected.

Specify the directory to scan for source code

Source code directory:

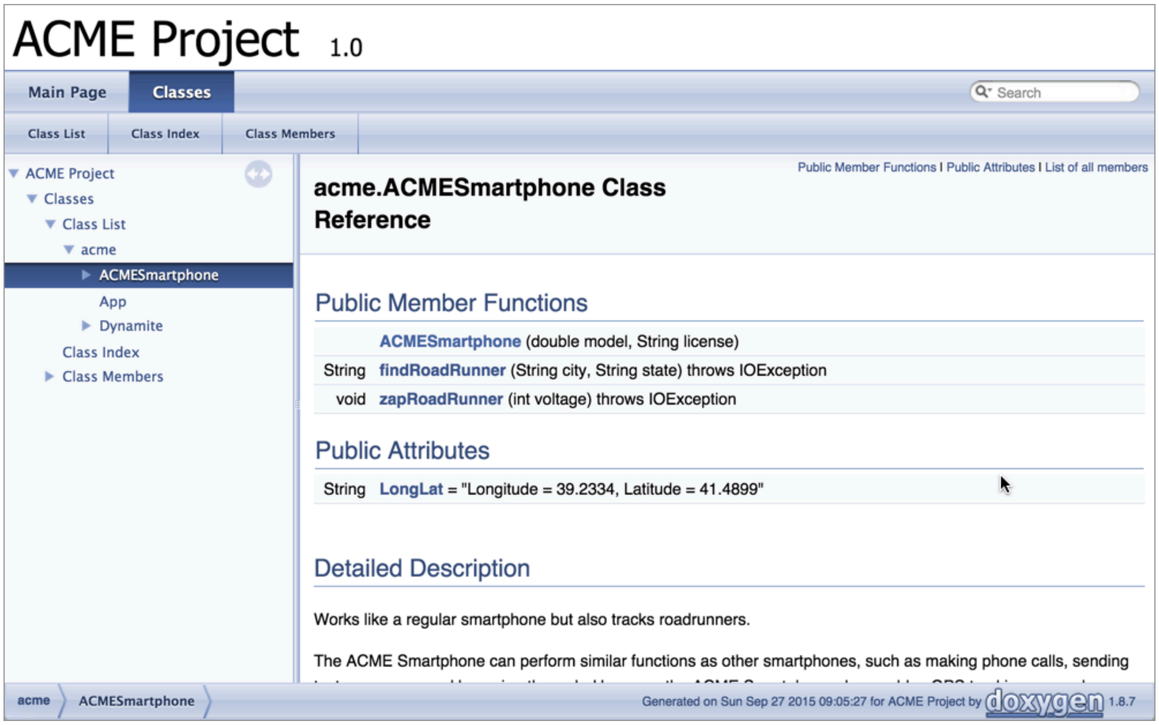
☐ Scan recursively

Specify the directory where doxygen should put the generated documentation

Destination directory: s/tomjoht/Desktop/doxygen_output

Doxygen front-end GUI generator

Here's the Doxygen output:



acme.ACMESmartphone Class Reference

Public Member Functions

	ACMESmartphone (double model, String license)
String	**findRoadRunner** (String city, String state) throws IOException
void	**zapRoadRunner** (int voltage) throws IOException

Public Attributes

String	**LongLat** = "Longitude = 39.2334, Latitude = 41.4899"

Detailed Description

Works like a regular smartphone but also tracks roadrunners.

The ACME Smartphone can perform similar functions as other smartphones, such as making phone calls, sending

Generated on Sun Sep 27 2015 09:05:27 for ACME Project by doxygen 1.8.7

Doxygen Sample

You don't need to use the wizard. You can also generate Doxygen through a configuration file, which is typically how developers run Doxygen builds from a server.

In contrast to Javadoc, Doxygen also allows you to incorporate external files written in Markdown. And Doxygen provides a search feature. These are two features that Javadoc lacks.

Doxygen is maintained by a single developer and, like Javadoc, hasn't changed much over the years. In my opinion, the interface is highly dated and somewhat confusing. But C++ developers will be accustomed to it.

Integrating builds automatically

In a lot of developer shops, document generators are integrated into the software build process automatically. Doxygen allows you to create a configuration file that can be run from the command line (rather than using the frontend GUI). This means when developers build the software, the reference documentation is automatically built and included in the output.

Other document generators

You don't need to limit yourself to either Javadoc or Doxygen. There are dozens of different document generators for a variety of languages. Just search for "document generator + {programming language}" and you'll find plenty. However, don't get very excited about this genre of tools. Document generators are somewhat old, produce static front-ends that look dated, are often written by engineers for other engineers, and not very flexible.

Perhaps the biggest frustration of document generators is that you can't integrate the rest of your documentation with them. You're mostly stuck with the reference doc output. You'll also need to generate your how-to guides and other tutorials, and then link to the reference doc output. As such, you won't end up with a single integrated experience of documentation. Additionally, it will be hard to create links inline within topics between the two outputs. Fragmentation of outputs is a topic I explore with more detail in Integrating Swagger UI with the rest of your docs (p. 241).

Create non-ref docs with native library APIs

Although much attention tends to be given to the reference documentation with APIs, the bulk of documentation that technical writers work on (as opposed to developers) is conceptual documentation (p. 358). Developers rarely write more conceptual or tutorial-based documentation.

Engineers will throw a quick description of a class in a file and generate a Javadoc (p. 707), and they'll give that Javadoc to the user as if it represents a complete set of documentation — but reference docs don't tell even half the story.

Reference docs can be an illusion for real doc

Jacob Kaplan Moss says ↗ that reference docs can be an illusion:

> … auto-generated documentation is worse than useless: it lets maintainers fool themselves into thinking they have documentation, thus putting off actually writing good reference by hand. If you don't have documentation just admit to it. Maybe a volunteer will offer to write some! But don't lie and give me that auto-documentation crap. – Jacob Kaplan Moss

Other people seem to have similar opinions. In general, document generators don't tell you a whole lot more than you would discover by browsing the source code itself. Some people even refer to auto-generated docs as a glorified source-code browser.

Reference docs are feature-based, not task-based

One of the main problems with reference documentation is that it is feature-based rather than task-based. It's the equivalent of going tab-by-tab through an interface and describing what's on each tab, what's in each menu, and so on. We know that's an ineffective way to approach documentation since users often organize their mental model by the tasks they want to perform.

When you write API documentation, consider the tasks that users will want to do, and then organize your information that way. Reference the endpoints as you explain how to accomplish the tasks. Users will refer to the reference docs as they look for the right parameters, data types, and other class details. But the reference docs won't guide them through tasks alone.

Chapter 8: Documentation processes and methodology

The process for managing developer portal documentation includes a number of different tasks outside of content development. The role of a DX (developer experience) content strategist plays a key role in defining workflows, standards, user flows, processes, and other aspects of the site. workflows for different types of documentation. In this section, I cover details for how to manage the doc work, including processes for fixing doc bugs versus tackling larger doc projects. I also explain processes for releasing SDKs, managing external contributions, and more.

Chapter sections:

DX content strategy with developer portals 728
Following agile scrum with documentation projects 733
Managing large documentation projects 737
Managing small documentation requests 750
Managing SDK releases 755
Documentation kickoff meetings and product demos 760
Processes for reviewing documentation 764
Maintaining existing documentation 770
Collecting feedback post-release 775
Managing content from external contributors 782
Changing internal doc culture 785
Sending doc status reports -- a tool for visibility and relationship building 789
Broadcasting your meeting notes to influence a wider audience 794
Ensuring documentation coverage with each software release 798

DX content strategy with developer portals

As the content grows on a developer portal, there's an increasing need for some technical writers to expand their documentation roles from individual contributors creating and publishing new content to dedicated content strategists. These dedicated content strategists manage the processes, standards, tools, governance, and workflows for the content that is primarily authored by contributing writer and engineering teams.

The general common industry term for this activity might be *content strategy*, but in the context of developer documentation, I'm calling this activity "DX content strategy." (*DX* refers developer experience, similar to how *CX* refers to customer experience.) Anyone who engages in DX content strategy would be considered a DX content strategist to some degree. An organization can have multiple people filling DX content strategist roles.

Risks of developer portals with no DX content strategy

If organizations don't have any DX content strategists dedicated to their growing developer portals and enhancing the developer experience, the resulting trajectory of a developer portal will follow the same path more or less as an internal wiki. Internal wikis tend to be content dumping grounds that are a wild west of standards and styles, and where you never know if content is current, outdated, accurate, deprecated, or under construction.

Finding the right content on an internal wiki is a constant challenge, and there's no consistency from one page to the next. In the wiki model, teams create docs during their times of need and abandon them when their teams or priorities change. The result is a mishmash of fragmented, often unowned content.

Because documentation for external developers is a high priority, it shouldn't befall the same fate as content on internal wikis. In the State of API 2019 report, Smartbear found that "accurate and detailed documentation" ranked third in priority (just below "ease of use" and "responsiveness/performance") from 15 ranking factors in an API experience. You can't achieve a high level of content quality without playing a more strategic role with the content and expanding beyond the individual contributor model. Individual contributors are usually focused on docs for a current project only and rarely have bandwidth to expand their concerns at a broader, more encompassing level across the developer portal.

Additionally, the individual contributor model doesn't scale for the typical engineer-to-writer ratio in most organizations — a ratio where engineers and other tech roles vastly outnumber writers. You might have just a handful of tech writers to support thousands of engineers. Tech writers often work with different product teams on a continually shifting basis, covering as much ground as possible but not having capacity to meet all needs. In this "itinerant" tech writer model, you can't always play individual contributor roles to write the content that each product team needs. Instead, writers have to shift to playing more content strategy roles around content that others develop.

To elevate contributor contributions to any standard, you have to equip contributors with easy tools, simple markup, fill-in-the-blank templates, clear standards, careful feedback, immediate tools support, and other information and tooling for their effort. You can't do this if you're an individual contributor fully dedicated to three or more full-time projects — often without enough time to finish, let alone teaching and helping other contributors.

Overall, the DX content strategist provides the mechanism to elevate the content in ways that improve the developer experience on the portal. As a strategist, this elevation of content doesn't come from creating better content yourself but rather from all the meta-activities you do around content development and publishing. At a high level, these meta content-development activities might include the following:

- Onboarding and training contributors (and providing materials to facilitate self-service onboarding)
- Providing standard templates and guidelines for content creation
- Defining processes and standards (such as for releasing SDKs or writing release notes)
- Making tools available that will automate content checks and workflows
- Defining authoring and publishing workflows and content reviews

These tasks don't involve creating new documentation. Instead, the DX content strategist focuses on the many activities required outside of content development to ensure the content quality meets a high bar.

Tasks for a developer portal content strategist

The following sections expand in more specific ways on the roles, tasks, and needs that DX content strategists play with a developer portal. I've divided the tasks into general groupings related to tools, contributors, processes and standards, and information architecture. Depending on the size of your doc team (e.g., 5 writers versus 150 writers), you might have specialized teams focusing on each of these areas.

Tools

- Set up, maintain, and enhance the authoring and publishing toolchain

- Troubleshoot the build when it breaks, and then train contributors about proper usage of the system to avoid future breaks

- Understand how the doc toolchain builds and publishes content from end to end (from static site generator outputs to build pipelines, CI/CD server deployments, and more)

- Reduce the risk of Git catastrophes and broken builds in the system (such as due to misunderstandings with Git or insufficient safeguards with permissions)

- Develop and enhance the doc theme's design and style as well as the available components (e.g., navtabs, accordions, or menus) to use for different scenarios

- Implement style/grammar checkers (e.g., Acrolinx) either with in-tool plugins or perhaps at the platform level

- Build out and verify scripts (e.g., Travis CI scripts, GitHub Actions) to perform automated checking, such as looking for broken links or style inconsistencies

- Integrate and configure the search experience (e.g., Algolia, Swiftype, other), including SEO and discoverability of content in search engine results

- Define the metadata that feeds into the search experience to provide navigable facets or other structured data for search; also monitor the documentation to ensure contributors are using the right metadata

- Address security tickets related to the developer portal (e.g., reverse tabnabbing, linking to external JavaScript sources, errant permissions)

- Configure scripts for PDF generation and output (e.g., Prince scripts) to distribute content to pre-release partners

- Manage the assets for your developer portal and permissions through the web console related to your website host

Contributors

- Develop training and guides for contributors to onboard with the toolchain and authoring/publishing processes
- Create templates for how-to content and other format types (release notes, getting started, first-time setup, CLI docs, schemas) to help guide authors as well as maintain consistency across docs
- Define Git workflows and manage Git permissions for contributors, including pull request workflows and code review workflows
- Create reports to assess each doc set to identify stale content (which hasn't been updated for a long time), missing metadata, outdated localization, and other problem areas
- Define a process for regularly auditing content to avoid stale, abandoned, or outdated content. Include timelines, requirements, and automated notifications to send out when content owners haven't reviewed their docs for a long period of time (e.g., one year)
- Create and maintain a list of contacts for every doc set, listing current owners and roles
- Review and monitor Git commits from contributors to identify problems or other issues with contributor content
- Analyze trending support tickets and hot topics (even when not filed against the doc team) and sync with engineering teams on resolutions
- Identify support paths and escalation processes for the different products on the developer portal

Processes and standards

- Define the process for managing large doc projects (p. 737), including processes for releases that have impact across field engineers, marketing, and support, etc, not just docs.
- Define a process for assessing incoming small doc requests (p. 750) and decide how to approach and prioritize them. Regularly groom the backlog so that you don't have a sea of stale tickets.
- Define and enforce the release process for SDKs (p. 755), sample apps (p. 476), and other code artifacts on the developer portal
- Set up and define localization processes, priorities, and workflows with regards to doc tools
- Define processes for granting access to docs prior to release for early access partners, such as during beta periods
- Define and set up on-call workflows and rotations to handle incoming feedback, emergency tickets, or other daily doc emergencies
- Define and enforce the doc review process prior to release, including any required sign-offs
- Define processes for deprecating documentation, including outreach and notification workflows as well as migration/upgrade paths

- Define strategies for versioning content, including approaches for incremental versions versus non-incremental versions, deprecated versions versus active versions, small version variants versus entire doc sets
- Promote awareness of Legal red flags and danger zones (p. 756) (e.g., forbidden terms or IP violations), as well as the Legal review process
- Define, implement, and enforce a docs style guide for the site. Also, define and contribute to an API design style guide and understand how it relates to documentation
- Ensure that all docs (especially from contributing authors but also from other technical writers) meet common standards, including release notes for changes published, getting started tutorials, proper syntax highlighting for code samples, etc.
- Define policies for when PDFs are delivered, how they are styled, what legal boilerplate appears on them, and how they are stored and distributed
- Define and promote standards for API reference content, such as using OpenAPI with REST APIs
- Define and review standards for tagging in auto-generated reference docs (p. 712) from library-based APIs (e.g., Javadoc, Doxygen)
- Define process and standards for developer product naming and branding, and then enforce this against individual teams that might launch new feature names in unapproved ways

Strategic alignment

- Align the team’s efforts and priorities with larger org’s priorities.
- Coordinate and partner with other documentation teams, aligning or sharing similar assets, workflows, tools, or processes
- Champion the doc team’s correct fit in the organizational chart (e.g., Engineering, Product, Marketing) and understand the reporting chain
- Report upwards with weekly, monthly, and yearly metrics or other data that resonates with the priorities of the leadership chain
- Identify and cultivate documentation sponsors at the executive level
- Review analytics and regularly investigate trending pages, then prioritize updates based on analytics
- Identify the 20% of pages that receive 80% of the traffic so you can identify and prioritize content for those pages
- Develop strategies for funneling information from users (e.g., external devs) back to internal product teams
- Review incoming doc feedback as well as developer satisfaction surveys and then take action on the results
- Build rapport with key documentation owners by providing regular updates about the analytics and performance of their docs, etc.
- Publicize updates to stakeholder mailing lists to communicate changes to all relevant parties
- Ensure the homepage and other marketing pages on the site properly match messaging in the documentation, including language, links, high-level processes, terms, etc.

- Contribute to regular communication that is sent to developers (e.g., a newsletter) to build their trust and awareness with the company and dev portal
- Guide contributing bloggers to address doc needs, and capture blog information that might be worthwhile to incorporate into documentation

User flows

- Define user journeys from marketing landing pages and other incoming sources to the docs
- Arrange multiple doc sets into a master index or starting point that makes sense for the organization
- Integrate a support path from the docs to support channels or contact us pages so that users know how to get help when they need it
- Define and promote flows from any admin or console pages to the docs through contextual help links
- Research and understand developer journeys on competitor sites
- Understand how every product fits together as a whole across the developer portal (rather than only understanding the docs you work on)
- Build in mechanisms for collecting feedback, including design of a feedback form and questions, the automated triggers when feedback is received, and the workflow for handling feedback
- Develop awareness and review of activity outside the docs on social channels such as Stack Overflow, GitHub, Twitter, etc., and assess whether the docs align with the terms, hot issues, and priorities of the external activities
- Drive developers to sign up to newsletters and other forms of outreach from the docs
- Build funnel metrics in multi-page tutorials to analyze where developers drop out of tutorials
- Build user profiles that provide authors and contributors with a better understanding of the audience, such as their technical level, companies, purposes, and other needs

Bandwidth for all of these tasks

Reading through these tasks is probably overwhelming, but my purpose is to demonstrate the need for these activities outside of content development. All too often, people assume that technical writers merely “write docs” and don’t understand that there’s a large body of work involved in managing a developing portal that doesn’t include writing content. These tasks try to answer the question, “what else do technical writers do besides write docs?” When you allocate resources for DX content strategists, they can help build a healthy developer portal so that users can easily find and consume accurate, relevant content that addresses their needs.

It’s not always feasible to have a dedicated person fill the DX content strategist role. Many times different technical writers and managers will play the role in varying degrees. However, it’s good to have these tasks be part of an official role rather than an add-on to individual contributor role whose primary job is creating docs. Unless these tasks fall under someone’s official title, they are likely to be overlooked. This creates more burden on writers to figure out their own tools, processes, and other details.

Following Scrum with documentation projects

Just as doc-as-code tools (p. 551) mirror tools used by software implementation teams, work processes that tech writers follow can also mirror processes used by software implementation teams. The most common software development methodology today is probably Scrum, which is a form of agile development methodology.

For a longer, more detailed version of this workflow, see the Managing large documentation projects (p. 737) and Managing small documentation requests (p. 750).

Although Scrum is both loved and hated by engineers, and nearly everyone modifies their implementation of it, this methodology has resonance with engineers because so many engineering groups loosely follow the same Scrum process. Scrum is an extremely common approach within the software development industry. When tech writers adopt a similar methodology, engineers they're working with better understand the tech writer's processes and workflows.

Scrum isn't the only software development methodology engineers follow. Your engineers might follow a Kanban, Waterfall, DevOps, Rapid Application, or other approach. In fact, even if they're following Scrum, it will likely be adapted/customized/altered. The general principle here is to adopt a methodology that syncs with the way your company develops software. I've chosen to focus on Scrum here because it's the most common approach.

Introduction

If you're unfamiliar with Scrum, consider becoming familiar with the methodology first before reading my adaptation of it below. Start by reading The Scrum Guide. If you prefer an earlier book version, see Scrum: The art of doing twice the work in half the time. This is the manual for the approach (it's a short book.).

Plugging into an engineering scrum versus creating your own doc scrum

The first question is whether to join an existing engineering scrum versus creating your own documentation scrum. When it makes sense, such as for large, ongoing engineering projects where you'll be a regular contributor for a number of months or so, prioritize joining an engineering scrum rather than creating your own documentation scrum.

There are several advantages of joining an engineering scrum:

- You'll have better rapport with engineers. They'll know you and you'll know them (often through the daily standups and other meetings). This will simplify working together and getting the information and doc reviews you'll need.
- You'll be in touch with the needs and priorities of the project. There won't be a huge gulf between you and the engineering team where you'll wonder what they're doing and vice versa.
- You'll be much more accountable to keep on track as you report into a daily scrum to let others know what you accomplished the day before and what you're working on today. More than anything else, this helps you stay committed to the project.

Despite the advantages of integrating with engineering scrums, there are a few disadvantages as well:

- If you're a temporary resource on the project, with a work duration of a month or so only, then it probably doesn't make sense to join the engineering scrum. There's too much onboarding, familiarization with their process, and more.

- If the scrum is run poorly, such that daily standups last 30+ minutes, and you have multiple scrum teams you're integrated with, this can be a real time killer that will drain your bandwidth while giving very little in return.
- Most likely you'll have several projects going on at once. If you have to modify your approach to each with a different flavor of Scrum, then your own workflow and methodology can get to feel a bit disjointed. For example, if each scrum calculates points differently, has different sprint durations, and has other variations, this inconsistency with your methodology can be tiring.
- If the engineering team wants you to attend all their scrum meetings but won't treat you as a full-fledged member of the scrum (e.g., no tasks, no points, no demos, etc.), then consider creating your own documentation scrum instead.

Adapting scrum for documentation projects

If it doesn't make sense to join an engineering scrum, you can create your own scrum process. My adapted process for managing docs following an Scrum methodology involves the following steps:

1. **Identify upcoming projects and other work (sprint planning)**. Before each sprint, review upcoming projects and other work, such as by reviewing launch calendars, intake buckets, support forums, planning roadmaps, and more. Get a sense of the work and priorities. The point is not to be surprised by work that shows up on your door two weeks before the deliverables are needed.

2. **Create a documentation plan for larger projects**. My documentation plan typically follows this documentation plan template and is adapted for the project at hand. This plan provides a host of details that I need to keep straight about the project. It isn't a waterfall approach or a doc outline but rather a list of notes about the project, such as who's who, where QA test scripts are, expected deliverables, when release dates are planned, where key product documents are, and so on. This doc plan functions as a kind of run book for the project, with a section that lists running notes.

3. **Chunk doc work into tickets**. From the doc plan, create tickets (e.g., JIRA issues) related to the work. The tickets should roughly outline the major tasks for each project. The tickets don't need to be comprehensive from the start, but they should give a sense of the work required. Also, you don't have to log all the tickets from the beginning, as they will likely sit in a backlog and become outdated before you even start working on them. The basic idea is to simplify complex tasks by chunking up the work into small tasks.

   Since larger projects might have numerous tickets, you can create a master ticket that functions as an umbrella for all tasks related to that master ticket. This umbrella doc ticket can simply point to a folder or label that aggregates all the other tickets for that project.

4. **Estimate the point weighting for each ticket**. Points communicate the complexity of the project. Every team seems to vary a bit in how they weight their tickets, but this is the approach I follow. A full day of work is 2 points; a half day or less is 1 point. I like to chunk my tickets at no more than 5 points because I want to show progress and feel like I'm completing work. Even if it's a short fix that takes me 10 minutes, I still log a point for it. (More granular point weighting is typically discouraged in agile methodology.) Points are important because outsiders to documentation often don't understand if a task is complicated or easy.

5. **Assign tickets to bi-weekly sprints**. Tickets should be assigned to sprints. Each sprint is usually two weeks (but can be a different duration, if you want). For each sprint, the total effort per writer should add up to the number of points you can burn (complete) during that two-week

period. This point completion rate is called your “velocity.” This number is based on previous velocity calculations, so you won’t know your velocity at first. You’ll only know after several sprints. My velocity is about 10 points per week. Calculating and communicating your velocity is important so that you know if you’re staffed appropriately for the work given the release deadlines.

6. **Stakeholders should be aware of the work assigned to tech doc sprints so they can see the progress of their projects** (and have realistic expectations for when their docs will be worked on). Sprints shouldn’t change up their assigned items unless a doc has higher priority. Accommodating doc emergencies and crises needs should be something to avoid by following this process.

   Maintaining the plan of the sprint is extremely difficult in a documentation scrum. Different teams might have immediate needs for quick updates. These quick updates might involve a half a day of work or less, or might even involve just fixing a typo. For these quick tasks, I add them in an ongoing, dynamic way to the sprint as needed. However, if someone approaches me with a significant doc project, I tell them that I’ll assign it to the next doc sprint (which will likely be two weeks away). People can’t really expect me to drop everything and work on a large doc project immediately, without some leeway, so telling them that I’ll assign it to an upcoming sprint usually appeases them and gives them some reassurance knowing that their work is scheduled, even if nothing is done currently.

   More than anything, this is why the scrum process is important — it protects your current workload from becoming too frantic/heavy/scattered. You shouldn’t have to exceed your current velocity because of the documentation tasks that are pending — you just push the work out into the future. I realize that releases and high severity tickets might require you to reprioritize on the fly, but this shouldn’t be the norm as it is an approach that will likely exhaust you in the long run.

7. **Post bi-weekly reports at the end of each sprint**. At the close of each sprint, share details of what you completed with all stakeholders interested in their work. This typically involves sending updates to email distribution lists. The reports show the tickets completed from the closed sprint and the tickets planned for the next sprint. This same report might be repurposed into other monthly reports from your team to upper management.

   The sprint report is one of the most important tasks you can do. First, it lets people know what you’ve been working on. It also provides a record of your work in a way that you can show off. Others are usually fascinated to see the doc work you’re doing and will be excited to see the details. Sending regular reports like this might be one of the most influential activities you can do inside the enterprise to promote your team.

8. **Follow the review process prior to publishing docs**. Prior to publishing docs, try to follow a stringent review process to ensure the docs meet your quality bar. This review process is similar to the sprint demo with software development, where you check in with customers to see if what you’ve developed is meeting their needs. Typically, I try to review documentation chunks related to the tickets completed. If I try to review too much content at once, people often lack the bandwidth to review it. My review process includes six quality checkpoints:

   - *Review with the doc team.* The doc team refers to the tech writers creating the content. Test all the instructions end-to-end yourself, walking through each step. This might involve developing a test app or other sample code.
   - *Review with the product team.* The product team includes the engineers who coded the product and the PMs involved. They should sign off on the accuracy and completeness of the docs.

- *Review with field engineerings, business development, and support.* Expand the review circle to include additional groups and stakeholders with interest in the docs. Send docs to these groups for review and then meet with them to gather their feedback.

  Some groups refer to read the documentation for the first 20 minutes of a meeting and then provide their review, completing the whole read and review task during the meeting itself.

- *Review with Legal.* If the doc has any red flags that might cause concern with Legal, interface with your Legal group for review.
- *Review with early release partners.* For larger projects, you might release your docs to beta partners first prior to general availability. Generally, field engineers funnel back feedback from early release partners.

If documentation doesn't go through this review to some degree, avoid publishing it. Otherwise, if you skip some of these steps, you risk pushing out poorly written docs. Again, the review process aligns with agile methodology in that it provides the check-in with your customers to make sure you're on the right track. This review process is where your customers are inspecting your work and providing course corrections as needed.

- *Collect feedback post-release*. After you publish the documentation, embed a Feedback button directly in your docs to collect additional feedback from customers in an ongoing way. This incoming feedback trickles in and might not contain significant or actionable information, but customers should have some way of relaying their feedback. As a tech writer, I want to know if there is some major issue with docs so that I can fix them.

Conclusion

Without a process for managing tech docs, projects might arrive from seemingly anywhere at random times, assigned to the doc team, with little info or scope or resource allocation. As a result, tech writers can end up in crisis mode, or product owners might have unrealistic expectations about deliverables, etc. Tech writers might get exhausted or feel like they don't have the time or resources to produce the needed quality with docs.

By implementing a more formal process and methodology for managing tech doc projects (specifically, an adaptation of Scrum), you can avoid this scenario. Additionally, managing and tracking projects this way gives each team member greater visibility and accountability towards the work of documentation.

Additional resources

See these posts on my site for more reading about Scrum and tech docs:

- Increase efficiency 24 times faster when fixing errors — implications for technical writers on agile teams
- Tech docs and Agile: Problems with integrating tech writers into engineering scrums (Part 1)
- Tech docs and Agile: Alternatives to integrating into engineering scrums (Part 2)
- How can technical writers thrive in agile environments? Event recording and details
- How to apply agile processes to manage your life's projects

Managing large documentation projects

When you’re managing larger documentation projects, you need to incorporate project management skills to break the large work into smaller chunks. You can pick and choose from Scrum and other agile methodologies (like Kanban) to create a workflow that works for you. The following sections describe a process for working with large documentation projects.

I also wrote about some of these processes in Following Scrum with documentation projects (p. 733). This section expands on that initial framework with more process details and commentary.

Definition of a documentation project

A “documentation project” is a larger effort tied to some new feature or product launching, whereas a small documentation request (p. 750) might be a minor update or fix to an existing doc. In both cases, you can manage documentation work by loosely following the same adapted Scrum processes used in many engineering workflows. This is because documentation processes are closely coupled with software engineering in terms of its cadence and workflow. In both cases, you’re building something complex.

However, because tech writers are often allocated to multiple Scrum teams, and because tech writers are often resourced under a different group organizationally (e.g., tech comm), implementing Scrum in the strict, traditional way usually doesn’t work without some adaptations. In the sections that follow, I’ll elaborate on what I think works in a practical sense.

Process overview

The following sections outline a process for *managing doc projects*.

1. Identify the documentation projects

First, try to identify all documentation projects by reviewing different project lists for the teams you’re working with. The whole need to identify the projects might seem unintuitive — shouldn’t product managers reach out to technical writers to let them know what doc needs they have? Yes, in an ideal world that would be the case. But we all know the reality in corporate settings. About two weeks before release, the product manager suddenly starts thinking about docs, and an email winds around to your inbox with the request. If you were to start from the time you received the email heads down until the release, working on that project and only that project, you might be able to finish it — meanwhile postponing/re-prioritizing everything else you’re working on as you enter crisis mode. Most people dislike working for extended periods in crisis modes.

To get out of crisis mode, you have to scan the project landscape like a scout looking across a plain for emerging wildlife and identify the projects early on. Identifying projects early on also gives you more opportunity to engage more deeply in the project, such as providing input on the API design and usability (p. 488).

So this first step is one of *identifying* the work — what projects are out there, what’s on each team’s roadmap, when are new features and products going to be released, which of these projects will require documentation, and so on. Gather up a list of these projects and put them onto a documentation planning spreadsheet.

While this project identification task might seem common to any documentation team, this work is magnified in scenarios where you're publishing on a developer portal. A developer portal tends to be a publishing hub for many different teams, often working independently. A thriving developer portal might have documentation input from dozens of different teams, many of whom might be unfortunately siloed from each other but publishing through a common documentation team steward/gatekeeper.

As you're scouring for this project information, look for roadmap trackers maintained by all the teams you support. For example, you might track about half a dozen different roadmaps from teams. Some projects might be listed on multiple roadmaps because different teams might play different roles with the projects. For example, visual designers might only have UX mocks on their roadmap, while engineering teams might have engineering tasks related to the project, and so on.

Sometimes a large project can have lots of sub-teams that are working on different dimensions — an API team working on handling databases to store information, a UX team working on the admin console, another team developing a CLI to handle input or other commands, other teams working on other features, and so on. Usually as a technical writer, you see this larger picture and support all the teams, even if these teams only have awareness of their own projects.

In this stage, you're looking only for "projects." What is the definition of a project versus a small documentation request? Here we get into a semi-gray area, but it's essential to separate the two types of work because they have different workflows and needs.

A *documentation project* is any work that typically has associated business documents, design documents, or other requirements documents with it, along with a product manager who is defining a large scope of work and a team implementing some feature or product. Projects usually have a release as part of their timeline.

Sometimes smaller efforts can be grouped under larger projects. For example, a project might include a lot of sub-projects released at different timelines, but the larger effort overall is managed by a PM. (Later on in the process, I'll describe why it's essential to break these larger projects down into smaller tasks.)

Doc work that isn't associated with a project can be referred to as a "small documentation request." Small requests can usually be made in a few days or less. Some examples of small requests would be to remove an outdated paragraph, update availability information for a product, review a contributor's updates, publish an SDK minor version with release notes, etc. These small tasks aren't listed on your documentation project planning spreadsheet. (I go into detail about managing small documentation requests here (p. 750).)

What about extensive doc efforts that might take 1-2 weeks but which aren't part of any larger project? In these cases, you're essentially managing the project yourself, and I would recommend adding it to your documentation project spreadsheet. If you need a more definitive separation between projects and small doc requests, maybe consider defining a project as any effort requiring more than a week of work.

The converse scenario might also apply: documentation needed as part of a project but which only takes 1-2 days. In that case, I would still list the effort as a project because projects have different rhythms and workflows — there are often multiple phases (internal, beta, general availability, fast-follows), as well as multiple iterations as teams evolve product designs based on customer feedback. You might have to rewrite the topics three or four different times as part of the development churn. As a result, what might initially begin as a small amount of work can turn into a much larger work.

When do you sit down and identify all the projects? Who reaches out and contacts the different teams and finds their roadmaps and tracks what's coming and so on? You could designate a doc lead for this. However, it might be better to designate different writers who are specialized in different areas to do the information collection work. For example, suppose a documentation team has 5 writers, each of whom is embedded with 2-3 teams in different areas of the org. Writer 1 might be assigned to projects in Group A, Writer 2 might be assigned to projects in Group B, and so on.

This specialization not only lets tech writers become SMEs in these areas of the business but builds rapport with the leaders in that group. As such, it makes sense for these embedded writers to interface with the leaders in the group around roadmaps. If a doc lead instead performs this function, the analogy becomes more like a waiter (doc lead) interfacing with the customers at tables and then handing orders to cooks (tech writers), who remain isolated from the customers at tables. Instead, if each writer interfaces with customers to gather up the orders, chances are this interaction will build a stronger rapport and relationship with the teams, which will serve the documentation efforts more in the long run (such as when you ask for doc reviews or other information). In short, all writers should be scouts to identify the projects in their respective domains.

To enforce these efforts on a regular basis, doc teams should gather bi-weekly for a “Documentation Planning” meeting where they specifically review the various roadmaps of the teams they support. If multiple people are invited to the meeting, you’re more likely to keep it going. During the project planning, list out projects onto a documentation planning spreadsheet.

If you’re not part of a doc team but are fully embedded with a product team as a solo writer, you can still perform the same function but might need to drive the whole process yourself. Solo writers often support many different disconnected teams within an organization as well.

2. Assess the identified work against strategic priorities

The next step is to assess the identified work against your department’s strategic priorities and decide which projects and tasks should be given more importance. This prioritization is probably only needed in situations where the tech writers have more work than they can feasibly handle. However, this is almost always the case, not an anomaly to the norm.

Even in cases where each tech writer has a healthy allocation of projects, you will still probably have a lot of latitude/flexibility about which projects you devote most of your energy. It’s worth understanding which projects are considered a strategic priority.

Track down your organization’s strategic goals from your C-suite execs or other senior leaders as a way to figure out these priorities. If you’re so far down the org hierarchy that you’ve never seen such a document nor are allowed to access it, find out from your manager what the goals are. Your manager should be aware of his or her manager’s goals, who should be aware of his or her manager’s goals, and so on up the line. If there’s a disconnect somewhere, it represents a broken line of communication and potentially an organization that isn’t aligned.

Decide which projects should receive your most thorough attention (based on the projects that your senior leaders are prioritizing). This alignment is key for boosting your value to the organization, as I described in Part V: On being strategic, interpersonal, and sponsored ↗. After you get a sense of your organization’s strategic priorities, assign each project on your doc planning spreadsheet one of the following priorities: low, medium, or high.

Keep in mind that strategic priorities differ by group. What might be a strategic priority for one group might not be a strategic priority for another group. This prioritization can be especially problematic if your group is cross-functional, meaning you support many organizations with different services (e.g., documentation, marketing, support, dev relations, and so on). In these cases, you’ll have to figure out which priorities your own group wants to embrace.

The review of strategic priorities and the assessment of projects should ideally take place at a regular cadence, and although filling up your calendar with meetings isn’t ideal, this meeting (especially when it involves multiple people) can be a forcing function for this task. You might call this meeting “Project Prioritization” and even include field engineers in the meeting (especially if the field engineers have a close ear to important business partners).

3. Assign the projects to different tech writers

At this point, you have a documentation project spreadsheet with dozens of projects from different areas of the organization. You need to group the projects into buckets that make sense and which can be correlated with different tech writers into areas of specialization.

For example, suppose you have 25 different projects, and you have about 5 different tech writers. You might divide 25 by 5 to come up with 5 different groupings of the project (or create some other logic that makes sense here based on the project size, scope, and complexity).

These different buckets become areas of specialization for the writers. For example, “Bob” takes on all projects originating from the ACME group and which deal with Widgets. Bob becomes immersed in ACME’s widgets and teams, and builds rapport with that group the more he embeds with them. Meanwhile, “Cassandra” takes on all projects originating from the Globex group or which deal with Components. Cassandra immerses herself as much as possible in Globex teams, attending their standups and other meetings and such. In this way, everyone in Globex gets to know who Cassandra is and learns the process for docs.

It’s important that each tech writer is assigned an area of specialization so that they can accrue the necessary knowledge and rapport with teams in those areas. Each project on your doc planning spreadsheet should be categorized into an area of specialization correlating to a tech writer’s specialization.

What happens if the groups don’t balance out evenly? Suppose Globex pushes out tons of features while ACME is much smaller and only has 1-2 releases a year? In that case, you might have to make adjustments so that the work is more balanced. These adjustments can only be decided based on the number of writers at your disposal, your org structures, and other factors. But if you have a list of strategic priorities for your org, these strategic priorities can serve as the guiding core for balancing the work. Ideally, balance writers based on strategic priorities rather than simply the number of projects.

4. Create project plans for the documentation projects

Now that you have a list of projects and priorities, along with different specialization categories and assigned writers, you’re ready to move on to the next phase: creating project plans for these projects. Each tech writer will need to sync with the different product managers needing information for these plans.

Each tech writer should reach out to the respective product manager (PM) to find out what docs are needed for their project. PMs might reach out proactively as well, which is great when they do.

After some back and forth over email or other channels, the tech writer would then ask the PM to create a project-intake ticket for the needed documentation. The project-intake ticket should require the PM to complete a comprehensive, in-depth set of fields related to the project. Here’s an example of what this project-intake ticket might consist of:

Documentation project plan

This project plan defines the scope and details for a larger doc project. Complete as much information as you can, as this will help us get better oriented and familiar with the doc needs.

Project overview

Tell us about the project:

- Product name:
- Product code name:
- Brief product description: [3-4 lines]
- Project status page:

- Business requirements doc:
- Design documents:
- Functional analysis documents:

Business group

Which business does this project fall under:

- Organization:
- Business group:
- Team:

Product team

Who's who on the project team:

- Product manager:
- Project manager:
- Software developers:
- Quality assurance:
- Marketing:
- Developer outreach:
- Legal:
- Field engineer:
- Evangelist:
- Security:

Priority

Is this documentation a strategic priority:

- Does this project roll up under a strategic leadership goal? If so, which one?

Target users

Who is the documentation for:

- Developers
- Internal engineers
- Business decision-makers

Issue tracking

Where can the tech writer track work related to the project:

- Ticketing system:
- Wiki:
- Code repository:

Release timelines

- Beta partner release date:
- General availability (GA) launch date:
- Date docs must be ready to distribute:

Information sources

How does information flow within your team:

- Key team meetings:

- Team sprints:
- Scrum leader:
- Email lists:
- Slack channels:

Code repositories

Where the code is stored:

- Build system:
- Other storage:

Testing

How is the product tested:

- Test environment:
- Test scripts:

Localization needs

Is localization needed:

- Localization required: y/n
- Date localization needs to be complete: [mm/dd/yyyy]
- Localized languages:

Restricted access

Will this documentation be public or protected:

- Protected behind a login: y/n
- Public to everyone: y/n

Documentation size estimate

How much documentation do you anticipate for this project:

- 0-5 pages: y/n
- 6-9 pages: y/n
- 10-19 pages: y/n
- 21+ pages: y/n

Scope of content requests

What type of content do you need:

- Documentation: y/n
- UX microcopy: y/n
- Screencasts: y/n
- Email campaign content: y/n
- Blog articles: y/n
- Other: y/n

Reviewers for content

Who will review the docs and other content:

- Person 1:
- Person 2:

Support post-launch

Who will support the product post-launch:

- Support team:
- Is the support aware of this project: y/n
- Long-term owner post-launch:

Internal product wiki/resource pages

What wiki pages are important to be aware of for this project:

- page:
- page:
- page:

Marketing support

What marketing support do you need for this project:

- Marketing landing page: y/n
- Email to general devs: y/n
- Email to a targeted subset of devs: y/n
- Paid advertising efforts: y/n
- Have you reached out to anyone in Marketing? y/n

Other

List any other details not requested above that might be important to know.

(Note that you can usually populate tickets based on a template like this. For example, JIRA lets you do this.)

To instill a sense of ownership and commitment, it's important that the PM complete the project-intake ticket, populating the necessary fields. This information helps the tech writers more appropriately scope and assess the work, as well as get oriented on the tasks.

This information collection reduces email churn and the need for tech writers to figure out the answers themselves through meetings and other email threads. If the PM leaves the fields blank, consider setting up a meeting with the PM and go over the needed fields in the meeting. You would be extremely lucky to find all information completely filled out from the start. Most of the time, PMs fill out about a third of these forms because they themselves lack more details (e.g., test scripts, code repositories, etc). You might want to loop in the Engineering manager to complete the other details.

Note that the time required to collect this information can either be absorbed upfront through the requester or later through the tech writer. Placing the responsibility on the requester helps tech writers be more efficient. It also gives you a filter to decide just how desperately product teams want documentation for their projects. If a PM can't be bothered to complete the intake ticket, why should tech writers be bothered to create the documentation?

However, just because a PM or other requester is slow to make the request, don't rush to judgment about their interest in the docs, especially if you're making the requests over email. For many people, email is just noise, especially if they don't know you well. Sometimes I've followed up with PMs about emails I've sent, and they were oblivious to the context or purpose of the email (if they even read it). PMs might not understand what you're asking them to do unless you explain it to them through a meeting. Unfortunately, meetings are often the only language PMs understand.

5. Convene a documentation kickoff meeting

After the project-intake ticket has been completed, the tech writer schedules a documentation kickoff meeting with the PM and other interested stakeholders. The tech writer reviews the information in the project-intake ticket and asks questions and gathers other details during the meeting.

During this meeting, the tech writer should set expectations about what will be delivered, clarify timelines, push for demos or sample apps, describe the process for working together on the project, understand who the reviewers are, describe the review process, what's in scope or out of scope, and more. If the requester left fields blank, the tech writer should push on reasons why or who might have the information. It's better to collect the information upfront while you're in information-collecting mode rather than later, when you're already supposed to be writing the documentation.

The tech writer might decide to copy this information into a note-taking wiki and store it with other doc plans. After the kickoff, the personal wiki can become a place for the tech writer to store running notes, questions, or other details specific to the project. This wiki can become a log of important information for the project, perhaps where the tech writer organizes all the information needed for the project.

In cases where one tech writer leaves or transitions to another project, a new tech writer can review this wiki and get a sense of the project's status, contacts, and other details.

6. Chunk project work into tickets

The incoming project request acts as the parent ticket. Your next task is to chunk this large documentation project ticket into smaller tickets (still grouped under the parent). (Any project management methodology will have you break up large work into smaller chunks.)

Each child ticket represents a chunk of work for the larger project. Each chunk of work typically represents 1-2 days of work, no more. This chunking helps keep tasks manageable and provides a sense of progress on the project. Also, if you're able to identify all the sub-pieces of work, it will help you more accurately represent the scope of the work.

Tech writers can create the smaller tickets themselves, rather than requiring the PM to create each individual ticket. This is because the PM often doesn't know how to break down the work of a larger project, or because the requester might not know all the details of what is needed.

Compare this to an engineering team where a PM might ask for a certain feature — a mobile app for Android, or something. The engineering team analyzes the goal and then breaks the request into multiple sub-tasks (e.g., set up the mobile templates, create databases to store input, define layouts for the pages, and such) that are beyond the PM's original understanding of what is needed to bring about the end goal. Even though the tech writer creates these tickets, the requester should always be added as a watcher and the child ticket should be linked to the parent ticket.

All child tickets don't need to be created from the start, since it's not always possible to know the end from the beginning. And sometimes you only begin to see the work required as you get into the tasks. Part of the philosophy of agile is to plan your work in short sprints (usually two weeks) rather than six-month + intervals. As such, you need only sketch out the tickets for the next month or so. Also, avoid creating too many tickets that you drown in tickets. You can only see so far into the future.

Each tech writer does this chunking for his or her own projects. Exactly how you chunk up the tasks depends on the nature of the project. You might identify a list of expected tasks the user would need to perform and make the doc for each task a separate ticket. Or the chunking could involve identifying common doc types (e.g., Overview, Getting Started tutorial, Configuration, Sample App, Testing) and devoting each ticket to a different doc type. See Conceptual Topics in Docs (p. 358) for some ideas here beyond the Reference material (p. 118).

These chunked tickets should originally appear in the backlog of your ticketing system (to be later assigned to a sprint — described in the next step).

7. Assign tickets to a sprint board

With projects chunked into tickets, the next task is to next decide what tasks to promote from the backlog to a sprint. Sprints are typically two-week work cycles in which the team attempts to complete a defined set of work. (Sometimes sprints are one week, other times three weeks — try to align with the general approach engineering teams take in your organization, as it will breed better understanding of your processes when you have discussions with engineers.)

Each sprint has a capacity. For example, the sprint capacity might be 10 points per writer. For a team of 5 writers, the sprint would allow for 50 points. This number represents the team's *velocity*, or how fast your team can get work done. Theoretically, this velocity can be used to estimate how fast teams can get work done.

Suppose your project has a backlog of items totaling 1,000 points, and a stakeholder wants to know when the project will be completed. The team could say that, based on the current team's velocity (which is driven by the team's resources), the team can complete the backlog in 20 sprints, or 40 weeks.

But agile teams don't usually define all the needed tasks upfront in an endless backlog. The whole idea is to constantly course-correct by demonstrating the output of your sprint to the owners, and then make adjustments for the next sprint.

During sprint planning, writers assign points to the tickets. Each ticket might be weighted as follows:

- 1 point = 0.5 days work or less
- 2 points = 1 day of work
- 3 points = 1.5 days of work
- 4 points = 2 days of work

Larger weightings are discouraged. For example, if the task would take 5 days, as a best practice, break this up into smaller chunks to feel a greater sense of progress and so that you more accurately identify the work that needs to be done. However, avoid chunking tasks too granularly. If the task takes 10 minutes, see if you can group these smaller tasks into a single ticket.

Although I've used points here, many times teams will use non-point sizing, such as using T-shirt sizes or dog breeds (is it a Great Dane type task or a Chihuahua). Psychologically, it's easier to identify an XL task versus a small task rather than equate them with time. However, these non-number sizings ultimately resolve back down to points in the ticketing systems.

Typically, each writer usually doesn't take on more than 10-20 points per sprint. This provides a more practical sense of timelines and forces you to decide on priorities. When you realize that you can't do everything at once, you're forced to decide what you *should* work on first, and this is where prioritization comes into play.

During the bi-weekly Sprint Planning meeting, each writer weights their items and assigns the highest priority items from the backlog into the sprint.

8. Start creating content

As tech writers begin developing content, there are two primary ways to go about it. For new content that involves a significant amount of new material, which will require review, comments, iterations, and other discussions, I recommend starting the content in a collaborative wiki platform (e.g., Google Docs, Quip, Confluence). Use the platform that your team already uses for collaboration, as it will probably a familiar place for interaction. These platforms allow you to easily ask questions, let reviewers provide input, and collaboratively shape the material in a more interactive way.

There's no point in focusing on your authoring tool's syntax (e.g., Markdown, DITA, XML) or struggling with build issues at this point if you're not even sure if the pages you've written will make it into the published documentation. I wrote about this scenario extensively in Treat code like code and prose like prose ↗. While it might be tempting to drive everything from a docs-as-code tooling perspective here, asking engineers to review content using code review tools, this workflow typically excludes non-engineers from the documentation review process.

Although engineers might like providing comments through GitHub or other code review tools, usually product managers, field engineers, QA, or others will find these tools much less familiar and might forego participating in reviews altogether. This scenario is bad news for docs because you end up with the reviewers being the same people who created the systems you're documenting. You need more of an outside perspective to avoid problems related to the curse of knowledge.

Note that I'm only recommending writing in Google Docs, Quip, or Confluence as you're working on totally new pages with substantial amounts of content that needs to be heavily reviewed. If you're working on small tasks, it might make sense to make the changes directly in your authoring system from the start. Also, at some point, you'll need to migrate the content from the collaborative wiki into your authoring system as well, with some iterations thereafter. A good time to port it into your authoring system is after the content is no longer being actively reviewed and shaped in the wiki.

As you're creating content, you don't need to start from a blank page. You can use common templates to help get started and shape the material. You can find templates in the following places:

- Conceptual topics in API docs (p. 358)
- API reference tutorial (p. 118)
- The Good Docs Project ↗

Engineers and other product team members almost invariably have some product information on a wiki, but this information often needs to be reworked, sometimes from scratch. Avoid just lightly editing content that someone gives you, without assessing it on a deeper level.

One of your first steps is usually to get the product or code working (to some extent) yourself. This testing will usually help you understand the product and identify information gaps in the existing material. See Testing your docs (p. 343) for more details here.

Push for sample apps (p. 476) or sample code (p. 464) (often from a QA team) in order to test the code. Also, recognize that clarifying these information gaps is a primary value we add as technical communicators to the information process. If you don't understand something on a wiki that engineers wrote, find out whether your audience will understand it (often by asking field engineers or support teams to represent the customer's point of view).

9. Follow a review process for your docs

It can be tempting to prematurely consider docs done when an engineer gives you his or her stamp of approval. However, engineering approval is only part of the review. In general, as you develop content, follow these five stages of review:

1. Review with the doc team (e.g., yourself)
2. Review with the product team (e.g., engineers, PMs)
3. Review with field engineers and support group (e.g., solutions architects, evangelists, field engineers, support engineers)
4. Review with your Legal group (e.g., lawyers)

5. Review with early release partners (e.g., beta partners)

The general pattern is to increase the circle of reviewers wider and wider with each iteration. It doesn't make sense to blast everyone with the first draft. Instead, before you invite a larger number of reviewers (field engineers, support, other stakeholders), you want the draft to have at least been vetted by you and the product team first. Also, before you push content through your legal process, make sure it's not still changing.

For more details, see Processes for reviewing documentation (p. 764).

10. Create a code review for your changes

Let's say you've finished the content development in your collaborative platform, everyone has reviewed it (to the extent needed at this stage), and you are ready to integrate the content into your authoring system (e.g., Jekyll, Sphinx, Gatsby, Hugo).

When you're creating content in your authoring tool, if it's a docs-as-code system that uses Git, follow a Git workflow that makes sense for your team. For example, you might choose to align with a standard branching workflow when working with docs.

Usually, you create a feature branch from your production branch and then make changes in the feature branch. Consider following the general workflow described here: Git Feature Branch Workflow ↗.

When you're ready to review this version with reviewers, merge your feature branch into a beta environment, push out the build, and provide a link where reviewers can see the output.

If you're working in a docs-as-code environment, you should also create code reviews for the changes you're making. The code review can be especially helpful for writers or others who haven't been working with you closely in other spaces on the content.

For example, another writer might want to know what you're committing into the repository, or a software development manager might want to see the docs written for a feature. These people might not have been reviewing the docs in Google Docs, Quip, or Confluence earlier — maybe they only want to be looped in at the end. Code reviews provide diffs that show file changes at a granular level. They help everyone know specifically what you're changing.

Even if no one looks at your code reviews, I still recommend creating code reviews for docs for the sole purpose that they build visibility with engineering teams for the work you're doing. Each code review you create usually sends a notification (via some chatbot) to the engineering channel indicating that a new code review has been posted. This code review makes you more part of the team, more visible, and your tech credibility also increases.

You will also become more familiar with how engineering code review tools work, how to use and interpret the file diffs, the workflows of code review, and such. You can then begin to monitor changes that your engineering teams are making. By staying aware of code reviews, you can more easily track what's going on in your team. With most engineering teams, almost nothing gets pushed to production unless engineers push code into a code review system and others approve it. If you become intimately familiar with doing code reviews yourself for your docs, you will be more familiar with them when interfacing with code reviews for engineering changes.

You could ask other engineers to review and approve code reviews for docs, but if you've already gone through the review process elsewhere, there's not much point in asking for reviews again. The code review is mostly a notification that you've transitioned the content into your authoring system and published it.

11. Close the sprint

At the close of the sprint, the doc team re-groups and closes the doc sprint. For your sprint closure meeting, consider doing the following activities:

- Demo the work completed, perhaps talking through it at a high-level
- Engage in a retrospective discussion about what went well and what needs improvement

This meeting is a chance to re-evaluate the work and make any adjustments, course correct, get feedback, and more. During the demos, consider giving each writer 5-10 minutes to actually read through the completed work.

Items not completed during the sprint can either be reverted to the backlog or left as uncompleted in the sprint. (A single ticket can usually have more than one sprint label.)

Typically, writers embedded with product teams should also attend the Scrum meetings, demos, and retrospectives of that team. (And of course, if you're a lone writer, that product meeting would be your only sprint closure meeting.) If you want to show off your docs as part of the list of product team's demo, great. I find that it only makes sense to showcase the docs every so often when I've added something significant or noteworthy and want to call attention to it in a more visible way.

In many cases, engineers are specialized and often unaware of additions to the documentation. By occasionally providing a demo of the docs, you build awareness with engineers that you're writing and contributing docs, and they start to understand your role and what you do. (Yes, some engineers might not fully understand what your role as a technical writer is on the project!)

12. Share monthly reports

At a monthly cadence, the doc team shares details of what they completed with all stakeholders interested in their work (outside the engineering teams). This typically involves sending updates to an email distribution list and might include your own senior leadership, field engineers, support teams, and so on.

The monthly report contains a high-level summary of the latest doc updates as well as a list of completed tickets. For significant updates (e.g., launches), you can send info immediately to the list (or relevant stakeholders) rather than waiting for the end of the month.

Meetings summary

Here's an overview of the meetings that drive the process for managing documentation projects.

Meeting	Cadence
Doc Project Planning	Biweekly
Doc Project Prioritization	Biweekly
Doc Project Kickoff	After a project intake ticket is completed
Sprint Planning	Biweekly

Meeting	Cadence
Sprint Closure	Biweekly

Related topics

See the Process for managing small documentation requests (p. 750).

Managing small documentation requests

With any developer portal, you'll receive not only incoming documentation projects (p. 737) but also a healthy dose of small documentation requests too. These small requests can be managed in a similar way as documentation projects but can also be scaled down with a simpler workflow. Even so, small requests have unique challenges when it comes to prioritization, weighting, and cadence.

Definition of small documentation requests

Small requests by definition are tasks that exist outside of a more formal documentation project. Some examples of small requests might be to update an existing page, correct some misinformation, expand existing content with more details, review a contributor's updates, document a new option on a screen, publish an SDK minor version with release notes, and so on.

Small doc requests can usually be made in a few days or less. These tasks are submitted by requesters from various groups and usually relate to products that have already been released. These small tasks typically wouldn't be included in a list of documentation projects.

Assignment of small requests

Who handles these small doc requests? If the request originates in one of the general groups related to a tech writer's specialization, then typically that writer would automatically take on the request. For incoming requests outside of projects and divisions, the item can be assigned to an on-call writer assigned for the rotation that week, or to whomever has the most available bandwidth. This can be determined during your sprint planning.

With engineering teams, there are often engineers who have available bandwidth and will voluntarily take on additional requests. Sometimes with writing teams, fewer people willingly step up to take on requests. In that case, the on-call might be more appropriate.

Assigning the request to a sprint

During the sprint planning meeting, you review all the small documentation requests that have been submitted through your intake process and assign the items into the sprint as appropriate. In general, try to include a few of these items each week in addition to the project work. The small items can help provide balance with larger projects (like a break from some longer, more in-depth task you might be working on), and should be handled with some regularity even if not a major priority. For example, fixing a broken link might not align with a strategic priority for your organization, but it doesn't mean the task should be moved last on your to-do list.

Small doc requests can be prioritized and weighted similar to other projects (e.g., low, medium, or high priority). As described in documentation projects (p. 737), the weighting can be as follows:

- 1 point = 0.5 days work or less
- 2 points = 1 day of work
- 3 points = 1.5 days of work
- 4 points = 2 days of work

If there are a lot of small tickets that each takes 10 minutes, and you can't logically group them together, you can choose how you weight them. This where a more exact weighting system breaks down — it probably makes sense for disparate tasks to remain in separate tickets, but since many ticketing systems usually require whole number weightings, how do you represent a 10-minute task? You don't want to use 0 because the task requires *some* work, but if you use 1 point, wouldn't this distort the actual effort? Ten minutes of work is hardly the same as another task that takes half a day.

It depends on the purpose of the weighting. If the purpose of weighting is to evaluate the team's productivity, recognize that there can be a danger in using these rough point estimates to assess the amount of work completed. The way you chunk up tasks and weight them can easily be distorted. For example, suppose a writer breaks down tasks for a project in an extremely granular way, and then weights each task as 1 point. The writer might appear to be finishing a ton of work when in reality the writer just manipulated the tracking system to appear to be doing more work than was actually done. If your velocity and weighting can so easily be distorted, it might not mean a whole lot to others.

At the same time, best practices for chunking work and providing time estimates usually don't recommend such granular, hourly chunks of time. Perhaps one solution might be to attach a code review to every ticket so that others can see exactly how much text was changed. However, this could also short-sell the work. Suppose the edit required tracking down a SME who had more knowledge about a particular value or setting that needed clarification. Maybe the file diff is minimal but the effort to get the information was large. Conversely, suppose a writer receives a large chunk of code or other content from an engineer that is production-ready, and all the writer needs to do is paste it into an existing page.

I don't have a good solution for this situation. I think at the end of the day, if managers evaluate writers based on the points they tackle each week, writers will trend towards more granular chunking and points to show more work. As such, weighting and work assessments probably shouldn't be used to assess writer productivity. But if the weighting doesn't factor into any productivity metric, it probably won't be abused.

Intake template for small requests

As with the required fields in the documentation project intake, the intake for small requests should include required fields that the requester must populate. Here's a sample intake form for small documentation requests:

General details

- Page that needs to be updated:
- Section on the page that requires updates:
- Product owner:
- Business group:
- Related project:
- Team:
- Date update needs to be published:
- Related project:
- Category for this product:

Update details

Describe the update that needs to be made. Be as specific as possible:

Existing resources

- Does draft documentation already exist?
- Are there wiki pages related to the update?
- Any related design, business docs, or other information?
- Who should review the content?

You can see that these fields also ask for existing business documents, informational wiki pages, or related projects. This is because sometimes releases might have a lot of complexity on the backend but might not require much documentation work. It's not always easy to draw the line between documentation projects and small documentation requests based on whether the update originates from a larger project or not. This isn't an exact science. Requesters will probably use the intake form that makes more sense to them.

Think about the different kinds of doc requests you receive, and then customize an intake form for each type of document. For example, suppose you receive requests for these doc types:

- Legal documents
- Blog posts
- UX microcopy
- Screencasts
- Announcements

Create a custom intake form that solicits the information you need for each of these different doc types. Advertise your intake process on your team wiki page and point users to it when they ask for docs.

It's important that small doc request tickets be completed by the requester. This ensures that the tech writer isn't stuck digging around for the information needed to complete the task. It also ensures commitment and allows the requester to receive updates on the status of the ticket. The ticket provides a channel for communication not only for the requester but for anyone else following the ticket, and also provides a record of the work.

It's easy for someone to request an update via email or chat, and perhaps asking the requester to formalize the request by submitting a ticket might be extra overhead — wouldn't it be easier for you to just copy/paste the email thread into the ticket, rather than making a request to the person to locate your intake page, create a ticket, and then populate the right information in the right way?

Here is where the intake fields can do some work. Email threads have scattered information. As part of the formalization of the work, you usually organize the information in a more structured, logical way. Rather than doing the work of restructuring this information to specify what page needs to be updated, what updates need to be made, who should review the updates, when they should be published, etc., you can ask the requester to do this. This can reduce your own workload. When the task is made more formal and structured in a ticket, it will be easier for other writers who aren't part of the email thread to work on it. In fact, if you can specialize different intake forms for different types of requests, you can more acutely focus your questions around the information needed for that task.

Requests should come through product teams

In your small intake process, avoid allowing anyone to make a doc request. Instead, the ones making requests should be product managers (or those who own the products related to the docs).

To understand why, consider this scenario. Suppose your DevRel engineers (the ones who work closely with partners on implementations) make a doc request for more information about some feature. They don't have the information themselves, but they they've noticed that the lack of information is a partner friction, so they file a doc request. It sounds like a good idea, right, because you want to drive your doc priorities based on partner-facing groups? However, the problem is that DevRel groups (or others) usually don't have the information needed to make the request actionable.

If anyone can make a request for information they wish would be included in the docs, this can leave technical writers trying to chase down the information, knocking on the doors of different groups without getting much traction. This puts all the burden on technical writers for information collection and SME identification, and none on the requester. This is the wrong approach.

It’s much better to direct DevRel groups to make documentation requests to product teams. Then product teams should formally go through your intake process. This might seem like an extra hop, but it results in a world of difference. When the product team files a request for documentation help, you’ve got them at your disposal. They can provide the information that you actually need to write the documentation. The PM can allocate bandwidth to the collection and sharing of this information so you can complete the request. They will also review the docs.

Without this backing of the product team, the DevRel request places all the pressure on the technical writer to figure out who has the info, what the details of the info are, and more. In most organizations, the DevRel groups are much stronger than the documentation groups because DevRel groups work closely with key partners. Hence DevRel can place much greater pressure on product teams to act, while product teams might simply disregard requests from technical writers.

Indicating progress on the ticket

As tech writers work on the ticket and make progress, they communicate the status, questions, or other information in the ticket thread. This allows others who might not be part of email threads to see the ticket’s history and work.

In contrast, if you manage communication through email threads, you’re bound to lose track of key information, and those not included on the thread might not have all the context if added later. Email threads expand and contract based on the Cc list someone decides to use — sometimes adding people, or removing them, etc.

Suppose a thread goes a dozen rounds and multiple Cc lists. By the time the thread settles down, it’s much harder to trace back through an email thread’s history than a ticket’s history. In contrast, a ticket history shows details about dates for each response, profile information that lets readers quickly click names to see who’s who, and status information about the ticket.

Starting work

Most small doc requests can be made directly within your authoring system rather than starting out in a collaborative platform like Google Docs or Quip. When you’re creating content in your authoring system, you might choose to follow a standard branching workflow (or whatever Git workflow you’ve decided to implement in your group). Usually, create a branch from production and make your changes in the branch.

When you’re ready to review this version with reviewers, merge your feature branch into beta, push the build into a beta environment of some kind, and provide a link to reviewers. Also, create a code review so the requester can see specifically what you changed. I find that code reviews provide more granular detail if people want it, though I’m somewhat mixed about whether code reviews are useful for more extensive updates. I personally like that code reviews objectively demonstrate the changes made so that readers get a sense of the effort involved.

Closing the ticket

When you’re finished with the work, ask the requester to resolve the ticket. This indicates their acceptance of the work. Then make the change live by merging your feature branch into production. Afterward, delete the feature branch.

If the ticket doesn’t require any review (e.g., fixing a broken link), you can just push the content to prod and let the reviewer know.

Reporting on progress

In your monthly report listing doc work completed, consider listing all closed tickets. You might decide to put the small doc requests into a section at the end (e.g., “Small Doc Requests Resolved”) so that readers can focus on the important changes. If you simply list all tickets in order of resolution, it might juxtapose trivial changes alongside more substantial efforts and dilute your communication.

In the Small Doc Requests Resolved section, you might also provide a link to your intake form for these types of requests. This helps others know your process for managing small doc requests. The next time they have a request, they’ll know what the process is and what to expect.

Managing SDK releases

So far we've talked about processes for managing developer documentation projects (p. 737) and small doc requests (p. 750). When you're working in a developer portal, you might also be responsible for managing and distributing SDKs and other sample code.

Even if someone else distributes the SDKs, they might look to you for guidance and input on the Readme, the distribution process, and other aspects. The process here describes a few callouts that you should look for before publishing SDKs and other code artifacts.

Definition of an SDK

A software developer kit might include a language-specific library for your API (for example, a Java JAR library file or an Android AAR file that developers would integrate into their projects) along with a sample app, debugging tools, and other tools necessary for building software. I explored SDKs (software developer kits) in depth earlier in SDKs (software development kits) (p. 480) but covered the topic from a documentation perspective — what do you need to document when it comes to SDKs, what level of instruction is required, etc. The section here covers details about distributing and publishing the SDK.

Sample scenario with tech writers and SDK distribution

Let's assume that you've written the documentation for the SDK, and now the developers are ready to publish it. They've got binary assets, such as a JAR file, AAR file, or C++ libraries, along with sample apps and other code that they need to push out to users.

Since you're controlling the distribution mechanisms on the developer portal, they might send you the SDK zip file to publish along with other documentation assets you're already publishing. Or they might ask you to review or sign off on the SDK, including the release notes, readme, implementation instructions, license, and other details.

Your role with the SDK's release and distribution will differ by company, for sure, as well as how you're distributing it (S3 bucket, JCenter, Maven, GitHub, or other online repository). How do you review these code artifacts to make sure they're publish-ready?

The intake process

First, as with other requests, consider pointing developers to a specialized intake form for their SDK request. This request can require developers to provide information that will let you know if the SDK is ready to be published. The SDK intake ticket might have fields like this:

SDK release request

- Name of the SDK:
- Legal signoff: y/n | name
- Security signoff: y/n | name
- QA signoff: y/n | name
- Javadoc or Doxygen Reference Docs: y/n
- How-to documentation: y/n | link:
- Version information: y/n
- Release notes: y/n | link:
- Contents description: y/n | link:
- README file: y/n

Does the README contain the following:

- Overview of the SDK's functionality, at least several sentences long: y/n
- Link back to documentation on the developer portal: y/n
- Links to tutorials for implementing the SDK and any other important tasks: y/n
- Information about where to find support: y/n
- Details about the version: y/n
- Release notes or link to release notes: y/n

Is your binary available on JCenter or Maven? y/n

Requirements

Let's go into more detail about each of these requirements. Each of the following sections will expand on the points listed in the intake form.

Legal Signoff

In cases where your company has a Legal team, the Legal team's approval should be required to release any SDK, sample app, and any other code artifact. The Legal team can ensure that you are using the correct licenses for the software and that each page appropriately displays the license information and other details. They will assess whether the SDK includes any third-party libraries, including open-source libraries, and whether you have the rights to publish and distribute the SDK with these other libraries.

Legal might also look at trademark usage and names. For example, the sample code shouldn't reference other companies or trademarked names (e.g., Mickey Mouse) in examples or text, and your product's name should align with your company's branding guidelines and not conflict with other company product names. Legal might also check to ensure that you're not using any prohibited words, image assets, or copyrighted material (such as sample videos or thumbnails).

Security signoff

The SDK should also receive approval from security before it can be distributed. Security review can ensure that the SDK doesn't create any vulnerabilities, that it doesn't expose risks that can be exploited, and so on.

For example, your security group might have restrictions about using implicit grant authorizations or some other method for authorizing requests. They might review how the code handles usernames, passwords, and API keys. They might ensure that manifest files request the correct permissions from users before functions can be called in apps. They might look at how you're managing and storing customer information, and so on.

Security isn't an area you need to be familiar with yourself beyond a high-level — rather, the requirement here is to get signoff from a security expert before distributing an engineering team's SDK.

QA Signoff

The SDK should be tested through a formal QA process. Usually, before you receive a request to release an SDK, it's assumed that QA has signed off on the code. However, this isn't always the case, and if you're managing incoming assets for many different teams, inside and outside your immediate business unit, you might be working with different development teams with different standards. Asking to confirm QA signoff shouldn't pose any challenges to the request unless they don't have a QA process.

It's still a good idea to ensure QA signs off, though. Think about how many version updates SDKs often have in which the update addresses some bug or issue with the SDK. By asking for QA signoff, you can better identify QA groups and their processes.

If you want to take an extra step here, you can ask the QA team for a list of their validation tests (which they probably store in a QA test system like Rally), and then review the documentation against the test cases. Does the documentation cover the main features and tasks that QA is testing against? By reviewing the requirements against the documentation, you can assess whether the documentation is complete.

Javadoc or Doxygen Reference Docs

SDKs should include reference documentation in the form of a Javadoc (p. 707), Doxygen (p. 723), or other auto-generated reference documentation appropriate to the library. The reference documentation can be included in the download and can also be made available online as a link in the documentation. Delivering this is a standard and is usually added inside of a folder called "docs." Check to make sure that the reference documentation is correctly generated.

If you want to go a step further, you could verify that the annotations and tags in the source are correct. I covered Javadoc tags (p. 712) in the Java section of the course. You might look over the Javadoc and Doxygen files to evaluate whether all the elements have descriptions and other details.

How-to Documentation

Reference documentation alone is insufficient. The SDK should also have sufficient how-to documentation or other tutorials so that developers can implement the SDK or other code. In particular, the documentation should include the following:

- Implementation guidelines that explain how to incorporate the SDK into a project
- Explanations about how to perform common tasks that involve the SDK

Presumably, the main documentation covers this in detail. However, suppose you're playing a publisher role for a team and not working specifically with their documentation. Check the documentation to make sure you can identify the implementation tutorial. See if other tutorials address some of the classes/functions in the reference documentation. If the reference file has a lot of classes and other details not mentioned at all in the documentation, investigate whether the documentation is sufficient.

Readme file

The SDK needs to include a README file (typically named `README.md`) that provides the following:

- Overview of the SDK's functionality
- Link back to SDK's documentation on your developer portal (unless the documentation is entirely included with the SDK itself)
- Links to important SDK tutorials (such as implementation or configuration)
- Information about where to find support
- Version details
- Release notes link

There's a lot more one could say about Readme files. For example, browse this curated list of Readme files in the Awesome Readme repo. Also see Write the Readable README by Daniel Beck at the WTD 2016 conference.

In general, some Readme files might compress a lot of documentation into the Readme. The Readme might provide brief summaries and descriptions but mostly link to the main documentation for this information (e.g, installation, implementation, configuration, etc.).

Preparing this Readme might actually be the main request from the engineering team to technical writers.

Most developers expect the Readme to have detailed information, and it might make sense to pack your Readme with condensed versions of information that is also available in your docs, especially so that the repository has the right keywords for searches and other visibility on whatever site/platform you're distributing your code.

But to avoid duplicating information in multiple places online (between the Readme and your main documentation), as long as you provide summaries plus links of the expected information, this should satisfy developer needs. Keeping the bulk of your information in your main documentation (rather than duplicating it in the Readme) also makes it easier to update your docs (since presumably you can update your documentation pages dynamically but might require engineering involvement to update the Readme).

Version Information

The SDK should include the version number for the release. This version number lets users know whether their existing file is outdated, and lets you establish guidelines about compatibility based on different versions. The version number might be indicated both in the Readme file and code.

Version numbering should follow a consistent pattern for the organization (or at least for the project). As a general rule, minor releases are indicated by a dot version (e.g., 2.1); major releases by a whole number (e.g., 3.0). Sometimes teams will have internal releases that increment the version, so the actual released versions might skip a few versions. If so, you can note that the skipped versions were internal releases.

Release Notes

The release of any SDK should include release notes indicating what has changed. Release notes can indicate whether the release is backwards compatible or the extent of support/compatibility for intended devices for each version. For example, if the latest version creates breaking changes with previous versions, this needs to be called out in the release notes. If the latest version is compatible with all previous versions (a best practice), you might note this as well.

What exactly does backwards compatibility mean? Suppose version 2.1 of an SDK has a class called ACME that developers have implemented in their projects. Version 3 presumably wouldn't change ACME in a way that would break the code that developers implemented with version 2.1. If Version 3 does break previous implementations, this is sometimes called a "breaking change." Breaking changes should be called out prominently in release notes. The release notes should indicate how to handle the upgrade (for example, maybe you'll need to flush the cache of existing implementations).

Release notes usually indicate both a version number and a date. Exactly how version numbers and dates are highlighted in release notes varies from product to product. Follow a consistent and readable pattern. For example, consider putting the release version as a heading 2 tag with the date as a subtitle below the heading.

Contents description

The documentation for the SDK should provide a description of contents in the SDK. For example, if the SDK includes a library, sample app, debugging tools, and other files, these should be itemized and described in the documentation so that users know what is included in this kit of software development tools. This practice is similar to purchasing any product that you assemble — the first page of documentation usually has a list of parts included in the product.

If you're receiving the SDK from an engineering team and not actively working on their documentation, review their documentation to identify where they itemize the contents of the SDK.

Hosting Java and Android on JCenter, Maven Central

Now that we've covered requirements about the SDK content itself, the next question concerns hosting. Where do you host your SDK files so that developers can download them?

Your organization might already have a file storage and distribution in place (such as storing the files on S3 or some other web server). However, SDKs tend to have some common distribution sites that you should be aware of.

For Java, the SDK will usually include a JAR file; for Android, the SDK often includes an AAR file. C++ might include a DLL. These are binaries that developers will download and incorporate into their projects. Rather than manually downloading the files, developers can automatically pull from online repositories to dynamically include the files when they build their projects.

Two common distribution sites for Java-related projects are JCenter and Maven Central. JCenter is an online repository for Java and Android libraries. JCenter has everything that Maven Central has and more.

Code for other libraries will have different online repositories. For example, with Jekyll, the jekyll gem is distributed through RubyGems, a gem hosting service.

When developers build their projects, they usually have a build tool that retrieves needed resources from online repositories and pulls them directly into the project. Android uses Gradle to manage dependencies. For example, with Android, to declare a dependency on an Android library in JCenter, you list `jcenter()` in your list of project dependencies, and then include the package name and version you want to pull. Now Gradle will handle getting that dependency automatically into your project, without requiring developers to download it. You can read more here: Dependency management in Gradle.

Jekyll projects use Bundler (a package and dependency manager) to automatically retrieve the right gem versions as declared by the projects. When you run `bundle install` on a Jekyll project, Bundle ensures that your project has the gems listed in your project's Gemfile.

GitHub

Many SDKs have a sample app (p. 476) that demonstrates integration with the SDK and the provided functionality, even if the sample app is just bare-bones simple and only includes some basic buttons in the UI to initiate events. Many times, sample apps are included with the SDK. If the sample app is more extensive and stands alone, consider distributing the sample app through GitHub as open-source project instead.

Releasing your project on GitHub doesn't require the code to be open source, but the project source must be viewable. (You can't just upload a compiled binary onto GitHub.) You probably can't freely create a new GitHub repo on your own in your organization, but you can instruct engineering teams to do so following any outlined corporate processes. Many times engineering groups have organizations and teams already set up in GitHub and standard processes for pushing out code.

Keep in mind that distributing code on GitHub entails some support for the code. For example, if users log issues about bugs, who will respond? Will it be important to track the number of stars and follows for the repo? Who will be the one who actually pushes the code into the repo? You want engineering teams to own this process not because it's technically difficult to push code to GitHub, but because you want them to feel a sense of ownership and commitment for the code there. If the tech writers handle all aspects of the GitHub repo, tech writers end up playing support roles in responding to issues and handling other administrative details. The flip side of managing the GitHub repo, though, is that engineering teams are less likely to push out changes without going through you.

Documentation kickoff meetings and product demos

At some point after receiving a new documentation project, the first step in the project is to hold a documentation kickoff meeting and product demo. These meetings are mostly about gathering information so you can create the documentation. The following are some initial questions and topics for these meetings.

Meeting agendas

The general purpose of this documentation kickoff meeting is to review the intake ticket, which should include a host of detail about the project. (See Create project plans for the documentation projects (p. 740) in Managing documentation projects for a list of fields.) Chances are, a product manager has completed half or fewer of the intake fields, as they just don't know all the possible information. No worries, though, as you'll be going through this information in more detail here.

After reviewing the ticket, you want to go over the product or features that you'll be documenting. If this is a large project, you might have a separate meeting for the product demo. If it's smaller, you can combine them. Obviously, adjust the advice here for the nature of the product, product teams, and company. During the demo, consider the following questions as a way to gather more information about the product or feature.

Project overview

First, get some basic information. At a high-level, what's the project here? What's the scope? Complexity? Timelines? Is it a new product, or are you just adding a feature to an existing product? What doc set are you modifying? What team will you be working with? Is this external or internal documentation? What format will the documentation be in, and how will you deliver it?

Product overview

After gathering some high-level project details, dive into more detail about the product.

Product overview

- What does this product do?
- What problem does it solve?
- Is this an API? Is there reference documentation?

Use cases

- When would you use this?
- Larger picture scenario?
- Common use cases?

Target users

- Who are the intended users?
- How do you know they actually want this feature?
- Do you have any voice-of-the-developer type feedback from users about this feature?
- What roles do users have? Are they engineers? What type of engineers are they?
- What's the user's technical level?

- Are users familiar with certain programming languages or systems?

Technical prerequisites

- What do you need to install or set up before you can run this product?
- Do you assume they already have an app? Is there a starter app or toolkit?
- Are there any required permission groups?
- Are there language prerequisites (e.g., familiarity with Android)? What language is this? For example, is this for Java apps? Is this an agnostic REST API? Are there SDKs for any specific languages?
- Are there device prerequisites? 32-bit? 64-bit? Mac/PC/Linux? Android/iOS?

Background knowledge prerequisites

- What do you expect the user to know? For example, do you assume proficiency with any systems or frameworks? What framework? What specific areas?
- Do you assume they understand any existing tools or processes?

Development effort

- How long do you anticipate that the implementation will be?
- From a developer's perspective, what's the scope and complexity of the integration?
- How hard is this going to be for devs? On a scale of 1 to 10, what's the difficulty level here? And how do you know?

Components involved

- What are all the components involved in this situation?
- What are the relationships between each of the components?
- How do each of the components influence each other? What actions exist between the components?

Tasks Overview

- What's the larger workflow of how to use the various classes in an end-to-end workflow (initialization, termination, etc.)
- What is the overall user journey here?
- What are the main tasks or core functions (e.g., classes) people will be using?
- What fields, methods, and returns does each class have?

Setup and initialization tasks

- Do you have to declare certain permissions or initialize something, or add a library to your project?
- Do you need to declare dependencies? Does the project include dependencies? How are they retrieved?
- Does the user need to request permissions? What does a manifest look like for the project?
- Is there an initialization process?

Sample usage

- Can you show the usage for one API call so that users can extrapolate how to work with other APIs?
- Can you talk through a code sample of how it works – e.g., a representative example from the API?
- Are there constructors to initialize a class or other ways to initiate processes?

Common tasks

- What are some common tasks users would perform?
- Can you provide a walkthrough of some common tasks in an end-to-end way?

Reference docs

The essence of an API is inputs and outputs. Something goes in, and something comes out. Is this input and output already described in reference documentation? No matter the language, always focus on inputs and outputs to understand an API.

Overview

- In general, what does this code do?
- What are the inputs and outputs?

Inputs

- What parameters does this take in?
- What are the data types for each of the parameters?
- What max/min values do the parameters have?
- What are sample values for the parameters?
- Are each of the parameters required?
- Are there callbacks?
- How often can developers call the functions?

Outputs

- What does the object do with the inputs? Is there an algorithm that does something?
- Does the speed of the function's algorithm increase as the size of the input increases? (Big O notation)
- What gets returned to the caller?
- How do you access the output?
- Do you have to page through the results?

Best practices

- Any general best practices to be aware of?
- What are some best practices for efficient inputs? Does processing time increase as size of parameter inputs increase? Would some inputs cause this function to run slowly? How can a developer make this run as fast as possible?
- What are some best practices for reducing memory? Are there best practices for making this run as efficient as possible? Do you have to release resources, destroy objects, remove heaps, or do some other memory freeing of resources?
- How frequently can you call the API? Are there access limits?
- What gotchas or quirks should developers be aware of?

Error messages and handling

- What error messages might users see?
- Are there error scenarios to be aware of?
- What happens if you input invalid data?
- What happens if you exceed your rate or number of allowed calls? throttled?

Testing

- Are there any test classes or tests you can do to make sure that it works?
- Can they test out their code before submitting their app or something?

Sample app

- What is QA using to test this code? Can you use this as a sample app?
- What would you need to run this yourself? Can you test it in a sample app?
- Is there a demo of a sample app?

Release notes

Details of release

- What’s being released?
- What are the version numbers, dates etc., of the release?

Known limitations

- What known limitations or issues exist?

Processes for reviewing documentation

Conducting a successful documentation review is challenging, especially with developer docs because the content is often highly technical and requires a lot of engineering input and review. At the same time, getting this engineering input and review doesn't come easy. In this topic, I'll outline a tactical approach to conducting doc reviews for large amounts of content.

How to get reviewers to read long content

Getting people to review short amounts of content (one topic or less) isn't so challenging. What's challenging is getting engineering types to review dozens of pages. A recent project I worked on had 75 new pages. How exactly do you get engineers, PMs, and others to read and review that many pages, especially in a short amount of time (a few weeks before release)? Long-form reading of tech docs is not usually a characteristic of many people in tech. After 20 minutes, most people want to get back to work. Few will spend all afternoon going through your docs to provide a detailed review.

This puts technical writers in a bind. You end up in a situation where you're highly dependent on the review and input of others (because the content is so technical or complex), but getting this input is increasingly hard because so few have the patience to read in a careful, meticulous way.

In Processes for managing large documentation projects (p. 737), I already outlined five general stages of document review:

1. Review with doc team
2. Review with product team
3. Review with field engineers
4. Review with legal, support, and other stakeholders
5. Review with beta partners

Previously, I didn't go into detail about more granular processes or the tactical how-to detailing ways to approach these groups, what feedback tools to use, how to prompt action, and so on. In this topic, I'll dive into these details in a practical, tried-and-true way. Remember that these are tips from a practitioner, based on actual experiences working as a technical writer in the real world doing doc reviews. This approach has worked for me, but also note that each company has a different culture and rhythm, so decide what might work for you in your situation.

Meeting 1: Outline review

In most writing classes, you learn to distinguish between higher-order concerns (e.g., organization, story) before lower-order concerns (e.g., grammar, word choice). The review process uses the same tiered approach. With long content, you want to first be sure that you have the big pieces in place. Draw up an outline of the steps and get the product team to agree on these large pieces and their order.

For example, suppose you identify about 12 steps for implementing a product, and the bulk of the docs are related to this implementation. Get buy-in and approval from the product team for the steps. If you have mini-outlines for each step, even better. In this outline review, the product team doesn't need to actually read the content. You can talk through the steps and explain the outline at a high level. Once you've nailed down these large pieces, start drafting out the details in each topic.

In this phase of the document review, you're drawing a *map* of the instructions. You're not yet blazing the trail (writing the content). For tips about creating workflow maps in docs and why they're important, see Principle 1: Let users switch between macro and micro views.

Meeting 2: Questions review

As you start drafting the content, no doubt you'll have many questions about areas you're unsure about. Especially if you've been trying to make the product work for yourself, you should have many questions about issues, unclear points, and other details that you need clarity about. List out about 20 of these questions in a collaborative document (such as Salesforce Quip or Google Docs), and then set up a meeting to ask these questions to the product team. People love to be asked questions, and having a list focuses the meeting on a specific agenda. Again, at this point, you haven't asked the product team to review any documentation. You're just asking them to answer questions.

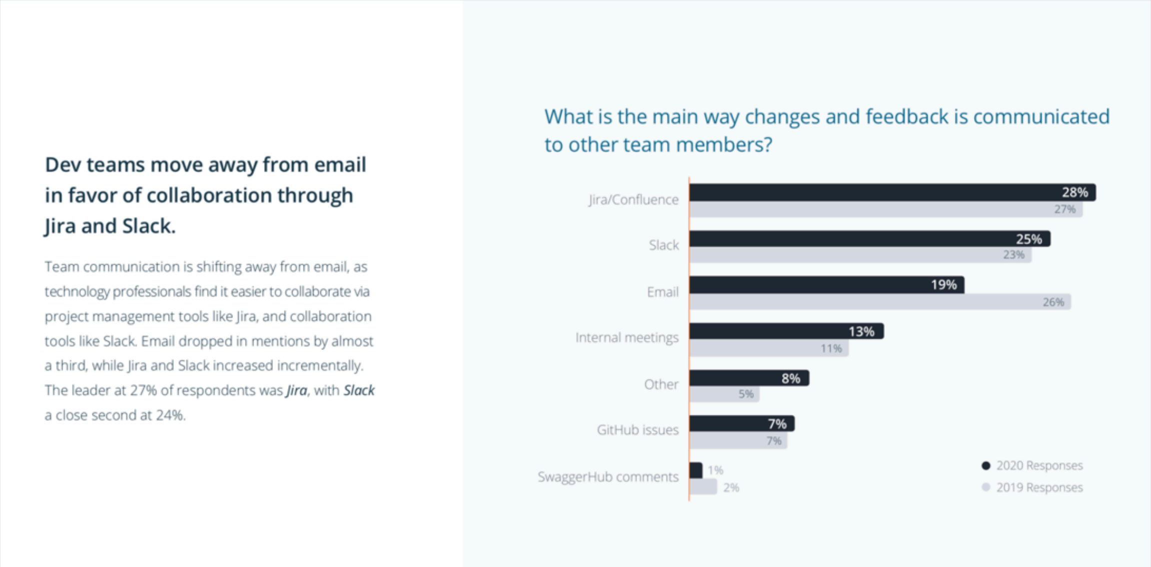
See my article and video titled A tip for doc reviews – bring a list of questions for more details here.

Create a Slack channel

Batching up your questions for a meeting is great, but you will likely have many questions over the life of the project *after* the meeting. Also, you'll find that during the meeting, reviewers will have some of the answers, but not all. Maybe questions you asked the product manager and engineering lead during the meeting drew blanks and shrugged shoulders for responses, while they indicated that some other person (e.g., "Sam" or "Sally") might know. Any sizable project probably has 20+ people working on it, each with different perspectives and specialities. You can't round them all up each time you have a question.

If your organization has Slack, use it. Create a Slack channel specific to documentation for the project (e.g., acme-tech-docs) and invite people to it. If you ask someone a question they don't have the answer to, it's easy for the person to tag another person for the answer, adding them to the channel. Having a dynamic channel like this to ask questions can be incredibly helpful and keep everyone informed about the documentation status.

Also, note that Slack is trending as a way for teams to communicate. The SmartBear 2020 API survey found that JIRA and Slack (not email and meetings) are how engineering teams prefer to communicate feedback:



Dev teams move away from email in favor of collaboration through Jira and Slack.

Team communication is shifting away from email, as technology professionals find it easier to collaborate via project management tools like Jira, and collaboration tools like Slack. Email dropped in mentions by almost a third, while Jira and Slack increased incrementally. The leader at 27% of respondents was ***Jira***, with ***Slack*** a close second at 24%.

You can read more insights about SmartBear's 2020 trends in this post: SmartBear's 2020 API report finds 'Accurate and detailed documentation' to be second-most important characteristic of APIs.

Meeting 3: Doc review with the product team

At some point during the content development process, you will have finished a first draft of the documentation. Reaching this point is a huge milestone, so pat yourself on the back. When you finish this first draft, it's time to formally review the material with the product team to make sure it's accurate at a foundational level. You're essentially walking them down the trail you blazed.

The documentation might not be entirely usable or understandable for outsiders to the team (the trail might have many rough spots), but at this point you want to ensure it's accurate and complete from the product team's perspective.

It's helpful to set up a meeting to review this first draft. One technique I've learned at Amazon is to dedicate part of the meeting to actually reading through the doc. At Amazon, it's normal to start meetings by distributing a document that everyone reads for 20 minutes. In Jeff Bezos: This is the 'smartest thing we ever did' at Amazon, the author explains:

> Jeff Bezos has a nontraditional management style at Amazon, and he says Amazon's unique twist on meeting structure is the "smartest thing we ever did."
>
> "Many, many years ago, we outlawed PowerPoint presentations at Amazon," Bezos said at the Bush Center's Forum on Leadership in 2018. "And it's probably the smartest thing we ever did."
>
> To replace the PowerPoint presentations, Bezos created a new way to hold meetings: Meetings start with each attendee sitting and silently reading a "six-page, narratively-structured memo" for about the first 30 minutes of the meeting.
>
> "[The memo is] supposed to create the context for what will then be a good discussion," Bezos said.

Although this passage describes business documents ("six-pagers" as they're called), the same document review culture exists for technical documentation as well. People don't often feel expected to read documents outside of meetings. It's not the corporate culture. If you want someone to review something, you set up a meeting and give them time to read the document.

Part of me dislikes this approach because it suggests that people are too lazy/busy to read anything on their own time, and you're limited to what people can read in 20 minutes (6-10 pages, more or less). But if you're struggling to get people to read something, this approach works (for about 6 pages). If you have 75 pages to review, you could break up the document review into a series of meetings.

After people finish 20 or so minutes of reading, devote the rest of the meeting to collecting their feedback. You can start with any overall comments and then proceed page by page.

The more executive the reviewer, the more they tend to want to control the meeting and discussion. If I notice this alpha behavior, I usually let the person lead out the discussion rather than forcing them down a predefined path of review topics — assuming the discussion stays focused on documentation.

If you have so many reviewers in different time zones and with different schedules that a focused meeting for the doc review isn't feasible, or if the documentation is too long to review in one sitting, and you dread setting up 5 meetings to review it, you can encourage the review outside of a meeting. However, you need to set a deadline for collecting feedback. There must be a due date to kick people into action. About the worst thing you can do is send a blanket email to a group asking them to review a lengthy document, without any due dates or commitments to follow up. In Outlook, create an artificial meeting that simply has a due date associated with it as a reminder.

Meeting 4: Review with field engineers

After the product team has reviewed and approved the documentation, incorporate the changes and widen the circle to the next level of reviewers: the field engineers. The "field engineer" role varies from company to company, but if you're working in developer docs, you probably know which role interfaces with the partners or third-party developers. Who can represent the partner/customer? Who works with partners/customers on a regular basis to implement the company's products? Maybe it's an evangelist, a sales engineer, solutions architect, customer experience, technical integrator, etc. Find these people (you should probably already be working with them) and set up a doc review meeting.

In the doc review, you can start by talking through the documentation at a high-level. Then follow the same pattern as before with dedicated meetings to read and review the documentation. Or if it's not feasible to read the documentation during the meeting, assign them the review as homework with a follow-up due date for feedback. If these field engineers will be guiding partners with this documentation, they are intrinsically motivated to make sure the docs are accurate, clear, and complete. Otherwise, customers/partners will ping them with questions and issues.

To collect feedback from field engineers, try putting your docs on the same collaborative platform for collecting feedback that your company has already established. For example, in many companies, teams use Salesforce Quip or Google Docs as collaboration tools. Both are highly similar, as these tools allow you to annotate text and make comments in the margins, and then reply to the comments. Commenters get notified about replies, and so on. Collaborative tools invite more of a discussion around content, not just a static reading experience. If you can write and edit your docs in a collaborative space, this is ideal.

However, suppose your docs aren't already in a collaborative space (e.g., maybe they're already in your authoring system because the project involves a high degree of integration that isn't feasible to do last minute by copying and pasting from Quip or Google Docs). In this case, you could create a blank page in Quip or Google Docs and invite people to list out questions and issues there, with their initials before their comment. This works well because many times comments apply to the documentation as a whole, or are topics not answered in the documentation.

When a reviewer adds a question, if you follow up with a response, the reviewer will likely add more questions — they learn that you're listening to their feedback. The reverse is also true. If you don't respond to questions or comments, reviewers stop leaving them.

If your workplace has another common practice for review, follow it. For example, maybe it's common that people use track changes in Microsoft Word documents, passing them from one team member to another in a baton-like way. Or maybe everyone uses code review tools to handle comments and doc reviews. Identify the common doc review culture and toolset at your company and plug into it. You'll have the most success that way. Many documentation systems might have special reviewing features, but if you require people to learn new tools, or worse, to log in to unfamiliar third-party systems, you might not get many people reviewing your docs.

See my post A simple way to write, edit, and publish documentation online using Google Docs and Markdown ↗ for stories about the success I had using Google Docs to review content at a former company. See also Matching documentation review practices to company culture ↗.

Review with support and legal

After incorporating the edits from the field engineering review, widen the circle even further to include your support teams, legal team, and any other stakeholders interested in the doc. At this level, the reviewers might not have substantial insights here or technical expertise with the product, but they might need to be aware of it (especially if support will be handling cases about it), and your legal team might need to be involved with any code distribution (sample apps, SDKs, client libraries, etc.). Legal mostly gets involved with your SDK releases (p. 755). You don't need to create a meeting for this review, usually. Instead, you can just send these groups a link to the docs and request feedback asynchronously.

At this point, one challenge you might run into is how to deal with all the questions and issues people bring up during the review process. Suppose you end up with dozens and dozens of questions about every possible scenario and fringe case. Adding them into the documentation might make the docs long and verbose. You don't want to balloon your docs with information that applies to only a small number of cases or partners. Every extra sentence you add dilutes the other sentences.

In these cases, if the question and answer doesn't logically fit in with the core flow of documentation, consider adding an "Additional Questions" topic (or "FAQ," if you want to erroneously call it that). Or you can add a section called "Additional information" at the end of the relevant topic. This can be a place where you at least address a question, even if you don't give more space to it in the regular documentation.

Final signoff

By this time, you should have amassed a significant amount reviewer feedback, and your doc should have gone through multiple iterations and edits. Hopefully, your documentation is nearly ready for release. Now you're ready for the signoff stage. Identify about five key people who you want to formally sign off on the docs. This might be the product manager, the engineering lead, the manager of field engineers, your legal representative, and the support director. Create a document (maybe a Quip or Google Doc) with their names and an option to select *Approve*. Send them an email asking for their formal signoff prior to publishing the docs.

This is important: Do not publish the docs until you get their signoff. As a technical writer, you don't have much leverage to force people to review docs (nevertheless, you'll be held responsible for any errors or inaccurate information). Your one piece of leverage is to not click the publish button. Especially if you're working on a new release, many teams are eagerly looking forward to release day. By requiring signoff before you publish the docs, you can force teams to review the content and assume some responsibility. After you click publish, if you failed to get a comprehensive review from key people, they will not likely be incentivized to review documentation post-release. Lack of documentation might not hold up a software release at many companies, but most product teams require documentation as part of their product release. Use your leverage as needed.

Post-release doc reviews

After release, you're still not done with the documentation. You might have entered a beta period, where beta partners are trying out the docs. Or maybe the docs are entirely public and generally available. Pay attention to the first users and other external feedback. Incorporate a feedback button that allows users to reach out with comments. Also, monitor support channels and forums for feedback.

However, at this stage, you will likely have identified the most significant questions and issues already and addressed them. I've found that the best doc reviews happen prior to release, not after. If users have issues, it's usually with the actual product rather than the documentation. You can't fix a poorly designed product with great docs — this is why some tech writers tend to gravitate toward product design. (For more on that direction, see Playing a product design role as a content designer – podcast with Jonathon Colman ↗.)

If you have tips for doc reviews, feel free to add them in the comments below.

Maintaining existing documentation

As soon as new docs are published, docs begin a trajectory of decay. The natural progression of technology makes documentation outdated within a matter of months or years. New versions of web browsers, operating systems, supporting utilities and tools, etc., are released, and the whole technology landscape keeps moving forward, evolving, improving, and adjusting — all while documentation remains static. The more your documentation relies on third-party components, the faster it goes out of date.

Most documentation efforts focus on creating *new* documentation, but what happens to all the *existing* documentation that is decaying? In this section, I cover ways to maintain existing documentation and prevent it from rotting.

Working on new docs versus maintaining existing docs

If documentation isn't actively maintained, it rots (becomes Redundant, Outdated, Trivial), loses ownership, and turns into "legacy documentation." The tech writers who created the initial content transition elsewhere, and the product team members move on to other areas of the org or leave. Soon no one knows the history of the content — it was just there when writers arrived. As such, tech writers new to the team might prefer to avoid and neglect it, being unfamiliar with the docs. The neglected docs begin a downward spiral of decay as no one actively looks after the docs anymore. The docs are like orphaned children, without a caregiving parent.

As more time passes, the amount of legacy documentation becomes more and more abundant. The rate of new documentation that tech writers create versus the amount of documentation the tech writers retire tends to be an uneven ratio: you create more than you retire. Most documentation requests that come in from product teams relate to new documentation, as everyone is focused on the shiny and new, the possibilities of a new feature, the excitement of a release, and so forth. Few want to take responsibility for documentation related to products or features that fizzled but are still used.

It's easy to get caught up in constantly creating new content. After all, this is what requests to the documentation team usually consist of — product managers say, *We're releasing a new product/feature, and we need docs!* Product teams rarely reach out to you to review two-year-old docs, or to update docs against support cases to see if better docs could have prevented the support efforts. Once you publish your docs, major content updates or overhauls are rare except with major product updates.

Updating and maintaining existing content isn't sexy or exciting, but it's essential if you want to stay on top of your docs. When you look at your documentation repository and see hundreds of docs that look unfamiliar to you, that you can't vouch for (identifying owners, the last time it was reviewed, who worked on it, how accurate it is, etc.), this is unsettling.

Rather than stay heads down in always creating new content, doc teams should regularly cycle in a few maintenance tickets with each sprint — even though no product teams request the maintenance. Regular maintenance should be part of any content strategy to maintain a healthy set of documentation. But with hundreds of pages in your repository, where do you begin? How do you approach maintenance in any sane way, especially when you can barely keep ahead of the incoming requests for new content? The following sections list possible maintenance activities you can do with docs. (Although I numbered the sections, they are in no particular order.)

1. Define priority docs

First, identify which docs are a priority. If some products have been sunset or abandoned internally, don't waste too much time improving those docs. Just establishing which docs are a priority can help you with planning. You can identify which docs are a priority by looking at page views in your analytics. Additionally, you should have a sense of your organization's priorities that are independent of site traffic. When you know which docs are a priority, you can weight them higher in the maintenance tasks. Try to identify your top 5 documentation sets. This way, if other efforts have to be triaged, you can at least focus your attention where it matters.

2. Identify broken links

Identifying broken links is a time-honored tradition in the doc community, and it's an easy place to start. Fixing broken links is a straightforward, concrete task that doesn't require outreach to other teams. Put on your favorite Netflix show in the background, and slip into a mind-numbing afternoon of fixing links. There are different tools and scripts for identifying broken links. For example, you could use linking-checking tools (like Dr. Link Checker ↗), custom scripts, or even Chrome extensions (like Broken Link Checker ↗) to identify broken links.

We often look within our own documentation repositories for errors, but what about content outside of the documentation? How about all the marketing pages, blog posts, support threads, and other content that link to your documentation? If your documentation portal has a lot of other pages, consider running link checkers across those pages to identify broken links. Fortunately, you might not have to fix broken links on these pages. You could simply print the report and hand it off to the content owners in these areas.

3. Identify broken or outdated images

As with broken links, identifying broken images is another task that you can tackle by running scripts or other validators. Broken images are much less common than broken links. More often, the issue is that existing images become outdated. If you're familiar with all the interfaces, you can glance through screens and identify older designs. Alternatively, you can revisit workflows, retracing your steps through tasks to see if the screens have changed. This would give you a chance to check the accuracy of the steps as well. There's nothing that signals "this documentation is old" than having outdated screenshots.

4. Delete unused branches

How often have you looked at your list of branches in origin and found half a dozen or more old branches or branches you no longer recognize? Long-lived branches outside of your production environment are rarely a good thing, as the branch usually becomes more and more divergent. If you've already merged the branch, get rid of it. If the content is unpublished, consider merging it anyway (assuming it's a new page and doesn't update existing pages), and then use other techniques to enforce access control. You could even set the page as unpublished.

There might be other maintenance tasks to perform on your Git repository as well, such as pruning large files to reduce the repo's size, verifying permissions, or running other maintenance tasks.

5. Identify unpublished files

In our documentation repo, there are quite a few files that I've started and then abandoned. Sometimes I started working on a file only to have the product team ghost me before launch. Other times I might have set about an idea that I eventually forgot about. In many cases, I set the page as unpublished and move on. However, when I do searches in the repository to look for phrases or instances of something, those unpublished files get included and slow me down. Look through your repo and identify all the unpublished content. Assess whether it still needs to remain. If you don't anticipate actually working on the content, delete it. Or copy it into a ticket and archive it. Someone can always find and retrieve it.

6. Identify files or doc sets ready for deletion

You might have some files in your repo that you can just delete. For example, suppose you deprecated a product two years ago, and all the remaining files just have a banner indicating the deprecation. At some point, you can just delete the files. Or maybe you have files and doc sets related to products that never launched. Maybe docs were written for a feature that was canned at the last moment and the product team long since disbanded. Or maybe you have leftover files from a previous migration, or old scripts that are no longer used, and such. At a regular cadence, look through your docs and ask what you can remove.

Note that I'm not recommending that you delete actively used content. Almost always, when you deprecate documentation, you add a banner announcing the deprecation and include a migration path. I'm instead encouraging you to think about the "deletion" part of the content life cycle. In Kristina Halvorson's definition of content strategy, Halvorson says, "Content strategy guides decisions about content throughout its lifecycle, from discovery to deletion." Think about that last stage in the content life cycle: *deletion*.

Deleting content can be scary — *what if people complain about broken links? What if it turns out we need that content? Before removing it, shouldn't we get approval from all the product team members (who are no longer there)?*

You know what? The great thing about managing content in Git is that it's nearly always retrievable. You can usually resurrect content that you've deleted. If you delete something and no one complains about its absence, you probably made a good choice.

7. Identify content owners for each doc set

Understanding who owns all the documentation content, especially legacy documentation, is a huge task. In the moment that you're creating new documentation, this kind of content ownership metadata hardly seems necessary. But as soon as the docs are published, teams evolve, people leave, and before too long, support might reach out to you with questions about a specific doc, and you won't have any idea who to contact about it. Content ownership usually doesn't need to be granular at the page-level, but the general doc sets should have some metadata about which team owns the product. Team ownership of products doesn't change so often as the team members. (The average lifespan of an employee at many companies is less than 2 years.) The engineers and PMs that seem so present and thriving one month might vanish the next.

8. Identify last-reviewed dates

Probably the most significant metadata for your docs is the last-reviewed date. Although Git timestamps your edits, Git won't distinguish between a single-character update and a major doc overhaul that was carefully reviewed by SMEs. In your page metadata, consider maintaining two fields: "last modified" and "last reviewed." *Last modified* is helpful to quickly see who recently worked on the file. You can implement a Git hook to print the Git user email into a value on the page, as well as the last modified date. The "last reviewed" metadata can't easily be automated, but even if you have a general date like Q1 2020, Q2 2020, and so on, it will probably be fine.

After you add this *last reviewed* date to your docs, consider taking the bold step of printing the timestamp on each page (perhaps in the footer). If users see that the page hasn't been updated since 2016, it can be a red flag to users that perhaps the content is old. You can create scripts and reports and automate other processes as long as you have these dates on your files.

9. Require content owners to periodically review docs

Probably the most powerhouse technique you can do to keep docs updated is identify docs that haven't been reviewed for a long time and submit a ticket to the product team to review the content. As long as your "last reviewed" dates are accurate, you could create scripts to identify these pages at a regular cadence (e.g., yearly) and then automate notifications or tickets to content owners to complete reviews. This regular "baselining" is common with security teams and permission settings.

However, it would be easy for engineers or other reviewers, no doubt busy with other projects, to half-heartedly click through automated review notifications to confirm that the content is up-to-date. In contrast, it's much harder to ignore a human. A genuine outreach to a team to review their docs, followed by a meeting where you actually step page by page through content that the team owns, might be the only way to get attention from engineers.

I admit that, although I've had intentions of doing this for years, it's never been something I've actually implemented. I'm usually too busy creating new content, and why bother to spend time fixing content that no one is complaining about? Aren't there more efficient starting points for identifying friction points, such as looking at support logs? Sure, there's validity to this argument, but I don't think these doc reviews need to be extensive. Product owners can quickly glance through doc headings and flag whether features are still valid, even if they don't read through each individual step. Also, this meeting will help establish contacts with product teams so they know who to reach out to when they do have updates. You could consider this a training and awareness campaign (assuming you want teams to reach out to you).

10. Identify docs that are missing metadata

Another maintenance activity is to look through all the metadata in your pages to make sure each page is properly tagged. Your metadata might require fields like this:

- Title
- Description
- Keywords
- Last modified
- Last reviewed
- Content owner

Unless you're working with a tool that enforces structured metadata, I find that maintaining a long list of custom fields in the header is a real challenge. (At some point, I plan to implement a wrapper like Netlify CMS on my docs so that I'm forced to choose metadata from a structured list, but I haven't yet.)

If you're working in a static site generator, you can easily generate a list of all the metadata for each page in your docs. Although keywords might not be important for Google's search, many times your site search uses keywords. Audit this metadata as a way of performing routine doc repository maintenance.

11. Identify top articles in knowledge bases that aren't in the documentation

Unless you work closely with your support team, the support agents might create KB articles related to trending topics without informing the doc team about these trends. (I find this lack of communication kind of mind-blowing, but silos seem to be the norm in most places.) Glance through the top KB articles in your support site or system and see if some of the information should be added to your documentation. This analysis might highlight content gaps that you previously missed.

12. Identify docs with the highest bounce rate and shortest visit time

Bounce rate refers to the number of visitors that land on a page and then bounce off your site domain to some other site, without visiting any other pages on your site. If users land on a page and then immediately bounce off, it could indicate that the content has an SEO-optimized title but lacks the answer users are looking for. Look at your top 10 pages with the highest bounce rate to see if the information looks right. Maybe some of these pages need to be re-titled, or maybe the information needs more development. Or maybe the page is actually working well and allowing users to zero-in on exactly what they're looking for.

13. Review doc commit logs to check commits from people you don't recognize

If you share permissions to your doc repo with a large number of people, you might want to periodically review your commit logs to see what people are committing. Especially if you have many engineers or people in other roles writing and publishing docs, they might not be familiar with your documentation conventions or writing styles. This is an easy way for poor content to slip into your repo undetected. Unless someone is actively monitoring commit logs for repos, it can be easy for people to submit content with incorrect formatting or other issues. While you're at it, revisit your repo's permissions to make sure they're current and don't include people you added long ago for temporary purposes.

14. Check engineering commit dates against release notes

If you're tracking different engineering repositories for various projects, you might want to look at the latest commits to see if they track with your latest release notes. This might let you identify changes to the code that you've missed. Not every engineer knows how to contact tech writers for changes, and many engineers might be unaware of the impact of code changes on docs. Granted, keeping aware of which repositories correspond to which docs sets is a feat in itself, but if the code repo details are available, they can be helpful. Just look at the commit logs and glance at the titles to see if anything looks recent or relevant.

15. Look for broken documentation formatting

If you write docs in Markdown, I guarantee that somewhere, on some page, your Markdown is broken. There are many Markdown linters that will look for broken Markdown syntax. You could also look for common error patterns in full-text searches.

One easy formatting issue to check for is Git conflict markers in your content. Few things are as embarrassing having these markers exposed in your content, as this makes you look like a Git novice and also leads to interesting formatting in Markdown files. You can easily catch Git conflict markers from commits by adding a pre-commit hook ↗ into your Git files that looks for the markers. See this sample code from Patrick Vacek ↗ (just drop the code into your `pre-commit` file inside .git > hooks). This can help prevent these conflict markers from polluting your repo going forward.

Conclusion

The above items might seem daunting to tackle all at once, but if you can cycle in at least one maintenance ticket during each sprint, you can make maintenance a more strategic activity and make progress over a period of time.

Collecting feedback post-release

Although you already reviewed some of your docs (p. 764) prior to release with internal stakeholders, after you publish the content you now need to pivot to collecting feedback from external users post-release across your entire landscape of docs. Collecting feedback post-release involves a host of new challenges and processes that I haven't yet covered, such as how to optimize your feedback form, how to account for random externally driven requests across your dev portal, how to process the feedback from incoming requests, and so on.

What's different about collecting feedback post-release

When you're working on documentation prior to release, you have the full attention of the product team for reviewing and approving the docs. You have a group of product managers, engineers, and others at your present disposal. But when you receive feedback on docs post-release, that same group of people might no longer be present, or they might be hard to locate, especially if you're updating docs you didn't work on. You might have to assess feedback on products you know next to nothing about and reach out to people you've never interacted with before.

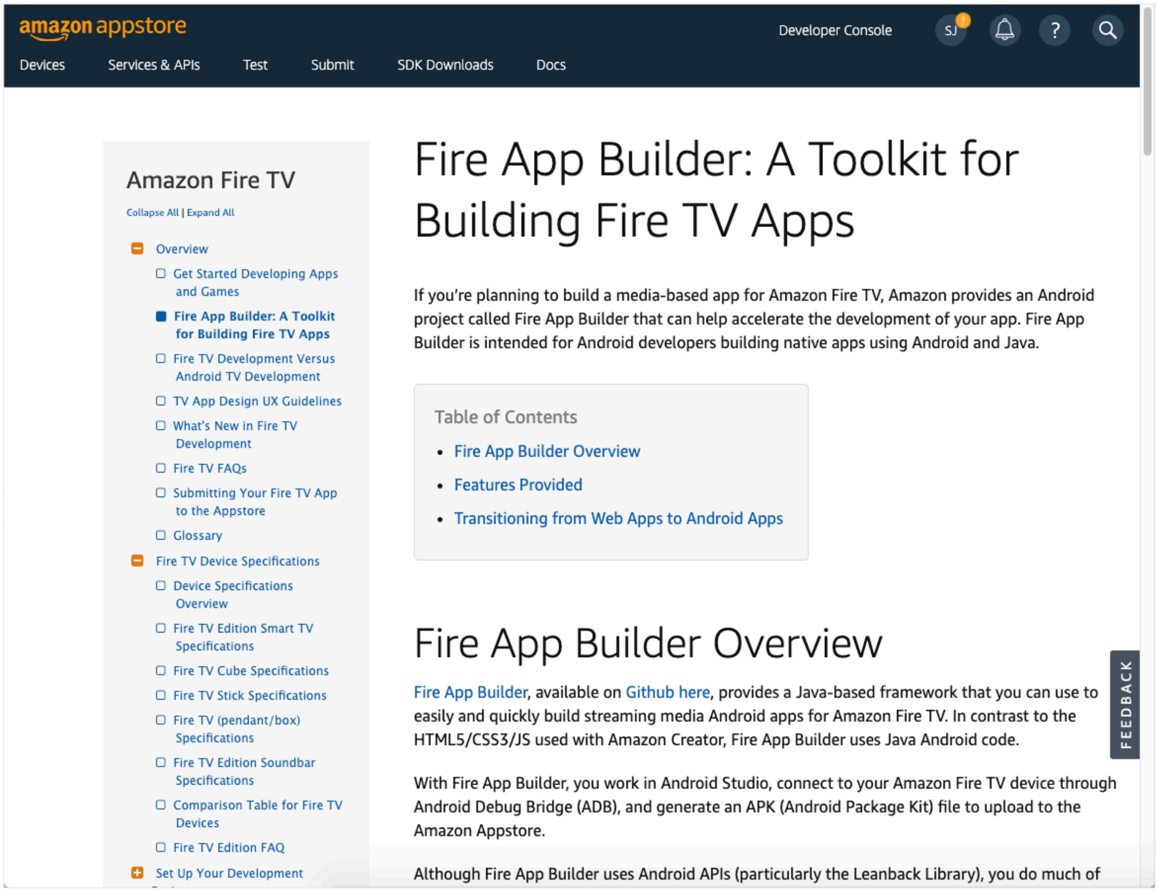
When working with post-release feedback, you also have the challenge of sifting through feedback from people who might not even be your target audience. The feedback might be from customers rather than developers, for example. Or the feedback might be from people looking for a support lifeline when other channels fail them. Or it might just be spam.

Finally, there's no built-in bandwidth to handle tasks like this, so you'll likely have to deal with these random requests on top of a full project load. Because of all these factors, handling feedback post-release presents many new challenges.

Location/placement of the feedback form

Let's start by talking about the feedback form's design — after all, without a good feedback form, you won't have any comments to deal with at all (which might be a strategy). There are many different approaches to designing your form, some better than others. In general, most web articles typically have a comment form *below the content*, particularly with web content. Readers are used to scrolling to the bottom of a web page and finding a feedback form. You'll have the most success with your feedback form if you follow this general web pattern for providing feedback.

You might be tempted to put your feedback form as a floating button on the side of your docs. Here's a sample with the feedback button on the side:


amazon appstore
Developer Console
SJ
?
Devices
Services & APIs
Test
Submit
SDK Downloads
Docs
Amazon Fire TV
Collapse All | Expand All
Overview
Get Started Developing Apps and Games
Fire App Builder: A Toolkit for Building Fire TV Apps
Fire TV Development Versus Android TV Development
TV App Design UX Guidelines
What's New in Fire TV Development
Fire TV FAQs
Submitting Your Fire TV App to the Appstore
Glossary
Fire TV Device Specifications
Device Specifications Overview
Fire TV Edition Smart TV Specifications
Fire TV Cube Specifications
Fire TV Stick Specifications
Fire TV (pendant/box) Specifications
Fire TV Edition Soundbar Specifications
Comparison Table for Fire TV Devices
Fire TV Edition FAQ
Set Up Your Development
FEEDBACK

Fire App Builder: A Toolkit for Building Fire TV Apps

If you're planning to build a media-based app for Amazon Fire TV, Amazon provides an Android project called Fire App Builder that can help accelerate the development of your app. Fire App Builder is intended for Android developers building native apps using Android and Java.

Table of Contents

- Fire App Builder Overview
- Features Provided
- Transitioning from Web Apps to Android Apps

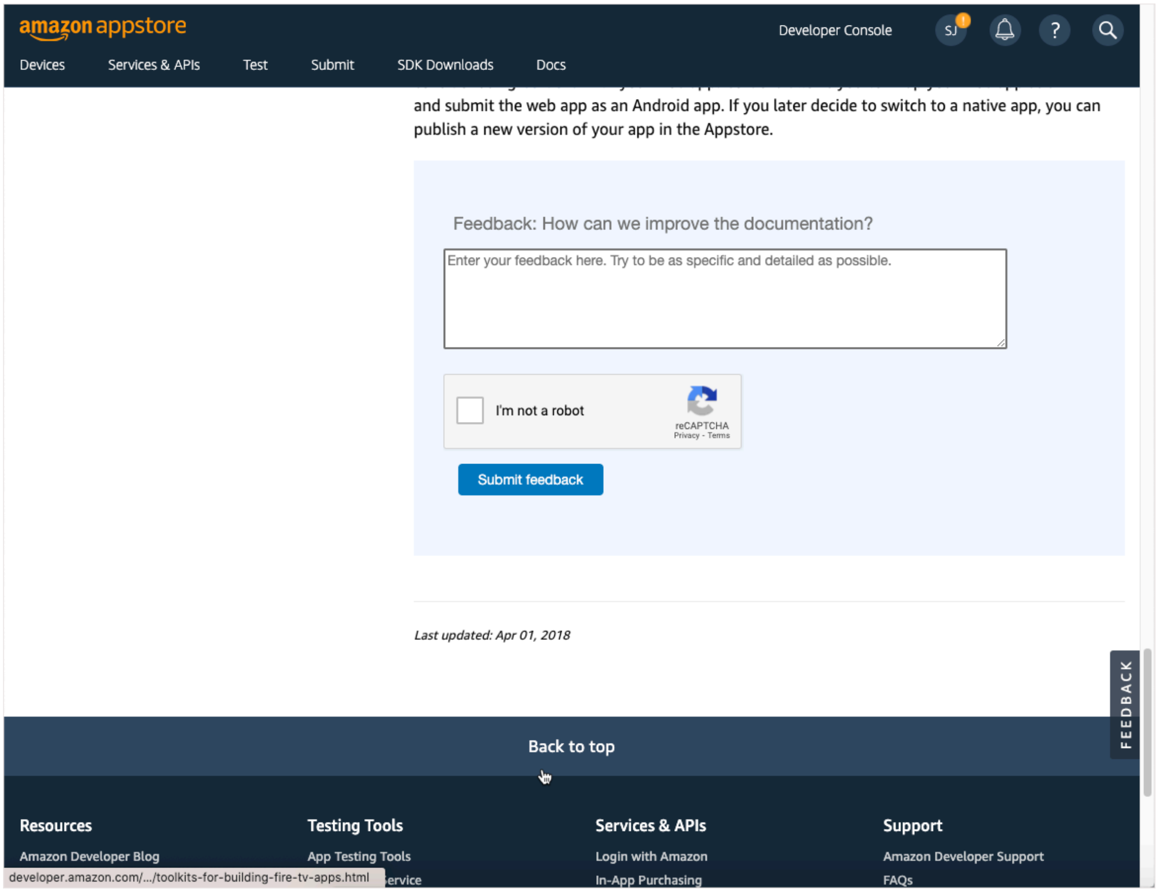
Fire App Builder Overview

Fire App Builder, available on Github here, provides a Java-based framework that you can use to easily and quickly build streaming media Android apps for Amazon Fire TV. In contrast to the HTML5/CSS3/JS used with Amazon Creator, Fire App Builder uses Java Android code.

With Fire App Builder, you work in Android Studio, connect to your Amazon Fire TV device through Android Debug Bridge (ADB), and generate an APK (Android Package Kit) file to upload to the Amazon Appstore.

Although Fire App Builder uses Android APIs (particularly the Leanback Library), you do much of

Although this side position is becoming more common, in tests that I've done, most people don't see the feedback button on the side. The feedback form at the bottom was used much more than the feedback button on the side (a ratio of 100:1 or so).



and submit the web app as an Android app. If you later decide to switch to a native app, you can publish a new version of your app in the Appstore.

Feedback: How can we improve the documentation?

Enter your feedback here. Try to be as specific and detailed as possible.

☐ I'm not a robot

Last updated: Apr 01, 2018

Again, follow standard web practices for placement of common elements.

Requirements to submit text

In addition to putting your feedback form in a highly visible place, if you can lower the friction required for users to enter and submit feedback, you'll get more feedback. If your feedback form requires developers to log in, or to proceed through a series of authentication screens, or do anything else requiring more than 5-10 seconds of their time, your engagement rate will go down.

The problem with reducing friction, however, is that the easier you make it to enter comments, the more spam you get. Without any spam prevention controls, you'll be overrun with spam comments in a short time. Try to find a good balance between controls that prevent spam but which don't challenge users too much. For example, you could implement reCAPTCHA or require some form of login (e.g., login with Google, GitHub, or Twitter), or require users to complete some special task to weed out bots. (Whatever you do, spam will usually find a way to slip through.)

Finally, in order for feedback to be actionable, it needs to be detailed. Consider adding minimum character requirements to the form and encourage users to be as specific and detailed as possible with their feedback. Users rarely want to sink energy into completing a feedback form with no guarantee that the form will go to a human and be read, but a quick one-liner from a user is rarely helpful enough to take any intelligent action on. You might require that the form has at least 100 characters of content to be valid.

You can also choose to show the comments submitted on the article (rather than just collecting the feedback). If you show the comments, and then actually respond to them, it can increase the sense of trust from users to enter their feedback (knowing that the comment will be publicly visible and responded to). However, managing comments places more of a support drain on your team and potentially redirects users away from better support forums and tools, so most doc teams avoid engaging in interactive comments like this. I agree with routing users to support for help.

Workarounds for obfuscated http referrer values in Chrome

One new challenge to collecting feedback in 2020 is getting around the http referrer obfuscation. When you collect feedback, you typically want to know the page that users were on when they clicked the feedback form so that you can make sense of the comments. For example, if the only text users write is "This page needs more examples" or "There's an error in the code sample here," unless you know the specific page, it will be difficult to take any action on the feedback. Especially given how brief the comments usually are, you need more context to make sense of the incoming information.

Unfortunately, more restrictive data privacy rules in Chrome now obfuscate the path from the http referrer, showing only the domain. For example, if you submitted feedback from this page, Chrome would show the referrer only as `https://idratherbewriting.com`, not `https://idratherbewriting.com/learnapidoc/docapis_collecting_feedback_post_release.html`. This new policy went into effect in October 2020, so if you're analyzing your feedback metrics and wondering why http referrer shows only the domain, recognize that it's not due to VPN or incognito modes in browsers but rather due to the new Chrome data privacy policies in place. (Other browsers haven't necessarily followed suit yet here.)

Depending on your feedback tool, you might need to implement a creative workaround. For example, in Qualtrics (which is a survey tool rather than a proper website feedback tool), you can add custom JavaScript to your form. For example, you could use this script from Stack Overflow that gets the query string from the URL using JavaScript and include that in the form submission. The logic looks like this:

```
Qualtrics.SurveyEngine.addOnload(function()
{

        var getUrlParameter = function getUrlParameter(sParam) {
    var sPageURL = window.location.search.substring(1),
        sURLVariables = sPageURL.split('&'),
        sParameterName,
        i;

    for (i = 0; i < sURLVariables.length; i++) {
        sParameterName = sURLVariables[i].split('=');

        if (sParameterName[0] === sParam) {
            return sParameterName[1] === undefined ? true : decodeURIComponent(sParameterName[1]);
        }
    }
};

        var docPage = getUrlParameter('page');
        var formLocation = getUrlParameter('location');

  jQuery('#page').val(docPage);
  jQuery('#feedbackLocation').val(docPage);

});
```

This script assumes that the feedback URL has query strings added, such as `https://idratherbewriting.com/learnapidoc/docapis_collecting_feedback_post_release.html?page=docapis_collecting_feedback_post_release&location=bottom` The script parses the query strings from the url and then sets variables `docPage` and `formLocation` for each query string. Then jQuery is used to assign those values to unique form elements on the page. You could also add CSS that hides these element where the values get assigned.

(Yes, this is a hack, but if you need to workaround the http referrer restrictions, a technique like this might be necessary.)

Qualitative versus quantitative feedback

Another aspect of the feedback form is whether to solicit qualitative or quantitative feedback, or both. Qualitative feedback refers to free-form, descriptive text whereas quantitative refers to multiple choice options (or similar). Although the idea of multiple choice options seems appealing, as it would allow you to create metrics around the content (e.g., rate the docs based on clarity, accuracy, relevance), quantitative data is actually rarely helpful. If a user ranks the doc low on clarity or relevance but doesn't add clarifying information, this makes the feedback hard to decipher and take any action on. Additionally, you'll have to sift through spam bot submitted forms versus forms from actual humans. Every time I've incorporated quantitative options into feedback forms, the results have been useless.

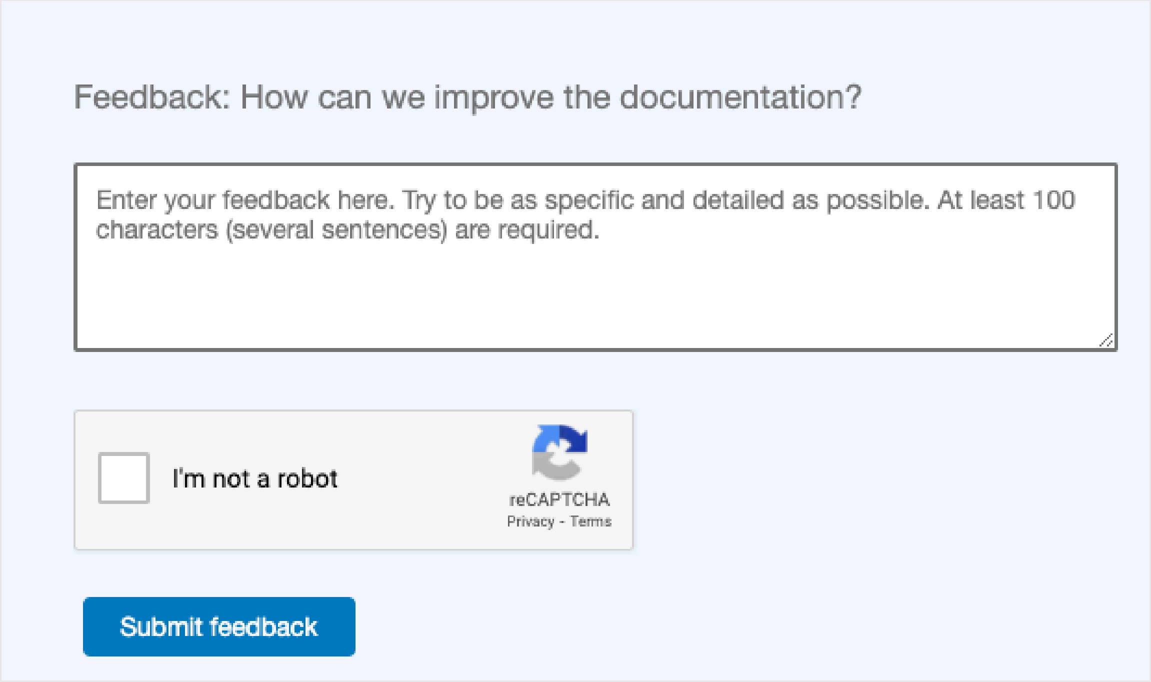
What questions to ask

Your feedback form can be as simple as asking one qualitative question:

> How can we improve the documentation?

Remember, people usually give 30 seconds or less of time in providing feedback. If you try to add a lot of questions, they should be specific to the information on the page, not general task-analysis questions.

I also recommend adding some placeholder text that encourages users to be descriptive, such as “Enter your feedback here. Try to be as specific and detailed as possible. At least 100 characters (several sentences) are required.” For example, like the following screenshot:



Feedback: How can we improve the documentation?

Enter your feedback here. Try to be as specific and detailed as possible. At least 100 characters (several sentences) are required.

If you need metrics around feedback, you could quantify the number of comments received for each doc set, and then indicate what actions were taken. What percentage of the feedback was actionable, and how many tickets did you close related to doc feedback, etc.? How many tickets were from humans versus bots?

Process for handling feedback

Just collecting feedback is only the first step. The next step is to implement a process for sorting, assessing, and taking action on the feedback that comes in. This might sound easy but can be especially challenging if no one is asking you to take action on this feedback. Product teams usually submit requests to doc teams for upcoming features and releases; they are often quiet when it comes to documentation for existing features. Taking action on incoming feedback is almost like an extracurricular task.

A process for handling incoming feedback could look like this:

1. Every time you do sprint planning (p. 737) (e.g., every two weeks), look through the feedback you’ve received and identify any feedback that is detailed and actionable. Create tickets for each of these actionable items.

2. Decide which tickets you want to work on for the sprint (e.g., due to priority, ease of handling, or other reasons) and assign the tickets to the sprint during your sprint planning.

   Given that these tickets will rarely be a priority, to avoid neglecting and forgetting them, consider making it a requirement that each team member takes on at least one documentation feedback ticket per sprint.

3. Reach out to product teams to keep them aware of the incoming tickets. The product owner should be Cc'd as a watcher on the ticket so they are aware of what's going on with the user experience of their product.

   Product teams rely on various teams (documentation, support, field engineers, developer advocates, etc.) to gather feedback about their product. Especially if you see trending issues related to specific pages, try to identify what might be driving the issue and escalate this to the product owner.

   To allow contact for docs across your portal, it's a good idea to maintain a list of contact points that identify at least one product owner per doc set. It's even better if you can identify the team that owns the product and any ticket assignment categories for the team.

Conclusion

Although collecting doc feedback post release is important, most feedback I've seen in comments rarely goes beyond identifying simple issues. Most users submit feedback to let you know about broken links, typos, broken code samples, or other obvious issues. You won't get the detailed, substantive feedback from doc reviews that you do as part of pre-release review process (p. 764) from internal product teams and stakeholders. The best time to review your docs is before release, not after. The feedback you collect post-release can help catch errors and other problems, though, and so shouldn't be avoided. This feedback can also help you funnel insights from customers back to product teams, which helps increase your value to the organization.

Managing content from external contributors

One of the main advantages of a version-control-based system, especially using open-source technologies, is the promise of collaboration. Not just collaboration with your immediate team, but scaling beyond your team to also include other contributors within your organization and even contributors from the community. The idea of crowdsourcing documentation like this, removing the burden from a small number of gate-keeping writers, has been appealing since the emergence of wikis and continues as a driving force behind docs-as-code implementations.

Many people embrace docs as code with the hope and expectation that many engineers will contribute to the docs. This is, after all, why so many embrace an engineer's tools and workflow in the first place. In this section, I cover processes to consider when external contributors (external to your team, not necessarily external to the company) write content.

The complexity factor is Git

When I started at Amazon and we were discussing authoring and publishing tools, we adopted a docs-as-code approach using Jekyll and Markdown for several reasons. Not only was this tooling free and fit seamlessly within the engineering build systems, in adopting Markdown and a static site generator, we rationalized that we could scale the solution to allow other teams to author and publish content without having to go through us. We envisioned lots of small, autonomous engineering/product teams writing and publishing content within the doc system, without necessarily involving us. We would set up the system, and they would use it.

For the first couple of years, when a team approached us and we lacked bandwidth to write their docs, or when the team fell outside our department's stewardship, we gave them permissions in our doc repo, set them up with some basic training, and invited them to own their own authoring and publishing tasks. We gave them full permissions to merge in content, build the project, and push out the updates to the live site.

Admittedly, we didn't have nearly as many teams as we envisioned writing content. Maybe about 10 teams over the years, with off-and-on spurts of writing. But as I've observed this process play out, I've noticed one thing: Git is the complexity factor that trips everyone up. Engineers understand Git because they live and breathe its commands in their own software team workflows, but do they follow your team's specific Git workflows, which might differ from theirs?

You might have multiple branches corresponding to different environments (beta, prod, etc.) and different workflows uniquely designed for a documentation repo. But each engineering team might have their own preferred Git workflow. Git is flexible and robust enough to allow for many different processes and workflows — this flexibility is part of why Git has survived and thrived so long as a way to collaborate. Will those same engineers bother to follow your own team's defined Git workflows? If not, get ready to deal with merge conflicts and scenarios where you try to understand why certain files keep getting flagged as problematic.

Besides engineering contributors, consider also the non-engineering contributors. Not every contributor understands Git. Many marketers and product managers might want to contribute to docs but not understand Git well enough to use it correctly. They might type enough commands to get something to commit but then actually create massive conflicts and other issues for everyone else. You'll have to sort all this out. Alternatively, you might end up just taking input requests that they describe in tickets.

Also note if you give external contributors permissions to author and publish content, you allow them to introduce new content into your document repository that hasn't been vetted and reviewed by your team. If you review the content, you might find that these contributors are much more sloppy about writing than you are, committing broken links, incorrectly cased product names, bad grammar, broken formatting, eternal paragraphs, and other embarrassing issues. Will others understand that commits from these external contributors weren't actually written by your team?

Some groups might not want to join in the same doc repo as your team's repo but instead create their own repo based on the theme and other tools setup you created. Different static site generator tools handle theming better than others, but a separate repo might end up duplicating many theme files that these contributing teams actually don't want to maintain and update. You might help set up a team with a new instance of the theme, only to find that they never update the theme files as the original theme evolves.

After dealing with so many support cases, broken builds, and poor content committed by external teams, we decided to pivot to more of a GitHub-pull-request model. In this model, rather than giving contributors permissions as full-fledged authors in a repository, we required them to submit pull requests (aka "code reviews" in some build systems) for their contributions.

With pull requests, the contributor clones the repository, makes a change, and then submits the change to a system that lets the repository owner evaluate the file difference that the contributor is proposing. You (as a repo owner) review the contribution, adding line-by-line comments as desired, and then either accept or reject the pull request. You can go through multiple iterations with this proposed change until the contribution meets your standard. After you approve the pull request, you then merge the changes into your repository and push it out.

This pull request model is the standard contribution model for most groups (especially on GitHub). With this model, contributors don't need to understand the details of your Git workflow (beyond the pull request workflow) or other specifics about your environment's branches (except which branch to start from). They just need to interact with the code review tool to submit their pull requests. This is the model I recommend implementing.

Community contributors

So far I've talked about external contributors within your company, but what about community contributors? You could apply a similar pull request model as with community contributors outside your company, but there are added complexities with community contributors:

- You'll need to expose your Git repository to an external site like GitHub. This might be problematic if you aren't allowed to be transparent about content prior to release. For example, you'll need to find a way to hide content for upcoming releases. Of course, you could keep it on a local branch, but then it wouldn't be backed up. If you add the GitHub repo as a remote, this could work easily enough, but will you also expose your commit messages (which might reference internal tickets) and other workflows?
- Extensive contributions might require rights-usage signoff from your Legal group. Fixing typos and broken links probably wouldn't require signoff, but if someone is contributing new pages of content or substantial sections, this will probably prompt Legal to require the contributor to sign an agreement. You might find yourself constantly doing paperwork for contributions.
- You might get contributions that don't align with your organization's strategic objectives, and then you'll have to spend time editing and publishing that content. Suppose someone wants to add a new article; you'll now need to vet the content, get it reviewed and approved by the appropriate internal SME, make stylistic edits, and do other shepherding (for which you might not have bandwidth). If the content doesn't line up with anything you're currently working on, this sudden request might be a disrupting detour from your regular work. Additionally, you might not like the approach or style the person wants to contribute; rejecting the contribution might make

for an awkward conversation.

I've tried leveraging community contributions in the past and found that the efforts aren't usually worthwhile (at least for the products I was documenting.) Community members rarely have the internal knowledge needed to create documentation. More importantly, they lack the incentive to do so, as this task doesn't usually benefit them. I wrote about this problem when I abandoned wikis a decade ago — see My Journey To and From Wikis: Why I Adopted Wikis, Why I Veered Away, and a New Model ↗. The hassle required to make docs public and available for pull-request workflows in GitHub rarely pays off with community edits.

I wouldn't recommend trying to orchestrate community contributions unless you already have some heavy community contributors who are itching to write and publish content. It also might make sense for some products more than others (such as company products that are integrations into third-party platforms — e.g., *here's an article how you integrate your product with my product*). You might be better off adding a note in your docs inviting contributors to reach out to you with documentation ideas/requests, and then work with them individually through some tool like Google Docs or Microsoft Word.

If you do allow contributions from external contributors or community members, be explicit about the processes they should follow. GitHub projects often have common templates and guidelines (often in a the readme) about how community members can contribute.

The content abandonment problem

One pitfall to watch out for with external contributors is the content abandonment problem. It's common for external contributors to get involved in adding documentation in a time of need, and then abandon the content when their product fizzles or when they switch teams. Then you're stuck with content that you don't know much about, with no clear owner, reviewer, or maintainer. Essentially, these scenarios are like adopting an orphan child that you now you have to watch and care for. This is often how teams end up with a lot of legacy content.

To avoid this content abandonment scenario, make sure you have a clear policy around the review and ownership of the content from contributors. Track who owns the content, the contact points for questions related to the content, and expectations for regular review of the content. It takes a lot of proactivity to track and enforce this, though, because again, no one will be knocking on your door to regularly review and look over existing content. As I wrote in Processes for maintaining existing documentation (p. 770), tech writers are mostly focused on new content, not maintaining old content. Almost every tech writer I know has more work than they can currently handle, and reviewing old docs that no one is complaining about or requesting rarely climbs up the priority list.

Conclusion

In a docs-as-code model, contributions from external people in your company are expected. You need to define specific workflows that make sense for your organization. In general, you'll have the most success if you stick to standard, simple workflows (I'm mostly referring to Git and pull requests or code reviews) that align with common practices the engineers are already using within your company, or industry patterns. There's no need to reinvent the wheel when it comes to Git workflows. Pick a model such as the GitHub pull request workflow (see this GitHub tutorial ↗ or this Atlassian tutorial ↗) and provide code examples to help people easily understand and follow the right commands.

Changing internal doc culture

One of the most influential aspects that will determine your experience as a technical writer at a company is the company's documentation culture and environment. If you find yourself in an organization with a poor documentation culture, it can be difficult if not impossible to change it. A poor documentation culture/ environment leads to a high turnover on doc teams, loss of motivation for existing writers (especially as their colleagues constantly leave, which increases the workload), and contributes to a downward spiral of tasks you can never quite get a handle on. In this topic, I outline six strategies you can implement to influence change in your company's documentation culture, bringing about a more positive and healthy environment.

Characteristics of a poor documentation environment

Poor documentation environments can be characterized by the following:

- **Lack of executive sponsorship or support**. In these companies, no one at the executive level is a strong believer in docs. They might give lip service to the importance of docs but then fail to adequately staff resources, not only resources for writers but also for the tools infrastructure and any supporting engineers. Docs rarely bubble up to their radar, and tech writers are invisible to them. These leaders give the impression that they don't care about docs, and the attitude tends to flow downward through leadership chains.
- **Lack of integration in formal processes**. Engineering processes that don't require documentation as part of the "definition of done," or which don't include documentation as any requirement for product development and release, can also devalue the tech writer role. Documentation requests might come in at the last minute, when the features are fully coded and ready for release in a few days. In these companies, there's no formal way documentation is integrated into development and release processes. It's all ad hoc and last minute.
- **Lack of enough resources for the required work.** In a company with poor documentation culture, you might always feel short-handed and unable to ramp up on the needed work. You might move from project to project in a cursory way, never able to adequately learn the technology or become a SME before you have to start on the next project, which might be a completely different technical realm. The lack of resources goes hand in hand with high turnover rates. As soon as someone arrives, another leaves, and you end up doing more work than your role alone, which also motivates more turnover because people are overworked.
- **Check-box mentality is common.** If product and department leaders see documentation as simply a box to check box to select and nothing more, this can also trivialize your experience at that company, especially if that check box has no standards or minimum bar associated with it. This "check-box mentality" can be characteristic of people who show little interest in the actual user experience. As soon as the docs are done, they're crossed off the list and rarely revisited.
- **Support for tooling is underfunded for the requirements.** You might need robust tools to provide the kind of experience people require (PDF, localization, gated docs, etc.), but the company might not want to fund any tooling resources, insisting that you use open-source tools instead, or not providing enough time to build out tooling you need.

Most tech writers are familiar with some of these aspects. I've written about these topics multiple times on my blog, especially in these two series:

- Reflecting seven years later about why we were laid off ↗
- Value arguments for docs and tech comm ↗

In the following sections, I'll explore processes you can implement to change your documentation culture.

One question is whether a company's documentation culture can actually be changed through efforts from the tech comm group. I believe change is possible, to an extent. You might not convert your CEO into a doc champion, but you can influence your environment and culture if you work hard at doing so. Especially if your company has a business reason that heavily depends on the customer experience, you have a shot at influencing upward. These efforts should be factored into your team's regular processes and goals.

Among all processes described in this section, changing culture is the most difficult because it requires changing the attitudes and minds of others for which you have little control. It also requires a lot of marketing work that you might not have the bandwidth for. I also can't say that I've had a ton of success in this area, but I will share the strategies that have worked for me.

Six ways to change documentation culture

The following are six ways to change documentation culture at a company.

1. Attend engineering Scrum/Kanban meetings with engineers

My best experiences have been when I'm integrated closely with engineering teams, attending their daily standups, sprint demos, and other scrum/kanban meetings. Organizational setups don't always allow for this type of embedded engineering-team integration, but hands-down, whenever I've done this, it has absolutely changed my experience with the product team in extremely positive ways. Integrating with engineering teams builds relationships with these groups, and they see your value, and you can anticipate their doc needs.

The constant problem is that tech writers usually support multiple teams, so integrating fully would drain the tech writer's time too much. It doesn't really work to attend regular standups for more than a couple of engineering teams, but consider maybe attending 1-2 standups a day, rotating to different engineering teams depending on how many teams you support. Even if you just drop in once a week to a different team each day, this will dramatically change your visibility and rapport with the teams. Usually, teams don't have so much doc work that they need you there daily (and if you do attend the same team's standup on a daily basis, you might actually feel it's a waste of time), so occasional attendance might be ideal and build the needed rapport just fine.

2. Become visible by letting people know what you've written or updated

A critical step to influencing change is to make your influence visible. As one of the most visible tech comm bloggers in this space, I know how to make myself known to those around me. The recipe for visibility on the web is simple: write relevant content and share it with those around you — *over and over and over*.

In the context of documentation teams, you don't need to write blog posts. Instead, make a list of all your recent updates each month, and then send it out to as many relevant stakeholders as you can. This will help them see on a regular basis what your team is creating. They'll come to understand your role, what kind of content you create, and what's being changed. When the topic of documentation comes up, you'll be first on their mind.

3. Relay customer insights back to product teams

One of the most valuable assets at a company is customer knowledge. If you can communicate customer insights to internal product teams, this raises your value and relevance in a major way. Add a feedback form (p. 775) at the bottom of each of your doc pages. On a monthly basis, review all the feedback, analyze it, and send it out to product teams and other stakeholders. Include other metrics about the most popular pages and trending search results. People love this kind of customer insight, especially business leaders trying to understand market trends.

If you want to take it a step further, undertake a competitive analysis highlighting how the developer's experience on your company's products compares with the developer experience on competitor products.

4. Move the needle in a noticeable way with doc quality

Doc champions are converted to a pro-docs mentality through close experiences with documentation. I remember hearing a developer petition and advocate for product teams to have technical writers work on a specific product's documentation because of the way we turned around documentation for another product. If you can turn docs around in impressive ways, translating engineering-speak into real English, making the steps easy to follow and concrete, people will notice. You can't just fix grammar and punctuation — you have to actually move the quality needle in noticeable ways. For example, implement a workflow map ↗ to visualize a complex process.

When you do improve docs in significant ways, others will start to ask for tech writers to play closer roles in creating the documentation, not just editing and publishing docs that product teams write. If you insist on high standards in docs, such as requiring teams to provide a sample app, testing out all the steps yourself and ensuring they work, adding troubleshooting topics, glossaries, search, etc., people will sense the value you're providing and start to champion the inclusion of docs as a requirement for each product release. I've had many experiences where product teams wrote some documentation, floated it to developer advocates, and the developer advocates told the product team to send it to the doc group to "polish." But by polish, they really meant fix the organization, structure, clarity, readability, and more. These developer advocates had learned what we could do with content and started insisting on it as a standard.

5. Be in the right group organizationally

Okay, this last point is controversial and opinionated, but it can't be left out. Where you are in the organization matters. Are you grouped in Product, Marketing, Engineering, Support, or some other department? I've been in nearly every organizational discipline, and my best experiences are when I'm in Engineering. This is because engineers are heavily involved in the product *development* phase, and this is where you want to be as a technical writer. Marketers are too involved in the release phase, support in the post-release phase, and product in the pre-development phase. But engineers are in the development phase. As a technical writer, you're *developing* content, and you need the sprint cadence of weekly development and workflows. Engineering understands this flow, and documentation groups fit well into it. (Many of these other groups don't even follow Scrum/Kanban workflows.)

If you're not in the right group now, don't worry. In my 5 years at Amazon, I had 5 different managers and experienced about the same number of re-orgs. I'm sure I could have influenced where the doc group was moved during different re-org periods, but I rarely had any foreknowledge of upcoming re-orgs; they just happened without warning.

6. Apply your energy to the right projects

Most organizations have key strategies or goals they're focusing on. For example, your organization might have a goal to get 30 partners to integrate a particular technology. If you align your documentation efforts in this same technology direction, helping fulfill the executive's key objective, it will raise your importance to the executive leaders in the organization and make documentation a more important function. I wrote about this in more detail in 2. Assess the identified work against strategic priorities (p. 739) in "Processes for managing large documentation projects." In short, don't assume all documentation is equal. It isn't. If you spend a lot of time and effort on docs that no one uses or cares about, you'll have misused your energy.

Conclusion

Company culture towards documentation isn’t easy to change, and just like people don’t change overnight, company culture doesn’t transform overnight either. Many perceptions people have around documentation have been hardened by years of disappointment or frustration with documentation. You might be facing an uphill battle that began when the previous tech writers in your role failed to deliver anything of substantial value, or you might be battling against people who believe no one reads docs and so docs are a formality only, not actually something that is used. Changing people’s minds about the importance of documentation might be a gradual process that you influence over several years. But if you follow the six strategies I outlined here (in greater degrees if you want faster change), you will eventually have success.

Sending doc status reports -- a tool for visibility and relationship building

Sending documentation status reports can help foster trust and awareness with your business stakeholders. These stakeholders might be the core leadership within your organization or simply your management chain the next level up. Besides building visibility and relationships with these stakeholders, creating these status reports each month gives you a regular cadence for doc assessment and analysis, which is also helpful.

In my post Some good decisions and minor mistakes, I said one of my good decisions at Amazon was sending regular doc reports to our business leadership. Here's a passage from the post: "Broadcasting all the recent doc updates made during the past month or so did an amazing job at increasing the visibility of our team. This had more impact than any doc-related metrics." In the post's comments, a lot of people asked me to expand on this point with more details. This article provides a follow-up about that point.

Sample status report

A sample status report has sections such as these:

- Purpose and intro
- About your team
- Recently published docs
- Upcoming doc work
- Support deflection efforts
- Doc metrics
- Strategic initiatives

Here's a bit more about each of these sections.

Purpose and intro

This section provides a few lines about the report and its purpose, and might start out like this: "This documentation report covers such and such time period and includes information about newly published docs, metrics, support deflection efforts, a review of strategic initiatives, and more. The purpose is to keep others updated about doc efforts and strategies to ensure alignment across groups, etc." Hopefully, you can make your intro more exciting than this, but you get the point.

About your team and where you publish docs

This section explains a bit about your team, such as "The documentation team consists of X number of writers supporting external documentation on X site. The team works with engineers across A, B, C product teams to provide documentation for such and such products. We also help edit and publish contributions written by engineers directly."

Although these details might seem obvious to you, in any large organization, there are usually many groups that aren't aware of each other, despite working under the same organizational umbrella. In my experiences in big tech companies, only about half of the engineers have ever worked with technical writers, and of those, many have only a vague idea of what we do or even where the docs are published. They also don't know the process for requesting doc work, so you could include a line about that workflow too.

Recently published docs

List out what your team has recently published. Ideally, you should be able to copy and paste this content from your release notes. In the list of published docs, include the article titles, short descriptions, and when they were published. The docs should link to the actual content so that people can read more.

If you don’t have many recently published docs, you could link to some drafts in progress. However, I find that this section helps keep me accountable. I know that if it comes time to write this report and our team didn’t publish anything for the month, it’s going to look bad. I want to have a handful of articles to show each month. It makes me feel like I’m earning my keep.

Upcoming doc work

List a few high-priority projects you’re currently or soon-to-be working on, so that people know where you’re headed. Describe your roadmap more or less, focusing at a high level. Sometimes readers want to know if such and such project is on your radar, and this section provides describes that horizon.

Support deflection efforts

Through an analysis of ticket logs, explain how documentation is reducing the drain on your support team, whether that support team consists of support engineers, partner engineers, or some other contact point when users/partners/developers have issues.

This activity forces you to regularly review the ticket log and see what frictions the users/partners/developers are having. For example, you might have an issue tracking system with hundreds of bugs logged each month. Look through the bugs and pick out bugs that could potentially have been averted had the information been available in the documentation.

For example, suppose a partner says they are confused about some data element and its availability within an API. That ticket should likely never have been filed; the partner should have been able to find the information within the documentation directly.

In my experience, support groups and product teams are blind at identifying tickets that are preventable through better documentation. Extrapolating potential doc work from the ticket log (as tedious as it may be) is something tech writers excel at. Others are typically blind about how a ticket might relate to documentation additions.

In this section of the report, comment on trends you see for the tickets filed. Create doc tickets related to the bug tickets. Then you can report on your progress against those tickets. This activity — looking through bugs at a monthly cadence — is one of the best ways to stay connected to user pain points and frictions. This section also presents an implicit argument about the value of tech comm — reducing the workload of engineers and support groups.

Doc metrics

Report on some aspect of your site’s traffic and analytics. For example, you could report on your core metrics: pageviews, users, sessions, pageviews per session, time on page, and bounce rate. You could also describe user profile characteristics (if available), such as location, age, gender, operating system, and browser. The location can be particularly useful in deciding about localization.

You could also list the most popular pages in your documentation and explore reasons why. Especially if the pages have unexpectedly high traffic, it could be good fodder for analysis.

If you have an aptitude for metrics analysis, you could also provide insights about these numbers. For example, analyze why there are spikes or dips. Look at trends about user growth or shrinkage over time and probe why. Look at search keyword hits and analyze whether the searches connect with docs. Look at your average time on page and assess whether it's too long or short. Look at flows that users take within your site, and why users might be going to the pages they navigate to (for example, why are users going to Requirements after the Overview instead of the Getting Started tutorial?), and more. Showcase your ability to crunch and interpret data.

Warning: Analytics can be a rabbit hole that can consume a lot of time without leading to actionable results. For example, whether the average time on page is 3 minutes or 5 minutes, is that going to dictate your information architecture strategy? Probably not. The most valuable metrics are usually your top 10 pages, which can give you a sense of where to prioritize your content development efforts. Dive into metrics, but come up for air soon after.

Even if analytics aren't always actionable, it's worth periodically reviewing the numbers to try glean as many insights as possible. Every year, I regularly review analytics on my blog. See the section Posts analyzing site analytics.

The biggest action item to come from these blog analytics assessments has been to see the large traffic going to this API doc site, which prompted me to focus more efforts in this direction. At Amazon, when I looked at analytics, I realized that device specifications regularly trended as the most popular pages, so I put a lot of effort in into Fire TV device specifications. In short, your most popular pages can indicate where you should prioritize your time.

Strategic initiatives

Discuss a few points of your doc strategy at a high level. Remember, many people might not realize that technical writers even have a strategy or what it could possibly be for docs. You can comment on a few high-level goals, such as making sure each product has a well-developed overview and getting started tutorial. Or your strategy might be to allow for more hands-on testing of content by technical writers in an effort to improve documentation quality. Or your strategy might be to fill in some content gaps according to your critical user journeys. Or maybe you're trying to align better across product areas with other teams with more consistent structure and naming. Whatever your strategic goals, comment briefly on your progress against them. This reinforces the idea that tech writers are strategic thinkers, not just content creators.

Other sections you could include

For other ideas about doc report content, see the suggestions from Saul Carliner in Eight Tips Healthy Contractor for with Clients Relationships. Saul addresses status reports from contractors to clients, so it's a bit different from full-time employees and stakeholders, but not so much. Saul writes:

> … provide reports anyway to address the natural concern of clients that you will complete their work on time and within budget. Take the initiative to send reports. The reports should identify:
>
> - Most recently completed milestone and whether that occurred on time, early, or late (and if late, why).
> - Next milestone, who has responsibility for meeting it, and whether it is likely to occur on schedule, early, or late (and if late, once again, explain why).
> - Major issues that need to be resolved. (March/April 2018, *Intercom*)

Saul says that these reports build confidence with the employer that you, as a contractor, will be able to meet the project's documentation goals on time and on budget. Business stakeholders also want to know that you'll finish the docs by the expected release schedules.

Saul also notes that the initial reports might take a while, but once you've created a few, you'll have an established format and routine, and the effort will be less time intensive. With each report you send, you can fine tune your structure, language, and style in these reports.

Finally, as Saul notes in his last point, you could also add a section on issues and obstacles. Be careful of throwing other teams under the bus in a public way here, though. For example, if your biggest obstacle is that engineers don't review docs, or that you're understaffed for the amount of work, or that your company's culture is anti-collaborative when it comes to docs ("docs aren't my job!"), you could call this out in the report, but do it tactfully, without laying blame with any particular group or person.

Frequency, format, audience

Now that you have a sense of what the status update involves, you have some other decisions to make:

- **Frequency**: I recommend sending out a status report on a monthly basis. Biweekly is too frequent for all the sections noted above, and quarterly is too long for people to remember anything from the previous report.
- **Format**: You could package the content up into a snazzy newsletter format, but a simple text email is fine. I start the doc report in Google Docs and then just paste it into an email when I'm ready to send it. I think people respond better to raw email rather than shiny newsletter templates.
- **Audience**: Send the report to your business leadership groups, your team's alias, other adjacent writing teams working in the same general space, and more. You don't want to spam everyone, of course, but within an enterprise setting, people more or less expect to receive these reports unsolicited. Look at other reports being sent by other groups, and perhaps copy the same groups listed in their "To" list. The risk of *not informing others* is worse than inappropriately including them. A frequent complaint in most tech orgs is poor communication. People prefer to be informed.

Addressing the discomfort factor

Sending these emails (which might go to hundreds of people depending on the email lists) is usually something that makes writers feel uncomfortable. You might feel like you're tooting your own horn, and most writers, often somewhat introverted, tend to prefer not to call attention to themselves. Many writers like to stay in the shadows.

Staying in the shadows is an approach to avoid if you want to get ahead at work. As a blogger, I often feel uncomfortable sharing posts on Twitter, Linkedin, and in a newsletter. Every couple of weeks, I'm basically saying "Hey, look at what I wrote here! Read my thoughts! See what ***I*** have to say." This isn't my nature. I'm not overtly charismatic, extroverted, social, or prone to transparency. At a party, I don't naturally introduce myself to strangers or jump into the middle of conversations. But if you fail to promote your team in reports like this at work, your team will suffer from invisibility, reduced status, and marginalization (to a degree, anyway).

I've written previously about this topic, as these themes have characterized the tech writing profession for years. See these posts for elaboration on this topic:

- Guest post: Why are technical writers often treated as such an unimportant part of a company? ↗

- Value arguments for docs and tech comm ↗
- Reflecting seven years later about why we were laid off ↗

Can sending monthly doc reports fix the devaluation of the tech writer role? Maybe not, but this is certainly a way to help address visibility. The formula for visibility and recognition on the web with blogging is more or less the same as within an enterprise: write and promote, write and promote, write and promote, and so on.

Also, know that business leaders like to read these reports because they usually try to gather input and data from many different groups under their leadership to inform their decision-making. Especially if your upper leaders don’t understand documentation (because they’re in another business role), these reports can help educate and inform them. It’s a way to influence up.

A time for self-reflection on your docs

Another important aspect of writing these reports is the time and space they give you for reflection, planning, strategizing, course correction, and more. Without the report, when do you review these things? When do you look at metrics? When do you take stock of what everyone on the team is publishing, and how your work aligns with the upcoming roadmap? When do you look to see if the doc-related tickets are increasing or decreasing? When do you assess how you’re doing with your doc strategies? Creating the report benefits you as much as anyone else. (Same as with writing blog posts — these posts help me refine my thoughts and perspectives more than they might help anyone else.)

So even if you create a lengthy report that few read, that’s okay. The report is for you as much as anyone else.

Outcomes of regularly sending status reports

What are the outcomes of regularly sending status reports? If you send regular status reports, and the content is insightful and well-presented, you can expect the following to happen:

- People who you didn’t know previously will suddenly reach out to you.
- Your manager’s manager will love it and will reply to the report with praise.
- You’ll become more visible to the people around you, especially if you’re the one sending the report on behalf of your team.
- People will reach out to ask if certain docs or projects are on your radar.
- You get better at planning and anticipating long-term doc work and needs.
- When it comes time to write your annual review, you’ll already have a body of content to draw from.

See also Broadcasting your meeting notes to influence a wider audience (p. 794) for another approach to influencing those around you by sharing documentation topics and issues you face.

Broadcasting your meeting notes to influence a wider audience

In a previous topic, Sending doc status reports – a tool for visibility and relationship building (p. 789), I explained how to create and distribute documentation reports. Another tool for accomplishing a similar purpose — that of making others in your company aware of documentation processes, newly published articles, how to work with your team, etc. — is to broadcast your meeting notes after each meeting. Although sharing meeting notes with meeting participants after the meeting isn't anything new, with a few small adjustments, especially broadening your distribution list to those outside the meeting, your notes can be a powerful way to influence those around you.

The basic process

The basic process of sharing meeting notes hardly needs explanation. It's mostly just a bit time-consuming and might put you out of your comfort zone. First, you need to determine which meetings you'll generate notes from. Usually, you have some meetings with partner engineers, meetings with other teams publishing in your dev portal, meetings with your own team, and meetings with some teams for high priority projects. Decide which meetings you'll distribute notes for. Select the meetings where you focus on docs and topics others might care about.

Each meeting should have an associated meeting doc. You or someone else takes notes during the meeting. (Hopefully, in this same doc, you have an agenda to prime the meeting's focus and discussion.) Afterward, you type up the meeting notes and send them via email to all relevant groups. The relevant groups are usually broader than the meeting attendees, and that's part of the strategy. That's really all there is to it, though I've expanded on some more aspects of the process below.

Meeting notes are often more interesting to read

Meeting notes provide a natural story foundation to build around. Usually, meetings involve discussions of issues. In writers' terms, issues are really conflicts. And as all writers know, conflict drives story, and story engages readers. This is why you might find that typing up meeting notes is more fun than writing documentation reports. You can describe real issues you're running up against, strategies you took to solve the issues, outcomes, and more.

With each meeting, you've got the foundation for interesting content. Now you can shape and leverage this content to provide other themes and messages that you want to get across.

What if sensitive topics are discussed during the meeting?

As with any real issues, the matters discussed might be sensitive. If people are complaining about what a pain it is to work with Frank (who wasn't in the meeting), you probably don't want to highlight that in the notes broadcast. You'll have to incorporate some euphemisms around sensitive topics. For example, don't write, "Some writers said Frank is impossible and should not be leading the publishing tools project," but rather "Some writers noted some challenges in collaboration in the beta process with the new publishing system."

As you get into a rhythm of sharing meeting notes after the meeting, people might begin to recognize that anything they say might be shared with others outside the meeting. This fear can be a detractor for people opening up during the meeting, so make sure you don't expose anyone in a way that would be inflammatory or embarrassing. For example, don't write, "John said he can't stand working with the ACME engineering team because they never review his docs, so he seems to have deprioritized their project." Instead, tone it down: "Some members noted challenges in getting timely doc reviews; getting ignored until the last day before release can be demotivating and create unnecessary crisis situations."

On the flip side, sometimes surfacing issues is a good thing, as it forces the issues into open daylight where some action can be taken. It doesn't make sense to hold everything inside all the time, fuming in silence. One of the purposes of sharing notes is to effect change. If you have an ugly issue to deal with, sometimes writing about it with transparency and honesty is a good approach, even if it ruffles some feathers or induces pain. For example, "Writers were unable to get timely reviews from the ACME engineers, which caused us to shift priorities to groups that value our time and energy more." A sentence like that might send shock waves into the ACME group, but hey, you wanted change, right?

The structure of meeting notes

The meeting notes tend to have the following basic components:

- Meeting title
- Meeting date
- Meeting attendees
- Meeting description
- Meeting topics and discussions
- Action items

In the list of attendees, use company aliases so that others can get more context. If you just list first names (Dave, Sue, Ashwin), others might not know who you're referring to. If you have aliases in the attendee list, you can simply refer to people by their first names within the notes.

You might also provide a brief description of the meeting's purpose. Write this from the mindset that others who don't know your team, who you are or what you do, etc., will be reading it. Don't just write "Doc sync" but rather "Doc sync with partner engineers for the ACME project to identify doc needs and issues as well as coordinate review and publishing."

How long should the meeting notes be?

Remember that others will lack the context of those in the meeting. Even those in the meeting might not have been fully following the discussion and details. Take the time to fill in the blanks. This might mean the meeting notes consist of longer, more narrative summaries. Make the content skimmable with lots of subheadings and short paragraphs. As a tech writer, you know how to structure content so that it's readable. Use a subheading for every 1-2 paragraphs of content.

Sometimes people provide stenographic-like notes of a meeting, noting what everyone said in a play-by-play fashion. Avoid this approach and instead prefer more concise summaries of the issues and discussion. In short, make it readable.

How do you find the time for this?

If you grabbed time from half a dozen peoples' calendars, you can probably spend 30 minutes post-meeting doing the write-up by yourself. If you're busy, you can postpone the task 1-2 days later. The write-up doesn't need to be a masterpiece, but you'll probably polish and refine the content more than a non-writer would. Just don't obsess. Keep it simple and recognize that meeting notes reflect the discussion of the meeting, and that meeting might have been somewhat disjointed and scatterbrained, touching a lot of different topics and points. A write-up that reflects the meeting might need a bit of re-organization and artificial structure to be readable.

Also, recognize that the first few write-ups are the most time-consuming (as with any activity). Once you get into a rhythm and style, the write-up becomes fairly easy. One way to find time is to recognize the value of doing the write-up in the first place. The write-up is a great way to take a closer examination of the meeting and make sure you've accounted for all action items and other loose ends. The write-up reinforces that you take the meeting seriously, and that it's worth everyone's time to attend.

Overall, strike a balance between context and concision that makes sense time-wise. Expect that readers will likely glance through the notes and see if anything merits a closer read.

How do you decide on who to include in the broadcast?

Who you include in the notes broadcast is how you "manage up," as they say. This is where broadcasting meeting notes becomes strategic. For example, suppose you have a channel where many groups post their meeting notes and the business stakeholders read them to stay updated across these groups. This means you'll have some readers who are 2-3 levels above your team. What do you want them to know about your team? About documentation? Raise issues and help them get a better sense of your challenges, successes, and other matters.

For example, suppose you faced challenges in figuring out the context for a new documentation topic. Maybe the documentation covers a new API that doesn't fit into any specific product's docs. This product ambiguity is causing confusion in the organization of docs. Maybe a senior leader can share some insights that would help clarify the product direction and fit with other content. And if a senior leader starts chiming in with insights, they'll suddenly become aware of the docs, what they contain, and that you've been working on components relevant to the user experience for the API.

Besides influencing up, you can also include parallel teams (especially those outside your immediate organization) to influence across. Other writer groups might have more insight into processes and techniques that could help you with the issues you face. The challenges your group encounters are likely similar to issues that other writer groups have faced (and maybe solved). Others might have been around much longer and have solutions, so loop them in if appropriate.

How can you leverage post-meeting participation from non-meeting participants?

Ideally, you want to allow those who didn't attend the meeting to add their thoughts. This is how sharing meeting notes can create a boomerang of information coming back to you and allow you to leverage the wisdom of others. If you keep your meeting notes in Google Docs or other documents that others can comment on, it makes it easy for them to jump in with some comments. Add a short note in the email that invites others to comment on the issues, explicitly inviting them to participate. Sometimes those who didn't attend might feel they shouldn't "butt in" to comment on a meeting they weren't a part of, but if you make the invite to participate more explicit, they might be more inclined.

Are there downsides to sharing issues, frustrations, or problems with others? Won't that make us look bad/incompetent?

You might think that sharing issues, frustrations, or problems you're running into isn't a strategic way to communicate your expertise at authoring and publishing. Suppose you have a laundry list of technical challenges, behavioral hurdles, and organizational culture issues that you're dealing with. It can be hard to be transparent about these issues and to share them with others. We often want to communicate a facade of expertise.

However, I’ve learned that as a blogger, readers value transparency. We like to see what issues others are facing and how they’re approaching those issues. Reading a blog of someone who always knows how to handle every situation, who always knows the right answers for every problem, etc., isn’t a very interesting read. Dare to be human. That humanity endears readers to your side. In this light, think of your meeting notes almost as a sort of blog entry, detailing your forays against your enemies (that enemy might be processes, technologies, etc.).

What do you do with all of this visibility?

One consequence of sharing your meeting notes is that you become visible. It takes guts to share a meeting write-up with 50 people. Do that repeatedly, and pretty soon they come to know you. This is what happened with my blog, idratherbewriting.com. I rarely go to any tech comm event without running into people who know me (even if I don’t know them). Blogging has made me super visible. Is visibility a good thing? Why would you want to be visible, and what do you do with that visibility?

Visibility for the sake of visibility alone is vanity. Almost no introverted writer wants to be this visible, with everyone carefully seeing your every move and thought. I never wanted to be a “famous” technical writer, if such a description is warranted. However, with visibility comes *influence*. Your visibility gives you a platform to influence others in ways that those without the platform lack. This is why NBA players often talk about issues outside of basketball — because they recognize that their visibility gives them more influence with people. Their spotlight on national television gives them a platform that many have used to highlight racial injustices.

What will you use your influence for? You’ve earned the influence through your hard work, writing up meeting notes and daring to share them with broad groups. At some point, you’ll need to recognize that you’ve become an influencer and ponder what that means. With my blog, I’ve tried to influence the tech comm community to see API documentation as more than just generating reference material, and to move toward docs-as-code tooling. I’ve also tried to show the many interesting facets and angles of a “boring” tech comm career. But I admit, I don’t actively sit around thinking about influence. I just write articles or record podcasts about issues that matter to me, without thinking of the wider impact. Consequences related to influence naturally follow, which is what I suspect will happen when you broadcast your meeting notes over the course of a year or more.

At any rate, whether you’re conscious of influence or not, consider focusing on your strategic initiatives and other plans. Focus on the issues that truly matter to you, and let the stars align in whatever pattern naturally follows from that. Don’t sit around thinking like a rhetorician, carefully calculating what buttons to push to achieve the desired results. That assumes too much control over influence. But as you develop strategies that matter to your team and pursue them, these topics will naturally surface in your notes and others will become more aware of them. Just making your strategies more visible will help portray your tech comm group as more than just documentarians. You’ll be seen as analytic and strategic thinkers, engaged in important projects to improve the user experience.

Conclusion

As you think about influence, typing up your meeting notes becomes more than just a secretarial task. It becomes a content problem, with a user audience and potential opportunities to influence those around you.

Ensuring documentation coverage with each software release

Getting a good handle on your release process — such as understanding the cadence of releases, how features are tracked and tagged in different phases, and other checkpoints prior to the release signoff — is central to thriving in any documentation role. Providing doc coverage for each release ensures you don't accrue documentation debt, and it boosts user satisfaction for the new features being released.

Key details to understand about your release process

To integrate into your release process, try to get a better understanding of the following:

- **How to view the release roadmap**. The roadmap allows you to anticipate what's coming 6+ months down the line. Look for those big tickets that will require more than a few paragraphs of content. You want to avoid being ambushed one week before release with a last-minute request that puts you into crisis mode. Some tickets might require an afternoon of work, while others require integration into a team's meetings over the course of several weeks.
- **Understand the definition of done.** How do product teams qualify their feature for release? Surely QA must sign off on the feature, but is there a docs requirement prior to release as well? First, look for any definition of done embraced by your product group. Is documentation included? If not, look for ways to insert docs into the process, even if informally.
- **Where to see dashboards and tracking mechanisms.** What sites, tools, or documents are used to track what's planned for upcoming releases, and how do you adjust the toggles on these dashboards to look at the next few releases? Each company has its own tools for this, and they aren't always obvious to those outside of product management. Features might start as idea pitches before being moved to another tracking mechanism, each with their own statuses and workflows, timelines, checkpoints, and other details.

Checking for doc coverage

When you see the release items surface on the horizon, look through each item included in the release and assess whether you have doc coverage. If you have a goal for 100% doc coverage for each release, you could score the doc coverage against all items in the release. For example, suppose you have six new features being released in the next version. Can you indicate a changelist or other commit that covers each of the features being released? When you start tracking doc coverage meticulously like this, it forces you to account for each necessary update.

What needs docs?

What items in the release actually need docs? Basically, if there's a release, it means something changed. These changes usually need documentation. However, sometimes the changes are backend fixes and optimizations that don't affect users. Maybe the dev team spent a month fixing technical debt around spaghetti code, and there's no need to document anything for external users. If so, great, you can avoid doc coverage for that release item.

Another scenario that might not need docs are bug fixes that don't change any existing functionality. Release notes often include a list of bug fixes, and if the fixes simply address glitches in the code, they might not need docs either.

But by and large, most items in the release should point to some page in the documentation. Release notes are meant to be short teasers for content that is elaborated upon in the pages of your documentation. For example, release notes for a feature might have a one paragraph summary with a link to a documentation page that expands in more detail. A good-looking release notes page will have a lot of paragraphs with “For more information, see …” links.

Teaching others about the need for docs

As you ensure doc coverage with each release, you’ll be reaching out to product teams and others to ask questions about docs. Each time you reach out, you reinforce the need for docs, teaching others about how the doc process works. This helps product teams understand your role and anticipate the need for docs in the future in advance.

In short, you build visibility, rapport, and awareness of your role (almost like marketing for your group). This is partly why inserting yourself in the release process as a checkpoint, even if informally, helps your tech comm group become more integrated with product teams.

Expanding the doc coverage with a doc quality measure

When analyzing whether you have adequate docs for the release, another question to consider is how much doc coverage for a feature is actually needed. What if a new feature only has a few inline comments in a code file? Does that count as documentation? In the scope of documentation, a feature might merit anything from a brief description to a full-fledged tutorial, sample app, walk-through code, extensive step-by-step tasks, and more.

Here’s where you have some options for a doc coverage goal. You could expand the doc coverage with an additional category of doc quality. For each feature covered by docs, you could score the documentation with a quality rating. As a tech writer, you typically know when docs are good or not. For example, did you personally test the docs, or just copy and paste content an engineer gave you? Does the documentation describe the request and response in a complete way? Are the step-by-step tasks described with enough detail for users to make their way through the docs?

I know that the features I personally test usually end up with much better documentation. While it might seem like a given that tech writers test all the features they document (p. 342), that’s hardly the case. The setup and execution of the features often requires extensive engineering knowledge and expertise. If you’re working in dev docs, testing each feature is a feat unto itself.

Conclusion

Integrating into the release process helps build the foundation for a healthy, aware documentation group that is plugged into product team releases and roadmaps. It helps you avoid documentation debt as well as stay looped in early for larger features requiring more advanced planning. The more you ensure doc coverage for each release, the more others become aware of your role with documentation and the need for docs in the future.

Chapter 13: Metrics and measurement

Metrics and measurement addresses ways to measure API documentation quality and how to track your progress on improvement. You can use the quality checklist here to review essential components of documentation and decide how your API docs measure up. The checklist can be a way to investigate, analyze, and interrogate your documentation from another perspective and discover ways to improve it.

Chapter sections:

Measuring documentation quality through user feedback 801
Different approaches for assessing information quality 804
Quality checklist for API documentation 814
Quantifying your progress 825

Measuring documentation quality through user feedback

As you set goals for your role or team, you might want to measure your impact on documentation quality in some way. The main reason for measuring your impact should be to evaluate your progress against documentation improvement goals. If you don't have any data to provide feedback on your efforts, it's hard to know if you're making a difference.

Also, metrics are essential for business reasons: upper management will invariably ask you for metrics of some kind (because what you can't measure, you can't manage). Metrics will also be key if you're making the case for a promotion (leveling up) or defending your performance during annual review time. Ultimately, though, you need metrics to answer this question: Is what I'm doing making a difference?

Despite the importance of metrics about documentation quality, they are an elusive, holy-grail type task that almost no one in the industry has nailed down. How do you know if your docs are any good? The first response people usually give is that you measure documentation quality through user feedback. In this section, I'll explain why trying to measure documentation by collecting user feedback is problematic.

The problem with doc surveys

The first inclination many have when measuring documentation quality (how good your docs are) is to conduct surveys with users. After all, it doesn't matter what the technical writer thinks of the documentation; it's what the user thinks, and how successful the documentation was in helping the user in some task, right?

To a degree, yes, you can only really evaluate the usefulness of documentation from the perspective of a user going through the documentation with some tasks in mind. However, surveys are a difficult, inefficient way to go about measuring documentation quality for several reasons:

- Survey responses aren't usually actionable
- Users aren't regularly available for surveys
- Survey questions are difficult to formulate
- Users are sometimes unreliable

Survey responses aren't usually actionable

First, feedback from users about documentation is often too general and unactionable to be helpful. If you do a satisfaction survey and find that only 30% of users would recommend the documentation to a friend, what can you do with that information? If surveys indicate that your docs are poor and need improvement, can you act on that? Which docs? Which parts need improvement? Are they "confusing and useless"? What's confusing or unhelpful about them? All of the docs, or just parts? Is the feedback a general impression overall, or focused on a specific user scenario from a single documentation topic?

Without more specifics, the information falls flat and doesn't lead to any actionable follow-up. If you can't identify the friction points and areas that need to be fixed, what good are the survey results? They might only serve to depress the technical writers.

In looking into ways to measure information quality, I found Yoel Strimling's article Beyond Accuracy: What Documentation Quality Means to Readers ↗ to be spot on in identifying the issues with surveys. Strimling says that questions in all types of feedback (whether asked in surveys, focus groups, face-to-face interactions, or other) need to be specific to be actionable. Echoing the conclusions of other researchers, Yoel explains:

> … it is not enough for us to collect meaningful feedback about our documentation from our readers. We also need to be able to use the information we collect to take actions that will help us directly address and prioritize the issues that are important to them.

In other words, surveys alone, without actionable feedback, aren't enough. A user might provide general feedback, saying that the docs felt like they needed polish, or that they had trouble understanding/following them, or that they seemed incomplete. But without more specifics, this feedback isn't all that actionable.

Users aren't available for surveys

Another problem with surveys is that most technical writers don't have direct and continued access to users to survey them on a regular basis. I wrote about the challenge of reconstructing the absent user ↗ in my series on Simplifying Complexity. It's unfortunate but the norm that technical writers have very little contact with their target users — not just access to users for surveys, but access to users through support interactions, training, presales, onboarding, or other situations. If users aren't available to survey, collecting their feedback becomes difficult.

Unbiased survey questions are difficult to formulate

Let's say you do have access to users, another challenge with surveys is how to formulate the questions. You have to be skilled in the way you pose survey questions to avoid bias in the responses. Strimling dives into research about survey formulation and lists the characteristics of a good survey. For example, questions must be "long enough to be useful, and short enough to encourage participation" (quoting Barnum). They need to be unambiguous, non-threatening, easily answerable, clear, unassuming, etc. Overall, survey questions are hard to formulate.

If you're a regular reader of my blog, you've probably seen a lot of small surveys I include with each article. I've learned that formulating survey questions is an art, and questions need to be simple, unambiguous, and easy to answer. At the same time, you have to consider what the actual goals of the survey are and what you plan to do with the feedback. Surveys are hard to do right.

Users are sometimes unreliable

Another problem with surveys is audience unreliability. We base the value of our docs around the user's perspective, but what if users are unreliable or unhelpful? There's a lot of variation, subjectivity, and unreliability with survey responses. Sometimes people use docs as a scapegoat for other issues related to poor product design, company dissatisfaction, or other frustrations. Often when I'm taking a survey, I base my responses on one part of the product or docs that I've been using, not on the whole. One bad topic can taint the user's response about the entire docs, distorting the quality measure.

Additionally, people who respond negatively in satisfaction surveys might not be your target audience. You might have been writing for an audience already familiar with a certain framework, but your survey might include someone new to the framework who found the docs worthless because they don't have the needed background. Other times, users might respond so hastily that their responses are cryptic and vague.

What can replace surveys?

I'm not entirely dismissive of surveys — I'm only being realistic. They have rarely worked for me. If the survey's purpose is to highlight the importance of documentation in general, e.g., how critical it is for complete and accurate documentation to help users complete tasks, great. But that's a different purpose than measuring documentation quality. A metric about the importance of docs would support resourcing for doc efforts or other goals for establishing tech comm's importance. But to make improvements with quality, you need information that is much more specific and actionable, and that's where general surveys tend to fall short.

Rather than sink too much time and energy into surveys as the only way to measure doc effectiveness, I propose a much more concrete and realistic approach to measuring doc quality: assessing the documentation against a detailed quality checklist. The checklist can take the general qualities of measurement (e.g., clarity, accuracy, relevance, findability, etc.) and make concrete what these abstract qualities mean and how they can be implemented.

Also, while many scholars have defined characteristics for information quality, I've yet to see a checklist specifically focused on developer documentation. In the sections that follow, I'll go into specific detail for describing information quality with developer docs, moving beyond high-level abstract definitions of quality to focus on more tangible, concrete ideas.

Continue on to the next section, Assessing information quality (p. 804).

Different approaches for assessing information quality

In the previous topic, Measuring documentation quality through user feedback (p. 801), I explained the challenges of getting feedback from user surveys as a way to measure documentation quality. In this section, I'll survey the landscape on criteria and rubrics for assessing documentation quality.

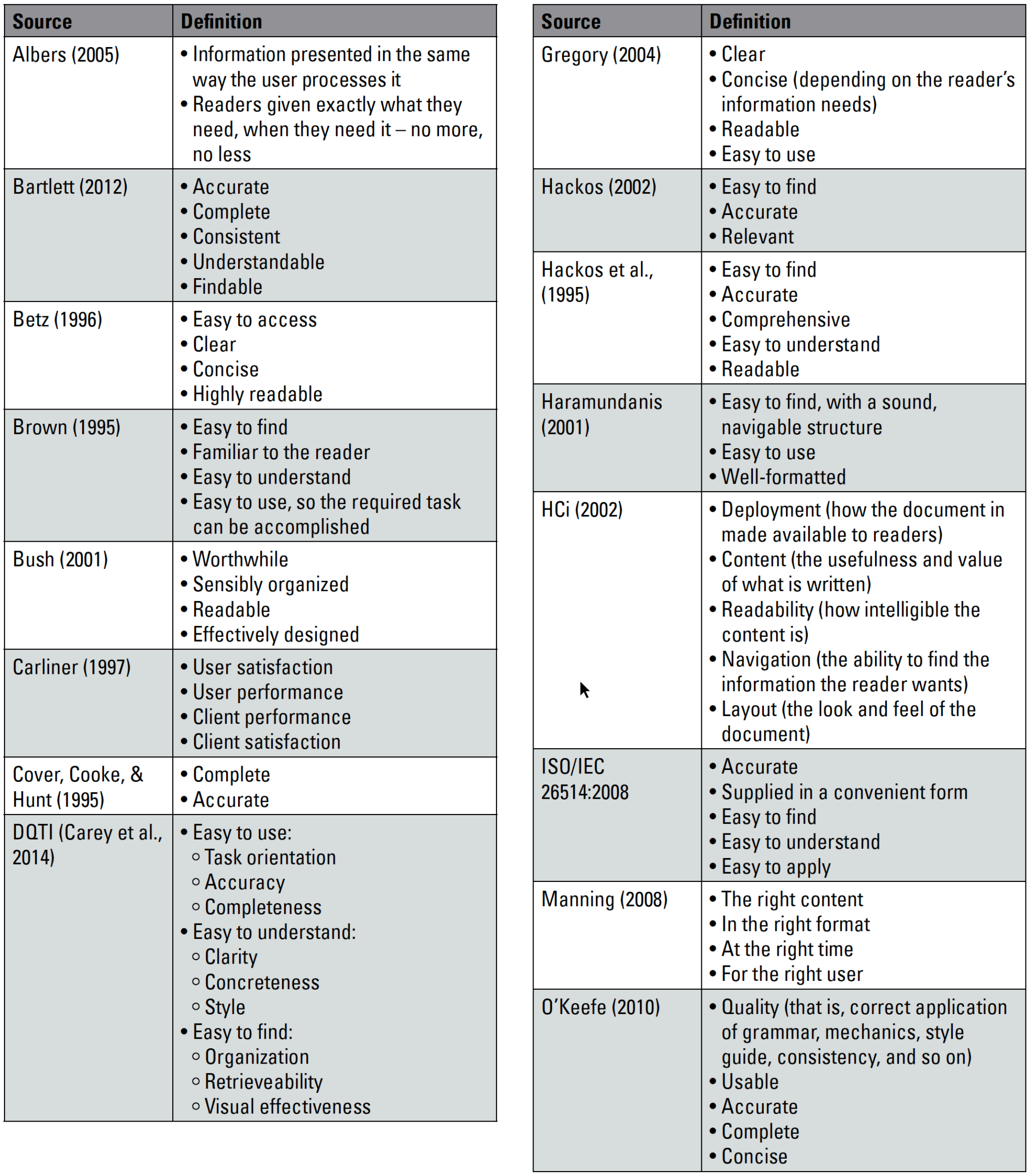
Common categories for information quality

Documentation quality is generally assessed against the following criteria:

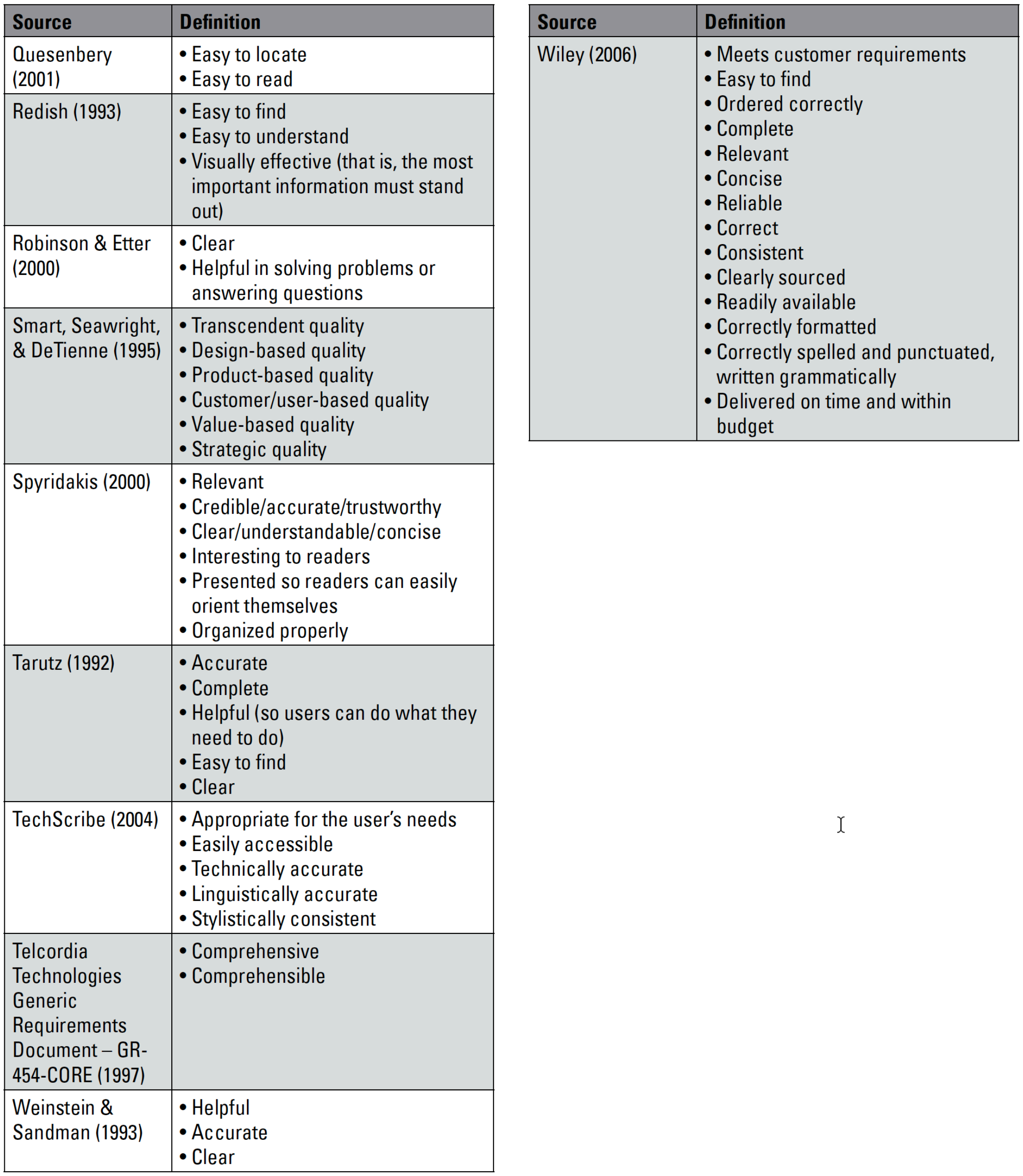
- Readability
- Clarity
- Context
- Accuracy
- Organization
- Succinctness
- Completeness
- Findability

In Beyond Accuracy: What Documentation Quality Means to Readers ↗, Yoel Strimling looks at previous research on the attempt to define information quality and finds a wide range of different quality definitions:

Table 3. Documentation quality definitions



Source	Definition
Albers (2005)	• Information presented in the same way the user processes it • Readers given exactly what they need, when they need it – no more, no less
Bartlett (2012)	• Accurate • Complete • Consistent • Understandable • Findable
Betz (1996)	• Easy to access • Clear • Concise • Highly readable
Brown (1995)	• Easy to find • Familiar to the reader • Easy to understand • Easy to use, so the required task can be accomplished
Bush (2001)	• Worthwhile • Sensibly organized • Readable • Effectively designed
Carliner (1997)	• User satisfaction • User performance • Client performance • Client satisfaction
Cover, Cooke, & Hunt (1995)	• Complete • Accurate
DQTI (Carey et al., 2014)	• Easy to use: ◦ Task orientation ◦ Accuracy ◦ Completeness • Easy to understand: ◦ Clarity ◦ Concreteness ◦ Style • Easy to find: ◦ Organization ◦ Retrieveability ◦ Visual effectiveness
Gregory (2004)	• Clear • Concise (depending on the reader's information needs) • Readable • Easy to use
Hackos (2002)	• Easy to find • Accurate • Relevant
Hackos et al., (1995)	• Easy to find • Accurate • Comprehensive • Easy to understand • Readable
Haramundanis (2001)	• Easy to find, with a sound, navigable structure • Easy to use • Well-formatted
HCi (2002)	• Deployment (how the document in made available to readers) • Content (the usefulness and value of what is written) • Readability (how intelligible the content is) • Navigation (the ability to find the information the reader wants) • Layout (the look and feel of the document)
ISO/IEC 26514:2008	• Accurate • Supplied in a convenient form • Easy to find • Easy to understand • Easy to apply
Manning (2008)	• The right content • In the right format • At the right time • For the right user
O'Keefe (2010)	• Quality (that is, correct application of grammar, mechanics, style guide, consistency, and so on) • Usable • Accurate • Complete • Concise



Source	Definition
Quesenbery (2001)	• Easy to locate • Easy to read
Redish (1993)	• Easy to find • Easy to understand • Visually effective (that is, the most important information must stand out)
Robinson & Etter (2000)	• Clear • Helpful in solving problems or answering questions
Smart, Seawright, & DeTienne (1995)	• Transcendent quality • Design-based quality • Product-based quality • Customer/user-based quality • Value-based quality • Strategic quality
Spyridakis (2000)	• Relevant • Credible/accurate/trustworthy • Clear/understandable/concise • Interesting to readers • Presented so readers can easily orient themselves • Organized properly
Tarutz (1992)	• Accurate • Complete • Helpful (so users can do what they need to do) • Easy to find • Clear
TechScribe (2004)	• Appropriate for the user's needs • Easily accessible • Technically accurate • Linguistically accurate • Stylistically consistent
Telcordia Technologies Generic Requirements Document – GR-454-CORE (1997)	• Comprehensive • Comprehensible
Weinstein & Sandman (1993)	• Helpful • Accurate • Clear
Wiley (2006)	• Meets customer requirements • Easy to find • Ordered correctly • Complete • Relevant • Concise • Reliable • Correct • Consistent • Clearly sourced • Readily available • Correctly formatted • Correctly spelled and punctuated, written grammatically • Delivered on time and within budget

As you can see, defining information quality is a constant theme in tech comm research. While the characteristics are somewhat similar, they aren't described in the same way, and they are mostly general and abstract. Troubled by the lack of a unified approach to documentation quality, and by the slipperiness of terms and lack of consistency, Strimling asks, which of these qualities matter most to users?

Strimling aligns with researchers Richard Wang and Diane Strong (1996) because of the way their research aligns with the "voice of the data customer" and because of the soundness of their framework and research. Wang and Strong's research is the foundation for many other articles here as well. After describing 118 information quality dimensions, Wang and Strong boil them down to four main criteria. Yoel explains:

> Based on their categories, Wang and Strong (1996) concluded that high-quality data must be:
>
> - Intrinsically good
> - Contextually appropriate for the task
> - Clearly represented
> - Accessible to the consumer

You can read the original article by Wang and Strong in the *Journal of Management Information Systems* if you have access to it, or online here. These researchers made a pivot in how they measured data quality — rather than considering the accuracy of the information on its own, they looked to see *what aspects were important to users*, and factored that user perspective into the quality assessment. They explain:

> The salient feature of this research study is that quality attributes of data are collected from data consumers instead of being defined theoretically or based on researchers' experience.

In other words, you can't measure data quality (DQ) without analyzing what quality dimensions are important to users. In their research, they settled on the four criteria that Strimling summarized:

> 1. [Intrinsic DQ] The extent to which data values are in conformance with the actual or true values;
> 2. [Contextual DQ] The extent to which data are applicable (pertinent) to the task of the data user;
> 3. [Representational DQ] The extent to which data are presented in an intelligible and clear manner and
> 4. [Accessibility DQ] The extent to which data are available or obtainable.

Wang and Strong's emphasis is on data quality, not necessarily documentation. Building on Wang and Strong, Strimling identifies 15 different dimensions to documentation quality and then asks users to rate them by importance. He concluded that these four categories are most important to users: accurate, relevant, easy to understand, accessible. These criteria are based on the level of importance assigned to them by the readers who participated in his study. He proposes that you can measure quality by asking users these four questions:

> - Could you find the information you needed in the document?
> - Was the information in the document accurate?
> - Was the information in the document relevant?
> - Was the information in the document easy to understand?

Strimling says you can ask users these questions during various interaction points (doc surveys, training situations, support interactions, onboarding, and more). The questions aren't simply yes/no questions but would include follow-up questions asking for more details if problems are noted (you can see a sample survey here ↗). Similar to Wang and Strong's user-based DQ framework, these criteria aren't priorities from writers but rather from users. (Note: In later research, Yoel found that "completeness" might be more important than "relevance.")

The problem with abstract definitions

These four criteria seem like a solid way to evaluate documentation if you have a way to frequently interact with your users. But even if you could regularly survey your users, these abstract categories don't provide details about *how* you might go about making the information more clear, relevant, accurate, and findable.

In other words, these categories are too high-level and general to be more actionable. For example, what does it mean for something to be clear when you consider different audiences and varying technical backgrounds? Is well-written code clear even if doesn't have comments? The categories fail to specify *tactics and tools* for executing clarity, relevancy, accuracy, and findability. How do you make something more clear and relevant? What specific steps do you take? So even if you were to get feedback from a user who says that the documentation is not clear, is not relevant, and isn't easy to understand, it would be difficult to take any specific actions based on this feedback without the user unpacking the detailed reasons for which he or she felt this way.

If you're not a user (but rather a technical writer) trying to assess documentation through these four questions, the questions are also not helpful. They can't be fully answered by a non-user. For example, "Could you find the information you needed in the document?" Only the reader can answer this. "Was the information in the document relevant?" Again, only the reader can answer this, not the writer. "Was the information easy to understand?" Again, only the reader can answer this. So while these questions seem like a good approach, I'm not sure how useful they are.

How can we break away from the dependence on user surveys but still develop a method for quality based on the user's perspective? This is my central question in this section.

Fortunately, if we take the starting categories here (accessibility, accuracy, relevance, clarity), and we are confident that these attributes align with user priorities, then we only need to define how these attributes can be implemented in documentation in specific, concrete, and actionable ways. This is a point Strimling starts to make in So You Think You Know What Your Readers Want? ↗. He says, "In lieu of feedback, what we need is a proven model of how readers actually define documentation quality (DQ), which we can then use to ensure that what we produce is useful to our audience." The checklist that I'll define in the next section is a model that identifies specifics from these general qualities.

Other research

Before jumping into the rubric, let's survey the information quality landscape a bit more, as there are a few other sources worth mentioning. First, Pronovix, a company that specializes in creating developer portals, holds regular Developer Portal Awards ↗. As such, they provide general reasons why they rate some developer portals higher than others. For example, in What is the MVP for a Developer Portal? ↗ they write:

> We compiled a first list of questions that provides users with the information they might need while working with your API product:
>
> - What is this API?
> - How do I get started with this API?
> - What do I need to understand about this API?
> - How do I get X done with this API?
> - Do I know all the details of this API?
> - How do I use your API in Y?
> - Is somebody still working on this API?
> - Where do I go when I have a problem with this API?
> - How do I get access to this API?
> - Can I afford this API?
> - Can I trust this API?

When a user says the documentation is “unclear” or lacks “relevance,” it’s probably because the documentation does not address some of these questions. This is what I mean by being specific about how to make documentation clear without solely relying on survey feedback.

These bullet points are all good questions that one would expect documentation (or a developer portal) to cover. See these articles from Pronovix describing more best practices for documentation and developer portals:

- What goes into an award winning developer portal?
- How to Improve Developer Adoption and Onboarding
- The Best Developer Portals of 2020

Keep in mind that Pronovix’s focus is on developer portals, not standalone API documentation sites (they explain the difference here). As such, they place more emphasis on how users interact both inside and outside the documentation, such as getting API keys from an admin portal, checking service status pages, participating in a community, and more. Since most companies have multiple documentation sites, often aggregated in a portal, I think the emphasis on developer portals is actually more relevant than on documentation alone.

Also, unlike scholarly research, Pronovix looks for best practices and successful patterns in the field, without trying to justify their criteria based on research or from studies that objectively verify and rank these characteristics. Some standards they recommend include API explorers for interactivity, mechanisms to scan and locate reference material, site designs that inspires trust, clear use cases for the API, code samples available in multiple languages, frictionless onboarding, community integration, and more.

Another great resource is Nordic APIs. In 5 Examples of Excellent API Documentation (and Why We Think So), Thomas Bush evaluates 5 documentation sites based on these criteria:

- Authentication guide
- Quickstart guide
- Endpoint definitions
- Code snippets
- Example responses

Bush highlights reasons for admiring certain sites, noting that the lesson with Stripe is “don’t overdo it.” For Twilio, it’s “be beginner-friendly.” For Dropbox, it’s “cater to unique dev backgrounds.” For GitHub, it’s “save developer time wherever you can.” And for Twitter, it’s “be flexible with how you present information.”

Another Nordic API article, 7 Items No API Documentation Can Live Without ↗, discusses 7 essential components in API docs:

1. An Authentication Scheme
2. HTTP Call Type Definitions
3. Endpoint Definitions
4. URI Structures, Methods, and Parameters
5. Human Readable Method Descriptions
6. Requests and Examples
7. Expected Responses

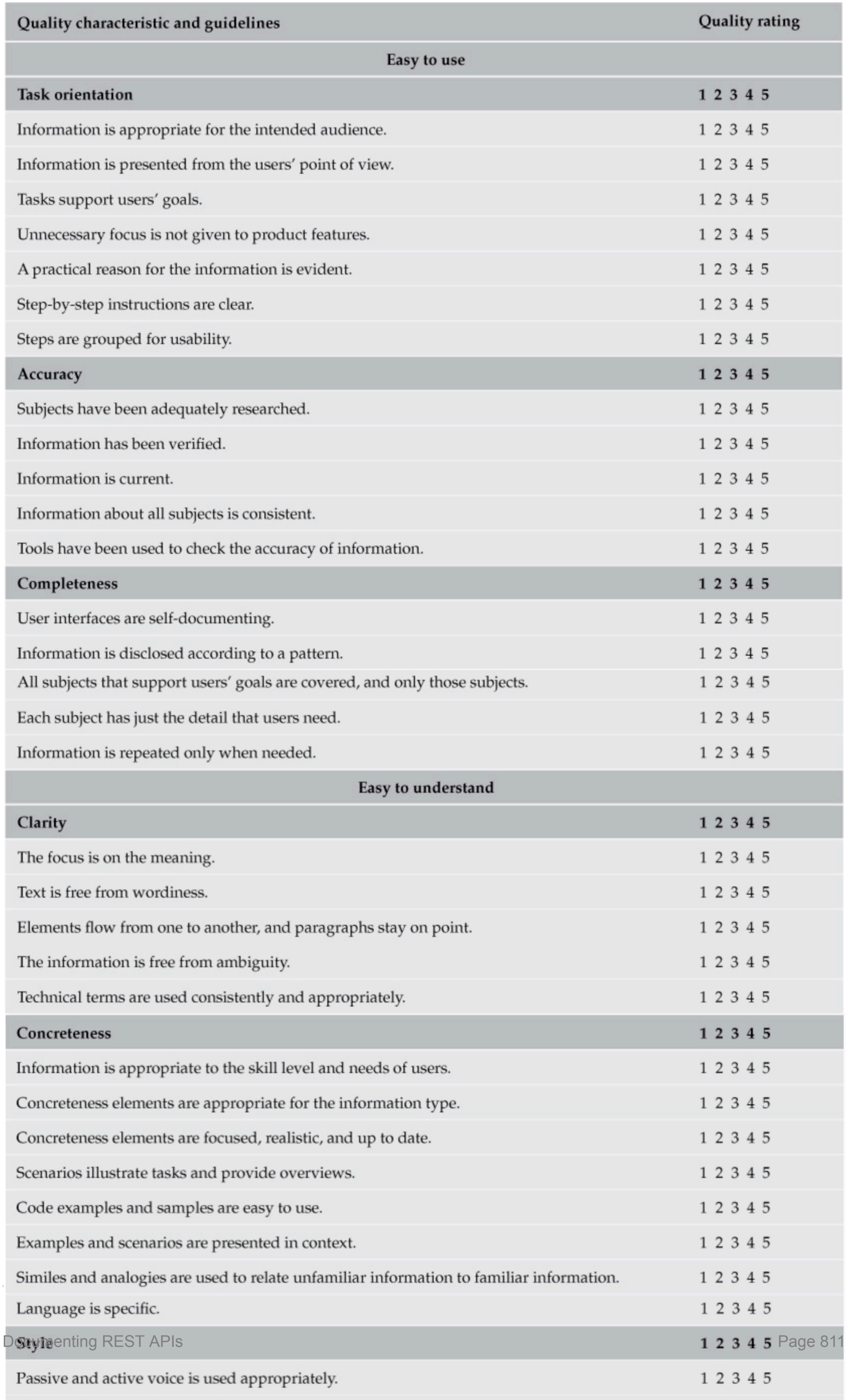
In my quality checklist (p. 814), I’ve listed each of these items but only briefly and generally. Sites like Nordic APIs ↗ and Pronovix ↗ provide more detailed guidance about how to optimize your documentation in each of these areas.

Technical writing handbooks

Another place to find quality checklists and guidance for implementing general characteristics like clarity, relevance, accuracy, etc., is in technical writing handbooks. In *Developing Quality Technical Information: A Handbook for Writers and Editors* ↗, the authors provide a mountain of detail for best practices. They divide their guidelines into these categories and subcategories:

- Easy to use
  - Task orientation
  - Accuracy
  - Completeness
- Easy to understand
  - Clarity
  - Concreteness
  - Style
- Easy to find
  - Organization
  - Retrievability
  - Visual effectiveness

In this model, you might score docs from 1 to 5 depending on how well the docs fulfill each characteristic. The textbook has a lot of examples and detail about how to go about each of these characteristics. There’s even treatment of code samples. Here’s the comprehensive quality checklist provided in the Appendix:



Quality characteristic and guidelines	**Quality rating**
Easy to use	
Task orientation	**1 2 3 4 5**
Information is appropriate for the intended audience.	1 2 3 4 5
Information is presented from the users' point of view.	1 2 3 4 5
Tasks support users' goals.	1 2 3 4 5
Unnecessary focus is not given to product features.	1 2 3 4 5
A practical reason for the information is evident.	1 2 3 4 5
Step-by-step instructions are clear.	1 2 3 4 5
Steps are grouped for usability.	1 2 3 4 5
Accuracy	**1 2 3 4 5**
Subjects have been adequately researched.	1 2 3 4 5
Information has been verified.	1 2 3 4 5
Information is current.	1 2 3 4 5
Information about all subjects is consistent.	1 2 3 4 5
Tools have been used to check the accuracy of information.	1 2 3 4 5
Completeness	**1 2 3 4 5**
User interfaces are self-documenting.	1 2 3 4 5
Information is disclosed according to a pattern.	1 2 3 4 5
All subjects that support users' goals are covered, and only those subjects.	1 2 3 4 5
Each subject has just the detail that users need.	1 2 3 4 5
Information is repeated only when needed.	1 2 3 4 5
Easy to understand	
Clarity	**1 2 3 4 5**
The focus is on the meaning.	1 2 3 4 5
Text is free from wordiness.	1 2 3 4 5
Elements flow from one to another, and paragraphs stay on point.	1 2 3 4 5
The information is free from ambiguity.	1 2 3 4 5
Technical terms are used consistently and appropriately.	1 2 3 4 5
Concreteness	**1 2 3 4 5**
Information is appropriate to the skill level and needs of users.	1 2 3 4 5
Concreteness elements are appropriate for the information type.	1 2 3 4 5
Concreteness elements are focused, realistic, and up to date.	1 2 3 4 5
Scenarios illustrate tasks and provide overviews.	1 2 3 4 5
Code examples and samples are easy to use.	1 2 3 4 5
Examples and scenarios are presented in context.	1 2 3 4 5
Similes and analogies are used to relate unfamiliar information to familiar information.	1 2 3 4 5
Language is specific.	1 2 3 4 5
Style	**1 2 3 4 5**
Passive and active voice is used appropriately.	1 2 3 4 5

Overall, there are 60 specific characteristics within the various categories. Why not simply adopt this quality checklist? It wouldn't be a bad approach, for sure. And the principles are so widely held that few would object to them. But I wanted something even more concrete, actionable, and focused on API documentation and developer portals.

Standards specifications

Another place to look at information quality would be standards such as the ASD-STE100. The ASD-STE100 standard was developed by the Aerospace and Defense Industries Association (ASD) to encourage Simplified Technical English (STE). STE consists of a dictionary of about 900 allowed words and a set of 65 writing rules intended to encourage more simplified English.

Another standard is ISO/IEC 26514:2008 - Systems and software engineering — Requirements for designers and developers of user documentation, which is a standard that "specifies the structure, content, and format for user documentation, and also provides informative guidance for user documentation style."

In the realm of documentation standards, there's also IEC/IEEE 82079-1 - Preparation Of Information For Use (Instructions For Use) Of Products - Part 1: Principles And General Requirements. Referencing an ISO standard might make your embrace of the standard more defensible. Embracing standards defined here would allow you to benefit from principles already debated, vetted, and finalized. If only the ISO publications were more accessible (e.g., without paywalls), these information resources could be much more valuable.

Another resource developed by SAP and later generalized and adopted by tekom is Standards and Guidelines for API Documentation, by Anne Tarnoruder. You can read a summary of the 68-page book in a tcworld article here: Standardizing API documentation. Tarnoruder emphasizes clear naming guidelines for APIs, noting:

> Names are the user interface of APIs. Meaningful, clear, and self-explanatory naming is a key factor in API's usability and adoption.

Technical writers might work with developers on names to ensure best practices with API design, especially regarding names. I covered some of these principles in my summary of Arnaud Lauret's book, *The Design of Web APIs* (p. 488). However, my focus here is more on documenting an API that has already been finalized rather than providing input on best practices for API design.

Tarnoruder's book provides comprehensive guidelines for writing the descriptions of API elements in the OpenAPI definition files, illustrated by examples. Tarnoruder also provides templates for REST and OData APIs, if you're not already using something like OpenAPI. And she provides detailed guidelines for documenting APIs such as Java with Javadoc.

For developer guides, Tarnoruder provides guidelines such as including "conceptual, setup, quick start and how-to information" and avoiding "implementation details irrelevant to users." This advice is fairly commonplace. However, more interesting, she also encourages writers to address both a code-first learning style and a concepts-first learning style. She writes:

> Various usability studies show that API documentation users differ in their learning preferences:
>
> Those with a top-bottom approach would first read all the conceptual topics, and only then start trying the API calls. Those who prefer a bottom-up approach would delve right into code samples to get a quick hands-on experience with the APIs. (Standardizing API documentation)

This is a pattern I described in How Developers Use API Documentation: An Observation Study (p. 444) and is based on research by Michael Meng, Stephanie Steinhardt, and Andreas Schubert in How Developers Use API Documentation: An Observation Study. The authors describe “opportunistic” behavior (people who learn by immediately trying out code and learning by trial and error) versus “systematic” behavior (people who start by carefully reading the manual before acting) as two common patterns of observed usage for developers using documentation. They encourage documentation to accommodate both learning styles.

Other sources for quality

Many other sources can inform documentation quality. For example, the Good Docs project aims to create templates that incorporate best practices. For example, by using the Overview template, you’ll automatically address the various questions and topics needed here. The project has templates for an overviews, quickstarts, reference material, discussions, how-to tasks, logging, tutorials, and more.

Another place to look for information quality is perhaps with information typing models (Information Mapping, DITA, and more). But I’ve already surveyed the landscape sufficiently here. My intent is not to exhaustively survey research on information quality. As Strimling’s earlier research pointed out, most people generally agree on the high-level categories. I want to instead provide specifics on implementation, especially for developer docs.

Continue on to the next section, Quality checklist for API documentation (p. 814), where I’ll list the details of my information quality checklist for developer docs.

Quality checklist for API documentation

This section continues from the previous page, Different approaches for assessing information quality (p. 804).

As indicated earlier, my goal is to create a practical guide for measuring quality. Instead of looking at docs against a list of general, abstract criteria, I recommend another approach: assessing docs against a list of characteristics that, if fulfilled, should lead to a high-quality user experience automatically. Each of the characteristics must be specific, actionable, and unambiguous in how it would be implemented in your docs. In this section, I'll present a comprehensive quality checklist for API docs and developer portals.

API documentation quality checklist

The following checklist is a checklist that involves a deep look at your docs. The checklist's criteria are in no particular order Also, the list shouldn't be seen as definitive or as a foolproof recipe for perfect documentation. Some points might apply more than others, depending on your product, domain, and audience. But overall, these are criteria/characteristics that will likely lead to a better experience with developer docs.

API documentation quality checklist (comprehensive version)

Findability

Findable in search. The content is indexed in a general search engine and findable when you create searches with the product name and some key tasks.

Release notes present. Release notes track changes to the product and documentation.

Site-specific search available. The doc site has a site-specific search that lets users search within the documentation site itself.

Developer portal hub context. If you have lots of documentation relevant to users, the documentation is organized in a developer portal of some kind.

UI pages link to docs. If there's a UI (such as with an administrative console), links within the UI connect users to the relevant documentation.

Main organization isn't an FAQ. The content doesn't have an endless FAQ with information that should have been integrated into more logical places in the document.

Version selection is available. If content has multiple versions, the versions are called out visibly in the topic and might have a selector or link allows users to navigate to the other versions.

Easy path to top 5 pages. There's an easy path for users to find the top 5 most-visited pages on the site. This requires you to look at metrics to determine these pages, and then assess the flow to those pages.

Topics have article-like length. Each topic consolidates much of the necessary information about the subject in a standalone, modular way. The topics don't fragment the subject into too many small topics that require the user to bounce around, losing context. At the same time, entire guides aren't shoved into single topics, making them eternally long. The length of topics should be about like a web article (for example, 500-3,000 words long).

Content follows task-based organization. Content is organized primarily by task and from a user's perspective rather than solely by doc type. The exception is that reference content is often consolidated on its own in an auto-generated output.

Developer portal has IA flow. The information architecture (IA) in the developer portal organizes content within a larger context or story, even if it's just a list of links under various categories.

Accuracy

Steps are accurate. The steps in the tasks accurately lead to the results promised by the task, without missing any details. For example, if the instructions say to click a button name, the button is named the same way in the interface. If the instructions say to use a class, the class is spelled as it appears in the code library, etc.

Code samples work. Code samples that can be copy and pasted actually work.

Content reviewed within past year. Content has been reviewed by a subject matter expert within the past year. Ideally, each topic should include metadata such as the last-reviewed timestamp, last author, and the group that owns the content.

Timestamps are visible. The documentation provides a visible timestamp of the last time it was edited so that users can gauge how current the documentation is.

No broken links. Links point to correct pages or are appropriately handled by redirects to equivalent pages.

Instructions are consistent. Information isn't repeated in confusing, redundant, or inconsistent ways. For example, the documentation doesn't explain how to do a task one way in Topic A but then a different way in Topic B. If content is re-used, the re-use is usually single-sourced to reduce inconsistency.

Screenshots are up to date. Screenshots and other visuals are up to date with the state of the interface (if there is one).

Deprecated features are noted. Features that are no longer supported (or which have been deprecated) are clearly noted as such in the documentation. Preferably, if a feature has been deprecated, a migration path to an alternative solution is provided.

App code matches doc code. Code in sample apps matches the code described in the documentation. The sample app hasn't evolved in ways that no longer match the documentation.

Relevance

Key use cases are documented. The documentation doesn't just provide reference information (e.g., auto-generated API documentation) but also explains how to use the API with tutorials guiding users through common use cases and journeys. The content should address the *most common* use cases intended for the product.

Code samples exist. Code samples showing sample ways to use the API (or similar tools) are provided. Ideally, the code samples are available in the user's target language. This might mean providing multiple code samples.

Support options noted. Options for contact or support are provided, even if the support merely involves posting to a peer-monitored forum.

Reference material is interactive. The docs allow users to interact with the functions, API calls, or other commands often using their custom data. Typically, with REST APIs, an API explorer allows users to try out requests using their own data. Other sites might use Jupyter Notebooks to run code that can be customized. Seeing responses with your own data can make the API seem more relevant.

Feedback options available. A feedback option is included in the docs so readers can relay feedback about their experience. When users provide feedback, they get the sense that someone is listening. This helps users feel more relevant.

Docs are published when features released. Documentation is available at the same time the feature is released and made available to users, not after the fact.

GitHub is actively monitored. If the product has a code repository on GitHub (or similar), this repository is actively monitored when users log issues or submit pull requests. Users aren't simply ignored.

Clarity

Product overview page answers "wh" questions. The overview explains the big picture and describes the problem that the tool or service addresses. Common who/what/

where/why questions are answered here.

Access and authorization explained. Details about how to get access, permissions, and authorization to use the API are provided. For example, this topic might cover how to authorize an API call with API keys.

Overview addresses use cases. The overview provides a high-level description of the main use cases or business objectives of the product. This allows users to get a sense of what the API is all about.

Overview has architectural diagram and explanation. The overview has a diagram of the main components and how they interact. This provides users with a glimpse of the whole.

Overview has index of assets that the product offers. If there's an SDK or developer kit that users can download, the contents of this download are described. This is similar to product instructions that start by identifying all parts that should have arrived in a package.

Subsystems have their own overview pages. For larger systems that might have multiple subsystems (e.g., groups of APIs for different scenarios), these subsystems have their own landing pages that resemble the higher-level overview (with use cases, diagrams, getting started links) but scoped to that specific subsystem.

Getting started tutorial exists. A getting started tutorial is provided for users to get started in an end-to-end way with the product, producing a sample output that builds their confidence. This topic might provide info on how to sign up, register, get API keys or permissions, and start using the API. (This topic might link to the authorization topic but is more comprehensive in scope. The purpose of this topic is frictionless onboarding.)

Sandbox environment available. A sandbox environment is available for users to try out the tool/system/API on their own and get a better understanding of how it works.

README file exists in code repo. If the product has a code repository, the repository contains a README file that provides information similar to the overview (but abbreviated), including information on how to contribute if applicable. (The README might simply link to the documentation overview but also include unique sections such as how to contribute in the repo.)

Tasks have prerequisites sections. Each task has a "Prerequisites" section that explains knowledge requirements, tool requirements, essential concepts, etc., necessary for completing the task.

Tasks have numbered steps or sections. Tasks are articulated through a series of specific, usually numbered steps or sections that reduce the complexity of the task. More complex tasks might require more granular and numerous steps or sections so as to reduce the complexity. (This is the primary way that technical writers simplify complexity.)

Sample app available. In more robust docs, a sample app might serve as a reference implementation for users.

Code sample has inline comments. Code samples are appropriately commented, either with inline comments, comments that preface or follow the sample, or both. Long code samples with no explanation aren't just dropped in.

Any role-based permissions are noted. If there are different roles and permission levels that correspond to varying capabilities, these roles are noted in all places they apply, such as within the individual API reference. For example, if only admin roles can use a certain endpoint, this is noted in the endpoint's reference and in any places that use the endpoint.

Steps are detailed rather than general. Specific steps for completing tasks are described in detail rather than glossed over in general ways — unless the task is something the audience is already familiar with. Overall, the documentation strikes a balance between providing enough detail for users to get the job done but not including too much detail that the reader is inundated with extraneous information.

Workflow maps connect topics in larger flows. Larger tasks that span multiple topics have a workflow map (or similar mechanism) that connects the topics together.

Visuals aid in areas of confusion. In places that are particularly confusing, appropriate visuals or additional examples have been added to reduce confusion.

Support cases are minimal for the product. Users can follow the documentation by themselves without relying on peer or customer support. A large number of support cases or forum threads containing questions that could have been answered in the documentation would indicate that the users aren't self-sufficient through the docs.

Technical level is appropriate to audience. The documentation's technical level is appropriate to the *target audience* but might not serve every possible audience (for example, total newbies to a programming language might struggle with documentation intended for developers already experienced in that language). Usually, general concepts in a programming language that you assume the audience knows are not explained in the documentation. Instead, your company's product, configuration, and usage are covered in the context of the programming language. One exception is when the implementation requires a non-standard process or workflow that merits some explanation.

Experiential learning paths are available. The documentation provides opportunities for experiential/opportunistic users to start learning immediately through code and trial/error, and for more systematic users to learn by reading concepts first.

Doc recommend the simplest path when multiple options exist. If there are multiple paths to a solution, the documentation focuses on the simplest path (though other possibilities might be briefly mentioned).

Docs call out relevant sections in a sample app. In cases where a sample app complements the documentation as a reference implementation, the documentation should refer to different aspects of the sample app.

Completeness

Reference docs follow industry standards. For native library APIs (or other API types), reference docs (auto-generated from source code comments) are available. This might mean Javadoc, Doxygen, OpenAPI outputs like Swagger or other reference docs specific to the library. The reference docs should be populated and generally follow tagging standards.

Parameter docs have complete info. Parameter documentation typically includes a description, data type, min/max values, sample values, and optional/required usage.

Reference content has consistent structure. Reference material such as APIs follow a common structure within each topic, mostly following a request-response type structure. Typical sections include descriptions, parameters, sample requests or usage, and sample responses.

Error messages are documented. Error messages that users can encounter are documented and discoverable through search. This supports the opportunistic/experiential user behavior.

Responses includes both sample and schema (REST APIs). The response documentation for REST APIs provides both a sample response and schema. The response provides an example of what might be returned, while the schema defines all possible elements that might be returned and describes attributes such as data types and whether the elements are required or optional in the response.

Troubleshooting section exists. The documentation has a troubleshooting section (either standalone or included within the section/topic it relates to) that provides information on how to solve common problems. The troubleshooting information indicates where things might go wrong and how to fix them. In other words, the documentation doesn't just describe the happy path.

Limitations/thresholds documented. If there are limitations and thresholds (for example, max number of API calls, max file sizes, max string lengths, max/min parameter values, max server load), these limitations are called out in the docs.

Locale limitations noted. If a feature is available only in certain contexts (locales, languages, platforms, roles, versions), that information is noted clearly in the feature. For example, an API that is only available for enterprise versions might have a label that says “Enterprise Version Only,” or if only available for a particular platform, might say “Linux Only” or the equivalent.

Unhappy paths are documented. If there are pitfalls or other traps, gaps, and gotchas to avoid, these are noted in the documentation rather than hidden from the user. A section called Known Limitations often contains this information. The documentation doesn’t lie or mislead the user but rather is transparent, honest, and helpful even if it means exposing the product’s warts and revealing problems users will like encounter.

Readability

Grammar isn’t distracting. Sentences are grammatically correct and read well, without distracting the user or calling attention to the language.

Placeholder text in code is visually apparent. In code samples, placeholder text that needs to be customized is clearly indicated to the user. It’s not confusing what is code and what needs to be changed, like `APIKEY`.

Sidebar nav has consumable organization at a glance. The sidebar navigation lets users take in a sense of the whole while also allowing users to expand more details as desired. The sidebar isn’t a massive list of seemingly endless scrolling and expansion + expansion + expansion but rather divides up doc sets into logical groups, like chapters in a book. For systems with large numbers of topics, progressive disclose techniques might be implemented across primary, secondary, and tertiary levels of information.

Sidebar nav highlights the current topic. As the user navigates each topic, the sidebar navigation makes it clear where the user is in the navigation (for example, the topic highlights clearly and the navigation sticks open at that level). Breadcrumbs might also help establish site context.

Context remains consistent when navigating. When a user clicks topics in the navigation, the UI doesn’t shift context in jarring ways, such as unexpectedly taking the user to another doc set or changing stable navigation areas like the sidebar and header (which should be consistent for every page). This jarring navigation often happens when sidebar entries point to topics in other doc sites. If this is the case, the external links have an icon indicating the link takes them to another site.

Doc types have consistent names across product docs. Common topics have similar names across doc sets in the developer portal. For example, the Overview, Getting Started, Troubleshooting, Glossary, Release Notes, and Reference are named consistently to help users understand how to navigate the site. One doc set shouldn’t call a topic “Latest updates” and “First steps” while another uses “What’s new” and “Quickstart.”

Subheadings are descriptive. The subheadings are descriptive enough to allow users to get a gist of the topic by merely reading the subheadings. The subheadings also follow a parallel structure so they can be more easily scanned and read.

Topics start with summaries. Each topic usually has a short summary below the title ↗ that encapsulates the main purpose of the topic. This helps users get a sense of the topic at a glance.

Glossary exists. Unfamiliar words and jargon are defined in a glossary ↗. At times, the glossary terms are linked to their glossary definitions.

Glossary entries match the actual terms used in the content. Glossary terms (as defined in the glossary) are actually used consistently across the documentation. For example, one doc set doesn’t use a certain term while another uses a synonym of the term, with the admin UI using yet another term. If the glossary lists a term for a particular concept, the documentation content consistently uses that term.

Code samples have proper formatting and highlighting. The formatting in code samples follows standard white spacing, line breaks, and other syntax for the language. Code syntax highlighting appropriate to the language has been applied to increase the code’s readability.

Responsive view presents content in a readable way. The content can be read on a mobile device (e.g., iPhone) in a usable way. For example, the responsive view allows users to navigate the sidebar links and view code samples.

Navigation mechanisms are consistent across docs. Navigation mechanisms work consistently across all docs in the developer portal. For example, in one set of docs, if top-level folders expand to show child items rather than opening to their own page, the same behavior is found in other docs.

Sentences and paragraphs are somewhat short. Sentences are somewhat short, paragraphs are relatively small, and subheadings are frequent. A readability score will place the content at the high-school level, not college.

Language uses active voice. The language uses active voice (where warranted) with clear subjects and verbs positioned closely together.

For a version of this checklist that is easy to copy and paste, see Quality checklist for API docs (simplified html) – comprehensive version (p. 0). This output strips away most formatting and just list the various criteria in a basic HTML file. Copy and paste the content into Google Docs or Microsoft Word. Then as you go through the content, make your notes in the "Assessment" area.

Is this list complete? No

The checklist has about 75 different characteristics that I've described. Did I miss anything? Sure, you could probably expand this list with 100 more characteristics.

However, there's also something to be said for a practical list of key points. The more criteria you add, the more laborious it becomes to evaluate docs. The most popular style guides (e.g., Strunk and White) are those that briefly cover the main points you need rather than providing an exhaustive list, especially if many guidelines are already followed.

Considerations as you evaluate your docs

As you evaluate your docs, consider the following:

- **Doc scope**: If you're working on a developer portal, chances are you don't own the entire portal. You might just own one little section of the portal. That's okay. You can limit your review to just the scope that you own. Granted, the user journeys might extend beyond this scope, but start with your stewardship first. The last thing you want to do is start a war with other authors by identifying all kinds of issues with their content (at least not before you address your own issues first).
- **Levels of assessment**: Another consideration is just how much you can assess without more familiarity with docs. You can't know if the steps are accurate unless you go through the steps. You can't know if the docs are consistent unless you've read all the documentation. You can't know if the code works unless you can run it in a test environment. It might take more than a year working with the docs to be able to make these kinds of assessments. So pick and choose the criteria that are appropriate for your level of familiarity with the docs.
- **Good docs can't fix bad design**: Poor API design will make even good docs problematic, no matter how well-written your content is. If the API has inconsistent naming, incomplete parameters, doesn't map to user journeys, and is cumbersome to use, then documentation also becomes more cumbersome to follow and implement. Good docs can't fix bad API design, though docs can try to salvage the user experience.

Short version of the API documentation quality checklist

Feedback I've received about the checklist is that it's too long — isn't there a lightweight version? Based on this feedback, I selected what I think are the highest priority criteria in each section. But again, as I've said elsewhere, my selections here are somewhat arbitrary and might depend on your particular product, user, and domain.

API documentation quality checklist (short version)

Findability

Findable in search. The content is indexed in a general search engine and findable when you create searches with the product name and some key tasks.

Release notes present. Release notes track changes to the product and documentation.

Accuracy

Steps are accurate. The steps in the tasks accurately lead to the results promised by the task, without missing any details. For example, if the instructions say to click a button name, the button is named the same way in the interface. If the instructions say to use a class, the class is spelled as it appears in the code library, etc.

Code samples work. Code samples that can be copy and pasted actually work.

Relevance

Key use cases are documented. The documentation doesn't just provide reference information (e.g., auto-generated API documentation) but also explains how to use the API with tutorials guiding users through common use cases and journeys. The content should address the *most common* use cases intended for the product.

Code samples exist. Code samples showing sample ways to use the API (or similar tools) are provided. Ideally, the code samples are available in the user's target language. This might mean providing multiple code samples.

Clarity

Product overview page answers "wh" questions. The overview explains the big picture and describes the problem that the tool or service addresses. Common who/what/where/why questions are answered here.

Access and authorization explained. Details about how to get access, permissions, and authorization to use the API are provided. For example, this topic might cover how to authorize an API call with API keys.

Completeness

Reference docs follow industry standards. For native library APIs (or other API types), reference docs (auto-generated from source code comments) are available. This might mean Javadoc, Doxygen, OpenAPI outputs like Swagger or other reference docs specific to the library. The reference docs should be populated and generally follow tagging standards.

Parameter docs have complete info. Parameter documentation typically includes a description, data type, min/max values, sample values, and optional/required usage.

Readability

Grammar isn't distracting. Sentences are grammatically correct and read well, without distracting the user or calling attention to the language.

Placeholder text in code is visually apparent. In code samples, placeholder text that needs to be customized is clearly indicated to the user. It's not confusing what is code and what needs to be changed, like `APIKEY`.

For the copy/paste version of this checklist, see Quality checklist for API docs (simplified html) — short version. Similar to the simplified form of the comprehensive version, this output strips away most formatting and just list the various criteria in a basic HTML file.

Quantifying your progress

This section continues from the previous page, Quality checklist for API documentation (p. 814).

Analysis and quantitative metrics

It's hard to imagine that documentation that checks all of the boxes in the quality checklist (p. 814) wouldn't also score highly with user satisfaction surveys. Can you honestly see any documentation that legitimately satisfies all of these criteria as falling short with users?

And yet, to achieve the level of information quality, we didn't have to rely on constant user surveys to gather feedback. By identifying best practices for content design (specifically for API/developer documentation), we're able to increase the documentation quality in more self-sufficient, self-directed ways.

Moving towards quantification

In my initial go-around with the quality checklist, I tried to move towards quantification by including scores and weights for each criteria, and then dividing the achieved number of points by the total points. From this scoring, I tried to move from qualitative to quantitative measurement.

However, in practice, I found that assigning scores for each section felt arbitrary and subject to personal whims. I don't think others found the scores meaningful either. Instead, just having a checklist of criteria to consider was valuable enough. That's why i stripped out the section on quantification in later revisions to the content — because I want the advice I give here to be helpful in practice, not just theory.

I'm not saying that some approach to quantifying documentation wouldn't work, just that my approach did not. Also, recognize that the quality checklist has no official data to support it — instead, these best practices come from experience in the industry and from best practices that I and others have observed within the realm of developer documentation.

This is likely the problem with my approach: who's to say that documentation needs each of these criteria to succeed? It's possible that documentation might still be findable, accurate, relevant, and clear without many of these more concrete components. I don't have any user-based research to say that docs should be this way, that they should have an overview, that reference material should follow a consistent structure, that tasks should be detailed in steps, or that error messages should be documented, etc. And honestly, I doubt any checklist can prove its objective value.

Remember that user surveys (p. 801), which I criticized earlier as problematic, should both complement and confirm the criteria in the checklist. User surveys specifically for docs that rate highly with the quality checklist should also rate higher in satisfaction surveys than surveys for docs that score more poorly. But again, to establish a kind of correlation through surveys relies on a host of factors (objective, unbiased, unambiguous survey questions from a large sample of a representative audience across domains), which is likely difficult to pull off on a regular basis.

Overall, I am confident that few would object to most of the criteria in the checklist (p. 814). Most of the checklist's criteria would be agreed upon by both readers and writers with enough common ground as to be a practical guide for improving documentation quality. Also, the criteria should be seen as a first draft, a starting point that can be refined and improved, checked against industry standards, confirmed against docs that are loved by users, refined through constant feedback, and more.

Chapter 15: AI tools and API documentation

AI tools can help speed up authoring, demystifying code, automating publishing tasks, populating templates, and more. This section will look at how to use AI tools within the context of API documentation.

This section is under construction, as I'm working on the content right now. Expect for the content to be complete probably by end of year. Currently, the PDF contains only the available sections.

Chapter sections:

First look at the Oxygen XML AI Positron Assistant 827
AI document engineering with pattern-based prompts 842
Using AI tools to build, stage, and publish API reference docs 851
Using AI to learn coding 856
Using AI for language advice 860
Using AI to create doc updates based on bugs 864
Using AI For thematic analysis 870
Using AI for comparison tasks with API responses 879

First look at the Oxygen XML AI Positron Assistant

The Oxygen XML Positron Assistant lets you use AI tools inside Oxygen XML to help with a variety of writing tasks, such as writing short description elements, correcting grammar, improving readability, adding index terms, and more. Positron hooks into an AI provider (currently ChatGPT 3.5) to pass your topic content to the AI with a specific instruction. It then returns the content and allows you to preview the diff, seeing what has changed and inserting the modified text in place. By integrating directly with your project, Positron helps you use AI when and where you need it, without switching contexts or resorting to external tools.

In this post, I'll explain what Positron can do, how it works, and include my thoughts and analysis along the way. Although this is a sponsored post, I'll still include my own perspective about the uses of Positron and, more generally, AI. For Positron documentation, see Oxygen AI Positron Assistant Add-on.

Note: This is a sponsored post.

Level-setting expectations

When ChatGPT first launched, many thought roles like technical writer would soon be extinct because ChatGPT could simply write the documentation. What strengthened this perception was the lack of sources for ChatGPT responses. It seemed like ChatGPT was an uber-intelligent machine that had learned all human knowledge and could render it in an infinite variety of ways.

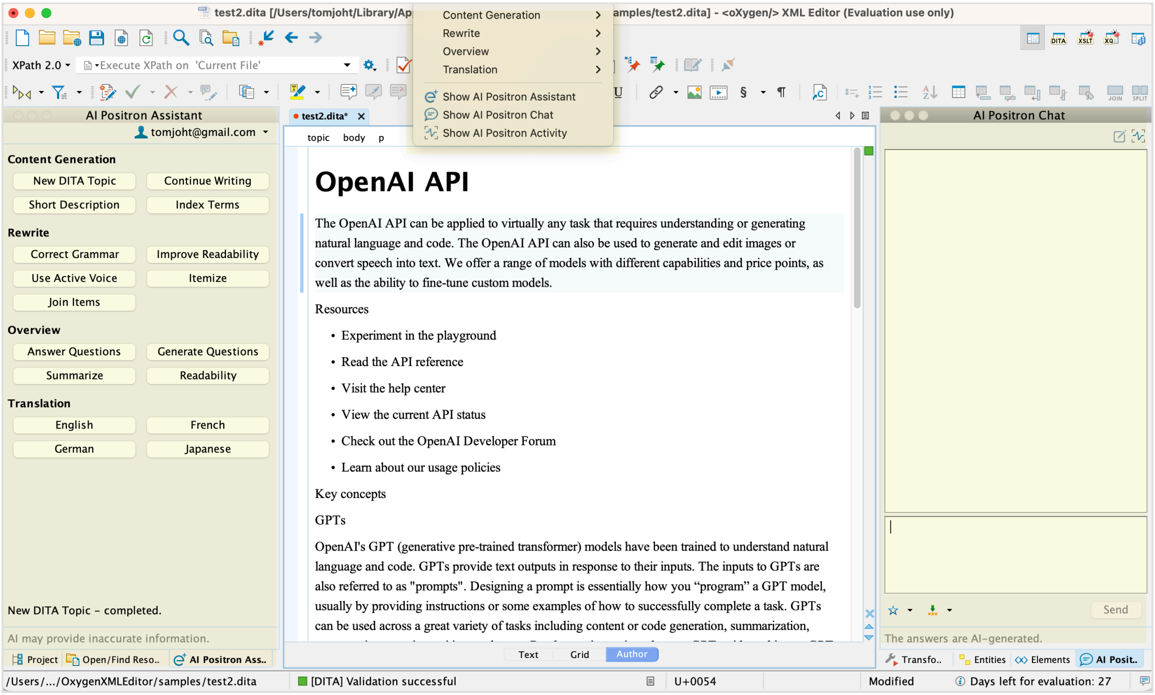
The enthusiasm around AI-generated content has cooled off as some realizations have settled in. If ChatGPT doesn't have good sources, its responses are poor. For websites that have trained AI on pages within their site only, if those sites lack strong content, the bots routinely come up empty handed. AI tools don't invent content from thin air. If you try to use ChatGPT to write your documentation for you, without feeding it informative, accurate content, you end up with made-up, cliche, pseudo-explanatory content.

The more I work with AI, the more I think that raw content generation isn't the best use of these tools. Instead, AI tools work better when you feed it some content and ask it to perform a specific task, such as correcting grammar, extracting index terms, or structuring the content into a specific pattern/format. For example, instead of generating entire sections from scratch, ask AI to rewrite a few confusing sentences.

Positron gives you tools for this more surgical approach to content development. While Positron is already built with the most common prompts for ChatGPT, you can also extend it with your own. What I like best about Positron is that it provides a diff of the response, letting you see what ChatGPT changed. You can accept the response entirely or pick and choose what you want to accept. The diff tool is actually my favorite feature in Positron!

Get started

To get started with Positron, see the Manual Installation section in the Positron documentation. Note that Positron is supported only in versions 25.1 and later of Oxygen XML. After activating the extension, an AI menu appears at the top. From the AI menu, select **Show Positron Assistant**.



New DITA Topic – completed.

AI may provide inaccurate information.

The answers are AI-generated.

[DITA] Validation successful

OpenAI API

The OpenAI API can be applied to virtually any task that requires understanding or generating natural language and code. The OpenAI API can also be used to generate and edit images or convert speech into text. We offer a range of models with different capabilities and price points, as well as the ability to fine-tune custom models.

Resources

- Experiment in the playground
- Read the API reference
- Visit the help center
- View the current API status
- Check out the OpenAI Developer Forum
- Learn about our usage policies

Key concepts

GPTs

OpenAI's GPT (generative pre-trained transformer) models have been trained to understand natural language and code. GPTs provide text outputs in response to their inputs. The inputs to GPTs are also referred to as "prompts". Designing a prompt is essentially how you "program" a GPT model, usually by providing instructions or some examples of how to successfully complete a task. GPTs can be used across a great variety of tasks including content or code generation, summarization,

The AI menu appears after you activate the AI Positron Assistant

An "AI Positron Assistant" pane appears on the left with buttons for quick actions. On the right, an "AI Positron Chat" pane shows ChatGPT's responses.

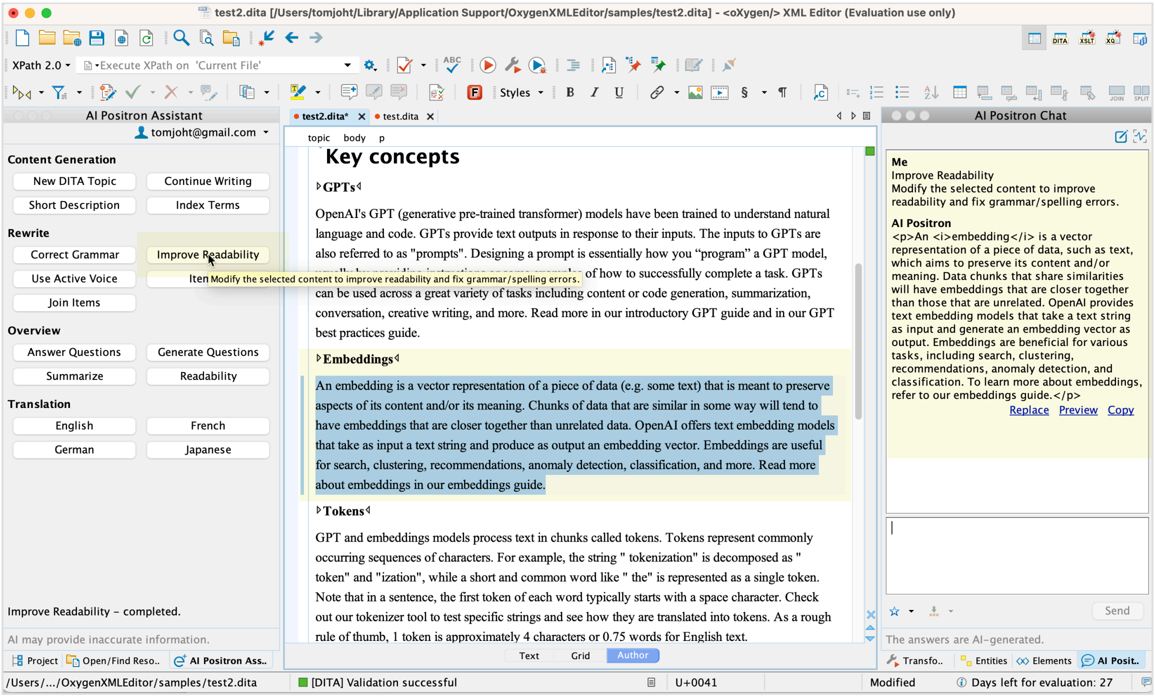
Sample workflow

The general workflow with Positron involves selecting text, then clicking an action button in the AI Positron Assistant pane. For example, suppose you're trying to explain a confusing concept. In this scenario, pretend you're a tech writer for OpenAI explaining the concept of "embeddings." You asked engineers what an embedding is, but their response was confusing. They wrote,

> An embedding is a vector representation of a piece of data (e.g. some text) that is meant to preserve aspects of its content and/or its meaning. Chunks of data that are similar in some way will tend to have embeddings that are closer together than unrelated data. OpenAI offers text embedding models that take as input a text string and produce as output an embedding vector, classification, and more.

(Note: I just pulled this paragraph randomly from the OpenAI documentation here.)

Dissatisfied with the definition's clarity, you select the text and click **Improve Readability** in the AI Positron Assistant pane:



Modify the selected content to improve readability and fix grammar/spelling errors.

Improve Readability – completed.

AI may provide inaccurate information.

Key concepts

GPTs

OpenAI's GPT (generative pre-trained transformer) models have been trained to understand natural language and code. GPTs provide text outputs in response to their inputs. The inputs to GPTs are also referred to as "prompts". Designing a prompt is essentially how you "program" a GPT model, of how to successfully complete a task. GPTs can be used across a great variety of tasks including content or code generation, summarization, conversation, creative writing, and more. Read more in our introductory GPT guide and in our GPT best practices guide.

Embeddings

An embedding is a vector representation of a piece of data (e.g. some text) that is meant to preserve aspects of its content and/or its meaning. Chunks of data that are similar in some way will tend to have embeddings that are closer together than unrelated data. OpenAI offers text embedding models that take as input a text string and produce as output an embedding vector. Embeddings are useful for search, clustering, recommendations, anomaly detection, classification, and more. Read more about embeddings in our embeddings guide.

Tokens

GPT and embeddings models process text in chunks called tokens. Tokens represent commonly occurring sequences of characters. For example, the string " tokenization" is decomposed as " token" and "ization", while a short and common word like " the" is represented as a single token. Note that in a sentence, the first token of each word typically starts with a space character. Check out our tokenizer tool to test specific strings and see how they are translated into tokens. As a rough rule of thumb, 1 token is approximately 4 characters or 0.75 words for English text.

Me
Improve Readability
Modify the selected content to improve readability and fix grammar/spelling errors.

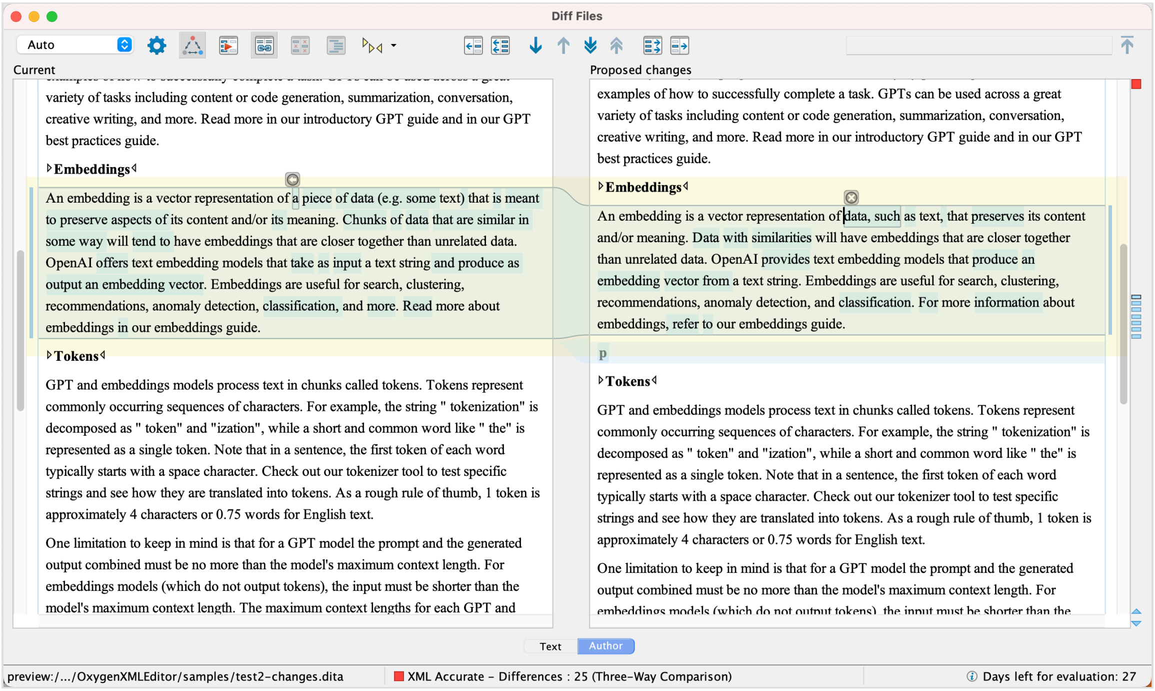
AI Positron
`<p>An <i>embedding</i> is a vector representation of a piece of data, such as text, which aims to preserve its content and/or meaning. Data chunks that share similarities will have embeddings that are closer together than those that are unrelated. OpenAI provides text embedding models that take a text string as input and generate an embedding vector as output. Embeddings are beneficial for various tasks, including search, clustering, recommendations, anomaly detection, and classification. To learn more about embeddings, refer to our embeddings guide.</p>`

The answers are AI-generated.

/Users/.../OxygenXMLEditor/samples/test2.dita [DITA] Validation successful U+0041 Modified Days left for evaluation: 27

The general workflow with Positron is to select text, then choose an action in the left AI Positron Assistant pane

The AI-generated response appears in the right pane. In this example, there don't seem to be many changes. To see a diff of what changed, click the **Preview** link below the modified text. This launches the diff editor, showing the changes:



Current:

variety of tasks including content or code generation, summarization, conversation, creative writing, and more. Read more in our introductory GPT guide and in our GPT best practices guide.

Embeddings

An embedding is a vector representation of a piece of data (e.g. some text) that is meant to preserve aspects of its content and/or its meaning. Chunks of data that are similar in some way will tend to have embeddings that are closer together than unrelated data. OpenAI offers text embedding models that take as input a text string and produce as output an embedding vector. Embeddings are useful for search, clustering, recommendations, anomaly detection, classification, and more. Read more about embeddings in our embeddings guide.

Tokens

GPT and embeddings models process text in chunks called tokens. Tokens represent commonly occurring sequences of characters. For example, the string " tokenization" is decomposed as " token" and "ization", while a short and common word like " the" is represented as a single token. Note that in a sentence, the first token of each word typically starts with a space character. Check out our tokenizer tool to test specific strings and see how they are translated into tokens. As a rough rule of thumb, 1 token is approximately 4 characters or 0.75 words for English text.

One limitation to keep in mind is that for a GPT model the prompt and the generated output combined must be no more than the model's maximum context length. For embeddings models (which do not output tokens), the input must be shorter than the model's maximum context length. The maximum context lengths for each GPT and

Proposed changes:

examples of how to successfully complete a task. GPTs can be used across a great variety of tasks including content or code generation, summarization, conversation, creative writing, and more. Read more in our introductory GPT guide and in our GPT best practices guide.

Embeddings

An embedding is a vector representation of data, such as text, that preserves its content and/or meaning. Data with similarities will have embeddings that are closer together than unrelated data. OpenAI provides text embedding models that produce an embedding vector from a text string. Embeddings are useful for search, clustering, recommendations, anomaly detection, and classification. For more information about embeddings, refer to our embeddings guide.

Tokens

GPT and embeddings models process text in chunks called tokens. Tokens represent commonly occurring sequences of characters. For example, the string " tokenization" is decomposed as " token" and "ization", while a short and common word like " the" is represented as a single token. Note that in a sentence, the first token of each word typically starts with a space character. Check out our tokenizer tool to test specific strings and see how they are translated into tokens. As a rough rule of thumb, 1 token is approximately 4 characters or 0.75 words for English text.

One limitation to keep in mind is that for a GPT model the prompt and the generated output combined must be no more than the model's maximum context length. For embeddings models (which do not output tokens), the input must be shorter than the

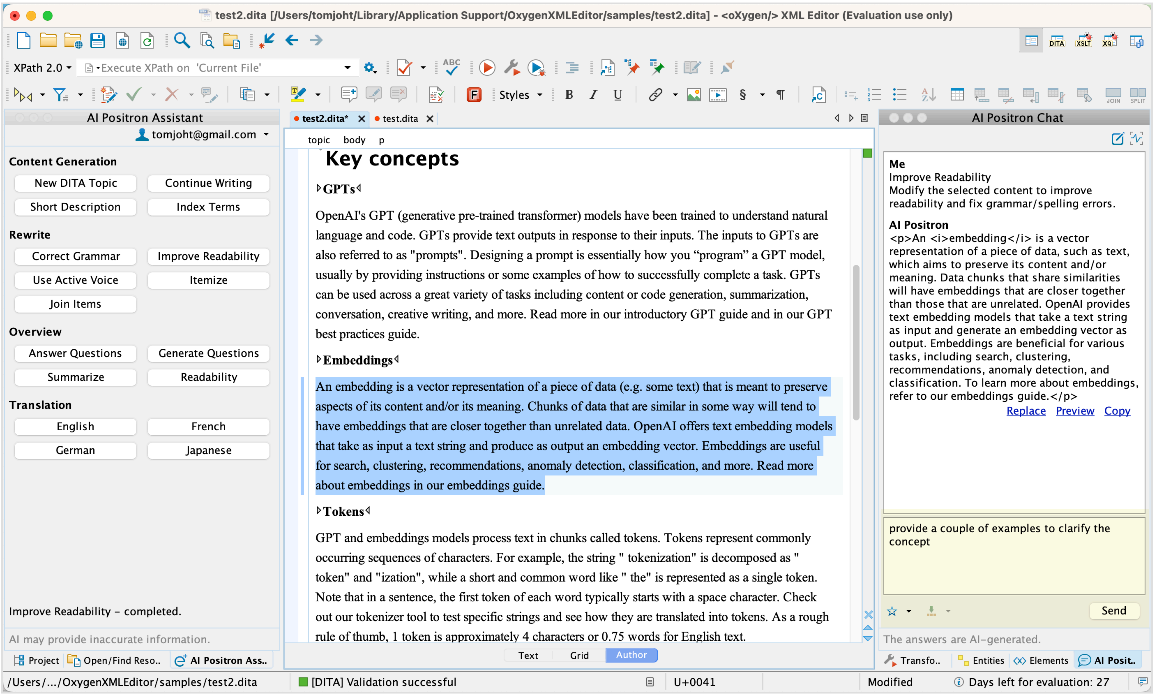
preview:/.../OxygenXMLEditor/samples/test2-changes.dita XML Accurate – Differences : 25 (Three-Way Comparison) Days left for evaluation: 27

The diff editor lets you preview what changed in the text Positron modified

The diff shows all the language changes. You can see the AI adjusted more text than it initially seemed. I LOVE the diff preview!

But this AI modified text isn't that helpful. You still don't have a clear sense of what an "embedding" is. The problem wasn't with poor wording but rather inadequate explanation. You could go back to the engineers for more details, but you don't want to look slow.

You decide to further refine the AI prompt. In the right pane, you add this instruction: "provide a couple of concrete examples to clarify the concept."



Key concepts

GPTs

OpenAI's GPT (generative pre-trained transformer) models have been trained to understand natural language and code. GPTs provide text outputs in response to their inputs. The inputs to GPTs are also referred to as "prompts". Designing a prompt is essentially how you "program" a GPT model, usually by providing instructions or some examples of how to successfully complete a task. GPTs can be used across a great variety of tasks including content or code generation, summarization, conversation, creative writing, and more. Read more in our introductory GPT guide and in our GPT best practices guide.

Embeddings

An embedding is a vector representation of a piece of data (e.g. some text) that is meant to preserve aspects of its content and/or its meaning. Chunks of data that are similar in some way will tend to have embeddings that are closer together than unrelated data. OpenAI offers text embedding models that take as input a text string and produce as output an embedding vector. Embeddings are useful for search, clustering, recommendations, anomaly detection, classification, and more. Read more about embeddings in our embeddings guide.

Tokens

GPT and embeddings models process text in chunks called tokens. Tokens represent commonly occurring sequences of characters. For example, the string " tokenization" is decomposed as " token" and "ization", while a short and common word like " the" is represented as a single token. Note that in a sentence, the first token of each word typically starts with a space character. Check out our tokenizer tool to test specific strings and see how they are translated into tokens. As a rough rule of thumb, 1 token is approximately 4 characters or 0.75 words for English text.

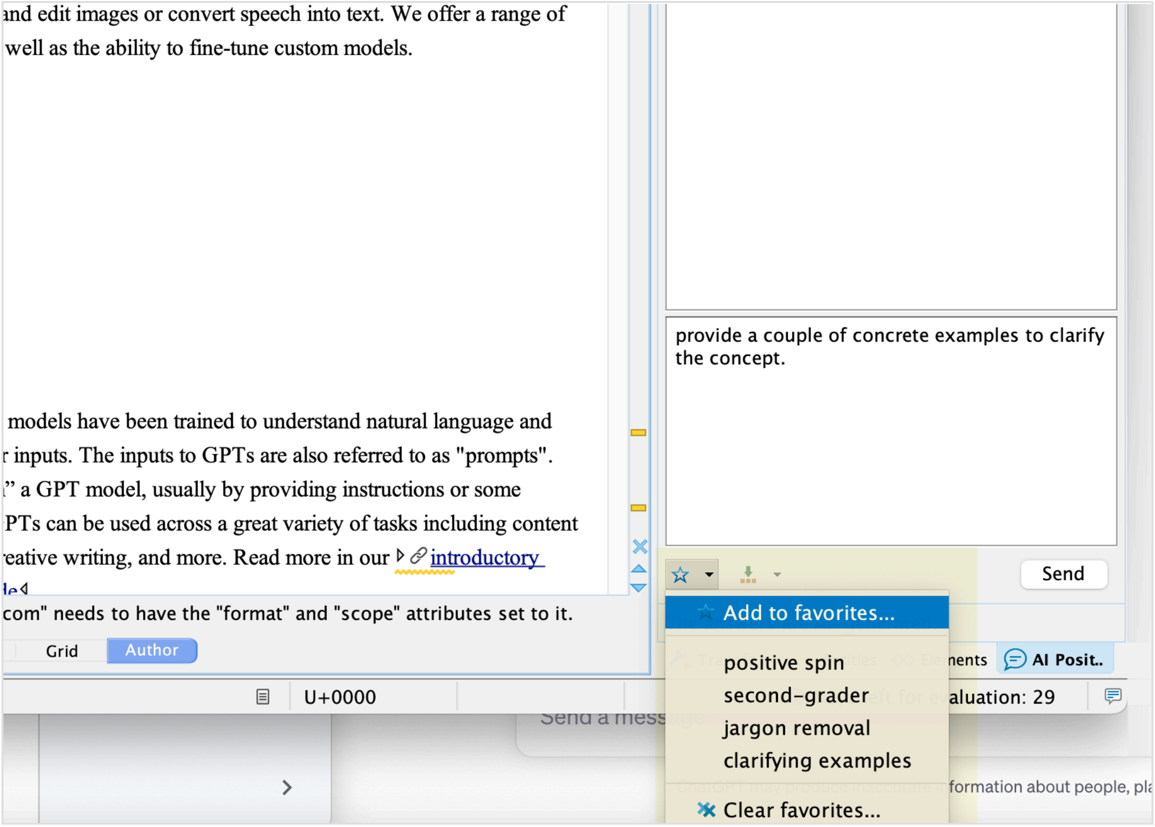
Me
Improve Readability
Modify the selected content to improve readability and fix grammar/spelling errors.

AI Positron
<p>An <i>embedding</i> is a vector representation of a piece of data, such as text, which aims to preserve its content and/or meaning. Data chunks that share similarities will have embeddings that are closer together than those that are unrelated. OpenAI provides text embedding models that take a text string as input and generate an embedding vector as output. Embeddings are beneficial for various tasks, including search, clustering, recommendations, anomaly detection, and classification. To learn more about embeddings, refer to our embeddings guide.</p>

provide a couple of examples to clarify the concept

Refining the AI prompt

In fact, you like this "provide examples" trick so much that you want to save it for future prompts. You click the blue-outlined star icon below this panel and add this modification to your favorites. Now you can easily insert this refinement without retyping it.


and edit images or convert speech into text. We offer a range of well as the ability to fine-tune custom models.

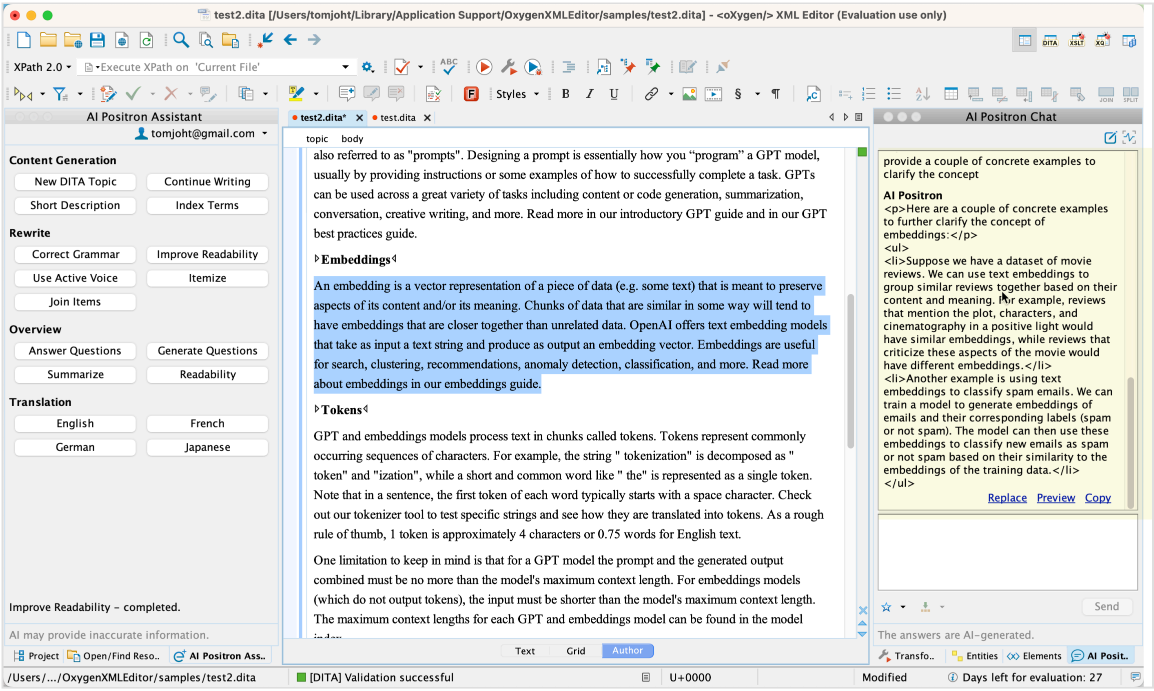
models have been trained to understand natural language and inputs. The inputs to GPTs are also referred to as "prompts". ” a GPT model, usually by providing instructions or some PTs can be used across a great variety of tasks including content eative writing, and more. Read more in our introductory

com" needs to have the "format" and "scope" attributes set to it.

provide a couple of concrete examples to clarify the concept.

You can add your own prompts to a list of favorites

After adding and saving your favorite, click **Send**.



also referred to as "prompts". Designing a prompt is essentially how you "program" a GPT model, usually by providing instructions or some examples of how to successfully complete a task. GPTs can be used across a great variety of tasks including content or code generation, summarization, conversation, creative writing, and more. Read more in our introductory GPT guide and in our GPT best practices guide.

Embeddings

An embedding is a vector representation of a piece of data (e.g. some text) that is meant to preserve aspects of its content and/or its meaning. Chunks of data that are similar in some way will tend to have embeddings that are closer together than unrelated data. OpenAI offers text embedding models that take as input a text string and produce as output an embedding vector. Embeddings are useful for search, clustering, recommendations, anomaly detection, classification, and more. Read more about embeddings in our embeddings guide.

Tokens

GPT and embeddings models process text in chunks called tokens. Tokens represent commonly occurring sequences of characters. For example, the string " tokenization" is decomposed as " token" and "ization", while a short and common word like " the" is represented as a single token. Note that in a sentence, the first token of each word typically starts with a space character. Check out our tokenizer tool to test specific strings and see how they are translated into tokens. As a rough rule of thumb, 1 token is approximately 4 characters or 0.75 words for English text.

One limitation to keep in mind is that for a GPT model the prompt and the generated output combined must be no more than the model's maximum context length. For embeddings models (which do not output tokens), the input must be shorter than the model's maximum context length. The maximum context lengths for each GPT and embeddings model can be found in the model index.

provide a couple of concrete examples to clarify the concept

AI Positron

```
<p>Here are a couple of concrete examples to further clarify the concept of embeddings:</p>
<ul>
<li>Suppose we have a dataset of movie reviews. We can use text embeddings to group similar reviews together based on their content and meaning. For example, reviews that mention the plot, characters, and cinematography in a positive light would have similar embeddings, while reviews that criticize these aspects of the movie would have different embeddings.</li>
<li>Another example is using text embeddings to classify spam emails. We can train a model to generate embeddings of emails and their corresponding labels (spam or not spam). The model can then use these embeddings to classify new emails as spam or not spam based on their similarity to the training data.</li>
</ul>
```

Improve Readability – completed.

AI may provide inaccurate information.

The answers are AI-generated.

[DITA] Validation successful

Adding custom prompts to your Favorites

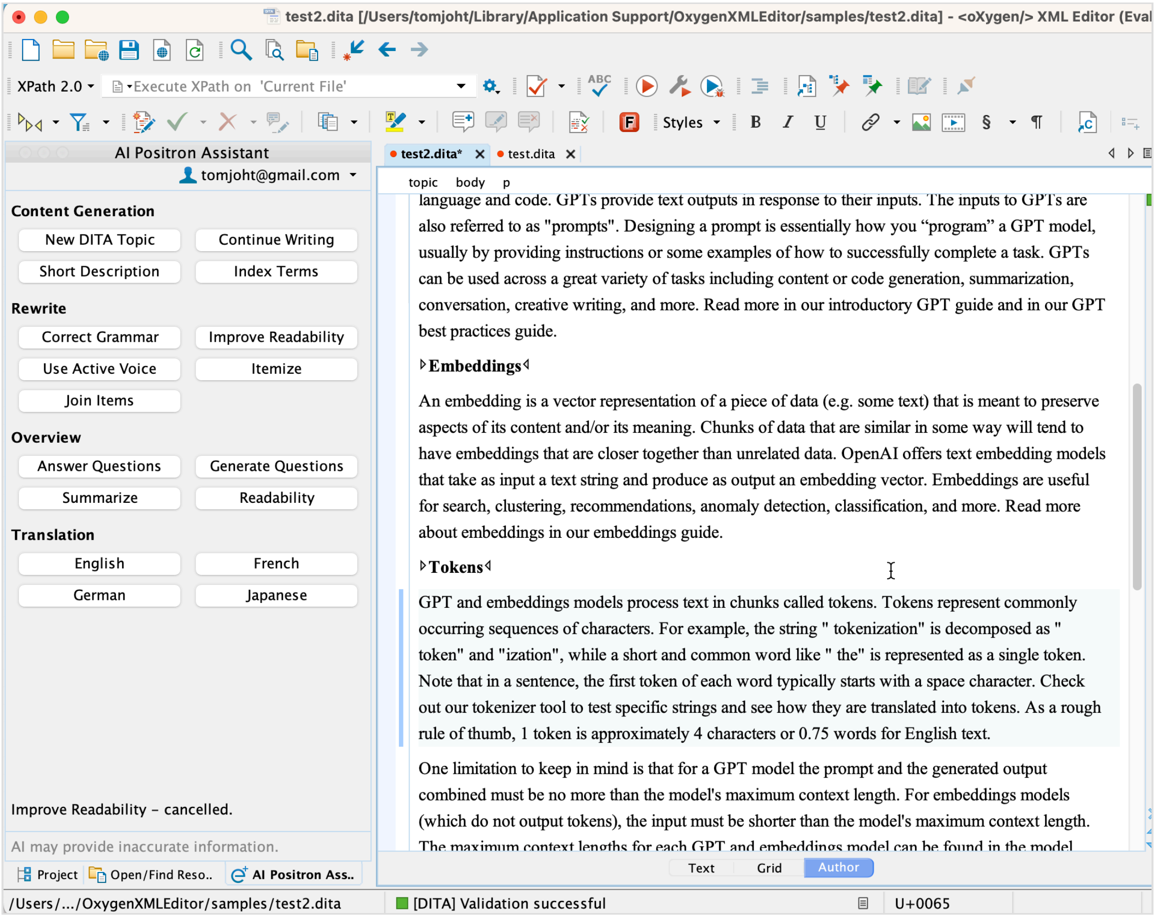
The examples offer more clarity! As a tech writer, you know that providing examples is a secret to clarifying complex concepts, and you've been using this technique for years. Now you know how to tease clarity out of AI as well. *That's why you're a technical writer.*

If you like the response, you can click **Replace** to insert the response in place of text you initially highlighted. In this case, because you're expanding on the concept with examples (not replacing the original), click **Copy** and manually insert it in the Text view (because the response already has DITA tags already).

This scenario provides you an idea of the workflow for using Positron. You haven't left your documentation project context, and you surgically selected parts of your existing documentation to improve. Of course, you'll want to review the AI-generated additions with the engineers, but even if they adjust the examples, you've already improved the document.

Available action buttons

"Improving Readability" is just one action button. The Positron Assistant offers many more actions:



language and code. GPTs provide text outputs in response to their inputs. The inputs to GPTs are also referred to as "prompts". Designing a prompt is essentially how you "program" a GPT model, usually by providing instructions or some examples of how to successfully complete a task. GPTs can be used across a great variety of tasks including content or code generation, summarization, conversation, creative writing, and more. Read more in our introductory GPT guide and in our GPT best practices guide.

Embeddings

An embedding is a vector representation of a piece of data (e.g. some text) that is meant to preserve aspects of its content and/or its meaning. Chunks of data that are similar in some way will tend to have embeddings that are closer together than unrelated data. OpenAI offers text embedding models that take as input a text string and produce as output an embedding vector. Embeddings are useful for search, clustering, recommendations, anomaly detection, classification, and more. Read more about embeddings in our embeddings guide.

Tokens

GPT and embeddings models process text in chunks called tokens. Tokens represent commonly occurring sequences of characters. For example, the string " tokenization" is decomposed as " token" and "ization", while a short and common word like " the" is represented as a single token. Note that in a sentence, the first token of each word typically starts with a space character. Check out our tokenizer tool to test specific strings and see how they are translated into tokens. As a rough rule of thumb, 1 token is approximately 4 characters or 0.75 words for English text.

One limitation to keep in mind is that for a GPT model the prompt and the generated output combined must be no more than the model's maximum context length. For embeddings models (which do not output tokens), the input must be shorter than the model's maximum context length. The maximum context lengths for each GPT and embeddings model can be found in the model

Improve Readability – cancelled.

AI may provide inaccurate information.

/Users/.../OxygenXMLEditor/samples/test2.dita

[DITA] Validation successful

Positron includes more than a dozen quick actions

Each button indicates its function when sending the prompt. For convenience, I copied the button prompts below and also included my own thoughts about their usefulness. Keep in mind, I've only briefly explored these buttons. They might be more useful for different scenarios.

Content Generation buttons

Action	Description	Tom's thoughts on usefulness
New DITA Topic	Generate a new DITA XML topic based on a text description entered in the popup dialog box.	Very useful. Suppose you're pasting content from Google Docs into your project. This prompt automatically structures the content with the right DITA XML tags.

Continue Writing	Generate additional text based on the content preceding the text cursor position.	Not useful for me. This prompt puts the AI into content creation mode, so it will only predict what might come next. I haven't found a way to use this productively. ChatGPT isn't a fortune teller. Expect hallucination and creativity here. Might be useful if you have writer's block or something.
Short Description	Generate a DITA XML <shortdesc> element based on a summary of the selected text or on the entire document content and insert it in the document.	Moderately useful. By providing the existing context, the AI is less likely to fabricate. If a request exceeds the 4k token limit, Positron automatically switches to a model that supports 16k tokens. (100 tokens is about 75 words. Keep in mind that the token limit includes both the input and response.)
Index Terms	Generate a DITA XML <keywords> element that contains index terms based on the selected text or on the entire document content and insert it in a <prolog> element.	Moderately useful. Indexing is one of those mundane publishing tasks no one likes to do, but we all know good search depends on index terms.

Rewrite buttons

Action	Description	Tom's thoughts on usefulness
Correct Grammar	Correct the grammar and spelling of the selected content.	Very useful. We all want to avoid grammar mistakes and typos, right? This tool can potentially replace other grammar checkers. Especially when combined with the diff tool, you can see exactly what grammar changes Positron suggests. Again, the diff tool is worth its weight in gold. Otherwise, you have to compare the AI responses line by line or by copying/pasting the original and modified versions into external diff tools.
Improve Readability	Generate additional text based on the content preceding the text cursor position.	Moderately useful. Beyond the default responses, I improve readability by asking for clarifying examples and requesting an explanation that a second grader can understand.

Use Active Voice	Convert the selected text into active voice.	Moderately useful. I'm not sure why this grammar principle is singled out from Correct Grammar, but tech docs are easier to understand when there's a clear actor performing an action (active voice). Passive voice obfuscates meaning.
Itemize	Convert the selected content into a summary and a list of items.	Not useful for me. This command reformats a paragraph into a list, which makes it more scannable. I'm not sure how much I need AI for this. It might be more useful for SMEs writing documentation (who are often unfamiliar with list conventions in technical writing style).
Join Items	Convert the selected list of items into a paragraph.	Not useful for me. This converts a list into a paragraph (the opposite of the previous command). Perhaps if you write by making an outline first, this command could speed up the narrativization of content. It's often easier to create a bulleted list instead of fully formed sentences. So this prompt might help you move from an outline to a first draft.

Overview buttons

Action	Description	Tom's thoughts on usefulness
Readability	Generate suggestions for changing the selected content or the entire document to improve its readability.	Could be useful. Rather than providing suggestions for improving readability, the AI response often just rewrites the text, making this somewhat redundant with Improve Readability. (I think the point of this action is to teach you to recognize readability patterns in your content.)
Summarize	Generate a summary of the selected content or of the entire document.	Useful. AI tools excel at summarizing content. AI's default explanatory style fits well with the style of short description elements.

Generate Questions	Generate questions that are answered by reading the selected text or the entire document content.	Not useful for me. If you want to see what questions your content answers, great. I can't imagine actually using this. I'm too lazy to test how my content might appear in chat or search engines. I'm more focused on making sure the content is accurate and complete. Then again, suppose you want to restructure your content into a Q&A format because that format might surface better in AI chat responses. This prompt can help you do that.
Answer Questions	Generate answers to questions you ask based on the selected text or on the entire document content.	Moderately useful. This turns Positron into a wise person whom you can ask questions. You can use this prompt to get a better understanding about different parts of your documentation (e.g., what is this code doing, why would someone use X, what is an embedding). Given that we often struggle to understand deeply technical content, having an AI assistant help explain the confusing parts, even if sometimes wrong, is a welcome tool.

Translation buttons

Action	Description	Tom's thoughts on usefulness
English	Translate the selected content to English while preserving its original DITA XML markup.	Not useful for me. These buttons aren't intended to push your content through translation workflows. Instead, they're more useful for a non English-native writer (or for content received from a SME who does not know English well) to quickly get a correct version as close as possible to the original.
French	Translate the selected content to French while preserving its original DITA XML markup.	(same as above)

German	Translate the selected content to German while preserving its original DITA XML markup.	(same as above)
Japanese	Translate the selected content to Japanese while preserving its original DITA XML markup.	(same as above)

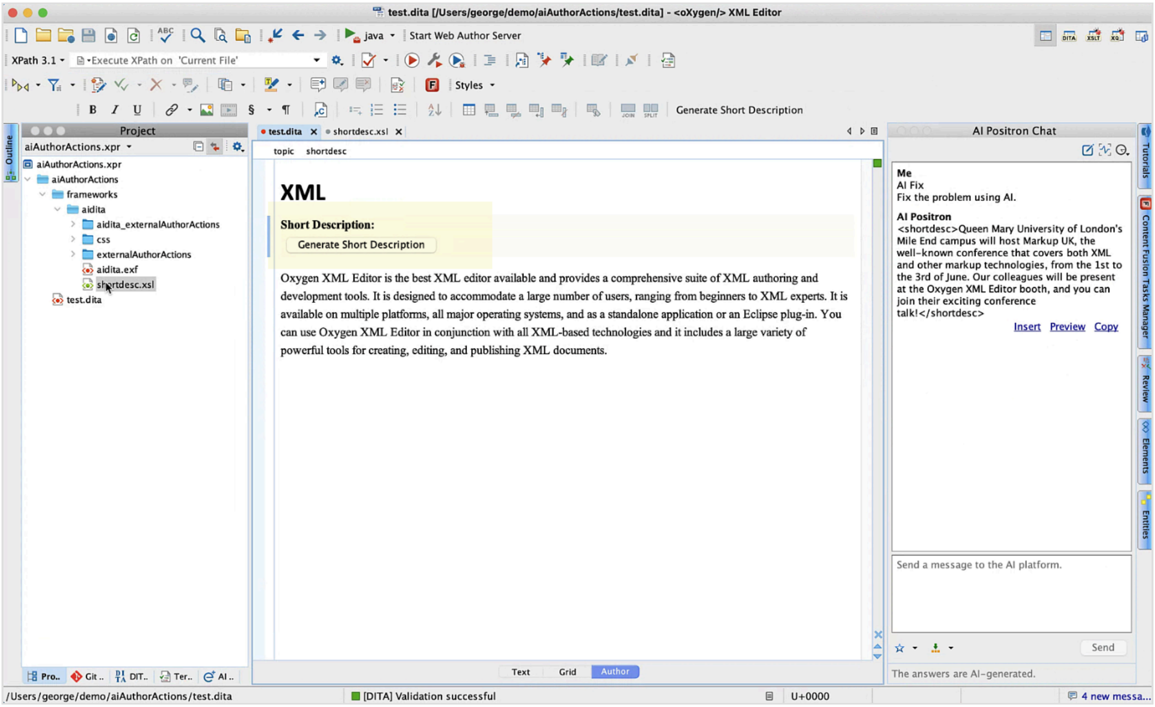
Keep in mind that Positron only reveals summaries to the authors, not the full prompts that it sends to ChatGPT. A significant part of mastering AI interaction lies in the art of prompt engineering. If you're skilled at eliciting the desired response from an AI bot, a practice akin to magical incantations, your answers will far surpass those of others. In fact, you can actually purchase prompts for a variety of scenarios, so I can see why Oxygen XML keeps their full prompts hidden. The skill of creating the perfect prompt for each tech comm situation is the pivotal factor in a high-quality AI integration.

Error correction

Positron is also integrated into the Results pane to perform error correction. If you have an error with the DITA code, underlined in yellow, you can right click the error message and select **AI Positron Fix**. This will prompt Positron to send the error to ChatGPT to suggest a fix for the error. This will result in speedier fixing and debugging of issues.

Button prompts

You can also configure Oxygen XML to include buttons in areas where elements are missing content. For example, if the `shortdesc` element is missing or empty in a topic, you can configure Oxygen XML to show a button that will prompt Positron to send a query to ChatGPT to populate that missing content.


test.dita [/Users/george/demo/aiAuthorActions/test.dita] - <oXygen/> XML Editor
Start Web Author Server
Generate Short Description
Project
aiAuthorActions.xpr
test.dita
shortdesc.xsl
topic
shortdesc
AI Positron Chat
Text
Grid
Author
/Users/george/demo/aiAuthorActions/test.dita
[DITA] Validation successful
U+0000
4 new messa...

XML

Short Description:

Generate Short Description

Oxygen XML Editor is the best XML editor available and provides a comprehensive suite of XML authoring and development tools. It is designed to accommodate a large number of users, ranging from beginners to XML experts. It is available on multiple platforms, all major operating systems, and as a standalone application or an Eclipse plug-in. You can use Oxygen XML Editor in conjunction with all XML-based technologies and it includes a large variety of powerful tools for creating, editing, and publishing XML documents.

Me
AI Fix
Fix the problem using AI.

AI Positron
<shortdesc>Queen Mary University of London's Mile End campus will host Markup UK, the well-known conference that covers both XML and other markup technologies, from the 1st to the 3rd of June. Our colleagues will be present at the Oxygen XML Editor booth, and you can join their exciting conference talk!</shortdesc>

Insert Preview Copy

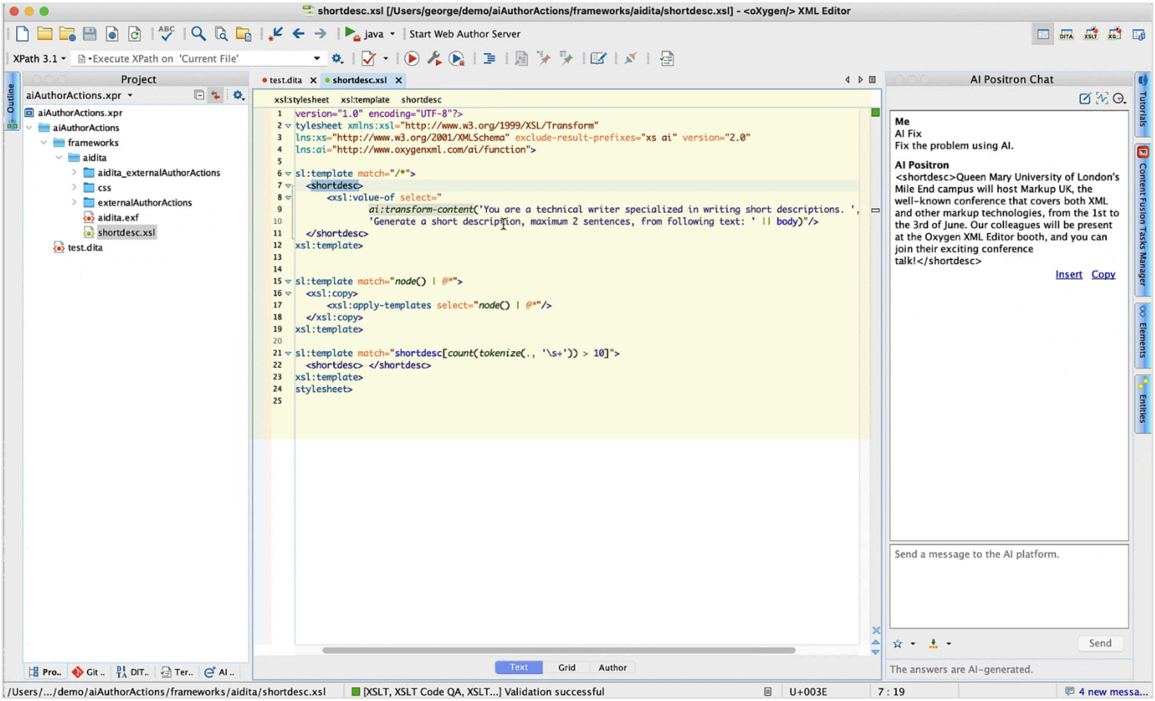
Send a message to the AI platform.

Send

The answers are AI-generated.

You can configure Oxygen XML with buttons to generate AI content in places that are missing content

To configure this, first you define a custom action using an XSL rule. These rules use XPath, an expression language used for Schematron, XQuery, and more. You would configure the rule to look for a root element and `shortdesc` element; if empty, then invoke an action that sends a query to Positron. Here's a sample configuration of the custom action.


shortdesc.xsl [/Users/george/demo/aiAuthorActions/frameworks/aidita/shortdesc.xsl] - <oXygen/> XML Editor
java
Start Web Author Server
XPath 3.1
Execute XPath on 'Current File'
Project
aiAuthorActions.xpr
aiAuthorActions.xpr
aiAuthorActions
frameworks
aidita
aidita_externalAuthorActions
css
externalAuthorActions
aidita.exf
shortdesc.xsl
test.dita
test.dita
shortdesc.xsl
xsl:stylesheet
xsl:template
shortdesc
Text
Grid
Author
AI Positron Chat
Insert
Copy
Send a message to the AI platform.
Send
The answers are AI-generated.
Tutorials
Content Fusion Tasks Manager
Elements
Entities
/Users/.../demo/aiAuthorActions/frameworks/aidita/shortdesc.xsl
[XSLT, XSLT Code QA, XSLT...] Validation successful
U+003E
7 : 19
4 new messa...

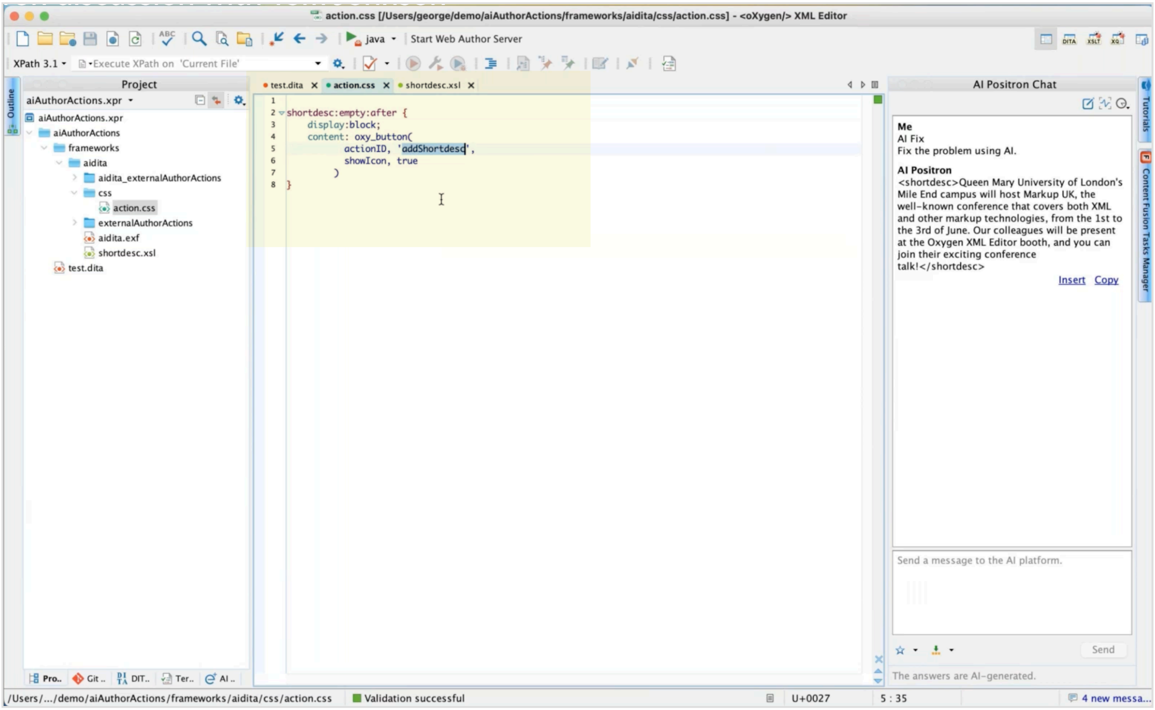
```
version="1.0" encoding="UTF-8"?>
tylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
lns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs ai" version="2.0"
lns:ai="http://www.oxygenxml.com/ai/function">

sl:template match="/*">
  <shortdesc>
      <xsl:value-of select="
              ai:transform-content('You are a technical writer specialized in writing short descriptions. ',
              'Generate a short description, maximum 2 sentences, from following text: ' || body)"/>
  </shortdesc>
xsl:template>


sl:template match="node() | @*">
  <xsl:copy>
      <xsl:apply-templates select="node() | @*"/>
  </xsl:copy>
xsl:template>

sl:template match="shortdesc[count(tokenize(., '\s+')) > 10]">
  <shortdesc> </shortdesc>
xsl:template>
stylesheet>
```

Me
AI Fix
Fix the problem using AI.

AI Positron
<shortdesc>Queen Mary University of London's Mile End campus will host Markup UK, the well-known conference that covers both XML and other markup technologies, from the 1st to the 3rd of June. Our colleagues will be present at the Oxygen XML Editor booth, and you can join their exciting conference talk!</shortdesc>

Configuring custom actions based on XSL rules

After creating the action, you create a style rule in `action.css` to add a button for an empty `shortdesc` element. This button will invoke the `shortdesc` action. Here’s a sample configuration:


action.css [/Users/george/demo/aiAuthorActions/frameworks/aidita/css/action.css] - <oXygen/> XML Editor
java
Start Web Author Server
XPath 3.1
Execute XPath on 'Current File'
Project
aiAuthorActions.xpr
aiAuthorActions.xpr
aiAuthorActions
frameworks
aidita
aidita_externalAuthorActions
css
action.css
externalAuthorActions
aidita.exf
shortdesc.xsl
test.dita
test.dita
action.css
shortdesc.xsl
AI Positron Chat
Insert
Copy
Send a message to the AI platform.
Send
The answers are AI-generated.
Tutorials
Content Fusion Tasks Manager
/Users/.../demo/aiAuthorActions/frameworks/aidita/css/action.css
Validation successful
U+0027
5 : 35
4 new messa...

```

shortdesc:empty:after {
    display:block;
    content: oxy_button(
            actionID, 'addShortdesc',
            showIcon, true
         )
}
```

Me
AI Fix
Fix the problem using AI.

AI Positron
<shortdesc>Queen Mary University of London's Mile End campus will host Markup UK, the well-known conference that covers both XML and other markup technologies, from the 1st to the 3rd of June. Our colleagues will be present at the Oxygen XML Editor booth, and you can join their exciting conference talk!</shortdesc>

Sample configuration of css action

For more details on configuring buttons, see the following:

- Webinar: Improving the Oxygen Authoring Experience Through Custom Actions
- Button Form Control
- Form Controls

Limitations

Now that we've covered Positron's functionality, I'll comment on a few limitations.

ChatGPT 3.5 instead of 4.0

Positron connects with ChatGPT 3.5 instead of 4.0. For simple tasks, there might not be a huge difference, but I prefer 4.0 quite a bit and think it's several levels better. That said, I don't think APIs can interact with 4.0 yet except by waitlist. Also, 4.0 is literally 30x more expensive per query. Additionally, with 4.0, the response time is slower, you're limited to fewer requests per minute, and 4.0 doesn't provide the 16k token limit model (as with 3.5). For all those reasons, 4.0 is problematic. However, Oxygen XML plans to allow for more flexibility in choosing models in the future. If someone wants to pay for a ChatGPT 4.0 upgrade, it will probably become an option.

Choosing other language models

Another limitation is extensibility. Right now, you can connect only to ChatGPT. Future releases will give you options to connect to other AI providers. (This seems to be the most frequent request in the comments on the Positron documentation.) Imagine if your company has its own internal AI models. If so, you'd definitely want to connect to them.

Token limits

In playing around with Positron for an afternoon, I used 62 of the 250 requests. (You can check usage by clicking the drop-down menu for your email under the "AI Positron Assistant" pane.) My guess is that a professional technical writer working on documentation for 4 hours a day will exhaust the monthly request limit within a week. This is because a lot of the tasks require greater amounts of context passed in. When you want to create a `shortdesc` of a topic, you pass in your entire document. If it's 1,000 words, that's about 750 tokens right there. Do that for 20 topics in your documentation, and you start using a lot of requests.

Note that in the first release, Oxygen XML measures usage in "requests," but future releases will switch to tokens.

Keep in mind that the Positron Assistant is a free extension. Most other API integrations charge a high amount and cap the usage at a certain number of queries. (For comparison, the Madbot plugin costs $129/month.)

Confidential data

Using Positron means sending your content through APIs to OpenAI services to be parsed, transformed, and returned. If your company doesn't allow its content to be sent to third-party servers due to the confidentiality of the data, this extension may be off-limits to you. You can't connect Positron to a language model that runs from your local computer only.

For more details about how your content passed into Positron might be used, see OpenAI's Terms of Use, specifically section 3c, "Use of Content to Improve Services." This section says OpenAI won't use the content you send through the API to train their model, but they might scrape your public website to train its model.

The confidentiality of data is what's slowing down big tech companies from using tools like ChatGPT, Bard, or others. Many companies are afraid of their confidential code and information suddenly appearing in publicly available language models.

Conclusion

Oxygen's AI Positron assistant provides a practical approach to using AI when writing documentation. The integrated experience allows you to stay focused on your documentation in Oxygen XML, and the pre-made prompt actions help you quickly achieve the desired results.

One detail I haven't yet addressed is the name: Positron. The name was inspired by Isaac Asimov's "positronic brain," which is a fictional device that provides robots with a sense of consciousness, reasoning, and simulated emotion. The character Data on Star Trek is an example of a robot with a positronic brain.

In Oxygen XML, the Positron Assistant isn't conscious, nor is it about to start simulating emotions. But you can think of the Positron integration as having a character like Data on your bridge, available to ask it questions, improve language, suggest fixes, and more. Who wouldn't want that?

Also, keep in mind that this is the first round of AI integration. We're at the beginning of what will no doubt be many more iterations and enhancements with AI integration. We're getting better at prompt engineering, understanding strategic ways to integrate AI, and reducing hallucination by passing in documentation context. It's an exciting time to be a tech writer. Explore and experiment with these new AI tools and see what works for you. Then share your experiences with the larger community.

AI document engineering with pattern-based prompts

Using patterns in prompts

Pattern prompting involves teaching the AI a specific structure or template, then having it populate information into that template. Pattern prompts are similar to few-shot prompts, but in this case, rather than having the language model populate the template with its own information, we'll have it sort and structure a mess of information into the template, thus reducing hallucination and error.

Pattern prompts are among the most interesting strategies for making requests with language models; this is really where the "engineering" aspect of prompting comes into play. In this case, it's the practice of AI document engineering. By constructing prompts into patterns for the AI to complete, or templates to populate, you'll see the true power of language models emerge. Change your prompt into rules and a template for language models to populate, and the responses will be much better than open-ended questions.

The following sections describe a four step process for AI document engineering. These process involves the following steps: 1. Gather the information, 2. Create rules, 3. Create templates, and 4. Assemble the information into a prompt.

To use a pattern prompt, follow these sections.

1. Gather the information

First, gather a lot of information about the API you're documenting. This information will provide the unstructured information that will be shaped by the template. Usually, when you're writing documentation, this information exists in myriad internal documents, scattered across internal sites, written by engineers and program managers, stuffed into engineering designs, project plans, analysis documents, and more. Gather up as much of the information as you can and make a list of the documents.

As you read through the documents, sort the relevant information into different buckets based on topic types. For example, group the information into the following different buckets:

- Product overview
- Troubleshooting
- Authorization
- Best practices
- Reference
- Installation
- Architecture
- Workflows
- Tasks
- Quick start tutorial
- Conceptual explanation
- Release notes
- Rate limiting
- Status and error codes
- Glossary

Consider this task the equivalent of sorting laundry. Put each type of clothing into its own pile—socks, underwear, pants, shirts, etc. For now, just throw everything into distinct buckets, unstructured and chaotic.

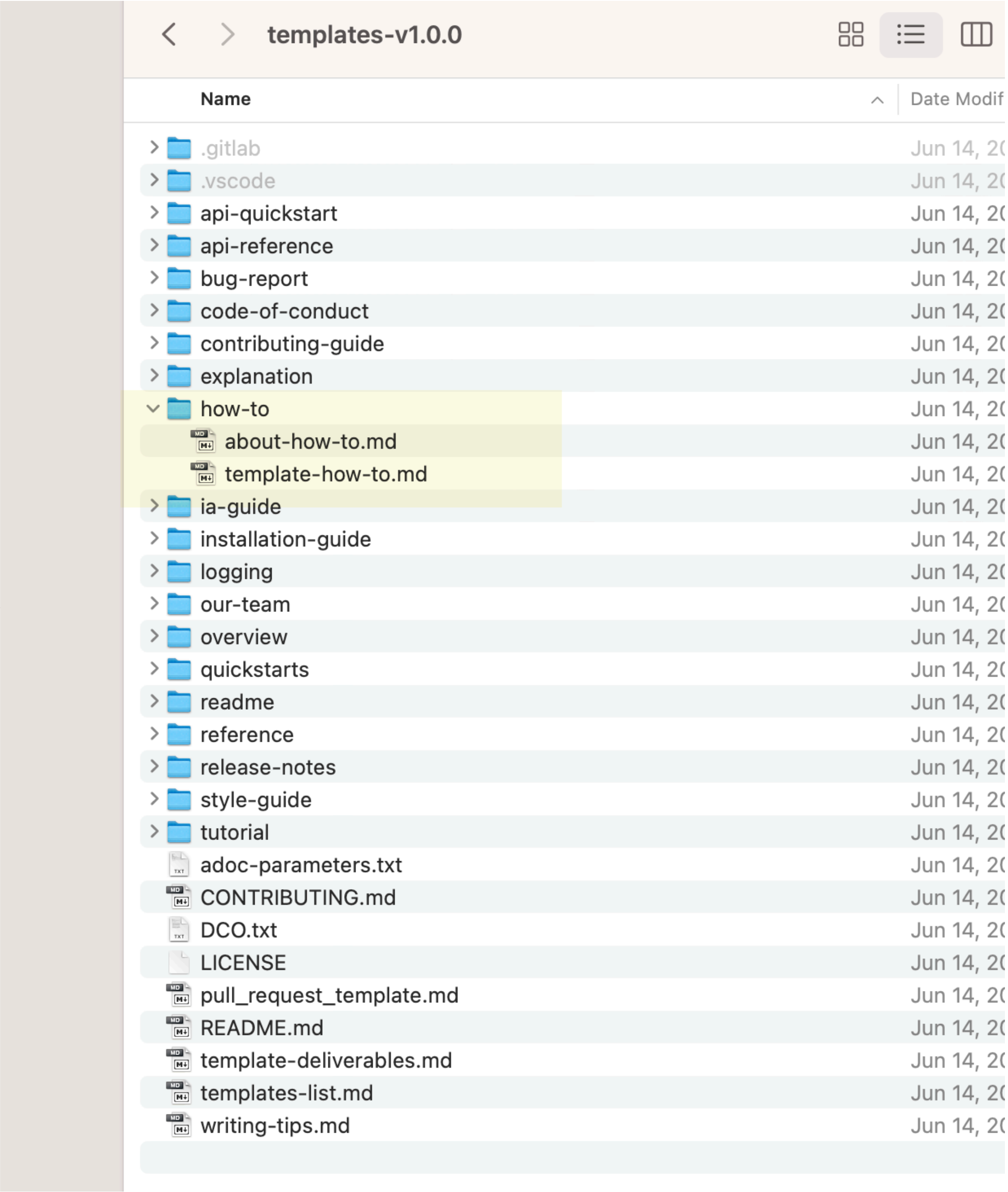
Because of token limits, each bucket should be relatively short. If you’re working with ChatGPT 4.0, consider a max of about 700 words or so. For ChatGPT 3.5, you can use the higher token limit, but the response might not be as good.

Admittedly, if you don’t have any of this information, the auto-generation of content isn’t going to work. I mean, you could still try, but without providing more context to the language model, chances are the model will end up hallucinating or predicting what it thinks the content should/could/might be. You might need to spend a lot of time gathering and reading through this information.

2. Create rules for the topic type

Now we need to create some rules for the AI language model to follow. To create the rules, you can leverage information resources such as the Template Suite from the Good Docs project, or information from my API course, or from DITA. Create a list of brief rules for each topic type, and then a template.

The Template Suite from Good Docs makes this easy. For example, suppose you want to construct a task template. From the Template Suite, open the “how-to” and see the `about-how-to.md` topic.



templates-v1.0.0

Name	Date Modif
.gitlab	Jun 14, 20
.vscode	Jun 14, 20
api-quickstart	Jun 14, 20
api-reference	Jun 14, 20
bug-report	Jun 14, 20
code-of-conduct	Jun 14, 20
contributing-guide	Jun 14, 20
explanation	Jun 14, 20
how-to	Jun 14, 20
about-how-to.md	Jun 14, 20
template-how-to.md	Jun 14, 20
ia-guide	Jun 14, 20
installation-guide	Jun 14, 20
logging	Jun 14, 20
our-team	Jun 14, 20
overview	Jun 14, 20
quickstarts	Jun 14, 20
readme	Jun 14, 20
reference	Jun 14, 20
release-notes	Jun 14, 20
style-guide	Jun 14, 20
tutorial	Jun 14, 20
adoc-parameters.txt	Jun 14, 20
CONTRIBUTING.md	Jun 14, 20
DCO.txt	Jun 14, 20
LICENSE	Jun 14, 20
pull_request_template.md	Jun 14, 20
README.md	Jun 14, 20
template-deliverables.md	Jun 14, 20
templates-list.md	Jun 14, 20
writing-tips.md	Jun 14, 20

Paste it into an AI language model with the following prompt:

> The following is a list of guidelines for writing tasks. These guidelines provide tips and best practices to follow. Condense this information into a brief, concise set of rules that an AI prompt can follow. Make the statement of the rules machine friendly and oriented.

<PASTE IN TASK GUIDELINES FROM ABOUT-HOW-TO.MD>

The result might look like this:

Rules for tasks:

- Single Task Focus: One guide equals one task.
- Advanced Users: Guides cater to advanced users, not beginners.
- No Concept Teaching: Guides should not explain concepts.
- Clear Step Sequences: Provide ordered, straightforward steps.
- Unexpected Scenarios Alert: Inform about possible error scenarios.
- Avoid Over-Documentation: Document the most common or recommended task completion method only.
- Ensure Accuracy: Test and confirm the technical accuracy of instructions.
- Simplicity and Brevity: Limit each guide to 8-10 steps.
- Prerequisites: Specify required knowledge, tools, and setups.
- Regular Updates: Re-test instructions post product updates for continued accuracy.

You could also do more or less the same thing from the DITA Task topic. For example, following a similar process as above, here's how the AI language model rendered these rules:

Rules for tasks:

- Ensure document type compliance with the latest standards.
- Answer "How do I?" with clear step-by-step instructions.
- Use distinct sections for prerequisites, context, steps, results, troubleshooting,
- examples, and follow-up tasks.
- Keep prerequisites succinct and clear.
- Provide brief but helpful context for the task.
- Make steps sequential, logical, and easy to follow.
- Detail specific actions for each step and optionally include additional information.
- Clearly state the expected outcome in the results section.

- Provide solutions in the troubleshooting section when expected results may not be met.
- Support tasks with relevant examples.
- Guide users to the next steps or tasks after task completion.

If you don't have a list of rules or template for a particular topic type, find a good example in documentation and reverse engineer it through the AI model. For example, you might say:

> Analyze the following software documentation text and identify core features which will allow further documents to imitate the same style, consistency, and formatting as the document below:
>
> <INSERT DOCUMENTATION SAMPLE>

For the documentation sample, pick out a task from good documentation, such as this Update existing subscriptions task from Stripe. (Copy only part of the task due to token limits.)

The AI response might be too detailed. Trim it down with a prompt such as the following:

> Make the list more concise and machine oriented, as it will be fed as instructions to an AI language model for processing data.

Here's what I ended up with following this reverse-engineering approach:

> Rules for tasks:
>
> - Headers: Use informative and concise headers for each section.
> - Ordered Steps: Present instructions in a clear, sequential order.
> - Nested Instructions: Include detailed sub-steps within high-level steps.
> - Direct Addressing: Use direct speech, addressing the reader as "you".
> - Definitions: Provide explanations and definitions for key terms.
> - Tabular Data: Use tables for conveying key information.
> - Links: Incorporate cross-references and links to related content.
> - Bold Formatting: Highlight important sections or actions using bold format.

- Standardized Phrasing: Maintain consistent phrasing for similar steps.
- Logical Structure: Arrange content in a logical flow from introduction, body to conclusion.

For fun, I also asked ChatGPT to merge the three lists of rules. Here's the outcome:

> Rules for writing tasks:
>
> - Single Task Focus: One guide per task.
> - Sequential Steps: Order steps logically with clear actions.
> - Headers and Structure: Use concise headers, organize content logically: introduction, body, conclusion.
> - Definitions and Context: Include key term definitions and task context.
> - Prerequisites: Specify necessary knowledge, tools, and setups concisely.
> - Outcome and Troubleshooting: State expected outcome, provide troubleshooting solutions.
> - Formatting and Cross-References: Highlight important sections, incorporate related links.
> - Unexpected Scenarios: Alert about possible error scenarios.
> - Accuracy and Updates: Ensure and maintain instruction accuracy, update post product changes.
> - Simplicity and Brevity: Keep guides to 8-10 steps, avoid over-documenting, focus on common/recommended methods.

Seems pretty good to me. Note that you don't want to be too wordy with the rules because you have limited tokens to pass. The more tokens the rules occupy, the less text you can process. Also, there's a balance between providing too little or too much information—both extremes can lead to poorer results.

3. Create a template

The Good Docs Template Suite also has a template that accompanies each topic type (for the how-to, it's `template-how-to.md`). Copy this template into the following prompt to your AI language model:

> From the following task template, strip this down into a template for an AI language model to follow as a pattern:
>
> <PASTE IN TASK TEMPLATE>

Here's a sample response:

> Task template:
>
> Title: "{Insert Brief Description of Task}"
>
> Overview: "This guide explains how to {insert a brief description of the task}."
>
> Before you start: (Optional) "Before you {insert brief description of task}, ensure: Prerequisite 1, Prerequisite 2, Prerequisite 3."
>
> Task: "{Task Name}"
>
> * {Optional description of the task.}
>
> Steps:
>
> 1. {Write the first step here. Start with a verb.}
> 2. {Write the second step here. Start with a verb.}
>   - {Substep 1}
>   - {Substep 2}
>
> Sub-task: (Optional, if the task is big and complex)
>
> See also: (Optional) {Include references to other related guides or information.}

Here's a similar template rendered from the DITA Task topic. (It took a little more massaging to get rid of the angle brackets and technical emphasis.)

> Task ID: {task_id}
>
> Task Title: {task_title}
>
> Task Body:
>
> Context: {context}
>
> Prerequisite: {prerequisite}
>
> Steps:
>
> - Step 1: {step_1}
> - Step 2: {step_2}
> - Step 3: {step_3}
> - Step 4: {step_4}
>
> Result: {result}
>
> Example: {example}

Coming up with these rules and templates shouldn't be too hard. The Good Docs Project, DITA OASIS committees, style guides, and other groups have been meeting for years to come up with the rules and templates for technical writing. Plus, you have all the best practices listed in my API doc course (for example, see the Conceptual documentation section (p. 358)). Now all this hard work can be used to teach machines how to write documentation.

4. Assemble the instruction for the AI

In this step, you'll pass information into an AI, such as ChatGPT, Bard, Claude, Poe, or another language model. (If you don't want to use ChatGPT or Bard, here's a long list of alternatives.)

If you company prohibits you from pasting internal content into third-party AI models like this, you might run into a roadblock here.

With your set of rules and template, apply it to the unstructured content in a prompt like this:

> INPUT:
>
> You are a technical writer creating documentation, specifically a task. You follow a list of rules and arrange information into a given template.
>
> Rules: Here are the rules for writing tasks:
>
> <INSERT RULES>
>
> Template: Here is the template for writing tasks:
>
> <INSERT TEMPLATE>
>
> Make the following content fit into the task template above:
>
> <INSERT UNSTRUCTURED CONTENT>

And voila! You should have a draft of documentation ready to review and add to your documentation project. It won't be perfect, but it will probably be 70% there.

Demo

To demo this process, let's say you're creating documentation about how to make a basic call with the OpenWeatherMap API (an API I've used elsewhere in this course). You gathered the info about making a basic call here in this Google Doc. The information is just pasted in randomly without much shape or form.

For this task, I used ChatGPT 4.0. You can see the response here. I also pasted it into Bard and received a similar response (but Bard doesn't offer an easy way to share the link to the output).

In looking at the output, I can see that the template might need to be modified in some ways. Also, this how-to task might not be as applicable to API documentation as other types of documentation. That said, the AI language models do a nice job of taking content and fitting into the shape and feel of documentation. Overall, this approach feels like the right way to do AI document engineering. The next steps might be to create rules and templates for all the documentation types listed above, providing them as a ready-made toolkit for writing documentation.

Conclusion

For each of the topic types, you can assemble a set of rules and a template that you can easily pass into your AI language model. You could even store your set of prompts in a prompt manager extension, though I imagine storing them in a Google doc would be easier to update and maintain. Using this approach, your content will follow a more predictable format and structure. Theoretically, you could write the documentation for your project in half the time.

Of course, due to the high degree of errors from AI models, you should review all the documentation with subject matter experts. Recognize the language models have an uncanny ability for sounding correct and authoritative when they are lying.

Like I said in my disclaimer above, this is a method I’m still experimenting with. I won’t know how well it all works until I’ve been able to run an actual project through the process.

Using AI tools to build, stage, and publish API reference docs

At the core of API documentation work is building, staging, and publishing of API reference content. Whether it's Javadoc ↗, Doxygen ↗, OpenAPI ↗, or other reference output, almost every API has reference documentation that you build, stage, and publish with each release.

Given the centrality of documentation building and publishing tasks, AI tools can be a great help when it comes to configuring scripts to perform these tasks. This is one AI area few people are focusing on, but scripts are an easy way to incorporate AI to improve your productivity and reduce the tediousness of document production.

In this article, I'll focus on shell scripts (`.sh`) because they're commonly used in Linux environments when working with files. For docs-as-code systems, shell scripts can do a lot of tasks. You run a shell script from your terminal like this: `./some-script.sh`. The script then performs a process on one or more files. However, you could just as easily create other types of scripts, such as Python scripts.

Publishing example

One product I support includes Java API documentation, including a Javadoc for the reference docs. To build the Javadoc, I roughly do the following:

- Get a link to the release build from the build management system
- Configure a build command using the release build link
- Switch into a special branch for the build
- Build the reference output for multiple API variants
- Copy the generated outputs from the build output directory to the documentation directory
- Insert a note into the pre-release builds
- Push the docs to the staging server
- Set up a changelist to review with engineers

Doing this all manually required us to follow a series of steps that took about 10-15 minutes, tediously following an internal guide through the process. Because the reference documentation is generated from comments in the source files, if we spotted something we wanted to fix in the comments, we would need to rebuild the Javadoc output and re-transfer everything to the staging server again.

After doing this a few times per release (for example, trying to get code for links correct, figuring out some Markdown formatting, or adjusting the language in a comment), it became extremely tedious.

One day I created two shell scripts that automated the entire process down to about a minute, with almost no cognitive load. You just ran the scripts and waited for them to process. All of the complexity was in the scripts. We fed the shell script commands a couple of parameters and then watched them work, speeding through each step of the process with breakneck speed. It almost felt like cheating.

Building a shell script

For a developer familiar with shell scripting, creating a script to automate a build process probably isn't too hard. It depends on the complexity of the build process. But I'm a shell scripting novice, and I didn't feel like devoting a summer's worth of time learning shell scripting how-to's. (Actually, I did read through a 20-page shell scripting guide, but it didn't get me any closer to the code I needed.)

Using AI tools, I stepped my way through each piece of logic that I needed until I completed the script in about two days. Due to confidentiality of data and processes, I generalized my AI queries using pseudo code like this:

> Build a shell script that does the following:
>
> - Run the command foo. The output of the foo command is bar.
> - From the bar response, capture the part after path acme.
> - Store the part after the path acme into a variable called barResponse.
> - Pass the barResponse variable as a parameter into the beta command.
> - The output is a doc called mydocs.zip. Unzip the mydocs.zip and copy it over to another directory here ...
> - On each page of the output, insert a note after the <body> element that says ...

And so on. You get the point. AI would return code that worked (most of the time) and also taught me about shell scripting along the way.

Writing pseudo code is fairly easy. If the AI tool starts recommending methods not available in your environment (e.g., configuring Flask or Ruby plugins), steer it back in the direction of shell scripts or other technology supported in your environment. The key is to write specific instructions as pseudo code.

Granted, you will need *some* technical familiarity with code to make your through this process. But if you're a technical writer, you probably have this technical acumen already. And if not, you can learn.

Reference content is high value–tech writers should be involved

Due to the complexity of building and configuring files in reference generation processes, many technical writers actually delegate the reference docs to engineers. For example, tech writing teams might say they don't have bandwidth to build the reference output and will require engineering teams to do it instead.

However, this puts the reference content in the engineer's domain, removing technical writers from the process. If technical writers don't participate in editing and crafting reference content for an API, they put themselves outside of the perhaps most valuable aspect of API documentation.

Because reference content is high-value content, I prefer to get my hands dirty with field definitions, cross-references, identifying incomplete definitions, and so on. Here's a little-known secret about API technical writing: you don't have to be a developer to provide valuable input about language-specific API reference documentation, such as for Java or any other language. For the most part, a Java API provides various classes and methods that return data. The developers working with your API are primarily interested in the data that gets returned. Usually, the data in an API is poorly described. I rarely see developer questions about how to use the API's classes and methods; instead, developers ask for more details about the data.

In fact, with the prevalence of AI tools to explain code, general knowledge about working with a programming language is less needed. What's essential is the clarity of information about what's specific to your API. What's specific is the data the API returns, not the constructs and mechanics of the programming language and usage.

If you want to be a key player with API documentation, you probably need to manage the building and publishing of the reference content. Maybe your company has a push-button system already in place. If so, great. Chances are, however, especially if you're working in a docs-as-code shop, there are probably custom processes all over the place, and scripts that manage it all. By creating your own shell scripts to automate part of the building and publishing of reference docs, you can stay involved as a key player with reference docs without losing bandwidth and energy in library building and publishing tasks.

Another script-building example

I was feeling pretty good about my first advanced shell script, so I started wondering what else I could do. I have another scenario that involves disambiguating between preview API documentation and current release documentation. I'd been generating two reference variants, but they were hard to distinguish in the developer portal. Could I write a script to identify specific pages in the preview (based on a diff of two table of contents), and then insert custom notes on those specific pages, and other notes for the release variant?

Again, I started proceeding line by line with the logic. It didn't take more than two afternoons before I'd finished the script. This one helped disambiguate the various outputs and insert custom notes. Although I'm sure developers could have done this, they're usually too involved in writing code to focus on the documentation. (In my experience, it's rare for engineers to sink time and energy into anything related to documentation, even if it involves coding build logic.)

Blog example

Outside of building reference docs, you can use scripts for other tasks. A few months ago I posted a script about creating a new post on my site that populated the post with the needed frontmatter, tags, and also made a call to Rebrandly to create a shortlink for a post. See A script that creates a new Jekyll post and populates it with YAML frontmatter, and also makes a curl call to add a Rebrandly shortlink (p. 0), or view the code here: `post.sh` ↗.

I also created some scripts to customize the PDF build process (the output of which I publish here (p. 0)). I haven't explained much about that process, but the PDF build logic I've created for this site is pretty nifty. My PDF build command is a script here: pdf_build_docapis_all.sh ↗. This script references a page `prince-list-all.txt` ↗ that has its own code to gather up all the pages for Prince XML (the PDF generation tool). After the build, another script, `pdf_upload_docapis.sh` ↗, uploads the content into Wasabi, the server where I host binary assets like PDFs and images.

I also have a simple upload script for images as well, `upload.sh` ↗, that lets me upload images to Wasabi from the command line and returns the code for easy copy/pasting into a site. Given how common it is to upload images, this small script has saved me lots of time over the past few years.

Other applications for scripting

Depending on your systems for technical authoring and publishing, you could create scripts for many aspects of authoring and publishing, especially as you combine the shell scripts with other command line interfaces available to you. Here are a few example applications, including the ones I've already mentioned:

- Build reference docs and copy them into your documentation system.
- Create new pages and populate them with the needed frontmatter and sections based on the doc type.
- Upload images to your web server, with responses that include copy-and-paste ready HTML code with figure references and captions.
- Generate and publish PDF versions of your content.
- Configure release processes that kick off builds based on specific events, such as a bug that gets closed or a code release.

- Identify the last time a page was edited, correlated with analytics that list the most viewed pages of your site.
- Trigger actions that kick off when people create changelists for documentation edits, such as notification workflows for specific groups or product owners of those groups.
- Create pre-staging checks (linters) on existing content, such as style checking, Markdown formatting, link validation, alt tags on images, and more.
- Convert file formats using tools such as Pandoc to automate the conversion.
- Version content after a release by archiving content and updating file names and directories.
- Perform post-publishing actions, such as updating a search index, closing bugs, or pinging groups with links to the new docs.
- Send build notifications to specific groups about the status of docs during build, staging, and publishing to keep groups informed.
- Script a process that generate bugs for each step of a release.

Look for opportunities to automate processes with scripts.

The downsides of using AI for help with scripting

The picture I've painted so far is a positive one. However, there are a few downsides to using AI to help you with scripting. First is what happens when an engineer who actually knows shell scripting reviews your code and makes helpful, high-level suggestions to improve the logic, without realizing that you have only a superficial understanding of how it all works. This actually happened to me.

When this happens, do you openly admit that you used AI to come up with the code? Do you feed the suggestions to the AI and ask it to incorporate the suggested tweaks? Do you shrug your shoulders and just say thanks? In my case, I again interfaced with AI to incorporate the suggested script enhancements.

Consider another scenario where your AI reliance becomes extreme and constant. Suppose you've started writing all kinds of scripts and apps that you have no business creating because you don't understand how any of the code works, only that it does. At some point, you might end up so reliant on AI for code that you pass even the simplest of requests to AI, trying out its responses and changing the queries when the code doesn't work. You shortcut attempts to actually learn the syntax you're using and instead just ask AI tools to help you with every little tweak to the code.

Well, if you'll doing a lot of shell scripting, it would be a good idea to actually learn how the code works. Fortunately, the AI code gives you a great starting point to learn, especially because most AI tools explain the code that they provide. This brings me to another point that I'll make in later post: Using AI tools to *learn* code.

But chances are, right now you're in a hurry, and you just need this *one script* for this *one task*, then you don't need to work with that scripting language for anything else … for now. Is it really worth spending a week or two to learn it, or can you get by with partial understanding? Fortunately, you have the choice to take either path. AI has given you more options.

Apathy from lack of understanding/expertise

There's also a sense of hollowness that follows when you let AI write the code, a sense of apathy that follows machine reliance. When we lose our expertise and self-sufficiency, we become less interested in the tasks we're performing. We see ourselves as an intermediary between the code and the machine, a kind of middle-person who articulates instructions and relays the responses, but who doesn't create the information or code anymore. This apathy can lead to boredom in our role. And with boredom, loss of interest and care.

However, again, if you see this happening, I would recommend studying the code that AI generates and using it as a springboard for your own learning and study.

Conclusion

AI tools empower technical writers with scripting capabilities, whether it be shell scripts, Python scripts, CLIs available at your work, or more. In particular, shell scripting can help you automate parts of your build process that are tedious, making it easier to push docs through advanced build and publish processes. In a world of doc ops, where continuous builds and publishing are becoming the norm, tech writers need as much automation as possible with these processes.

Using AI to learn coding

The challenge of learning code

One challenge API technical writers face is understanding developer code and tools (p. 664). This is by far the most intimidating aspect of being an API technical writer. Chances are, you're working with code from various programming languages without a deep understanding of them. Yet your job is to explain to a target audience of developers the purpose of the code and how to use it. At first, this requirement seems like API technical writers would need to be developers themselves to succeed.

As if documenting code for one project weren't enough, API technical writers must also support multiple projects simultaneously, often with different types of code. You might document a Java API for one project, a REST API for another project, some Go code for an SDK, some C++ code for another project, and so on. It can be nearly impossible to be fluent in all of these languages. Even keeping the terminology straight (a "function" in one language, a "class" is another, etc.) can be challenging.

Fortunately, you can use AI tools to learn code. AI tools can act like a friendly programming buddy who is sitting next to you, ready to explain anything you want, at whatever technical level you need. You can zero in on a specific question or broaden it out to increase your understanding from ground zero.

An example: Learning Javadoc tags

The other week I was trying to refamiliarize myself with all the Javadoc tags (p. 712) I needed to know to make sure the Java project I was documenting was properly tagged. One area I specifically wanted to focus on was the `@link` tag. I prompted AI for a course on Javadoc tags:

> i'm a technical writer documenting a java api. i want to better understand how to tag the source with comments that get generated as javadoc. create a course that i can follow. i want you to teach me step by step, like a friend.

You can see ChatGPT's response and my thread here. For general topics like this, in which there are likely abundant sources online to train the model, the responses are pretty good.

If you read my thread, you can see that I got stuck along the way and was able to use ChatGPT to work through issues. For example, I had a sample Java project (p. 0) that I wanted to use to experiment with tags, but I'd forgotten how to generate the Javadoc (p. 707). Running the command that AI gave me didn't work and resulted in errors, so I asked it for help in sorting through the errors. In a few minutes, I installed the right VS Code Java extensions and got back on track.

The ability to zoom in on specific issues, work through error messages, and control and pace and direction of the course makes for a phenomenal learning experience. If you're not already an AI enthusiast, using AI to learn a technical topic like this might make you a convert.

During the AI-generated course, I could also ask why my sample code didn't work. For example, as I experimented with the `@return` tag, the Javadoc didn't show the return comment in the output. Why not?

> i added a return tag to the following method, but i don't see a return

> parameter in the output. Why?

Not only did the AI explanations cover the general reasons (the method used a `void` tag, which meant it didn't return anything, so my `@return` command was ignored), but it used the specific method names and details from the example in its explanation. There was no need for me to adapt and fit general knowledge to my specific problem.

In other areas, I wanted to expand my knowledge of specific documentation style around the `@param` tag.

> what should the syntax be with @param? do i start with a lowercased verb phrase? what happens if there's a period in there? does the javadoc output put the content after the period in another place?

Again, the AI was able to provide information specific to my documentation-oriented interests. In other places, I wanted more detail as well, such as how to handle definitions for the fields returned by the method.

> for the @return tag, in the example above you said "the area of the rectangle." suppose my method returns a bunch of data, such as rectangle area, rectangle style, rectangle depth. where would i define those definitions? in the method? or elsewhere?

AI explained that field definitions are often encapsulated in another class. Now things were clicking and I started to understand the structure of the code I was documenting better.

What I appreciated most about the Javadoc course is that I could customize the content specific to my needs as a technical writer. Almost every book on Java I've ever read has been targeted for developers. There simply aren't books on Java written for technical writers. There might be a brief section on Javadoc nested within a gargantuan book that will take weeks to get through. And after making your way through the thick text or online course, do you ever get the answer to your documentation questions? Often not. Instead, using AI, I could go right to documentation-specific questions:

> tell me more about the {@link} tag. what's the reason for the #? if i am linking to a method inside a class, how would i do that?

The structure of the `@link` tag can be pretty complex, actually, especially when you're trying to link to a section within a class in another package.

Finally, I wanted to focus on the larger question:

> as a tech writer, i need to identify places in the java source files that are missing the needed tags. how can i do that?

Overall, this AI-generated Javadoc course probably took me an hour to get through. Granted, if you spend 10x the time learning Java in a course targeted at developers, you'll probably have a much greater understanding. But here's the problem: I've read through multiple books and gone through online courses for programming languages. The experience is almost always the same:

- The material teaches you the basics, but those basics won't help much with documentation-specific questions beyond maybe orienting you to the right terminology.
- The material doesn't stick long term. After a few months of not using the code, I forget it.
- It takes too long to get to relevant material. It might be hours into a course before I even get to something that feels relevant.
- I'm a passenger in someone else's car. I go where the driver takes me, and that driver assumes I'm a developer learning to code, not a technical writer trying to understand how to document that code.

The larger challenge: Finding the time

Whether you learn technical topics from AI, online courses, books, or other methods, the larger challenge is simply finding the time to do it. Chances are, you're busy as a technical writer, and you have an endless number of tickets and deadlines. You don't have time to carve out two hours a day for general tech learning. It takes a tremendous amount of discipline and long-term commitment to make your way through any learning endeavor.

The larger skill needed is the long-term, relentless commitment to a learning regiment. If you can set aside even one hour during the day to learn, and do that consistently over not just weeks but months and years, that will help you be a successful technical writer. How exactly do you muster this discipline for learning? What approaches or systems actually work?

Here's what I've found that works best:

- Focus on learning code related to the documentation projects you're working on. If someone isn't relevant, it will get deprioritized easily.
- Recognize what kind of learning is helpful for documentation tasks. You're a technical writer, not a programmer.
- Keep plugging away little by little. You'll be surprised the amount you can learn spending just one hour a day for a couple of weeks.

I'm confident that using AI tools might lead to more efficient code learning. Instead of taking hours to get to something relevant, I can go right to those topics and questions. Instead of pretending to be a junior developer learning to code, I can fashion the course so that it's tailored to technical writers. Instead of getting stuck on errors or other unexplained obstacles and going down Stack Overflow rabbit holes, I can use AI to work through them. Learning to code might be a whole lot easier with AI.

Best of all, AI tools excel at general programming topics and understanding, which might be too superficial for a programmer working at an advanced level but could be a perfect fit for technical writers. We usually don't need an advanced level of understanding, only a grounding in the basics, a familiarity with the right terms to use, and other beginner-type details for a programming language.

Activity

To put theory to practice, try the following:

1. Choose a technology you want to learn more about.
2. Customize this prompt:

> I'm a technical writer documenting a [Java API]. I want to better understand how to [tag the source with comments that get generated as Javadoc]. Create a course that I can follow. Teach me step by step, like a friend.

3. As you're learning about the topic, steer it in the direction that would be most helpful and relevant to you as a technical writer.

More resources

For more resources on this topic, see the following:

- You're using ChatGPT WRONG: How I would learn to CODE in 2023 with the help of AI

Using AI for language advice

When I first started using AI tools, I tried to get them to write blog posts or other large pieces of content. These efforts almost always failed. The content sounded flat, unassertive, impersonal, boring, and voiceless. For more on why AI can't approximate real human writing, see Lauren Hartenberger's insightful essay What AI Teaches Us About Good Writing. Hartenberger says that human writers often purposefully deviate from convention or rules as they engage with topics in authentic ways; this makes their writing connect emotionally with readers.

But even if we don't use AI tools to write content for us, it doesn't mean we have to abandon the language capabilities of AI tools. AI tools actually work great for language advice. One common task I do as a technical writer, on a near daily basis, is review content that others write. When I put on my editor's hat, I want to make precise, knowledgeable comments that reference grammar or stylistic rules rather than just rewriting sentence so they sound better. In particular, I've found that engineers want to know specifically why their sentences are wrong. They frequently reject comments that say "It sounds better this way." Sounds better to whom? To you?

Even 20 years ago, when I was in graduate school teaching first-year composition classes, students wanted precise references to grammar rules. In part this is because writing "awkward construction" or something similar didn't help them understand what was wrong. The composition director required all composition instructors to not only circle every grammar and style error but also to reference rule in a grammar handbook. By the end of the year, the pages of my grammar handbook looked so used, the book resembled someone's favorite novel. (Trust me, it wasn't my favorite book to read.)

AI tools can do an excellent job at identifying the particular grammar or style rule or reason, and they can provide guidance about why one phrasing is preferable to another. In asserting a preference, AI will often make a convincing argument for one style over another, such as noting that a word could be interpreted in different ways and so is more ambiguous than the other phrasing.

No matter your command of the language, articulating and referencing grammar and style rules can be helpful when you put on your editor's hat. Try this initial prompt for language advice:

> As a technical editor for a documentation group, you are an expert in grammar and plain language. Provide guidance about the following usage options, indicating which is better and explaining the rule behind it. Reference grammar and style rules by name in your reasoning.

After using this prompt, you can just paste in multiple similar-sounding sentences and the AI will tell you which is better and why. If you're worried about privacy, note that it's easy to swap in generic terms while preserving the sentence's structure. Here are some examples:

> You first click the red button, and then you click the blue button.
> You first click the red button and then you click the blue button.
> You first click the red button, then you click the blue button.

The system is composed of five modules.
The system is comprised of five modules.
The system is made up of five modules.

The app displays a screen that says...
The app shows a screen that says...

The user must opt into the terms...
The user must opt in to the terms...

The widget allows you to try the feature...
The widget enables you to try the feature...
The widget lets you try the feature...

With the widget, you can do the following:
- Schedule flights
- Manage bookings

With this app, you can:
- Schedule flights
- Manage bookings

One thing to remember — set the initial parameter.
One thing to remember: set the initial parameter.

The function accepts two arguments...
The function accepts two parameters...

Many vehicles drive on this road. For example: trucks and RVs.
Many vehicles drive on this road — for example, trucks and RVs.

Enter your desired input into the form.
Enter the input you want into the form.

The API only searches for lakes.
The API searches only for lakes.

This is the master view...
This is the main view...

You must be added to the safelist to call the API.
You must be added to the allowlist to call the API.

Enter the input to the widget, for example: "acme" or "beta."
Enter the input to the widget — for example, acme or beta."
Enter the input to the widget; for example, search for acme or beta."

You can perform various actions (configure the widget, initialize the system, etc.) as an administrator.
You can perform various actions (such as configure the widget or initialize the system) as an administrator.

You can configure permissions via the app.
You can configure permissions using the app.
You can configure permissions through the app.

Use the library's methods to make your requests.
Use the library methods to make your requests.

To make requests, you must have an API key.
To make requests, you need an API key.

Over the years, the university provided me with a great education and set me on the right path - which I want to share with you now.
Over the years, the university provided me with a great education and set me on the right path, which I want to share with you now.

The widget has various buttons, namely, a power button, an off

> button, and a channel button.
> The widget has various buttons, namely: a power button, an off button, and a channel button.

You can see ChatGPT's responses to a lot of these prompts here.

As you can see, a lot of sentences have subtle differences in grammar and style, with no clear answers about which is right or wrong. Many instances don't have a clear grammar or style rule that one sentence abides by while another doesn't. No matter, AI generally makes a convincing argument about which sentence is better and why. Or if both are equally acceptable, it lets you know. If they are equally acceptable, I try not to change text just to align with my own preferences.

Notice that I included the following in my prompt: "Reference grammar and style rules by name in your reasoning." As you're making editorial comments, if the content does relate to a rule, you can call it out by name. For example:

> Click the big blue button in the upper-right corner.
> Click the big, blue button in the upper-right corner.

Do you remember what grammar rule this comma usage relates to? Rather than just adding a comma there, you can note that in this usage, the adjectives are coordinate adjectives rather than cumulative adjectives. This gives users terms they can use to better understand the sentence's mechanics.

By the way, once you provide the initial prompt as I indicated above, your future grammar questions can simply provide the pairs of similar sounding sentences. AI will understand your intent from the context. You could even bookmark the thread so that with future questions, you just paste in the two sentences without having to re-establish the question.

Update: Since I wrote this post, there's now a grammar check feature soon to be added into Google Search. The Grammar Check is in Bard now. For any problematic sentence, try typing "grammar check" and then paste the sentence into Bard. The response explains any grammar issues, related rules, and suggested changes for the sentence. It's pretty nifty.

Using AI to create doc updates based on bugs

As a technical writer, I'm frequently making doc updates based on user-reported bugs. A doc bug is a fix that needs to be made to documentation that has already been released, rather than documentation for a new feature.

For example, here's a common scenario: users provide feedback about one of the confusing fields returned in an API. The partner files a bug that kicks off a long back-and-forth thread with the engineers. The engineering team might make a fix or unravel the confusion a bit. As a final step to the fix, the engineers create a doc bug for the technical writer to update the documentation. Sounds simple, right? It's usually anything but simple.

The doc bug process

The total diff you submit might be just a few changed lines, but here's what the process usually looks like with a doc bug:

- During a meeting users have with the product team (or other Developer Relations group), the users bring up an issue with the API. Meeting notes briefly describe the issue, but if you weren't there, much of the context is already lost here.
- The meeting notes indicate that an action item is to log a ticket, but the ticket only briefly notes the discussion. It does not add many details, as it assumes that others involved in the project have the context of the meeting.
- In addition to the ticket description, there's usually a dozen or more comments on the ticket, as the implementation often evolves from the original description of the fix. Some bugs might have such a lengthy comment thread that reading it requires at least an hour or more. Really gnarly bugs can have 75+ half-readable comments, with comments linking to other comments. There could also be associated email threads. Discussions in the threads flip back and forth with different solutions and views.
- At the end of the ticket and comment thread, it's not clear what the problem or resolution actually was, or what docs need to be updated. If you're lucky, there's a code change attached to the ticket, but that code change might not be intelligible or straightforward. Someone just fixed the bug and closed the ticket. You have to guess which comment, if any, describes what the actual fix was. The ticket's resolution might have been privately discussed in a team meeting that the tech writer wasn't part of (e.g., all those daily standups engineers attend).
- The product team communicates the fix back to the users through conversations, chat, team meetings, or other sources.
- After implementing the fix, the developers file a bug for the tech writer to make a doc update. However, the developers are overstretched already and don't have the time or interest to add easy-to-read descriptions in the ticket. They quickly file a doc bug, describing the needed change in a vague way, and reference the engineering ticket that describes the issue and fix.
- The tech writer sees the ticket in his or her queue. He or she spends a lot of bandwidth parsing through the ticket trying to figure out exactly what the doc update should be.
- Despite reading through the ticket threads and meeting notes, the tech writer still has questions, so he or she reaches out to the product team for clarification.
- The engineers are working on the next release and don't have much time to devote to explaining the maintenance fixes. Plus, isn't all the information in the engineering ticket and other notes already?
- After spending an hour or two deciphering the needed update and interfacing with engineers

through chat, the tech writer updates a few lines in the docs and sends the change out for review.
- There's a bit of back and forth with the review, with engineers correcting and adding to the tech writer's updates before finally approving the updates and publishing the content.

If this sounds familiar, welcome to the club. Making doc updates from an assortment of bugs isn't at the top of my favorite activities. I can't tell you how many times I've gathered up a long list of doc fixes needed based on bugs only to abandon them after a short while. The work is tedious and often doesn't have hard deadlines.

Plus, once the engineers have resolved the bug, they've moved on to other work. If you ask them questions, they have to re-familiarize themselves with the bug's context as well. The user who complained was usually notified about the fix, and since no one else is making a fuss about the bug, the priority of the update fizzles. Tech writers also tend to focus most of their energy on documentation for the next release, not on bug fixes for existing code.

Decomposing excellence into a thousand tiny fixes

I'm just as guilty as focusing on docs for the next release as anyone. However, last week I had an interesting thought about bugs. One afternoon I wanted to know what our partner engineers focus on, so I started looking at all the bugs they've had in our ticket system. Reading through the tickets, I found that the partner engineers were mostly loaded with small bugs with the product, investigating causes and fixes. The extent of their workload seemed to consist of troubleshooting and fixing bugs, and being the interface between users and the product teams.

And then the larger realization hit me: What if what makes a product excellent isn't some large feature but rather the fixing of thousands of tiny bugs surfaced from users? And if a product achieves excellence through these thousand bug fixes, maybe documentation elevates in the same way? What if documentation achieves excellence through a tech writer's attention to a thousand tiny user-reported bugs? Fix a sentence here, a definition there, a concept on this page, a section on another page, and so on, accruing 1,000 bug fixes until the docs become truly next level.

If a thousand tiny fixes paves the road to a great user experience, then I need to develop a much better system for triaging and fixing bugs—not just identifying them, but powering through them on a regular basis. In short, I need to develop a better system for making updates based on bugs.

Using AI to help with bugs

Can AI help in making doc updates based on bugs? Yes, I think. This is one application I'm particularly excited about. I've only just started trying this out, so keep that in mind. Here's my general approach to using AI to help with bugs:

First, copy all the relevant information from the bug, meeting notes, code diffs, and other sources discussing the bug into a Google Doc.

Then use an AI tool with a high input limit, using the gathered content as an input source. Some tools you could potentially use for this include Claude.ai, which accepts an extraordinarily high input length, and NotebookLM, which is still in its experimental phase and so access is limited, but you can join the waitlist. (See Introducing NotebookLM for more details.) Other AI tools could also be useful as well. Always consider your company's data policies when using AI tools. In this case, due to data confidentiality, I'm using NotebookLM.

After you've gathered the source material, ask the AI a question such as the following:

> Based on the input sources, provide a summary of the issue {about ...} and the resolution taken to fix the issue.

If you can supply a pattern based on the output you want (p. 842), great. For example, usually the release notes describe the fix like this:

> A bug related to [X feature] was fixed. Previously, [X feature] was doing [Y]. The [X feature] was updated to produce [Z]. For more details, see [X feature].

I haven't quite nailed the prompt here, as I'm still experimenting, but so far this has been helpful. In my experiences, the AI's summaries and other explanations give me confidence that I'm interpreting the bug correctly. I can also ask for simplified explanations if the AI summaries don't make sense.

Note that even if AI provides a beautiful summary and resolution in the style of a release note or other format, it will likely need tweaking and adjustment. Plus, you have to figure out where to apply the update in the docs, and then graft in the update smoothly within the context around it. The AI tool won't do that for you. You might need to make updates across the docs on multiple pages. Searching for all the sections that need to be updated can be time consuming, as well as sharing links to previews to all those updated pages in a changelist that you review with developers.

So the process is still time-consuming, but hopefully AI has done some of the heavy lifting with the bug interpretation and reduced the bandwidth required to parse and interpret engineering shorthand and jargon.

Like I've said before, I'm still experimenting with these techniques. But my hope is that using AI to more quickly fix bugs might lead to a speedier and less painful process related to bug fixes. If I can fix twice as many bugs in half the time, it will be a huge win.

Experience in fixing a bug and some risks

Recently I was tackling a difficult bug and realized some adjustments I needed to make in my input sources. The bug I was working on had an associated engineering doc. As is common in the genre, the engineering doc defined the problem, presented a proposed solution, then presented several alternate solutions. I had to reach out to engineers to find out which approach was taken, and which sections were future-facing versus content that had been implemented. Before adding this content as an input to the AI, I stripped out the alternate approaches and future-facing roadmaps, as this confused the AI's responses.

The bug I was working on followed all the hallmarks of what I described earlier. The concepts were actually extremely technical and hard to understand. I found myself asking the AI to define terms, clarify concepts, and simplify explanations over and again. In reading the responses, I had to check their validity.

To check, I couldn't rely on the explanations it generated as truth. So I also read through the engineering tickets, the engineering docs, the email threads, and more. I also had some chats with the product team. You might think, geez, if you have to do all that, what's the point of using the AI? The point is that AI made everything much easier to understand.

Consider an analogy here. Suppose that instead of a technical problem, you're trying to describe a Calculus-related bug with some equations and the adjusted approach that the team took. Well, it's been 25+ years since I took Calculus, and so despite the explanations that the math team might have written up, it still sounds Greek to me. There's too much jargon, needed background information, and other detail for the content to make sense without some serious study.

But what if AI can make that math content more readable, as it defines unfamiliar terms, simplifies explanations, provides examples, and explains and re-explains things as many times as I want? In this way, AI becomes a kind of tutor, helping me make my way through the landscape more efficiently.

More and more, this is how I'm using AI. Sure, AI could also probably draft some content, and I could copy and paste it into an update that I share and review with engineers, without fully understanding what I'm sending, but that process doesn't scale well. It might present a short-term gain while compromising long-term productivity. It's important for technical writers to have at least a basic understanding of the issues and solutions for the products they document. That knowledge becomes helpful as you work on other parts of the documentation.

For example, understanding the issues in the bug might unlock understanding in other documentation areas, or raise awareness of other content that needs expansion or clarification. All that insight becomes lost if the tech writer just blindly copies and pastes what AI writes and relies on engineers to review it. That said, if the technical writer leverages the AI-written content as a kind of Cliffnotes to understanding a longer Shakespearean work, it can be a practical, efficient way to learn just enough without spending multiple days fixing a single bug.

Another risk: rubber-stamped LGTM

Relying on engineers and product team members to vet AI-written content also poses another risk: rubber-stamped LGTMs (looks good to me). By rubber-stamped LGTMs, I mean approvals given without careful analysis and understanding of the content. This YouTube video How we write/review code in big tech companies ↗ captures the scenario perfectly. In the video, an engineer doesn't fully understand some code but makes an attempt at a fix and submits the updated code for review, assuming that the reviewer will vet the code and examine whether it's correct. However, the reviewing engineer doesn't fully understand the code either and just adds a quick LGTM and approves it. Although this scenario seems comical, there's truth to it. A lot of people will LGTM a fix without really understanding it or having knowledge to know whether it's right or wrong. They just want to get the pull request or changelist out of their queue, and they assume the author knows the information is correct.

In the case of AI, suppose the tech writer copies AI-written content that looks and sounds plausible but isn't. The reviewer reads it and, not realizing it's written by AI, gets duped by the believability of the content (AI tools are great at sounding convincing). Unless the tech writer discloses that the content was written by AI, chances are the reviewer won't be extra cautious in the review. At least when humans are writing, tech writers are much less likely to slip into fiction mode and will press engineers for answers.

Gathering more context

As I described earlier in the typical bug scenario, the main challenge is lack of information and context, which makes the bug hard to understand and take action on. One step I've taken to improve the amount of context/information available is to require people filing bugs to dig deep in providing needed information. Here's a sample bug template that tries to squeeze all the possible information from a requester. I generalized the info, but reading through it you will see my point.

Please provide answers to all questions below. Note: Supplying as much info as possible below avoids the need for a follow-up meeting with you to get more details. Remember that we have almost no context for the doc update (no partner meetings, no engineering coding sessions, no post-mortems, etc.), so the information you provide is critical to making this doc update actionable.

1. What release is this for? By providing the release, we can better prioritize this work against the other bugs in our queue. If the update is unrelated to a release, indicate “No Release.”
2. What’s the priority for this update? In your view, is this a P0, a P1, P2, or P3? (Feel free to explain/justify a P0 request.)
3. Which API are you updating?
4. Did you review this change with the API steering committee? If so, provide any related bugs and change request documents presented to the council. If there’s a product launch entry, include the link too.
5. Are there any other source materials (for example, pitch docs, engineering docs, product review documents) that describe the updates? In the documents, please note which solution (among several that might be proposed) was implemented.
6. Is there an engineering bug related to this doc request? For example, an engineering bug that points to a change request with some code diffs? Are there other bugs that provide more detail as well? List all related bugs that might inform needed doc changes.
7. If the changelist with related code changes isn’t included in the related bugs, please list the changelists here. From the diffs, we can get a better sense of needed doc updates.
8. Are there any email threads that you can PDF and attach here (or forward)? Any meeting notes that you can point us to? Anything that provides more context for us?
9. What proto, class, or other source files are being changed in this update? (Since docs are generated from source comments in proto and public class files, identifying the source helps us make any changes to comments.)
10. Write the release note for this update. Be specific about field or class names. Provide as much detail as is needed for users to understand the update.
11. Indicate the release status of the feature.
12. Does the field/method, etc. appear in other APIs? Which ones? (We want to avoid having duplicate and conflicting descriptions of the same fields across different APIs.)
13. Does this change affect any code samples in the documentation?
14. Who should review and approve this content before we submit the updated documentation? Include any relevant stakeholders as well.

If you have any questions, reach out through email or chat.

As you can see, filing a bug will be time-consuming and somewhat painful for the requester, but ever so sweet for the tech writer. Imagine having all of this information gathered at the time someone requests an update. It might actually make the bug actionable!

From this gathered context, you can copy the information into a single Google doc, remove all the parts that might mislead an AI, and then provide the doc as context. Some parts to remove might include the following:

- Wrong answers in email threads
- Alternative solutions never implemented
- Future roadmaps not yet relevant
- Unnecessary fluff unrelated to the needed updates

Conclusion

Although there are many potential issues with using AI, fixing doc bugs is one area that seems ripe to leverage AI tools. I'll continue exploring different techniques, seeing what works and doesn't, and sharing my experiences here.

Using AI For thematic analysis

One of the main ways I use AI is with thematic analysis, which involves identifying, analyzing, and reporting patterns (themes) within qualitative data. After you identify major themes, you can use least-to-most prompting techniques to go into more detail.

I recently used this technique in preparing notes for a book club and, though I haven't yet used it for API documentation, I think it could work well for a number of documentation-related scenarios.

Step 1: Thematic analysis

Let me introduce thematic analysis through an example. I run a book club at work focused on auto and transportation topics (since I work with geo-related APIs for maps in cars). I haven't talked about this book club much on my blog, but it's something that grew out of my Journey away from smartphones series.

Every month we read a book and then meet to discuss it. As the book club facilitator, I select the book and prepare notes and discussion for it. Like everyone else, I don't have a lot of time for this preparation, given that I squeeze it in among work tasks.

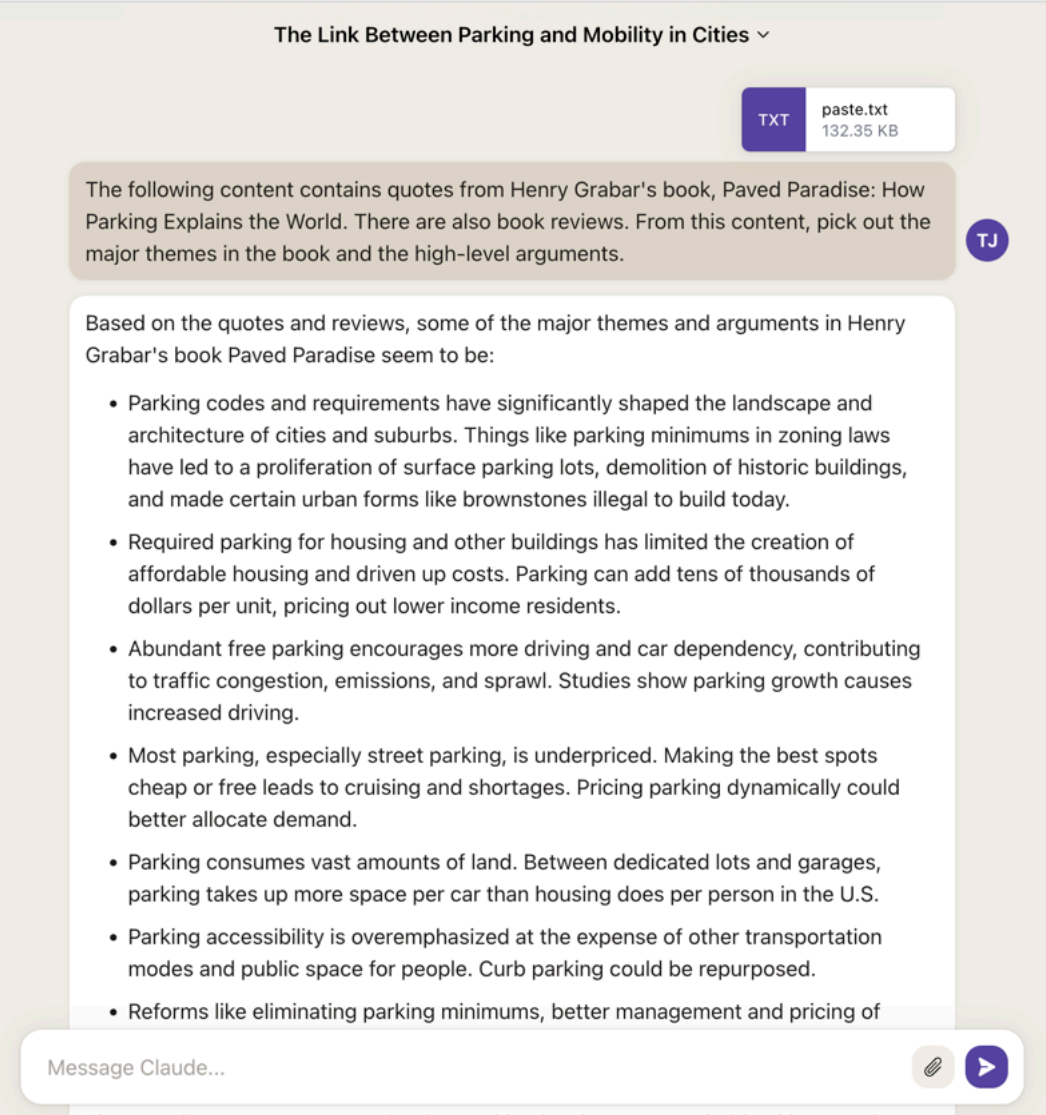
The latest book we read was *Paved Paradise: How Parking Explains the World*, by Henry Grabar. I like to annotate pages as I read because it makes the reading experience more active. I usually finish the book a few days before the book club meeting. In a short amount of time, how do you prepare detailed notes and discussion from a 280 page book covering a variety of topics?

Here's my process:

1. Go back through the book and *voice type* the key passages and ideas into a Google Doc. (I used to type the notes out; then I discovered voice typing by going to Tools > Voice typing in Google Docs, and it sped things up.) This part is the most tedious because I put stars next to far too many passages when I read. However, collecting this material is key to creating the input source for the AI. I don't mind if the voice-typed passages have errors because the AI tools are generally great at correctly interpreting the intent and meaning anyway.
2. Find about 10 book reviews and copy and paste the review content into the same Google Doc.
3. Ask a high-level question to the AI. I chose to use Claude.ai because it accepts a large input, and also because the book club doesn't involve confidential data. (If you haven't tried Claude, it's pretty amazing. The large text input allows you to supply more context to the AI, which can prevent the wacky hallucinations and fictitious responses.) Here's the prompt I used:

   > The following content contains quotes from Henry Grabar's book, Paved Paradise: How Parking Explains the World. There are also book reviews. From this content, pick out the major themes in the book and the high-level arguments.

   Here's a screenshot:



The following content contains quotes from Henry Grabar's book, Paved Paradise: How Parking Explains the World. There are also book reviews. From this content, pick out the major themes in the book and the high-level arguments.

Based on the quotes and reviews, some of the major themes and arguments in Henry Grabar's book Paved Paradise seem to be:

- Parking codes and requirements have significantly shaped the landscape and architecture of cities and suburbs. Things like parking minimums in zoning laws have led to a proliferation of surface parking lots, demolition of historic buildings, and made certain urban forms like brownstones illegal to build today.
- Required parking for housing and other buildings has limited the creation of affordable housing and driven up costs. Parking can add tens of thousands of dollars per unit, pricing out lower income residents.
- Abundant free parking encourages more driving and car dependency, contributing to traffic congestion, emissions, and sprawl. Studies show parking growth causes increased driving.
- Most parking, especially street parking, is underpriced. Making the best spots cheap or free leads to cruising and shortages. Pricing parking dynamically could better allocate demand.
- Parking consumes vast amounts of land. Between dedicated lots and garages, parking takes up more space per car than housing does per person in the U.S.
- Parking accessibility is overemphasized at the expense of other transportation modes and public space for people. Curb parking could be repurposed.
- Reforms like eliminating parking minimums, better management and pricing of

The result was a list of major themes and high-level arguments in the book. The ability to synthesize general themes from small pieces of data is a technique used in many domains. For example, when researchers make sense of feedback, they often code the feedback with terms and then quantify the instances of the terms.

The following sections list other use cases for thematic analysis, more applicable to tech comm.

Thematic analysis of job ads

I wrote about this technique in Looking at job advertisements to extrapolate the evolution of tech comm. In “The Evolution of Technical Communication: An Analysis of Industry Job Postings” (Nov 2015 Technical Communication Journal), Eva Brumberger and Claire Lauer analyze hundreds of job advertisements, looking for patterns. They want to know what skills employers are looking for so they can prepare their students to be successful in the job market.

After identifying the patterns in the job ads, they quantify them to assess trending patterns. From this research, they arrive at a list of desired skills. The most sought-after professional competencies for technical writers/editors include written communication (75%), Editing (51%), Project planning/mgmt (49%), Visual communication (49%), Subject-matter familiarity (45%), Working with SMEs (41%), and Style guides/standards (40%).

Thematic analysis of feedback

Affinity diagramming is another common technique used by UX researchers to synthesis user feedback into more actionable groups and themes. An affinity diagramming session starts with writing hundreds of qualitative statements on post-it notes; then a dozen or so people group the post-it notes into thematic groupings. Finally, someone writes a statement that expresses the grouping of items. This technique allows researchers to go from hundreds of points of data (fragmented, ungrouped, and miscellaneous) into actionable insights.

Thematic analysis for search analytics

You could also do thematic analysis for search analytics. Previously, I wrote a post called What's the point of site search? The problem is that you end up with a lot of little pieces of information that are hard to analyze. Specifically, see the section The actual searches people are making on my blog. It's hard to extrapolate themes by looking at the individual keywords. If I wanted to do it scientifically, I could code each search result with a keyword, then group all searches that have the same keyword, then analyze that grouping to identify common themes, and so on. This task is usually too tedious for any technical writer to undertake.

But if you plug the same long list of keywords into Claude.ai and use a thematic analysis prompt, you can quickly arrive at thematic analysis:

Prompt:

> The following is a list of keywords that people search for using my site search. from this list of keywords, synthesize the predominant themes.

Response:

> Okay, here is my analysis of the key themes based on the list of keywords you provided:
>
> - Technical writing skills - There are many keywords related to core technical writing skills like "technical writing," "simplifying complexity," "readme," "instructions," "troubleshooting," "getting started," "tutorials," "API documentation," etc. This indicates visitors are interested in developing technical writing abilities.
> - Tools and technologies - Terms like "AsciiDoc," "DITA," "Markdown," "Jekyll," "Hugo," "CMS," "static site generators," "git," "GitHub," etc point to an audience looking

for information on documentation tools and coding/technology skills.
- Career development - Keywords such as "skills," "interview," "salary," "freelance," "laid off," "starting as technical writer," and "jobs" suggest visitors want to develop their tech comm careers, find work, and interview successfully.
- Writing techniques - Queries like "minimalism," "every page is page one," "simple language," "progressive disclosure," and "context-sensitive help" revolve around writing methods and best practices.
- API documentation - Many keywords focus exclusively on API documentation, like "API documentation," " OpenAPI," "API reference," "endpoint documentation," "Swagger," etc. Developing API skills is a priority.
- DITA and structured authoring - There is heavy emphasis on XML, DITA, content types, specialization, etc, pointing to strong interest in structured authoring.

Any time you have a thousand pieces of information that you need to analyze for patterns, your first-stop shop for analysis should probably be an AI tool.

Thematic analysis for tags and related pages

Another use case could be tagging pages. Most technical writers work in developer portals with hundreds of pages of content. In many cases, it's a best practice to create a taxonomy describing the content, so that pages with the same taxonomy terms can be surfaced as related pages. As a user, you're probably accustomed to seeing Related pages or Recommended pages.

To power these related pages, writers would need to index the keywords from each page in their content, and then standardize those index keywords in a structured list of terms (for consistency). From those keywords, a script could surface other pages that have similar tags.

Again, this sort of task is extremely tedious. There's a reason many of us became technical writers rather than professional indexers. AI tools could speed this up considerably.

Thematic analysis for FAQs

Another use case for thematic analysis is with FAQs. Some websites have hundreds of FAQs for their sites. When I worked at Amazon, we had a giant FAQ document that people kept adding to over the years. Eventually we divided it into smaller FAQs (see Index of FAQs ↗), but even this index page was embarrassing. Generally, it's better to structure the information from FAQs into the topics where the information belongs, plugging the information into the larger body of information rather than cobbling it together in a random list. (See The problem with Frequently Asked Questions (FAQs) in documentation ↗.)

Using the techniques here, you could use AI tools to group the FAQs into themes. With this approach, you might go from 100 randomly ordered FAQs into half a dozen thematic groupings. But then from those thematic groupings, you could use a prompt like this to match the FAQs with their appropriate pages:

> The following is a list of FAQs for my site [paste FAQs]. Here also is a site index listing all the web pages of my site [paste index of web pages]. For each FAQ, list the most appropriate web page this information should appear on.

You get the idea here. There are certain scenarios where we have lots of individual pieces of information. From these pieces, we can use AI tools to synthesize larger groupings and themes. This is one area that AI tools perform much more efficiently than humans.

Thematic groupings of glossary items

Another use case for thematic analysis is with glossary items. With glossaries, we have hundreds of small pieces of information. Usually, the glossary items have a searchable filter at the top, and people come in, look up one term, and then continue on in their information journey.

But what if you want to help users learn a collection of terms for a specific situation. For example, rather than listing hundreds of terms in alphabetical order, you could provide thematic groupings of terms to help people learn specific domains. If you were writing about finance, you might have a “Fixed Income Securities” glossary group that includes *bond*, *yield*, *maturity rating*, *credit rating*, and *coupon rate*. You could create another grouping titled “Investment Fund Types” that includes the terms *mutual fund*, *index fund*, *exchange-traded fund*, *hedge fund*, and *money-market fund*.

By creating specialized glossaries, you draw attention to related terms and can help people differentiate between the terms.

Thematic analysis of bugs

In looking for scenarios to apply thematic analysis, who can forget bug lists? Suppose you have 150 bugs in your team’s ticket management system (e.g., JIRA). You probably want to group them into different sublists so that you can tackle theme more efficiently. You might want to group all the bugs about the “ACME API” in one list, all the bugs about the “Omega API” in another, all the bugs about the “service widget” in another, and so on. Sorting through the bugs is time consuming and tedious.

To leverage AI, you could first use summaries to generate more appropriate titles for each of the bugs. Then you could export the bugs into a long list. Feed that list into an AI to sort them into thematic groupings. Then from those groupings, apply group tags to the bugs.

But don’t stop there. What if you were to analyze the bugs of all the software teams you support? Let’s say you support 5 teams, and each team has their bugs in different components. Export the bugs of a component, then feed them into an AI to group and classify the bugs. Prompt the AI to identify common patterns. Using this technique, you could identify problem areas to focus on. In my experience, many engineering bugs could benefit from some documentation updates (clarifying notes, notes about gotchas and bugs, etc.), but engineering teams don’t often think of the bugs as documentation-related. Reading through random bugs tends to be more tedious than tech writers can stand.

Taking this one step further, suppose you’re an internal documentation team that supports hundreds of engineers. You have limited bandwidth and must identify which projects to support. Where do internal developers feel the most pain? Which projects have the most usage plus the most tickets?

Here again you could leverage AI for the analysis. You have a list of the most popular pages (based on your analytics). You also have a long list of keyword queries (from search analytics). Then finally, you have a list of hundreds of bugs. Plug that data into an AI and see if you can use its analytical engines to arrive at the most important projects to work on.

Thematic analysis of documentation pages

Finally, let's explore one more area where thematic analysis might apply: organizing documentation. In every documentation project, you have a long list of individual topics. As part of the organizing task of documentation, you typically group those topics into different folders. The many folders then form the sidebar groupings in our help systems. For this API course, there are about 15 different folders, each with an average of 10 or more topics.

Figuring out the right way to organize topics is not easy task. If you've ever flipped the tables and looked at a help system from the perspective of a user, you know that locating the right topic is anything but intuitive.

As a technical writer, if you want to get your documentation system's organization right, the best approach is to have a handful of real users do a card sort with your topics. You could print your topics on little pieces of paper and have users arrange them into the groups that make the most sense. (Tools like Treejack do this digitally.) But who has time to actually do this? Most technical writers don't have users at their disposal, nor the time.

In this scenario, you could use AI to do two things:

- You could ask an AI to group your topics into the most logical groupings by theme, based on the topic titles. This would at least give you the benefit of another perspective. You could even prime the AI with a specific persona to induce a different perspective (e.g., AI by saying, "You are a new user…", "You are a business executive rather than a coder…." You are an advanced developer…")
- Then you could analyze the AI groupings against your own groupings to identify areas that might be less intuitive. If the AI tool predicts that a topic about "API errors" would more logically appear under "Support" than under "API reference," a move might be worth considering.

Each of these scenarios involves sorting and grouping lots of little pieces of information. This is a common cognitive task that we perform countless times in our role as technical writers. (Information synthesis is a topic I wrote about in my series on Simplifying Complexity: Principle 3: Ensure information harmony in the larger landscape ↗.) We are frequently taking small pieces of information and fitting them into a larger information landscape.

Step 2: Expansion of detail

Now that we covered thematic analysis in a variety of scenarios, let's explore the logical follow-up to the thematic analysis: detail expansion. Specifically, expanding the detail using a prompting technique known as least-to-most prompts.

In the context of AI prompt engineering strategies, least-to-most prompts involve asking for high-level information/outlines first, followed by subsequent prompts that dive into each point for greater detail.

In *Prompt Engineering for Generative AI,* James Phoenix and Mike Taylor describe the least to most prompt technique as follows:

> The Least To Most prompt technique allows you to sequentially create/extract more knowledge in any topic.
>
> This is often a series of prompt chains, with each prompt input utilizing the generated information from previous prompt outputs.
>
> This technique can also be applied to code generation, as demonstrated in a Flask Hello World app example.
>
> …
>
> First, we start by asking the AI to outline the basic architecture for a simple Flask Hello World application.
>
> …
>
> Next, we request the AI to provide the code snippet for a simple "Hello World" route in Flask. (See "Chapter 2. Standard Practices for Text Generation")

In other words, you start by asking for high-level information first; then based on the responses, you zoom into the detail for each response.

You already asked for high-level information through the thematic analysis prompts. It's now time to zoom into the details of those high-level themes.

In many ways the least-to-most technique is just common sense, but it's worth noting that it mirrors the same strategies technical writers use when breaking down complex information into usable documentation: we start with a high-level to paint the big picture, then break it down into detailed sections and granular steps. This style of documentation helps the user complete the larger, more complex task.

When you're working with AI tools to help you address a large, complex issue, you can follow the same strategy: start at the high-level to get a general picture, then zoom into each of the general points with a more specific section and details.

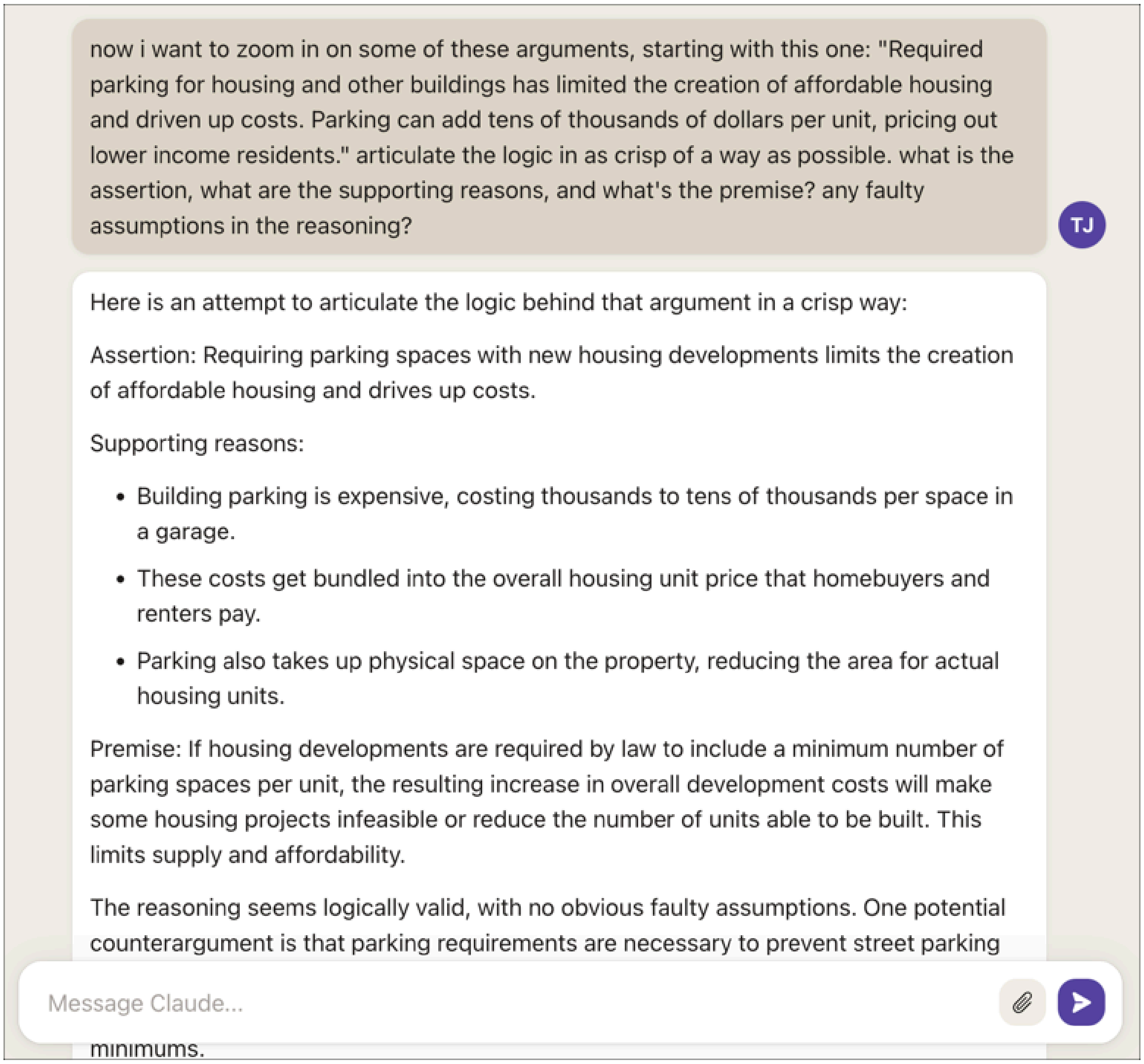
Book club

With my book club, after I gathered a list of high-level book themes and arguments, I wanted to provide more detail about each one. So I followed up those thematic analysis prompts with this:

Prompt:

> Now I want to zoom in on some of these arguments, starting with this one: "Required parking for housing and other buildings has limited the creation of affordable housing and driven up costs. Parking can add tens of thousands of dollars per unit, pricing out lower income residents." Articulate the logic in as crisp of a way as possible. What is the assertion, what are the supporting reasons, and what's the premise? Any faulty assumptions in the reasoning?

Here's a screenshot:



now i want to zoom in on some of these arguments, starting with this one: "Required parking for housing and other buildings has limited the creation of affordable housing and driven up costs. Parking can add tens of thousands of dollars per unit, pricing out lower income residents." articulate the logic in as crisp of a way as possible. what is the assertion, what are the supporting reasons, and what's the premise? any faulty assumptions in the reasoning?

Here is an attempt to articulate the logic behind that argument in a crisp way:

Assertion: Requiring parking spaces with new housing developments limits the creation of affordable housing and drives up costs.

Supporting reasons:

- Building parking is expensive, costing thousands to tens of thousands per space in a garage.
- These costs get bundled into the overall housing unit price that homebuyers and renters pay.
- Parking also takes up physical space on the property, reducing the area for actual housing units.

Premise: If housing developments are required by law to include a minimum number of parking spaces per unit, the resulting increase in overall development costs will make some housing projects infeasible or reduce the number of units able to be built. This limits supply and affordability.

The reasoning seems logically valid, with no obvious faulty assumptions. One potential counterargument is that parking requirements are necessary to prevent street parking

minimums.

Because I'd read the book, I could easily distinguish whether Claude's responses were on target. About 80% of the time or more, it was accurate—in part because I'd primed the AI with accurate input sources. My approach in compiling the discussion notes was to clarify and articulate the author's main arguments, and AI's responses made this easy.

Could I do this same task myself, without relying on AI? Sure, in the same way I could calculate long-division by hand. It probably would have exercised more critical muscles in my brain to do so, but again, time is the issue.

Overall, this process allowed me to quickly arrive at a good summary of the book's main arguments. I added some more discussion questions (not using AI) at the end of each section, and I was ready to go. The book club turned out to be one of the best of the year!

Applying least-to-most prompts to thematic groupings

Following up thematic analysis with least-to-most prompts doesn't always make sense. But for the scenarios I described earlier, here could be some logical next steps to add more detail:

- **Feedback**. From the general feedback themes, AI can provide more extended summaries of the themes, with some detail. It's probably not enough to list general categories of feedback; people will want summaries that describe what each category contains.

- **Search analytics**. Provide descriptions of the analytics trends. It's not enough to say that there are many keywords searches about "Foo." You want to follow this up with a summary paragraph that describes the information patterns around Foo keyword searches and why this topic is important to users.
- **Tags and related pages**. If you have a list of tags for your pages, like I do here ↗, it would be more helpful to see descriptions of each tag.
- **FAQs**. You could use AI to synthesize/integrate the FAQ into the relevant topic. For example, ask the AI where the best place to add a detail about, says, service widget limitations would fit into the topic.
- **Glossary items**. You could use AI to expand definitions of any poorly defined terms.
- **Bugs**. You could use AI To describe the general bug trends for each high-level grouping, along with noted patterns for the bugs.

Conclusion

In general, the direction we've followed is like this:

- First we synthesized larger themes from smaller pieces of information. In other words, we develop order from chaos.
- Now that we have larger themes, we can zoom into each theme to provide more detail.

These are flows of information that usually take place unconsciously in our minds as we think through problems. I've simply made them explicit here because AI tends to be good at performing these tasks.

Using AI for comparison tasks with API responses

The best scenarios to implement AI are those tasks that humans perform poorly but robots perform excellently. One of these task domains is comparative analysis, specifically comparing two sets of information to identify inconsistencies.

Comparing JSON

The idea for this technique comes from a post by Francis Elliot titled Proofread documented JSON blobs using LLMs. Elliot uses AI to compare an API's output with the documentation. Elliot writes:

> One of my more annoyingly manual tasks when writing API docs is to compare an actual returned JSON payload made with a test Postman call to the JSON structure I've documented.

Here's Elliot's prompt:

> Compare the following JSON blobs. Sort the blobs alphabetically by their keys, then for the sorted blobs, tell me if the blobs are structurally identical in terms of key names. If they're different, tell me specifics of how they're different. Ignore different values for the keys, ignore repeated array items, and ignore empty arrays in the comparison.

API responses can have a lot of fields returned in the response, and the fields returned depend on the input parameters and the available data.

For more background on API responses, see Response example and schema (p. 154). In short, an API responses can be broken down into the following:

- **schema** - this describes all possible fields returned and the rules for which they're returned, as well as definitions of each field
- **sample responses** - these usually provide a subset of the total fields described by the schema, often determined by different input parameters. In other words, if you use parameter `foo`, the response includes `acme` objects; but if you use parameter `bar`, the response includes `beta` objects, etc.

It's this relationship between the schema and the sample responses that makes understanding the API responses difficult. Is a sample response missing certain fields because the data didn't include those fields, because of the input parameters used, or due to error? Are there fields the tech writer documented that don't actually align with the API responses? Are there fields present in the response that aren't listed in the documentation? And which fields might be confusing to users?

This is exactly the kind of task that robots are better are doing than humans(by robots, I just mean LLMs or AI). We're not great at line by line comparison of hundreds of words to identify the diffs between information objects. But this eye for detail is what we need when we write docs. This kind of meticulousness can be taxing and straining cognitively.

There can also be some drift between engineering specifications that a tech writer might have used in creating the documentation (specifications that likely included the fields and their definitions) and the actual implementation.

To identify drift, the tech writer usually runs some sample tests (p. 342) to confirm that the responses match the documentation. But unless your API has only a simple number of fields in the response, this can be a daunting task. The API response might have an array with repeated fields, or it might have deeply nested fields, or other complexities that make it difficult to evaluate. With Java APIs, the reference documentation often names the objects, but those names don't appear in the output.

Overall, this is one area prone to error. Here let's see if AI tools can help with the comparative analysis. Using AI tools, we will ask whether the documentation about our API's responses matches the API's actual responses.

Experiment

I wanted to experiment with comparison tasks around responses to see how useful this would be. Due to confidentiality of data, I used a general public API for this rather than a work project, so my experiment is superficial and exploratory only. I used Claude.ai for this because it allows for greater input length. As I've mentioned before, expanding the input length is a game-changer with API tools because it allows for more input to train the LLM, which leads to more accurate responses and less hallucination.

Here is the scenario: as a tech writer, you're working with some API responses and you want to see if the responses match the documentation. Are there responses that you forgot to document, or responses that don't match the casing or spelling in the docs? Are there missing fields in the response that are mentioned in the docs, and which should be marked as optional?

In this experiment, we'll use the Forecast API from OpenWeatherMap, which is an API I've used elsewhere in this course. First, I created a Forecast API response using Postman (p. 71) or curl (p. 80). (I went over this earlier in the course.) As an alternative to Postman, or if you want to import this command into Postman, here's the curl to make a call to OpenWeatherMap's Forecast endpoint:

```
curl --location 'https://api.openweathermap.org/data/2.5/forecast?zip=9805
8&YOURAPIKEY'
```

Swap in your own API key (p. 69) for `YOURAPIKEY`.

Or literally just paste in this URL into the browser: https://api.openweathermap.org/data/2.5/forecast?zip=98058&YOURAPIKEY

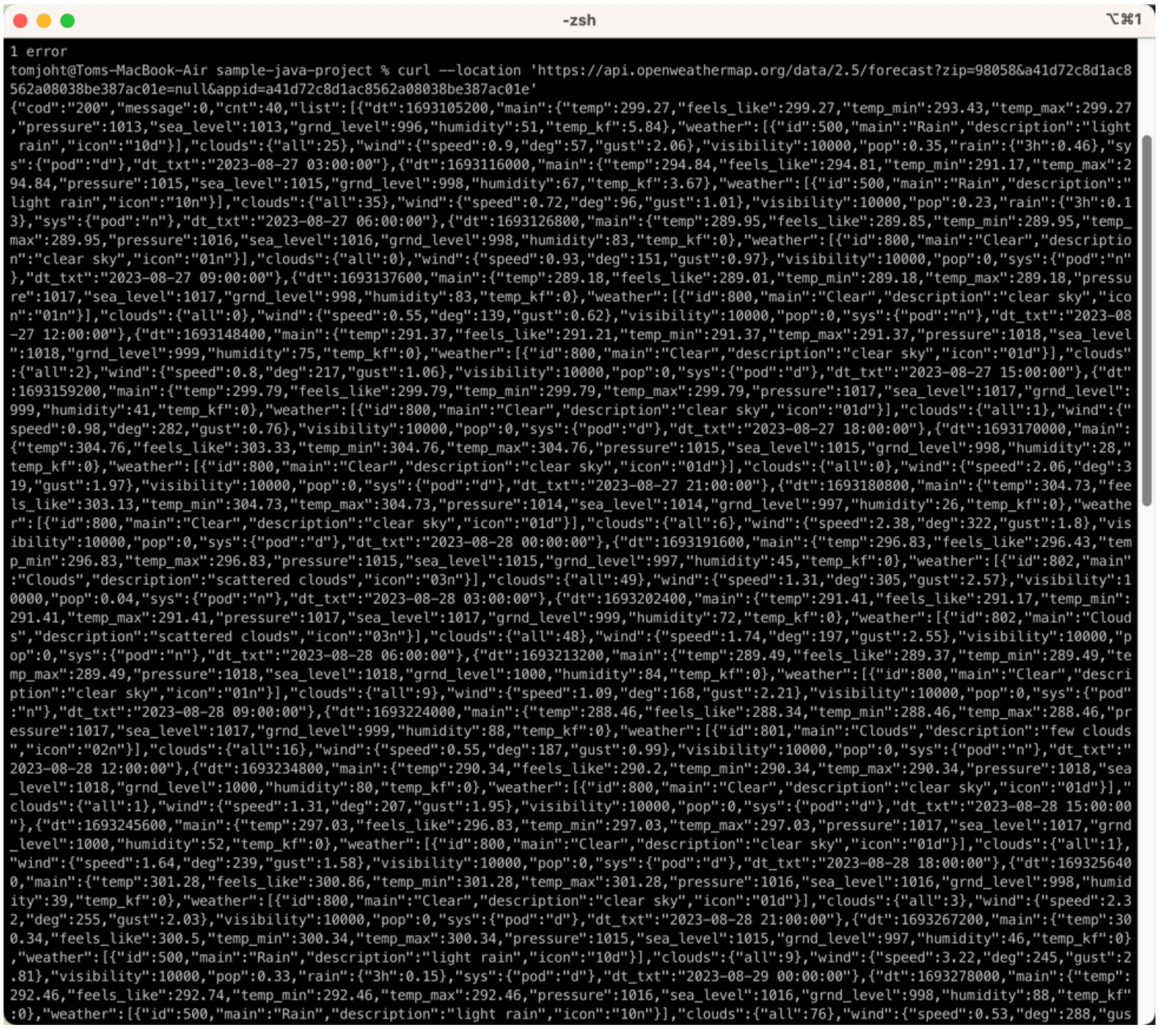
Here's the truncated response:

```
{
    "cod": "200",
    "message": 0,
    "cnt": 40,
    "list": [
        {
            "dt": 1693072800,
            "main": {
                "temp": 293.24,
                "feels_like": 293.03,
                "temp_min": 293.24,
                "temp_max": 299.75,
                "pressure": 1014,
                "sea_level": 1014,
                "grnd_level": 998,
                "humidity": 66,
                "temp_kf": -6.51
            },
            "weather": [
                {
                    "id": 800,
                    "main": "Clear",
                    "description": "clear sky",
                    "icon": "01d"
                }
            ],
            "clouds": {
                "all": 0
            },
            "wind": {
                "speed": 2.69,
                "deg": 352,
                "gust": 3.52
            },
            "visibility": 10000,
            "pop": 0,
            "sys": {
                "pod": "d"
            },
            "dt_txt": "2023-08-26 18:00:00"
        },
...
```

To see the full response for the 16 days, go to forecast-response.txt

The Forecast API shows some of the complexity in evaluating responses. In this case, the API returns 16 days of weather forecast, so the `list` array has 16 objects. (I truncated the sample after the first object.) Each object has some parent fields like `main`, `weather`, `clouds`, `wind`, and `sys`. Then there are some standalone fields: `dt`, `visibility`, `pop`, and `dt_text`.

You can see this so much better in Postman, and you can expand/collapse the sections. In the terminal, the response is compressed and almost unintelligible unless prettified.



```
1 error
tomjoht@Toms-MacBook-Air sample-java-project % curl --location 'https://api.openweathermap.org/data/2.5/forecast?zip=98058&a41d72c8d1ac8
562a08038be387ac01e=null&appid=a41d72c8d1ac8562a08038be387ac01e'
{"cod":"200","message":0,"cnt":40,"list":[{"dt":1693105200,"main":{"temp":299.27,"feels_like":299.27,"temp_min":293.43,"temp_max":299.27
,"pressure":1013,"sea_level":1013,"grnd_level":996,"humidity":51,"temp_kf":5.84},"weather":[{"id":500,"main":"Rain","description":"light
 rain","icon":"10d"}],"clouds":{"all":25},"wind":{"speed":0.9,"deg":57,"gust":2.06},"visibility":10000,"pop":0.35,"rain":{"3h":0.46},"sy
s":{"pod":"d"},"dt_txt":"2023-08-27 03:00:00"},{"dt":1693116000,"main":{"temp":294.84,"feels_like":294.81,"temp_min":291.17,"temp_max":2
94.84,"pressure":1015,"sea_level":1015,"grnd_level":998,"humidity":67,"temp_kf":3.67},"weather":[{"id":500,"main":"Rain","description":"
light rain","icon":"10n"}],"clouds":{"all":35},"wind":{"speed":0.72,"deg":96,"gust":1.01},"visibility":10000,"pop":0.23,"rain":{"3h":0.1
3},"sys":{"pod":"n"},"dt_txt":"2023-08-27 06:00:00"},{"dt":1693126800,"main":{"temp":289.95,"feels_like":289.85,"temp_min":289.95,"temp_
max":289.95,"pressure":1016,"sea_level":1016,"grnd_level":998,"humidity":83,"temp_kf":0},"weather":[{"id":800,"main":"Clear","descriptio
n":"clear sky","icon":"01n"}],"clouds":{"all":0},"wind":{"speed":0.93,"deg":151,"gust":0.97},"visibility":10000,"pop":0,"sys":{"pod":"n"
},"dt_txt":"2023-08-27 09:00:00"},{"dt":1693137600,"main":{"temp":289.18,"feels_like":289.01,"temp_min":289.18,"temp_max":289.18,"pressu
re":1017,"sea_level":1017,"grnd_level":998,"humidity":83,"temp_kf":0},"weather":[{"id":800,"main":"Clear","description":"clear sky","ico
n":"01n"}],"clouds":{"all":0},"wind":{"speed":0.55,"deg":139,"gust":0.62},"visibility":10000,"pop":0,"sys":{"pod":"n"},"dt_txt":"2023-08
-27 12:00:00"},{"dt":1693148400,"main":{"temp":291.37,"feels_like":291.21,"temp_min":291.37,"temp_max":291.37,"pressure":1018,"sea_level
":1018,"grnd_level":999,"humidity":75,"temp_kf":0},"weather":[{"id":800,"main":"Clear","description":"clear sky","icon":"01d"}],"clouds"
:{"all":2},"wind":{"speed":0.8,"deg":217,"gust":1.06},"visibility":10000,"pop":0,"sys":{"pod":"d"},"dt_txt":"2023-08-27 15:00:00"},{"dt"
:1693159200,"main":{"temp":299.79,"feels_like":299.79,"temp_min":299.79,"temp_max":299.79,"pressure":1017,"sea_level":1017,"grnd_level":
999,"humidity":41,"temp_kf":0},"weather":[{"id":800,"main":"Clear","description":"clear sky","icon":"01d"}],"clouds":{"all":1},"wind":{"
speed":0.98,"deg":282,"gust":0.76},"visibility":10000,"pop":0,"sys":{"pod":"d"},"dt_txt":"2023-08-27 18:00:00"},{"dt":1693170000,"main":
{"temp":304.76,"feels_like":303.33,"temp_min":304.76,"temp_max":304.76,"pressure":1015,"sea_level":1015,"grnd_level":998,"humidity":28,"
temp_kf":0},"weather":[{"id":800,"main":"Clear","description":"clear sky","icon":"01d"}],"clouds":{"all":0},"wind":{"speed":2.06,"deg":3
19,"gust":1.97},"visibility":10000,"pop":0,"sys":{"pod":"d"},"dt_txt":"2023-08-27 21:00:00"},{"dt":1693180800,"main":{"temp":304.73,"fee
ls_like":303.13,"temp_min":304.73,"temp_max":304.73,"pressure":1014,"sea_level":1014,"grnd_level":997,"humidity":26,"temp_kf":0},"weathe
r":[{"id":800,"main":"Clear","description":"clear sky","icon":"01d"}],"clouds":{"all":6},"wind":{"speed":2.38,"deg":322,"gust":1.8},"vis
ibility":10000,"pop":0,"sys":{"pod":"d"},"dt_txt":"2023-08-28 00:00:00"},{"dt":1693191600,"main":{"temp":296.83,"feels_like":296.43,"tem
p_min":296.83,"temp_max":296.83,"pressure":1015,"sea_level":1015,"grnd_level":997,"humidity":45,"temp_kf":0},"weather":[{"id":802,"main"
:"Clouds","description":"scattered clouds","icon":"03n"}],"clouds":{"all":49},"wind":{"speed":1.31,"deg":305,"gust":2.57},"visibility":1
0000,"pop":0.04,"sys":{"pod":"n"},"dt_txt":"2023-08-28 03:00:00"},{"dt":1693202400,"main":{"temp":291.41,"feels_like":291.17,"temp_min":
291.41,"temp_max":291.41,"pressure":1017,"sea_level":1017,"grnd_level":999,"humidity":72,"temp_kf":0},"weather":[{"id":802,"main":"Cloud
s","description":"scattered clouds","icon":"03n"}],"clouds":{"all":48},"wind":{"speed":1.74,"deg":197,"gust":2.55},"visibility":10000,"p
op":0,"sys":{"pod":"n"},"dt_txt":"2023-08-28 06:00:00"},{"dt":1693213200,"main":{"temp":289.49,"feels_like":289.37,"temp_min":289.49,"te
mp_max":289.49,"pressure":1018,"sea_level":1018,"grnd_level":1000,"humidity":84,"temp_kf":0},"weather":[{"id":800,"main":"Clear","descri
ption":"clear sky","icon":"01n"}],"clouds":{"all":9},"wind":{"speed":1.09,"deg":168,"gust":2.21},"visibility":10000,"pop":0,"sys":{"pod"
:"n"},"dt_txt":"2023-08-28 09:00:00"},{"dt":1693224000,"main":{"temp":288.46,"feels_like":288.34,"temp_min":288.46,"temp_max":288.46,"pr
essure":1017,"sea_level":1017,"grnd_level":999,"humidity":88,"temp_kf":0},"weather":[{"id":801,"main":"Clouds","description":"few clouds
","icon":"02n"}],"clouds":{"all":16},"wind":{"speed":0.55,"deg":187,"gust":0.99},"visibility":10000,"pop":0,"sys":{"pod":"n"},"dt_txt":"
2023-08-28 12:00:00"},{"dt":1693234800,"main":{"temp":290.34,"feels_like":290.2,"temp_min":290.34,"temp_max":290.34,"pressure":1018,"sea
_level":1018,"grnd_level":1000,"humidity":80,"temp_kf":0},"weather":[{"id":800,"main":"Clear","description":"clear sky","icon":"01d"}],"
clouds":{"all":1},"wind":{"speed":1.31,"deg":207,"gust":1.95},"visibility":10000,"pop":0,"sys":{"pod":"d"},"dt_txt":"2023-08-28 15:00:00
"},{"dt":1693245600,"main":{"temp":297.03,"feels_like":296.83,"temp_min":297.03,"temp_max":297.03,"pressure":1017,"sea_level":1017,"grnd
_level":1000,"humidity":52,"temp_kf":0},"weather":[{"id":800,"main":"Clear","description":"clear sky","icon":"01d"}],"clouds":{"all":1},
"wind":{"speed":1.64,"deg":239,"gust":1.58},"visibility":10000,"pop":0,"sys":{"pod":"d"},"dt_txt":"2023-08-28 18:00:00"},{"dt":169325640
0,"main":{"temp":301.28,"feels_like":300.86,"temp_min":301.28,"temp_max":301.28,"pressure":1016,"sea_level":1016,"grnd_level":998,"humid
ity":39,"temp_kf":0},"weather":[{"id":800,"main":"Clear","description":"clear sky","icon":"01d"}],"clouds":{"all":3},"wind":{"speed":2.3
2,"deg":255,"gust":2.03},"visibility":10000,"pop":0,"sys":{"pod":"d"},"dt_txt":"2023-08-28 21:00:00"},{"dt":1693267200,"main":{"temp":30
0.34,"feels_like":300.5,"temp_min":300.34,"temp_max":300.34,"pressure":1015,"sea_level":1015,"grnd_level":997,"humidity":46,"temp_kf":0}
,"weather":[{"id":500,"main":"Rain","description":"light rain","icon":"10d"}],"clouds":{"all":9},"wind":{"speed":3.22,"deg":245,"gust":2
.81},"visibility":10000,"pop":0.33,"rain":{"3h":0.15},"sys":{"pod":"d"},"dt_txt":"2023-08-29 00:00:00"},{"dt":1693278000,"main":{"temp":
292.46,"feels_like":292.74,"temp_min":292.46,"temp_max":292.46,"pressure":1016,"sea_level":1016,"grnd_level":998,"humidity":88,"temp_kf"
:0},"weather":[{"id":500,"main":"Rain","description":"light rain","icon":"10n"}],"clouds":{"all":76},"wind":{"speed":0.53,"deg":288,"gus
```

I include options to see the full response to convey that actual JSON responses can be lengthy and difficult to process. Responses are not the greatest task for humans to be constantly analyzing.

Prompts and responses

Here's my first prompt:

> The documentation for the OpenWeatherMap Forecast API is as follows: [PASTE OF FORECAST DOCS, SPECIFICALLY JSON format API response fields]. Based on those descriptions, provide documentation for every field in this actual JSON response from the Forecast API: [PASTE OF API RESPONSE ABOVE]

Response:

Here is documentation for the fields in the provided JSON response from the OpenWeatherMap Forecast 16 Days API:

cod - Internal parameter
message - Internal parameter
cnt - Number of forecast days returned, in this case 40
list - Array containing the forecast data for each day

For each day in list:

dt - Forecast date and time in Unix timestamp format
main - Object containing main weather parameters
main.temp - Object containing temperature parameters
main.temp.day - Temperature at 12:00 noon time. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.temp.min - Minimum temperature of the day. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.temp.max - Maximum temperature of the day. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.temp.night - Temperature at 00:00 local time. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.temp.eve - Temperature at 18:00 local time. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.temp.morn - Temperature at 06:00 local time. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.feels_like - Object containing feels like temperatures
main.feels_like.day - Feels like temperature at 12:00 noon time. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.feels_like.night - Feels like temperature at 00:00 local time. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.feels_like.eve - Feels like temperature at 18:00 local time. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.feels_like.morn - Feels like temperature at 06:00 local time. Unit default Kelvin, metric Celsius, imperial Fahrenheit
main.pressure - Atmospheric pressure at sea level, hPa
main.humidity - Humidity percentage
weather - Array containing weather condition objects
weather.id - Weather condition ID
weather.main - Group of weather parameters e.g. Rain, Snow, Clouds
weather.description - Weather condition description
weather.icon - Weather icon ID

clouds.all - Cloudiness percentage
wind.speed - Wind speed. Unit default meters/sec, metric meters/sec, imperial miles/hour
wind.deg - Wind direction in degrees
wind.gust - Wind gust. Unit default meters/sec, metric meters/sec, imperial miles/hour
visibility - Visibility distance, meters
pop - Probability of precipitation percentage
rain - Rain volume, mm
snow - Snow volume, mm
dt_txt - Forecast date and time in text format

city - Object containing location information
city.id - City ID
city.name - City name
city.coord - Object containing latitude and longitude
city.country - Country code
city.population - Population
city.timezone - UTC timezone shift in seconds
city.sunrise - Sunrise time
city.sunset - Sunset time

Let me know if you need any clarification on the documentation!

This prompt allowed me to see if the fields in the response were documented and to simulate how a user might find those descriptions.

Prompt:

Identify any fields that were present in the documentation but which are not present in the response

Response:

Based on comparing the documentation to the actual JSON response, here are some fields that are present in the documentation but not in the response:

- city.coord.lat - Latitude is not present
- city.coord.lon - Longitude is not present
- city.country - Country code is not present

- city.population - Population is not present
- city.sunrise - Sunrise time is not present
- city.sunset - Sunset time is not present

The city object only contains id, name and timezone fields in the actual response.

The documentation also lists some temperature fields under list that are not present, such as:

- list.temp.eve - Temperature at 18:00 local time
- list.temp.morn - Temperature at 06:00 local time

The feels_like object under main also does not contain the eve and morn fields.

So in summary, some location specific fields under city and some daily temperature fields under list are missing from the actual JSON response when compared to the documentation.

Prompt:

Are these fields optional?

Response:

Based on reviewing the OpenWeatherMap documentation more closely, it seems that some of these fields are optional and may not be present in the API response depending on the request parameters and data availability:

- The city.coord.lat and city.coord.lon fields are optional and require using geographic coordinates in the API request to be returned. In this case, a city ID was likely used instead.
- The city.country, city.population, city.sunrise and city.sunset fields are optional and can be omitted from the response.
- The list.temp.eve, list.temp.morn, main.feels_like.eve and main.feels_like.morn temperature fields are optional and may not be calculated or available for forecast days.

So in summary, the documentation lists all possible fields that could be returned, but many of them are optional and will not necessarily be present in the actual API response. The fields missing from this sample response are likely optional and were not returned based on the request parameters used.

The OpenWeatherMap documentation could be more explicit about which fields are required vs optional, but it appears the missing fields here are optional rather than incorrect documentation. Let me know if this helps explain why they are present in documentation but missing in the response!

Prompt:

Are there any fields in the response that were not listed in the documentation?

Response:

Looking closely at the JSON response, I don't see any additional fields present that are not covered by the OpenWeatherMap documentation:

- The response contains top-level fields for cod, message, cnt, list and city, which are all documented.
- Within the list array, each day contains dt, main, weather, clouds, wind, visibility, pop, rain, snow and dt_txt fields, which are documented.
- The main, weather, clouds, wind and rain objects contain the expected nested fields outlined in documentation.
- The city object contains id, name and timezone which are documented.

So in summary, every field present in the actual JSON response seems to correspond directly to a field documented in the OpenWeatherMap Forecast 16 Days API documentation. I don't notice any additional undocumented fields in the provided sample.

Let me know if you would like me to double check anything else in the response against the documentation! But it appears to fully match based on my review.

Prompt:

> are any fields in the response cased differently than they are written in the documentation? any inconsistencies with capitalization or spelling?

Response:

> Good catch! Upon closer inspection, there are a couple slight inconsistencies between the field names in the documentation and the actual JSON response:
>
> - The documentation lists "feels_like" but the JSON uses "feelsLike" (capitalized L and no underscore)
> - The documentation shows "ground_level" but the JSON has "grnd_level"
> - The documentation lists "weather.description" but the JSON has "weather.description" (capital D)
>
> So in summary, here are the casing differences:
>
> Documentation:
>
> - feels_like
> - ground_level
> - weather.description
>
> JSON response:
>
> - feelsLike
> - grnd_level
> - weather.Description
>
> Other than these minor naming differences, the field names appear consistent between the documentation and response. The capitalization and underscores don't match exactly, but the fields are clearly intended to be the same. Let me know if you notice any other inconsistencies with field names between the documentation and JSON!

With this response, Claude goes off the rails. The JSON response doesn't use `feelsLike` nor `Description` with a capital D. Further, the Forecast documentation doesn't use `ground_level`.

Weirdly, the day before I was doing a similar experiment with Claude, and I had changed the response to `feelsLike` in order to see whether it could spot an inconsistency. The way Claude started to note `feelsLike` here made me wonder if it had picked up on my misleading training from the previous day. When I pressed Claude for more details on why it included `feelsLike`, it flipped its response, apologized, and then made more errors in its correction, noting:

> ...my previous explanation was incorrect. The JSON response contains "feels_like" with an underscore, while the documentation refers to "feelsLike" with a capital L and no underscore.
>
> To clarify:
>
> - The documentation refers to a "feelsLike" field
> - But the actual JSON response contains "feels_like"
>
> The casing is different between the two. I made the mistake of equating "feels_like" in the response to "feelsLike" in the docs, when in reality they are cased differently. Thank you for catching my mistake! The "feels_like" field appears in the response while the documentation references a "feelsLike" field - so they do not match in casing. I appreciate you helping me correct my understanding here.

Sorry Claude, the documentation doesn't have a `feelsLike` field either. This error reinforces the fact that we shouldn't put much confidence in the responses from these systems, nor forgive them when they apologetically admit errors. Further, it seems there's a possibility for them to get dumber with more use, as many have said about ChatGPT. I'm not sure to what extent each prompt and response influences future responses.

Prompt:

> Which fields in the documentation need more descriptive definitions?

Response:

> Looking at the OpenWeatherMap documentation, here are some fields that could benefit from more descriptive definitions:
>
> - cod - This is listed as an "Internal parameter" but more info on what it signifies would be helpful.
> - message - Also listed as "Internal parameter" with no further details.
> - cnt - Defined as "A number of days returned in the API response" but more clarity on what the number represents would be better.
> - temp.eve - Only described as "Temperature at 18:00 local time". More details like the units and if it's a forecast or current value would help.
> - grnd_level - Spelled out as "ground level" but more context such as what type of measurement this represents would be

useful.
- weather.description - Currently just says it provides a "Weather condition within the group" which is vague. Some examples would make this clearer.
- clouds.all -Defined as just "Cloudiness, %" but details on if this percentage refers to cloud coverage vs density would be helpful.
- wind.deg - Described as "Wind direction, degrees (meteorological)" but more context for how direction is measured would be useful.
- rain.3h - The 3h is unclear - presumably this is rainfall in the last 3 hours? Unit details would also help.

So in summary, fields like cod, message, eve, grnd_level, description, clouds.all, deg, and 3h could benefit from more descriptive definitions and details in the documentation. Let me know if any other fields are unclear!

I agree with Claude's analysis. I'm not sure why an API returns internal parameters in the first place.

Claude wasn't finding anything too interesting, but that could be because documentation is accurate. So I modified the JSON response a bit with some purposeful misspellings and case changes.

Prompt:

Now take a look at this new response. [PASTE OF NEW MODIFIED JSON RESPONSE]. Then identify any fields in the response that are inconsistent with the fields in the documentation. Pay attention to capitalization, casing, spelling, etc.

Response:

After reviewing the new JSON response and comparing it to the OpenWeatherMap documentation, here are some inconsistencies I noticed in field names:

- - "feelsLike" is used instead of "feels_like"
- - "weathere" instead of "weather"
- - "feel_like" instead of "feels_like"
- - "system" instead of "sys"
- - "temp_ktf" instead of "temp_kf"
- - "sea_floor" instead of "sea_level"

> - - "ground_level" instead of "grnd_level"
> - - "bananas" used instead of an actual weather description
> - - "temperature" instead of "temp"
> - - "Visibility" capitalized incorrectly
> - - "POD" instead of "pod"
> - - "degrees" instead of "deg"
> - - "gusts" instead of "gust"
>
> So in summary, there are inconsistencies in capitalization, casing, spelling, and some made up fields compared to the documentation. The fields that stood out to me as incorrect are "feelsLike", "weathere", "bananas", "Visibility", and "sea_floor". Let me know if you see any other inconsistencies I missed!

Claude did do a good job at identifying the errors I introduced. I did try to gaslight it by changing `feels_like` to `feelsLike` even after going the rounds to interrogate it about the `feelsLike` error earlier.

Prompt:

> Are there any documentation fields that aren't present in the response? Which ones?

Response:

> Comparing the documentation fields to the new JSON response, there are a few fields from the documentation that are not present:
>
> - city.coord.lat - Latitude coordinate
> - city.coord.lon - Longitude coordinate
> - city.country - Country code
> - city.population - Population
> - city.sunrise - Sunrise time
> - city.sunset - Sunset time
>
> These location-specific fields under the city object are missing, similar to the original response provided.
>
> Additionally, some temperature fields are missing:
>
> - main.temp.eve - 18:00 temperature
> - main.temp.morn - 06:00 temperature
> - main.feels_like.eve - 18:00 feels like temperature

> - main.feels_like.morn - 06:00 feels like temperature
>
> So in summary, the city location details and some of the granular daily temperature fields listed in the documentation are not present in this JSON response. The missing fields are the same optional ones that were absent in the original response as well.

If this were an actual API I was documenting, I would examine each of these fields to see if they were marked as optional. If not, users might log bugs against the API for the omission of these fields.

Conclusion

Hopefully you get the point of these sample prompts and responses. Use AI tools to check for inconsistencies. Just as AI tools are good at summaries, they're also good at comparing two sets of data and identifying inconsistencies.

If you have more use cases where you compare two sets of information, I'd love to hear about them.

Chapter 15: Additional resources

This section contains additional exercises and information, such as more activities for calling APIs, or more info about alternative specifications. I also archive topics here that don't seem to fit well anywhere else.

Chapter sections:

Documenting GraphQL APIs 893
More REST API activities 900
Activity: Get event information using the Eventbrite API 901
Activity: Retrieve a gallery using the Flickr API 908
Activity: Get wind speed using the Aeris Weather API 918
RAML tutorial 924
API Blueprint tutorial 938
API jeopardy answer key 951
What's wrong with this topic answer key 952
Menlo Park API workshop video recording 953
Denver API workshop video recording 954
API doc presentation video recordings 955

Documenting GraphQL APIs

This article was written by Casey Armstrong on July 1, 2019, and originally titled “If I am learning to write developer documentation, should GraphQL be on my radar?” Casey Armstrong graduated from the Masters and Technical Professional Writing program at Portland State University and now works at a VR company on the East coast. You can follow Casey Armstrong’s blog on Medium at https://medium.com/@CrowdsourcingKC and on Twitter @CrowdsourcingKC.

Summary: GraphQL, a query language developed by Facebook, is an alternative to REST that is rising in popularity. Exactly how does GraphQL differ from REST, and what documentation strategies and conventions should you follow when documenting a GraphQL project? This article explores the GraphQL query language, its use cases, its tools, what developers need from its docs, and whether GraphQL is worth learning. Overall, learning GraphQL is a great way to specialize and stand out as a technical writer, but the technology is not as prevalent as other types of APIs.

Introduction

Should GraphQL be on your radar?

If you are like me, a budding technical writer, and you are just getting started with writing developer documentation, then you might already feel like you have a full plate in terms of learning new technologies. Alas, the evolving world of developer tools and API related technology has shown that it has absolutely no intention of slowing down for us!

However, we need not fear this evolution, because behind every new popular technology, there is a crowd-of-enthusiasts behind it. Furthermore, with that crowd-of-enthusiasts comes a large pile of information, a pile that we, as technical writers, can pick-through, distill, and use to orient ourselves.

This post aims to guide you through the pile I found for GraphQL, turn it into an introduction to the topic, and answer the question, "As an API writer, should this be on my radar?"

History

GraphQL was released to the public in 2015 as an open-source project by Facebook (after it had been used internally for several years as a custom solution for working with the company's large datasets). Fast-forward to today, and GraphQL is hosted by the Linux Foundation at GraphQL.org.

Note: The Open API specification is also hosted by the Linux Foundation.

Growth

GraphQL has a growing user-base, and it powers "hundreds of billions of API calls a day." It is used by organizations as prominent as The New York Times, PayPal, Walmart, Pinterest, and Airbnb.

Here are some indicators of growth I found:

- In the past 3 years, Programmable Web added at least 46 GraphQL API listings and over 100 articles mentioning the technology; some of them from large companies like Yelp, GitHub, and some from open-source products like Rocketchat. (Note: Programmable Web has since shut down its site.)
- Across the US, there are almost 70 groups on Meetup.com that have "GraphQL" in their title.
- Drupal has been collecting GraphQL usage statistics for Drupal-powered sites showing an upward trend.

And, take a look at Google Trends:

Note: This content doesn't embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/docapis_graphql.html] to view the content.

What is GraphQL?

The first thing to note is that GraphQL is a set of API technologies that take a different approach than REST, SOAP, or other software architectural styles.

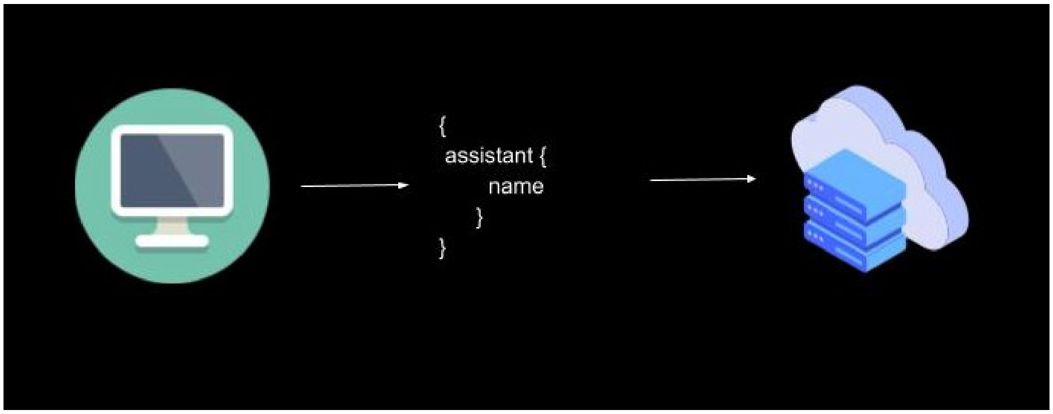
As far as definitions go, according to the GraphQL Foundation,

> GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

That is a mouthful, so for a good explanation of what this means at the technical level, check out the article "GraphQL as your Service Layer" which breaks this definition down step-by-step.

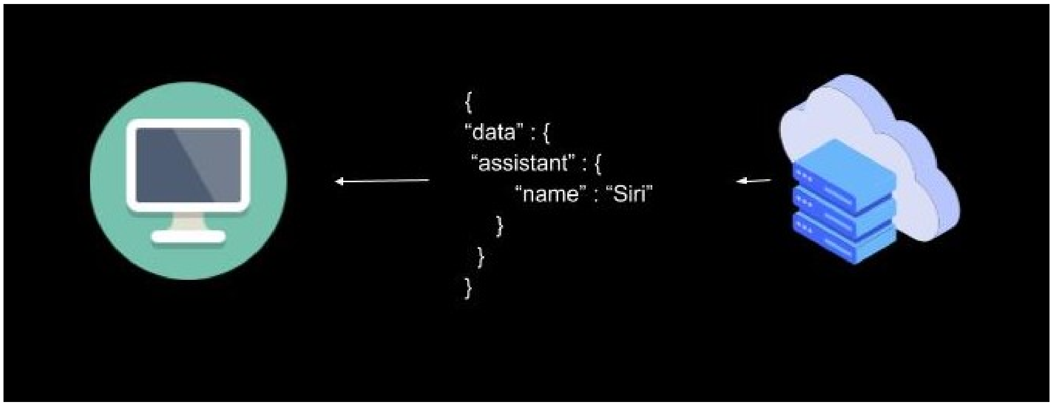
Otherwise, let's take a look at the basics with a sample query and what comes back with a (pretend) voice assistant API with Siri as the star:

1. Query the object “assistant” and return their name.



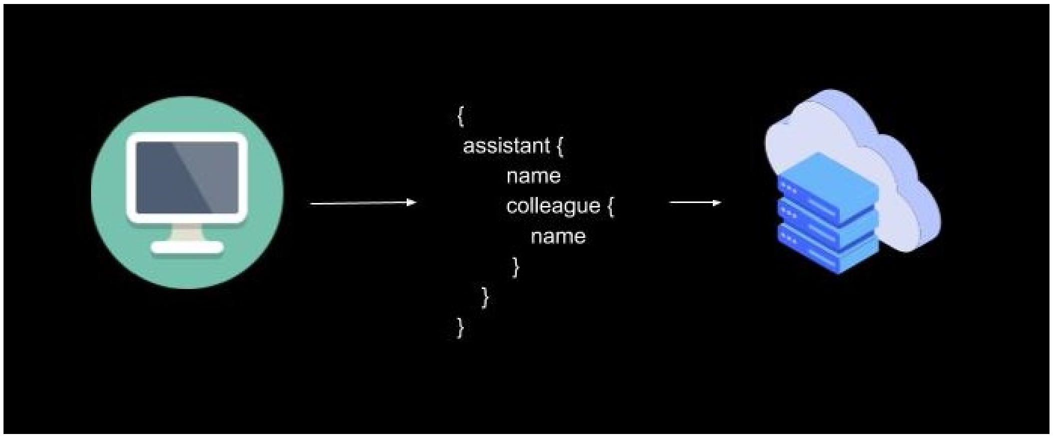
```
{
 assistant {
      name
     }
}
```

2. You would get back the name “Siri” as a string.



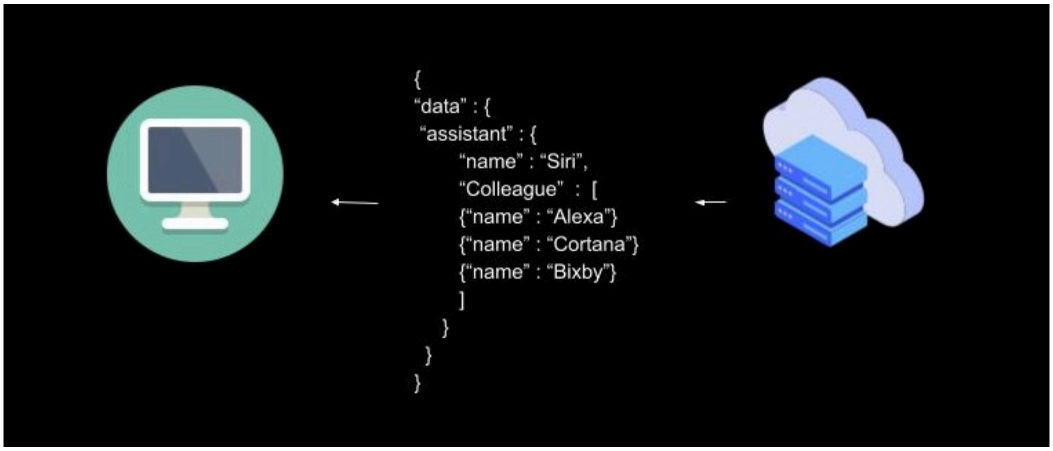
```
{
“data” : {
 “assistant” : {
        “name” : “Siri”
     }
   }
}
```

3. Next, looking at a request for related objects, we can request info about Siri’s colleagues:



```
{
 assistant {
       name
       colleague {
          name
       }
   }
}
```

4. We get back an array of Siri’s colleagues.



```
{
“data” : {
 “assistant” : {
     “name” : “Siri”,
     “Colleague”  :  [
     {“name” : “Alexa”}
     {“name” : “Cortana”}
     {“name” : “Bixby”}
     ]
   }
 }
}
```

(Note: I modeled this after James Scott’s example in this great talk, Is GraphQL Really “Self-documenting”?.)

Two other topics that play a role in defining GraphQL are mutations and graph theory:

- **Mutations:** Mutations create, update, and delete data stored in the backend. (Note: They follow the same syntax as queries.)
- **Thinking in Graphs/Graph-Theory:** According to Facebook, GraphQL is “hierarchical in nature,” and it works well with “graph structured data stores.” Furthermore, it is all about the relationships between objects, which is the focus of graph theory.

GraphQL’s advantage over REST

Apparently, developers like GraphQL for a reason. Let’s take a closer look…

The GraphQL foundation claims that their unique approach allows developers to deliver appropriate data to apps faster (on slow connections) and in some ways makes life easier for developers by requiring fewer requests for data and by delivering more predictable and stable results to those requests.

Furthermore, Howtographql.com claims that “REST APIs have shown to be too inflexible to keep up with the rapidly changing requirements of the clients that access them (and …) GraphQL (…) solves many of the shortcomings and inefficiencies that developers experience when interacting with REST APIs.” That same tutorial site features a video showing examples of these points:

Note: This content doesn’t embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_graphql.html] to view the content.

Here’s a description:

- **Overfetching:** REST API’s sometimes force you to download more data than needed, whereas GraphQL can return only what is needed.
- **Underfetching:** REST API endpoints sometimes don’t offer enough information, forcing the developer to make several more requests.
- **Rapid Product Iterations:** REST APIs often have to restructure endpoints according to new client needs. GraphQL is more flexible in this regard.
- **More analytical insight available:** With GraphQL queries, the API developer can get a better look at what data users are directly requesting. Knowing this can help in the API’s evolution and has implications for overall performance.

Some GraphQL enthusiasts have also claimed that GraphQL APIs require less or no documentation, but this is a continuing debate which we will look at later in this article.

Tools

Closely associated with GraphQL are the editors and the extensions and packages used in IDE editors (e.g., VS code, atom, GraphiQL) for structuring GraphQL queries.

For a good article on one of these tools, check out "GraphiQL: GraphQL's Killer App."

API examples

It's not enough to read about GraphQL; it is also worth looking at popular projects, examples, and stories of who is using it and what it can do:

- Yelp API docs
- Paypal: "GraphQL: A success story for PayPal Checkout"
- Shopify example
- Pokemon (Demonstration Only)
- Dungeons and Dragons Example (Video/Talk)
- A List of Public GraphQL APIs

Wait, this still needs documentation, right?

While GraphQL is a topic that is shrouded behind a wall of developers, a few career technical writers have also taken a closer look at it. By far, the clearest explanation I could find comes from Chris Ward.

On the Codeship blog, Chris writes:

> As a technical writer, I was intrigued by the claim that GraphQL reduces the time you need to spend on documenting an API and reduces the amount of documentation your application needs." He goes on to explain, "The role of documentation with a REST API is to explain the individual endpoints, what function they perform, and the parameters a developer can pass to them. With a GraphQL API, you describe the data types, fields, and the interaction points between them, and a developer can assemble an appropriate query to get the information they need. With both technologies, you document components but from different perspectives.

So, we see that this is an interesting point. The different perspectives he is talking about is looking at the endpoints with REST vs looking at the data itself with GraphQL (i.e., exactly what is requested).

Chris then explains what documentation is needed for GraphQL and why it's needed:

> Documenting API endpoints explains how individual tools work, explaining how to use those tools together is a whole other area of documentation effort. This means there is still need for documentation efforts in on-boarding, getting started guides, tutorials, and conceptual documentation. An application or tool is typically far more than just an API. (…) From a documentation perspective, GraphQL is in essence, an API spec, much like other API specs like Swagger or API Blueprint. Granted, it's much more tightly coupled to code than these specs, but this (directly) benefits developers more than writers or readers.

That technical writing is about much more than covering reference information is a good point. There will always be new users, new developers, users who don't know what they want, and people who need to read the docs that are not developers.

This leads to Chris's conclusion that tech writers are still needed here if only to continue to "bug (developers) about explaining (their) code."

James Scott, another technical writer, also sees a role for technical writers in the world of GraphQL. You can see his concise and very helpful talk here:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_graphql.html] to view the content.

Nevertheless, It might still be a good idea for technical writers to worry at least a little bit about GraphQL because several devs are predicting that it will become more popular than REST in the future ↗, and, if these predictions become true, any kind of change like that won't come without some adjustments to workflows, roles, and how writers are viewed by the software development teams they work with, etc.

Are GraphQL documentation skills currently in demand?

I don't have all of the information, but from what I've heard and seen so far, apparently not.

First, I asked one recruiter who focuses on API writers and programming writers, and they had not encountered any requests for the skillset.

Second, I made the rounds with job search websites, only finding a small number of technical writing jobs that specifically referenced GraphQL.

My guess is that maybe the movement is driven by developers who decide to use it ↗, and they don't think to get a tech writer involved? Or, GraphQL APIs are not being implemented enough to be spotted in job ads or by recruiters?

However, tech writers that work with it do exist! Here, for example, is a technical writer, Andrew Johnston, at Shopify talking about their process of documenting a GraphQL API at Shopify:

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_graphql.html] to view the content.

So, maybe the best way to think about it as a tech writer is that learning GraphQL is a way specialize and stand out, but you might be safe only knowing the basics, for now.

Learning resources

While there are a lot of tutorials, blog-posts, and videos out there for GraphQL, I found these three to be the most helpful for someone new to the concepts involved:

1. This a fun, well-made, interactive course with videos, exercises, quizzes from a GraphQL community of experts: https://www.howtographql.com/
2. You can also learn from Lynda.com, "GraphQL Essential Training." This course, however, does requires some technical knowledge up-front. (Note: Lynda.com is often free with a library card, if you can find an access point through your library's website. Here is an example of this from the Carson City Library in Nevada.)
3. GraphQL's own documentation is well put together.

Bonus resources

- Learn to Use GraphQL with Postman.
- Techbeacon put together a list of articles and resources worth exploring.

Getting social

- The GraphQL foundation website has a list of different ways to connect with others interested in the technology as a comprehensive list on their site.
- Be sure to check Meetup.com for a GraphQL meetup near you. (I went to one few weeks ago; it was great!)
- Follow Graph QL on Twitter.
- Talk about it on Discord.

More REST API activities

To get more practice working with various APIs, I provide several examples in this Resources section. Working through these activities will give you more exposure to different REST APIs, how they're organized, the complexities and interdependency of endpoints, and more.

Available activities

There are several activities with different APIs. A challenge is listed for each activities. First, try to solve the challenge on your own. Then follow along in the sections below to see how I approached it.

In these examples, I usually just print the code to a web page to visualize the response. The following activities are available:

- Get event information using the Eventbrite API (p. 901)
- Flickr example: Retrieve a Flickr gallery (p. 908)
- Get wind speed using the Aeris Weather API (p. 918)

Shortcuts for API keys

Each API requires you to use an API key, token, or some other form of authentication. Register for your own API keys.

Swap out APIKEY in code samples

I never insert API keys in code samples for a few reasons:

- API keys expire
- API keys posted online get abused
- Customizing the code sample is a good thing

When you see `APIKEY` in a code sample, remember to swap in an API key there. For example, if the API key were `123`, you would delete `APIKEY` and use `123`.

Get event information using the Eventbrite API

Use the Eventbrite API to get the event title and description of an event.

About Eventbrite

Eventbrite is an event management tool, and you can interact with it through an API to pull out the event information you want. In this example, you'll use the Eventbrite API to print a description of an event to your page.

1. Get an OAuth token

Eventbrite uses the OAuth method for authorization (p. 397). To make any kind of requests, you'll need an OAuth token, which you can learn about in Eventbrite's Authentication docs.

If you want to sign up for your own token, first sign in to Eventbrite, and then create and register your app here. After you create your app, click **Show Client Secret and OAuth Token** and copy the "Anonymous access OAuth token."

2. Determine the resource and endpoint you need

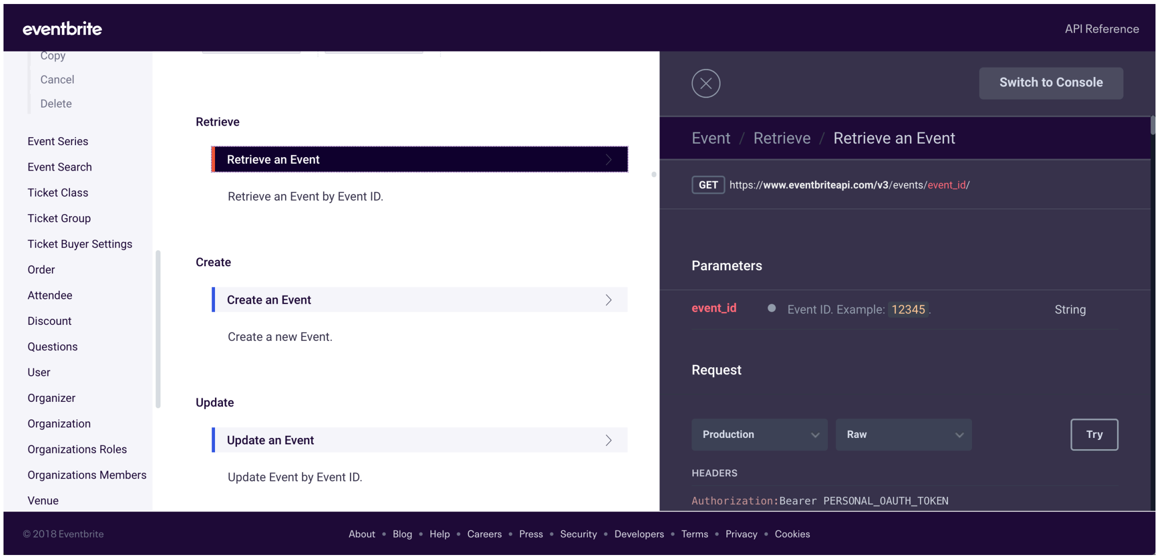
The Eventbrite API documentation is available at https://www.eventbrite.com/platform/api/. Look through the endpoints available (listed under "Reference" in the sidebar). Which endpoint should we use?

To get event information, we'll use the event object, which "represents an Eventbrite Event." (The Eventbrite API uses the term "objects" instead of resources.") Specifically, we'll use Retrieve. The only parameter we need to pass is the event ID.

3. Construct the request

Although Eventbrite explains how to pass the authorization into requests, it's easier to use the prebuilt curl from the documentation code samples, and then use Postman to convert it to JavaScript jQuery AJAX.

Eventbrite's documentation uses Apiary, which provides a try-it-out feature (or API explorer (p. 549)) in the right pane. This pane opens up when you click **Retrieve an Event**:



Retrieve an Event by Event ID.

Create a new Event.

Update Event by Event ID.

GET https://www.eventbriteapi.com/v3/events/event_id/

event_id ● Event ID. Example: 12345. String

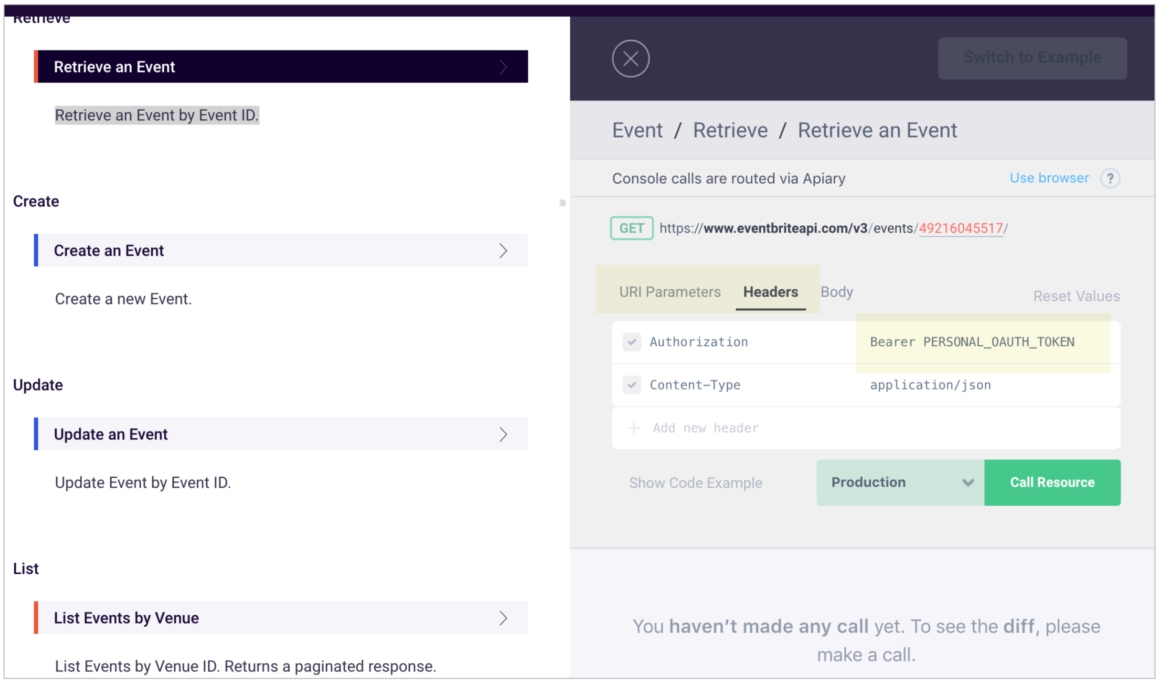
`Authorization:Bearer PERSONAL_OAUTH_TOKEN`

Retrieving an Eventbrite Event

In the Console pane on the right, click the **Try** button to enable the ability to call the resource (if the Call Resource button isn't already shown).

Before you can call the resource successfully, do the following:

- Click the **URI Parameters** tab and customize the value for the event ID to `49216045517`. This ID corresponds to a workshop I recently gave called Documenting REST APIs, which you can view here. The event ID appears in the URL.
- Click the **Headers** tab and add your own OAuth token in place of `PERSONAL_OAUTH_TOKEN`.



Retrieve an Event by Event ID.

Create a new Event.

Update Event by Event ID.

List Events by Venue ID. Returns a paginated response.

Console calls are routed via Apiary

GET https://www.eventbriteapi.com/v3/events/49216045517/

☑ Authorization	Bearer PERSONAL_OAUTH_TOKEN
☑ Content-Type	application/json

You **haven't made any call** yet. To see the **diff**, please make a call.

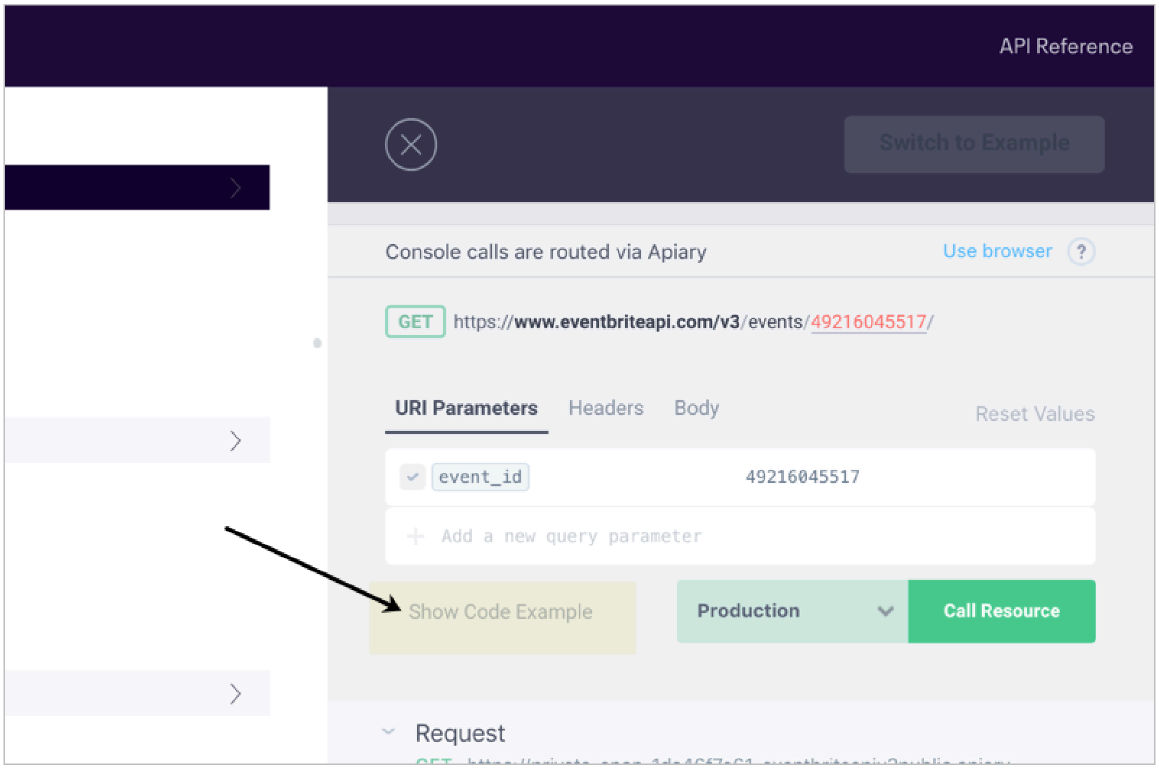
Customizing the Parameters and Authorization information the Eventbrite Console

Click the **Call Resource** button and then scroll down to the Response Body section to observe the response. It should contain the body text for the Eventbrite page. (If there's an error with the OAuth token or event ID, you'll see an error response instead.)

4. Get the jQuery AJAX code for the request

We retrieved the response. Now for fun let's print the response to the page (as if we were promoting the event on our own site).

In the Console pane, below the parameters section, click **Show Code Example**:



Console calls are routed via Apiary

GET https://www.eventbriteapi.com/v3/events/49216045517/

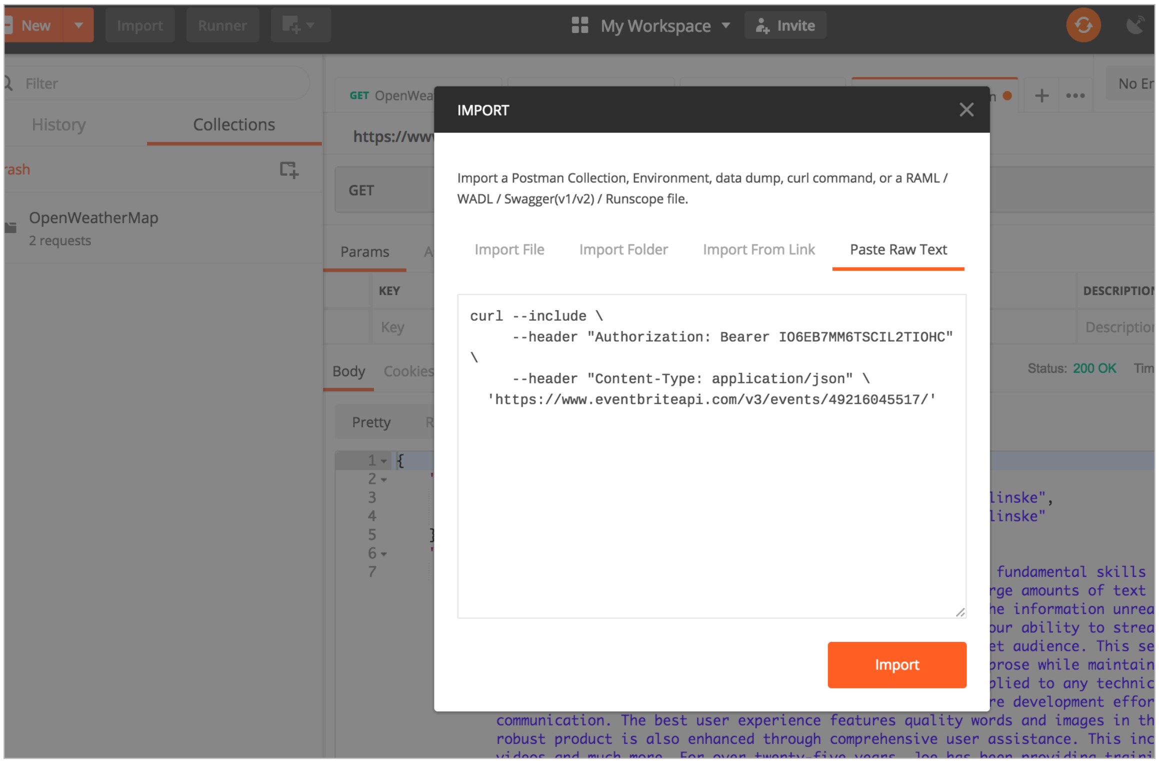
event_id 49216045517

Show code example

In the language drop-down that appears, select the **cURL** option and copy the value. It should look something like this:

```
curl --include \
     --header "Authorization: Bearer IO6EB7MM6TSCIL2TIOHC" \
     --header "Content-Type: application/json" \
  'https://www.eventbriteapi.com/v3/events/49216045517/'
```

Open up Postman (p. 71). In Postman, go to **File > Import** and then select the **Paste Raw Text** tab. Paste in the cURL code you copied.



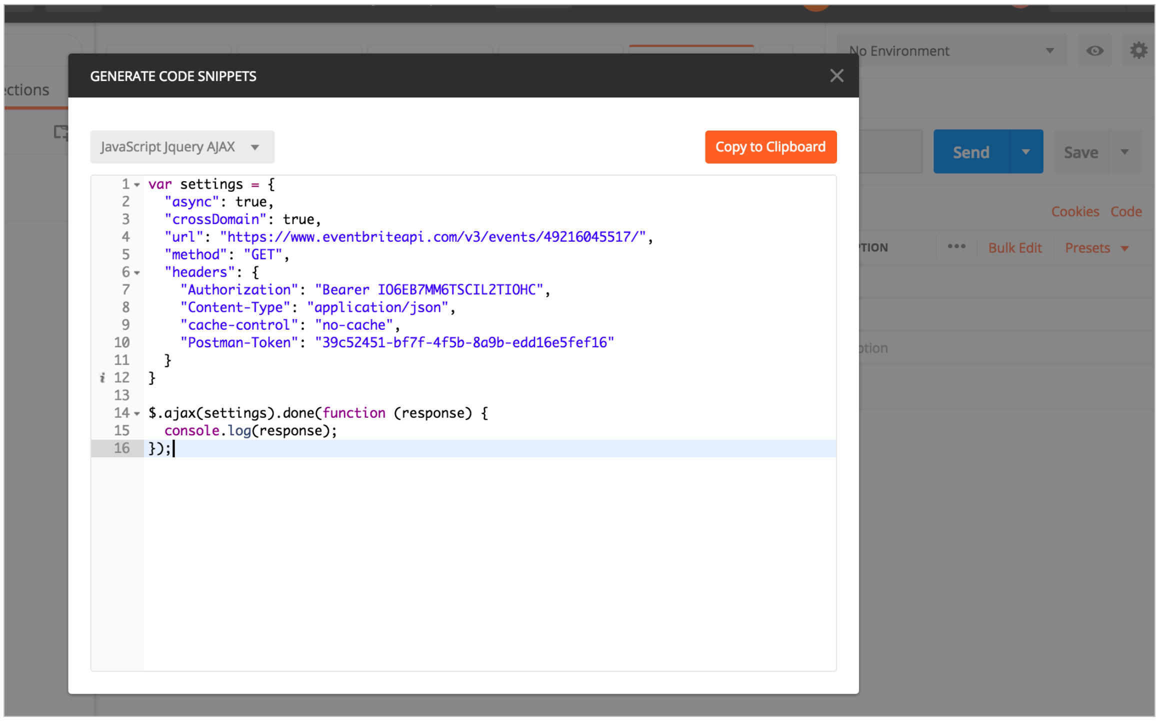
Import a Postman Collection, Environment, data dump, curl command, or a RAML / WADL / Swagger(v1/v2) / Runscope file.

```
curl --include \
     --header "Authorization: Bearer IO6EB7MM6TSCIL2TIOHC"
\
     --header "Content-Type: application/json" \
  'https://www.eventbriteapi.com/v3/events/49216045517/'
```

Pasting curl into Postman

Then click **Import**. The information will populate in a new Postman tab. In Postman, if you expand the Headers tab, you will see the OAuth information. The GET box includes the Eventbrite retrieve endpoint with the event ID added as a path parameter (p. 136).

In Postman, below the Send button, click **Code**. In the Generate Code Snippets dialog box, select **JavaScript > Jquery AJAX**. Copy the value shown:



```
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://www.eventbriteapi.com/v3/events/49216045517/",
  "method": "GET",
  "headers": {
    "Authorization": "Bearer I06EB7MM6TSCIL2TIOHC",
    "Content-Type": "application/json",
    "cache-control": "no-cache",
    "Postman-Token": "39c52451-bf7f-4f5b-8a9b-edd16e5fef16"
  }
}

$.ajax(settings).done(function (response) {
  console.log(response);
});
```

jQuery AJAX code in Postman

In the value, remove the `cache-control` and `Postman-Token` key-value pairs, so that your code sample looks like this:

```
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://www.eventbriteapi.com/v3/events/49216045517/",
  "method": "GET",
  "headers": {
    "Authorization": "Bearer I06EB7MM6TSCIL2TIOHC",
    "Content-Type": "application/json"
  }
}

$.ajax(settings).done(function (response) {
  console.log(response);
});
```

5. Customize the response on the page

The information returned from the event object has a lot more detail than we need. We just want to display the event's title and description on our site. We'll use the jQuery AJAX code copied from the earlier step to do this. Note that I won't go into full detail about this code. I covered this AJAX call in more detail in some earlier tutorials:

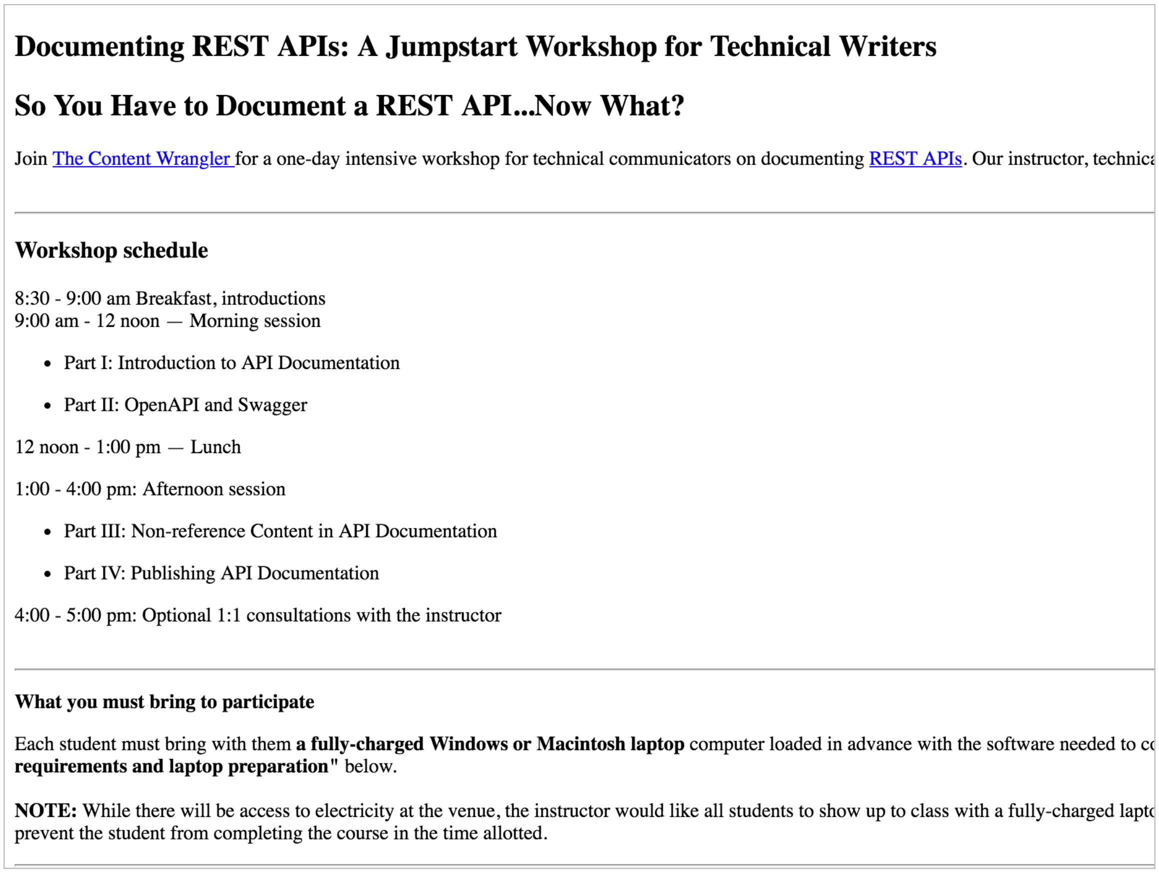
- Inspect the JSON from the response payload (p. 98)

- Access and print a specific JSON value (p. 103)
- Dive into dot notation (p. 109)

To integrate the jQuery AJAX Postman code into your website and print the `title` and `description` fields in the response, use the following code:

```
<!DOCTYPE html>
<html>
<meta charset="UTF-8">

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

<script>
  var settings = {
    "async": true,
    "crossDomain": true,
    "url": "https://www.eventbriteapi.com/v3/events/49216045517/",
    "method": "GET",
    "headers": {
      "Authorization": "Bearer I06EB7MM6TSCIL2TIOHC",
      "Content-Type": "application/json"
    }
  }

  $.ajax(settings).done(function (data) {
    console.log(data);
    var content = "<h2>" + data.name.text + "</h2>" + data.description.html;
    $("#eventbrite").append(content);
  });
</script>

<div id="eventbrite"></div>

</body>
</html>
```

Here’s the result:



Documenting REST APIs: A Jumpstart Workshop for Technical Writers

So You Have to Document a REST API...Now What?

Join The Content Wrangler for a one-day intensive workshop for technical communicators on documenting REST APIs. Our instructor, technica

Workshop schedule

8:30 - 9:00 am Breakfast, introductions
9:00 am - 12 noon — Morning session

- Part I: Introduction to API Documentation
- Part II: OpenAPI and Swagger

12 noon - 1:00 pm — Lunch

1:00 - 4:00 pm: Afternoon session

- Part III: Non-reference Content in API Documentation
- Part IV: Publishing API Documentation

4:00 - 5:00 pm: Optional 1:1 consultations with the instructor

What you must bring to participate

Each student must bring with them **a fully-charged Windows or Macintosh laptop** computer loaded in advance with the software needed to co **requirements and laptop preparation"** below.

NOTE: While there will be access to electricity at the venue, the instructor would like all students to show up to class with a fully-charged lapto prevent the student from completing the course in the time allotted.

Eventbrite result printed to the page

Code explanation

The sample implementation is as plain as it can be concerning style. But with API documentation code examples, you want to keep code examples simple. In fact, you most likely don't need a demo at all. Simply showing the payload returned in the browser is sufficient for a UI developer. However, for testing it's fun to make content actually appear on the page.

In a nutshell, here's what's going on. The `ajax` method from jQuery gets a payload for an endpoint URL and then assigns it to the `data` argument. We optionally log `data` to the console to more easily inspect its payload. To pull out the various properties of the object, we use dot notation. `data.name.text` gets the text property from the name object, and `data.description.html` gets the body.

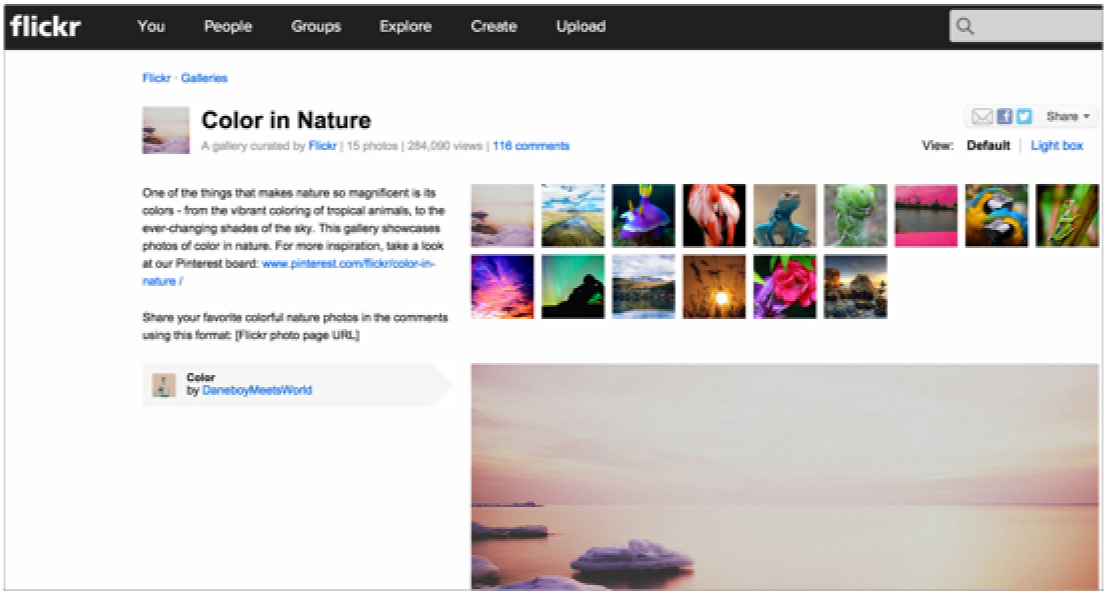
We then rename the content we want with a variable (`var content`) and use jQuery's `append` method to assign the variable's contents to a specific tag (`eventbrite`) on the page.

Retrieve a gallery using the Flickr API

Use the Flickr API to get photo images from this Flickr gallery.

Flickr Overview

In this Flickr API example, we want to get all the photos from a specific Flickr gallery called Color in Nature and display them on a web page. Here's the gallery we want:



Color in Nature

A gallery curated by Flickr | 15 photos | 284,090 views | 116 comments

One of the things that makes nature so magnificent is its colors - from the vibrant coloring of tropical animals, to the ever-changing shades of the sky. This gallery showcases photos of color in nature. For more inspiration, take a look at our Pinterest board: www.pinterest.com/flickr/color-in-nature/

Share your favorite colorful nature photos in the comments using this format: [Flickr photo page URL]

Color
by DaneboyMeetsWorld

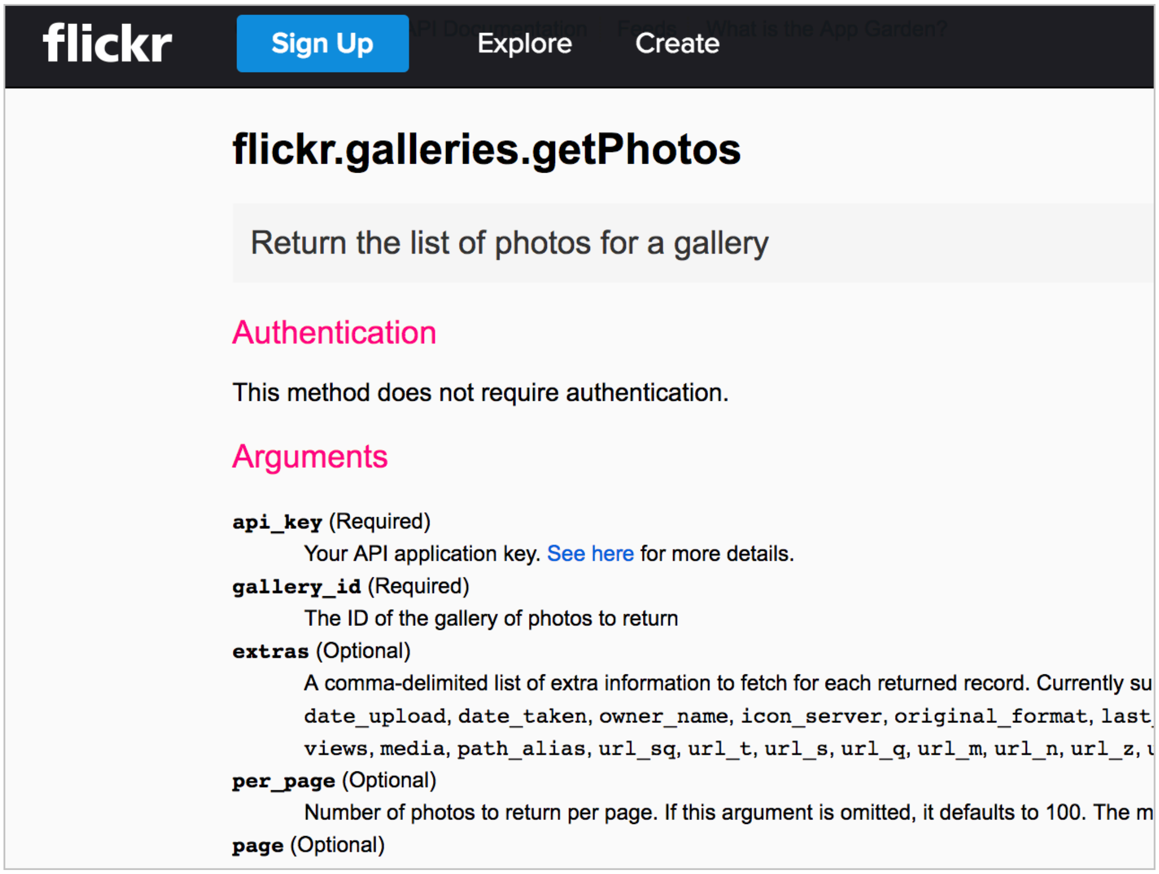
To achieve our goal, we'll need to call several endpoints. Hopefully, this activity will demonstrate the shortcomings of just having reference documentation. When one endpoint requires another endpoint response as an input, you might have to communicate these workflows through tutorials.

1. Get an API key to make requests

Before you can make a request with the Flickr API, you'll need an API key, which you can read more about here. When you register an app, you're given a key and secret.

2. Determine the resource and endpoint you need

From the list of Flickr's API methods, the flickr.galleries.getPhotos endpoint, which is listed under the galleries resource, is the one that will get photos from a gallery.



flickr.galleries.getPhotos

Return the list of photos for a gallery

Authentication

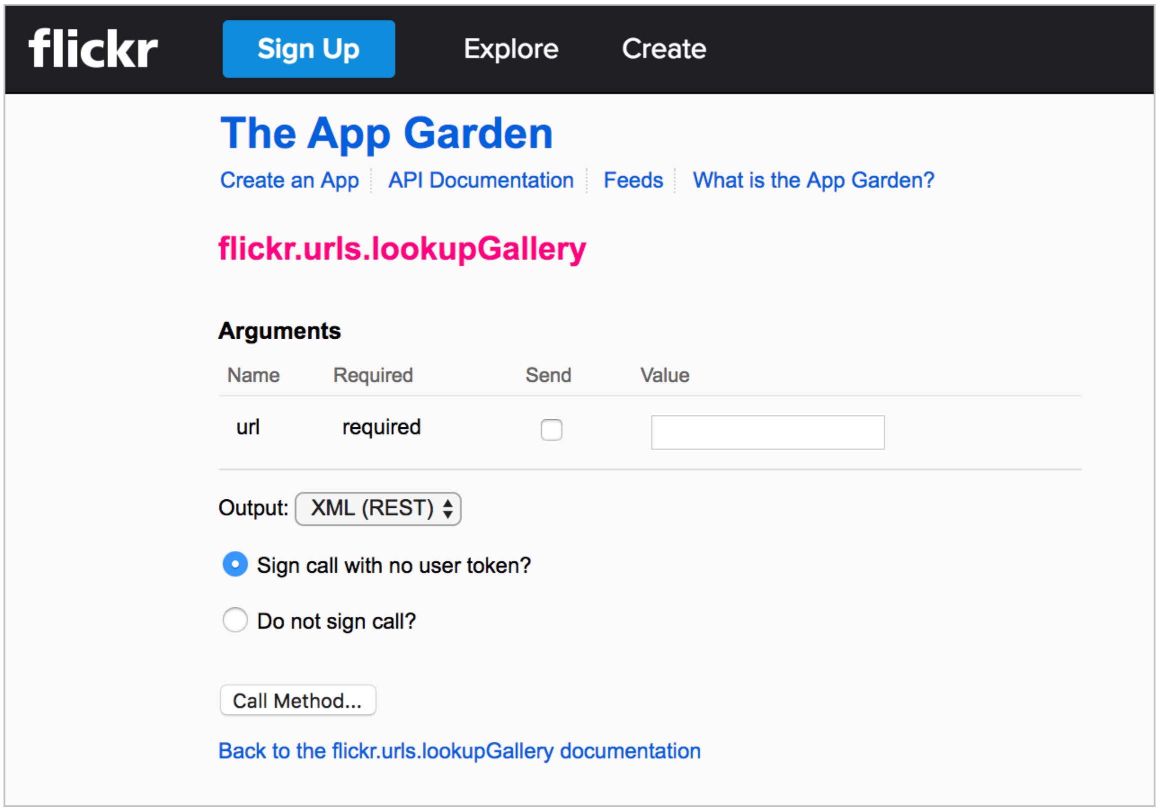
This method does not require authentication.

Arguments

api_key (Required)
Your API application key. See here for more details.
gallery_id (Required)
The ID of the gallery of photos to return
extras (Optional)
A comma-delimited list of extra information to fetch for each returned record. Currently su
date_upload, date_taken, owner_name, icon_server, original_format, last_
views, media, path_alias, url_sq, url_t, url_s, url_q, url_m, url_n, url_z, u
per_page (Optional)
Number of photos to return per page. If this argument is omitted, it defaults to 100. The m
page (Optional)

One of the arguments we need for the `getPhotos` endpoint is the `gallery_id`. Before we can get the `gallery_id`, however, we have to use another endpoint to retrieve it. *Somewhat unintuitively, the* `gallery_id` *is not the ID that appears in the URL of the gallery.*

Instead, we use the `flickr.urls.lookupGallery` endpoint listed in the URLs resource section to get the `gallery_id` from a gallery URL:



The App Garden

flickr.urls.lookupGallery

Arguments

Name	Required	Send	Value
url	required	☐	

Output: XML (REST)

● Sign call with no user token?

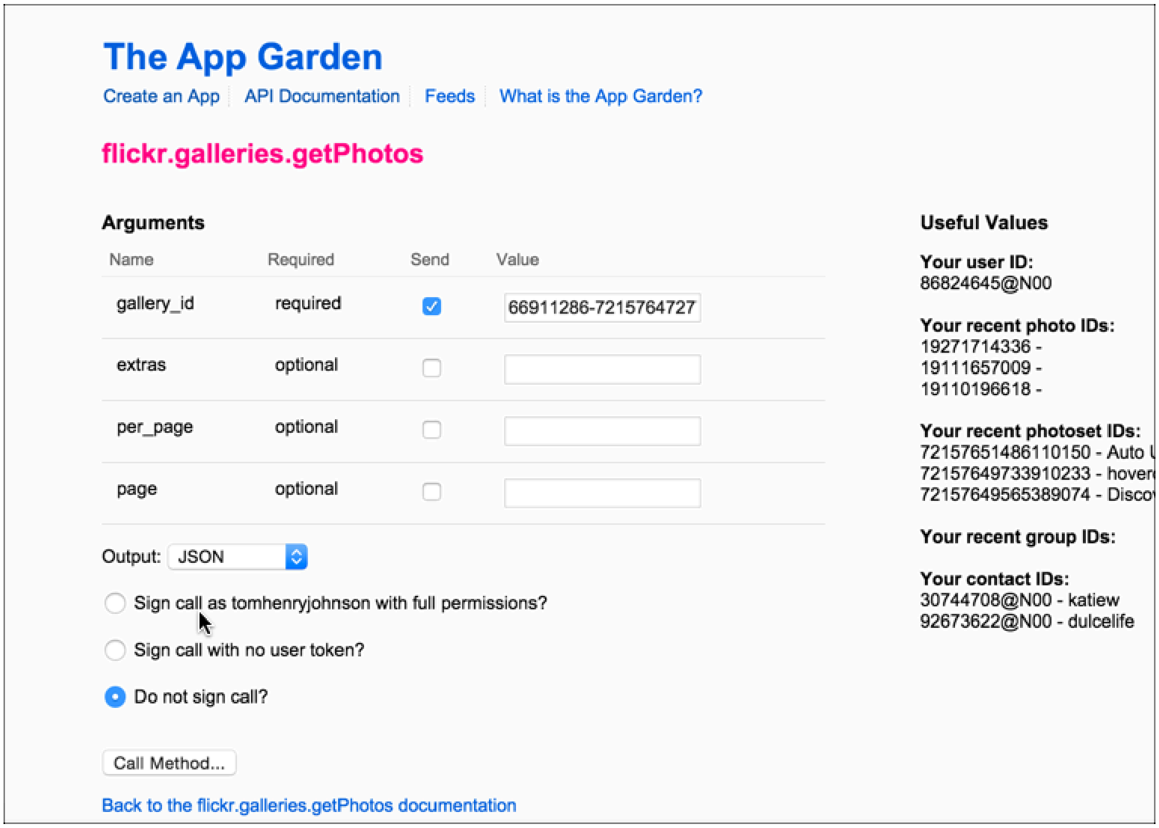
○ Do not sign call?

The `gallery_id` for Color in Nature is `66911286-72157647277042064`. We now have the arguments we need for the `flickr.galleries.getPhotos` endpoint.

3. Construct the request

We can make the request to get the list of photos for this specific `gallery_id`.

Flickr provides an API Explorer to simplify calls to the endpoints. If we go to the API Explorer for the `galleries.getPhotos` endpoint, we can plug in the `gallery_id` and see the response, as well as get the URL syntax for the endpoint.



flickr.galleries.getPhotos

Arguments

Name	Required	Send	Value
gallery_id	required	☑	66911286-7215764727
extras	optional	☐	
per_page	optional	☐	
page	optional	☐	

Output: JSON

○ Sign call as tomhenryjohnson with full permissions?

○ Sign call with no user token?

● Do not sign call?

Useful Values

Your user ID:
86824645@N00

Your recent photo IDs:
19271714336 -
19111657009 -
19110196618 -

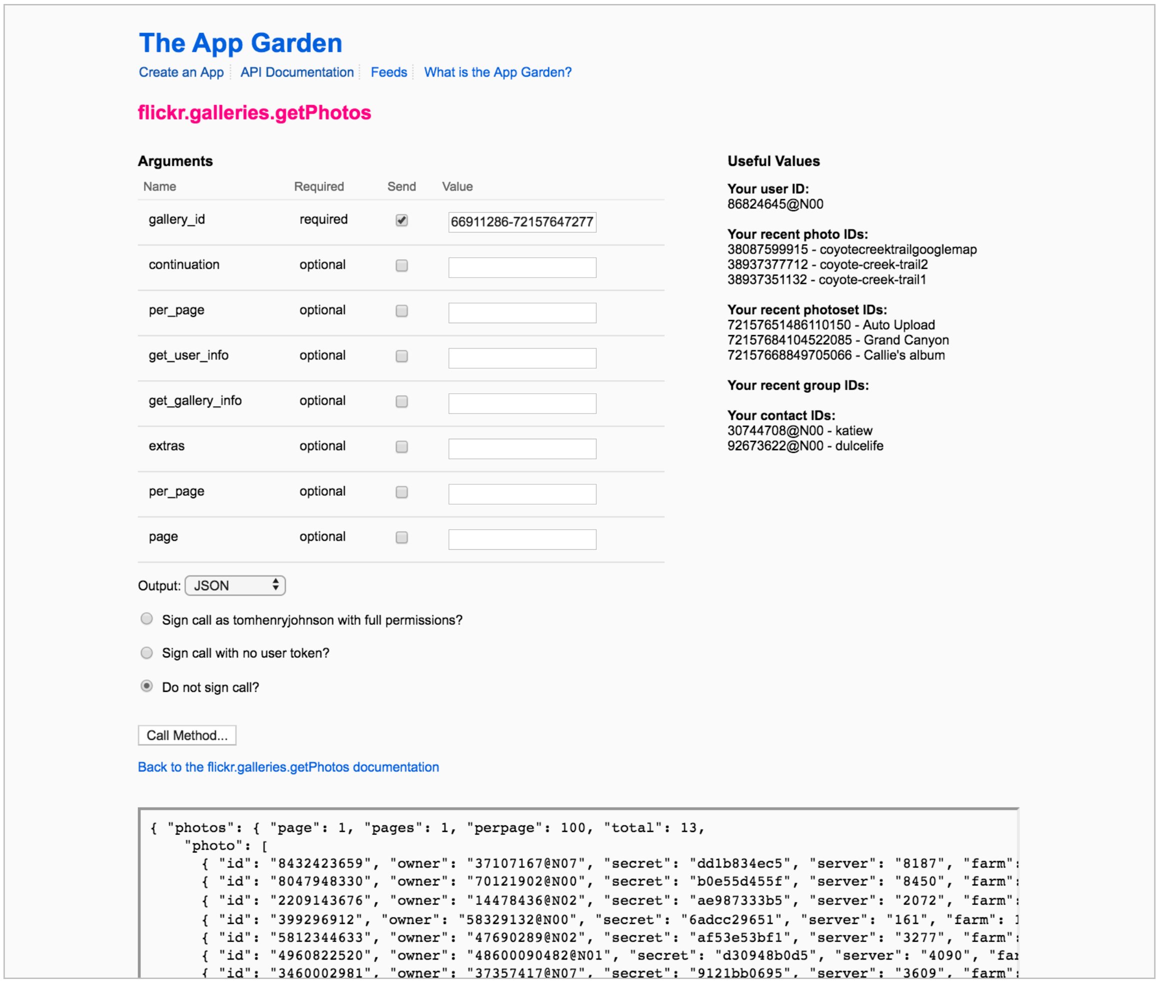
Your recent photoset IDs:
72157651486110150 - Auto
72157649733910233 - hover
72157649565389074 - Disco

Your recent group IDs:

Your contact IDs:
30744708@N00 - katiew
92673622@N00 - dulcelife

Insert the `gallery_id`, select **JSON** for the output, select **Do not sign call** (we're just testing here, so we don't need extra security), and then click **Call Method**.

Here's the result:



flickr.galleries.getPhotos

Arguments

Name	Required	Send	Value
gallery_id	required	☑	66911286-72157647277
continuation	optional	☐	
per_page	optional	☐	
get_user_info	optional	☐	
get_gallery_info	optional	☐	
extras	optional	☐	
per_page	optional	☐	
page	optional	☐	

Output: JSON

○ Sign call as tomhenryjohnson with full permissions?

○ Sign call with no user token?

● Do not sign call?

Useful Values

Your user ID:
86824645@N00

Your recent photo IDs:
38087599915 - coyotecreektrailgooglemap
38937377712 - coyote-creek-trail2
38937351132 - coyote-creek-trail1

Your recent photoset IDs:
72157651486110150 - Auto Upload
72157684104522085 - Grand Canyon
72157668849705066 - Callie's album

Your recent group IDs:

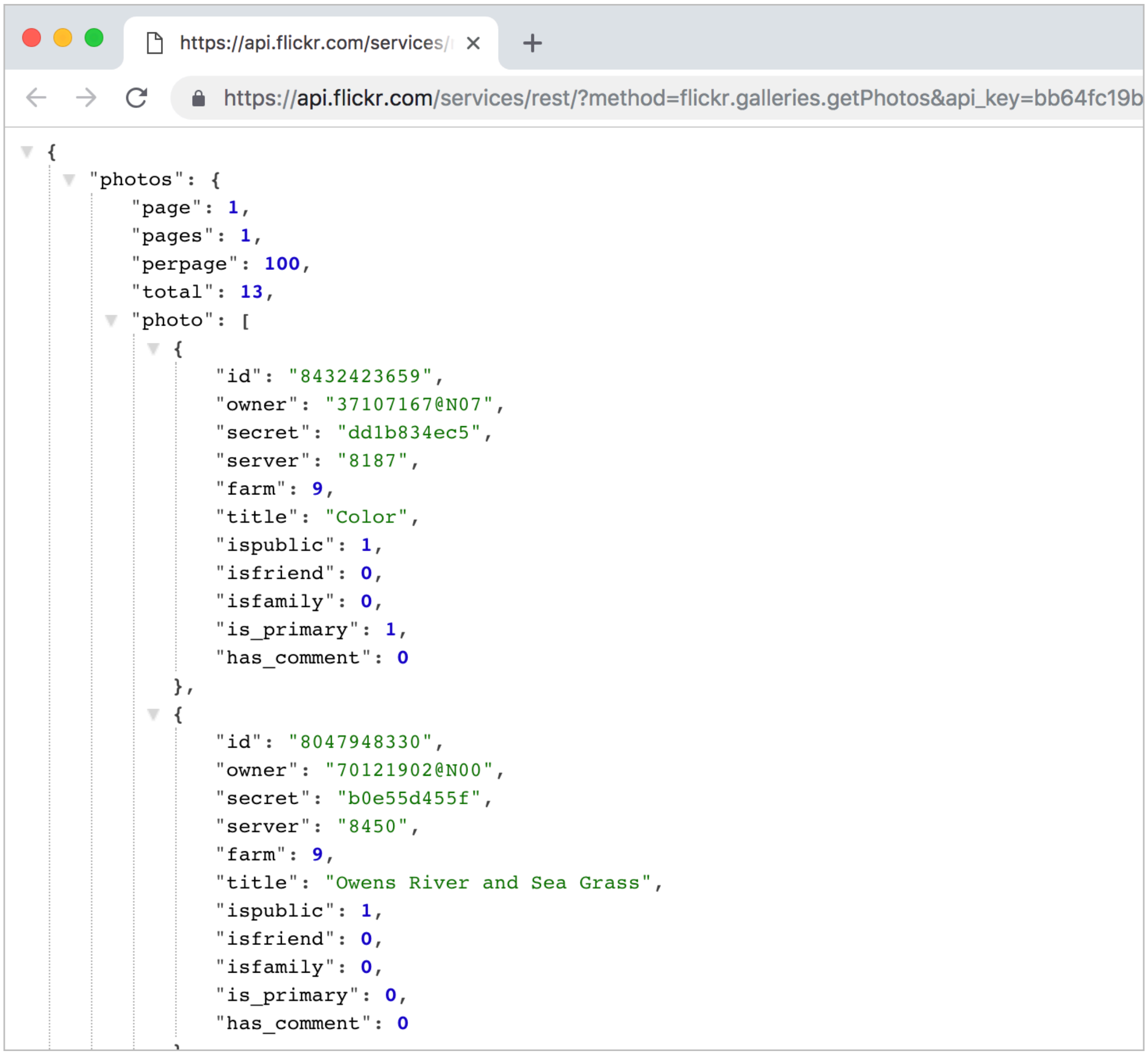
Your contact IDs:
30744708@N00 - katiew
92673622@N00 - dulcelife

```
{ "photos": { "page": 1, "pages": 1, "perpage": 100, "total": 13,
    "photo": [
      { "id": "8432423659", "owner": "37107167@N07", "secret": "dd1b834ec5", "server": "8187", "farm":
      { "id": "8047948330", "owner": "70121902@N00", "secret": "b0e55d455f", "server": "8450", "farm":
      { "id": "2209143676", "owner": "14478436@N02", "secret": "ae987333b5", "server": "2072", "farm":
      { "id": "399296912", "owner": "58329132@N00", "secret": "6adcc29651", "server": "161", "farm": 1
      { "id": "5812344633", "owner": "47690289@N02", "secret": "af53e53bf1", "server": "3277", "farm":
      { "id": "4960822520", "owner": "48600090482@N01", "secret": "d30948b0d5", "server": "4090", "fa
      { "id": "3460002981", "owner": "37357417@N07", "secret": "9121bb0695", "server": "3609", "farm":
```

The URL below the response shows the right syntax for using this method:

```
https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&ap
i_key=APIKEY&gallery_id=66911286-72157647277042064&format=json&nojsoncallbac
k=1
```

If you submit the request directly in your browser using the given URL, you can see the same response but in the browser rather than the API Explorer:


https://api.flickr.com/services/
https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api_key=bb64fc19b

```
{
  "photos": {
    "page": 1,
    "pages": 1,
    "perpage": 100,
    "total": 13,
    "photo": [
      {
        "id": "8432423659",
        "owner": "37107167@N07",
        "secret": "dd1b834ec5",
        "server": "8187",
        "farm": 9,
        "title": "Color",
        "ispublic": 1,
        "isfriend": 0,
        "isfamily": 0,
        "is_primary": 1,
        "has_comment": 0
      },
      {
        "id": "8047948330",
        "owner": "70121902@N00",
        "secret": "b0e55d455f",
        "server": "8450",
        "farm": 9,
        "title": "Owens River and Sea Grass",
        "ispublic": 1,
        "isfriend": 0,
        "isfamily": 0,
        "is_primary": 0,
        "has_comment": 0
```

I'm using the JSON Formatting extension for Chrome to make the JSON response more readable. Without this plugin, the JSON response is compressed.

4. Analyze the response

All the necessary information is included in this response in order to display photos on our site, but it's not entirely intuitive how we construct the image source URLs from the response.

In other words, the information a user needs to achieve a goal isn't explicit in the API *reference* documentation. The reference docs explain only what's returned in the response, not how to actually use the response.

The Photo Source URLs page in the documentation explains it:

> You can construct the source URL to a photo once you know its ID, server ID, farm ID, and secret, as returned by many API methods. The URL takes the following format:
>
> ```
> https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}.jpg
>         or
> https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}_[mstzb].jpg
>         or
> https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{o-secret}_o.(jpg|gif|png)
> ```

Here's what an item in the JSON response looks like:

```
{
  "photos": {
    "page": 1,
    "pages": 1,
    "perpage": 100,
    "total": 13,
    "photo": [
      {
      "id": "8432423659",
      "owner": "37107167@N07",
      "secret": "dd1b834ec5",
      "server": "8187",
      "farm": 9,
      "title": "Color",
      "ispublic": 1,
      "isfriend": 0,
      "isfamily": 0,
      "is_primary": 1,
      "has_comment": 0
      },
      ...
    ]
    }
}
```

You access these fields through dot notation. It's a good idea to log the whole object to the console just to explore it better.

5. Pull out the information you need

The following code uses jQuery to loop through each of the responses and inserts the necessary components into an image tag to display each photo.

```
<html>
<style>
img {max-height:125px; margin:3px; border:1px solid #dedede;}
</style>
<body>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>


<script>

var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api_key=APIKEY&gallery_id=66911286-72157647277042064&format=json&nojsoncallback=1",
  "method": "GET",
  "headers": {}
}

$.ajax(settings).done(function (data) {
  console.log(data);



$("#galleryTitle").append(data.photos.photo[0].title + " Gallery");
            $.each( data.photos.photo, function( i, gp ) {

var farmId = gp.farm;
var serverId = gp.server;
var id = gp.id;
var secret = gp.secret;

console.log(farmId + ", " + serverId + ", " + id + ", " + secret);

//  https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}.jpg

$("#flickr").append('<img src="https://farm' + farmId + '.staticflickr.com/' + serverId + '/' + id + '_' + secret + '.jpg"/>');

});
});

</script>


<h2><div id="galleryTitle"></div></h2>
<div style="clear:both;"/>
<div id="flickr"/>
```

```
</body>
</html>
```

Here's what the code is doing:

- In this code, the ajax method from jQuery gets the JSON payload. The payload is assigned to the `data` argument and then logged to the console.
- The data object contains an object called `photos`, which contains an array called `photo`. The `title` field is a property in an object in the `photo` array. The `title` is accessed through this dot notation: `data.photos.photo[0].title`.
- To get each item in the object, jQuery's each method loops through an object's properties. Note that jQuery `each` method is commonly used for looping through results to get values. For the first argument (`data.photos.photo`), you identify the object that you want to access. For the `function( i, gp )` arguments, you list an index and value. You can use any names you want here. `gp` becomes a variable that refers to the `data.photos.photo` object you're looping through. `i` refers to the starting point through the object. (You don't need to refer to `i` beyond the instance here unless you want to begin or end the loop at a certain point.)
- To access the properties in the JSON object, we use `gp.farm` instead of `data.photos.photo[0].farm`, because `gp` is an object reference to `data.photos.photo[i]`.
- After the `each` function iterates through the response, I added some variables to make it easier to work with these components (using `serverId` instead of `gp.server`, etc.). And a `console.log` message checks to ensure we're getting values for each of the elements we need.
- This comment shows where we need to plug in each of the variables:

```
//  https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}.jpg
```

The final line shows how you insert those variables into the HTML:

```
$("#flickr").append('<img src="https://farm' + farmId + '.staticflickr.co
m/' + serverId + '/' + id + '_' + secret + '.jpg"/>');
```

A common pattern in programming is to loop through a response. This code example used the `each` method from jQuery to look through all the items in the response and do something with each item. Sometimes you incorporate logic that loops through items and looks for certain conditions present to decide whether to take some action. Pay attention to methods for looping, as they are common scenarios in programming.

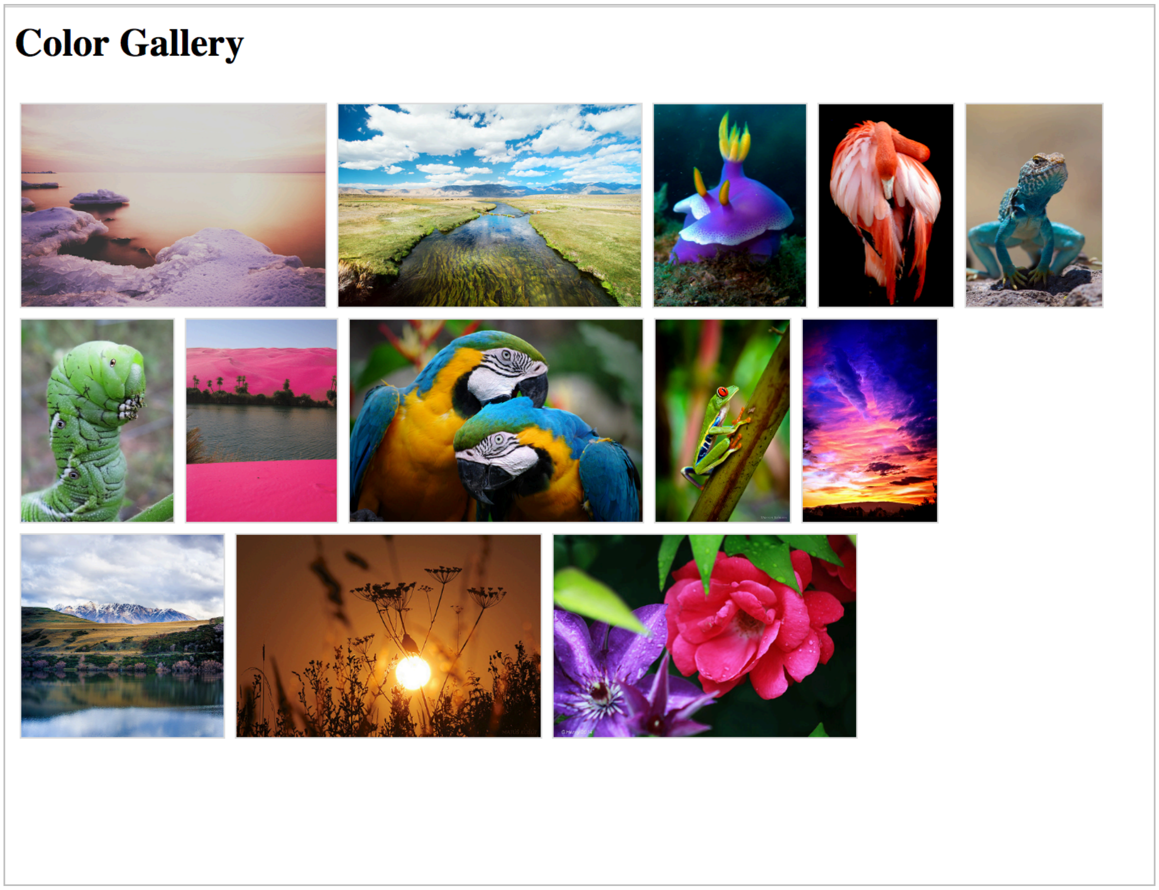
For more information, see these topics:

- Inspect the JSON from the response payload[]((p. 908)docapis_json_console.html)
- Access and print a specific JSON value (p. 103)
- Dive into dot notation (p. 109)

Final Result

You can view a demo of the Color Gallery integration here.

The result looks like this:



Color Gallery

Color Gallery integrated through the Flickr API

Get wind speed using the Aeris Weather API

In this activity, you'll use the Aeris Weather API to get the wind speed for a specific place (your choice). You'll then display the wind speed on a web page based on some conditional logic about the wind speed's value.

The Aeris Weather API

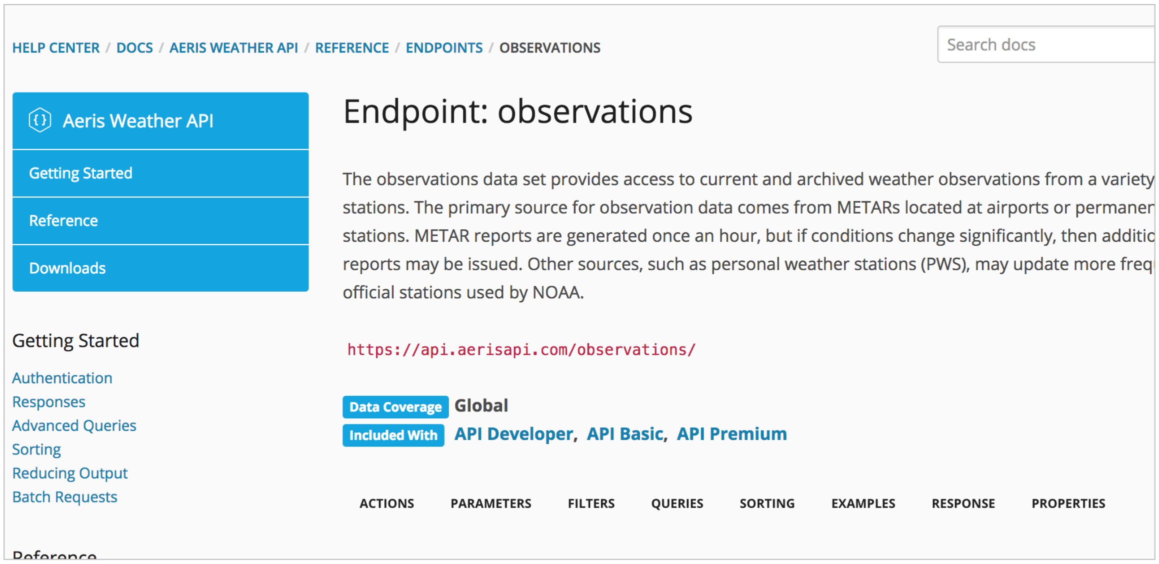
In earlier tutorials, you integrated information from the OpenWeatherMap API ↗. It's probably a good idea to compare this OpenWeatherMap API with another weather API. The Aeris Weather API ↗ is one of the most interesting, well-documented and powerful weather APIs I've encountered.

1. Get the API keys

See the Getting Started ↗ page for information on how to register and get API keys. (Get the free version of the keys available to development projects.) You will need both the ID and secret to make API calls. See Get the Aeris Weather API secret and ID (p. 70) for more detail.

2. Construct the request

Browse through the available endpoints ↗ in the Weather API and look for one that would give you the wind speed. The `observations` ↗ endpoint provides information about wind speed, as does `forecasts` ↗. The response from `observations` looks a little simpler, so let's use that endpoint.



Endpoint: observations

The observations data set provides access to current and archived weather observations from a variety stations. The primary source for observation data comes from METARs located at airports or permaner stations. METAR reports are generated once an hour, but if conditions change significantly, then additic reports may be issued. Other sources, such as personal weather stations (PWS), may update more freq official stations used by NOAA.

`https://api.aerisapi.com/observations/`

To get the forecast details for Santa Clara, California, add `santa%20clara,ca` after `/observations`, like this:

```
http://api.aerisapi.com/observations/santa%20clara,ca?client_id=CLIENT_ID&cl
ient_secret=CLIENT_SECRET
```

You will need to swap in your own client ID and secret in place of `CLIENT_ID` and `CLIENT_SECRET` in the code above.

3. Analyze the response

Here's the response from the request:

```
{
  "success": true,
  "error": null,
  "response": {
    "id": "KSJC",
    "loc": {
      "long": -121.91666666667,
      "lat": 37.366666666667
    },
    "place": {
      "name": "san jose",
      "state": "ca",
      "country": "us"
    },
    "profile": {
      "tz": "America/Los_Angeles",
      "elevM": 24,
      "elevFT": 79
    },
    "obTimestamp": 1544806380,
    "obDateTime": "2018-12-14T08:53:00-08:00",
    "ob": {
      "timestamp": 1544806380,
      "dateTimeISO": "2018-12-14T08:53:00-08:00",
      "tempC": 14.4,
      "tempF": 58,
      "dewpointC": 6.1,
      "dewpointF": 43,
      "humidity": 58,
      "pressureMB": 1016,
      "pressureIN": 30,
      "spressureMB": 1014,
      "spressureIN": 29.94,
      "altimeterMB": 1017,
      "altimeterIN": 30.03,
      "windKTS": 14,
      "windKPH": 26,
      "windMPH": 16,
      "windSpeedKTS": 14,
      "windSpeedKPH": 26,
      "windSpeedMPH": 16,
      "windDirDEG": 140,
      "windDir": "SE",
      "windGustKTS": 20,
      "windGustKPH": 37,
      "windGustMPH": 23,
      "flightRule": "VFR",
      "visibilityKM": 16.09344,
      "visibilityMI": 10,
      "weather": "Cloudy",
      "weatherShort": "Cloudy",
      "weatherCoded": "::OV",
      "weatherPrimary": "Cloudy",
      "weatherPrimaryCoded": "::OV",
      "cloudsCoded": "OV",
      "icon": "cloudy.png",
      "heatindexC": 14,
      "heatindexF": 58,
      "windchillC": 14,
      "windchillF": 58,
      "feelslikeC": 14,
      "feelslikeF": 58,
      "isDay": true,
      "sunrise": 1544800479,
      "sunriseISO": "2018-12-14T07:14:39-08:00",
      "sunset": 1544835063,
      "sunsetISO": "2018-12-14T16:51:03-08:00",
      "snowDepthCM": null,
      "snowDepthIN": null,
      "precipMM": 0,
      "precipIN": 0,
      "solradWM2": 55,
      "solradMethod": "estimated",
      "ceilingFT": 7000,
      "ceilingM": 2133.6,
      "light": 24,
      "QC": "O",
      "QCcode": 10,
      "windGustSpeedKTS": 20,
      "windGustSpeedKPH": 37,
      "windGustSpeedMPH": 23,
      "sky": 100
    },
    "raw": "KSJC 141653Z 14014G20KT 10SM FEW024 BKN070 OVC110 14/06 A3003 RMK AO2 SLP168 T01440061",
    "relativeTo": {
      "lat": 37.35411,
      "long": -121.95524,
      "bearing": 68,
      "bearingENG": "ENE",
      "distanceKM": 3.684,
      "distanceMI": 2.289
    }
  }
}
```

`windSpeedMPH` is the value we want.

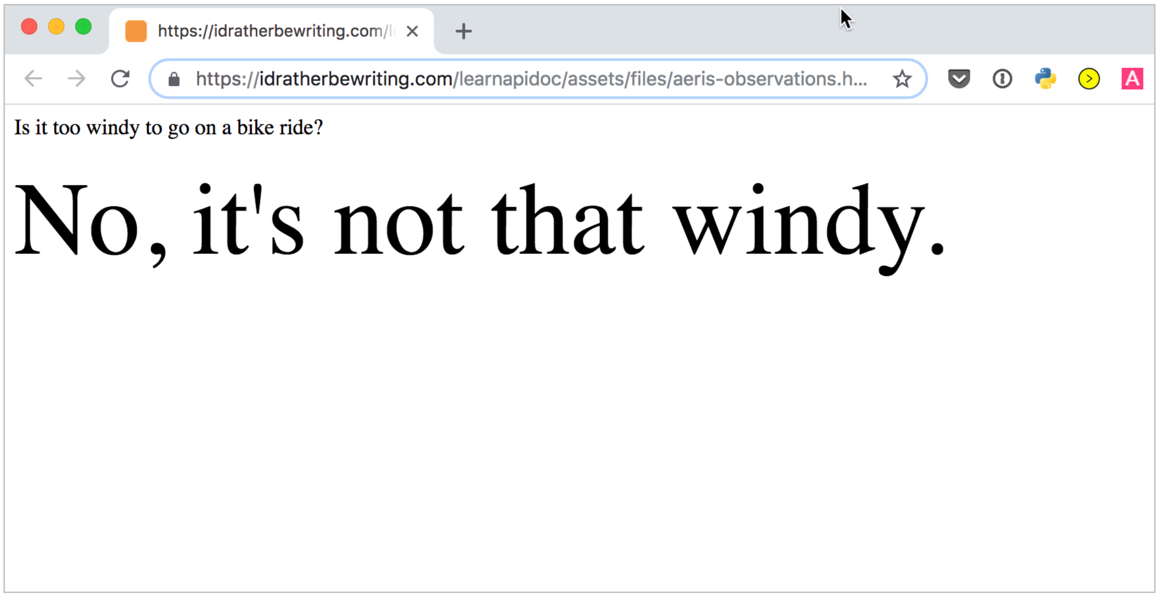
4. Pull out the values from the response

To get the `windSpeedMPH`, you would access it through dot notation like this: `data.response.ob.windSpeedMPH`.

To add a little variety to the code samples, let’s add some conditional logic in the display. The following code checks to see if `data.response.ob.windSpeedMPH` is greater than `15`. If so, it prints `Yes, it's too windy.` If not, it prints, `No, it's not that windy.`

```
<html>
   <body>
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
      <script>

         jQuery.ajax({
             url: "http://api.aerisapi.com/observations/santa%20clara,ca",
             type: "GET",
             data: {
                 "client_id": "CLIENTID",
                 "client_secret": "CLIENTSECRET",
             },
         })
         .done(function(data, textStatus, jqXHR) {
             console.log("HTTP Request Succeeded: " + jqXHR.status);
             console.log(data);
             if (data.response.ob.windSpeedMPH > 15) {
                 var windAnswer = "Yes, it's too windy.";
         }
                 else  {
                  var windAnswer = "No, it's not that windy.";
                 }
             $("#windAnswer").append(windAnswer)
         })
         .fail(function(jqXHR, textStatus, errorThrown) {
             console.log("HTTP Request Failed");
         })
         .always(function() {
             /* ... */
         });


      </script>
      <p>Is it too windy to go on a bike ride?</p>
      <div id="windAnswer" style="font-size:76px"></div>

   </body>
</html>
```

Here’s the result:


https://idratherbewriting.com/l
https://idratherbewriting.com/learnapidoc/assets/files/aeris-observations.h...

Is it too windy to go on a bike ride?

No, it's not that windy.

Aeris Weather example

For fun, change the value from `15` to `1` in the code and refresh the page. It will probably say “Yes, it’s too windy.” JavaScript is fun because it’s easy to adjust some parameters and see the results through your browser. With other programming languages, you would likely need to compile or run the code in an IDE to see the result. In both cases, always test out the code and make it run yourself.

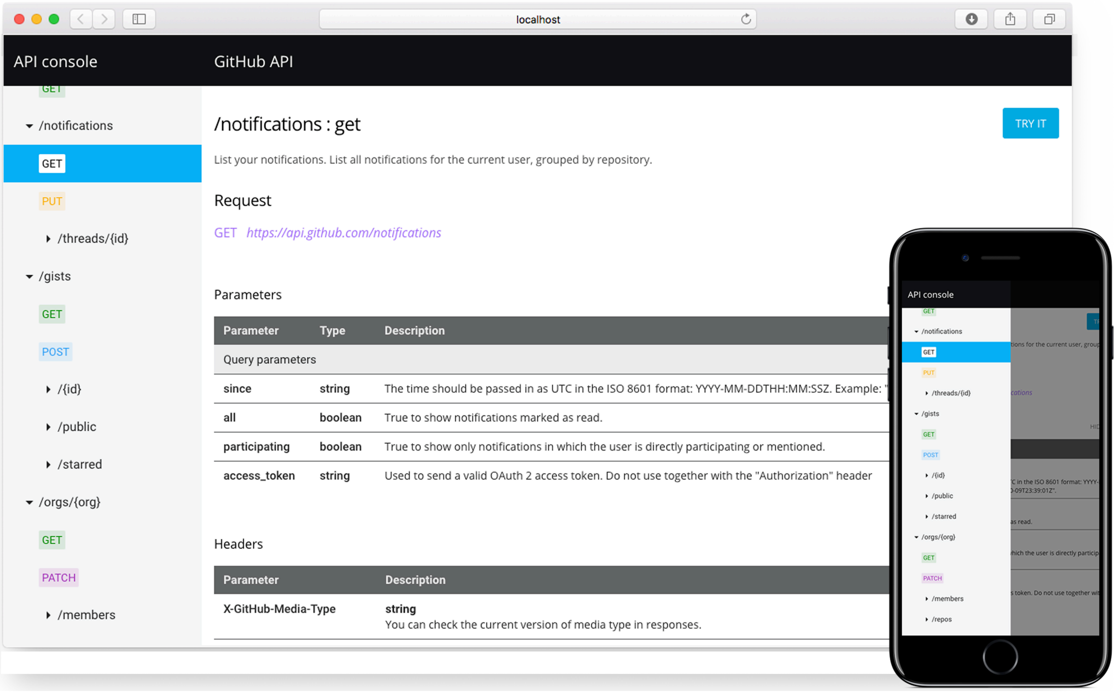
RAML tutorial

RAML stands for REST API Modeling Language and is similar to the OpenAPI specification (p. 272). RAML is backed by Mulesoft, a full-service, end-to-end API company.

Note that unless you're publishing your docs with Mulesoft or another platform that specifically requires RAML, I recommend using the OpenAPI specification (p. 272) instead. However, Mulesoft offers enterprise-grade API design, management, and deployment capabilities. If you're using Mulesoft for your API, you probably want to use RAML for your documentation specification.

RAML overview

Similar to OpenAPI, after you create a RAML file that describes your API, it can be consumed by different platforms to parse and display the information in interactive outputs. The RAML format, which uses YML syntax, is human-readable, efficient, and simple. Here's what the RAML output looks like in API Console (which is similar to Swagger UI):



/notifications : get

List your notifications. List all notifications for the current user, grouped by repository.

Request

GET https://api.github.com/notifications

Parameters

Parameter	Type	Description
Query parameters		
since	**string**	The time should be passed in as UTC in the ISO 8601 format: YYYY-MM-DDTHH:MM:SSZ. Example: "
all	**boolean**	True to show notifications marked as read.
participating	**boolean**	True to show only notifications in which the user is directly participating or mentioned.
access_token	**string**	Used to send a valid OAuth 2 access token. Do not use together with the "Authorization" header

Headers

Parameter	Description
X-GitHub-Media-Type	**string** You can check the current version of media type in responses.

Sample API Console output with RAML

You can see a demo with the Google Drive API.

Auto-generating client SDK code

It's important to note that with these REST API specifications (as with the OpenAPI spec), you're not just describing an API to generate a nifty doc output with an interactive console. There are tools that can also generate client SDKs and other code from the spec into a library that you can integrate into your project. These tools can make it easier for developers to make requests to your API and receive responses.

Additionally, the interactive console can provide a way to prototype and beta test your API before developers even start coding. Mulesoft offers a “mocking service” for your API that simulates calls and responses. The idea of the mocking service is to design your API the right way from the start, without iterating with different versions as you try to get the endpoints right.

Sample spec for OpenWeatherMap API

To understand the proper syntax and format for RAML, you need to read the RAML spec ↗ and look at some examples. See this RAML tutorial ↗ to get started.

Here’s the OpenWeatherMap API (which we’ve been using in this course (p. 65)) formatted in the RAML spec. (I actually just used API Transformer ↗ to convert my OpenAPI 3.0 spec to RAML.) As you can see, RAML is highly similar to the OpenAPI spec.

```
#%RAML 1.0
title: OpenWeatherMap API
version: 2.5
baseUri: https://mocksvc.mulesoft.com/mocks/082e051b-e960-48f7-9d75-2f49af8ccd86/data/2.5/ # baseUri: http://api.openweathermap.org/data/2.5/
baseUriParameters: {}
documentation:
- title: OpenWeatherMap API
  content: 'Get the current weather, daily forecast for 16 days, and a three-hour-interval forecast for 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API. <br/><br/> **Note**: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.'
securitySchemes:
  auth:
    type: Pass Through
    describedBy:
      queryParameters:
        appid:
          required: true
          displayName: appid
          description: API key to authorize requests.
          type: string
types:
  SuccessfulResponse:
    displayName: Successful response
    type: object
    properties:
      coord:
        required: false
        displayName: coord
        type: Coord
      weather:
        required: false
        displayName: weather
        description: (more info Weather condition codes)
        type: array
        items:
          type: Weather
      base:
        required: false
        displayName: base
        description: Internal parameter
        type: string
      main:
        required: false
        displayName: main
        type: Main
      visibility:
        required: false
        displayName: visibility
        description: Visibility, meter
        type: integer
        format: int32
      wind:
        required: false
        displayName: wind
        type: Wind
      clouds:
        required: false
        displayName: clouds
        type: Clouds
      rain:
        required: false
        displayName: rain
        type: Rain
      snow:
        required: false
        displayName: snow
        type: Snow
      dt:
        required: false
        displayName: dt
        description: Time of data calculation, unix, UTC
        type: integer
        format: int32
      sys:
        required: false
        displayName: sys
        type: Sys
      id:
        required: false
        displayName: id
        description: City ID
        type: integer
        format: int32
      name:
        required: false
        displayName: name
        type: string
      cod:
        required: false
        displayName: cod
        description: Internal parameter
        type: integer
        format: int32
  Coord:
    displayName: Coord
    type: object
    properties:
      lon:
        required: false
        displayName: lon
        description: City geo location, longitude
        type: number
        format: double
      lat:
        required: false
        displayName: lat
        description: City geo location, latitude
        type: number
        format: double
  Weather:
    displayName: Weather
    type: object
    properties:
      id:
        required: false
        displayName: id
        description: Weather condition id
        type: integer
        format: int32
      main:
        required: false
        displayName: main
        description: Group of weather parameters (Rain, Snow, Extreme etc.)
        type: string
      description:
        required: false
        displayName: description
        description: Weather condition within the group
        type: string
      icon:
        required: false
        displayName: icon
        description: Weather icon id
        type: string
  Main:
    displayName: Main
    type: object
    properties:
      temp:
        required: false
        displayName: temp
        description: 'Temperature. Unit Default: Kelvin, Metric: Celsius, Im
perial: Fahrenheit.'
        type: number
        format: double
      pressure:
        required: false
        displayName: pressure
        description: Atmospheric pressure (on the sea level, if there is no
sea_level or grnd_level data), hPa
        type: integer
        format: int32
      humidity:
        required: false
        displayName: humidity
        description: Humidity, %
        type: integer
        format: int32
      temp_min:
        required: false
        displayName: temp_min
        description: 'Minimum temperature at the moment. This is deviation f
rom current temp that is possible for large cities and megalopolises geograp
hically expanded (use these parameter optionally). Unit Default: Kelvin, Met
ric: Celsius, Imperial: Fahrenheit.'
        type: number
        format: double
      temp_max:
        required: false
        displayName: temp_max
        description: 'Maximum temperature at the moment. This is deviation f
rom current temp that is possible for large cities and megalopolises geograp
hically expanded (use these parameter optionally). Unit Default: Kelvin, Met
ric: Celsius, Imperial: Fahrenheit.'
        type: number
        format: double
      sea_level:
        required: false
        displayName: sea_level
        description: Atmospheric pressure on the sea level, hPa
        type: number
        format: double
      grnd_level:
        required: false
        displayName: grnd_level
        description: Atmospheric pressure on the ground level, hPa
        type: number
        format: double
  Wind:
    displayName: Wind
    type: object
    properties:
      speed:
        required: false
        displayName: speed
        description: 'Wind speed. Unit Default: meter/sec, Metric: meter/se
c, Imperial: miles/hour.'
```

```
        type: number
        format: double
      deg:
        required: false
        displayName: deg
        description: Wind direction, degrees (meteorological)
        type: integer
        format: int32
  Clouds:
    displayName: Clouds
    type: object
    properties:
      all:
        required: false
        displayName: all
        description: Cloudiness, %
        type: integer
        format: int32
  Rain:
    displayName: Rain
    type: object
    properties:
      3h:
        required: false
        displayName: 3h
        description: Rain volume for the last 3 hours
        type: integer
        format: int32
  Snow:
    displayName: Snow
    type: object
    properties:
      3h:
        required: false
        displayName: 3h
        description: Snow volume for the last 3 hours
        type: number
        format: double
  Sys:
    displayName: Sys
    type: object
    properties:
      type:
        required: false
        displayName: type
        description: Internal parameter
        type: integer
        format: int32
      id:
        required: false
        displayName: id
        description: Internal parameter
        type: integer
        format: int32
      message:
        required: false
        displayName: message
        description: Internal parameter
        type: number
        format: double
      country:
        required: false
        displayName: country
        description: Country code (GB, JP etc.)
        type: string
      sunrise:
        required: false
        displayName: sunrise
        description: Sunrise time, unix, UTC
        type: integer
        format: int32
      sunset:
        required: false
        displayName: sunset
        description: Sunset time, unix, UTC
        type: integer
        format: int32
/weather:
  get:
    displayName: Call current weather data for one location
    description: Access current weather data for any location on Earth inclu
ding over 200,000 cities! Current weather is frequently updated based on glo
bal models and data from more than 40,000 weather stations.
    securedBy:
    - auth
    queryParameters:
      q:
        required: false
        displayName: q
        description: '**City name**. *Example: London*. You can call by cit
y name, or by city name and country code. The API responds with a list of re
sults that match a searching word. For the query value, type the city name a
nd optionally the country code divided by a comma; use ISO 3166 country code
s.'
        type: string
      id:
        required: false
        displayName: id
        description: "**City ID**. *Example: `2172797`*. You can call by cit
y ID. The API responds with the exact result. The List of city IDs can be do
wnloaded [here](http://bulk.openweathermap.org/sample/). You can include mul
tiple cities in this parameter &mdash; just separate them by commas. The lim
```

```
it of locations is 20. *Note: A single ID counts as a one API call. So, if y
ou have 3 city IDs, it’s treated as 3 API calls.*"
        type: string
      lat:
        required: false
        displayName: lat
        description: '**Latitude**. *Example: 35*. The latitude coordinate o
f the location of your interest. Must use with `lon`.'
        type: string
      lon:
        required: false
        displayName: lon
        description: '**Longitude**. *Example: 139*. Longitude coordinate o
f the location of your interest. Must use with `lat`.'
        type: string
      zip:
        required: false
        default: 94040,us
        example: 94040,us
        displayName: zip
        description: '**Zip code**. Search by zip code. *Example: 95050,u
s*. Please note that if the country is not specified, the search uses USA a
s a default.'
        type: string
      units:
        required: false
        default: standard
        example:
          value: imperial
        displayName: units
        description: '**Units**. *Example: imperial*. Possible values: `metr
ic`, `imperial`. When you do not use the `units` parameter, the format is `s
tandard` by default.'
        type: string
        enum:
        - standard
        - metric
        - imperial
      lang:
        required: false
        default: en
        example:
          value: en
        displayName: lang
        description: '**Language**. *Example: en*. You can use lang paramete
r to get the output in your language. We support the following languages tha
t you can use with the corresponded lang values: Arabic - `ar`, Bulgarian -
`bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English -
`en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `g
l`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Kore
an - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `n
```

```
l`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedi
sh - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`,
Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese T
raditional - `zh_tw`.'
        type: string
        enum:
        - ar
        - bg
        - ca
        - cz
        - de
        - el
        - en
        - fa
        - fi
        - fr
        - gl
        - hr
        - hu
        - it
        - ja
        - kr
        - la
        - lt
        - mk
        - nl
        - pl
        - pt
        - ro
        - ru
        - se
        - sk
        - sl
        - es
        - tr
        - ua
        - vi
        - zh_cn
        - zh_tw
      Mode:
        required: false
        default: json
        example:
          value: json
        displayName: Mode
        description: '**Mode**. *Example: html*. Determines the format of th
e response. Possible values are `xml` and `html`. If the mode parameter is e
mpty, the format is `json` by default.'
        type: string
        enum:
        - json
        - xml
        - html
    responses:
      200:
        description: Successful response
        body:
          application/json:
            displayName: response
            description: Successful response
            type: SuccessfulResponse
      404:
        description: Not found response
        body: {}
```

RAML's specification format is highly similar to the OpenAPI spec. There's really not much point in having multiple specifications for REST APIs in the industry, so my guess is that in five years, RAML will be obsolete.

Outputs

You can generate outputs using the RAML spec from a variety of platforms. Here are three ways:

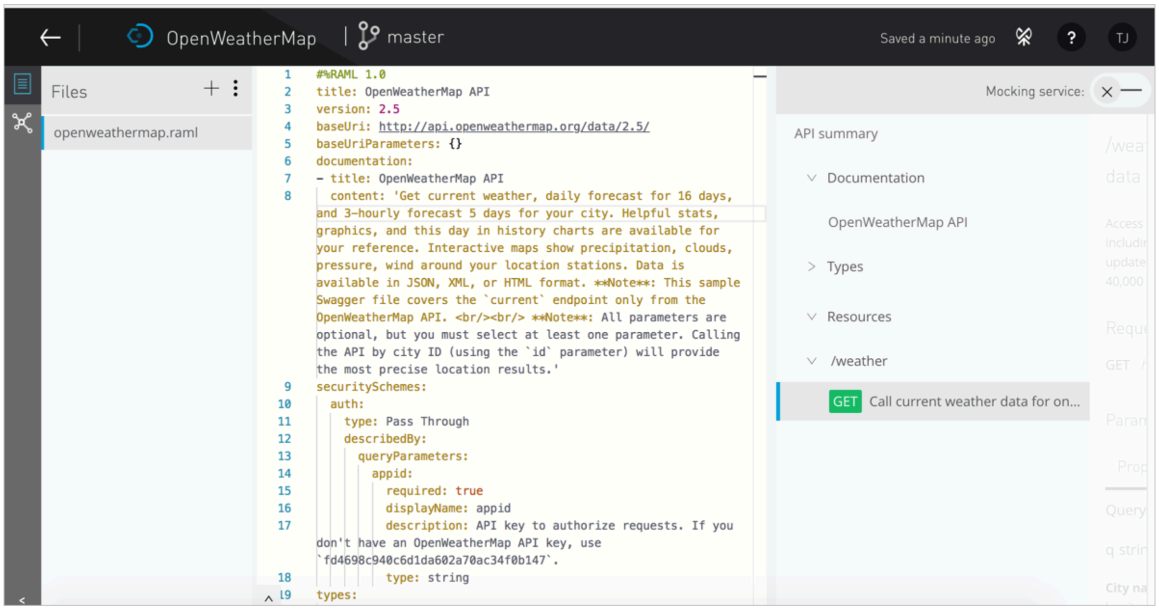
- Developer Portal on Anypoint platform (p. 934). Choose this option if you are developing and delivering your API on Mulesoft's Anypoint platform.
- API Console output (p. 936). Choose this option if you want a standalone API Console output delivered on your own server. (Note that this option also allows you to embed the console in an iframe.)
- RAML2HTML project (p. 936). Choose this option if you want a simple HTML output of your API documentation. No interactive console is included.

More detail about each of these options is provided in the following sections.

Developer Portal on Anypoint platform

Anypoint is the API development platform on Mulesoft. APIs you develop with Mulesoft can be shared and viewed on their Anypoint Exchange portal.

The Anypoint console has a dashboard where you can work on your RAML definition, add other documentation pages (outside the spec), configure mocking services, and more. If your company already uses Mulesoft for other API development services, it would make sense to use their documentation and portal features as well. You can also export your RAML definition to OpenAPI through the Anypoint console.

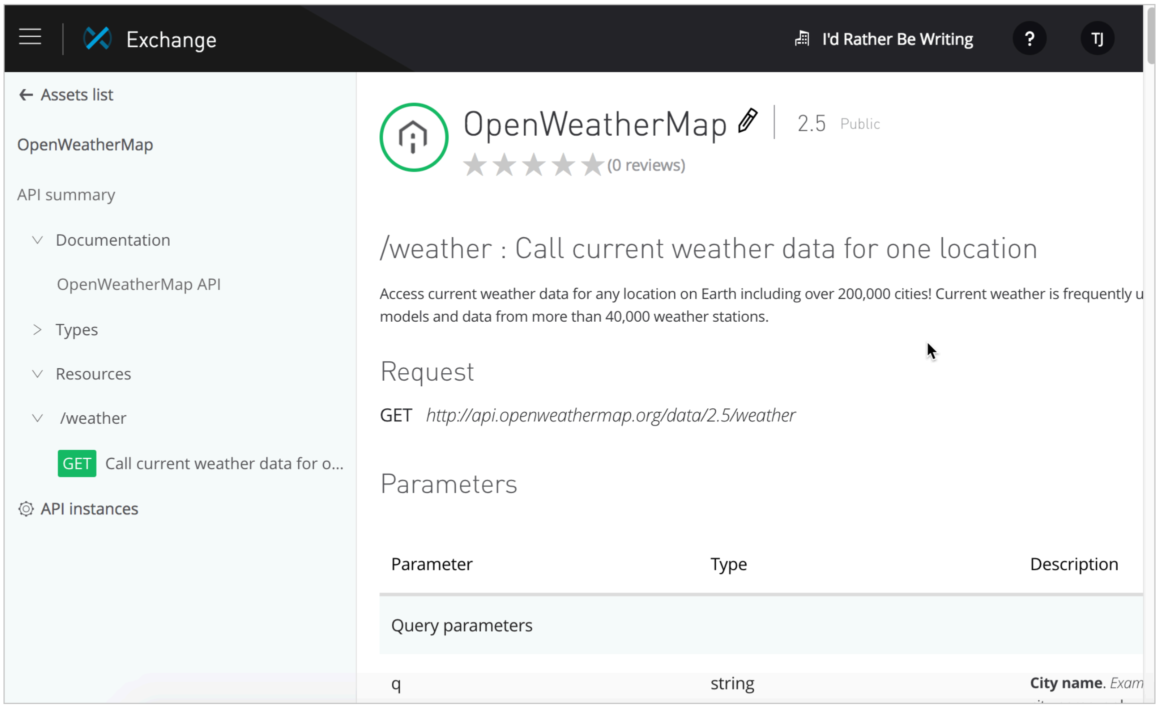


```
#%RAML 1.0
title: OpenWeatherMap API
version: 2.5
baseUri: http://api.openweathermap.org/data/2.5/
baseUriParameters: {}
documentation:
- title: OpenWeatherMap API
  content: 'Get current weather, daily forecast for 16 days,
and 3-hourly forecast 5 days for your city. Helpful stats,
graphics, and this day in history charts are available for
your reference. Interactive maps show precipitation, clouds,
pressure, wind around your location stations. Data is
available in JSON, XML, or HTML format. **Note**: This sample
Swagger file covers the `current` endpoint only from the
OpenWeatherMap API. <br/><br/> **Note**: All parameters are
optional, but you must select at least one parameter. Calling
the API by city ID (using the `id` parameter) will provide
the most precise location results.'
securitySchemes:
  auth:
    type: Pass Through
    describedBy:
      queryParameters:
        appid:
          required: true
          displayName: appid
          description: API key to authorize requests. If you
don't have an OpenWeatherMap API key, use
`fd4698c940c6d1da602a70ac34f0b147`.
          type: string
types:
```

The UI when working with RAML definitions in the Anypoint platform

Note that you can add additional pages to your documentation in Anypoint. (Kudos to the Mulesoft team for recognizing that API documentation is more than just a set of reference endpoints.)

Here's what the sample OpenWeatherMap API looks like in Anypoint Exchange:



/weather : Call current weather data for one location

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently u models and data from more than 40,000 weather stations.

Request

GET http://api.openweathermap.org/data/2.5/weather

Parameters

Parameter	Type	Description
Query parameters		
q	string	**City name**. Exam

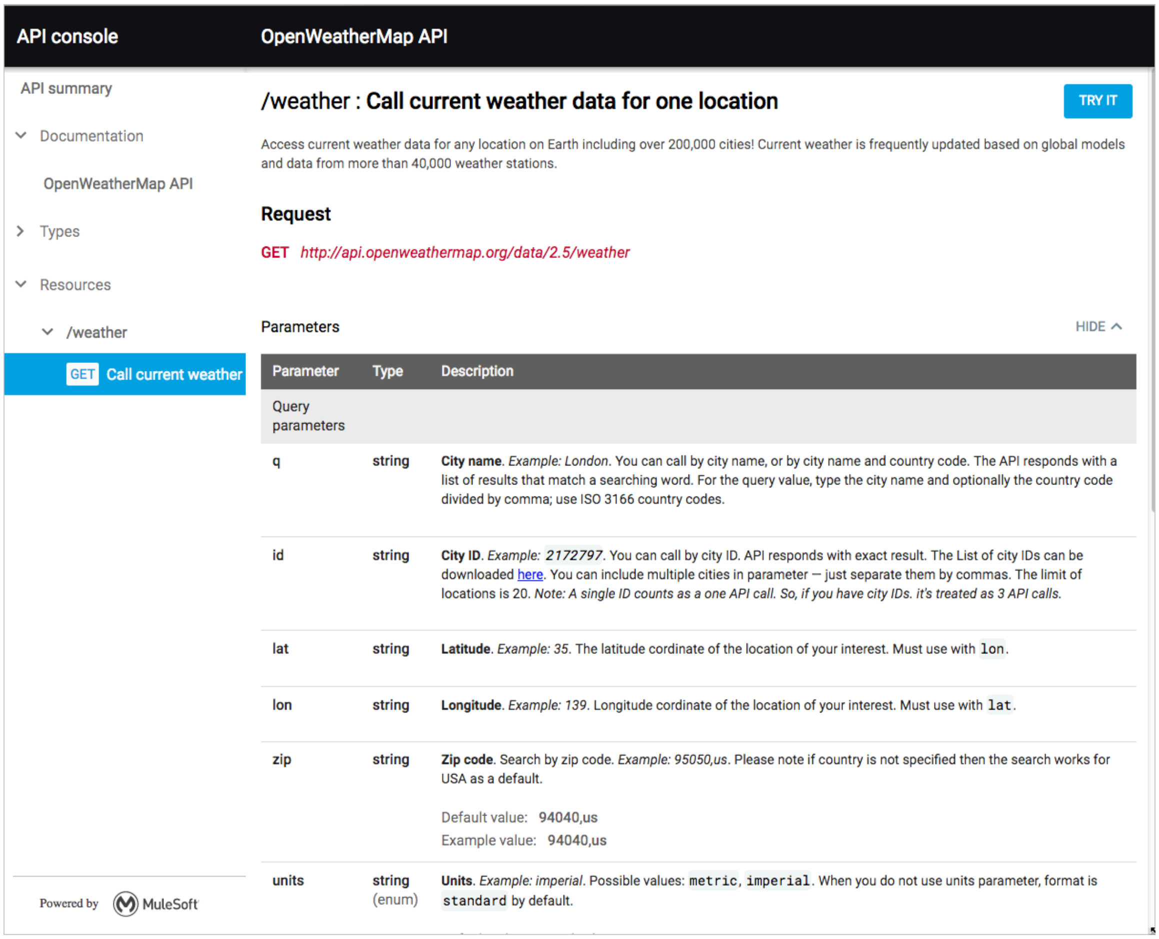
OpenWeather API documentation in Mulesoft's Anypoint Exchange

One of the unique options Mulesoft offers is called API Notebook. This is a unique tool designed by Mulesoft that allows you to provide interactive code examples that leverage your RAML spec.

API Console output

You can download the same code that generates the output on the Anypoint Platform and create your own interactive documentation. The standalone project (on GitHub) is called API Console. Here's a demo. Instructions for building API Console from a RAML file are available on the api-console project on GitHub. After you build the output, upload it to a web server and go to the index.html file.

Here's the OpenWeather API in API Console:



/weather : **Call current weather data for one location**

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

Request

GET *http://api.openweathermap.org/data/2.5/weather*

Parameters

Parameter	Type	Description
Query parameters		
q	string	**City name**. *Example: London*. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by comma; use ISO 3166 country codes.
id	string	**City ID**. *Example:* `2172797`. You can call by city ID. API responds with exact result. The List of city IDs can be downloaded here. You can include multiple cities in parameter — just separate them by commas. The limit of locations is 20. *Note: A single ID counts as a one API call. So, if you have city IDs. it's treated as 3 API calls.*
lat	string	**Latitude**. *Example: 35*. The latitude cordinate of the location of your interest. Must use with `lon`.
lon	string	**Longitude**. *Example: 139*. Longitude cordinate of the location of your interest. Must use with `lat`.
zip	string	**Zip code**. Search by zip code. *Example: 95050,us*. Please note if country is not specified then the search works for USA as a default. Default value: 94040,us Example value: 94040,us
units	string (enum)	**Units**. *Example: imperial*. Possible values: `metric`, `imperial`. When you do not use units parameter, format is `standard` by default.

Sample RAML output in API Console

You can also embed the API console as an HTML element.

RAML2HTML project

Finally, you can also use a tool called RAML2HTML project to generate HTML documentation from a RAML spec. Here's an example of what the RAML2HTML output looks like. It's a static HTML output without any interactivity. See the RAML2HTML documentation for instructions on generating this output.

Conclusion

Exploring Mulesoft in more depth is beyond the scope of this tutorial, but hopefully I've introduced you to RAML and Mulesoft. Overall, large platforms that process and display your API documentation can only do so if your documentation aligns with a spec their tools can parse. RAML provides this standard spec for the Mulesoft ecosystem of tools. Their enterprise-level API tools provide powerful capabilities for your API.

API Blueprint tutorial

Just as Swagger defines a spec for describing a REST API, API Blueprint is another specification for describing REST APIs. If you describe your API with this blueprint, tools that support API Blueprint can read and display the information.

Note that unless you're using a platform that specifically requires API Blueprint, I recommend using the OpenAPI specification (p. 272) instead.

What is API Blueprint

The API Blueprint spec is written in a Markdown-flavored syntax. It's not regular Markdown, but it has a lot of the same, familiar Markdown syntax. However, the blueprint is a particular schema that is either valid or not valid based on the element names, order, spacing, and other details. In this way, it's not nearly as flexible or forgiving as Markdown. But it may be preferable to YAML.

Sample blueprint

Here's a sample blueprint to give you an idea of the syntax:

```
FORMAT: 1A
HOST: http://polls.apiblueprint.org/

# test

Polls is a simple API allowing consumers to view polls and vote in them.

# Polls API Root [/]

This resource does not have any attributes. Instead, it offers the initial
API affordances in the form of the links in the JSON body.

It is recommended to follow the “url" link values,
[Link](https://tools.ietf.org/html/rfc5988), or Location headers where
applicable to retrieve resources. Instead of constructing your own URLs,
to keep your client decoupled from implementation details.

## Retrieve the Entry Point [GET]

+ Response 200 (application/json)

        {
            "questions_url": "/questions"
        }

## Group Question

Resources related to questions in the API.

## Question [/questions/{question_id}]

A Question object has the following attributes:

+ question
+ published_at - An ISO8601 date when the question was published.
+ url
+ choices - An array of Choice objects.

+ Parameters
    + question_id: 1 (required, number) - ID of the Question in form of an i
nteger

### View a Questions Detail [GET]

+ Response 200 (application/json)

        {
            "question": "Favourite programming language?",
            "published_at": "2014-11-11T08:40:51.620Z",
            "url": "/questions/1",
            "choices": [
                {
                    "choice": "Swift",
                    "url": "/questions/1/choices/1",
                    "votes": 2048
                }, {
                    "choice": "Python",
                    "url": "/questions/1/choices/2",
                    "votes": 1024
                }, {
                    "choice": "Objective-C",
                    "url": "/questions/1/choices/3",
                    "votes": 512
                }, {
                    "choice": "Ruby",
                    "url": "/questions/1/choices/4",
                    "votes": 256
                }
            ]
        }

## Choice [/questions/{question_id}/choices/{choice_id}]

+ Parameters
    + question_id: 1 (required, number) - ID of the Question in form of an integer
    + choice_id: 1 (required, number) - ID of the Choice in form of an integer

### Vote on a Choice [POST]

This action allows you to vote on a question's choice.

+ Response 201

    + Headers

            Location: /questions/1

## Questions Collection [/questions{?page}]

+ Parameters
    + page: 1 (optional, number) - The page of questions to return

### List All Questions [GET]

+ Response 200 (application/json)

    + Headers

            Link: </questions?page=2>; rel="next"
    + Body

            [
                {
                    "question": "Favourite programming language?",
                    "published_at": "2014-11-11T08:40:51.620Z",
                    "url": "/questions/1",
                    "choices": [
                        {
                            "choice": "Swift",
                            "url": "/questions/1/choices/1",
                            "votes": 2048
                        }, {
                            "choice": "Python",
                            "url": "/questions/1/choices/2",
                            "votes": 1024
                        }, {
                            "choice": "Objective-C",
                            "url": "/questions/1/choices/3",
                            "votes": 512
                        }, {
                            "choice": "Ruby",
                            "url": "/questions/1/choices/4",
                            "votes": 256
                        }
                    ]
                }
            ]

### Create a New Question [POST]

You may create your own question using this action. It takes a JSON
object containing a question and a collection of answers in the
form of choices.

+ question (string) - The question
+ choices (array[string]) - A collection of choices.

+ Request (application/json)

        {
            "question": "Favourite programming language?",
            "choices": [
                "Swift",
                "Python",
                "Objective-C",
                "Ruby"
            ]
        }

+ Response 201 (application/json)
    + Headers

            Location: /questions/2

    + Body

            {
                "question": "Favourite programming language?",
                "published_at": "2014-11-11T08:40:51.620Z",
                "url": "/questions/2",
                "choices": [
                    {
                        "choice": "Swift",
                        "url": "/questions/2/choices/1",
                        "votes": 0
                    }, {
                        "choice": "Python",
                        "url": "/questions/2/choices/2",
                        "votes": 0
                    }, {
                        "choice": "Objective-C",
                        "url": "/questions/2/choices/3",
                        "votes": 0
                    }, {
                        "choice": "Ruby",
                        "url": "/questions/2/choices/4",
                        "votes": 0
                    }
                ]
            }
```

For a tutorial on the blueprint syntax, see these resources:

- API Blueprint tutorial
- Apiary tutorial
- Github API Blueprint tutorial

You can find examples of different blueprints here. The examples can often clarify different aspects of the spec.

Parsing the blueprint

Many tools can parse an API blueprint. Drafter is one of the main parsers of the Blueprint. Many other tools build on Drafter and generate static HTML outputs of the blueprint. For example, Aglio can parse a blueprint and generate static HTML files.

For a more comprehensive list of tools, see the Tooling section on apiblueprint.org. (Some of these tools require quite a few prerequisites, so I omitted the tutorial steps here for generating the output on your own machine.)

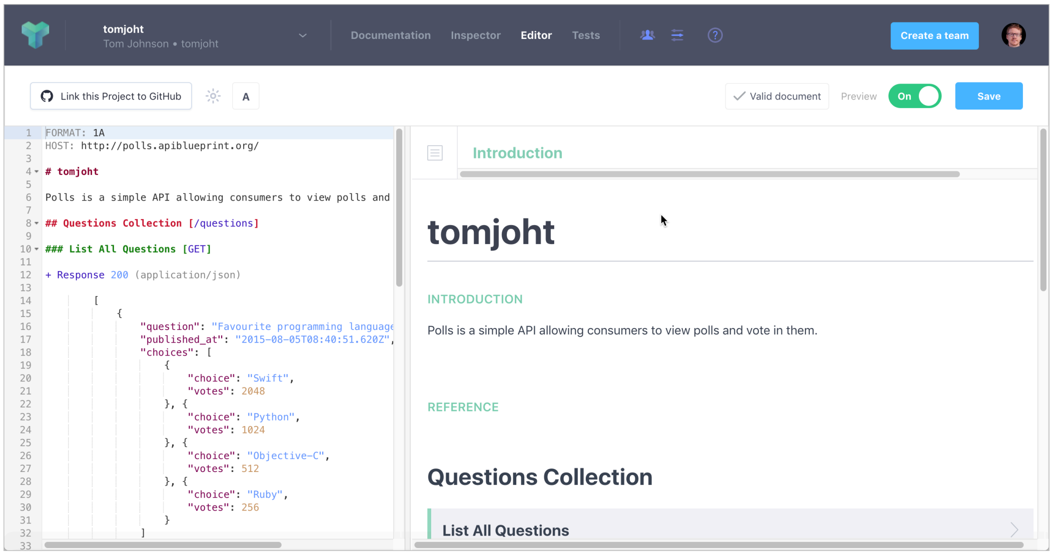
Create a sample HTML output using API Blueprint and Apiary

For this tutorial, we’ll use a platform called Apiary to read and display the API Blueprint. Apiary is just a hosted platform that will remove the need for installing local libraries and utilities to generate the output.

a. Create a new Apiary project

1. Go to apiary.io and click **Quick start with Github**. Sign in with your Github account. (If you don’t have a Github account, create one first.)
2. Sign up for a free hacker account and create a new project.

   You’ll be placed in the API Blueprint editor.

   ```
   FORMAT: 1A
   HOST: http://polls.apiblueprint.org/

   # tomjoht

   Polls is a simple API allowing consumers to view polls and

   ## Questions Collection [/questions]

   ### List All Questions [GET]

   + Response 200 (application/json)

           [
               {
                   "question": "Favourite programming language
                   "published_at": "2015-08-05T08:40:51.620Z",
                   "choices": [
                       {
                           "choice": "Swift",
                           "votes": 2048
                       }, {
                           "choice": "Python",
                           "votes": 1024
                       }, {
                           "choice": "Objective-C",
                           "votes": 512
                       }, {
                           "choice": "Ruby",
                           "votes": 256
                       }
                   ]
   ```

   tomjoht

   INTRODUCTION

   Polls is a simple API allowing consumers to view polls and vote in them.

   REFERENCE

   Questions Collection

   List All Questions

   API Blueprint editor on Apiary

   By default, the Polls blueprint is loaded so you can see how it looks. This blueprint gives you an example of the required format for the Apiary tool to parse and display the content. You can also see the raw file here.
3. At this point, you would start describing your API using the blueprint syntax in the editor. When you make a mistake, error flags indicate what’s wrong.

   You can read the Apiary tutorial and structure your documentation in the blueprint format. The syntax seems to accommodate different methods applied to the same resources.

   For this tutorial, you’ll integrate the OpenWeatherMap weather API information info formatted in the blueprint format.
4. Copy the following code, which aligns with the API Blueprint spec, and paste it into the Apiary Blueprint editor.

```
FORMAT: 1A
HOST: https://api.openweathermap.org/data/2.5/

# OpenWeatherMap API
Get the current weather, daily forecast for 16 days, and a three-hou
r-interval forecast for 5 days for your city. Helpful stats, graphic
s, and this day in history charts are available for your reference. I
nteractive maps show precipitation, clouds, pressure, wind around you
r location stations. Data is available in JSON, XML, or HTML format.
**Note**: This sample Swagger file covers the `current` endpoint only
from the OpenWeatherMap API. <br/><br/> **Note**: All parameters are
optional, but you must select at least one parameter. Calling the API
by city ID (using the `id` parameter) will provide the most precise l
ocation results.

## Authentication
This API uses Custom Query Parameter for its authentication.

The parameters that are needed to be sent for this type of authentica
tion are as follows:
+ `appid` - API key to authorize requests.

# Group Current Weather Data

## Weather [/weather{?q,id,lat,lon,zip,units,lang,Mode}]

### Call current weather data for one location [GET]
Access current weather data for any location on Earth including over
200,000 cities! Current weather is frequently updated based on global
models and data from more than 40,000 weather stations.
+ Parameters
    + q (string, optional)

        **City name**. *Example: London*. You can call by city name,
or by city name and country code. The API responds with a list of res
ults that match a searching word. For the query value, type the city
name and optionally the country code divided by a comma; use ISO 3166
country codes.

    + id (string, optional)

        **City ID**. *Example: `2172797`*. You can call by city ID. T
he API responds with the exact result. The List of city IDs can be do
wnloaded [here](http://bulk.openweathermap.org/sample/). You can incl
ude multiple cities in this parameter &mdash; just separate them by c
ommas. The limit of locations is 20. *Note: A single ID counts as a o
ne API call. So, if you have 3 city IDs, it’s treated as 3 API call
s.*

    + lat (string, optional)
```

```
        **Latitude**. *Example: 35*. The latitude coordinate of the location of your interest. Must use with `lon`.

    + lon (string, optional)

        **Longitude**. *Example: 139*. Longitude coordinate of the location of your interest. Must use with `lat`.

    + zip (string, optional) -

        **Zip code**. Search by zip code. *Example: 95050,us*. Please note that if the country is not specified, the search uses USA as a default.

        + Default: 94040,us
        + Sample: 94040,us
    + units (enum[string], optional)

        **Units**. *Example: imperial*. Possible values: `metric`, `imperial`. When you do not use the `units` parameter, the format is `standard` by default.

        + Default: standard
        + Sample: imperial
        + Members
            + `standard`
            + `metric`
            + `imperial`

    + lang (enum[string], optional)

        **Language**. *Example: en*. You can use lang parameter to get the output in your language. We support the following languages that you can use with the corresponded lang values: Arabic - `ar`, Bulgarian - `bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English - `en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `gl`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Korean - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `nl`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedish - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`, Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese Traditional - `zh_tw`.

        + Default: en
        + Sample: en
        + Members
            + `ar`
            + `bg`
            + `ca`
            + `cz`
            + `de`
            + `el`
            + `en`
            + `fa`
            + `fi`
            + `fr`
            + `gl`
            + `hr`
            + `hu`
            + `it`
            + `ja`
            + `kr`
            + `la`
            + `lt`
            + `mk`
            + `nl`
            + `pl`
            + `pt`
            + `ro`
            + `ru`
            + `se`
            + `sk`
            + `sl`
            + `es`
            + `tr`
            + `ua`
            + `vi`
            + `zh_cn`
            + `zh_tw`

    + Mode (enum[string], optional)

        **Mode**. *Example: html*. Determines the format of the respo
nse. Possible values are `xml` and `html`. If the mode parameter is e
mpty, the format is `json` by default.

        + Default: json
        + Sample: json
        + Members
            + `json`
            + `xml`
            + `html`


+ Response 200 (application/json)

        Successful response

    + Attributes (Successful response)
+ Response 404

        Not found response

# Data Structures

## Successful response (object)


### Properties
+ `coord` (Coord, optional)
+ `weather` (array[Weather], optional) - (more info Weather condition
codes)
+ `base`: `cmc stations` (string, optional) - Internal parameter
+ `main`: `cmc stations` (Main, optional)
+ `visibility`: `16093` (number, optional) - Visibility, meter
+ `wind`: `16093` (Wind, optional)
+ `clouds`: `16093` (Clouds, optional)
+ `rain`: `16093` (Rain, optional)
+ `snow`: `16093` (Snow, optional)
+ `dt`: `1435658272` (number, optional) - Time of data calculation, u
nix, UTC
+ `sys`: `1435658272` (Sys, optional)
+ `id`: `2172797` (number, optional) - City ID
+ `name`: `Cairns` (string, optional)
+ `cod`: `200` (number, optional) - Internal parameter


## Coord (object)


### Properties
+ `lon`: `145.77` (number, optional) - City geo location, longitude
+ `lat`: `-16.92` (number, optional) - City geo location, latitude


## Weather (object)


### Properties
+ `id`: `803` (number, optional) - Weather condition id
+ `main`: `Clouds` (string, optional) - Group of weather parameters
(Rain, Snow, Extreme etc.)
+ `description`: `broken clouds` (string, optional) - Weather conditi
on within the group
+ `icon`: `04n` (string, optional) - Weather icon id


## Main (object)
```

```
### Properties
+ `temp`: `293.25` (number, optional) - Temperature. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
+ `pressure`: `1019` (number, optional) - Atmospheric pressure (on the sea level, if there is no sea_level or grnd_level data), hPa
+ `humidity`: `83` (number, optional) - Humidity, %
+ `temp_min`: `289.82` (number, optional) - Minimum temperature at the moment. This is a deviation from current temp that is possible for large cities and megalopolises geographically expanded (use this parameter optionally). Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
+ `temp_max`: `295.37` (number, optional) - Maximum temperature at the moment. This is a deviation from current temp that is possible for large cities and megalopolises geographically expanded (use this parameter optionally). Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
+ `sea_level`: `984` (number, optional) - Atmospheric pressure on the sea level, hPa
+ `grnd_level`: `990` (number, optional) - Atmospheric pressure on the ground level, hPa


## Wind (object)


### Properties
+ `speed`: `5.1` (number, optional) - Wind speed. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour.
+ `deg`: `150` (number, optional) - Wind direction, degrees (meteorological)


## Clouds (object)


### Properties
+ `all`: `75` (number, optional) - Cloudiness, %


## Rain (object)


### Properties
+ `3h`: `3` (number, optional) - Rain volume for the last 3 hours


## Snow (object)


### Properties
+ `3h`: `6` (number, optional) - Snow volume for the last 3 hours
```

```


## Sys (object)


### Properties
+ `type`: `1` (number, optional) - Internal parameter
+ `id`: `8166` (number, optional) - Internal parameter
+ `message`: `0.0166` (number, optional) - Internal parameter
+ `country`: `AU` (string, optional) - Country code (GB, JP etc.)
+ `sunrise`: `1435610796` (number, optional) - Sunrise time, unix, UT
C
+ `sunset`: `1435650870` (number, optional) - Sunset time, unix, UTC
```

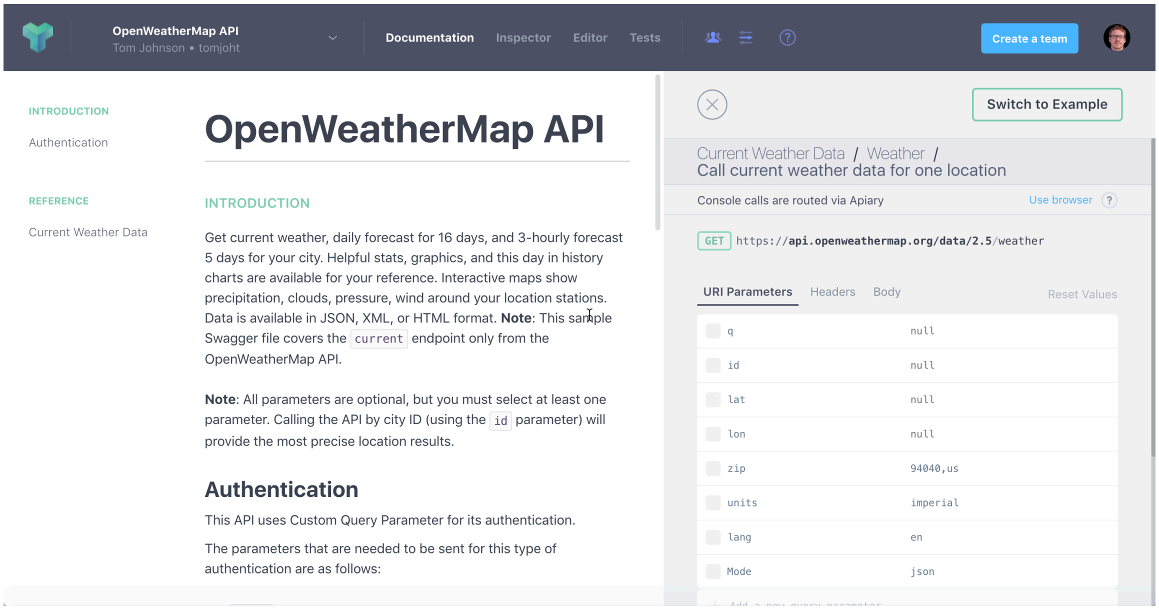
If the code isn't easy to copy and paste, you can view and download the file here ↗.

5. Click **Save**.

Similar to Swagger Editor, the Apiary editor shows you the output in the right pane.

b. Interact with the API on Apiary

In the Apiary's top navigation, click **Documentation**. Then interact with the API on Apiary by clicking first selecting an endpoint and then clicking **Switch to Console**. Call the resources and view the responses.



OpenWeatherMap API

INTRODUCTION

Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note**: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

Authentication

This API uses Custom Query Parameter for its authentication.

The parameters that are needed to be sent for this type of authentication are as follows:

Current Weather Data / Weather /
Call current weather data for one location

Console calls are routed via Apiary

GET `https://api.openweathermap.org/data/2.5/weather`

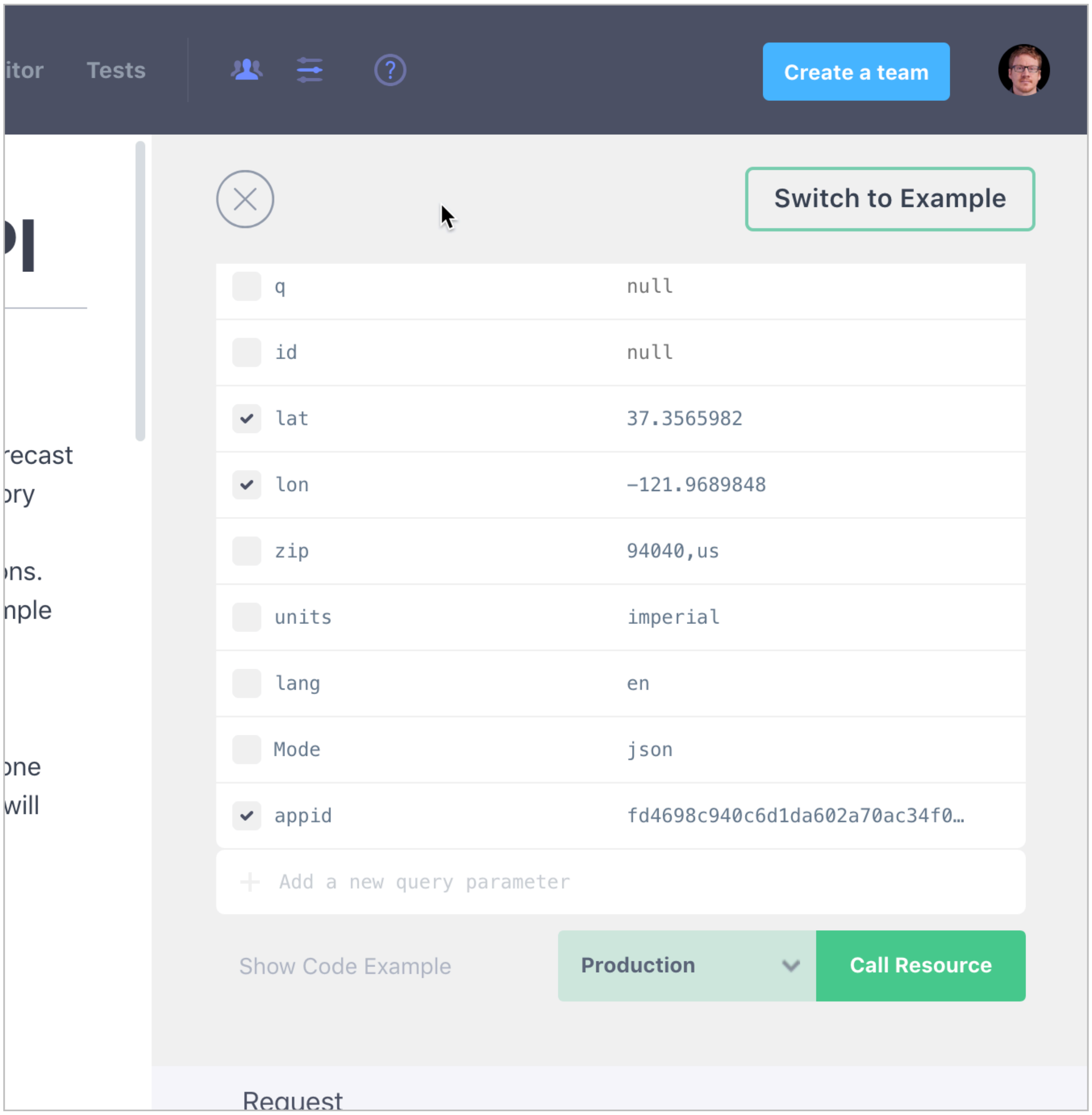
☐ q	null
☐ id	null
☐ lat	null
☐ lon	null
☐ zip	94040,us
☐ units	imperial
☐ lang	en
☐ Mode	json

Previewing the documentation for the OpenWeatherMap API definition in Apiary

You can switch between an Example and a Console view in the documentation. The Example view shows pre-built responses. The Console view allows you to enter your own values and generate dynamic responses based on your own API key. This dual display — both the Example and the Console views — might align better with user needs:

- For users who might not have good data or might not want to make requests that would affect their data, they can view the Example.
- For users who want to see how the API specifically returns either their data or specific parameters, they can use the Console view.

To call the resource successfully, you'll need to customize the `lat` and `lon` values (e.g., `37.3565982` and `-121.9689848`) and also add an `appid` parameter whose value is your API key. You can then call the resource successfully.



☐	q	null
☐	id	null
☑	lat	37.3565982
☑	lon	-121.9689848
☐	zip	94040,us
☐	units	imperial
☐	lang	en
☐	Mode	json
☑	appid	fd4698c940c6d1da602a70ac34f0…

Customizing the lat, long, and appid parameters in order to make a successful call

Apiary offers an impressive platform for both designing and publishing API reference docs. You don't have to use API Blueprint as your spec's format. You can also design your specification using OpenAPI (called "Swagger" in the Apiary UI).

API jeopardy answer key

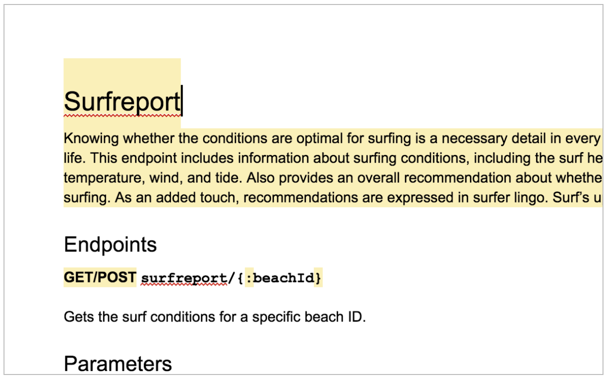
The API Jeopardy questions are in the Conclusion section of Workshop Activities (p. 17).

Note: This content doesn’t embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/jeopardy_answer_key.html] to view the content.

What's wrong with this topic answer key

This is the answer key for Activity: What’s wrong with this API reference topic (p. 173).

After you’ve finished commenting on the Google Doc, go to this annotated Google doc.



Surfreport

Knowing whether the conditions are optimal for surfing is a necessary detail in every
life. This endpoint includes information about surfing conditions, including the surf he
temperature, wind, and tide. Also provides an overall recommendation about whethe
surfing. As an added touch, recommendations are expressed in surfer lingo. Surf's u

Endpoints

GET/POST `surfreport/{:beachId}`

Gets the surf conditions for a specific beach ID.

Parameters

Annotated Google doc showing answers

Alternatively, expand the answers below to see how many issues you found. If you found additional issues beyond those noted here, feel free to drop me an email letting me know. You can also add your own comments on the Google Doc answer key.

Note: This content doesn’t embed well in print because it contains JavaScript. Please go to [https://idratherbewriting.com/learnapidoc/docapis_whats_wrong_answer_key.html] to view the content.

Menlo Park API Workshop Video Recording

I recorded this full-day API documentation in Menlo Park, California, on November 8, 2018. (For more details, see this blog post ↗.) This recording closely aligns with the latest content and activities in this course. I divided this all-day workshop into five separate videos.

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_course_videos.html] to view the content.

Denver API Workshop Video Recording

These videos are from an API documentation workshop that I gave in Denver, Colorado, on March 10, 2018. (For more details, see this blog post ↗. This version of the workshop is slightly different from the latest version of the course, but the content is still good. I divided the videos into three parts, since the workshop was slightly shorter than a full-day.

Note: This content doesn't embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_course_videos.html] to view the content.

API doc presentation video recordings

The following are other video recordings for API-related presentations.

Note: This content doesn’t embed well in print because it contains YouTube videos. Please go to [https://idratherbewriting.com/learnapidoc/docapis_course_videos.html] to view the content.