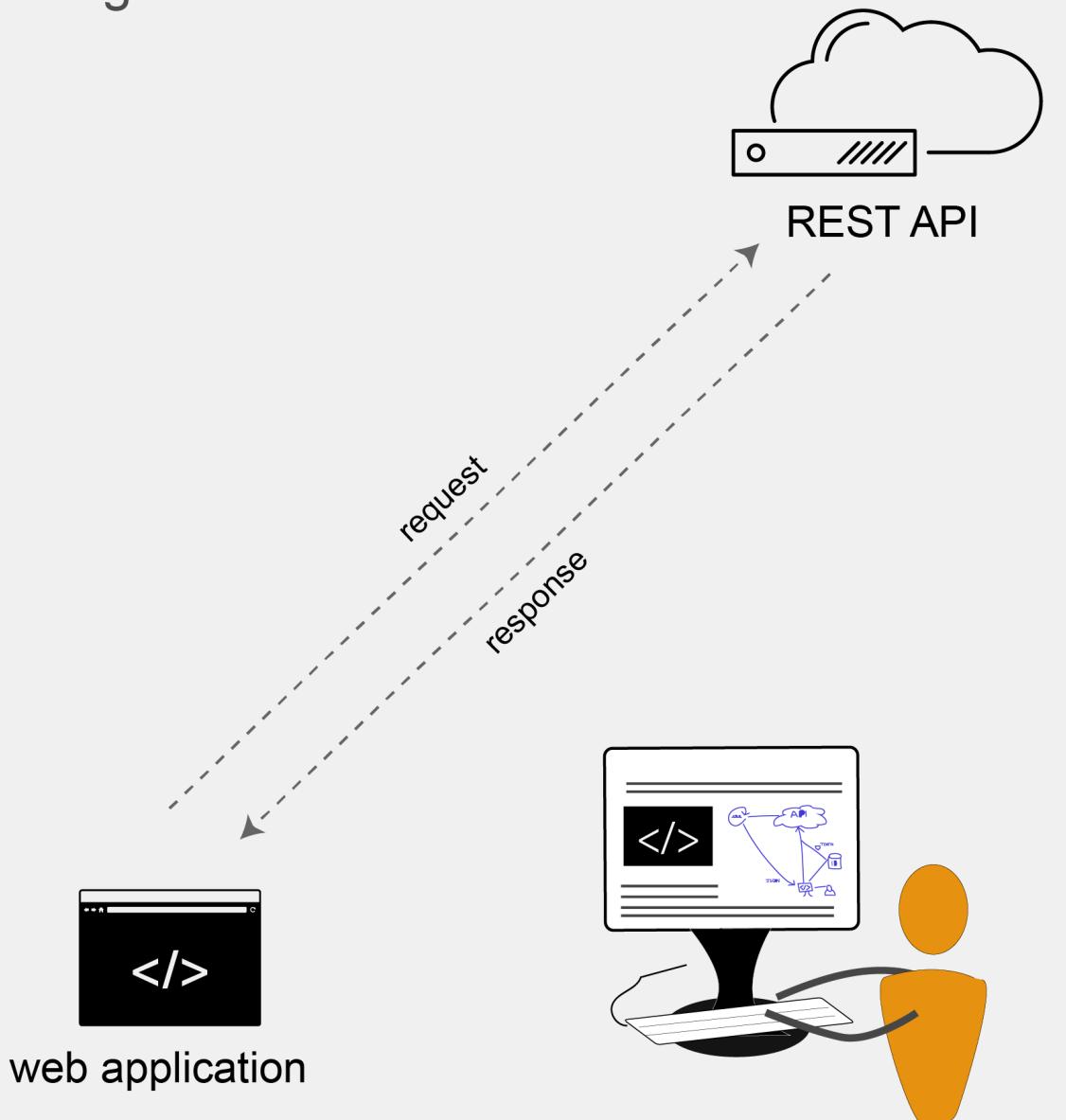


Documenting REST APIs

A guide for technical writers



by Tom Johnson

I'd Rather Be **Writing**

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.

Table of Contents

Part I: Introduction to REST APIs	6
Course Overview.....	7
Why I developed this course.....	11
About the author.....	13
The market for REST API documentation.....	15
What is a REST API?.....	22
Activity: Identify your goals.....	30
Part II: Using a REST API as a developer	31
Scenario for using a weather API.....	32
Get authorization keys	37
Submit requests through Postman	39
curl intro and installation.....	45
Make a curl call.....	47
Understand curl more	49
Use methods with curl	53
Analyze the JSON response	59
Inspect the JSON from the response payload.....	63
Access and print a specific JSON value.....	69
Dive into dot notation.....	73
Part III: Documenting API endpoints.....	78
A new endpoint to document	79
API reference tutorial overview	82
Step 1: Resource description (API ref tutorial).....	84
Step 2: Endpoints and methods (API ref tutorial)	89
Step 3: Parameters (API ref tutorial)	95
Step 4: Request example (API ref tutorial).....	104
Step 5: Response example and schema (API ref tutorial)	116
Putting it all together.....	131
Activity: Evaluate API reference docs for core elements	135
Activity: Find an Open Source Project.....	137
Part IV: OpenAPI specification and Swagger.....	141
Overview of REST API specification formats.....	142
Introduction to the OpenAPI spec and Swagger	143
Working in YAML	156
OpenAPI tutorial overview	160
Step 1: openapi object (OpenAPI tutorial)	165
Step 2: info object (OpenAPI tutorial)	168

Step 3: servers object (OpenAPI tutorial).....	170
Step 4: paths object (OpenAPI tutorial)	173
Step 5: components object (OpenAPI tutorial)	181
Step 6: security object (OpenAPI tutorial).....	200
Step 7: tags object (OpenAPI tutorial)	206
Step 8: externaldocs object (OpenAPI tutorial).....	209
Activity: Create an OpenAPI specification document.....	212
Swagger UI tutorial	214
Swagger UI Demo.....	223
SwaggerHub introduction and tutorial.....	224
Stoplight — visual modeling tools for creating your spec	233
Integrating Swagger UI with the rest of your docs	241

Part V: Testing your API documentation247

Overview to testing your docs	248
Set up a test environment.....	250
Test all instructions yourself	253
Test your assumptions.....	258
Activity: Test your project's documentation	262

Part VI: Documenting non-reference sections263

User guide topics.....	264
API overview	265
Getting started tutorial	269
Authentication and authorization	277
Status and error codes	286
Rate limiting and thresholds	293
Code samples and tutorials	298
SDKs and sample apps	308
Quick reference guide.....	316
API best practices.....	323
Glossary	326
Activity: Assess the non-reference content in your project	332

Part VII: Publishing your API documentation334

Overview for publishing API docs.....	335
List of 100 API doc sites	341
Design patterns with API doc sites.....	345
Docs-as-code tools	356
More about Markdown	360
Version control systems (such as Git).....	366
Static site generators	369
Hosting and deployment options.....	380
Headless CMS options	385

Which tool to choose for API docs — my recommendations.....	390
Activity: Manage content in a GitHub wiki.....	394
Activity: Use the GitHub Desktop Client.....	402
Pull request workflows through GitHub	412
Jekyll and CloudCannon continuous deployment	417
Case study: Switching tools to docs-as-code.....	425
Part VIII: Getting a Job in API Documentation	438
The job market for API technical writers.....	439
How much code do you need to know?	443
Locations for API doc writer jobs.....	453
Part IX: Glossary	461
API Glossary	462

Introduction to REST APIs

Course Overview	7
Why I developed this course	11
About the author.....	13
The market for REST API documentation.....	15
What is a REST API?	22
Activity: Identify your goals.....	30

Documenting APIs: A guide for technical writers

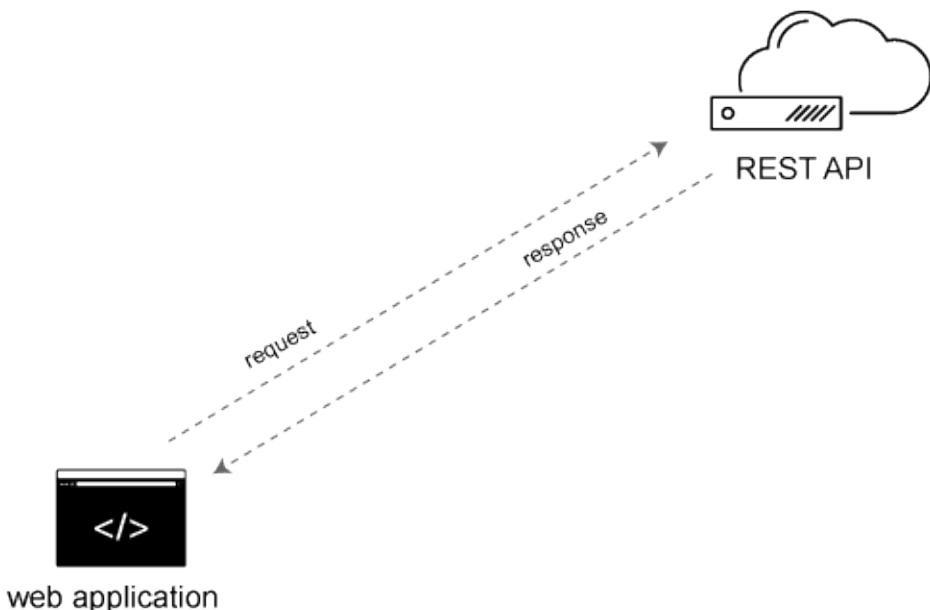
In this course on writing documentation for REST APIs, instead of just talking about abstract concepts, I contextualize REST APIs with a direct, hands-on approach. You'll learn about API documentation in the context of using some simple weather APIs to put a weather forecast on your site.

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 other standards, tools, and technologies associated with REST APIs.

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. This makes these tools and technologies more meaningful.

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? \(page 22\)](#). In your 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, you'll then shift perspectives and "become a technical writer" tasked with documenting a new endpoint that has been added to an API.

As a technical writer, you'll tackle each element of a reference topic in REST API documentation:

1. [Resource descriptions \(page 84\)](#)
2. [Endpoints and methods \(page 89\)](#)
3. [Parameters \(page 95\)](#)
4. [Request example \(page 104\)](#)
5. [Response example \(page 116\)](#)

Diving into these sections will give you a solid understanding about how to document REST APIs. You'll also learn how to document the [non-reference sections for an API \(page 263\)](#), such as the [getting started \(page 269\)](#), [status and error codes \(page 286\)](#), [request authorization \(page 277\)](#), and more.

Finally, you'll dive into different ways to [publish REST API documentation \(page 334\)](#), exploring tools and specifications such as [GitHub \(page 394\)](#), [Jekyll \(page 417\)](#), and other [Docs-as-code approaches \(page 356\)](#). 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 \(page 366\)](#).

I also dive into specifications such as the [OpenAPI specification \(page 160\)](#) and [Swagger \(page 143\)](#), which provides tooling for the OpenAPI specification. Additionally, I cover how to [document native library APIs \(page 0\)](#) and generate [Javadoc \(page 0\)](#). Throughout it all, I put these tools in a real, applicable context with examples and demos.

Course organization

This course is organized into the following sections:

- [Introduction to REST APIs \(page 6\)](#)
- [Using a REST API like a developer \(page 31\)](#)
- [Documenting endpoints \(page 78\)](#)
- [Testing your API documentation \(page 247\)](#)
- [Documenting non-reference sections \(page 263\)](#)
- [Publishing your API documentation \(page 334\)](#)
- [OpenAPI specification and Swagger \(page 141\)](#)
- [Documenting native library APIs \(page 0\)](#)
- [Getting a job in API documentation \(page 438\)](#)
- [Resources and glossary \(page 0\)](#)

You don't have to read the sections in order — feel free to skip around as you prefer. But some of the earlier sections (such as the section on [Using a REST API like a developer \(page 31\)](#), and the section on [Documenting endpoints \(page 78\)](#)) follow a somewhat sequential order with the same weather API scenario.

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 in various sections, but you can also see a consolidated list of activity content in the [Workshop Activities \(page 0\)](#).

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.

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

The most common reason people take this course is to transition to an 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. These activities are key to building experience and credibility with a portfolio.

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 out 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 exercises in this course:

- **Text editor.** ([Atom editor](#) or [Sublime Text](#) are good options, and they work on both Mac and Windows.)
- **Chrome browser.** [Chrome](#) provides a Javascript Console that works well for inspecting JSON, so we’ll be using Chrome. [Firefox](#) works well too if you prefer that.
- **Postman.** [Postman](#) is an app that allows you to make requests and see responses through a GUI client.
- **curl.** [curl](#) is essential for making requests to endpoints from the command line. Mac computers already have curl installed. Windows users should follow the instructions for installing curl [here](#). (Note: Choose one of the “free” versions to install curl.)
- **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](#) and also consider installing [iTerm2](#).
- **GitHub account.** [GitHub](#) will be used for various activities and is commonly used as an authentication service for developer tools. If you don’t already have a GitHub account, sign up for

one.

- **Stoplight account.** Stoplight provides visual modeling tools for working with the OpenAPI specification. Create a Stoplight account using your GitHub credentials. (You don't need the app.)
- **OpenWeatherMap API key.** We'll be using the [OpenWeatherMap API](https://openweathermap.org/) for some exercises. It might take a couple of hours for the API key to activate, 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, sign in and find your default API key from the developer dashboard. It's under the API Keys tab. Copy the key into a place you can easily find it.

Video recordings

For video recordings of this course, see the [Recorded Video Presentations \(page 0\)](#). The most recent full-length video of the entire course is a full-day API workshop I gave in Menlo Park, California in November 2018.

Slides

I have various slides that cover different sections of this course. See the following:

- Intro to API documentation
- Non-reference content in API docs
- OpenAPI and Swagger
- Publishing API documentation

These slides are all hosted on GitHub and are open source. I use [RevealJS](#) for slides, which lets me create the slide content in HTML. If you're a teacher using material from this course in your classroom, you can adapt the slides as needed for your lessons.

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](#).

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 were convinced that they either needed to train their existing technical writers on how to document APIs, or they would have 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 tech writing community needed this information. There are very few books or courses that dive into API documentation strategies for technical writers. (In fact, as of this writing, my course is the only one.)

Second, I knew that through feedback, I could refine the information and make it better. Almost no content is finished on its first release. Instead, content needs to iterate over a period of time through user testing and feedback. This is the case with help content, and it's the same for course material as well.

Finally, the content would help drive traffic to my site. 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.

Of course, not all comments or emails are praiseworthy. A lot of 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 went back and clarified or strengthened 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 go 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 skim ahead when you already know the material, or slow down when you need more time to absorb the material.

One of my goals for the content is to keep this course a living, evolving document. 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. My hope is that it will primarily serve the following audiences:

- Professional technical writers looking to transition from GUI 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.

If you have general feedback about this course, feel free to [drop me a line \(page 0\)](#).

About the author

In case you'd like to know a little bit about me, I'm currently based in the San Francisco Bay area of California working at Amazon. I mainly write documentation for third-party developers creating apps for the Amazon Appstore, primarily related to Fire TV.

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 out 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 for 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 a number of 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.

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, deep 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.



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.

The market for REST API documentation

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

Diversity of APIs

The API landscape is diverse. In addition to web service APIs (which include REST), there are web socket APIs, hardware APIs, and more. Despite the wide variety, there are mostly just two main types of APIs most technical writers interact with:

- Native library APIs (such as APIs for Java, C++, and .NET)
- REST APIs (which are a type of web API)

With native library APIs, you deliver a library of classes or functions to users, and they incorporate this library into their projects. They can then call those classes or functions directly in their code.

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.

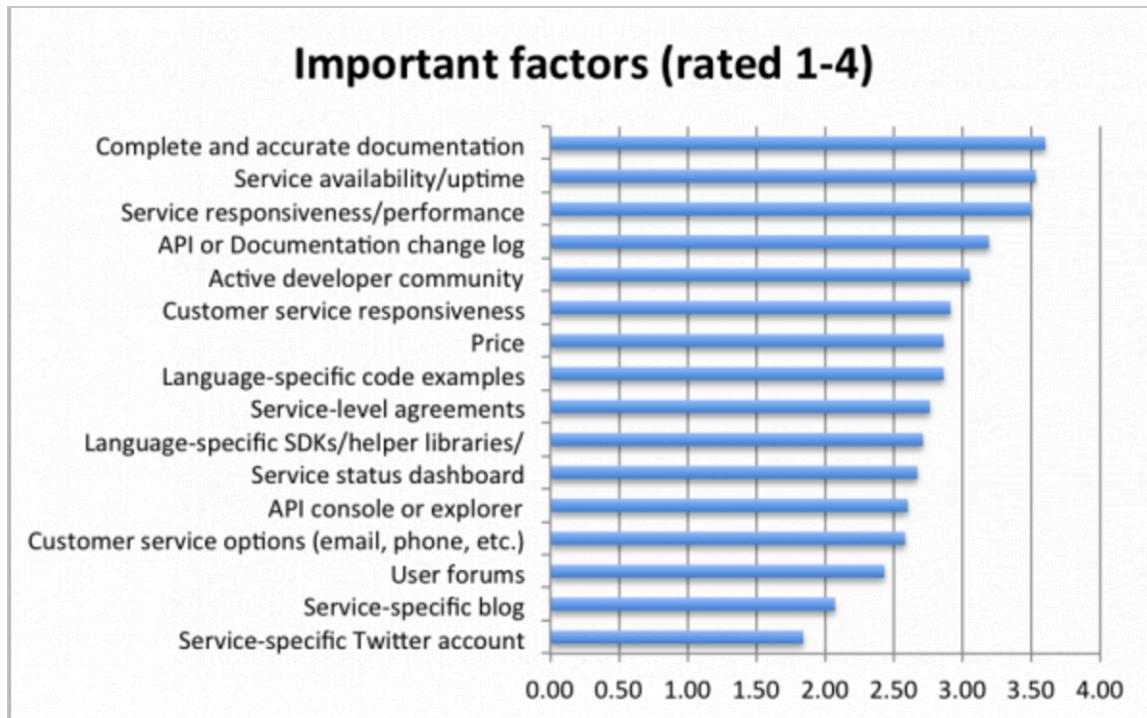
REST APIs follow the same protocol as the web. When you open a browser and type a website URL (such as <https://idratherbewriting.com>), you're actually making a GET request for a resource on a server. The server responds with the content and the browser makes the content visible.

This course focuses mostly on REST APIs because REST APIs are more popular and in demand, and they're also more accessible to technical writers. You don't need to know programming to document REST APIs. And REST is becoming the most common type of API anyway. (Even so, I also cover native library APIs in a [later section \(page 0\)](#).)

Programmableweb API survey rates doc #1 factor in APIs

Before we get into the nuts and bolts of documenting REST APIs, let me provide some context about the popularity of the REST API documentation market in general.

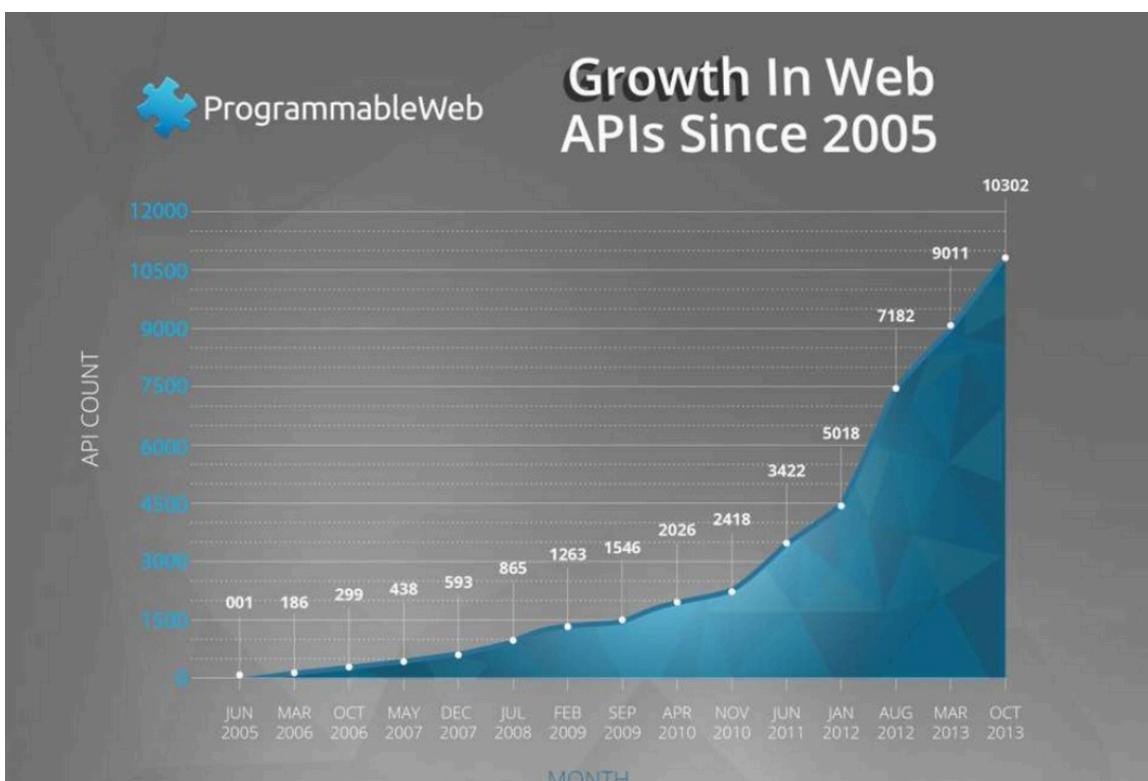
In a [2013 survey by Programmableweb.com](#) (which is a site that tracks and lists web APIs), about 250 developers were asked to rank the most important factors in an API. "Complete and accurate documentation" ranked as #1.



John Musser, one of the founders of Programmableweb.com, 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.”



If REST APIs were an uncommon software product, it wouldn't be that big of a deal. But actually, REST APIs are taking off in a huge way. Through the PEW Research Center, Programmableweb.com has charted and tracked the prevalence of 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 a 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.

Because REST APIs are a style not a standard, docs are essential

REST APIs are a bit different from the SOAP APIs that were popular some years ago. SOAP APIs (service-oriented architecture protocol) 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 is what fuels the need for accurate and clear documentation. As long as there is variety with REST APIs, there will be a strong need for technical writers to provide documentation for these APIs.

The web is becoming an interwoven mashup of APIs

Another reason why REST APIs are taking off is because the web itself is evolving into a conglomeration of APIs. Instead of massive, do-it-all systems, web sites are pulling in the services they need through APIs.

For example, rather than building your own search to power your website, you might use Swiftype instead and leverage their service through the [Swiftype API](#). Rather than building your own payment gateway, you might integrate [Stripe and its API](#). Rather than building your own login system, you might use [UserApp and its API](#). Rather than building your own e-commerce system, you might use [Snipcart and its 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.

The screenshot shows a dark-themed dashboard titled "Services". Below it, under the heading "Newsletters", there is a sub-section titled "Third-party services for Jekyll websites." This section contains five cards, each representing a different service: AWeber (blue icon), Campaign Monitor (white icon with a blue envelope), MailChimp (white icon with a cartoon monkey wearing a cap), MailerLite (white icon with the text "mailer lite"), and Sendicate (orange icon with a white stylized letter "S"). Below this, another section titled "Analytics" is partially visible.

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. In sum, the web is becoming an interwoven mashup of many different services from APIs interacting with each other.

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:

[Find Jobs](#) [Find Resumes](#) [Employers / Post Job](#)

 one search. all jobs.

what:
job title, keywords or company

where:
city, state, or zip

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.

You'll 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 you've 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, you'll 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

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 looks like:

The screenshot shows the Dropbox API v2 documentation website. At the top right, there's a user profile for 'Tom Johnson' with a dropdown arrow. To the left of the profile is a blue bell icon. The main title 'Build your app on the Dropbox platform' is centered above a sub-headline 'A powerful API for apps that work with files.' On the left side, there's a sidebar with navigation links: 'API v2', 'My apps', 'API Explorer', 'Documentation' (which is expanded to show 'HTTP', '.NET', 'Java', 'JavaScript', 'Python', 'Swift', 'Objective-C', 'Community SDKs', 'References', 'Authentication types', 'Branding guide', 'Content hash', and 'Data ingress guide'). The main content area features a central illustration of four people working on computers, each with a gear icon above them, set against a background of clouds and stars. Below the illustration are three call-to-action boxes: 'Read our docs' (describing docs organized by language from .NET to Swift), 'Create your app' (describing the App Console), and 'Test your ideas' (describing the API Explorer). The entire page has a clean, modern design with a white background and light blue accents.

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 \(page 439\)](#).)

API doc is a new world for most tech writers

API documentation is often a new world to technical writers. Many of the components may be new. 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. Additionally, the documentation content itself is often complex and requires familiarity with development concepts and processes.

Learning materials about API doc are scarce

Realizing there was a need for more information, in 2014 I guest-edited a special issue of Intercom dedicated to API documentation.



You can read this issue for free at <http://bit.ly/stcintercomapiissue>.

This issue was a good start, but many technical writers asked for more training. The Silicon Valley STC chapter held a couple of workshops dedicated to APIs. Both workshops sold out quickly (with 60 participants in the first, and 100 participants in the second). API documentation is particularly hot in the San Francisco Bay area, where many companies have REST APIs requiring documentation.

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 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 is one of those deep dive moments. Here we'll dive into what a REST API is, comparing it to other types of APIs like SOAP. REST APIs have common characteristics but no definitive protocols like their predecessors.

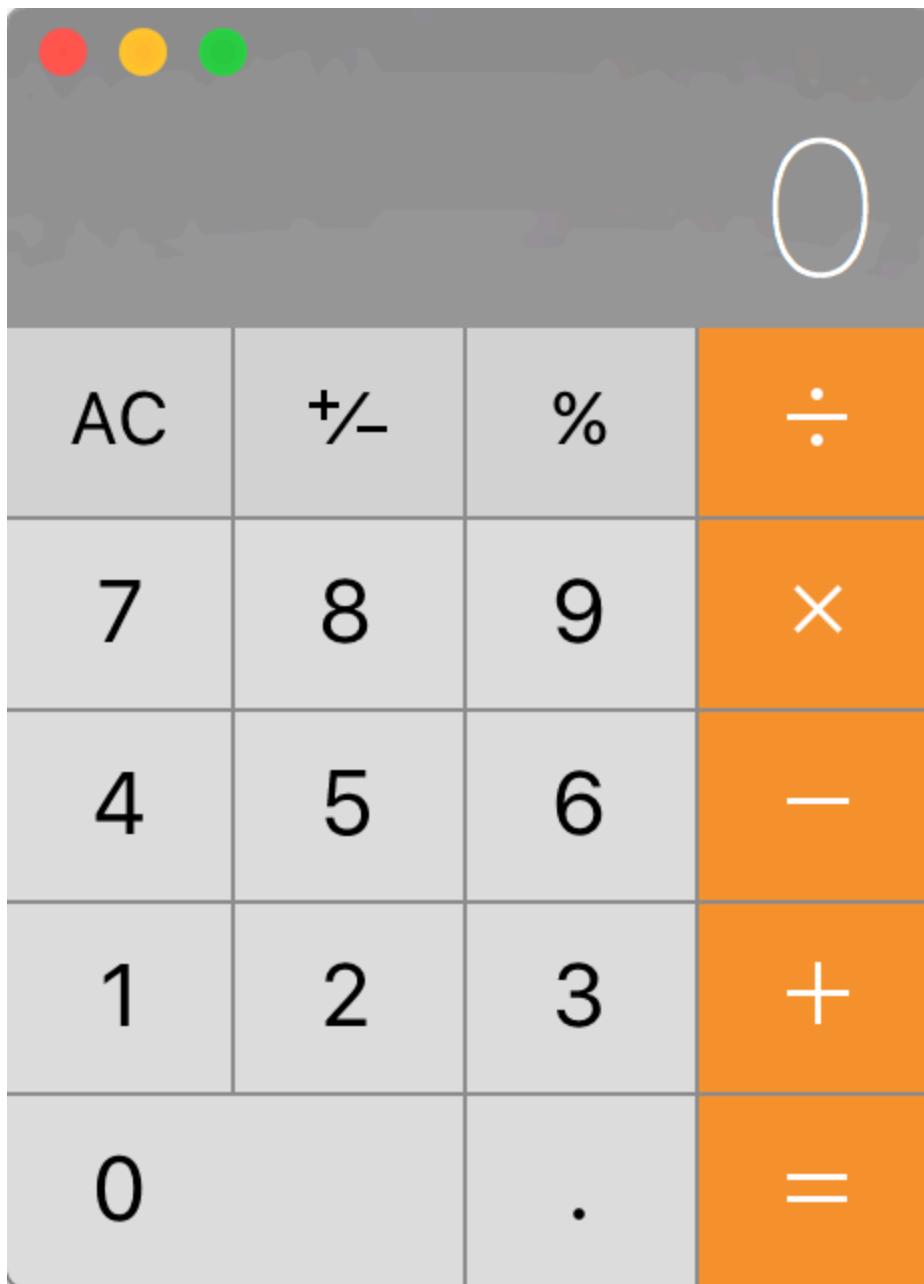
What is an API?

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.



spinning gears by Brent 2.0

Jim Bisso, an experienced API technical writer in the Silicon Valley area, describes APIs by using the analogy of your computer's calculator. When you press buttons, functions underneath are interacting with other components to get information. Once the information is returned, the calculator presents the data back to the GUI.



APIs often work in similar ways. When you push a button in an interface, functions underneath get triggered to go and retrieve information. But instead of retrieving information from within the same system, web APIs call remote services on the web to get their information.

Ultimately, developers use API calls behind the scenes to pull information into their apps. A button on a GUI may be internally wired to make calls to an external service. For example, the embedded Twitter or Facebook buttons that interact with social networks, or embedded YouTube videos that pull a video in from youtube.com, are powered by web APIs underneath.

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 makes the request for a resource, and the web service provides the response).

All APIs that use HTTP protocol as the transport format for requests and responses can be classified as “web services.”

Language agnostic

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.

This 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 the API is built with Java, Ruby, Python, or some other language. The requests are made over HTTP, and the responses are returned through HTTP.

Each programming language that makes the request will have a different way of submitting a web request and parsing the response, but the way requests for each language are made and the responses retrieved are common operations for the programming language and not usually specific to a particular REST API.

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 uses standardized protocols and WSDL files

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 Challenges](#).

Problems with SOAP and XML: Too heavy, slow

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 5 years, SOAP has largely been replaced by REST, especially for APIs on the open web.

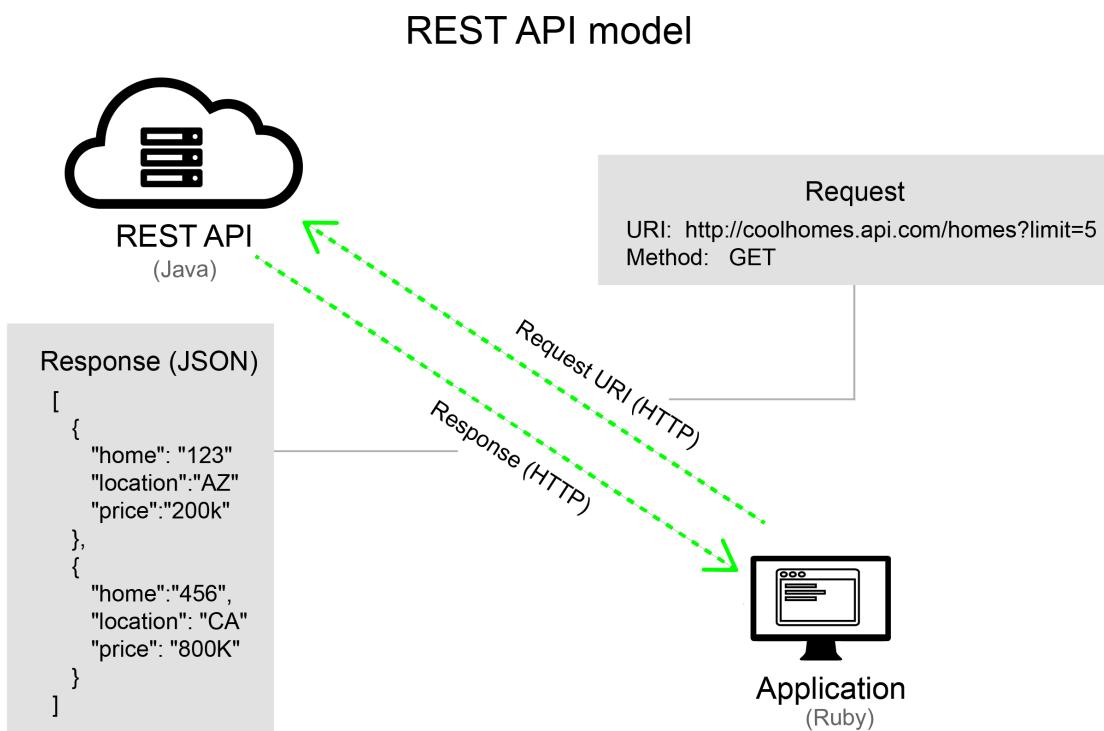
REST is a style, not a standard

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.)

Requests and Responses

The following diagram shows the 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.

Any message format can be used with REST

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. This is because JSON 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 of 5 instances of the resource.

Sample URLs for a REST API

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 out 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}
```

This 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.

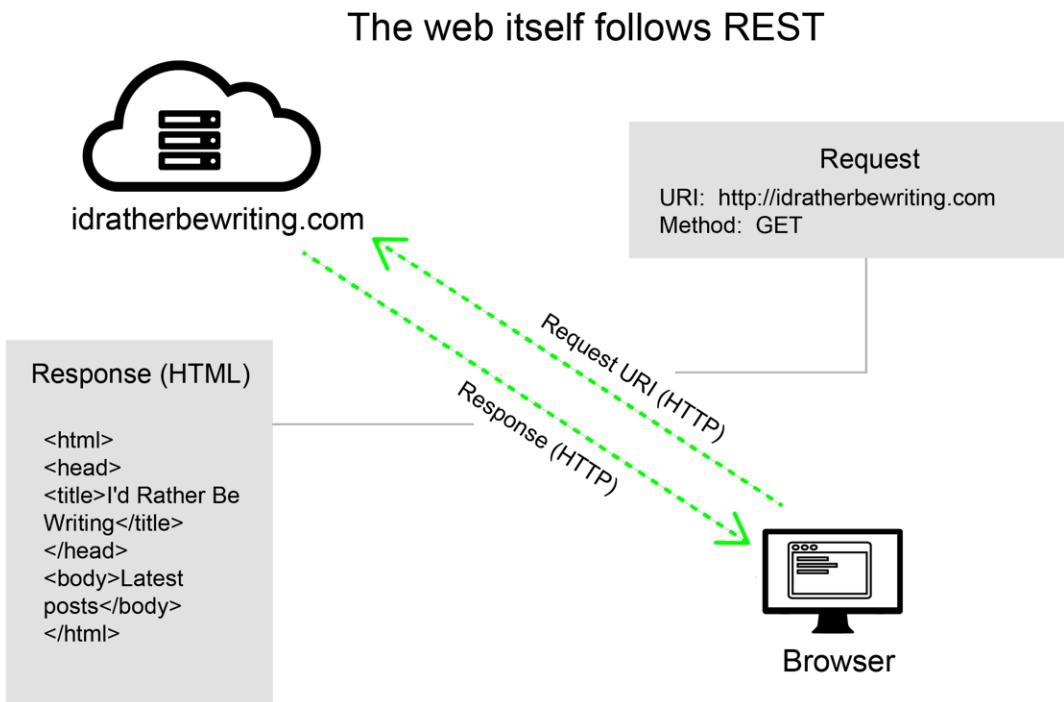
The relationship between resources and methods is often described in terms of “nouns” and “verbs.” The resource is the noun because it is an object or thing. The verb is what you're doing with that noun. Combining nouns with verbs is how you form the language in REST.

We'll explore endpoints in much more depth in the sections to come (for example, in the [API reference tutorial \(page 82\)](#) 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 “URIs” and “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 (`http://`) 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.



You can see this response in [curl \(page 45\)](#) if you open a terminal prompt and type `curl https://idratherbewriting.com`. (This assumes you have curl installed.)

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 in order to increase the performance. If the browser’s cache already contains the information asked for in the request, the browser can simply 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. This 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 a number of more 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 terms 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, they're rarely used since the WADL files 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.)

In order to understand how to interact with a REST API, you have to *read the documentation* for the API. Hooray! This makes the technical writer's role extremely important with REST APIs.

Some formal specifications — for example, such [OpenAPI \(page 143\)](#), have been developed to describe REST APIs. When you describe your API using the OpenAPI specification, tools that can read those specifications (such as [Swagger UI \(page 214\)](#)) 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 \(page 214\)](#) that read the specification documents are usually interactive (featuring API Consoles or API Explorers) and allow you to try out REST calls and see responses directly in the documentation.

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 \(page 277\)](#), details about workflows and interdependencies between endpoints, and so on. The Swagger or RAML output usually contains reference documentation only, which typically only accounts for 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 in order to understand how to interact with a REST API. As you explore REST APIs, you will find that they differ greatly from one to another (especially the format and display of their documentation sites), 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

Activity: Identify your goals

In this course, you'll do a variety of activities to build your skills, develop content, and gain experience. In this activity, you'll identify your goals so that you can better align your efforts. This activity also helps the instructor understand why attendees are taking the course and what they hope to get out of it.

Activity 1a: Identify your goals with API documentation

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. Identify your goals here and make sure they align with this course.

Answer 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.

Using a REST API like a developer

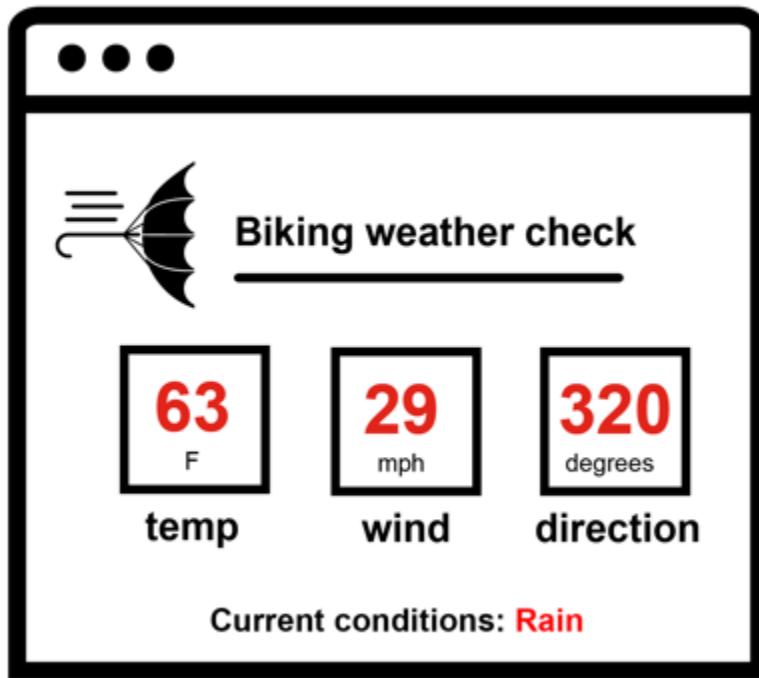
Scenario for using a weather API	32
Get authorization keys.....	37
Submit requests through Postman.....	39
curl intro and installation	45
Make a curl call.....	47
Understand curl more.....	49
Use methods with curl.....	53
Analyze the JSON response.....	59
Inspect the JSON from the response payload	63
Access and print a specific JSON value	69
Dive into dot notation	73

Scenario for using a weather API

Enough with the abstract concepts. 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 a weather conditions feature to your site. Your site is for bicyclists. You want to allow users who come to your site to see what the wind and temperature conditions are for biking. You want something like this:



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/learnapidoc/assets/files/wind-openweathermap.html. It's not necessarily styled the same as the mockup, but it answers the question, "What's the wind and temperature?"

Click the button to see wind and temperature details. When you request this data, an API goes out to the [OpenWeatherMap API service](#), retrieves the information, and displays it to you.

Check wind conditions

Wind conditions for Santa Clara

Temperature:

Wind speed:

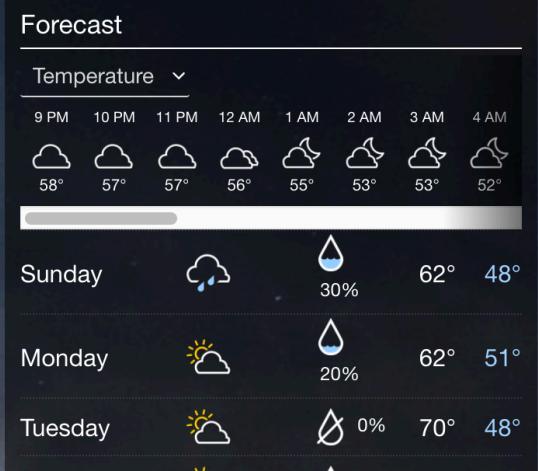
Wind direction:

Current conditions:

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

 Cloudy
↑ 62° ↓ 48°

58° F C

**Forecast**
Temperature: 58°
9 PM 10 PM 11 PM 12 AM 1 AM 2 AM 3 AM 4 AM
Cloudy 58° 57° 57° 56° 55° 53° 53° 52°

**Details**
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.

Explore the OpenWeatherMap API

Although there are [many good weather API options for developers](#), I decided to use the [OpenWeatherMap](#) because their service is easy to use, free, and stable.

Activity 2a: Get familiar with the OpenWeatherMap API

Explore the basic sections in the [OpenWeatherMap API](#):

1. Go to <https://openweathermap.org> and click **API** in the top navigation bar.
2. Explore the information available in the **Current weather data** by clicking the **API Doc** button in that section.

As you explore the site, 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.

3. 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? (Hint: Look at one of the sample API responses by clicking the **Example** link.)
 - 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 int 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.
3. Click **Weather API**.
4. Click **Data Endpoints**.
5. Click **Reference** and then click **Endpoints**. (Or just go [here](#) directly.)

The screenshot shows the Aeris Weather API documentation. At the top, there's a navigation bar with a sun icon (77°), a search bar ('Weather for...'), and links for 'LOG IN', 'SIGN UP', 'PRODUCTS', 'DOCUMENTATION', 'INDUSTRIES', 'BLOG', and 'CONTACT'. Below the navigation is a breadcrumb trail: 'HELP CENTER / DOCS / WEATHER API / REFERENCE / ENDPOINTS'. A search bar and a 'SEARCH' button are also present. On the left, a sidebar for the 'Weather API' includes 'Get Started', 'Data Endpoints' (which is highlighted in blue), 'Free Trial', 'Getting Started', 'Reference', and 'Downloads'. The main content area is titled 'API Endpoints'. It explains what endpoints are and lists supported endpoints. A note says, 'To query multiple endpoints with a single weather API request review the [Batch Requests](#) feature.' Below this is a table with columns 'Endpoint', 'Description', and 'Actions'. A filter bar at the top of the table allows filtering by 'Subscription' type: 'All' (selected), 'Developer', 'Basic', and 'Premium'.

6. In the list of endpoints, click **observations**.
7. 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/learnapidoc/assets/files/wind-aeris.html.

For this scenario, there are dozens of different weather APIs we could use. As you create your API documentation, think about whether your users have the same decisions to make. Are there several other APIs that developers can choose from for the same information? What will make your API stand out more? Although you probably can't define your product roadmap, 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:

- [Dark Sky API](#)
- [Accuweather API](#)
- [Weather Underground API](#)
- [Weatherbit API](#)

Each weather API has a totally 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 presentations vary, terminology with APIs varies as well.

As I mentioned in [REST is a style, not a standard \(page 26\)](#), REST APIs are an architectural style following common characteristics and principles; they don't all follow the same standard or specification. Users really have to read the documentation to understand how to use the API.

Answer some questions about the APIs

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 \(page 277\)](#) later, we need to get some API keys now in order to make requests to the 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

In order 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.

If you want to borrow my API keys, you can access them [here](#). I sometimes find that workshop participants get hung up in trying to acquire simple API keys, so I'd like to remove that roadblock from you if you're running into issues.

Get authorization keys

In order to work with the APIs in the upcoming activities, you'll need to get some API keys.

Activity 2a: Get an OpenWeatherMap API key

To make calls to the OpenWeatherMap API, you'll need authorization keys. To get the keys:

1. On <https://openweathermap.org/>, click **Sign Up** in the top nav bar and create an account.
2. After you sign up, sign in and find your default API key from the developer dashboard. It's under the **API Keys** tab.
3. Copy the key into a place you can easily find it.

(If your API key doesn't work or is inactive, you can use this one: [fd4698c940c6d1da602a70ac34f0b147](#).)

Get the Aeris Weather API secret and ID

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

1. Go to <http://www.aerisweather.com> and click **Sign Up** in the upper-right corner.
2. Under **Developer**, click **TRY FOR FREE**. (The free version limits the number of requests per day and per minute you can make.)
3. Enter a username, email, and password, and then click **SIGN UP FOR FREE** to create an Aeris account. Then sign in.
4. After you sign up for an account, click **Account** in the upper-right corner.
5. Click **Apps** (on the second navigation row, to the right of "Usage"), and then click **New Application**.

The screenshot shows the Aeris Weather API dashboard. At the top, there's a header with weather information (57°) and a search bar. Below the header is the Aeris Weather logo and a navigation menu with links like ACCOUNT, LOG OUT, SIGN UP, DASHBOARD, USAGE, APPS (which is underlined), MAP BUILDER, LEGEND GENERATOR, ADD SUBSCRIPTION, DOCS, SUPPORT, and BILLING. Under the APPS link, there's a PROFILE link. The main content area is titled "Registered Apps". It contains a list of registered applications, starting with "API testing". For this application, it shows the ID as "ByruDorHEne2JB64BhP1k" and the SECRET as "uBDNO535gYHULH8mqx3skZmU13EV4nf4GvB6jhY". Below the application details are edit and delete icons. At the bottom of the "Registered Apps" section, there's a "NEW APPLICATION" button with a red arrow pointing to it.

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 your app registers, 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 \(page 277\)](#) in more detail.

Text editor tips

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). Many developers use different text editors. Here are a few choices:

- [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. (Personally, I use Sublime Text when I'm working with 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 just mean there's a graphical user interface with boxes and buttons for you to click.) You can also use [curl \(page 45\)](#) (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.

Common GUI clients

Some popular GUI clients include the following:

- [Postman](#)
- [Paw](#)
- [Advanced REST Client](#) (Chrome browser extension)
- [REST Console](#)

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 some kind of action. In this course, I'm following more of an "act-first-then-understand" type of 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 a endpoint, hang in there. When we deep dive into these points later, these concepts will be a lot clearer.

Activity 2c: Make requests with Postman

Make a request

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 <http://www.getpostman.com>.
2. Start the Postman app (and enjoy the quirky messages that appear while the app loads).
3. If necessary, select **GET** for the method. (This is the default.)
4. Insert the following endpoint into the box next to GET: <https://api.openweathermap.org/data/2.5/weather>.
5. Click the **Params** button (to the right of the box where you inserted the endpoint) and then add the following three parameters. You'll need to insert the key into the Param's **key** field and the value into the Param's **value** field:

- key: `zip` / value: `95050`
- key: `appid` / value: `APIKEY`
- key: `units` / value: `imperial`

Customize the `zip` and `appid` values to your own zip code and API key.

Your Postman UI should look like this:

When you add these parameters, they will dynamically be added 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&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial> (but with different query string values). Query string parameters appear after the question mark `?` symbol and are separated ampersands `&`. The order of query string parameters doesn't matter.

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.)

6. Click **Send**.

The response appears in the lower pane. For example:

```

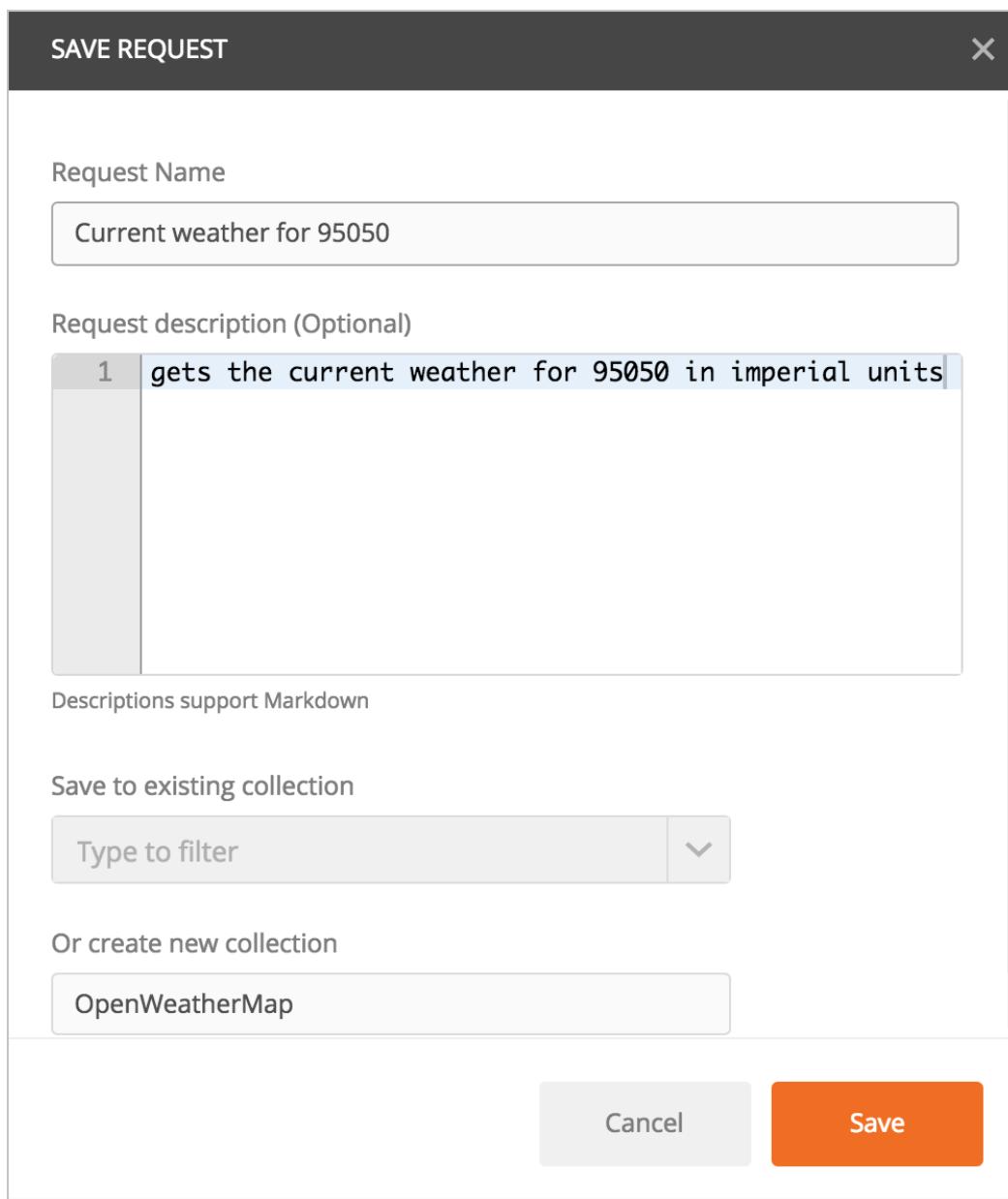
1  {
2   "coord": {
3     "lon": -121.96,
4     "lat": 37.35
5   },
6   "weather": [
7     {
8       "id": 701,
9       "main": "Mist",
10      "description": "mist",
11      "icon": "50d"
12    }
13  ],
14  "base": "stations",
15  "main": {
16    "temp": 66.2,
17    "pressure": 1017,
18    "humidity": 40,
19    "temp_min": 53.6,
20    "temp_max": 77
21  }

```

Save the request

1. In Postman, click the **Save** button (next to Send).
2. In the Save Request dialog box, in the **Request Name** box at the top, type a friendly name for the request, such as “Current weather for 95050.”
3. Scroll down a bit in the Save Request dialog box. Next to **Or create new collection**, create a new collection by typing the collection name in the box, such as “OpenWeatherMap.” Collections are simply groups to organize your saved requests.

Your Postman collection should look something like this:



4. Click **Save**.

Saved endpoints appear in the left side pane under Collections. (If you don't see the Collections pane, click the **Show/Hide Sidebar** button in the upper-left corner to expand it.)



Make a request for the OpenWeatherMap 5 day forecast

Now instead of getting the current weather, let's use another 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&units=imperial
```

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

Same request but in Paw instead of Postman

Although Postman is a popular REST client, you can also use others, such as Paw. For the sake of variety, the following image shows the same current weather request made in [Paw \(for Mac\)](#):

The screenshot shows the Paw application window with the following details:

- Title Bar:** Untitled Paw Document 3 — Edited
- Request Tab:** GET /data/2.5/weather
- URL Params:**
 - zip: 95050,us
 - appid: fd4698c940c6d1da602a70ac34f0b147
 - units: imperial
- Response:**
 - Status: 200 OK
 - Headers: JSON
 - Raw Response (JSON):


```
GET ...d1da602a70ac34f0b147&units=imperial [200 OK]
{
  "coord": {
    "lon": -121.96,
    "lat": 37.35
  },
  "weather": [
    {
      "id": 701,
      "main": "Mist",
      "description": "mist",
      "icon": "50d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 66.38,
    "pressure": 1018,
    "humidity": 87,
    "temp_min": 53.6,
    "temp_max": 77,
    "visibility": 9656
  },
  "wind": {
    "speed": 10.29,
    "deg": 320
  },
  "clouds": {}
}
```

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. This 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 \(page 32\)](#). 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 just paste the requests directly into the URL request box in Postman, 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 \(page 37\)](#)). If you don't have a client ID or secret, you can use my keys [here](#).

Get the weather forecast for your area using the [observations endpoint](#):

```
http://api.aerisapi.com/observations/Santa+Clara,CA?client_id=CLIENTID&client_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&client_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&client_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_secret=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 APIs are passed in the query string rather than the header). Use the [JSON Formatter plugin for Chrome](#) to automatically format the JSON response.

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.

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 then 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.

curl intro and installation

While [Postman \(page 39\)](#) is convenient, it's hard 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 explain how to make the calls using 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.

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 + space bar** to open Spotlight, 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 ldap
s pop3 pop3s rtsp smb smbs smtp smtps telnet tftp Features: AsynchDN
S IPv6 Largefile GSS-API Kerberos SPNEGO NTLM NTLM_WB SSL libz UnixSo
ckets
```

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 select the **With Administrator Privileges (free)** installer.

After curl is installed, 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: AsyncDNS 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=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

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**.

If you're on Windows 8.1 and you encounter an error that says, "The program can't start because MSVCR100.dll is missing from your computer," see [this article](#) and install the suggested package.

Notes about using curl with Windows

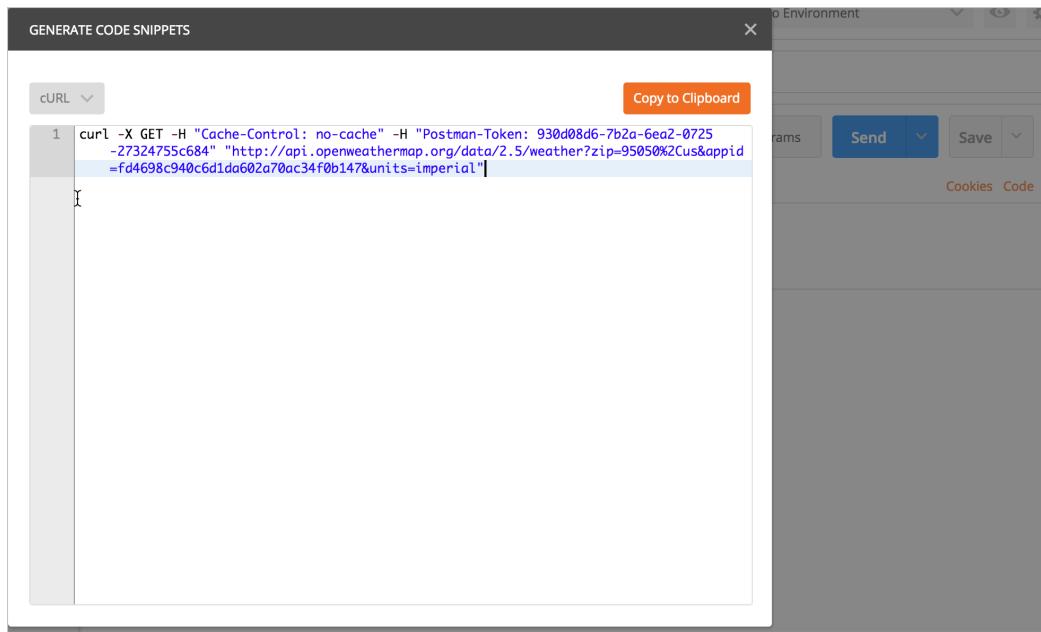
- Use double quotes in the Windows command line. (Windows doesn't support single quotes.)
- Don't use backslashes (\) to separate lines. (This is for readability only and doesn't affect the call on Macs.)
- By adding `-k` in the curl command, you can bypass curl's security certificate, which may or may 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 \(page 45\)](#) first.

Activity 2d: Make the OpenWeatherAPI request using curl

1. Assuming you completed the exercises in the Postman tutorial, go back into Postman.
2. On any call you've configured, below the Save button in Postman, click the **Code** link, then select **cURL** from the drop-down select, and click **Copy to Clipboard**.



(The official name is "cURL" but most people just write it as "curl.")

The Postman code for the OpenWeatherMap weather request looks like this in curl format:

```
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=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

Postman adds some of its own header information (designated with `-H`). You can optionally remove these extra header tags (including them won't hurt anything). Here's the curl call with the `-H` content removed:

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

3. Curl is available on Mac by default. If you're on Windows and you haven't installed curl, follow the instructions for installing curl [here](#). (Note: Choose one of the "free" versions).

4. Go to your terminal:

- If you’re on Windows, go to **Start** and type **cmd** to open up the command line. (If you’re on Windows 8, see [these instructions for accessing the commandline](#).) Right-click and then select **Paste** to insert the call.
- If you’re on a Mac, either open **iTerm** or Terminal (by pressing **Cmd + space bar** and typing **Terminal**). Paste the request you have in your text editor into the command line and then press the **Enter** key.

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

```
{"coord":{"lon":-121.96,"lat":37.35},"weather":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}],"base":"stations","main":{"temp":65.59,"pressure":1014,"humidity":46,"temp_min":60.8,"temp_max":69.8},"visibility":16093,"wind":{"speed":4.7,"deg":270},"clouds":{"all":20}, "dt":1522608960,"sys":{"type":1,"id":479,"message":0.1642,"country":"US","sunrise":1522590719,"sunset":1522636280}, "id":420006397,"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 on a Mac (with [Python installed](#)) you can add `| python -m json.tool` at the end of your cURL request to minify the JSON in the response. See [this Stack Overflow thread](#) for details.

5. Now make a similar curl request for the 5 day forecast request that you also have in Postman.

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 request require you to submit JSON in the request body, you might run into issues with single versus double quotes. Some API endpoints (usually for POST methods) require you to submit content in the body of the message request. The body content is formatted in JSON. 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 \(page 28\)](#), one reason REST APIs are so familiar is because REST follows the same model as the web. 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
```

You should see all the code behind the site example.com. The browser's job is to make that code visually readable. curl shows you what you're really 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 header that contains information about the request. The response also contains a 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:

```
HTTP/1.1 200 OK
Cache-Control: max-age=604800
Content-Type: text/html
Date: Sat, 07 Jul 2018 23:25:03 GMT
Etag: "1541025663+ident"
Expires: Sat, 14 Jul 2018 23:25:02 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (sjc/4FB8)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1270
```

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 (that is, 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](#) if you look on the Network tab.

3. Now let's specify the method. The GET method used by default, but we'll make it explicit here:

```
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 \(page 47\)](#):

```
curl -X GET -H "Cache-Control: no-cache" -H "Postman-Token: 930d08d6-7b2a-6e  
a2-0725-27324755c684" "https://api.openweathermap.org/data/2.5/weather?zip=9  
5050&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

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`) and app ID (`appid`) and units (`units`) parameters were passed to the endpoint using "query strings." The `?` appended to the URL is the query string where the parameters are passed to the endpoint:

```
?zip=95050&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial
```

After the query string, each parameter is concatenated with other parameters through the `&` symbol. The order of the parameters doesn't matter. The order only matters if the parameters are part of the URL path itself (not listed after the query string).

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
<code>-i</code> or <code>--include</code>	Includes the response headers in the response.	<code>curl -i http://www.example.com</code>
<code>-d</code> or <code>--data</code>	Includes data to post to the URL. The data needs to be url encoded . Data can also be passed in the request body.	<code>curl -d "data-to-post" http://www.example.com</code>
<code>-H</code> or <code>--header</code>	Submits the request header to the resource. This is very common with REST API requests because the authorization is usually included in the header.	<code>curl -H "key:12345" http://www.example.com</code>

curl command	Description	Example
<code>-X POST</code>	Specifies the HTTP method to use with the request (in this example, <code>POST</code>). If you use <code>-d</code> in the request, curl automatically specifies a POST method. With GET requests, including the HTTP method is optional, because <code>GET</code> is the default method used.	<code>curl -X POST -d "resource-to-update" http://www.example.com</code>
<code>@filename</code>	Loads content from a file.	<code>curl -X POST -d @mypet.json http://www.example.com</code>

See the [curl documentation](#) for a comprehensive list of curl commands you can use.

Example curl command

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

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

We could also format this 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.

To learn more about curl with REST documentation, see [REST-esting with curl](#).

Use methods with curl

Our [sample weather API \(page 32\)](#) doesn't allow you to use anything but a GET method, so for this exercise, we'll use the [petstore API from Swagger](#), but without actually using the Swagger UI (which is something we'll [explore later \(page 143\)](#)). 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. This is often how you configure a service. The list of JSON key-value pairs that the API accepts is called the "Model" in the Petstore API.

1. Insert the following into a file called mypet.json. 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 (whole number). 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 this directory: `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 by typing change directory: `cd ..`. Move down 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 a PC, just look at the prompt path to see your current directory. Then move up by typing `cd ..`. Move down 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:

```
curl -X POST --header "Content-Type: application/json" --header "Accept: application/json" -d @mypet.json "http://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"}
```

Feel free to run this same request a few times more. REST APIs are "idempotent," which means that running the same request more than once won't end up duplicating the results (you just create one pet here, not multiple pets). 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.

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` with the same curl content to update the pet's name:

```
curl -X PUT --header "Content-Type: application/json" --header "Accept: application/json" -d @mypet.json "http://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" "http://petstore.swagger.io/v2/pet/51231236"
```

The response contains your pet 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" "http://petstore.swagger.io/v2/pet/5123123"
```

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

```
curl -X GET --header "Accept: application/json" "http://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.

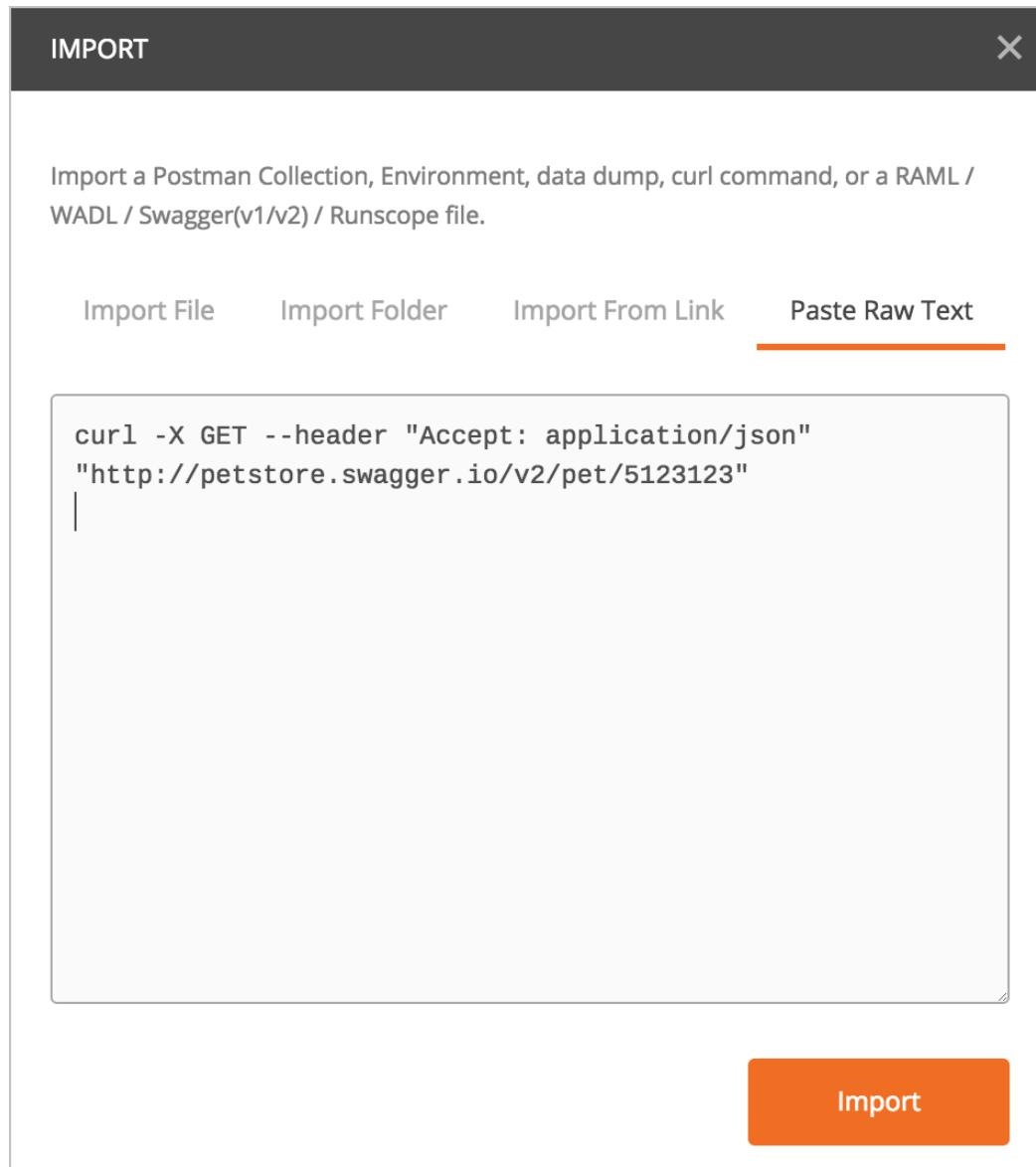
Import curl into Postman

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" "http://petstore.swagger.io/v2/pet/5123123"
```

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

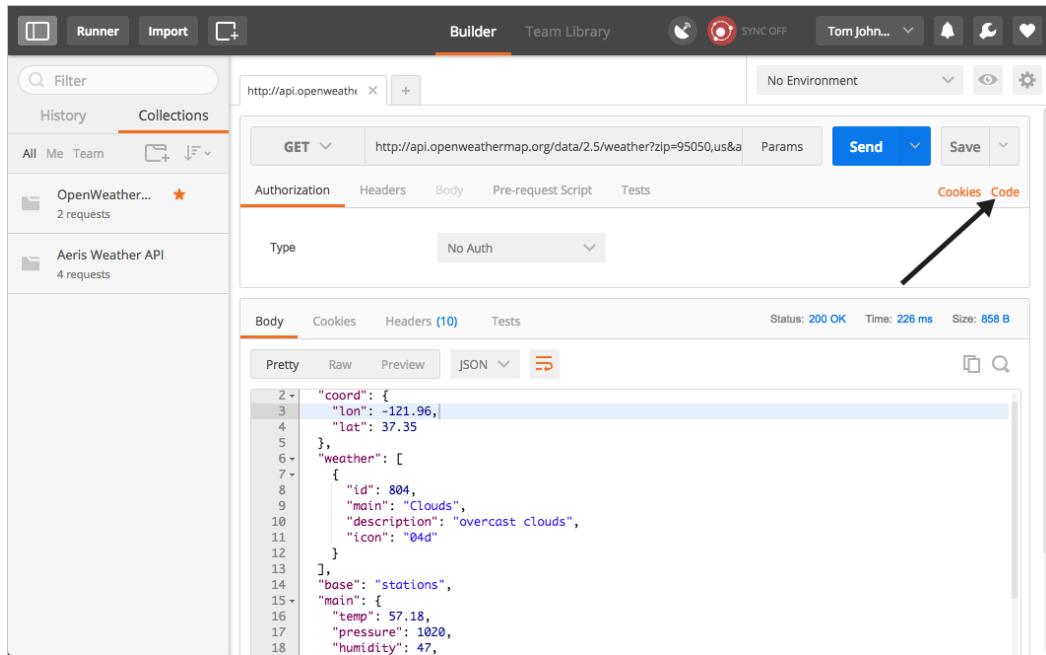


3. Click **Import**.
4. Close the dialog box.
5. Click **Send**.

Export Postman to curl

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

1. In Postman, click the **Code** button (it's right below Save).



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

```
curl -X GET -H "Accept: application/json" -H "Cache-Control: no-cache" -H "Postman-Token: e40c8069-21db-916e-9a94-0b9a42b39e1b" 'http://petstore.swagger.io/v2/pet/5123123'
```

You can see that Postman adds some extra header information (`-H "Cache-Control: no-cache"` `-H "Postman-Token: e40c8069-21db-916e-9a94-0b9a42b39e1b"`) into the request. This extra header information is unnecessary and can be removed.

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 formatting.

JSON response from OpenWeatherMap weather endpoint

JSON stands for JavaScript Object Notation. It's the most common way REST APIs return information. Through JavaScript, you can easily parse through the JSON and display it on a web page.

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 toolset that powers most web pages.

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  
}
```

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 surrounded with curly braces `{ }` and contain key-value pairs. Sometimes those values are arrays. Arrays are lists and are surrounded with 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.

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 2e: Make the request on a page with AJAX

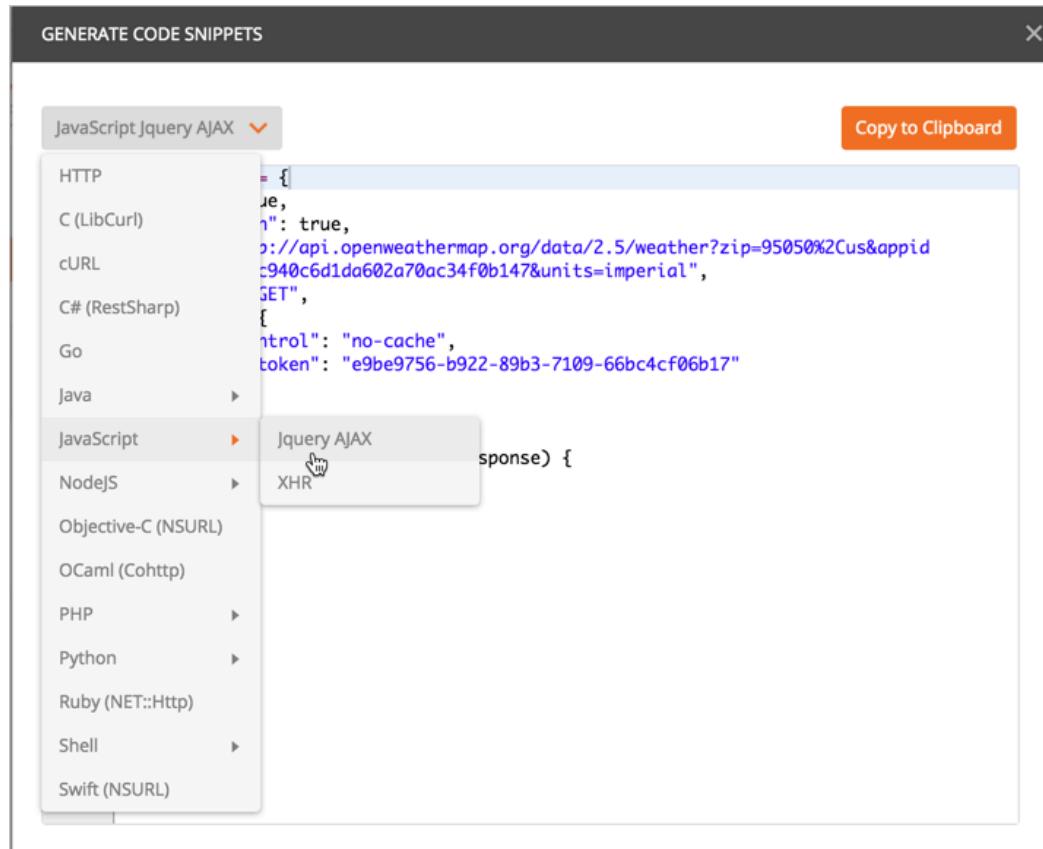
For this activity, you'll use JavaScript to display the API response on a web page. You'll use some auto-generated jQuery code from Postman to create the AJAX request.

1. In a text editor (such as Sublime Text), create a new HTML file and paste in the following boilerplate template (which contains basic HTML tags and a reference to jQuery):

```
<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>
</head>
<body>
    <h1>Sample page</h2>

</body>
</html>
```

2. Save your file (anywhere convenient) with a name such as weather.html.
3. Open Postman and go to the Current weather data (`weather`) endpoint that you configured earlier (page 39).
4. Click the **Code** link (below the Save button), and then select **JavaScript > jQuery AJAX**.



The AJAX code should look as follows:

```
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://api.openweathermap.org/data/2.5/weather?zip=95050&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial",
  "method": "GET",
  "headers": {
    "cache-control": "no-cache",
    "postman-token": "e9be9756-b922-89b3-7109-66bc4cf06b17"
  }
}

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

5. Click **Copy to Clipboard** to copy the code sample.
6. In the same template you started building in step 1, add a pair of `<script></script>` tags below the jQuery reference, and then insert the Postman code inside your `script` tags.
7. In the jQuery code, remove the `headers` object that Postman inserts:

```
"headers": {  
    "cache-control": "no-cache",  
    "postman-token": "e9be9756-b922-89b3-7109-66bc4cf06b17"  
}
```

8. And also remove the comma after `"method": "GET"`.

Your final code should look like this:

```
<!DOCTYPE html>  
<html>  
  <meta charset="UTF-8">  
  <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=fd4698c940c6d1da602a70ac34f0b147&units=imperial",  
        "method": "GET"  
      }  
  
      $.ajax(settings).done(function (response) {  
        console.log(response);  
      });  
    </script>  
  </head>  
  <body>  
    <h1>Sample Page</h1>  
  </body>  
</html>
```

You can view the file here: idratherbewriting.com/learnapidoc/assets/files/weather-plain.html

9. Start **Chrome** and open the JavaScript Console by going to **View > Developer > JavaScript Console**.
10. In Chrome, go to **File > Open File** and select the **weather.html** file.

The page body will be blank, but 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:

The screenshot shows the browser's developer tools with the 'Console' tab selected. It displays a JSON object representing weather data. The object has properties like 'coord', 'weather', 'base', 'main', 'visibility', 'base', 'clouds', 'cod', 'coord', 'dt', 'id', 'main', 'name', 'sys', 'weather', 'wind'. The 'name' property is expanded to show its value is 'Santa Clara'. The 'weather' property is shown as an array of two objects. The 'wind' property is also expanded.

```

{
  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}
}
  
```

This information is now available for you to integrate into your page.

- To use a value from the JSON response, add the following inside the `ajax` function:

```

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

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

});
  
```

- In the page body (inside the `body` tags), add the following div tag:

```

<body>
  <h1>Sample page</h1>
  <div id="windSpeed">Wind speed: </div>
</body>
  
```

- Refresh the page and you will see the wind speed printed to the page. (Here's [an example](#).)

The following sections will explain this AJAX code a bit more.

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 it. To recap, here's the `ajax` script:

```
<script>
  var settings = {
    "async": true,
    "crossDomain": true,
    "url": "https://api.openweathermap.org/data/2.5/weather?zip=95050&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial",
    "method": "GET"
  }

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

If you're working with JavaScript and APIs, probably the most useful method to know for showing code samples is the `ajax` method from `jQuery`. In brief, 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.

This is likely a bit fuzzy right now, but it will become more clear with an example in the next section.

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 \(page 298\)](#).

Logging responses to the console

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

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

Logging responses to the console is one of the most useful ways to test whether an API response is working (it's also helpful for debugging or troubleshooting your code). The console collapses each object inside its own expandable section. This allows you to inspect the payload.

You can add other information to the console log message. To preface the log message with a string, add something like this:

```
console.log("Here's the response: " + response);
```

Strings are always enclosed inside quotation marks, and you use the plus sign `+` to concatenate strings with JavaScript variables, like `response`.

Customizing log messages is helpful if you're logging various things to the console and need to flag them with an identifier.

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 \(page 69\)](#), 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 \(page 63\)](#). 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 a bit of 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 had to always 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.

Let's say you wanted to pull out the wind speed part of the JSON response. Here's the dot notation you would 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 \(page 59\)](#)), 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). This 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 \(page 63\)](#). 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/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=fd4698c940c6d1da602a70ac34f0b147&units=imperial",
        "method": "GET"
      }

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

To print a specific property from the response to the page, modify your code to look like this:

```
<!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=fd4698c940c6d1da602a70ac34f0b147&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>
```

You can view the result here: idratherbewriting.com/learnapidoc/assets/files/weather-windspeed.html.

Here's what we changed:

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

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

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

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

Below this same section, 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 append the `content` variable to 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. Although objects allow you to get a specific property, arrays require you to select the position in the list that you want.

More exercises

If you'd like to follow some more exercises that involve calling REST APIs, accessing specific values, and printing them to the page, see the Resources section in the online site.

Dive into dot notation

In the previous topic, [Access and print a specific JSON value \(page 69\)](#), 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 key to making 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.

2. Change `john.children[0].child1` to display the corresponding 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/learnapidoc/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 \(page 32\)](#), I showed an example of [embedding the wind speed \(page 32\)](#) 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:

```
<html>
  <head>
    <script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
    <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=fd4698c940c6d1da602a70ac34f0b147&units=imperial",
          "method": "GET"
        }

        $.ajax(settings)

        .done(function (response) {
          console.log(response);

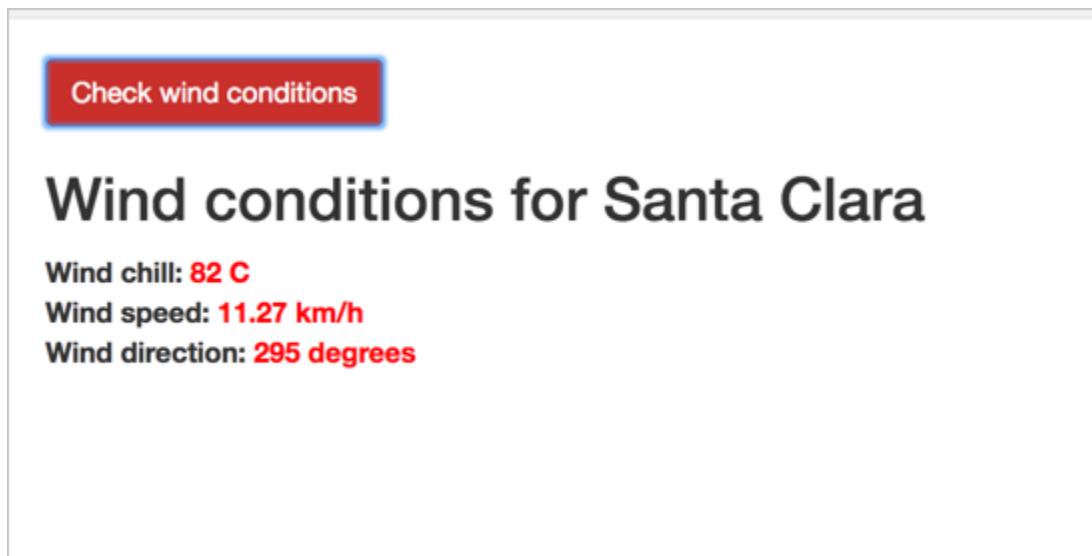
          $("#wind_speed").append (response.wind.speed);
          $("#wind_direction").append (response.wind.deg);
          $("#main_temp").append (response.main.temp);
          $("#weather_conditions").append (response.weather[0].main);
          $("#wind_speed_unit").append (" MPH");
          $("#wind_degree_unit").append (" 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>
    </span>
  
```

```
</body>  
</html>
```

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 \(page 69\)](#). 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:



You can view the file idratherbewriting.com/learnapidoc/assets/files/wind-openweathermap.html.

Next section

As you've gone through the exercise of 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 \(page 78\)](#), you'll assume the task of [documenting a new endpoint \(page 79\)](#) 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 the reference information.

Documenting API endpoints

A new endpoint to document	79
API reference tutorial overview	82
Step 1: Resource description (API ref tutorial)	84
Step 2: Endpoints and methods (API ref tutorial)	89
Step 3: Parameters (API ref tutorial)	95
Step 4: Request example (API ref tutorial).....	104
Step 5: Response example and schema (API ref tutorial)	116
Putting it all together	131
Activity: Evaluate API reference docs for core elements.....	135
Activity: Find an Open Source Project.....	137

A new endpoint to document

Until this point, you've been [acting as a developer \(page 31\)](#) 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.

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 individual endpoint as an "API" as well.)

"Here's the wiki page that contains all the data," the manager says. The information is scattered and random on the wiki page.

It's now your task to sort through the information on this 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, I will create documentation from this information as I proceed through each of the needed sections for an API reference topic.

Sorting out information

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, including test logins, access protocols, or code names), or have sections that are out-of-date.

Ultimately, the information will be oriented toward 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

Here's the mock internal wiki page:

The wiki page: "Surf Report API"

The new endpoint is `/surfreport/{beachId}`. This 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. This 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 http://example.com/surfreport/beaches_available. 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.

If the query is malformed, you get error code 400 and an indication of the error.

Next steps

Jump into the [API reference tutorial overview \(page 82\)](#) for an overview of the 5 steps we'll cover in creating the API reference topic for this new endpoint.

API reference tutorial overview

In this API reference tutorial 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. To provide some context (and to continue with our sample documentation scenario), we'll structure the information from the [new endpoint to document \(page 79\)](#) into these five sections.

Five common sections in REST API docs

Almost all API reference topics include these five sections:

[1. Resource description \(page 84\)](#)

"Resources" refers to the information returned by an API.

[2. Endpoints and methods \(page 89\)](#)

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

[3. Parameters \(page 95\)](#)

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 \(page 104\)](#)

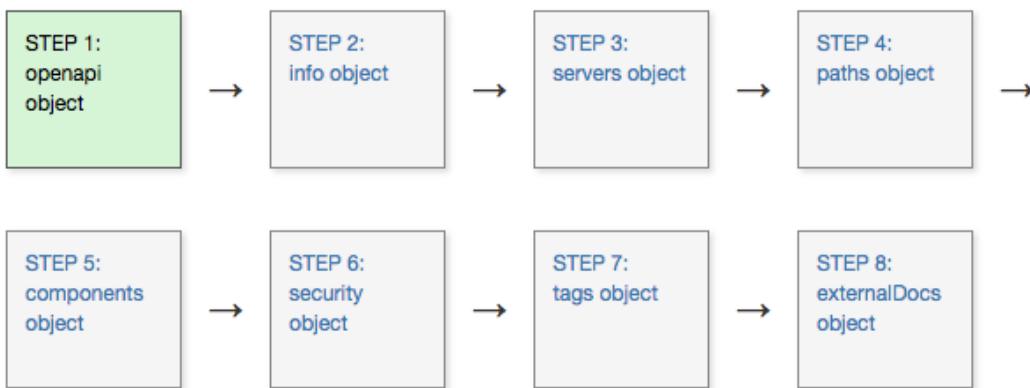
The request example includes a sample request using the endpoint, showing some parameters configured.

[5. Response example and schema \(page 116\)](#)

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 includes a workflow map to help guide and orient you each step of the way.



(page 84)

After the tutorial

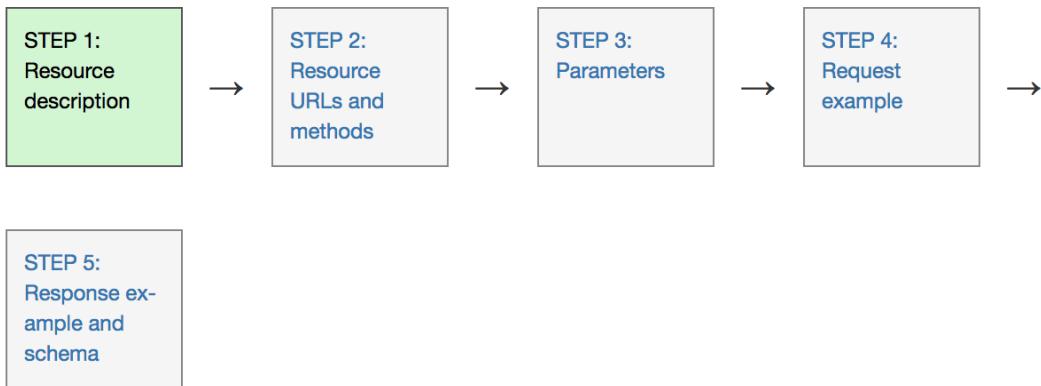
When we're finished, the end result will look [like this \(page 131\)](#). You'll then have an opportunity to [create an API reference topic \(page 135\)](#) with your own [open-source API project \(page 137\)](#).

Although there are automated ways to publish API docs, we're focusing on content rather than tools in this section. In the [Publishing your API documentation section \(page 334\)](#), we'll jump into publishing tools and methods.

Next steps

Now that you have an idea of the tutorial, let's get going with the first section: [Resource description \(page 84\)](#).

Step 1: Resource description (API reference tutorial)



"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 provides details about the information returned in each resource.

The resource description is brief (1-3 sentences) and usually starts with a verb. Resources usually have a number of endpoints to access the resource, and multiple methods for each endpoint. Thus, 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](#):

The screenshot shows the MailChimp Developer API documentation. At the top, there's a navigation bar with links for 'Developer' (which has a monkey icon), 'Documentation', 'Playground', and 'API Status'. Below this, on the left, is a sidebar with a 'Reference:' heading and a list of links: 'Overview', 'API Root', 'Authorized Apps', 'Automations', 'Batch Operations', 'Batch Webhooks', 'Campaign Folders', and 'Campaigns'. Under 'Campaigns', there are links for 'Content', 'Feedback', 'Send Checklist', 'Conversations', and 'E-commerce Stores'. To the right of the sidebar, the main content area has a title 'Campaigns' and a sub-section 'Subresources' with links for 'Content', 'Feedback', and 'Send Checklist'. Below this, there's a section titled 'Available methods' with a table showing actions for different endpoints. The table has columns for 'Create', 'Read', 'Edit', 'Delete', and 'Action'. It lists two rows: one for '/campaigns' (with 'GET' method) and one for '/campaigns/{campaign_id}' (also with 'GET' method). The first row is highlighted with a yellow background, and the second row is also highlighted with a yellow background.

	Create	Read	Edit	Delete	Action
<code>GET /campaigns</code>					Get all campaigns
<code>GET /campaigns/{campaign_id}</code>					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](#):

The screenshot shows the Box API documentation for the Membership Object. On the left, there's a sidebar with sections for GROUPS, COLLABORATIONS, and a yellow-highlighted Membership Object. The Membership Object section contains endpoints for Get Membership, Create Membership, Update Membership, Delete Membership, Get Memberships for Group, Get Memberships for User, and Get Collaborations for Group. The main content area has a title "Membership Object" and a "SUGGEST EDITS" button. A yellow box contains a description: "Membership is used to signify that a user is part of a group. Membership can be added, requested, updated and deleted. You can also get all members of a group, or all memberships for a given user." Below this are detailed descriptions for each field of the Membership object:

type	Type for membership is 'group_membership'
id	Box's unique string identifying this membership
user	Mini representation of the user, including id and name of user.
group	Mini representation of the group, including id and name of group.
role	The role of the user in the group. Default is "member" with option for "admin"
created_at	The time this membership was created.
modified_at	The time this membership was last modified.

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 let 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:

The screenshot shows the Eventbrite API Documentation for the Events resource. At the top, there's a navigation bar with links for Eventbrite, Search for events, BROWSE EVENTS, HELP ▾, SIGN IN, and CREATE EVENT. The main content area has a search bar for documentation and a breadcrumb trail: Eventbrite APIv3 Documentation > Events. The title is "Events". Below the title is a description for the GET /events/search/ endpoint: "Allows you to retrieve a paginated response of public event objects from across Eventbrite's directory, regardless of which user owns the event." A table titled "Parameters" lists the following fields:

NAME	TYPE	REQUIRED	DESCRIPTION
q	string	No	Return events matching the given keywords. This parameter will accept any string as a keyword.
sort_by	string	No	Parameter you want to sort by - options are "date", "distance" and "best". Prefix with a hyphen to reverse the order, e.g. "-date".
location.address	string	No	The address of the location you want to search for events around.
location.within	string	No	The distance you want to search around the given location. This should be an integer followed by "mi" or "km".

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 a lot of 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 with the same endpoint.

As another example, here's the Relationships resource in the [Instagram API](#).

The screenshot shows the Instagram API documentation interface. At the top, there's a navigation bar with links for "Sandbox Invites", "Manage Clients", and "Log in". Below the navigation, a search bar says "Search Documentation". On the left, a sidebar menu lists categories like "Overview", "Authentication", "Login Permissions", "Permissions Review", "Sandbox Mode", "Secure Requests", and "Endpoints". Under "Endpoints", "Relationships" is selected, which is highlighted in blue. The main content area has a yellow header box titled "Relationship Endpoints". Below it, a table lists five API endpoints with their descriptions:

<code>GET /users/self/follows</code>	Get the list of users this user follows.
<code>GET /users/self/followed-by</code>	Get the list of users this user is followed by.
<code>GET /users/self/requested-by</code>	List the users who have requested to follow.
<code>GET /users/ user-id /relationship</code>	Get information about a relationship to another user.
<code>POST /users/ user-id /relationship</code>	Modify the relationship with target user.

Below the table, a detailed view for the first endpoint is shown. It includes the method ("GET"), the endpoint ("`/users/self/follows`"), the URL (`https://api.instagram.com/v1/users/self/follows?access_token=ACCESS-TOKEN`), and a "RESPONSE" tab. The response description says "Get the list of users this user follows." It also specifies requirements ("Scope: follower_list") and 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/follows`

- GET `/users/self/followed-byGet`
- GET `/users/self/requested-byList`
- GET `/users/user-id/relationshipGet`
- POST `/users/user-id/relationshipModify`

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 \(page 316\)](#).

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.

Despite the variety with terminology, in general 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.)

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 non-reference sections \(page 263\)](#).

Although the description in an API reference topic provides a 1-3 sentence summary of the information 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 \(page 79\)](#) (which contains the information about the resource) and try to describe the resource in 1-3 sentences. Here’s my approach:

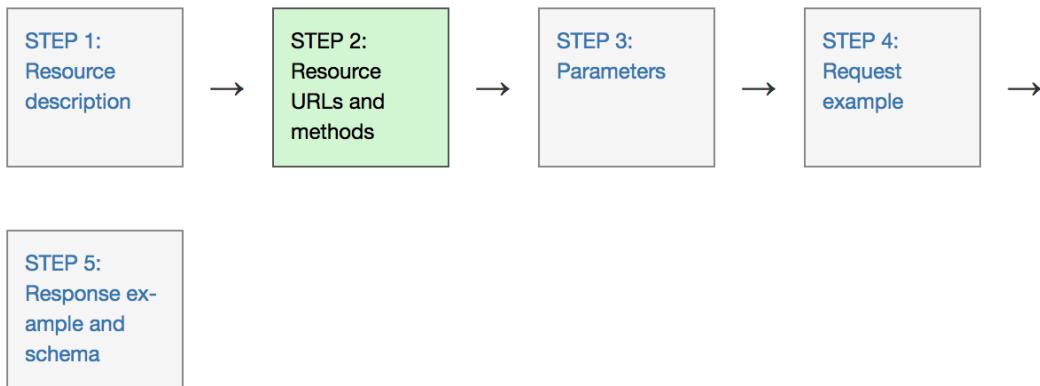
Surfreport

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 \(page 89\)](#) for the resource.

Step 2: Endpoints and methods (API reference tutorial)



The endpoints indicate how you access the resource, and the method used with the endpoint indicates the allowed interactions (such as GET, POST, or DELETE) with the resource. The endpoint shows the end path of a resource URL only, not the base path common to all endpoints.

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.

Example of an endpoints

Here's an example of the endpoints for the Relationships resource in the Instagram API:

The screenshot shows a web browser displaying the Instagram API documentation at <https://www.instagram.com/developer/endpoints/relationships/>. The left sidebar has a 'Endpoints' section with a 'Relationships' item selected. Two main API endpoints are listed:

- GET /users/self/follows**: Described as "Get the list of users this user follows." It requires a scope of "follower_list" and an ACCESS_TOKEN parameter.
- GET /users/self/followed-by**: Described as "Get the list of users this user is followed by." It also requires a scope of "follower_list" and an ACCESS_TOKEN parameter.

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, since this is what developers will implement to make their requests.

Represent path parameters with curly braces

If you have [path parameters \(page 98\)](#) in your endpoint, represent them through curly braces. For example, here's an example from Mailchimp's API:

```
/campaigns/{campaign_id}/actions/send
```

Better yet, put the path parameter in another color to set it off:

```
/campaigns/{campaign_id}/actions/send
```

Curly braces 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:

The screenshot shows the Facebook Graph API Reference page for the Achievement endpoint. At the top, there's a blue header bar with a message about email settings. Below it, the navigation bar includes 'All Docs' and 'Graph API Version v2.11'. The main content area has a yellow header 'Achievement /{achievement-id}'. It describes the resource as representing a user gaining an achievement in a Facebook game. A sidebar on the left lists various API categories like 'Achievement Type', 'Album', etc. Below the description, there's a 'Reading' section with a code block showing a GET request to 'GET /v2.11/{achievement-id} HTTP/1.1' and 'Host: graph.facebook.com'.

When the parameters are described, 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 beside 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 Box's API:

The screenshot shows the Box Developer API documentation for the 'Create Comment' endpoint. The left sidebar lists various API endpoints under 'Comment Object', with 'Create Comment' highlighted. The main content area shows a yellow box for the 'POST Create Comment' method. It describes the action as adding a comment to a specific file or comment. Below this is a 'Parameters' table:

	Type	Description
item	Object	The item that this comment will be placed on.
type	String	Child of <code>item</code> . The type of the item that this comment will be placed on. Can be <code>file</code> or <code>comment</code> .
id	String	Child of <code>item</code> . The id of the item that this comment will be placed on.
message	String	

On the right side, there's a 'Definition' section with the URL `/comments`, an 'Example Request' showing a curl command, and an 'Example Response' showing a JSON object.

And here's an example from LinkedIn's API:

The screenshot shows the LinkedIn API documentation for requesting data. It features a large yellow box for a 'GET' request to `https://api.linkedin.com/v1/people/~`. Below this is a 'sample response' section containing XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
  <id>1R2RtA</id>
  <first-name>Frodo</first-name>
  <last-name>Baggins</last-name>
  <headline>Jewelry Repossession in Middle Earth</headline>
  <site-standard-profile-navigation>
```

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 \(page 277\)](#), in an introductory section.

How to group multiple endpoints for the same resource

Another consideration is how to group and list the endpoints, particularly if you have a lot of endpoints for the same resource. In the [resource descriptions step \(page 84\)](#), we looked at a variety of APIs, and many provide different document 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 out 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 out 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 \(page 345\)](#), I explain that [long pages \(page 351\)](#) 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 non-reference 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 get problematic.

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 difficult the reference. These kinds of descriptions are more common in the [non-reference sections \(page 263\)](#) sections of your documentation. However, brief and clear references to the endpoints are sometimes challenging. There’s not a clear solution about how to refer to cumbersome endpoints. Adopt a convention that makes the most sense for your API.

Endpoint for surfreport API

Let’s create the Endpoints section for our [fictitious surfreport API \(page 79\)](#). Here’s my approach:

Endpoints

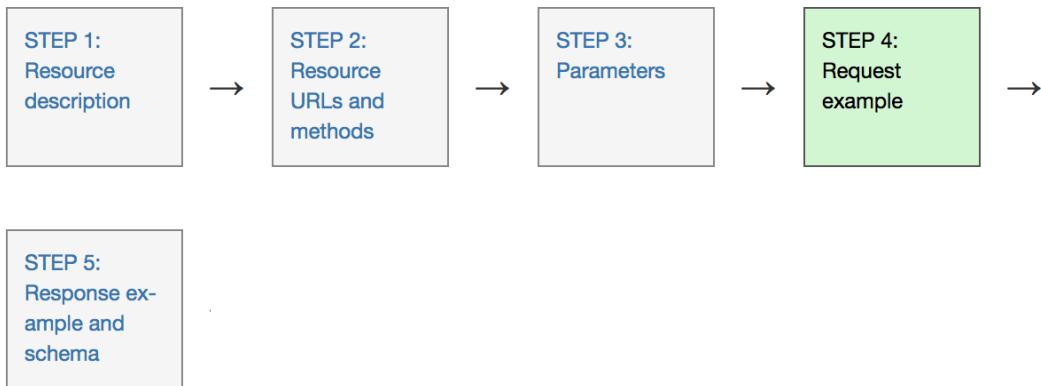
GET `surfreport/{beachId}`

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 \(page 95\)](#).

Step 3: Parameters (API reference tutorial)



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 four types of parameters: header parameters, path parameters, query string parameters, and request body parameters.

The different types of parameters are often documented in separate groups on the same page. Not all endpoints contain each type of parameter.

Example of parameters

The following screenshot shows a sample parameters section with the Box API:

Update Collaboration

Update a collaboration.

 SUGGEST EDITS

PUT https://api.box.com/2.0/collaborations/`collab_id`  

PATH PARAMS	
<code>collab_id</code>	REQUIRED string

QUERY PARAMS	
<code>fields</code>	Comma-separated list of fields to include in the response string

BODY PARAMS	
<code>role</code>	REQUIRED string The level of access granted. Can be <code>editor</code> , <code>viewer</code> , <code>previewer</code> , <code>uploader</code> , <code>previewer uploader</code> , <code>viewer uploader</code> , <code>co-owner</code> , or <code>owner</code> .
<code>status</code>	The status of the collaboration invitation. Can be <code>accepted</code> , <code>pending</code> , or <code>rejected</code> . string
<code>can_view_path</code>	Whether view path collaboration feature is enabled or not. View path collaborations allow the invitee to see the entire ancestral path to the associated folder. The user will not gain privileges in any ancestral folder (e.g. see content the user is not collaborated on). boolean

In this example, the parameters are grouped by type: path parameters, query parameters, and body parameters. The endpoint also sets off the path parameter (`collab_id`) in an 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
<code>format</code>	Optional	String

Here's an example from Yelp's documentation:

Yelp Fusion

[Return to Yelp Fusion](#)

API v2

[Get started](#)

[API console](#)

[Documentation](#)

[Introduction](#)

[Authentication](#)

Search API

[Business API](#)

[Phone Search API](#)

[iPhone Apps](#)

[Errors](#)

[Code samples](#)

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.

Four types of parameters

REST APIs have four types of parameters:

- **Header parameters (page 98):** Parameters included in the request header, usually related to authorization.
- **Path parameters (page 98):** Parameters within the path of the endpoint, before the query string (`?`). These are usually set off within curly braces.
- **Query string parameters (page 99):** Parameters in the query string of the endpoint, after the `?` .
- **Request body parameters (page 100):** Parameters included in the request body. Usually submitted as JSON.

The terms for each of these parameter types comes from the [OpenAPI specification \(page 160\)](#), which defines a formal specification that includes descriptions of each parameter type (see the [Path object tutorial \(page 173\)](#)). 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, consider noting the following:

- [Data types \(page 98\)](#)
- [Max and min values \(page 98\)](#)

Data types for parameters

APIs may not process the parameter correctly if it's the wrong data type or wrong format. Therefore, it's important to list the data type for each parameter. This is usually a good idea with all parameter types but is especially true for request body parameters, since these are usually 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 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. You can usually just write "number."

Max and min values for parameters

In addition to specifying the data type, the parameters should indicate the maximum, minimum, and allowed values. 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 unallowed values is a common pitfall in docs. Developers often don't realize all the "creative" ways users might use the APIs. The quality assurance team (QA) is probably your best resource for identifying the values that aren't allowed because it's QA's job to try to break the API.

When you test an API, try running a 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 \(page 286\)](#). I talk more about the importance of testing in [Testing your docs \(page 247\)](#).

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 parameters are documented in the [authorization requirements section \(page 277\)](#).

However, if your endpoint requires unique parameters to be passed in the header, you would document them in the parameters documentation here. (For more on request and response headers, see the [curl tutorial \(page 49\)](#) where we explored this with some examples.)

Path parameters

Path parameters are part of the endpoint itself, and are not optional. For example, `{user}` and `{bicycleId}` are the path parameters in the following endpoint:

```
/service/myresource/user/{user}/bicycles/{bicycleId}
```

Path parameters are usually set off with curly braces, but some API doc style's precede the value with a colon or use other 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. This 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}
```

Optionally, you could also use the same color for the parameters in your documentation:

URL Parameter	Description
user	Here's my description of the user parameter.
bicycles	Here's my description of the bicycles parameter.

By color coding the parameters, it's easy to see the parameter being defined 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, 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 body parameters

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 parameter, 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 a number of key-value pairs, like this:

```
{
  "days": 2,
  "units": "imperial",
  "time": 1433524597
}
```

Documenting complex request body parameters

Documenting JSON data (both in request body parameters and responses) is actually 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 carefully think 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.

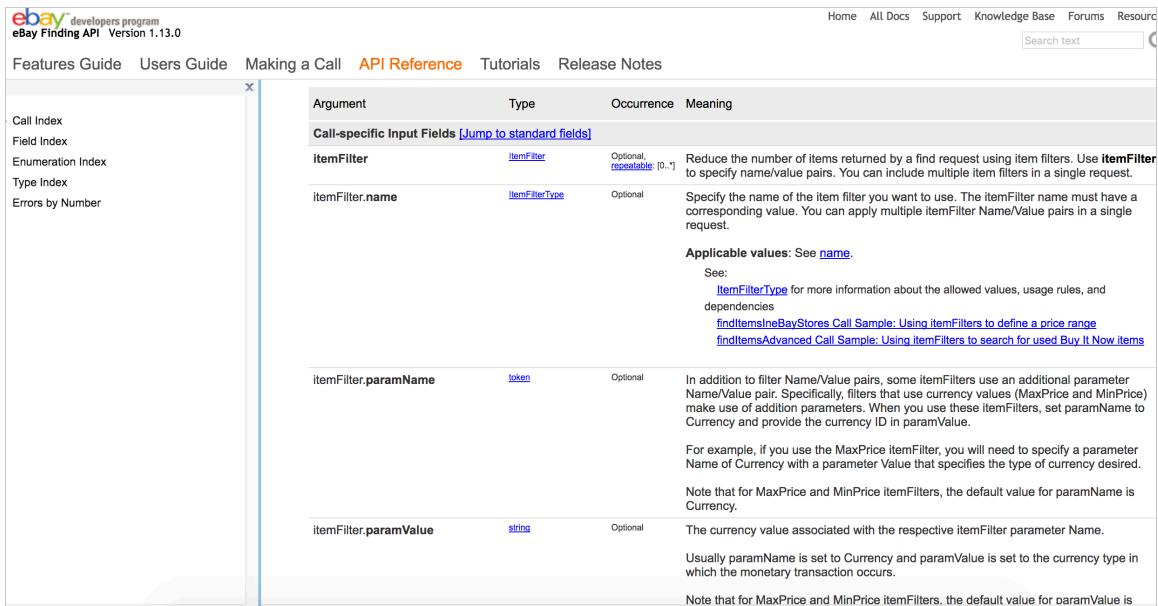
By all means, if the JSON object is relatively small, a table is probably your best option. But there are other approaches that designers have taken as well.

Take a look at eBay's [findItemsByProduct](#) resource. Here's the request body parameter (in this case, the format is XML):

The screenshot shows the eBay Developers Program API Reference interface. The top navigation bar includes links for Home, All Docs, Support, Knowledge Base, and Forum, along with a search bar. The main content area has a sidebar with links to Call Index, Field Index, Enumeration Index, Type Index, and Errors by Number. The main content area is titled 'Input' and contains the XML schema for the request body. The schema is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<findItemsByProductRequest xmlns="http://www.ebay.com/marketplace/search/v1/services">
  <!-- Call-specific Input Fields -->
  <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 parameter is a table that describes each parameter:



The screenshot shows a table titled "Call-specific Input Fields" from the eBay Developers program API Reference. The table has columns for Argument, Type, Occurrence, and Meaning. It includes rows for itemFilter, itemFilter.name, itemFilter.paramName, and itemFilter.paramValue. Each row provides detailed documentation, including applicable values and examples.

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.
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 additional parameters. When you use these itemFilters, set paramName to Currency and provide the currency ID in paramValue.
itemFilter.paramValue	string	Optional	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.
			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](#). This is likely because the parameter values are more complex and require more 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 for request body parameters

[Swagger UI \(page 214\)](#), which we explore later and also [demo \(page 223\)](#), provides another approach for documenting the request body parameter. Swagger UI shows the request body parameters 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 body parameters.

POST /pet Add a new pet to the store

Parameters

body * required Pet object that needs to be added to the store

(*body*) Example Value | Model

```

Pet ▾ {
    id          integer($int64)
    category   Category ▾ {
        id          integer($int64)
        name        string
    }
    name*       string
    photoUrls* string
    tags        ▾ [
        xml: OrderedMap { "name": "photoUrl", "wrapped": true }
        string
    ]
    status      Tag > {...}
                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 parameter and any descriptions of each element in the request body parameter.

The Model includes expand/collapse toggles with the values. (The [Petstore demo](#) doesn't actually include many parameter descriptions in the Model, but if any descriptions were included, they would appear here in the Model rather than the Example Value.)

In a later section, I dive into Swagger. If you want to skip there now, go to [Introduction to Swagger \(page 143\)](#).

You can see that there's a lot of variety in documenting JSON and XML in request body parameters. 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 clearly. I explore this topic in more depth in the [Response example and schema section \(page 116\)](#).

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}	Refers to the ID for the beach you want to look up. All Beach ID codes are available from our site at sampleurl.com.

Query string parameters

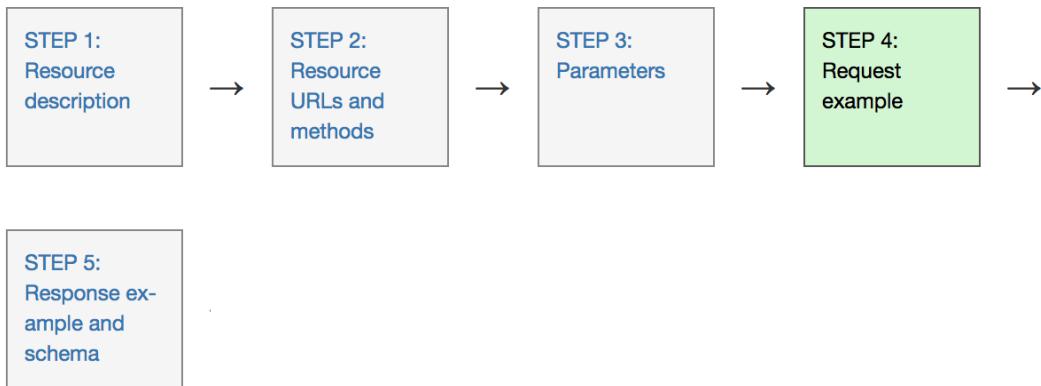
Query string parameter	Required / optional	Description	Type
days	Optional	The number of days to include in the response. Default is 3.	Integer
time	Optional	If you include the time, then only the current hour will be returned in the response.	Integer. Unix format (ms since 1970) in UTC.

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 \(page 104\)](#) for the resource.

Step 4: Request example (API reference tutorial)



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.

Example of a request

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 starting from 200. In response you can find a totalCount field.

curl -u "username:password" -H "Content-Type:application/json" -X GET "https://api.callfire.com/v2/texts?limit=50&offset=200"
```

Example Response

```
{
  "items": [
    {
      ...
      skipped ...
    },
    {
      ...
      skipped ...
    }
  ],
  ...
}
```

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 \(page 45\)](#).

```
curl -u "username:password" -H "Content-Type:application/json" -X GET "https://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, and other parameters.

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
```

cURL **Python**

You can add backslashes in curl to separate out each parameter onto its own line (though, as I pointed out in the [curl tutorial \(page 48\)](#), 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 <i>true</i> , <i>t</i> or <i>f</i> statuses will not be included in the returned user objects.	<i>false</i>	<i>false</i>
	include_user_entities	optional	The user object <i>entities</i> node will not be included when set to <i>false</i> .	<i>true</i>	<i>false</i>

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",
      "name": "Twitter API"
    }
  ]
}
```

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 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&where=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 \(page 126\)](#), I get into more details about dynamically showing the response when users click the 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 usually understand the pattern after a few examples.

Requests in various languages

One aspect of REST APIs that facilitates widespread adoption is that they aren't tied to a specific programming language. 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:

The screenshot shows the Twilio Voice API documentation for 'Making Calls'. On the left, there's a sidebar with links like 'Guides', 'Tutorials', 'API Reference', and 'SDKs'. The main content area has a title 'MAKING CALLS' and a rating section. Below that, it says 'Using the Twilio REST API, you can make outgoing calls to phones, SIP-enabled endpoints, and Twilio Client connections.' It also includes a note about rate limiting and a link to a step-by-step guide. On the right, there's a 'Make an Outbound Call' section with tabs for different languages: Java (selected), C#, CURL, JAVA, NODE.JS, PHP, PYTHON, and RUBY. The Java tab shows sample code for creating a call using the Twilio Java SDK:

```

JAVA
SDK Version: 6.x 7.x
View helper library from twilio.com/docs/java/install
View RI;
View URISyntaxException;
View Twilio;
View rest.api.v2010.account.Call;
View type.PhoneNumber;

public class Example {
10 // Find your Account Sid and Token at twilio.com/user/account
11 public static final String ACCOUNT_SID = "ACXXXXXXXXXXXXXXXXXXXX";
12 public static final String AUTH_TOKEN = "your_auth_token";
13
14 public static void main(String[] args) throws URISyntaxException {
15     Twilio.init(ACCOUNT_SID, AUTH_TOKEN);
16
17     Call call = Call.creator(new PhoneNumber("+14155551212"), new PhoneNumber("http://demo.twilio.com/docs/voice.xml")).create();
18
19     System.out.println(call.getSid());
}

```

You can select which language you want the sample request in: C#, curl, Java, Node.js, PHP, Python, or Ruby.

Here's another example from the Clearbit API:

The screenshot shows the Clearbit API documentation for the 'Combined API'. It starts with a brief description of the combined API, stating it's useful for looking up both a person and company simultaneously based on an email address. It then provides an example of an HTTP GET request to the 'combined/find' endpoint, specifying the email parameter. Below this, it lists supported parameters and provides a table of them. To the right, there's a section for the 'Combined API' with tabs for 'Shell', 'Ruby', 'Node', and 'Python'. The 'Node' tab is selected, showing sample code for performing a combined lookup on an email address:

```

var clearbit = require('clearbit')({key});

clearbit.Enrichment.find({email: 'alex@alexmaccaw.com', stream: 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 make the translate the curl request into a particular programming language.

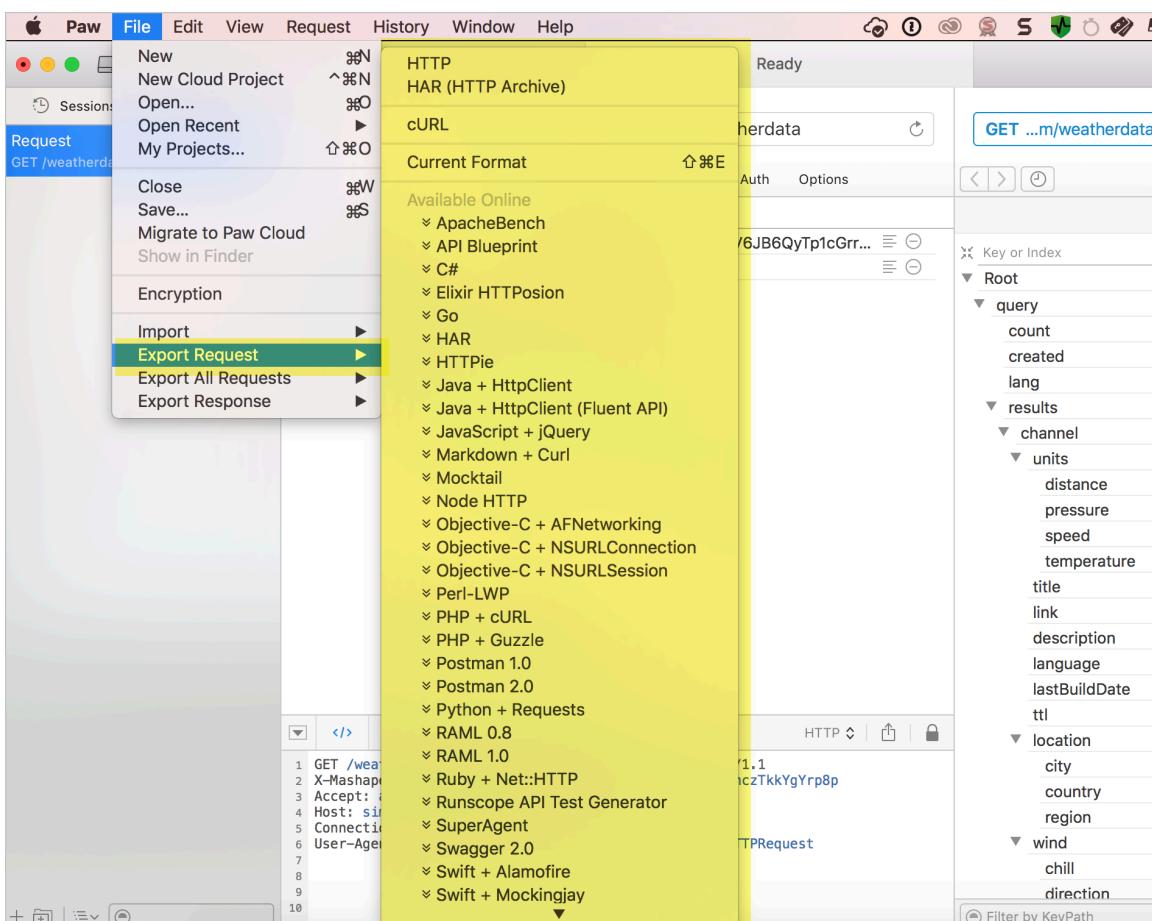
Providing a variety of requests like this, often displayed through [tabs](#), helps make your API easier to implement. It's even better if you can automatically populate the API keys with the actual user's API keys based on their logged-in profile.

However, don't feel so intimidated by this smorgasbord of code samples. Some API doc tools (such as [Readme.io \(page 387\)](#) or [SwaggerHub \(page 224\)](#)) can automatically generate these code samples because the patterns for making REST requests in different programming languages follow a common template.

Auto-generating code samples

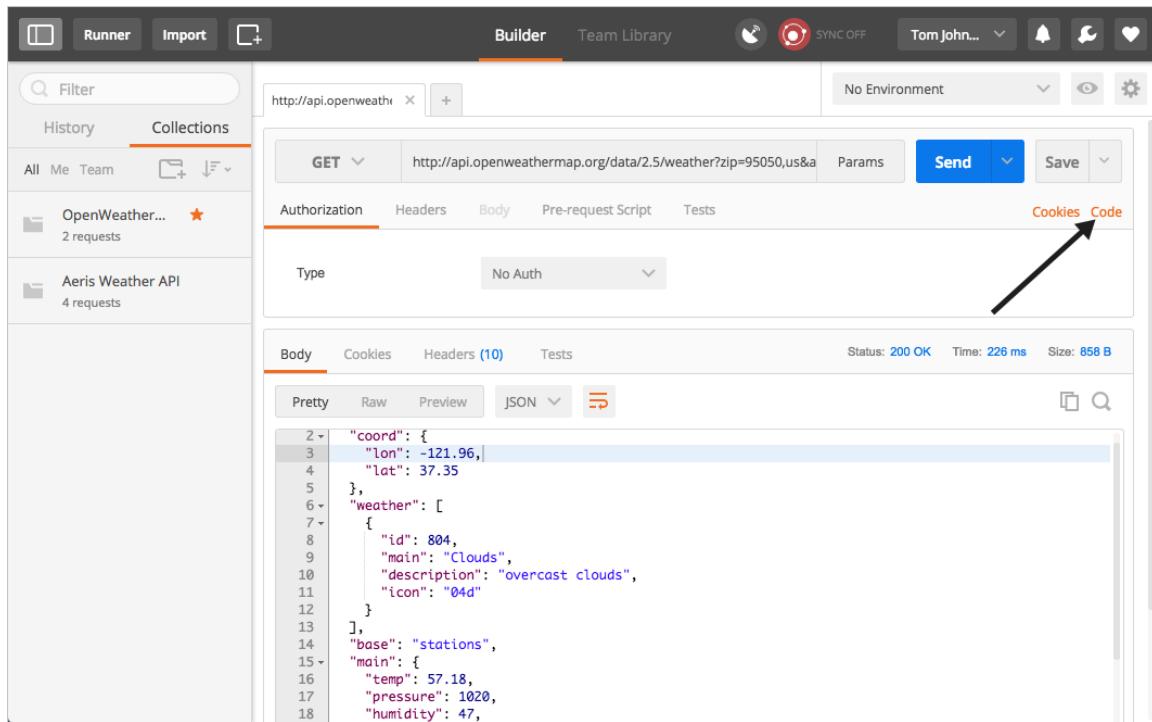
If you're not using an authoring tool that auto-generates code examples, and you want to provide these code snippets, you can auto-generate code samples from both Postman and Paw, if desired.

[Paw](#) (for Mac) lets you export your request into nearly every conceivable language:



After you have a request configured (a process similar to [Postman \(page 39\)](#)), you can generate a code snippet by going to [File > Export Request](#).

The Postman app also has the ability to generate code snippets in a similar way. I covered this process in an earlier tutorial on [using the JSON from the response payload \(page 63\)](#). In Postman, after you configure your request, click the **Code** link (which appears below the Save button in the upper-right area).

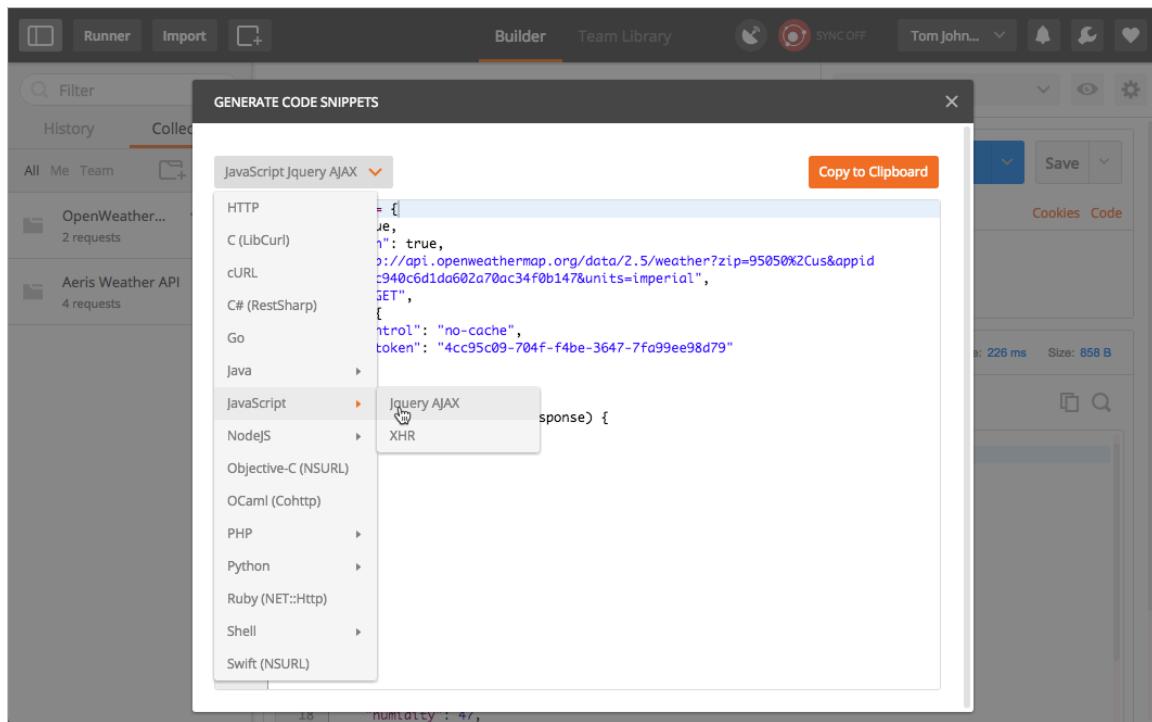


The screenshot shows the Postman Builder interface. A GET request is made to <http://api.openweathermap.org/data/2.5/weather?zip=95050,us&appid=c940c6d1a602a70ac34f0b147&units=imperial>. The response status is 200 OK, time is 226 ms, and size is 858 B. The response body is a JSON object containing weather information for San Jose, CA.

```

2 "coord": {
3   "lon": -121.96,
4   "lat": 37.35
5 },
6 "weather": [
7   {
8     "id": 804,
9     "main": "Clouds",
10    "description": "overcast clouds",
11    "icon": "04d"
12  }
13 ],
14 "base": "stations",
15 "main": {
16   "temp": 57.18,
17   "pressure": 1020,
18   "humidity": 47,
19 }
20 }
  
```

Then select the language you want, such as JavaScript > Jquery AJAX:



The screenshot shows the 'GENERATE CODE SNIPPETS' dialog in Postman. The selected language is 'JavaScript Jquery AJAX'. The code generator provides snippets for various languages, including Jquery AJAX, which is highlighted. The 'Copy to Clipboard' button is visible at the top right of the dialog.

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 \(page 63\).](#))

SDKs provide tooling for APIs

A lot of times, developers will create an [SDK \(software development kit\) \(page 308\)](#) that accompanies a REST API. The SDK helps developers implement the API using specific tooling.

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 probably 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 \(page 298\)](#) 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:

The screenshot shows the Spotify Developer API Explorer interface. At the top, there's a navigation bar with the Spotify logo, 'DEVELOPER', 'TECHNOLOGIES', 'SUPPORT', and 'NEWS'. Below that is a green header bar with 'SPOTIFY DEVELOPER / WEB API / API CONSOLE'. On the left, there's a sidebar with a tree view of API endpoints under 'Interactive Console'. The main area is titled 'Get an Album'. It shows the 'Description' (Get an Album), 'Endpoint' (https://api.spotify.com/v1/albums/{id}), 'HTTP Method' (GET), and 'OAuth' (Required). Below this, there are input fields for 'Spotify Album ID' (containing '4aawyAB9vmqN3uQ7FjRGTy'), 'Market' (set to 'ES'), and 'OAuth Token' (containing 'OAuth Access Token'). A 'GET OAUTH TOKEN' button is also present. At the bottom, there are 'TRY IT' and 'FILL SAMPLE DATA' buttons. A 'curl' command is shown at the bottom right: `curl -X GET "https://api.spotify.com/v1/albums/" -H "Accept: application/json"`.

Flickr's API docs also have a built-in API Explorer:

The screenshot shows the Flickr API documentation for the `flickr.photos.search` endpoint. At the top, there's a navigation bar with links for **Sign Up**, **Explore**, and **Create**. A search bar on the right contains the placeholder text "Photos, people, or groups". Below the navigation, the title "The App Garden" is displayed, along with links for "Create an App", "API Documentation", "Feeds", and "What is the App Garden?".

The main content area is titled "flickr.photos.search". It features a table titled "Arguments" with columns for "Name", "Required", "Send", and "Value". The table lists the following arguments:

Name	Required	Send	Value
user_id	optional	<input type="checkbox"/>	<input type="text"/>
tags	optional	<input type="checkbox"/>	<input type="text"/>
tag_mode	optional	<input type="checkbox"/>	<input type="text"/>
text	optional	<input type="checkbox"/>	<input type="text"/>
min_upload_date	optional	<input type="checkbox"/>	<input type="text"/>
max_upload_date	optional	<input type="checkbox"/>	<input type="text"/>
min_taken_date	optional	<input type="checkbox"/>	<input type="text"/>
max_taken_date	optional	<input type="checkbox"/>	<input type="text"/>
license	optional	<input type="checkbox"/>	<input type="text"/>

To the right of the table, under the heading "Useful Values", are two sections: "Recent public photo IDs" and "Popular public group IDs".

Recent public photo IDs:
26597171409 - 1510528962-chang-beo-lot-xac-sau-90-ngey-luyen-tap
37658935234 - 20171103_134720
38342677422 - 20171104_151753

Popular public group IDs:
1577604@N20 - ■:Group with Experience■
16978849@N00 - Black and White
34427469792@N01 - FlickrCentral

As does the New York Times API:

The screenshot shows the Article Search API's Swagger UI interface. At the top, it says "Article Search API" and "Source: [Swagger 2.0]". There are tabs for "README", "Documentation", and "Console". Below that, there are dropdown menus for "All" and "GET /articlesearch.json", and a green "View Results" button.

The left sidebar has sections for "Credentials" (with an "apikey" field containing "1929d386d72b432ea663872cc7b114e2") and "Remember Keys". It also has a "Sample Code" section with code examples for JavaScript, NodeJS, PHP, and Ruby, and a "Copy" button. Below that is a "Parameters" section with fields for "q" (containing "writing"), "fq", "begin_date", and "end_date".

The main right area shows a JSON response for a search query. The response includes a header with status "OK", copyright information, and a "response" object containing "docs". Each document in the "docs" array has a "web_url" (e.g., "https://www.nytimes.com/2017/11/10/books/review/john-mcpee-draft-no-4-on-the-writing-process-new-yorker.html"), a snippet ("John McPhee's 'Draft No. 4' collects eight essays that offer writing advice and take readers behind the scenes of his creative process."), and two multimedia objects: one "xlarge" image and one "wide" image, each with their respective URLs, widths, heights, and ranks.

```

{
  "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/review/john-mcpee-draft-no-4-on-the-writing-process-new-yorker.html",
        "snippet": "John McPhee's \"Draft No. 4\" collects eight essays that offer writing advice and take readers behind the scenes 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-Kummer/1112-BKS-Kummer-articleLarge-v2.jpg",
            "height": 701,
            "width": 600,
            "rank": 0,
            "legacy": {
              "xlargeWidth": 600,
              "xlarge": "images/2017/11/12/books/review/1112-BKS-Kummer/1112-BKS-Kummer-articleLarge-v2.jpg",
              "xlargeHeight": 701
            }
          },
          {
            "type": "image",
            "subtype": "wide",
            "url": "images/2017/11/12/books/review/1112-BKS-Kummer/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 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:

The screenshot shows a modal dialog box overlaid on a table of API endpoints. The dialog contains the following text:

This is not a sandbox.
These are real API calls
that affect your account,
possibly altering data and
consuming credits.

At the bottom of the dialog are two buttons: "Cancel" and "Save".

The background table has columns for "Parameter", "Value", and "Description". One row is visible:

Parameter	Value	Description
date	<input type="text"/>	Retrieve the timestamp of the Block records. It will return a date in a MySQL timestamp format - YYYY- MM-DD HH:MM:SS

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.

The screenshot shows the Foursquare API Explorer interface. On the left is a sidebar with links like "Getting Started", "Detailed Docs", "Overview", "Responses & Errors", etc. The main area is titled "API Explorer" and shows a code snippet:

```
https://api.foursquare.com/v2/ users/self
```

Below the URL is an OAuth token:

```
https://api.foursquare.com/v2/users/self?  
oauth_token=Z1JPN4QNO23USRXRWFHJCYI3UU2XVLRMKRIRJDNFWFEZDYP4&v=20150607
```

A note says: "The OAuth token above is automatically generated for your convenience. Please DO NOT use this token for live applications in production."

At the bottom of the code block, there is a JSON response:

```
{
  "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 a [OpenAPI specification document \(page 160\)](#)) and [Readme.io](#) (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 \(page 214\)](#).

Request example for the `surfreport` endpoint

Let's return to the `surfreport/{beachId}` endpoint in our [sample scenario \(page 79\)](#) and create a request example for it. Here's my approach:

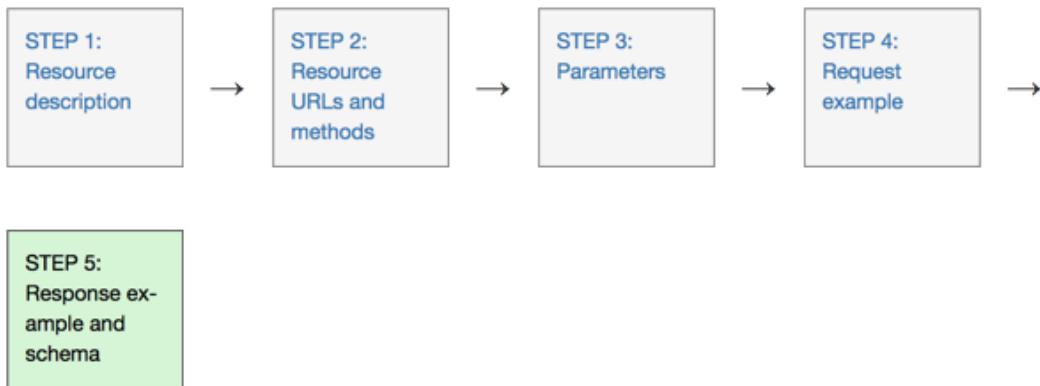
Sample request

```
curl -I -X GET "https://api.openweathermap.org/data/2.5/surfreport?zip=95050&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial&days=2"
```

Next steps

Now that we've created a sample request, the next steps naturally follow: include a [sample response \(page 116\)](#) that corresponds with the same request. We'll also document the model or schema of the response in general.

Step 5: Response example and schema (API reference tutorial)



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 returned, what each property contains, the data format of the values, the structure, and other details.

Example of a response example and response schema

The following is a sample response from the SendGrid API. Their documentation provides a tabbed display with an **Example** on one tab:

Responses application/json		
● 201	Schema	Example
● 400		{ "id": 986724, "title": "March Newsletter", "subject": "New Products for Spring!", "sender_id": 124451, "list_ids": [110, 124], "segment_ids": [110], "categories": ["spring line"], "suppression_group_id": 42, "custom_unsubscribe_url": "", "ip_pool": "marketing", "html_content": "<html><head><title></title></head><body><p>Check out our new products for Spring!</p></body></html>", "plain_content": "Check out our spring line!", "status": "Draft" }
● 401		

And the response **Schema** on another tab:

Responses application/json				
	Schema	Example	✖ collapse all	
● 201	object			
● 400	title		string	The display title of your campaign. This will be viewable by you in the Marketing Campaigns UI.
● 401	subject		string or null	The subject of your campaign that your recipients will see.
	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.
	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
	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.
	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 Sunset and Sunrise 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:**

Responses		
Code	Description	Links
200	<p><i>Sunrise and sunset times response</i></p> <p>application/json</p> <p>Controls <code>Accept</code> header.</p> <p>Example Value Model</p> <pre>{ "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" }</pre>	No links

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 API:

Responses

Code	Description	Links
200	Sunrise and sunset times response	No links
	<p>application/json</p> <p>Controls <code>Accept</code> header.</p> <p>Example Value Model</p> <pre> Response <v> { results <v> { sunrise <v> { string example: 7:27:02 AM time of sunrise } sunset <v> { string example: 5:05:55 PM time of sunset } solar_noon <v> { string example: 12:16:28 PM time when sun is at apex } day_length <v> { string example: 9:38:53 length of the day } civil_twilight_begin <v> { string example: 6:58:14 AM time when civil twilight begins } } } </pre>	

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 \(page 286\)](#)), 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 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 each of the possible 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 \(page 417\)](#) or markdown syntax with [GitHub \(page 394\)](#), you can probably use the [Rouge](#) built-in syntax highlighter. Other static site generators use [Pygments](#).

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 tool, you can add syntax highlighting manually for each code sample by pasting it into the [syntaxhighlight.in](#) highlighter.

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> MinimumAdvertisedPriceExpo
        <OriginalRetailPrice> AmountType (double) </OriginalRetailPi
        <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.			
Resource	Type	Conditionally	Description
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 this field 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.

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:

The screenshot shows the Stripe API documentation for the 'Create a charge' endpoint. The left sidebar lists various sections and methods. The main content area is a three-column table. The first column contains field names: 'currency', 'customer', 'source', and 'object'. The second column contains descriptions and requirements: '3-letter ISO code for currency. REQUIRED', 'The ID of an existing customer optional, either customer or source 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.', and 'The type of payment source. Should be "card". REQUIRED'. The third column displays a code sample in curl format:

```

curl Ruby Python PHP Java Node Go
{
  "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](#), [Spectacle](#), or [Readme.io](#).

Should you use a three-column layout with your API documentation? Maybe. However, if the response example and description doesn'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 of the example and the description to line up 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			Example json GET response
UID	Guid (36)	🔗	{
Name	String (30)	🔗	"UID" : "eb043b43-1d66-472b-a6ee-ad48def81b96", "Name" : "Business Bank Account #2",
DisplayID	String (6)	🔗	"DisplayID" : "1-1120", "Classification" : "Asset",
Classification	Classification	🔗	>Type" : "Bank", "Number" : 1120, >Description" : "Bank account clearwtr",
Type	AccountType	🔗	"ParentAccount" : {
Number	Integer (4)	🔗	"UID" : "f5cc9506-3472-4227-8c45-7a95c322c38b", "Name" : "Bank Accounts", "DisplayID" : "1-1100", "URI" : " {cf_uri}/GeneralLedger/Account 3472-4227-8c45-7a95c322c38b"
Description	String (255)	🔗	}, "IsActive" : true,
ParentAccount			"TaxCode" : {
UID	Guid (36)	🔗	"UID" : "94966872-b140-4da2-bc43-5dc74f33a09", "Code" : "N-T", "URI" : " {cf_uri}/GeneralLedger/TaxCode b140-4da2-bc43-5dc74f33a09"
Name	String (6)	🔗	}, },
DisplayID	String (6)	🔗	
U Description		🔗	
IsAct	Parent account code format includes separator ie 1-1000	🔗	
TaxCode		🔗	
UID	Guid (36)	🔗	
Code	String (3)	🔗	

This approach facilitates scanning, and the popover + collapsible approach allows you to compress the table so you can jump to the part you're interested in.

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. For example, look at the [Rhapsody API](#) and click an endpoint — the response is generated dynamically.

Another API with dynamic responses is the [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 dynamically returned in the browser.



The screenshot shows a browser window displaying a JSON response from the OpenWeatherMap API. The URL in the address bar is samples.openweathermap.org/data/2.5/weather?q=London,uk&appid=b697c185e2ca720a3c9f7d339c07a04. The JSON data is as follows:

```
{  
  "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": 80  
  }  
}
```

The Citygrid API, which we explored in the [requests example topic \(page 104\)](#), also dynamically generates responses.

This 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 common across all endpoints in the API, status and error codes are often documented in their own section, apart from a specific endpoint's documentation. I cover this topic in [Documenting status and error codes \(page 286\)](#).

Response example and schema for the surfreport endpoint

For the `surfreport/{beachId}` endpoint that we've been exploring in our [sample API scenario \(page 79\)](#), let's create a section that shows the response example and schema. Here's my approach for 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 great."  
        },  
        "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 get returned in the response.	Object
{time}	The time for the conditions. This item is only included 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, since 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 3.	String

Next steps

Now that you've completed each of the sections, take a look at all the sections together: [Putting it all together \(page 131\)](#).

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
<code>{beachId}</code>	Refers to the ID for the beach you want to look up. All Beach ID codes are available from our site at sampleurl.com.

Query string parameters

Query string parameter	Required / optional	Description	Type
<code>days</code>	Optional	The number of days to include in the response. Default is 3.	Integer
<code>time</code>	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=95050&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial&days=2"
```

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 great."
        },
        "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 get returned in the response.	Object
{time}	The time for the conditions. This item is only included 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, since 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 3.	String

Some notes on templates and tools

With the sample documentation here, I'm using Jekyll. Each of these sections is stored as a relative include that I've singled sourced to both this page and the previous pages. This ensures I'm not copying and pasting the same content in multiple areas of the site.

If you have a lot of endpoints to document, you'll probably want to create templates that follow a common structure. Additionally, if you want to add a lot of styling to each of the elements, you may want to push each of these elements into your template by way of a script. I'll talk more about publishing in the [Publishing API Documentation \(page 334\)](#).

Next steps

Now that you've completed the [API reference tutorial \(page 82\)](#), you're ready to either critique or create your own API reference topic. See the next topic: [Activity: Critique or create an API reference topic \(page 135\)](#).

Activity: Evaluate API reference docs for core elements

After completing the [API reference tutorial \(page 82\)](#), you're ready to start an activity that gives you more experience in creating and editing API reference documentation. In this activity, you'll either critique or create your own API reference topic for the open-source API project you [identified earlier \(page 137\)](#).

Activity 3a: 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. From this [list of about 100 API doc sites here \(page 341\)](#), identify three API documentation sites.
2. In each of the doc sites, look for the API reference documentation section (the list of endpoints).
3. In the reference documentation, identify each of the following sections:
 - [Resource description \(page 84\)](#)
 - [Endpoints and methods \(page 89\)](#)
 - [Parameters \(page 95\)](#)
 - [Request example \(page 104\)](#)
 - [Response example and schema \(page 116\)](#)

(The section names may differ in the API doc sites you find, but they're usually recognizable.)

4. 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, and request body parameters) for the endpoints?
- Are the data types (string, boolean, etc.) defined for each parameter? Are max/min 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?

Create or fix an API reference documentation topic

This part of the activity might be more difficult to do, but here is where you'll start building some examples for your portfolio.

1. 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 \(page 412\)](#) and contribute your edits to the project.

Next steps

Now that you've had your head buried in API reference documentation, it's time to dive into testing a bit more. 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 to testing your docs \(page 248\)](#).

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 a job in API documentation \(page 438\)](#). Without a portfolio that contains compelling API documentation samples, it will be extremely difficult to get a job in API documentation.

Here's the situation you want to avoid. Let's assume you don't have any experience in API documentation, but you're trying to get a 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? It's simple: you create these API doc samples through open source projects that you contribute to. This is where the activities in this course become important.

Rather than simply completing modules and tracking your progress toward the course's completion, I've included activities here that will actually 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 actually 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 \(page 0\)](#) 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.

Search for an open-source project with API doc needs

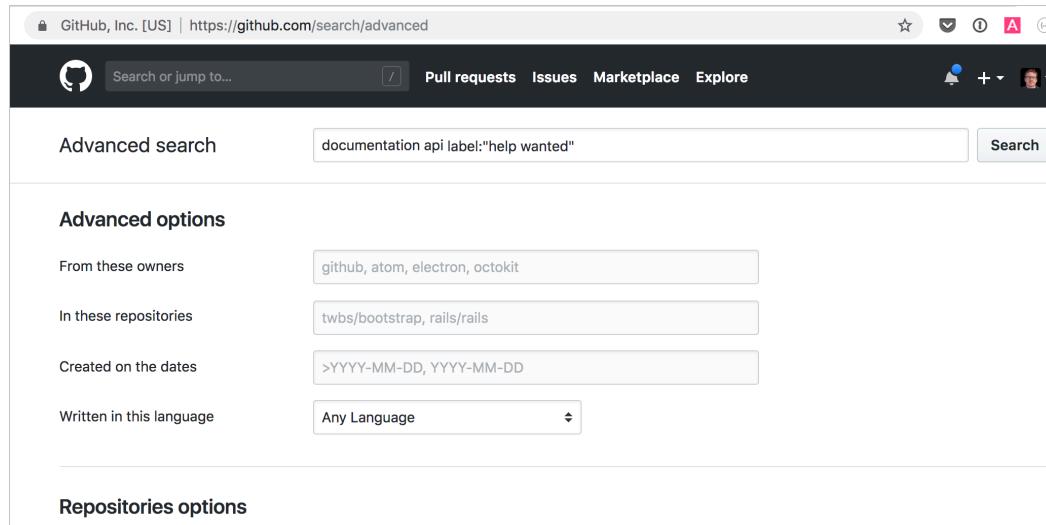
Activity 3b: Find an open-source project with API doc needs

Finding an open-source project to which you can contribute and work with can be invaluable for building a portfolio of developer documentation samples. 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 common tag 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 **documentation** and **api**) as well:



The screenshot shows the GitHub Advanced search page. At the top, there's a search bar with the query "documentation api label:\"help wanted\"". Below it, under "Advanced options", there are four filter boxes: "From these owners" containing "github, atom, electron, octokit", "In these repositories" containing "twbs/bootstrap, rails/rails", "Created on the dates" containing ">YYYY-MM-DD, YYYY-MM-DD", and "Written in this language" set to "Any Language". There's also a "Repositories options" section below.

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 \(page 0\)](#) such as a Java API). 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](#)
 - [Programmableweb.com API directory](#)

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 sort of 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.

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 \(page 412\)](#). This might require you to ramp up on [some 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 bad 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 a number of tutorials in 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 simply 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](#)

OpenAPI specification and Swagger

Overview of REST API specification formats.....	142
Introduction to the OpenAPI spec and Swagger.....	143
Working in YAML.....	156
OpenAPI tutorial overview	160
Step 1: openapi object (OpenAPI tutorial)	165
Step 2: info object (OpenAPI tutorial)	168
Step 3: servers object (OpenAPI tutorial)	170
Step 4: paths object (OpenAPI tutorial)	173
Step 5: components object (OpenAPI tutorial).....	181
Step 6: security object (OpenAPI tutorial)	200
Step 7: tags object (OpenAPI tutorial)	206
Step 8: externaldocs object (OpenAPI tutorial)	209
Activity: Create an OpenAPI specification document	212
Swagger UI tutorial	214
Swagger UI Demo	223
SwaggerHub introduction and tutorial	224
Stoplight — visual modeling tools for creating your spec.....	233
Integrating Swagger UI with the rest of your docs	241

Overview of REST API specification formats

When I [introduced REST APIs \(page 22\)](#), I mentioned that REST APIs follow an architectural style, not a specific standard. However, there are several REST specifications that have been developed to try to provide some standards about how REST APIs are described. The three most popular REST API specifications are as follows: [OpenAPI \(formally called Swagger\)](#), [RAML](#), and [API Blueprint](#).

In the early years of specifications, there was healthy competition between the formats. But now, without a doubt 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 fact, this entire section focuses on the OpenAPI specification. (I moved the [RAML \(page 0\)](#) and [API Blueprint \(page 0\)](#) pages into the 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. 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 documentation, tooling, and structure with your API documentation. Keep in mind that these REST API specifications mostly describe the [reference endpoints \(page 78\)](#) 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 [non-reference or user guide topics \(page 263\)](#).)

Nevertheless, the reference documentation 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 and Swagger

OpenAPI is a specification for describing REST APIs. You can think of the OpenAPI specification like the specification for DITA. 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 information.

With OpenAPI, instead of XML, you have 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 ingest the information about your API and display each component in a stylized, interactive display.

For step-by-step tutorial on creating an OpenAPI specification document, see the [OpenAPI tutorial overview \(page 160\)](#).

Glancing at the OpenAPI specification

To get a better sense of the OpenAPI specification, let's take a quick 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 actually an “object” (instead of an XML element) that holds a number of 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.
- `$ref` is actually a reference to another part of your implementation (`components`) where the response is defined. (OpenAPI has a lot of `$ref` references like this to keep your code clean and to facilitate re-use.)

It can take quite a while to figure out the OpenAPI specification. Give yourself a couple of weeks and a lot of example specification documents to look at, especially in the context of the actual API you're documenting. Remember that the OpenAPI specification is general enough to describe nearly every REST API, so some parts may be more applicable than others.

Validating the specification

When you're coding your OpenAPI specification document, instead of working in a text editor, you can write your code in the [Swagger Editor](#). The Swagger Editor dynamically validates your content to determine whether the specification document you're creating is valid.

The screenshot shows the Swagger Editor interface. On the left, a code editor displays a YAML-based OpenAPI specification for a "Petstore (Simple)" API. The specification includes details like the API version (2.0), title, description, contact information, license, and various API paths (e.g., /pets). On the right, a preview pane titled "Swagger Petstore (Simple)" shows the generated API documentation. It includes sections for "Contact information" (Swagger API team, email foo@example.com, URL http://swagger.io), "Terms of service" (http://helloworldverb.com/terms/), and "License" (MIT). Below this, under "Paths", there's a section for the "/pets" endpoint, which has a "GET /pets" operation listed. A blue box highlights this operation, and below it is a "Description" field. At the top right of the preview pane, a green checkmark indicates "Processed with no error".

```

1 swagger: '2.0'
2   info:
3     version: '1.0.0'
4     title: Swagger Petstore (Simple)
5     description: A sample API that uses a petstore
       as an example to demonstrate features in the
       swagger-2.0 specification
6     termsOfService: http://helloworldverb.com/terms/
7     contact:
8       name: Swagger API team
9       email: foo@example.com
10      url: http://swagger.io
11     license:
12       name: MIT
13       url: http://opensource.org/licenses/MIT
14   host: petstore.swagger.io
15   basePath: /api
16   schemes:
17     - http
18   consumes:
19     - application/json
20   produces:
21     - application/json
22   paths:
23     /pets:
24       get:
25         description: Returns all pets from the
           system that the user has access to
26         operationId: findPets
27         produces:
28           - application/json
29           - application/xml

```

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.

For your specification document's format, you have the choice of working in either JSON or YAML. The previous code sample is in [YAML](#). YAML refers to "YAML Ain't Markup Language," meaning YAML doesn't have any markup tags (`<>`), as is common with other markup languages such as XML.

YAML depends on spacing and colons to establish the object syntax. This makes the code more human-readable, but it's also sometimes trickier to get the spacing right.

Automatically generating the specification document

So far I've been talking about creating the OpenAPI specification document as if it's the technical writer's task and requires manual coding in a text editor based on close study of the specification. That's how I usually approach it, but it's not the only way to create the document. You can also auto-generate the specification document through annotations in the programming source 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. If this is the case, make sure you get access to the source code to make edits to the annotations. Otherwise, your developers will be writing your docs (which can be good but often has poor results).

Swagger offers a variety of libraries that you can add to your programming code to generate the specification document. These libraries are considered part of the [Swagger Codegen](#) project. For more information, see [Comparison of Automatic API Code Generation Tools For Swagger](#) by API Evangelist. For additional tools and libraries, see [Swagger services and tools](#) and [Open Source Integrations](#).

The annotation methods for Swagger doc blocks vary based on the programming language. For example, here's a [tutorial on annotating code with Swagger for Scalatra](#).

Auto-generating the Swagger file from code annotations

Instead of coding the Swagger file by hand, you can also auto-generate it from annotations in your programming code. There are many Swagger libraries for integrating with different code bases. These Swagger libraries then parse the annotations that developers add and generate the same Swagger file that you produced manually using the earlier steps.

By integrating Swagger into the code, you allow developers to easily write documentation, make sure new features are always documented, and keep the documentation more current.

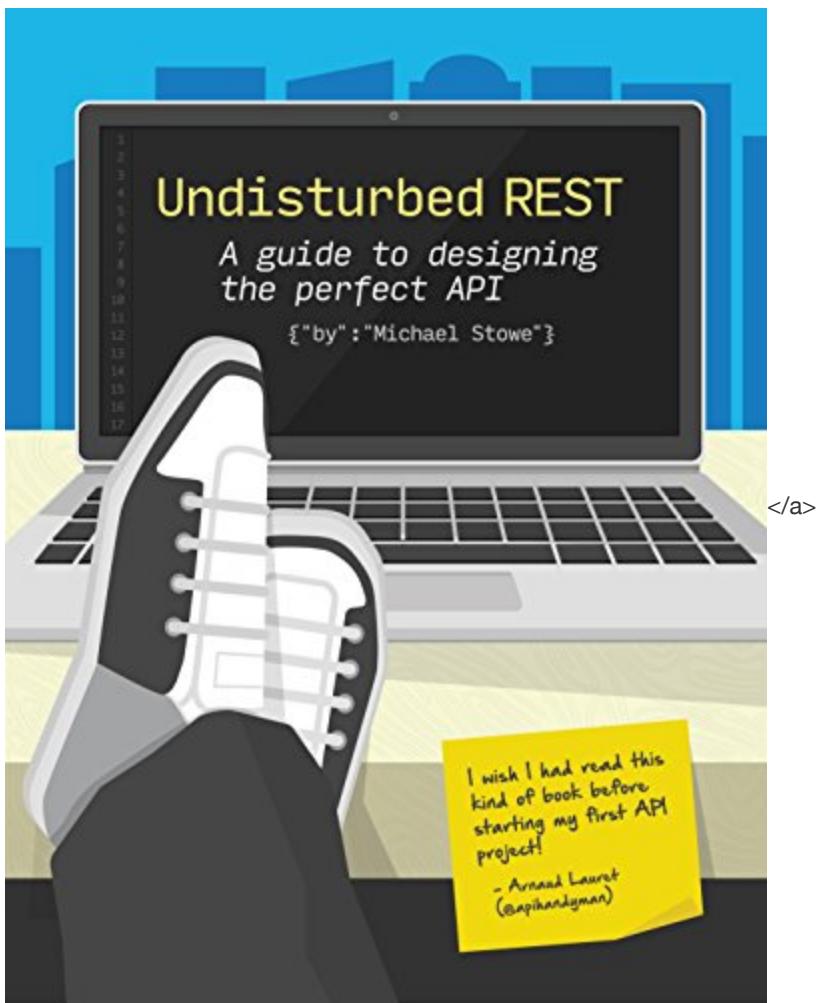
These libraries, specific to your programming language, will parse through your code's annotations and generate an OpenAPI specification document. Someone has to know exactly what annotations to add and how to add them (the process isn't too unlike Javadoc's comments and annotations). Then someone has 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 the OpenAPI schema. 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. Plus if engineers are writing the docs, they often prefer to stay within their own IDE to write.

Another approach: spec-first development

Although you can generate your specification document from code annotations, many say that auto-generation is *not* the best approach. In [Undisturbed REST: A Guide to Designing the Perfect API](#), Michael Stowe recommends that teams implement the 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."

```
<a target="_blank" class="noExtIcon" href="https://www.amazon.com/gp/product/B0125TOLNU?keywords=undisturbed%20rest%20michael%20stowe&qid=1444665700">
```



Spec-first development is a philosophy about how to develop APIs in a more efficient way.

In other words, developers consult the 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 to auto-generate the specification document.

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 backwards compatibility to previous versions), you want to avoid the “fail fast” approach that is so commonly celebrated with agile. There’s nothing worse than releasing a new version of your API that invalidates endpoints or parameters used in previous releases. Documentation versioning also becomes a nightmare.

In my conversations with [Smartbear](#), the company that makes [SwaggerHub](#) (page 224) (a collaborative platform for teams to work on Swagger API specifications), they say it’s now more common for teams to manually write the spec rather than embed source annotations in programming code to auto-generate the spec. The spec-first approach helps distribute the documentation work to more team members than engineers. Defining the spec before coding also helps teams produce better APIs.

Even before the API has been coded, your spec can generate a [mock response \(page 230\)](#) by adding response definitions in your spec. The mock server generates a response that looks like it's coming from a real server, but it's really just a pre-defined response in your code and appears to be dynamic to the user.

The tech writer's role with the spec

In most of my projects, developers haven't been that familiar with Swagger or OpenAPI, so I usually create the OpenAPI specification document by hand. Additionally, I often don't 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 Swagger or OpenAPI but are interested in using it as an approach to API documentation (the schema-based approach fits the engineering mindset). As such, 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 have a key role to play 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 be able to actually 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).

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 tutorial here: [Swagger UI tutorial \(page 214\)](#).)

The Swagger UI code generates a display that looks like this:

The screenshot shows the Swagger Petstore UI. At the top, there's a green header bar with the 'swagger' logo, the URL 'http://petstore.swagger.io/v2/swagger.json', and an 'Explore' button. Below the header, the title 'Swagger Petstore 1.0.0' is displayed, along with a note about the sample server and authorization. There are links for 'Terms of service', 'Contact the developer', 'Apache 2.0', and 'Find out more about Swagger'. A 'Schemes' dropdown is set to 'HTTP' and an 'Authorize' button is visible. The main content area shows two sections: 'pet' (Everything about your Pets) and 'store' (Access to Petstore orders). The 'store' section is expanded, showing four API endpoints: '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), and 'DELETE /store/order/{orderId}' (Delete purchase order by ID). Each endpoint is shown with its method, path, and a brief description.

You can also check out the [sample Swagger UI integration with a simple weather API](#) used as a course example.

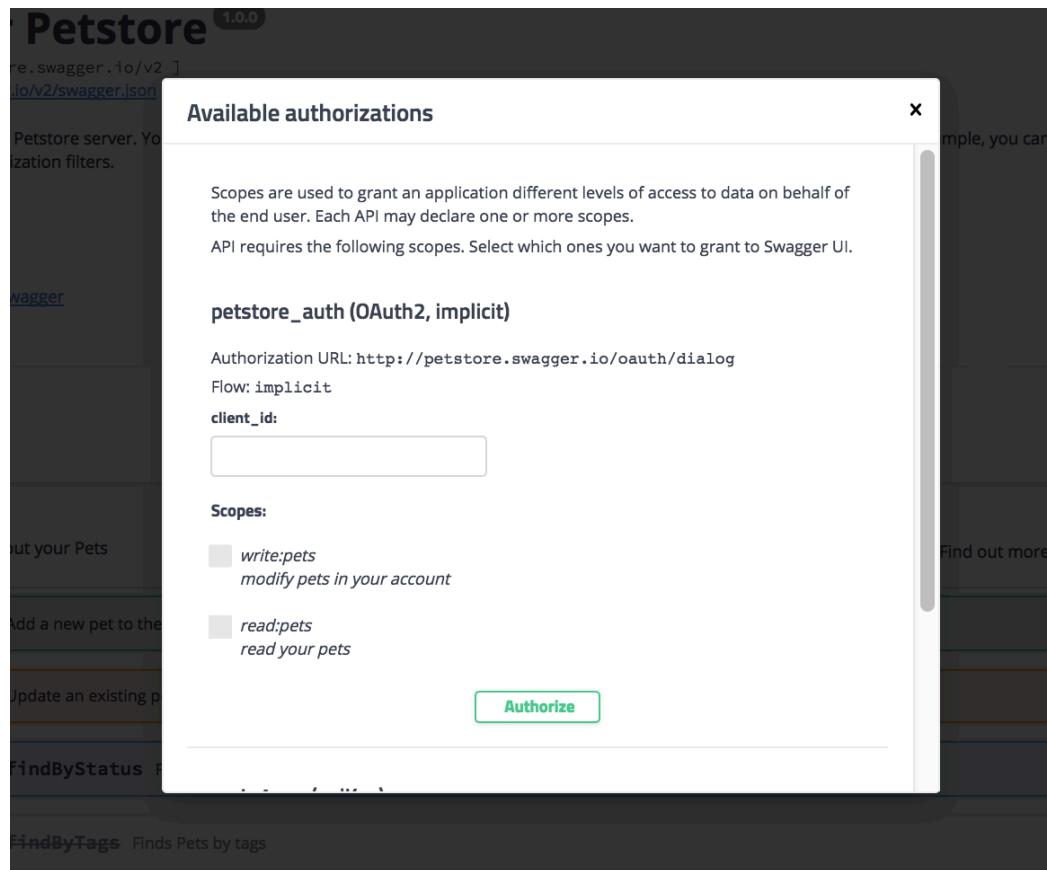
Some designers criticize Swagger UI's expandable/collapsible output as being dated. At the same time, developers find the one-page model attractive and like the ability to zoom out or in for details. By consolidating all endpoints on the same page in one view, users can take in the whole API at a glance. 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 for your API.

Activity 4a: Explore Swagger UI through the Petstore Demo

Let's get some hands-on experience with Swagger UI using the Petstore demo.

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, first you would normally authorize Swagger by clicking **Authorize** and completing the right information required in the Authorization modal.



The Petstore example has an OAuth 2.0 security model. However, the Petstore authorization modal is just for demo purposes. There isn't any real code authorizing those requests, so you can simply close the Authorization modal.

2. Expand the **Pet** endpoint.
3. Click **Try it out**.

The screenshot shows the Swagger UI interface for a 'pet' resource. At the top, it says 'pet Everything about your Pets'. Below that, a green button labeled 'POST' is followed by the endpoint '/pet'. A tooltip for this endpoint states: 'Add a new pet to the store'. To the right of the endpoint is a link to 'Find out more: http://swagger.io'.

The main area is titled 'Parameters'. It contains a table with two columns: 'Name' and 'Description'. There is one row for the parameter '(body)', which is marked as required ('body * required'). The description for '(body)' is 'Pet object that needs to be added to the store'. Below this is a 'Example Value Model' section containing a JSON object:

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

Below the example value model is a 'Parameter content type' dropdown set to 'application/json'.

At the bottom, there are sections for 'Responses' and 'Response content type' (set to 'application/xml').

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 repeated) whole number. Change the second `name` value to something you'd recognize (your pet's name — `Bentley`).
5. Click **Execute**.

This screenshot shows the same 'pet' resource and POST method as the previous one, but with a focus on the request body. The 'body' parameter's example value model is now being edited. The 'id' value has been changed to '193844' and the 'name' value has been changed to 'Bentley'. A red 'Cancel' button is visible at the bottom left of the edit area.

At the bottom of the screen, there is a large blue 'Execute' button with a hand cursor icon pointing to it.

Swagger UI submits the request and shows the curl that was submitted. The Responses section shows the response. (If you select JSON rather than XML in the “Response content type” dropdown box, you can specify that JSON is returned rather than XML.)

The screenshot shows the Swagger UI interface for a POST request to the '/pet' endpoint. The 'Responses' tab is active, and the 'Response content type' dropdown is set to 'application/json'. The 'Curl' section displays a command to make a POST request to 'http://petstore.swagger.io/v2/pet' with specific headers and a JSON payload. The 'Request URL' is also shown as 'http://petstore.swagger.io/v2/pet'. Below this, the 'Server response' section shows a 200 status code with the 'Undocumented' label. The 'Details' table has two rows: one for the 'Code' 200 and another for the 'Response body' which contains a JSON object representing a pet.

Code	Details
200 Undocumented	Response body <pre>{ "id": 193844, "category": { "id": 0, "name": "string" }, "name": "Bentley", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }], "status": "available" }</pre>

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.

Other tools for reading OpenAPI spec

There are other tools besides Swagger UI that can parse your OpenAPI specification document. Some of these tools include [Restlet Studio](#), [Apiary](#), [Apigee](#), [Lucybot](#), [Gelato](#), [Readme.io](#), [swagger2postman](#), [swagger-ui responsive theme](#), [Postman Run Buttons](#) 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 a 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](#). This allows you to expand your tool horizons even wider. (RAML and API Blueprint are alternative specifications to Swagger: they’re not as popular, but the logic of the specifications is similar. And if you’re using a platform like Mulesoft or Apiary, you might want to use the specification for which that platform is optimized.)

Customizing Swagger UI

You might be concerned that Swagger UI outputs look similar. 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 spliced in a reference to Bootstrap so that I could have pop-up 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'd probably need web development skills.)

Downsides to OpenAPI and Swagger UI

The first time I created a Swagger UI output for an API doc project, the project manager loved it. He and others quickly embraced the Swagger output in place of the PowerPoint slides (the previous demo tool) and promoted it among the field engineers and users. The vice president of Engineering even decided that Swagger would be the default approach for documenting all APIs.

Overall, delivering the Swagger output has been a huge feather in my cap at every company, and it establishes an immediate credibility for my technical documentation skills, since few others at companies I've been at know how to create the OpenAPI spec and Swagger UI output.

However, despite Swagger's interactive power to appeal to the "let me try" desires of users, there are some downsides to Swagger and OpenAPI.

First, the OpenAPI specification and Swagger UI's output only cover [reference documentation \(page 78\)](#). OpenAPI provides the basics about each endpoint, including a description, the parameters, a sample request, and a response. It doesn't provide space for a [getting started tutorial \(page 269\)](#), information about how to get [API keys \(page 277\)](#), how to run a [sample app \(page 308\)](#), information about [rate limits \(page 293\)](#), or the hundred other details that go into a user guide for developers.

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. As an analogy, delivering a Javadoc or Doxygen output for a library-based API won't teach users how to actually use your API. You still have to describe scenarios for using a class or method, explain how to set your code up, what to do with the response, how to troubleshoot problems, and so on. In short, you still have to write actual help guides and tutorials.

With OpenAPI in the mix, you now have some additional challenges. You have *two places* where you're describing your endpoints and parameters (potentially in both the Swagger UI 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. (I explore integration strategies in [Integrating Swagger UI with the rest of your docs \(page 241\)](#).)

Another limitation with OpenAPI and Swagger relates to the complexity of your API. [Peter Gruenbaum](#), who has published several tutorials on writing API documentation on Udemy, says that automated tools such as Swagger work best when the APIs are simple. I agree. When you have endpoints that have complex interdependencies and require special setup workflows or other unintuitive treatment, the straightforward nature of Swagger's Try-it-out interface 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 Swagger UI output won't make a lot of sense to users without a detailed tutorial to follow.

Additionally, 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 like Swagger 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.

If your API executes orders for supplies or makes other transactions, it can be even more challenging. For these scenarios, I recommend setting up sandbox or test accounts for users. 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.

Also, you might run up against CORS restrictions in executing API calls. Not all APIs will accept requests executed from a web page. If the calls aren't executing, open the JavaScript Console and check whether CORS is blocking the request. If so, you'll need to ask developers to make adjustments to accommodate requests initiated from JavaScript on web pages. See [CORS Support](#) for more details.

Finally, I found that only endpoints with simple request body parameters tend to work in Swagger. Another API I had to document included requests with request body parameters that were hundreds of lines long (the request body was used to configure an API server). With this sort of request body parameter, Swagger UI's display fell short of being usable. The team reverted to much more primitive approaches (such as tables and Excel spreadsheets) for listing all of the parameters and their descriptions.

Some consolations

Despite the shortcomings of OpenAPI, I still highly recommend it for describing your API. OpenAPI is quickly becoming a way for more and more tools (from Postman Run buttons to nearly every API platform) to easily 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.

Swagger UI definitely provides a nice visual shape for an API. You can easily see all the endpoints and their parameters (like a quick-reference guide). Based on this framework, you can help users grasp the basics of your API.

Additionally, I 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: "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 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 if your API has fairly simple parameters, without many interdependencies between endpoints, and if it's practical to explore the API without making the user's data problematic, OpenAPI and Swagger UI can be a powerful complement to your 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 Swagger UI, 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.

Resources and further reading

See the following resources for more information on OpenAPI and Swagger:

- [API Transformer](#)
- [APIMATIC](#)
- [Carte](#)
- [Swagger editor](#)
- [Swagger Hub](#)
- [Swagger Petstore demo](#)
- [Swagger Tools](#)
- [Swagger UI tutorial \(page 214\)](#)
- [OpenAPI specification tutorial \(page 160\)](#)
- [Swagger/OpenAPI specification](#)

- [Swagger2postman](#)
- [Swagger-ui Responsive theme](#)
- [Swagger-ui](#)
- [Undisturbed REST: A Guide to Designing the Perfect API](#), by Michael Stowe

To see a presentation that covers the same concepts in this article, see <https://goo.gl/n4Hvtq>.

For an excellent overview and comparison of these three REST specification formats, see [Top Specification Formats for REST APIs](#) by Kristopher Sandoval on the Nordic APIs blog.

Working in YAML

Before we dive into the steps of the [OpenAPI Tutorial \(page 160\)](#), it will help to have a better grounding in YAML, since this is the most common syntax for the specification document. (You can also use [JSON \(page 59\)](#), 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
    - TVaml        # 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 (JSON 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 later).

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. Here 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 thing 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": 42
  },
  {
    "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 is more tricky sometimes 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 maybe 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 (e.g., `yamlfile.api` or `yamlfile.application_programming_interface`), 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. See my [YAML tutorial in the context of Jekyll](#) for more details.

OpenAPI 3.0 tutorial overview

In this section, 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 into interactive documentation using Swagger UI.

General resources for learning the OpenAPI specification

Learning the [OpenAPI specification](#) will take some time. As an estimate, plan about two weeks of immersion, working with a specific API in the context of the specification before you become comfortable with it. 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 about 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 3.0.1](#) 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

Rather than try to reproduce the material in the guides or specification, in my OpenAPI/Swagger tutorial here, I give you a crash course in manually creating the OpenAPI specification document. I use a real API for context, and also provide detail about how the specification fields get rendered in Swagger UI.

[Swagger UI](#) is one of the most popular display frameworks for the OpenAPI specification. (The spec alone does nothing — you need some tool that will read the spec's format and display the information.) There are many display frameworks that can parse and display information in an OpenAPI specification document — just like many component content management systems can read and display information from DITA files.

However, Swagger UI is one of the most common and popular ways to render your specification document. Swagger UI is sponsored by SmartBear, the same company that is heavily invested in the [OpenAPI initiative](#) and which develops [Swaggerhub \(page 224\)](#). Their tooling is almost always in sync with the latest spec features. Swagger UI an actively developed and managed open-source project.

By showing you how the fields in the spec appear in the Swagger UI display, I hope the specification objects and properties will take on more relevance and meaning. Just keep in mind that Swagger UI's display is *just one possibility* for how the spec information might be rendered.

Terminology for Swagger and OpenAPI

Before continuing, I want to clarify a few terms for those who may be unfamiliar with the OpenAPI/Swagger landscape:

- Swagger was the original name of the OpenAPI spec, but the spec 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.
- Smartbear is the company that maintains and develops the open source Swagger tooling (Swagger Editor, Swagger UI, Swagger Codegen, and others). They do not own the [OpenAPI specification](#), as this [initiative](#) is driven by the Linux Foundation. The OpenAPI spec’s development is driven by [many companies and organizations](#).
- The Swagger YAML file that you create to describe your API is called the “OpenAPI specification document” or the “OpenAPI document.”

Now that I’ve cleared up those terms, let’s continue. (For other terms, see the [Glossary \(page 462\)](#).)

Terminology to Describe JSON/YAML

Let’s clear up some other terminology before we get started. The specification document in my OpenAPI tutorial uses YAML (which I introduced briefly [here \(page 156\)](#)), 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 here. You will see that the specification documentation 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](#).)

Note that YAML refers to data structures with 3 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.

So that we’re on the same page with terms, 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 spec 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. Confusingly, in the description for each field, the spec frequently refers to the *field* as a *property*, so I’m not sure why they chose to use “field” in subheadings and column titles.

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 that looks like in JSON:

```
{  
  "demographics": [  
    "population",  
    "land",  
    "rivers"  
  ]  
}
```

Hopefully those brief examples will help align us with the terminology used in the tutorial.

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 here](#). 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 8 steps. Each step corresponds with one of the root-level objects in the OpenAPI document.

- [Step 1: openapi object \(page 165\)](#)
- [Step 2: info object \(page 168\)](#)
- [Step 3: servers object \(page 170\)](#)
- [Step 4: paths object \(page 173\)](#)
- [Step 5: components object \(page 181\)](#)
- [Step 6: security object \(page 200\)](#)
- [Step 7: tags object \(page 206\)](#)
- [Step 8: externalDocs object \(page 209\)](#)

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 \(page 181\)](#) to fully explain how to reference a schema in one object and add a reference pointer to the full definition.

Remember that the specification document alone does nothing with your content. Other tools are required to read and display the spec document, or to generate client SDKs from it.

My preferred tool for parsing and displaying information from the specification document is [Swagger UI](#), but many other tools can consume the OpenAPI document and display it in different ways. See the [list of tools on Swagger here](#). Consider the screenshots from Swagger UI as one example of how the fields from the spec might be rendered.

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.

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 OpenAPI spec rendered with Swagger UI in the following links:

- [Swagger UI with OpenWeatherMap API](#)
- [Embedded Swagger with OpenWeatherMap API \(page 223\)](#)

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](#) to convert it automatically. You can also use APIMATIC to transform your specification document into a number of other outputs, such as RAML, Postman, or API Blueprint.

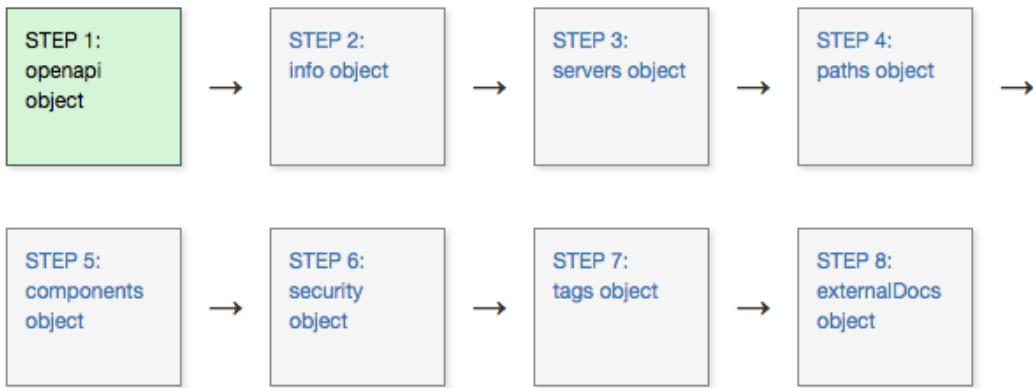
To see the difference between the 2.0 and the 3.0 code, you can copy these code samples to separate files and then use an application like [Diffmerge](#) to highlight the differences. The Readme.io blog has a nice post that provides [A Visual Guide to What's New in Swagger 3.0](#).

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.

Step 1: The `openapi` object (OpenAPI tutorial)



OpenAPI tutorial overview

Before diving into the first step of the OpenAPI tutorial here, read the [OpenAPI tutorial overview \(page 160\)](#) (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 ([page 32](#)) in this course.
- The OpenAPI tutorial shows how the spec information gets populated in [Swagger UI](#).
- The OpenAPI tutorial is a subset of the information in both the [OpenAPI specification](#) and the [OpenAPI specification commentary](#).
- The OpenAPI tutorial covers the 3.0 version of the OpenAPI spec, which is the latest version.

The root-level objects in OpenAPI spec

There are 8 objects at the root level in the OpenAPI 3.0 spec. There are many nested objects within these root level objects, but at the root level, there are just these 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 8 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. For more details on the terms, see [What Is the Difference Between Swagger and OpenAPI?](#)

Swagger Editor

As you work on your specification document, 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 [Atom editor](#)) 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 \(page 224\)](#), 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.

Add the `openapi` object

Go to 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 `3.0.1`.

```
openapi: "3.0.1"
```

Until you add more information in here, you’ll see error messages and notes such as “No operations defined in spec!” That’s okay — in the next step you’ll start seeing more info.

3.0 was released in July 2017, and 3.0.1 was released in December 2017. 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.

Appearance in Swagger UI

There’s not much to the `openapi` object, and right now there’s not enough content for the spec to validate. But when you later render your specification document through the Swagger UI display, you’ll see that an “OAS3” tag will appear to the right of the API name.

The screenshot shows the Swagger UI interface for the OpenWeatherMap API. At the top, there's a green header bar with the 'swagger' logo and navigation links for '/learnapidoc/docs/rest_api_s' and 'Explore'. The main title 'OpenWeatherMap API' is displayed prominently. Below it, a green button indicates the API version is 2.5 and follows the OAS3 specification. An arrow points from the explanatory text below to this button. The main content area describes the API's capabilities, including current weather, daily forecasts, and historical data. It also notes that parameters are optional and provides links to terms of service, the API website, and attribution information.

/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.

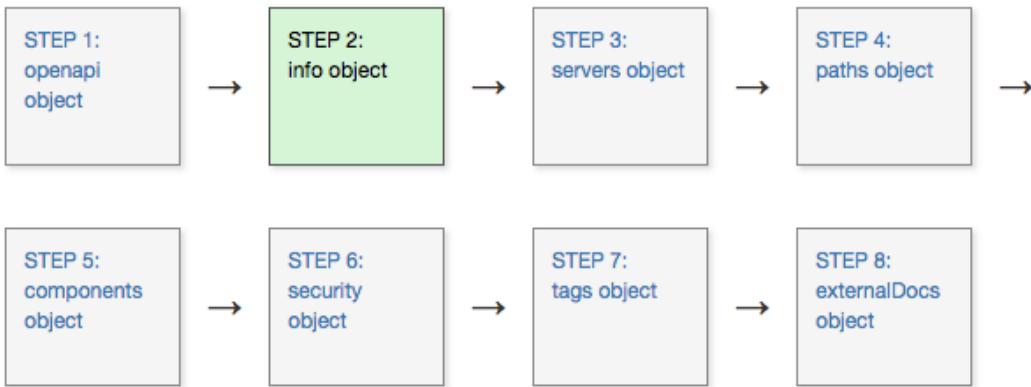
Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

[Terms of service](#)
[OpenWeatherMap API - Website](#)
[CC Attribution-ShareAlike 4.0 \(CC BY-SA 4.0\)](#)
[API Documentation](#)

On the backend, Swagger UI uses the 3.0.1 version of the spec to validate your content.

In the above screenshot, the “2.5” version refers to the version of the API here, not the version of the OpenAPI spec.

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:

```
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"
  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/license"
```

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 escape colons with a backslash or enclose the `description` value in quotation marks. You can use single or double quotation marks for the property values. (If you enclose the values in quotation marks, syntax highlighters can display better color coding between the properties and values.)

View the Appearance in Swagger UI

At this point, go ahead and paste this above code along with the `openapi` object from the previous step into the [Swagger Editor](#). You'll see some rendering errors (because the specification document doesn't yet have any `path` objects), but the content will still appear. (Just hide the errors section at the top for now).

In the Swagger UI display, the `info` object's information appears at the top:

```

1 openapi: "3.0.1"
2
3 info:
4   title: "OpenWeatherMap API"
5   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."
6   version: "2.5"
7   termsOfService: "https://openweathermap.org/terms"
8   contact:
9     name: "OpenWeatherMap API"
10    url: "https://openweathermap.org/api"
11    email: "some_email@gmail.com"
12   license:
13     name: "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)"
14     url: "https://openweathermap.org/license"

```

Errors

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.

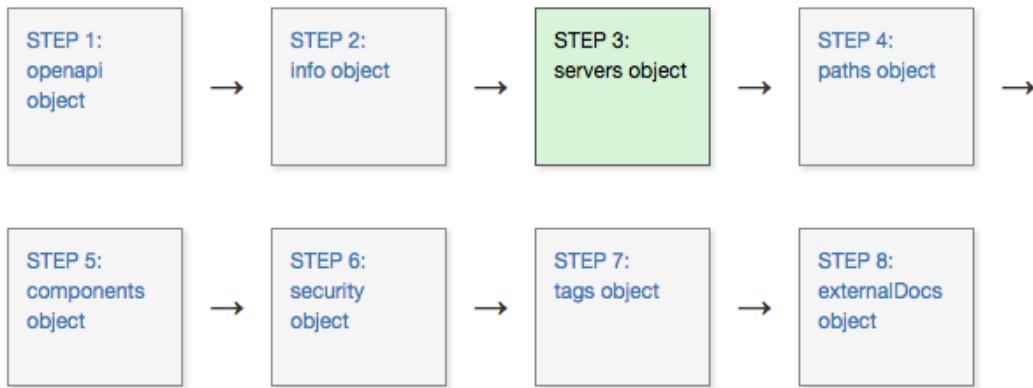
Note: All parameters are optional, but you must select at least one parameter. Calling the API by city ID (using the `id` parameter) will provide the most precise location results.

[Terms of service](#)
[OpenWeatherMap API - Website](#)
[Send email to OpenWeatherMap API](#)
[CC Attribution-ShareAlike 4.0 \(CC BY-SA 4.0\)](#)

No operations defined in spec!

In the `description` property, 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
```

In Swagger UI, here's how the servers appear to users with multiple server URLs:

The screenshot shows the Swagger UI interface. At the top right, there is a green "Authorize" button with a lock icon. Below it, the word "Server" is displayed in bold. A dropdown menu is open, showing the URL "http://api.openweathermap.org/data/2.5/" followed by a dropdown arrow. An arrow points from the text "In Swagger UI, here's how the servers appear to users with multiple server URLs:" to this dropdown menu. Below the dropdown, there is a section titled "Current Weather Data" with the subtitle "Get current weather details". Underneath, a "GET /weather" operation is listed with the description "Call current weather data for one location". A lock icon is next to the method. Below the operation, there is a detailed 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."

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` (page 173) object's operation object. The locally declared servers URL will override the global servers URL.

See “[Overriding Servers](#)” in the “API Server and Base URL” page for more details.

View the Appearance in Swagger UI

Paste the `servers` object (the first code sample above showing just one `url`) into your Swagger Editor, adding to the code you already have there. Swagger UI will look as follows.

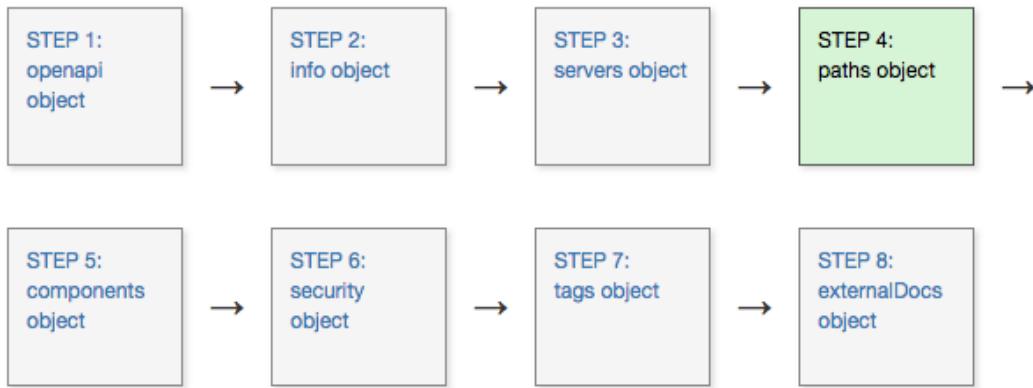
The screenshot shows the Swagger Editor interface. On the left, the API specification is displayed in JSON format:

```
1 openapi: "3.0.1"
2
3 info:
4   title: "OpenWeatherMap API"
5   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."
6   version: "2.5"
7   termsOfService: "https://openweathermap.org/terms"
8   contact:
9     name: "OpenWeatherMap API"
10    url: "https://openweathermap.org/api"
11    email: "some_email@gmail.com"
12   license:
13     name: "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)"
14     url: "https://openweathermap.org/license"
15
16 servers:
17 - url: "http://api.openweathermap.org/data/2.5/"
```

On the right, the API documentation is shown under the heading "OpenWeatherMap API". It includes a note about the API's functionality, a "Note" section, and links to the API's website, license, and terms of service. A "Server" section at the bottom contains a dropdown menu with the URL "http://api.openweathermap.org/data/2.5/".

Progress in Swagger UI with servers object

Step 4: The paths object (OpenAPI tutorial)



The `paths` object contains the meat of your API information. The `paths` object has a number of 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 with more of a deep dive.

Paths objects

My preferred term is “endpoint” rather than “path,” but to be consistent with the terminology of the openAPI spec, I refer to them as “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 \(page 89\)](#) of the API reference tutorial methods.)

Start by listing the paths (endpoints) and their allowed operations (methods). For the `weather` endpoint in the OpenWeatherMap API, there is just 1 path (`/weather`) and one operation (`get`) for that path:

```

paths:
  /weather:
    get:
  
```

Operation Objects

The operation object (`get` in the code above) contains a number of potential properties and objects:

- `tags` : A tag to organize the path under when displayed in the Swagger UI. Swagger UI will organize or group endpoints under tag headings.
- `summary` : A brief overview of the path. Swagger UI displays 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 body parameters, which are instead detailed in the `requestBody` object. The `parameters` object can also include a `reference object` that simply contains a pointer to the description in the `components` object (this is explained in [step 5 \(page 181\)](#)).
- `requestBody` (object): The request body parameter details for this path. The `requestBody` object can also include a `reference object` that simply contains a pointer to the description in the `components` object (explained in [step 5 \(page 181\)](#)).
- `responses` (object): Responses provided from requests with this path. The `responses` object can also include a `reference object` that simply 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 simply 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 \(page 200\)](#).
- `servers` (object): A servers object that might differ for this path than the [global servers object \(page 170\)](#).

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 detail 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:
```

Now we can remove a few unnecessary fields that we don’t need:

- There's no need to include `requestBody` object here because none of the OpenWeatherMap API paths contain request body parameters.
- There's no need to include the `servers` object because the paths just use the same global `servers` URL that we [defined globally \(page 170\)](#) 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 \(page 200\)](#)).
- There's no need to include `deprecated` because none of the paths are deprecated.
- There's no need to include `callbacks` because our paths don't use callbacks.

As a result, we can reduce the number of relevant fields as follows:

```
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 ([page 175](#)) and the `responses` object ([page 175](#)).

Parameters object

The `parameters` object contains an array (a list designated with dashes) with these properties:

- `name` : Parameter name.
- `in` : Where the parameter appears. Possible values are `header`, `path`, `query`, or `cookie`. (Request body parameters 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.

Responses object

The other substantial property in the operations object is the `responses` object. For the `responses` property, we typically just reference a full definition in the `components` object, so I'll cover the `responses` object in the next section — [Step 5: The components object \(page 181\)](#). (There's already too much detail in this step as is.)

For now, so that 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

Here's the `paths` object defined for the OpenWeatherMap API that includes the `parameters` and our placeholder `responses` object:

```
openapi: "3.0.1"

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 comma; use ISO 3166 country codes."
        - 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. 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.*"
        - name: lat
          in: query
          description: "**Latitude**. *Example: 35*. The latitude coordinate of the location of your interest. Must use with `lon`."
        - name: lon
          in: query
          description: "**Longitude**. *Example: 139*. Longitude coordinate of the location of your interest. Must use with `lat`."
        - name: zip
          in: query
          description: "**Zip code**. Search by zip code. *Example: 95050,u
```

```
  s*. Please note if country is not specified then the search works for 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 units parameter, 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 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"

  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

Paste the above code into Swagger UI (adding to the code you added in the previous tutorials). Swagger UI displays the `paths` object like this:

The screenshot shows the Swagger Editor interface. On the left, there is a code editor window displaying the OpenAPI specification. On the right, there is a detailed view of the 'Current Weather Data' endpoint. The endpoint is defined as a GET request to '/weather'. The description for this endpoint is: 'Call current weather data for one location'. Below the description, there is a 'Parameters' section. It lists several parameters: 'q' (string, query), 'id' (string, query), 'lat' (string, query), and 'lon' (string, query). Each parameter has a detailed description and a 'Try it out' button.

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 call &mash; you use only the parameters you want for the call you're making. (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.

Note that in Swagger UI's display, each path is collapsed 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 isn't part of the OpenAPI spec. Swagger UI has more than 20 different parameters of its own that control its display. For example, if you don't want the `Models` section to appear, add the parameter `defaultModelsExpandDepth: -1,` in your Swagger UI file.

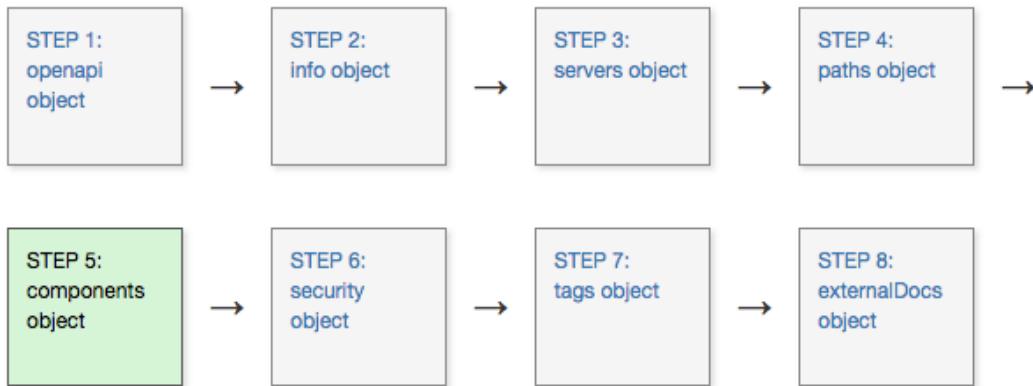
Note about parameter dependencies

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 by zip code. Although the parameters are optional, you have to use at least one parameter. Also, if you use the latitude parameter, you must also use the longitude parameter, as they're a pair. The OpenAPI spec can't programmatically reflect that structured logic, so you just 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 `$ref`) that refers to a specific definition in the `components object`. See [Using \\$ref](#) for more details on this standard JSON reference property.

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:

```
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:
                    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 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 cit
ies in parameter &mdash; just separate them by commas. The limit of location
s is 20. *Note: A single ID counts as a one API call. So, if you have city I
Ds. it's treated as 3 API calls.*"
  schema:
    type: string

  lat:
    name: lat
    in: query
    description: "**Latitude**. *Example: 35*. The latitude coordinate of t
he location of your interest. Must use with `lon`."
  schema:
    type: string

  lon:
    name: lon
    in: query
    description: "**Longitude**. *Example: 139*. Longitude coordinate of th
e 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 if country is not specified then the search works for 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 units parameter, format i
s `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 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"

```

Replace the existing `paths` object in your code in Swagger Editor with the above code sample, and include the new `components` object, and observe that the rendered display still looks the same.

Re-using response objects

In the [step 4 \(page 173\)](#), when we described the `responses` 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 deep 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 \(page 173\)](#)), 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:

```
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 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 one long description that contains all the hierarchy reflected. One challenge 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 objects remain shallow and you won't get lost in a sea of confusing sublevels.

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:

```
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 -- see previous section 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: 212797  
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 deviation
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: 289.8199999999999
```

```
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, you can use it within `components` too.

Notice how the schema definition includes an `example` property for element? Swagger UI will take this `example` and use it to automatically 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 example 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 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. (Note that 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`

A number of [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. Download [Stoplight](#) and use the **Generate JSON** feature to have Stoplight automatically create the OpenAPI schema description. Here's a short (silent) video showing how to do this:

Go to https://idratherbewriting.com/learnapidoc/pubapis_openapi_step5_components_object.html to view this content.

Basically, you copy the JSON response you want to document into the Stoplight Editor. Then you click **Generate JSON**. Then you go into the code view and copy the JSON. Then convert the JSON to YAML using an [online converter](#). For more details, see [Stoplight — visual modeling tools for creating your OpenAPI spec \(page 233\)](#).

The only catch is that Stoplight uses OpenAPI 2.0, not 3.0. You might need to use [API Transformer](#) to convert the 2.0 schema output to 3.0. Even so, this approach can save you a lot of time.

View the Appearance in Swagger UI

Copy the lengthy code sample (the one under [Responses object with components documentation \(page 189\)](#)) and paste it into Swagger Editor, adding to what's already there. Since you already have a `components` object in Swagger Editor, you need to merge the two so that there's just one instance of `components`. For example:

```
components:  
  parameters:  
    ...  
  schemas:  
    ...
```

The screenshot shows the Swagger Editor interface with the title bar "Swagger Editor" and menu items "File", "Edit", "Generate Server", and "Generate Client". The main area is divided into two sections: "Responses" and "Example Value Model".

Responses section:

Code	Description	Links
200	Successful response	No links

A dropdown menu under "Description" is set to "application/json". A note below it says "Controls Accept header."

Example Value Model section:

```
{
  "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, note how the `example` definitions have been dynamically pulled together from 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:

Code	Description																		
200	<p>Successful response</p> <p>application/json</p> <p>Controls <code>Accept</code> header.</p> <p>Example Value Model ←</p> <div style="background-color: #f0f0f0; padding: 10px;"><p>Successful response ↴ {</p><table><tr><td><code>coord</code></td><td><code>Coord > {...}</code></td></tr><tr><td><code>weather</code></td><td><code>> [...]</code></td></tr><tr><td><code>base</code></td><td><code>string</code> <i>example: cmc stations</i> Internal parameter</td></tr><tr><td><code>main</code></td><td><code>Main > {...}</code></td></tr><tr><td><code>visibility</code></td><td><code>integer</code> <i>example: 16093</i> Visibility, meter</td></tr><tr><td><code>wind</code></td><td><code>Wind > {...}</code></td></tr><tr><td><code>clouds</code></td><td><code>Clouds > {...}</code></td></tr><tr><td><code>rain</code></td><td><code>Rain > {...}</code></td></tr><tr><td><code>snow</code></td><td><code>Snow > {...}</code></td></tr></table></div>	<code>coord</code>	<code>Coord > {...}</code>	<code>weather</code>	<code>> [...]</code>	<code>base</code>	<code>string</code> <i>example: cmc stations</i> Internal parameter	<code>main</code>	<code>Main > {...}</code>	<code>visibility</code>	<code>integer</code> <i>example: 16093</i> Visibility, meter	<code>wind</code>	<code>Wind > {...}</code>	<code>clouds</code>	<code>Clouds > {...}</code>	<code>rain</code>	<code>Rain > {...}</code>	<code>snow</code>	<code>Snow > {...}</code>
<code>coord</code>	<code>Coord > {...}</code>																		
<code>weather</code>	<code>> [...]</code>																		
<code>base</code>	<code>string</code> <i>example: cmc stations</i> Internal parameter																		
<code>main</code>	<code>Main > {...}</code>																		
<code>visibility</code>	<code>integer</code> <i>example: 16093</i> Visibility, meter																		
<code>wind</code>	<code>Wind > {...}</code>																		
<code>clouds</code>	<code>Clouds > {...}</code>																		
<code>rain</code>	<code>Rain > {...}</code>																		
<code>snow</code>	<code>Snow > {...}</code>																		

Descriptions appear in the Model

The Models section – why it exists, how to hide it

You'll also notice a "Models" section at the end. 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 define each object in `components` separately. As a result, the Models section looks like this:

The screenshot shows the 'Models' section of the Swagger UI. It contains six items, each with a title and a right-pointing arrow: '200 >', 'Coord >', 'Weather >', 'Main >', 'Wind >', and 'Clouds >'. A vertical scroll bar is visible on the right side of the list.

The Models section is now in the latest version of Swagger UI. I'm not really sure why the Models section appears at all, actually. 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.

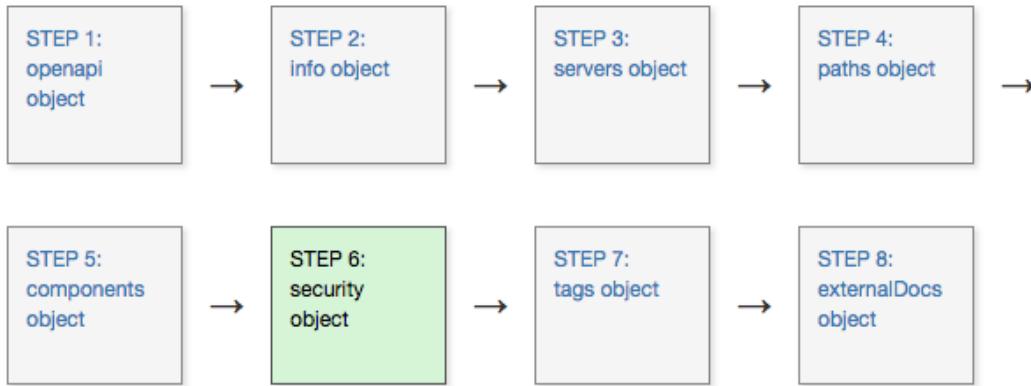
You don't need this Models section in Swagger UI because both the request and response sections of Swagger UI provide a "Model" link that lets the user toggle to this view. For example:

You might confuse users by including the Models section. To hide Models, add the parameter `defaultModelsExpandDepth: -1` parameter in your Swagger UI project. I provide a [Swagger UI tutorial \(page 0\)](#) in an upcoming section in this course. See the [Swagger UI parameters documentation](#) for more details on how to configure Swagger UI.

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 the [step 6 \(page 200\)](#).

Step 6: The security object (OpenAPI tutorial)



Swagger UI provides a “Try it out” feature that lets users submit actual requests. To actually 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 a number of different security approaches to authorize requests. I explored the most common authorization methods in [Documenting authentication and authorization requirements \(page 277\)](#). 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 \(page 0\)](#) or another method, you’ll need to read the [Security Scheme information](#) for details on how to configure it. However, all the security methods largely 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 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, we add a `security` object that defines the global method for our 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 \(page 173\)](#). 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`.

Referencing the security scheme in components

In the [components object \(page 181\)](#), we add a `securitySchemes` object that defines details about the security scheme we're using:

```
components:
  ...
  securitySchemes:
    app_id:
      type: apiKey
      description: API key to authorize requests. If you don't have an OpenWeatherMap API key, use `fd4698c940c6d1da602a70ac34f0b147`.
      name: appid
      in: query
```

Properties you can use for each item in the `securitySchemes` object include the following:

- `type` : The type of authorization — `apiKey`, `http`, `oauth2`, or `openIdConnect`.
- `description` : A description of your security method. In Swagger UI, this description appears in the Authorization modal (see 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, insert the following 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:
...

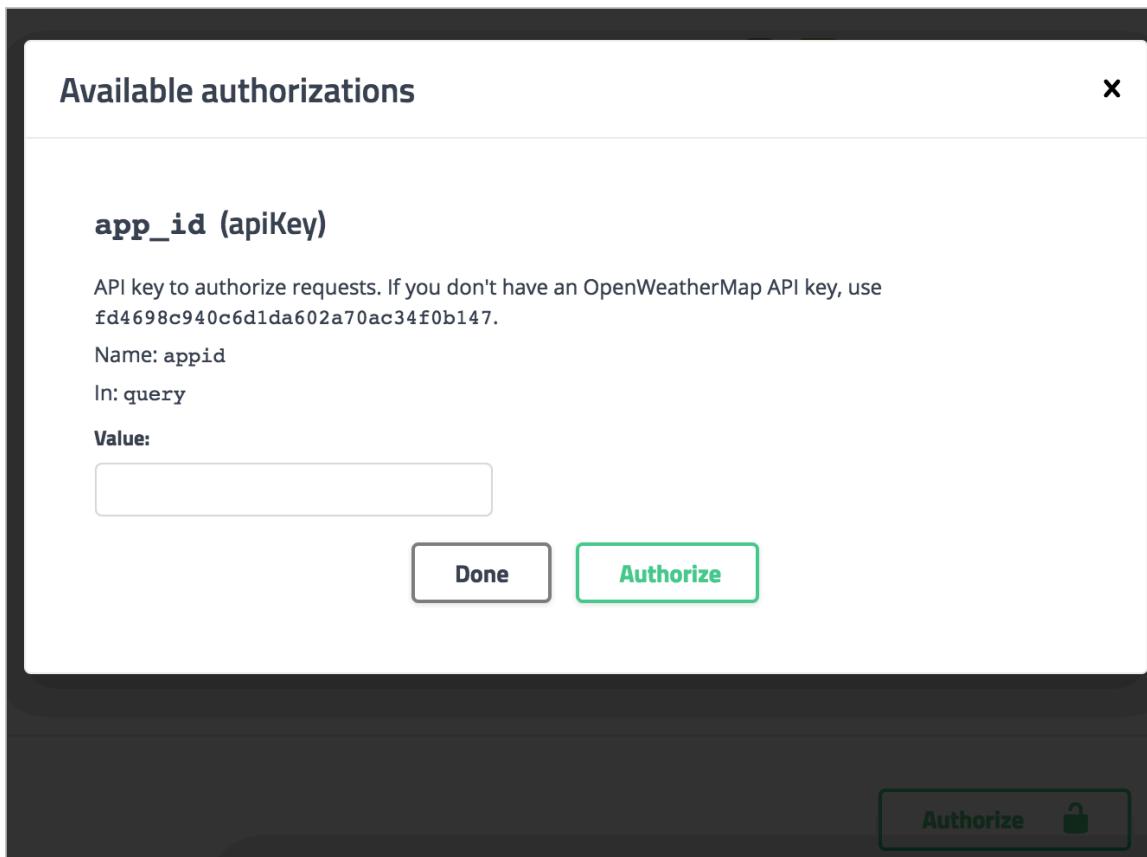
securitySchemes:
app_id:
  type: apiKey
  description: API key to authorize requests. If you don't have an OpenWeatherMap API key, use `fd4698c940c6d1da602a70ac34f0b147`.
  name: appid
  in: query
```

You'll see an "Authorize" button appear in the Swagger UI display.

The screenshot shows the Swagger Editor interface. On the left, the API specification code is visible, including the `security` and `components` sections. On the right, the Swagger UI displays the API documentation for the OpenWeatherMap API. The title is "OpenWeatherMap API 2.5 OAS3". Below the title, there is a brief description of the API's capabilities. A note states that all parameters are optional, but at least one must be selected. It also notes that calling by city ID provides the most precise location results. Below the note are links for "Terms of service", "OpenWeatherMap API - Website", "Send email to OpenWeatherMap API", and "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)". At the bottom right of the UI, there is a yellow button labeled "Authorize" with a lock icon.

Adding security information into the spec

When you click the Authorization button, the `description` and other security details appear:



After users enter an API key and clicks **Authorize**, the authorization method is set for as many requests as they want to make. Only when users refresh the page does the authorization session expire.

Test out how authorization works

Now that we've added authorization, let's test it out. In the Swagger Editor (the right pane), click the **Authorize** button, paste the sample API key shown in the description into the **Value** field and click **Authorize**. Then click **Close** to close the authorization modal.

Then 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 (**e.g., 95050,us**), and then click **Execute**.

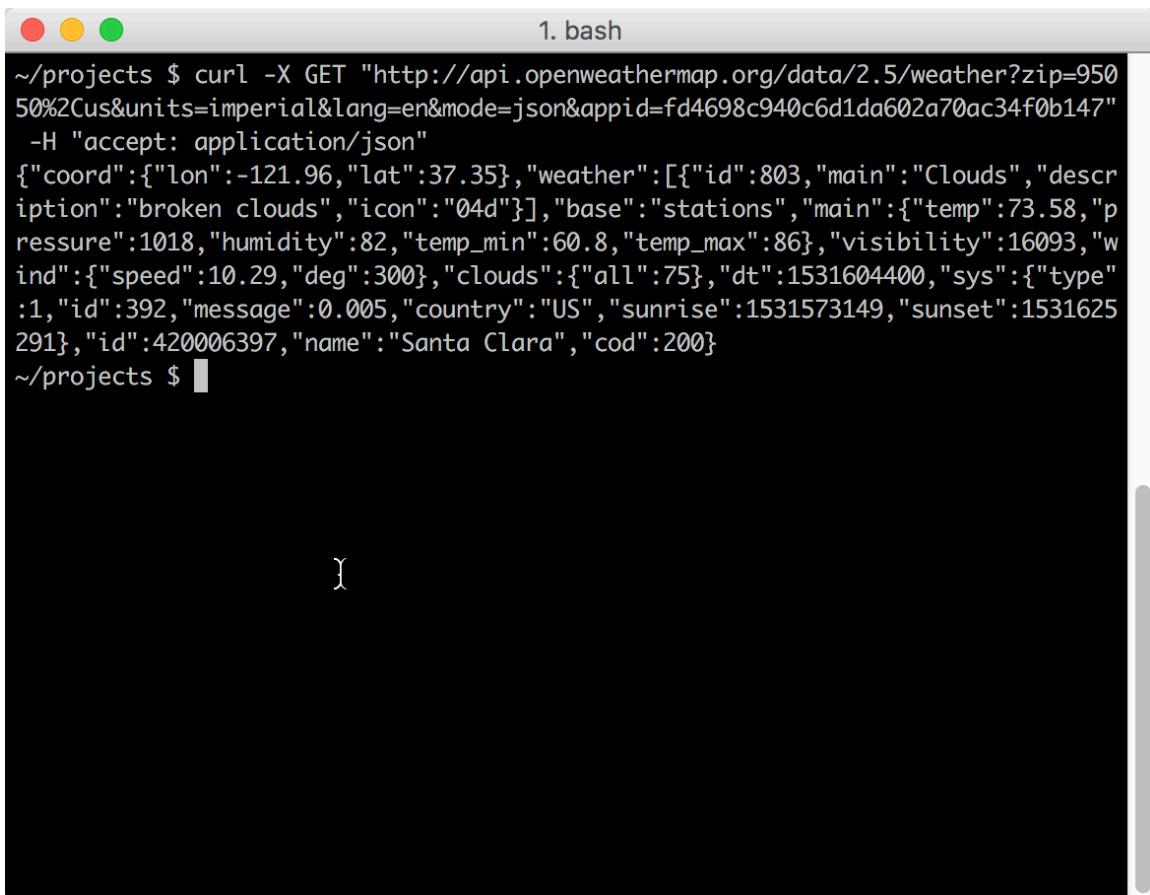
When you execute the request, Swagger UI shows you the [curl request \(page 0\)](#) that is submitted. For example, after executing a weather request, the curl is as follows:

```
curl -X GET "https://api.openweathermap.org/data/2.5/weather?zip=95050&unit=s=imperial&lang=en&mode=json&appid=fd4698c940c6d1da602a70ac34f0b147" -H "accept: application/json"
```

The **&appid=fd4698c940c6d1da602a70ac34f0b147** indicates that the API key is being included in the query string, so the request will be authorized.

However, in this sample scenario, OpenWeatherMap doesn't seem to allow requests from Swagger Editor, so you'll see "TypeError: Failed to fetch" as the response.

If you copy the curl submitted and paste it into the command line, you'll see a successful response:



The screenshot shows a terminal window titled "1. bash". The command entered is:

```
~/projects $ curl -X GET "http://api.openweathermap.org/data/2.5/weather?zip=95050&units=imperial&lang=en&mode=json&appid=fd4698c940c6d1da602a70ac34f0b147" -H "accept: application/json"
```

The output of the command is a JSON weather forecast for Santa Clara, California, including coordinates, current weather conditions, temperature, humidity, pressure, visibility, wind speed, and sunrise/sunset times.

```
{"coord":{"lon":-121.96,"lat":37.35}, "weather":[{"id":803,"main":"Clouds","description":"broken clouds","icon":"04d"}], "base":"stations", "main":{"temp":73.58,"pressure":1018,"humidity":82,"temp_min":60.8,"temp_max":86}, "visibility":16093, "wind":{"speed":10.29,"deg":300}, "clouds":{"all":75}, "dt":1531604400, "sys":{"type":1,"id":392,"message":0.005,"country":"US","sunrise":1531573149,"sunset":1531625291}, "id":420006397, "name":"Santa Clara", "cod":200}
```

~/projects \$ █

Successful curl response

With the successful message from the command line, you know that Swagger is submitting a successful request but the API server is rejecting it. See the next section for Troubleshooting tips.

Troubleshooting issues

CORS issues:

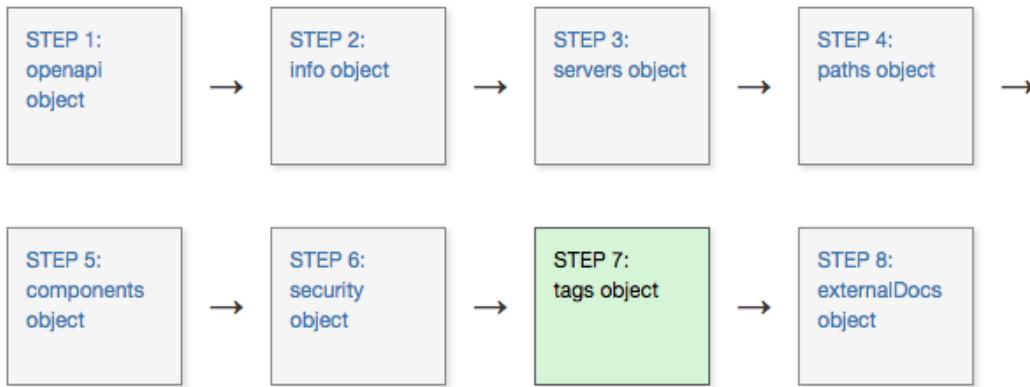
If you have security correctly configured but the requests are being rejected, it could be due to a CORS (cross-origin resource sharing) issue. CORS is a security measure that websites implement to make sure other scripts and processes cannot take their content through requests from remote servers. See [CORS Support](#) in Swagger UI's documentation for details.

If the requests aren't working, open your browser's JavaScript console (in Chrome, View > Developer > Javascript Console) when you make the request and see if the error relates to cross-origin requests. If so, ask your developers to enable CORS on the endpoints.

Host URL issues:

Another reason requests might be rejected is due to the host from your test server. Some APIs (like Aeris Weather) require you to create an App ID that's 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.

Step 7: The tags object (OpenAPI tutorial)



The `tags` object provides a way to group the paths (endpoints) 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 \(page 173\)](#)).

Here's an example of the `tags` object for our OpenWeatherMap API:

```

tags:
  - name: Current Weather Data
    description: "Get current weather details"
  
```

We just have 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.

Tags at the path object level

The `tags` object at the root level should comprehensively list all tags used within the operation objects at each path. Then in each path, you list the tag you want that path grouped under.

For example, in the operations object for the `/current` path, we already used the same tag `Current Weather Data`:

```

paths:
  /weather:
    get:
      tags:
        - Current Weather Data
  
```

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.

The screenshot shows the Swagger Editor interface. On the left, the API definition is displayed in JSON format. On the right, the generated UI is shown. The UI features a 'Server' dropdown set to 'http://api.openweathermap.org/data/2.5/'. A yellow header bar contains the title 'Current Weather Data' and a description 'Get current weather details'. Below this, there is a 'Models' section with three expandable items: 'Successful response', 'Coord', and 'Weather'.

```

41 -   application/json:
42 -     schema:
43 -       $ref: '#/components/schemas/200'
44 -   404:
45 -     description: Not found response
46 -     content:
47 -       text/plain:
48 -         schema:
49 -           title: Weather not found
50 -           type: string
51 -           example: Not found
52 - security:
53 -   - app_id: □
54 -
55 - tags:
56 -   - name: Current Weather Data
57 -     description: "Get current weather details"
58 -
59 - components:
60 -
61 -   parameters:
62 -     q:
63 -       name: q
64 -       in: query
65 -       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."
66 -       schema:
  
```

Tags defined at the root level

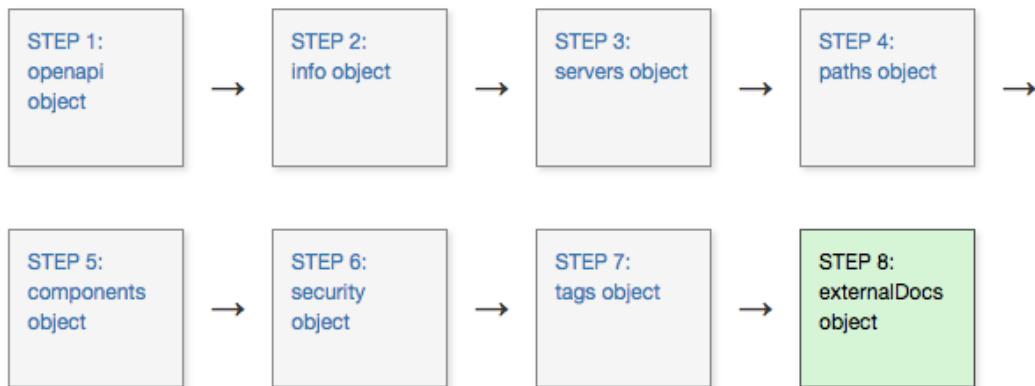
All paths that have the same tag are grouped together in the display. For example, paths that have the `Current Weather Data` tag will be grouped together under the title `Current Weather Data`. Each group title is a collapsible/expandable toggle.

The screenshot shows the Swagger UI interface for a weather API. At the top left, there is a dropdown labeled "Server" containing the URL "http://api.openweathermap.org/data/2.5/". Below this, under the heading "Current Weather Data" (which is expanded), there is a detailed description of the "/weather" endpoint. The description states: "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." To the right of the description is a "Try it out" button, which is highlighted with a red arrow pointing from the "Server" dropdown above. The "Parameters" section is also visible below the main description.

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 \(page 223\)](#) 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 \(page 173\)](#) section in Step 4 with paths.

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:

The screenshot shows the Swagger Editor interface. On the left, the raw OpenAPI YAML code is displayed, showing definitions for security, tags, externalDocs, components, parameters, and schemas. On the right, the rendered API documentation for 'OpenWeatherMap API' is shown. It includes a brief description of the API's purpose, a note about optional parameters, and links to terms of service, API documentation, and licensing information (CC Attribution-ShareAlike 4.0). A yellow box highlights the 'API Documentation' link. At the bottom right of the rendered UI is a green 'Authorize' button with a lock icon.

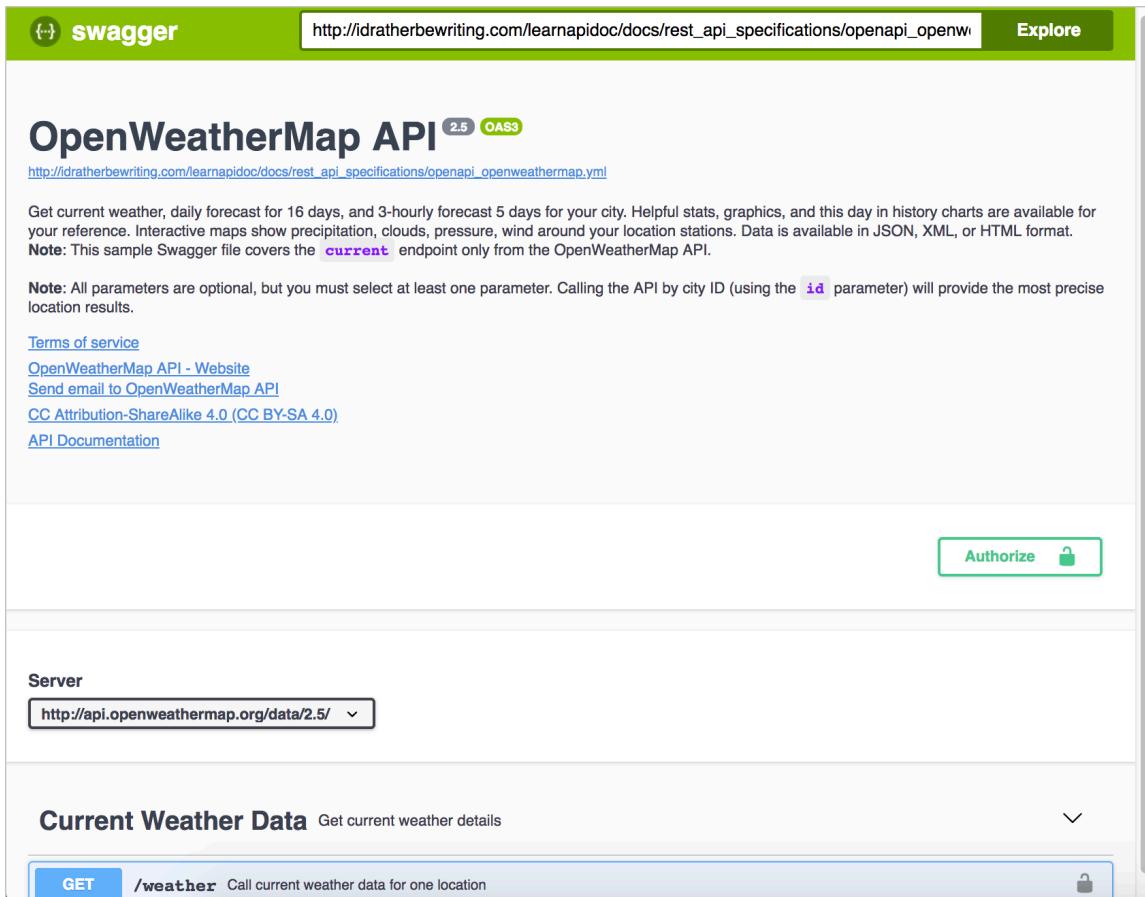
External documentation link

See the related topic, [Integrating Swagger UI with the rest of your docs \(page 241\)](#) for tips on how to integrate your Swagger UI output into your regular documentation.

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: https://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_openweathermap.yml.

Here's the specification document rendered by Swagger UI:



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 to evaluate whether the temperature outside matches the response.

You can actually insert any valid path to an OpenAPI specification document into the “Explore” box in Swagger UI (assuming it’s using a version that supports your version of the spec), and it will display the content. 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 \(page 160\)](#) walked you through 8 steps in building the OpenAPI specification document. Now it's your turn to practice building out an OpenAPI specification document on your own.

Activity 4b: 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 (you can skip creating the [security object \(page 200\)](#)). In this activity, you'll simply 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: openapi object \(page 165\)](#)
 - [Step 2: info object \(page 168\)](#)
 - [Step 3: servers object \(page 170\)](#)
 - [Step 4: paths object \(page 173\)](#)
 - [Step 5: components object \(page 181\)](#)
 - [Step 8: externalDocs object \(page 209\)](#)
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 `response` 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 \(page 137\)](#) 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.

Go each step of the OpenAPI specification tutorial to build out the specification document:

- [Step 1: openapi object \(page 165\)](#)
- [Step 2: info object \(page 168\)](#)
- [Step 3: servers object \(page 170\)](#)
- [Step 4: paths object \(page 173\)](#)
- [Step 5: components object \(page 181\)](#)
- [Step 6: security object \(page 200\)](#)
- [Step 7: tags object \(page 206\)](#)
- [Step 8: externalDocs object \(page 209\)](#)

Validate your specification document in the [Swagger Editor](#). Execute a request to make sure it's working correctly.

Swagger UI tutorial

Swagger UI provides a display framework that reads an [OpenAPI specification document](#) and generates an interactive documentation website. This tutorial shows you how to use the Swagger UI interface and how to integrate an OpenAPI specification document into the standalone distribution of Swagger UI.

For a more detailed conceptual overview of OpenAPI and Swagger, see [Introduction to the OpenAPI specification and Swagger \(page 143\)](#).

For step-by-step tutorial on creating an OpenAPI specification document, see the [OpenAPI tutorial \(page 160\)](#). In the tutorial, I showed how the OpenAPI specification gets rendered by the Swagger UI display that is built into the [Swagger Editor](#). Here I explain how to use Swagger UI as a standalone output.

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 and try the API. Additionally, Swagger UI is an [actively managed project](#) (with an Apache 2.0 license) that supports the latest version of the OpenAPI spec (3.0) and integrates with other Swagger tools.

In the following tutorial, I'll show you how Swagger UI works and how to integrate an OpenAPI specification document into it.

Before we dive into Swagger, it might help to clarify some key terms.

Swagger

Refers to API tooling that around 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. (See [What Is the Difference Between Swagger and OpenAPI?.](#))

OpenAPI

Refers to the name of the specification. The OpenAPI specification is driven by the [OpenAPI initiative](#), backed by the Linux Foundation and steered by [many companies and organizations](#). The YAML or JSON file that you create to describe your API following the OpenAPI specification is called the “OpenAPI specification document.”

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. This is the tool that transforms your spec into the [Petstore-like site](#).

Swagger Codegen

Tools that generate 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 details.

The Swagger UI Petstore example

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](#).

The screenshot shows the Swagger Petstore UI interface. At the top, there is a navigation bar with a logo, the URL <http://petstore.swagger.io/v2/swagger.json>, and an 'Explore' button. Below the navigation bar, the title 'Swagger Petstore 1.0.0' is displayed, along with a note about the base URL: '[Base url: petstore.swagger.io/v2] [http://petstore.swagger.io/v2/swagger.json]'. A message below states: '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.' There are links for 'Terms of service', 'Contact the developer', 'Apache 2.0', and 'Find out more about Swagger'. On the left, a 'Schemes' dropdown is set to 'HTTP' and an 'Authorize' button is visible. The main content area is titled 'pet Everything about your Pets'. It contains four API endpoints listed in a table:

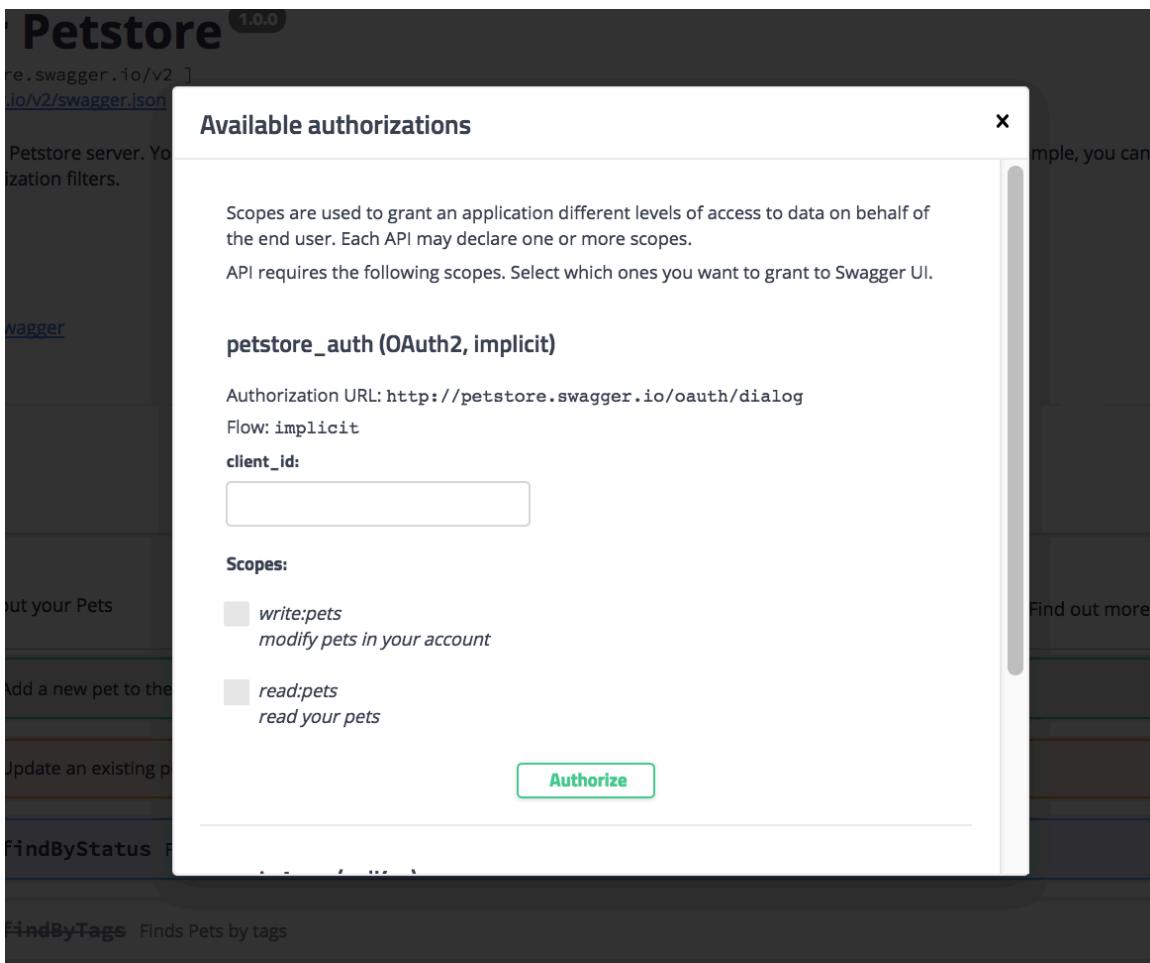
Method	Path	Description	Lock icon
POST	/pet	Add a new pet to the store	locked
PUT	/pet	Update an existing pet	locked
GET	/pet/findByStatus	Finds Pets by status	locked
GET	/pet/findByTags	Finds Pets by tags	locked

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:



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 Everything about your Pets

POST /pet Add a new pet to the store

Parameters

Name Description

body * required
(body)

Pet object that needs to be added to the store

Example Value Model

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

Parameter content type application/json

Responses

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**.

POST /pet Add a new pet to the store

Parameters

Name Description

body * required
(body)

Pet object that needs to be added to the store

Example Value Model

```
{
  "id": 193844,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "Bentley",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

Cancel

Parameter content type application/json

Execute

Swagger UI submits the request and shows the [curl that was submitted \(page 47\)](#). The Responses section shows the [response \(page 116\)](#). (If you select JSON rather than XML in the “Response content type” drop-down box, the response’s format will be JSON.)

Code	Details
200 <i>Undocumented</i>	Response body <pre>{ "id": 193844, "category": { "id": 0, "name": "string" }, "name": "Bentley", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }], "status": "available" }</pre>

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 actually 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 Petstore), check out a few Swagger implementations:

- [Reverb](#)
- [VocaDB](#)
- [Watson Developer Cloud](#)
- [The Movie Database API](#)
- [Zomato 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.

You'll notice the documentation is short and sweet in a Swagger UI implementation. This 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 \(page 78\)](#) of your documentation.

Activity 4c: 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](#) or [Sunrise/sunset Swagger UI](#).

```
<a target="_blank" class="noExtIcon" href="https://idratherbewriting.com/learnapidoc/assets/files/swagger/index.html">
```

The screenshot shows the Swagger UI interface for the OpenWeatherMap API. At the top, there is a green header bar with the word "swagger" and a "Explore" button. Below the header, the title "OpenWeatherMap API" is displayed, along with a "2.5 OAS3" badge and a link to the specification file: http://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_openweathermap.yml. A note below the title states: "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." A note below that says: "Note: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API." Another note below states: "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." On the left side, there is a sidebar with links to "Terms of service", "OpenWeatherMap API - Website", "Send email to OpenWeatherMap API", "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)", and "API Documentation". On the right side, there is a "Server" dropdown set to "http://api.openweathermap.org/data/2.5/", an "Authorize" button with a lock icon, and a "Current Weather Data" section. Under "Current Weather Data", there is a "GET /weather" button with the description "Call current weather data for one location".

```
</a>
```

Demo of Swagger UI rendering an OpenWeatherMap OpenAPI specification document

To integrate your OpenAPI spec into Swagger UI:

1. Prepare a valid OpenAPI specification document:
 - For instructions on creating an OpenAPI specification document from scratch, follow

- the [OpenAPI tutorial here \(page 160\)](#).
- To use a pre-built OpenAPI specification document, you can use the [OpenWeatherMap spec file](#) or the [Sunrise/sunset API spec file](#). (Right-click the link and save the YAML file to your desktop.)
2. Make sure your OpenAPI specification is valid. Paste your OpenAPI specification code into the [Swagger online editor](#) and make sure no warnings appear. The view on the right of the Swagger Editor shows a fully functional Swagger UI display.
 3. Go to the [Swagger UI GitHub project](#).
 4. Click **Clone or download**, 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.
5. Drag the **dist** folder out of the `swagger-ui-master` folder so that it stands alone. (Then optionally delete the `swagger-ui-master` folder and zip file.)
 6. Drag your OpenAPI specification file that you prepared earlier (in step 1) into the 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
        └── [your openapi specification file]
```

7. Inside your **dist** folder, open **index.html** in a text editor such as [Atom editor](#) or [Sublime Text](#).
8. Look for the following code:

```
url: "http://petstore.swagger.io/v2/swagger.json",
```

9. Change the `url` value from `http://petstore.swagger.io/v2/swagger.json` to a relative path to your YAML file. For example

```
url: "openapi_openweathermap.yml",
```

or

```
url: "openapi_sunrise_sunset.yml",
```

Save the file.

10. View the index.html file locally in your browser. Note that Chrome's security restrictions (CORS objections) prevent you from viewing the Swagger UI file locally. You have several workarounds:

- View the file locally using [Firefox](#).
- Use the hosted URL for [openapi_openweathermap.yml](#) or [openapi_sunrise_sunset.yml](#). (Right-click the link and select **Copy Link Address**.)
- Upload the **dist** folder to a web server and view it there.
- Put the YAML file onto a public [GitHub Gist](#) and then click **Raw**. Use the URL for this Gist.
- Use a local server such as [simple local HTTP server](#).

When you're ready to publish your Swagger UI file, you simply upload the folder 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>. (You can change the "dist" folder name to whatever you want.)

For more instructions in working with Swagger UI, see the [Swagger.io docs](#).

Configuring Swagger UI parameters

Swagger UI provides a number of [configuration parameters](#) (unrelated to your [OpenAPI parameters \(page 175\)](#)) 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 Model, 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 the Swagger UI demo \(page 0\)](#), 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 (since 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 endpoint in Swagger. 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 \(page 173\)](#), the [info object \(page 168\)](#), and the [externalDocs object \(page 209\)](#)
- The Swagger UI looks mostly the same for each output. You can [customize Swagger UI](#) with your own branding, but it will take some deeper UX skills.
- The Swagger UI might be a separate site from your other documentation. This means 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 \(page 241\)](#) for more details on workarounds.

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](#).

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.

This page can only be viewed online in your computer's web browser. Go to https://idratherbewriting.com/learnapidoc/pubapis_swagger_demo.html to view it.

Personally, I prefer to separate out Swagger UI from the rest of my docs simply because I dislike the sense of a website within a website. Developers can keep the Swagger UI page open as a [quick reference \(page 316\)](#) while they work on their projects. I view the Swagger UI reference output similar to the standalone nature of [Javadoc \(page 0\)](#).

SwaggerHub introduction and tutorial

Previously, I explored using the open-source [Swagger UI project \(page 214\)](#) as a way to render your [OpenAPI specification document \(page 160\)](#). The same company (Smartbear) that offers the free, open-source version of Swagger Editor and Swagger UI also offers a premium version with more robust features. This premium version 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 + UI approach 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 that are 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 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.

SwaggerHub is used by more than 15,000 software teams across the globe. As the OpenAPI spec becomes more of an industry standard for API documentation, SwaggerHub's swagger-specific tooling becomes 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](#).

This same company, 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.

The screenshot shows the SwaggerHub dashboard titled "MY hub". The interface includes a sidebar with navigation icons (Home, Create, Search, ID) and a top bar with user information (tomjohit-idbw) and settings. A search bar at the top right allows searching within the hub. Below the search bar, there are filters for "Sort by: Recently updated", "Spec", "Type", "Visibility", "State", and "Owner". The main content area displays three API entries:

- open-weather_map_api**: PUBLIC | PUBLISHED. 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, etc. Buttons: API, OAS3, Bell icon, Share icon.
- MashapeWeatherAPI**: PUBLIC | UNPUBLISHED. Description: MashapeWeatherAPI - [2.3]. Buttons: API, Bell icon, Share icon.
- sample**: PUBLIC | UNPUBLISHED. Description: This is an example of using OAuth2 Access Code Flow in a specification to describe security to your API. Buttons: API, Bell icon, Share icon.

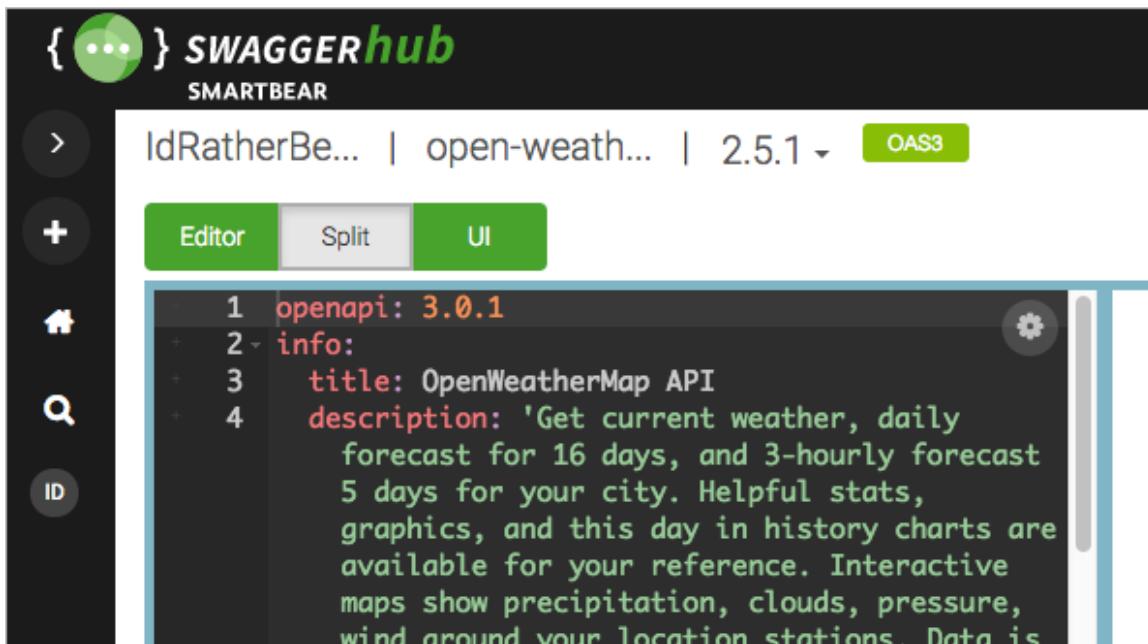
At the bottom left, it says "SHOWING 1-3 OF 3".

The dashboard shows a list of the APIs you've created. In this example, you see the [OpenWeatherMap API \(page 32\)](#) that I've been using throughout this course.

SwaggerHub Editor

SwaggerHub contains the same [Swagger Editor](#) that you can access online. This 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 3 modes:

- **Editor**: Shows the spec in full screen
- **Split**: Shows the spec on the left, the UI display on the right
- **UI**: Shows the UI in full screen (and is fully interactive, just as it will be when published)

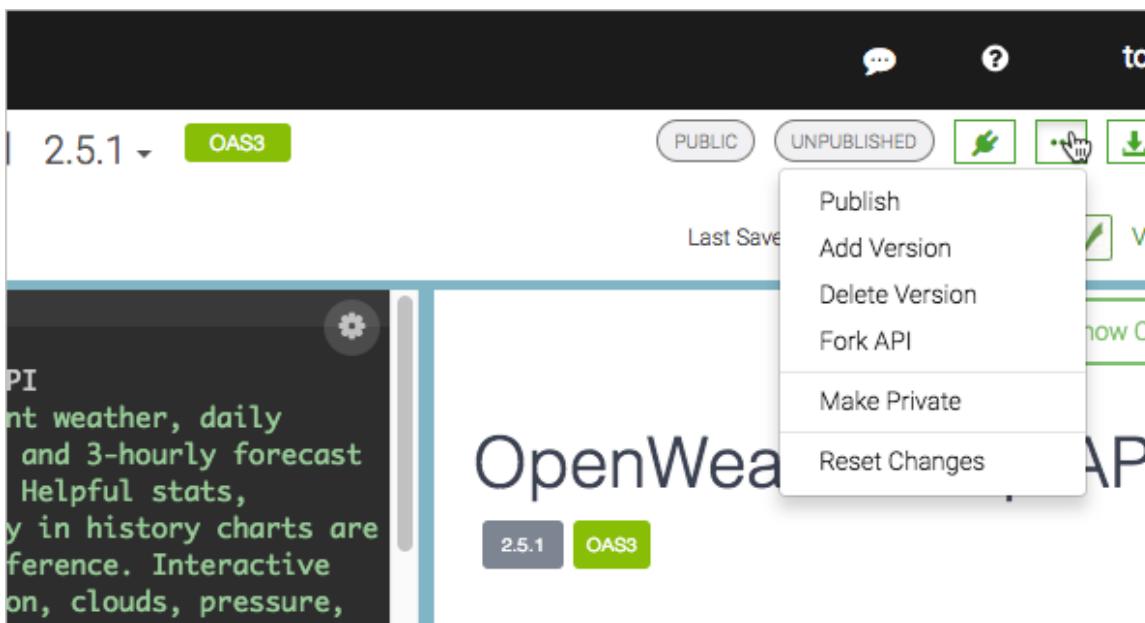


Most importantly, as you're working in the Editor, SwaggerHub allows you to save *your work*. With the free Swagger Editor, your content is just kept in the browser cache, with no ability to save the file in the cloud. When you clear your cache, your content is gone. As a result, if you use the standalone Swagger Editor, you have to regularly copy the content from the Swagger Editor into a file on your own computer each time you finish.

You can save your specification document directly in SwaggerHub, or you can reference and store it in an external source such as GitHub.

Versions

Not only does SwaggerHub allow you to save your OpenAPI spec, you can save different versions of your spec. This means you can experiment with new content by simply adding a new version. You can return to any version you want, and you can also publish or unpublish any version.



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 on 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.

You can also send users to the latest version by excluding the version number in the path: https://app.swaggerhub.com/apis/IdRatherBeWriting/open-weather_map_api/. When you go to this link, you are forwarded to the latest published version of the documentation automatically (2.5.2).

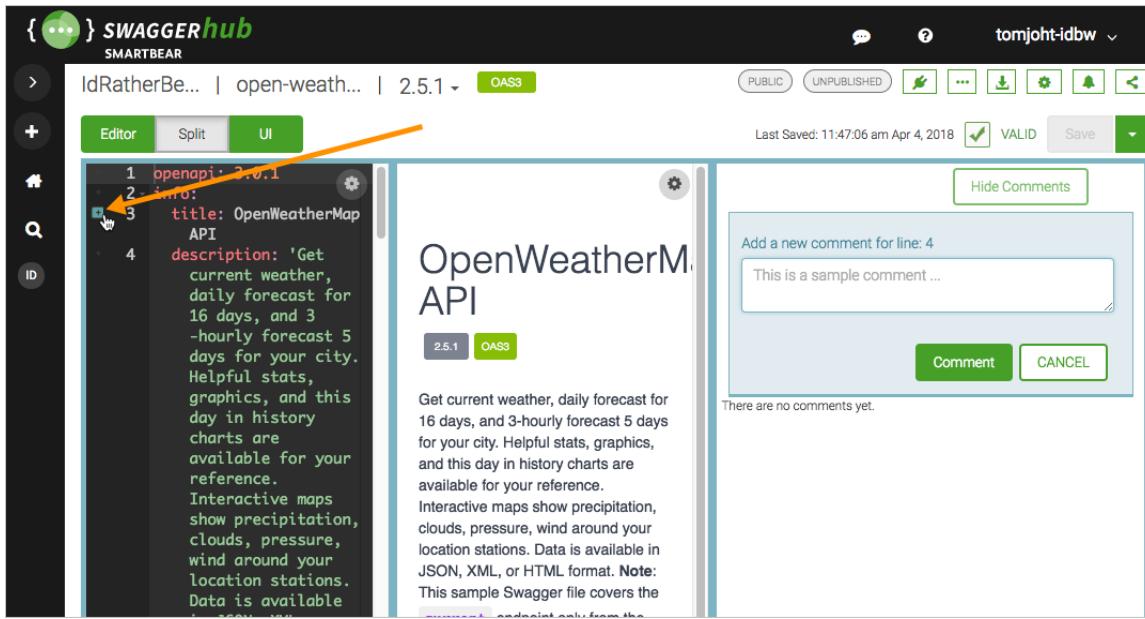
Versioning is helpful when you're collaborating on the spec with other team members. For example, suppose you see the original version drafted by an engineer, and you want to make major edits. Rather than directly overwriting the content (or making a backup copy of an offline file), you can create a new version and then take more ownership to overhaul that version with your own wordsmithing, 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 own 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.



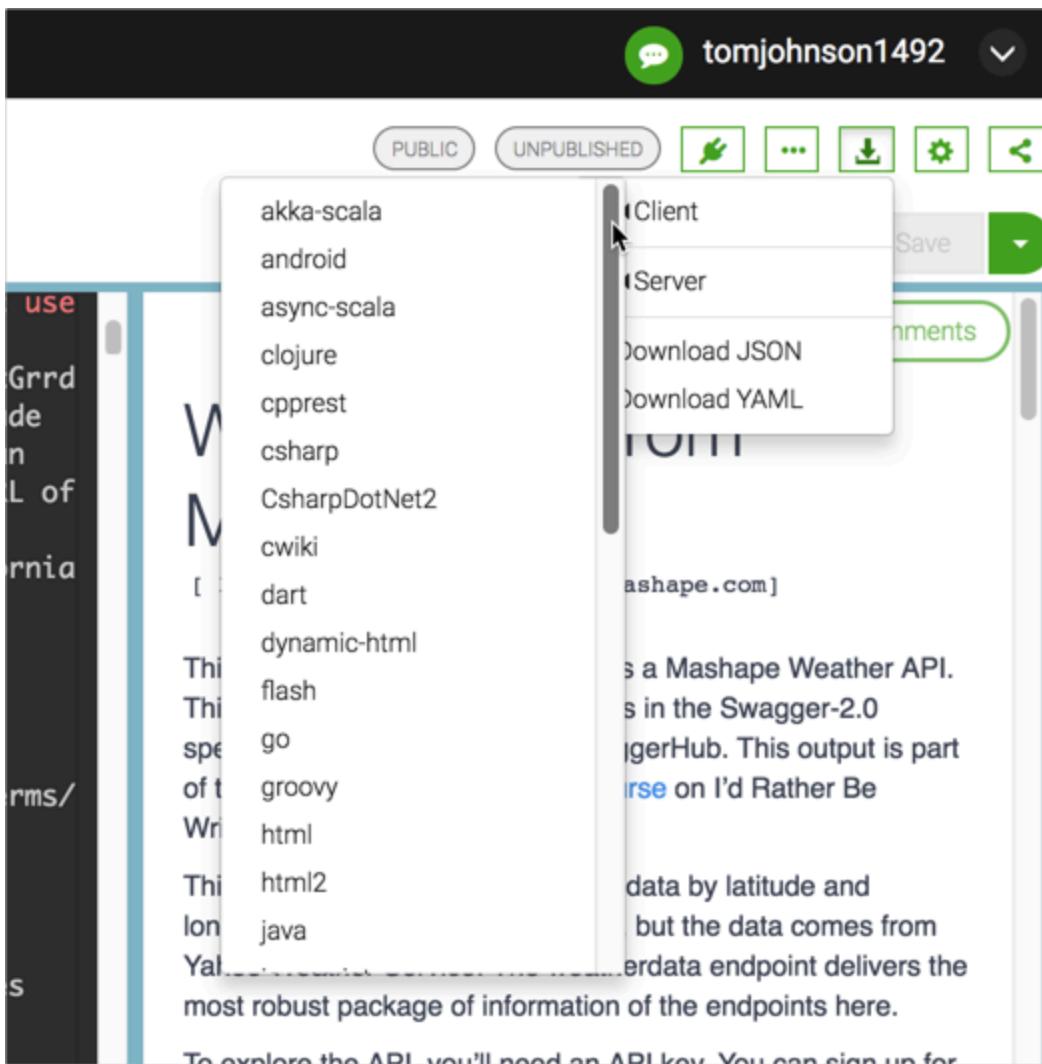
When you click the plus sign, a comment pane appears on the right where you can elaborate on comments, and where others can reply. Users can edit, delete, or resolve the comments. This commenting feature helps facilitate the review process in a way that tightly integrates with your content. You can also 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 alone would be onerous 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 in more than 30 formats.



For example, suppose a user is implementing your REST API in a Java application. The user can choose to download the Java client SDK and will see code showing 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.

Providing this code not only speeds implementation for developers, it also helps you scale your language-agnostic REST API to a greater variety of platforms and users, reducing the friction in adoption.

The client and server SDKs aren't yet available for OpenAPI 3.0 specs, just for 2.0 specs. As of April 2018, a note in the UI indicates that these features will be added soon. Until the features are added, you can see these menus in an [older weather API doc here](#) I created that uses the 2.0 version of the spec.

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 Weather 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 \(page 241\)](#).

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 6 different code samples:

The screenshot shows the SwaggerHub HTML2 export for the Weather API. At the top, a green button labeled "GET" is followed by the endpoint "/weatherdata". Below this, a section titled "Usage and SDK Samples" contains tabs for "Curl", "Java", "Android", "Obj-C", "JavaScript" (which is selected), "C#", and "PHP". A large code block below the tabs displays the following JavaScript code:

```
var = require('');  
var defaultClient = .ApiClient.instance;  
  
// Configure API key authorization: api_key  
var api_key = defaultClient.authentications['api_key'];  
api_key.apiKey = "YOUR API KEY"  
// Uncomment the following line to set a prefix for the API key, e.g. "Token"  
(defaults to null)  
//api_key.apiKeyPrefix['X-Mashape-Key'] = "Token"  
  
var api = new .WeatherDataApi()
```

Mocking Servers

Another cool feature of SwaggerHub is the ability to [create mock API servers](#). Suppose you have an API where 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 simply don't have test accounts/systems). At the same time, you probably want to simulate real API responses to give users 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 jacking 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 \(page 146\)](#)) is to avoid coding an API with endpoints and responses that users don't actually 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  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 basically re-useable code snippets that you can leverage to avoid duplication in your spec.

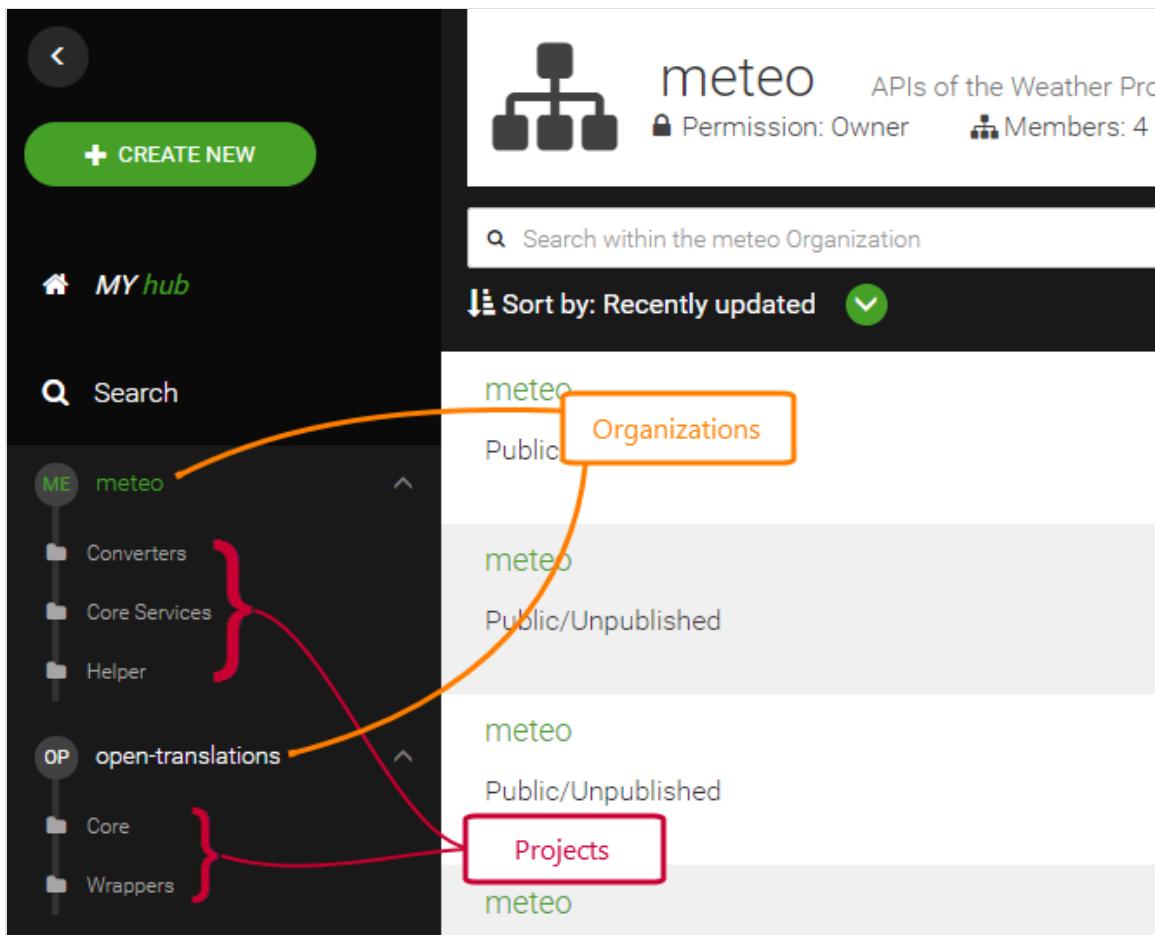
When you create definitions for your requests and responses, you may find yourself re-using the same code over and over. Rather than duplicating this code, you can save it as a domain. When you want to re-use the code, you select this domain.

Using the domain minimizes duplicate content and enables you to be more consistent and efficient. You can read more about domains [here](#).

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 he or she has access to.

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 but still have their APIs logically grouped.



This aspect of organizations and projects may not seem essential if you have just 1 or 2 APIs, but consider how you'll scale and grow as you have dozens of APIs and multiple teams. In these more robust scenarios, the organizations and projects features become essential.

Conclusion — 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, review, and client/server SDK generation.

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 the spec and associating 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 very company that maintains the Swagger tools, and sponsors the Swagger (OpenAPI) specification), you can plug into the tools you need to scale your API documentation efforts.

To read more about SwaggerHub, checkout my blog post [SwaggerHub: A collaborative platform for working on OpenAPI/Swagger specification files, and more.](#)

Stoplight — visual modeling tools for creating your OpenAPI spec

Previously, I talked about [SwaggerHub \(page 224\)](#) as a tool to simplify your authoring and publication of the [OpenAPI specification \(page 160\)](#). Now let's explore another tool called [Stoplight.io](#).

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 an important tool, and why I've both listed it in this course and created an [activity for using it \(page 237\)](#). 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 (page 156)* working on different parts of the OpenAPI spec definition. He said they actually spent most of the day in YAML, and it was kind of frustrating/tedious/boring. For this participant, this isn't what he imagined when he signed up to learn how to create interactive API docs.

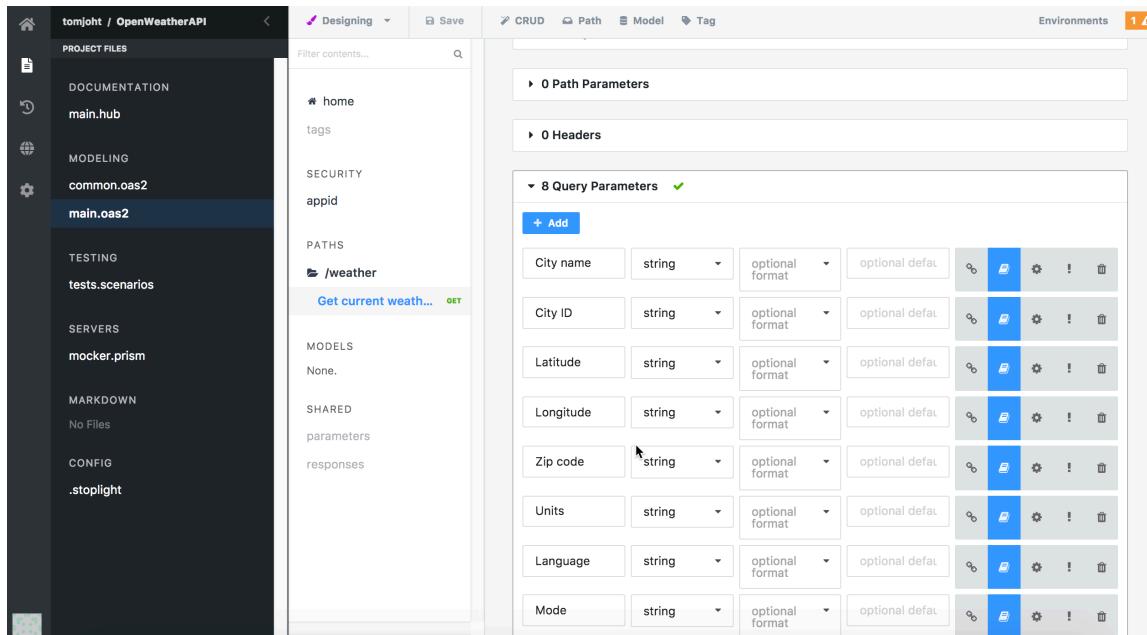
In the instructor's defense, I told my friend that describing an API using the OpenAPI spec *does* pretty much involve living in YAML all day, and it *is* tedious, highly prone to error, and technical. One of my favorite API bloggers, API evangelist Kin Lane, explains that "hand crafting even the base 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, too, 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, and I found Stoplight to be really 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 specification. You don't have to know the data type for each property, whether the property needs to be indented or defined directly and so forth. That level of complexity has been abstracted away in a GUI for designing your API.



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 simply 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, as I've demonstrated in this short video:

This content doesn't embed well in print, as 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 and the specification code easily. If you want to work in the code, your updates 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:

This content doesn't embed well in print, as 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 \(page 224\)](#) and [Readme \(page 387\)](#). 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 wider 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, after you have the specification document:

- 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 their own 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."

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 larger 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 leverage by 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 or not 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 in 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*.

This has, unfortunately, been the approach in 100% 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 done later.

As I was explaining the idea of spec-first development to our product manager and other leaders, they suddenly became very intrigued. I explained that we can use the OpenAPI specification to do prototype design and UX testing of the API before we even think about development.

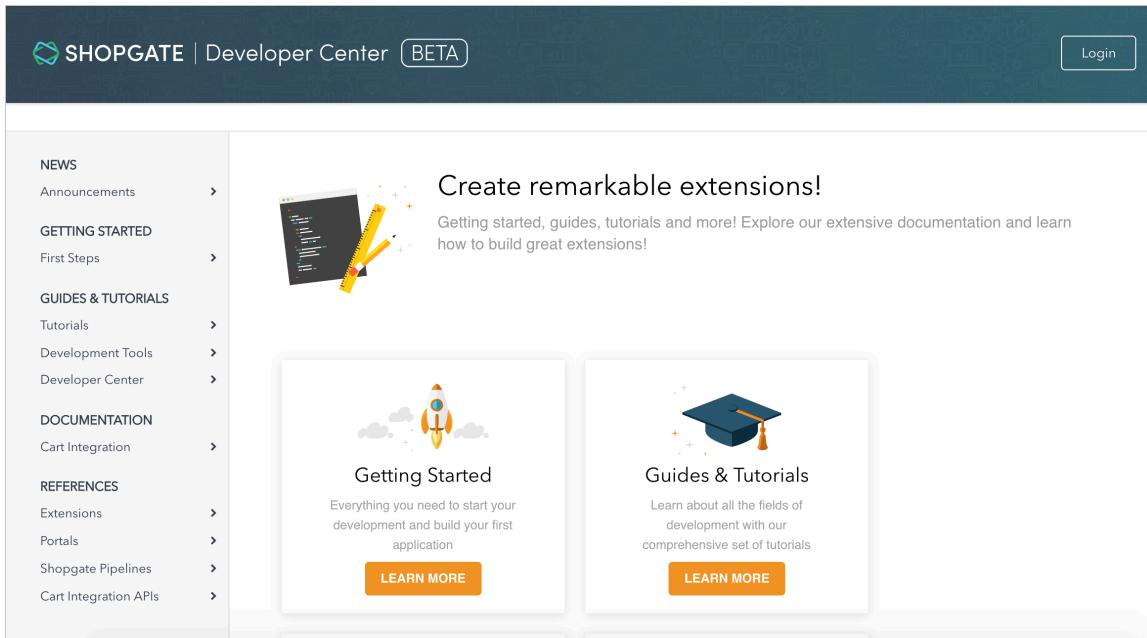
The product managers found this idea compelling no doubt because some previous attempts at creating an API weren't so successful. In fact, we were discussing plans for an API to replace the previous API that never caught on because the intended developer audience found it too clunky, cumbersome, and off-target with the information they needed. It was a failed API.

The light bulb was going off in these product managers' heads. They started to recognize a way to avoid a similar failure scenario. By pushing code development later on in the API lifecycle, we could avoid scenarios where we lose months of work due to unforeseen requirements or other poor planning. Through the API definition, we could build a nearly functional prototype of the API before code development, including mock servers to simulate actual responses that users could evaluate and provide feedback about.

In this 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 actually get traction. Once the API gets coded by developers, it's extremely hard to change even something as simple as the casing of 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 non-reference content (the tutorials, guides, and other how-to's) together with the reference API docs. Here's an example hosted doc site on Stoplight for a product called [Shopgate](#).



You can integrate your reference and non-reference documentation in Stoplight's hosted doc solution. Integrating these two content types has been a longtime challenge for tech writers in the API doc space.

Stoplight also allows you to create variables to use in *both* your specification and your how-to docs. Stoplight plans to take re-use one step further by allowing re-use of your [spec's component definitions \(page 181\)](#) in your [non-reference documentation \(page 263\)](#) as well. (But this feature is still forthcoming.)

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 definitely 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.

See the next section for a hands-on activity using Stoplight.

Activity: Use Stoplight to edit your OpenAPI spec

In previous activities, you created your own [OpenAPI specification document \(page 212\)](#) and also created a [Swagger UI display \(page 219\)](#) with that specification document. In this example, you'll make some modifications to your specification document using [Stoplight \(page 233\)](#), a visual editor.

Activity 4d: Download and populate Stoplight with an OpenAPI specification

In this activity, you'll work with an OpenAPI file in Stoplight, which provides a GUI editor for working with the OpenAPI specification information. To speed things up, you'll start with a pre-built OpenAPI file that you paste into the Stoplight editor, and then you'll make some modifications to it using Stoplight's visual modeling tools. You can use Stoplight in the browser or as a web app. For simplicity, we'll use the browser version.

To work with an OpenAPI file in Stoplight:

1. Stoplight only supports OpenAPI 2.0, not 3.0. You could convert your 3.0 spec using [APIMATIC Transformer](#) to 2.0, but to speed things up, download this [2.0 OpenAPI JSON file already converted](#).
2. Go to next.stoplight.io/.
3. Click **Login** in the upper-right corner and log in using your GitHub account.
4. Click **New Personal Project**.
5. Type a **Project name** (e.g., OpenWeatherMap API), choose whether you want the visibility public or private (it doesn't matter), and click **Next**.
6. On the next screen (Project Designer), select the **Import Existing** tab. Then click **Upload OpenAPI (Swagger), Markdown or HTML File** and select the `openweathermap_swagger20.json` file that you downloaded in step 1.
7. Click the `+openweathermap_swagger20.oas2.yml` button. The OpenAPI file gets uploaded into Stoplight and the data populates the Stoplight interface.
8. At the top of the screen, switch between the **Code** and **Design** views by clicking the corresponding buttons at the top. Make some edits in the code and then switch to the Design view to see the edits reflected. (Note that Stoplight prefers JSON as the code format.) The following video shows this process:

This content doesn't embed well in print, as it contains YouTube videos. Please go to https://idratherbewriting.com/learnapidoc/pubapis_stoplight.html to view the content.

9. Explore the different options in the Design editor (specifically, expand **Paths** and click `/weather > Call current weather data for...`) and make some arbitrary textual changes to see how to update information.

Automatically generate schema documentation

One of the coolest features in Stoplight is the ability to auto-generate the schema documentation from a sample JSON response. Try this out by following these steps:

1. In the middle column of Stoplight, click **PATHS**, and then click `/weather`, and then click **Call current weather data for**
2. In the **Responses** section, click **Raw Schema** and delete the existing response schema. (Even when you delete the info here, it will leave `{}` there when you switch tabs, which is good.)
3. Switch to the **Editor** tab, click **Generate from JSON** and paste in the following JSON response from the OpenWeatherMap weather endpoint:

```
{  
  "coord": {  
    "lon": -121.96,  
    "lat": 37.35  
  },  
  "weather": [  
    {  
      "id": 801,  
      "main": "Clouds",  
      "description": "few clouds",  
      "icon": "02d"  
    }  
  ],  
  "base": "stations",  
  "main": {  
    "temp": 75.51,  
    "pressure": 1014,  
    "humidity": 8,  
    "temp_min": 66.92,  
    "temp_max": 80.6  
  },  
  "visibility": 16093,  
  "wind": {  
    "speed": 11.41,  
    "deg": 330  
  },  
  "clouds": {  
    "all": 20  
  },  
  "dt": 1541544960,  
  "sys": {  
    "type": 1,  
    "id": 479,  
    "message": 0.0043,  
    "country": "US",  
    "sunrise": 1541515128,  
    "sunset": 1541552625  
  },  
  "id": 420006397,  
  "name": "Santa Clara",  
  "cod": 200  
}
```

4. Then click **Generate!**

Stoplight automatically generates the JSON schema that conforms to the OpenAPI spec. You just saved yourself an afternoon of work!

The screenshot shows the Stoplight API editor interface. On the left, there's a sidebar with sections like externalDocs, components, SECURITY (None), PATHS (a selected endpoint '/weather' with a description 'Call current weather data for ...'), MODELS (None), SHARED (parameters and responses). The main area has tabs: Viewer, Editor (which is active), and Raw Schema. Under the Editor tab, there's a 'Generate From JSON' button. The right side displays a hierarchical JSON schema with validation status and required fields:

Path	Type	Validations	Required
/weather	object {12}	0 validations	Required
coord	object {2}	0 validations	Required
weather	array[object]	0 validations	Required
base	string	0 validations	Required
main	object {5}	0 validations	Required
visibility	integer	0 validations	Required
wind	object {3}	0 validations	Required
clouds	object {1}	0 validations	Required
dt	integer	0 validations	Required
sys	object {6}	0 validations	Required
id	integer	0 validations	Required

Stoplight will autogenerated the JSON schema documentation

Here's a short video showing the process of auto-generating JSON. The Stoplight editor has evolved a bit, but it's still highly similar. (Instead of starting with the above sample JSON, the video makes request in Postman and then copies the response from there — but the idea should be clear.)

This content doesn't embed well in print, as it contains YouTube videos. Please go to https://idratherbewriting.com/learnapidoc/pubapis_stoplight.html to view the content.

5. Click the **Raw Schema** tab to see the code that Stoplight automatically wrote for you based on the sample JSON you copied in.

This auto-generated schema documentation will make your life easier. Even if you prefer to hand-code your OpenAPI specification files in your own editor, you might find that you visit Stoplight just to auto-generate your response schema documentation.

Integrating Swagger UI with the rest of your docs

Whenever discussions about Swagger and other REST API specifications take place, technical writers invariably ask if they can include the Swagger output with the rest of their documentation. This question dominates tech writer discussions perhaps more than any others when it comes to Swagger.

Single source of truth

When you start pushing your documentation into another source file — in this case, a YAML or JSON file that's included in a Swagger UI file set, you end up splitting your single source of truth into multiple sources. 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?

This conundrum is usually crystal clear to technical writers. API doc consists of more than reference material about the APIs. You've got all kinds of other information about getting API keys, setup and configuration of services, or other details that don't fit into the spec. I covered much of this in [Documenting non-reference sections \(page 264\)](#) part of the guide. You have sections such as the following:

- [API Overview \(page 265\)](#)
- [Getting started tutorial \(page 269\)](#)
- [Authentication and authorization requirements \(page 277\)](#)
- [Status and error codes \(page 286\)](#)
- [Rate limiting and thresholds \(page 293\)](#)
- [Code samples/tutorials \(page 298\)](#)
- [SDKs and sample apps \(page 308\)](#)
- [Quick reference guide \(page 316\)](#)
- [API best practices \(page 323\)](#)
- [Glossary \(page 326\)](#)

Other times, you just have more detail that you need to communicate to the user that won't fit easily into the spec. For example, in the `weather` endpoint in the [sample OpenWeatherMap API](#) that we've been using in this course, there's some detail about city IDs that needs some explanation.

```
...  
},  
"id": 420006397,  
"name": "Santa Clara",  
"cod": 200  
}
```

What does the code `200` mean? If you go to the [City ID section in the docs](#), you'll see a link to download a list of file city codes.

If you have a lot of extra information and notes like this in your reference docs, it can be difficult to fit them into the parameter descriptions allotted in the OpenAPI spec. Unfortunately, there's not an easy solution for creating a single source of truth. Here are some options.

Option 1: Embed Swagger UI in your docs

One solution is to embed Swagger UI in your docs. You can see an example of this here: [Swagger UI Demo \(page 223\)](#). It's pretty easy to embed Swagger into an HTML page. The latest version of Swagger has a more responsive, liquid design. It almost looks *designed* to be embedded into another site.

The only problem with the embedding approach is that some of the Models aren't constrained within their container, so they just expand beyond their limits. Try expanding the Model section in the demo and you'll see what I'm talking about.

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 [this page \(page 223\)](#) and check out the second `<style>` block, you can see the styles I haphazardly added.

With the embedded option, you can still use the official Swagger UI tooling to read the spec, and you can include it in your main documentation. Swagger UI reads the latest version of the [OpenAPI specification \(page 160\)](#), 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 falls overflows in ugly ways in your Model sections, you might want to avoid the embedded approach.

Option 2: Put all info into your spec through expand/collapse sections

You can try to put all information into the specification document itself. 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 add a lot of content here.)

```
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. 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 details.</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>  
                ...
```

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 section. (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, then you can look for alternative approaches. But try to fit it into the spec first. This keeps your information close to the source.

There are just too 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 output the display.

For example, [Spectacle](#) is a project that builds an output from a Swagger file — it requires almost no coding or other technical expertise. More and more tools are coming out that allow you to import your OpenAPI spec. For example, see [Lucybot](#), [Restlet Studio](#), the [Swagger UI responsive theme](#), [Material Swagger UI](#), [Dynamic APIs](#), [Run in Postman](#), [SwaggerHub \(page 224\)](#), and more. They all read the OpenAPI spec.

In fact, importing or reading a OpenAPI specification document is almost becoming a standard among API doc tools. 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 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></t
  r>
  </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 (page 369) can parse YAML, not just Jekyll.)

For more information on this approach, see Peter's writeup 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's sourced to both outputs

Another approach for integrating Swagger's output with your other docs might be to store your descriptions and other info in data yaml 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 [XAMPP 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've tried this approach, I grew frustrated at not being able to immediately validate my spec. It was more challenging to track down the exact culprits behind my validation errors, and I eventually gave up. But it's a technique that could work.

Option 5: Use a tool that imports Swagger and allows additional docs

Another approach is to use a tool like [Readme.io](#) 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 [Headless CMS options \(page 387\)](#). Readme.io requires third-party hosting, but there are some other doc tools that allow you to incorporate Swagger as well.

Sites like [Apiary](#) and [Mulesoft](#) let you import your OpenAPI spec while also add your own custom doc 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 a 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 tutorials and other information that aren't part of the reference. Programmers might find the reference information convenient in the way 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 non-reference information, they can consult the accompanying guide. Think of the Swagger UI output as your API's [quick reference guide \(page 316\)](#).

The truth is that programmers have been operating this way for years with [Javadoc \(page 0\)](#), [Doxygen \(page 0\)](#), 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 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 non-reference information. Your two outputs just became an organizational strategy for your docs.

Testing your API documentation

Overview to testing your docs.....	248
Set up a test environment	250
Test all instructions yourself	253
Test your assumptions	258
Activity: Test your project's documentation.....	262

Overview to testing your docs

Walking through all the steps in documentation yourself is critical to producing high-quality, accurate instructions. But 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 they’re testing.

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.

If you can hook into a set of test cases 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 \(page 250\)](#)
- [Test all instructions yourself \(page 253\)](#)
- [Test your assumptions \(page 258\)](#)



Photo from [Flickr](#) — 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.

Types of test environments

The type 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 \(page 250\)](#)
- [Testing local builds \(page 250\)](#)
- [Testing sample apps in specific programming languages \(page 251\)](#)
- [Testing hardware products \(page 251\)](#)

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. So 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 local instances 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. 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 station and nothing works, or it's much more complicated than he or she let on. 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 \(page 298\)](#) in your code deliverables. You often include a sample app (or multiple apps 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's 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. In my role at Amazon, I have a drawer full of Fire TV devices and prototypes. Through Android Studio, I often run an app on one of these physical devices to test how it works.

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 interface with it. Hopefully, the product is one that you can access to actually play around with.

If you encounter developer resistance ...

Many times developers don't expect that a technical writer will be do anything more than just transcribe and relay the information given to 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) them 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 good 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 for Fire TV to successfully build on my Fire TV hardware. 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 \(page 253\)](#).

Test all instructions yourself

After setting up your [test environment \(page 250\)](#), 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 to testing your instructions

One benefit to 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. This is helpful to the team overall and increases your own 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 own 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 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 \(page 47\)](#), [Postman \(page 39\)](#), or another REST client. Save the calls so that you can quickly run a variety of scenarios.

When you start to run your own 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 turns 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 make testing your docs vital to writing good developer documentation.

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 actually submitting my app to the Appstore — the whole workflow from beginning to end.

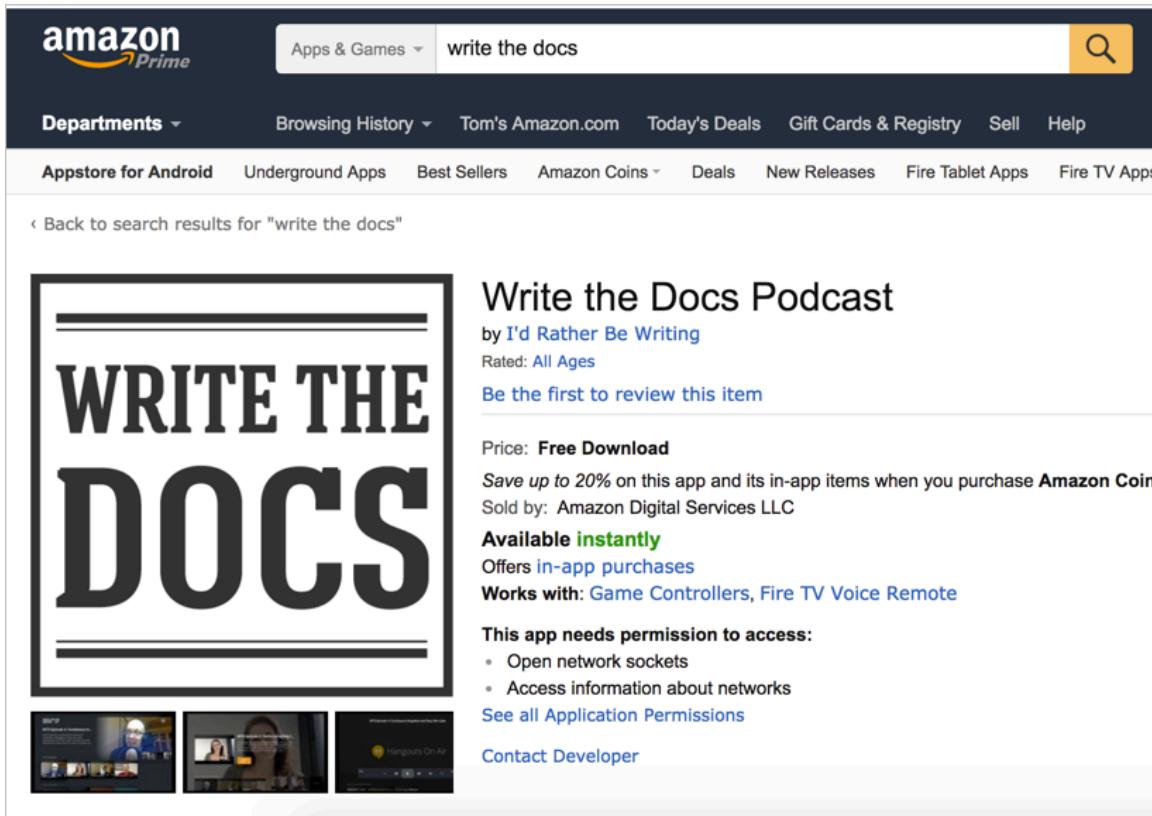
To build my sample app, I had to first figure out how to get content for my app. I decided to take the video recordings of podcasts and meetups that we had through the [Write the Docs podcast](#) and various WTD meetups and use that media for the app.

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.

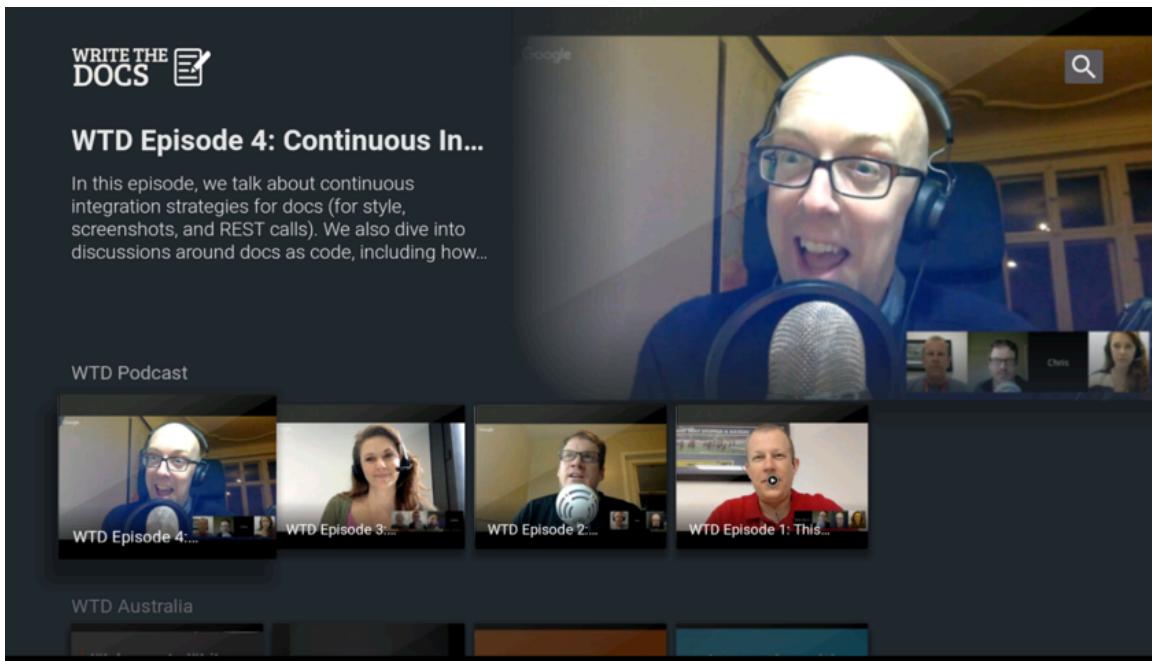
I used Jekyll to build my feed. (You can view my JSON-based feed at podcast.writethedocs.org/fab.json.) The most difficult part in setting up this feed was configuring the `recommendations` object. Each video has some `tags`. The `recommendations` object needed to show other videos that have the same 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.

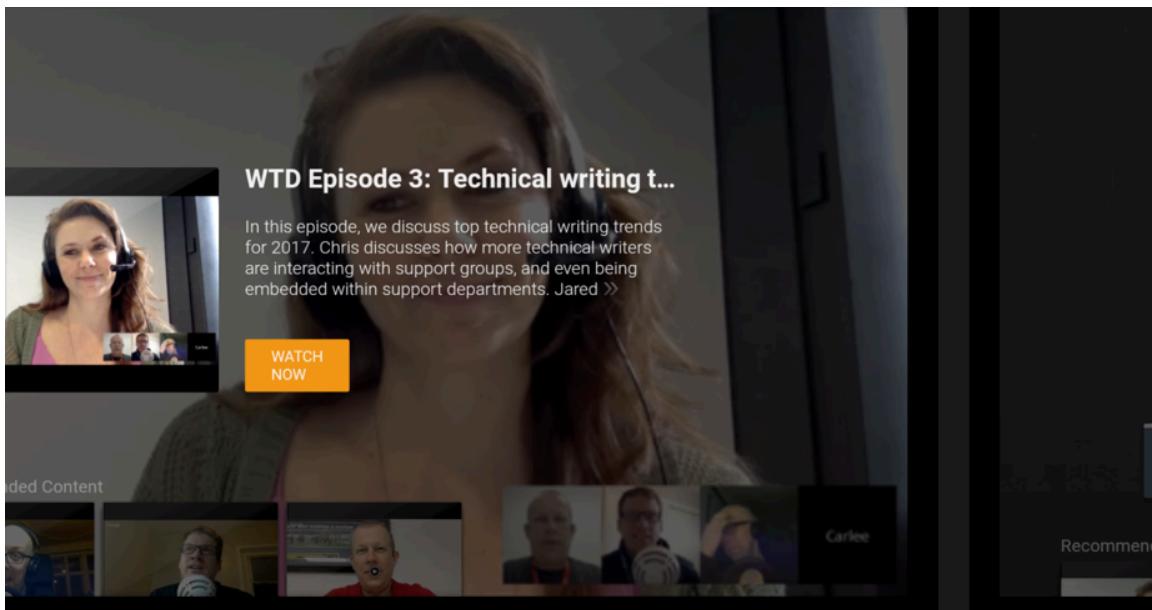
Submitting the app into the Appstore was fun and illuminated parts of the developer's workflow that I hadn't previously understood. You can view the Write the Docs podcast app in the Amazon Appstore website [here](#).



Here's what the app screens look like on your Fire TV:



When you select a video, you see a video preview screen:



The meetups are divided into various categories, which gives some order to the list of videos.

All seemed to go well, but then I discovered some bugs that I wouldn't 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 surprised the dev team, and it would have caused a lot of issues if all apps that third-party developers were building suddenly showed this in-app purchases 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 the importance of testing everything myself and not taking the developer's word for how something works. It also reinforced how absolutely 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's not always possible to run code through real situations, and there are times when I might just help edit and publish engineering-written docs, but that's not the scenario I prefer to work in. I love getting my hands on the code and actually 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, 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.)

Another team developer had a different tool for publishing apps, which I also set about documenting. This tool was designed for non-technical end users and was supposed to be so easy, it didn't have any more documentation than 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 discovered. I uncovered a number of 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 actually 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 actually send recommendations to Fire TV, Fire TV uses only *some* of the recommendations information that you submit. But in my initial docs, I didn't indicate which fields actually get used. This 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 have no effect on 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.

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 boring, 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 inspires or 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 gets 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.

Beyond just testing documentation yourself, you also need to [test it against users \(page 258\)](#).

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 \(page 253\)](#), recognize that what may seem clear to you may be confusing to your 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 know how to SSH onto a server, create [authorizations in REST headers \(page 277\)](#), [use curl to submit calls \(page 47\)](#), 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 and (figuratively) throw them in the trash.

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 or reference section 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, every tech writer should submit his or her instructions through a testing process of some kind, i.e., 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 got to play the editor role in 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 usually other technical writers don’t 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 with their own expertise. In fact, experts may pride themselves in being able to complete a task *despite the poor instruction*. Novices can’t compensate.

Another reason tech writers make good testers is because this kind of 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.

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 an embedded series button. 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 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 errors are identified in the software development process, the more costly it is. 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. At least with documentation, you can continually improve your docs based on incoming feedback.

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, 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.

1. Test a topic

With the [open-source API project you're working with \(page 137\)](#), find one of the following:

- Getting started tutorial
- An API endpoint
- A tutorial or other key task

Test the content. For example, run all the endpoint requests. Proceed through all the steps in the tutorial. Do all the identified tasks with the topic. As you test out the content, identify any incorrect or missing or inaccurate information.

2. 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 is done before a release?
- If you encounter a bug while testing, how should you report it?

Documenting non-reference sections

User guide topics	264
API overview	265
Getting started tutorial.....	269
Authentication and authorization.....	277
Status and error codes.....	286
Rate limiting and thresholds	293
Code samples and tutorials.....	298
SDKs and sample apps	308
Quick reference guide	316
API best practices	323
Glossary.....	326
Activity: Assess the non-reference content in your project.....	332

User guide topics

Up until this point, we've been focusing on the [reference aspect of API documentation \(page 78\)](#) (the endpoints). The reference documentation is only one part (granted, a significant one) in API documentation. In this section, I'll cover the main non-reference topics that are commonly found in API documentation. Rather than "non-reference topics," you might consider this type of information part of the "user guide."

Common topics in the user guide

The following are common non-reference topics common in API documentation:

- [API Overview \(page 265\)](#)
- [Getting started tutorial \(page 269\)](#)
- [Authentication and authorization requirements \(page 277\)](#)
- [Status and error codes \(page 286\)](#)
- [Rate limiting and thresholds \(page 293\)](#)
- [Code samples/tutorials \(page 298\)](#)
- [SDKs and sample apps \(page 308\)](#)
- [Quick reference guide \(page 316\)](#)
- [API best practices \(page 323\)](#)
- [Glossary \(page 326\)](#)

Since the content of these sections varies based on your API, it's not practical to explore each of these sections using the [same weather API \(page 32\)](#) like we did with the API endpoint reference documentation. Instead, I'll provide general descriptions and overviews of what these sections contain, following by examples from actual API documentation sites.

Other topics not covered here

Beyond the sections outlined above, you might want to include other tasks and tutorials specific to your API. For example, what do you expect your users to do? What are their real business scenarios for which they'll use your API? How do you accomplish these tasks?

Sure, there are innumerable ways that users can put together different endpoints for a variety of outcomes. And the permutations of parameters and responses also provide endless combinations. But no doubt there are some core tasks that most developers will use your API to do. For example, with the Twitter API, most people want to do the following:

- Embed a timeline of tweets on a site
- Embed a hashtag of tweets as a stream
- Provide a Tweet This button below posts
- Show the number of times a post has been retweeted

In your non-reference documentation, you'll want to provide documentation for these business goals, just like you would with any user guide. Seeing the tasks users can do with an API may be a little less familiar because you don't have a GUI to click through. But the basic concept is the same — find out what tasks users want to do with this product, and explain how to do it.

I provide some instruction for this general area in [Code samples/tutorials \(page 298\)](#), but you might need to go beyond these non-reference sections to include additional topics based on your user's goals.

API overview

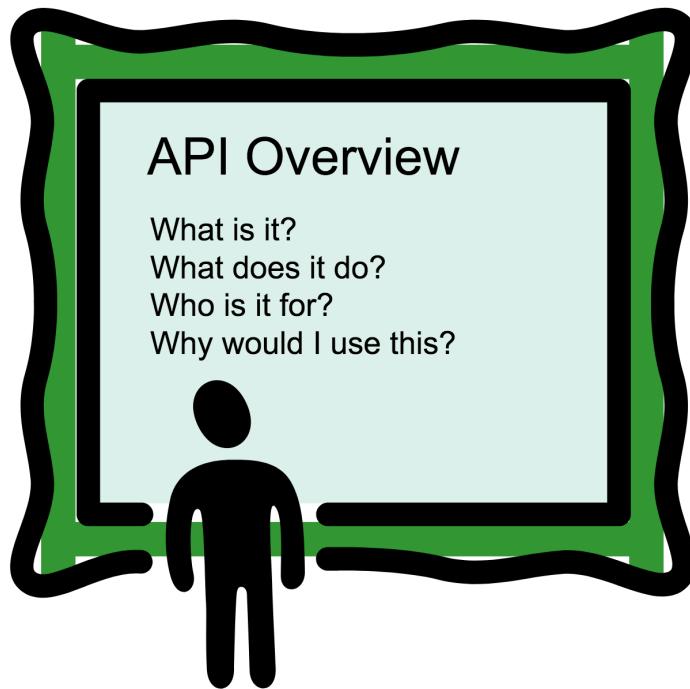
The API overview explains what you can do with the API, including the high-level business goals, the market needs (or pain points) it solves, who the API is for, and other introductory information.

Purpose of the API overview

Too often with API documentation (perhaps because the content is often written by developers), the documentation gets quickly mired in technical details without ever explaining clearly what the API is used for. Don't lose sight of the overall purpose and business goals of your API by getting lost in the endpoints.

The API overview grounds users with a high-level understanding of the system. This high-level understanding is critical to grasping the system as a whole. It allows the details to fit into a larger conceptual framework.

Start at the high-level, getting a gist of what something is about from the title and description, and work your way into more details. This overview provides this initial orientation for the user.



The API overview provides high-level detail about the purpose, audience, and market need for your API.

For more details on the importance of high-level overviews, see [Reduction, layering, and distillation as a strategy for simplicity](#).

In the overview, list some common business scenarios in which the API might be useful. This will give people the context they need to evaluate whether the API is relevant to their needs.

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 return? Is this information relevant and useful to my needs?

Explain the market problems your API 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*)

This overview focuses in on the market problem that the API solves.

The API overview usually appears on the homepage of the API. The homepage (the start of your docs) is a good place to put this overview, because in this overview you also define your audience. Understanding your audience helps you orient the content in your API documentation appropriately.

Sample API overviews

Here are a few sample API overviews.

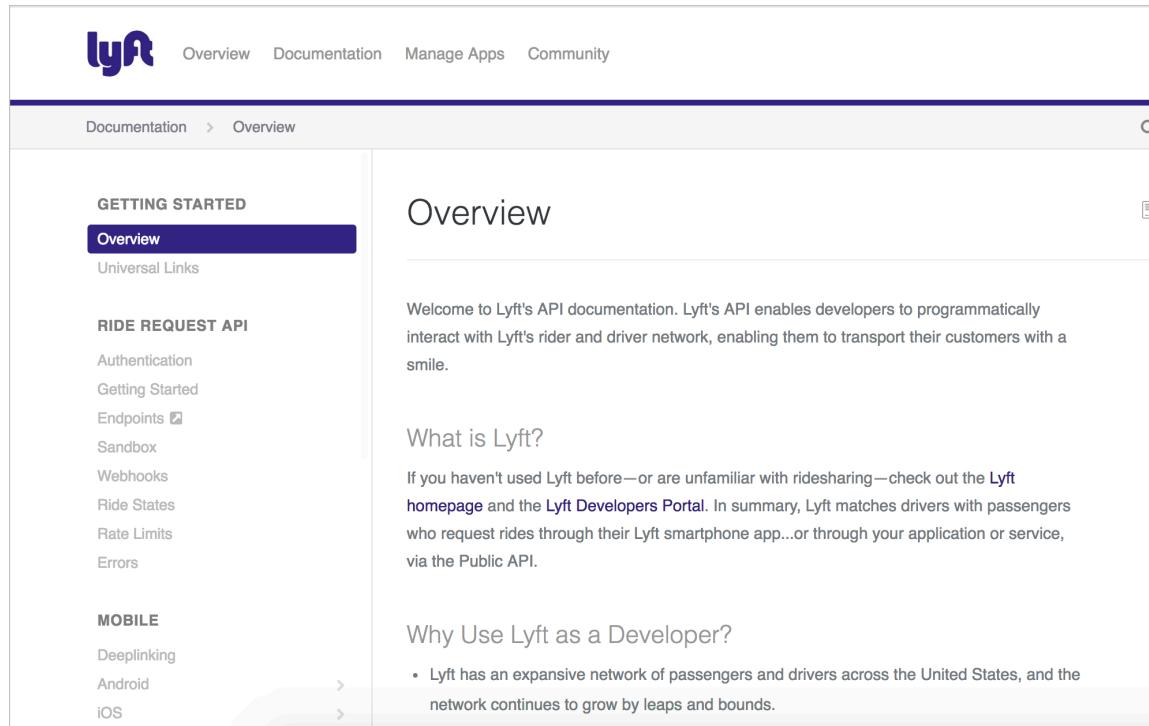
SendGrid

The screenshot shows the SendGrid API overview page. At the top, there is a navigation bar with 'Search Docs' and a magnifying glass icon. To the right of the search bar are links for 'Documentation > User Guide > SendGrid Overview' and a 'View and Edit' button. The main content area has a title 'SendGrid Overview'. On the left, there is a sidebar with a navigation menu. The 'SendGrid Overview' section is currently selected. Other menu items include 'Docs Home', 'User Guide' (with dropdowns for 'SendGrid Overview', 'Legacy Email Activity', 'Email Activity Feed', 'SendGrid for Mobile', 'Legacy Newsletter', 'Marketing Campaigns', 'Statistics', 'Suppressions', 'Transactional Templates', 'Settings', 'Transactional Email', 'API', 'Integrate', 'Classroom', and 'Release Notes'). The main content area contains two sections: 'What is SendGrid?' and 'Who is SendGrid for?'. The 'What is SendGrid?' section describes SendGrid as a cloud-based SMTP provider that manages technical details like scaling infrastructure and monitoring. The 'Who is SendGrid for?' section states that SendGrid is for anyone needing reliable, scalable email for their application, handling billions of emails monthly.

SendGrid API overview

The Sendgrid overview starts off with two key sections: “What is SendGrid?” and “Who is SendGrid for?” I like the straightforward approach. Even in the description of what SendGrid is, the authors don’t assume everyone knows what an SMTP provider is, so they link out to more information. Overall, in about 10 seconds you can get an idea of what the SendGrid API is all about.

Lyft

A screenshot of the Lyft API documentation website. The header features the 'lyft' logo and navigation links for Overview, Documentation, Manage Apps, and Community. Below the header, a breadcrumb trail shows 'Documentation > Overview'. A search bar is on the right. The main content area has a sidebar on the left with sections for 'GETTING STARTED' (Overview, Universal Links), 'RIDE REQUEST API' (Authentication, Getting Started, Endpoints, Sandbox, Webhooks, Ride States, Rate Limits, Errors), and 'MOBILE' (Deeplinking, Android, iOS). The main content on the right is titled 'Overview' and includes a welcome message: 'Welcome to Lyft's API documentation. Lyft's API enables developers to programmatically interact with Lyft's rider and driver network, enabling them to transport their customers with a smile.' It also contains a 'What is Lyft?' section and a 'Why Use Lyft as a Developer?' section with a bulleted list: '• Lyft has an expansive network of passengers and drivers across the United States, and the network continues to grow by leaps and bounds.'

Lyft API overview

Lyft’s API overview starts out in a similar way, with sections titled “What is Lyft?” and “Why Use Lyft as a Developer.” Their homepage also provides information on access, rate limits and throttling, and testing. The Lyft authors also recognize that each tech domain has its own lingo, so they provide a [glossary \(page 326\)](#) up front.

IBM Watson Assistant

The screenshot shows the IBM Cloud Docs interface for the Watson Assistant service. On the left, there's a sidebar with sections like 'LEARN' (Getting started tutorial, About), 'HOW TO' (Configuring a Watson Assistant workspace, Defining intents and entities, Building a dialog, Deploying, Improving understanding, Upgrading), and a status message 'Waiting for resource-catalogo.bluemix.net...'. The main content area has a search bar with 'watson-assistant' and a 'Search documentation' button. It shows the 'About' page for Watson Assistant, last updated on 2018-01-26, with a link to 'Edit in GitHub'. Below this is a section titled 'How it works' with a descriptive text and a diagram. The diagram illustrates the system architecture: 'Users' interact with the 'Interface' (represented by icons for Slack, Facebook, and a mobile device). The 'Interface' connects to the 'Application' layer, which in turn connects to the 'Watson Assistant service Workspace'. This workspace interacts with 'Other Watson services' (Tone analyzer, Speech to text, Text to speech) and 'Back-end systems' (represented by icons for a person, a folder, and a database).

IBM Watson Assistant overview

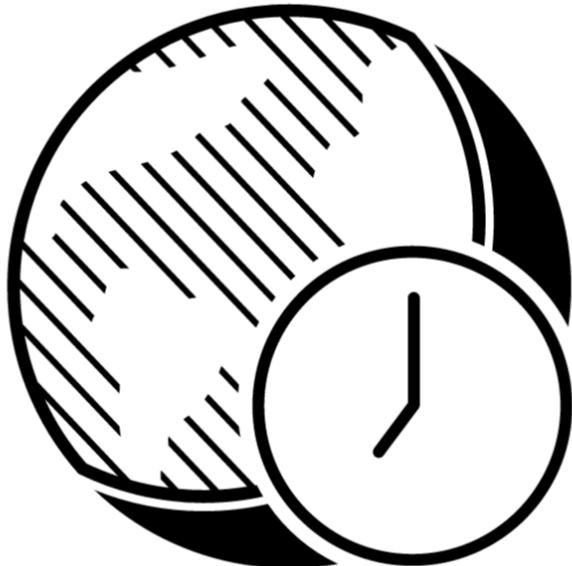
IBM Watson Assistant starts off with a brief summary of the service, followed by a high-level diagram of the system and a summary about how to implement it. Including a diagram of how your API works and how developers can implement it gives users a good grounding in what to expect, such as the level of complexity and time it will take to incorporate the API.

Getting started tutorial

Following the [API Overview section \(page 265\)](#), you usually have a “Getting started” section that details the first steps users need to start using the API. This section often includes the whole process from beginning to end, compressed as simply as possible.

Purpose of the API overview

The Getting Started topic is somewhat like the typical Hello World tutorial in developer documentation, but with an API. The tutorial holds 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?

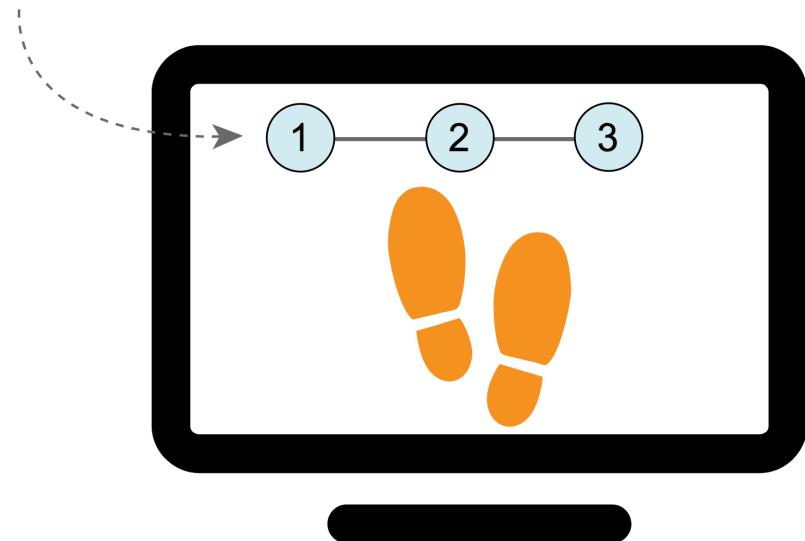
Both Hello World tutorials and Getting Started tutorials share the same goal: to show a user how to use a framework, API, or some other system to get the simplest and easiest result, so they get a sense of how it works and feel productive.

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 a developer can make that call successfully, he or she 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

Get started here



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. If this means using pre-provisioned accounts or setup configurations, do so.

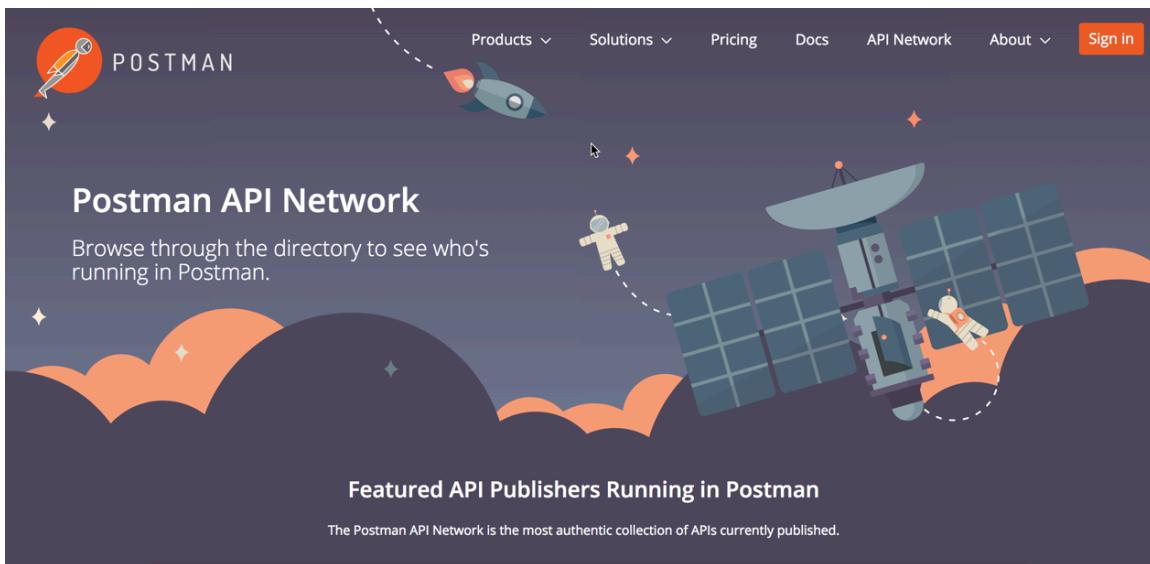
Run in Postman button

In your Getting Started tutorial, consider including a Run in Postman button. (Postman is a REST API GUI client that we explored earlier in [Submit requests through Postman \(page 39\)](#).) If you have your [API endpoints integrated in Postman \(page 39\)](#), you can export your Postman collections as a widget to embed in an HTML page.

The [Run in Postman button](#) provides a button that, when clicked, imports your API info into Postman so users can run calls using the Postman client. As such, the Run in Postman button provides a way to import the interactive, try-it-out API explorer for your endpoints into a web page.

To try out Run in Postman, you can either [import an OpenAPI spec into Postman](#) or enter your API information manually. Then see the Postman docs on how to [Create the Run in Postman button](#).

You can see the many [demos of Run in Postman here](#). Many of these demos are listed in [Postman's API Network](#).

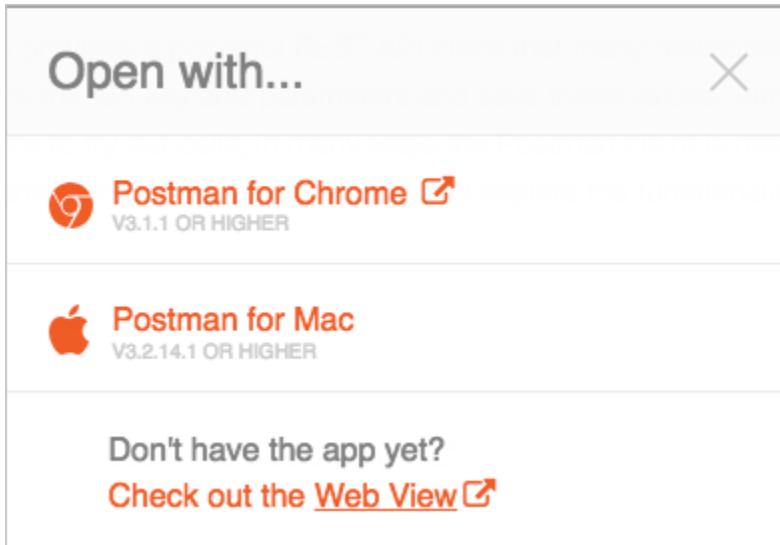


Postman API network

Here's a demo of Run in Postman using the OpenWeatherMap API's `weather` endpoint (which we worked with in [earlier tutorials \(page 32\)](#)):

To view this code, go to https://idratherbewriting.com/learnapidoc/docapis_doc_getting_started_section.html##postman.

When you click the button, you should be prompted to open the collection in a Postman client:



Options to open the Postman collection

[Postman \(page 39\)](#) provides a powerful REST API client that many developers are familiar with. It allows users to customize the API key and parameters and save those values. Although Postman doesn't provide the in-browser experience to try out calls as with [Swagger UI \(page 223\)](#), in many ways the Postman client is more useful, because it lets users configure and save the calls they make. This is what internal developers often use to save and store API calls as they test and explore the functionality.

Especially if your users are already familiar with Postman, Run in Postman is a good option to provide (especially as one option of many for users to try out your API), as it allows users to easily generate the needed code to make requests in practically any language. It 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.

The downside is that your parameter and endpoint descriptions don't get pulled into Postman. Additionally, if users are unfamiliar with Postman, they may struggle a bit to understand how to use it. In contrast, the "Try it out" editors that run directly in the browser tend to be more straightforward and do a better job integrating documentation.

Samples of API overviews

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 in working with the API.

Paypal

The screenshot shows the PayPal Developer REST APIs documentation. The top navigation bar includes links for PayPal Developer, Docs, APIs (which is the active tab), Support, and a search bar. A 'Log into Dashboard' button is also present. On the left, a sidebar lists 'REST APIs' and 'GET STARTED' sections like 'Get Started', 'Authentication and Authorization', 'API Requests', 'API Responses', and 'Make Your First Call'. Below these are 'API REFERENCE' sections for 'Activities API', 'Billing Agreements API', 'Billing Plans API', and 'Customer Disputes API'. The main content area features a 'Get Started' section with a 'REST APIs' heading and a 'Get Started' button. It explains that the PayPal APIs are OAuth 2.0-based and JSON-formatted. A callout box states: '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.' At the bottom, links provide additional resources: 'International Developer Questions' and 'Glossary'.

Paypal getting started tutorial

Paypal's getting started tutorial contains quite a bit of detail, starting out 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

The screenshot shows the Twitter Developer documentation homepage. At the top, there is a purple navigation bar with links for 'Developer', 'Use cases', 'Products', 'Docs', 'More', 'Apply', and a search icon. Below the bar, a search bar says 'Search all documentation...'. The main title 'Getting started' is prominently displayed. Under the 'Basics' section, there is a link to 'Getting started'. The main content area features a large heading 'Get started with Twitter's developer platform'. Below this, a sub-section titled 'Get started: Display Twitter content on the web and in your app' is shown. A bulleted list under this section includes: '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', and 'Questions? See our developer forums'. At the bottom of the page, a note states: '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

The screenshot shows the Parse Server Guide website. At the top, there's a navigation bar with the Parse logo, a search bar, and links for 'Docs' and 'Blog'. Below the header, the main title 'Parse Server Guide' is displayed, along with a 'Browse all platforms' button. On the left, a sidebar contains links for 'Quickstart', 'Getting Started' (which is currently selected), 'Database', 'Usage', 'Keys', 'Using Parse SDKs With Parse Server', 'Deploying Parse Server', and 'Saving Your First Object', 'Connect Your App To Parse Server', and 'Running Parse Server Elsewhere'. The main content area features a large heading 'Getting Started' and a paragraph explaining that Parse Server is an open source version of the Parse backend. It includes a link to the 'GitHub repo'. Below this, there's a list of bullet points: '+ Parse Server is not dependent on the hosted Parse backend.', '+ Parse Server uses MongoDB directly, and is not dependent on the Parse hosted database.', and '+ You can migrate an existing app to your own infrastructure.'.

Parse Server getting started

The Parse Server tutorial provides a good amount of detail and handholding through the various steps. For the more detailed steps of connecting your app and running the server elsewhere, the tutorial links out to more information.

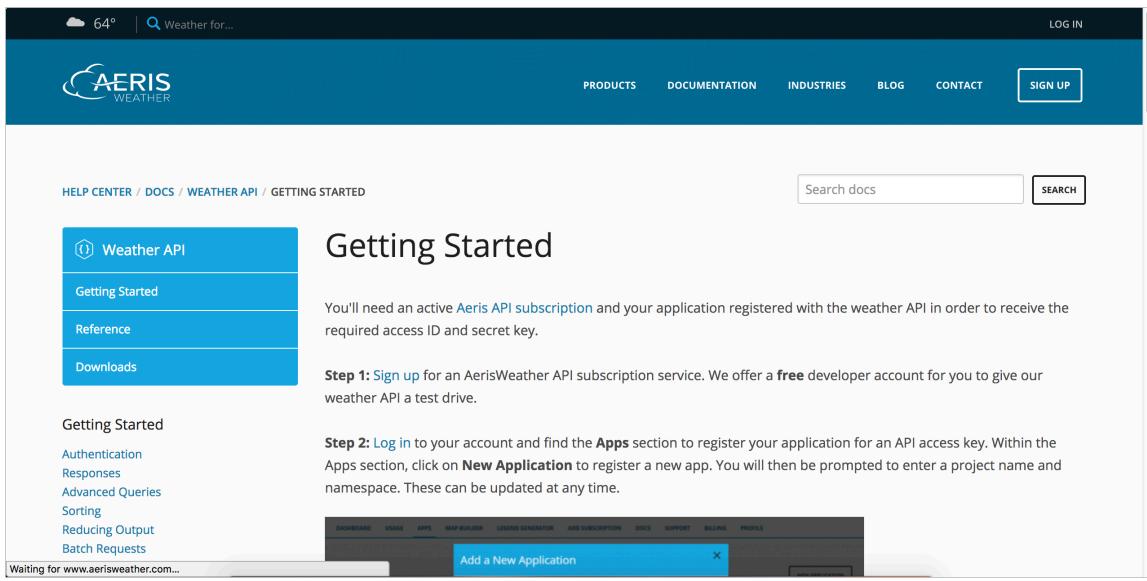
Adsense

The screenshot shows the AdSense Management API Guides page. The top navigation bar includes links for 'HOME', 'GUIDES' (which is highlighted), 'REFERENCE', 'SUPPORT', a search bar, and 'ALL PRODUCTS'. There's also a 'SEND FEEDBACK' button and a user profile icon. The main content area has a sidebar with sections for 'Client Libraries and Samples' (including 'Get Started', 'Make Direct Requests', 'Performance Tips', and 'Upgrade from an older version'), 'Reports' (including 'Dimensions and Metrics', 'Handle the Results', 'Filter', 'Common pitfalls', 'Use the Right Dimension', 'Metadata for Dimensions and Metrics', 'Fill in Missing Dates', 'Use Relative Date Keywords', 'Run Large Reports', and 'Batch'), and a 'Contents' sidebar on the right with links like 'Before you start', 'Get an AdSense account', 'Get familiar with AdSense', 'Choose your client library', 'Register your application', and 'Quick start tutorial'. The main content area features a large heading 'Get Started' with a five-star rating. It includes a paragraph about the document being for developers who want to use the AdSense Management API to get information about their AdSense account. Below this, there are three sections: 'Before you start' (with a sub-section 'Get an AdSense account' which explains the need for a test account), 'Get familiar with AdSense' (with a sub-section 'introductory information on AdSense' and instructions to experiment with the user interface), and a summary of the steps involved.

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



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 are irrelevant to other users. Focusing in a specific language is often a tradeoff.

Watson and IBM Cloud

The screenshot shows the IBM Cloud Docs interface. The left sidebar has a 'Watson' section under 'LEARN' with a link to 'Getting started with Watson and IBM Cloud'. Below it are sections for 'IBM Cloud development approaches', 'Programming models for Watson services', and 'Additional resources'. Under 'HOW TO', there are links for 'Service credentials for Watson services', 'Tokens for authentication', 'Controlling request logging for Watson services', and 'Using Watson SDKs'. A 'REFERENCE' section at the bottom links to 'Waiting for console.bluemix.net...'. The main content area has a search bar with 'watson' typed in. The page title is 'Getting started with Watson and IBM Cloud', last updated on 2018-01-04. It contains two main steps: 'Step 1: Getting a free IBM Cloud account' and 'Step 2: Finding a starter kit or service'. A 'Table of contents' sidebar on the right lists 'Step 1: Getting a free IBM account', 'Step 2: Finding a starter kit or service', 'Step 3: Creating a project', and 'Next steps: Starting to code'.

Watson and IBM Cloud getting started tutorial

The Watson and IBM Cloud getting started tutorial lists 3 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.

Authentication and authorization requirements

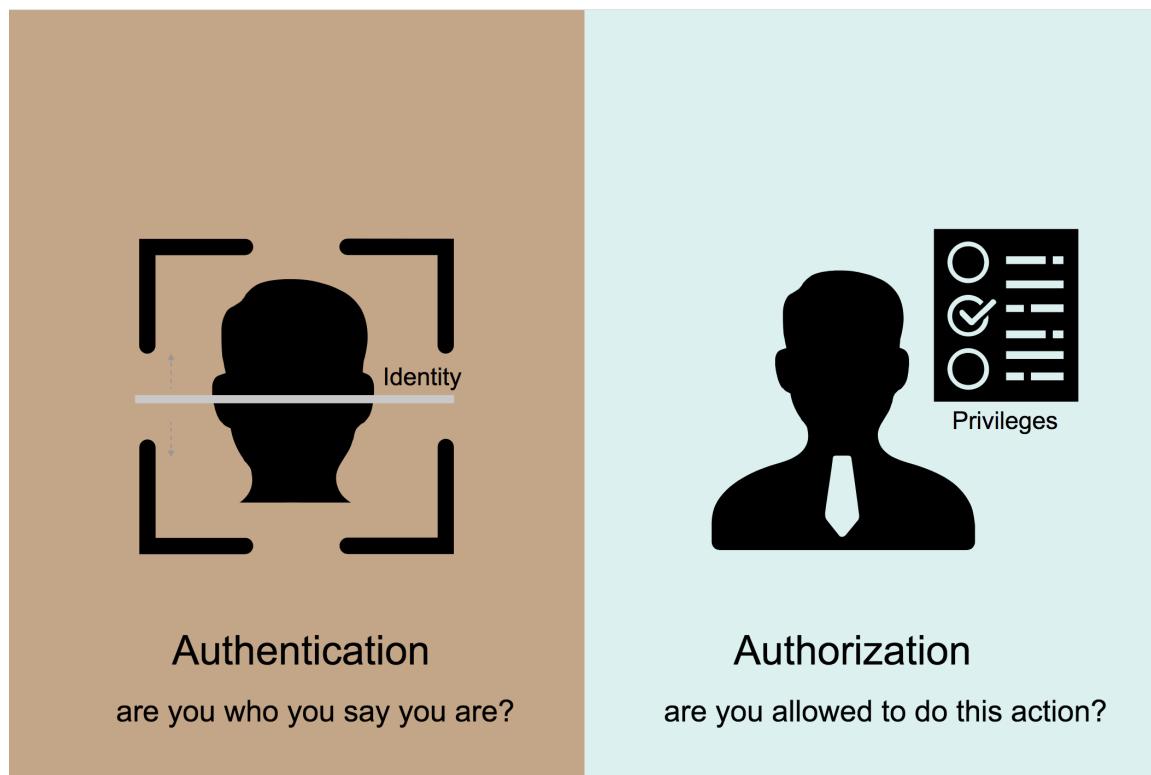
Before users can make requests with your API, they'll usually need to register for some kind of application key, or learn other ways to authenticate the requests. APIs vary in the way they authenticate users. Some APIs require you to include an API key in the request header, while other APIs require 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.



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 kind of security with your API, users could make unlimited amounts of API calls without any kind of registration. This 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 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 \(page 293\)](#)
- Apply different permission levels to different users

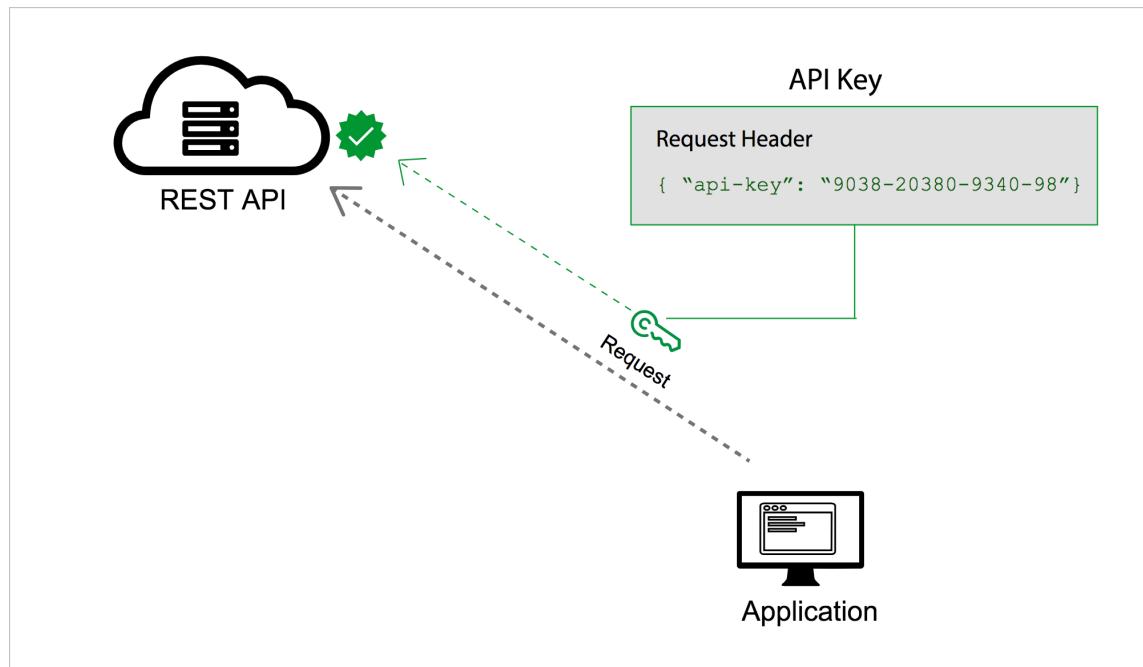
Different types of authorization

There are several different methods for authorization. The following are various types of API authorization you might encounter:

- [API keys \(page 278\)](#)
- [Basic Auth \(page 279\)](#)
- [HMAC \(page 280\)](#)
- [OAuth \(page 281\)](#)

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.



API 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 is 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 bG9sOnNlY3VzQ==
```

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, 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)

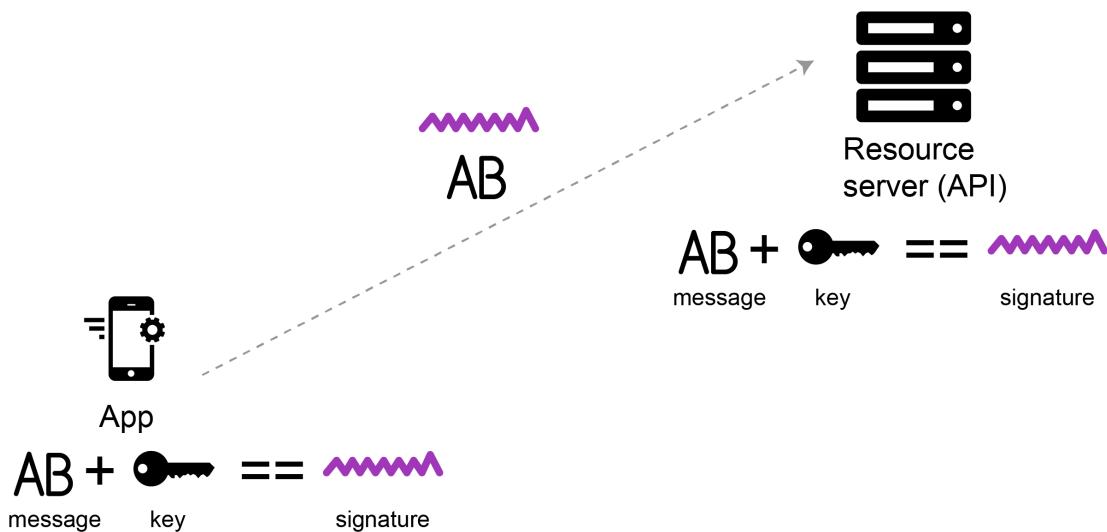
HMAC stands for Hash-based message authorization code and is a stronger type of authentication, more common in financial APIs. With HMAC, both the sender and receiver know a secret key that no one else does. The sender creates a message based on some system properties (for example, the request timestamp 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



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 in order 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.



OAuth login window

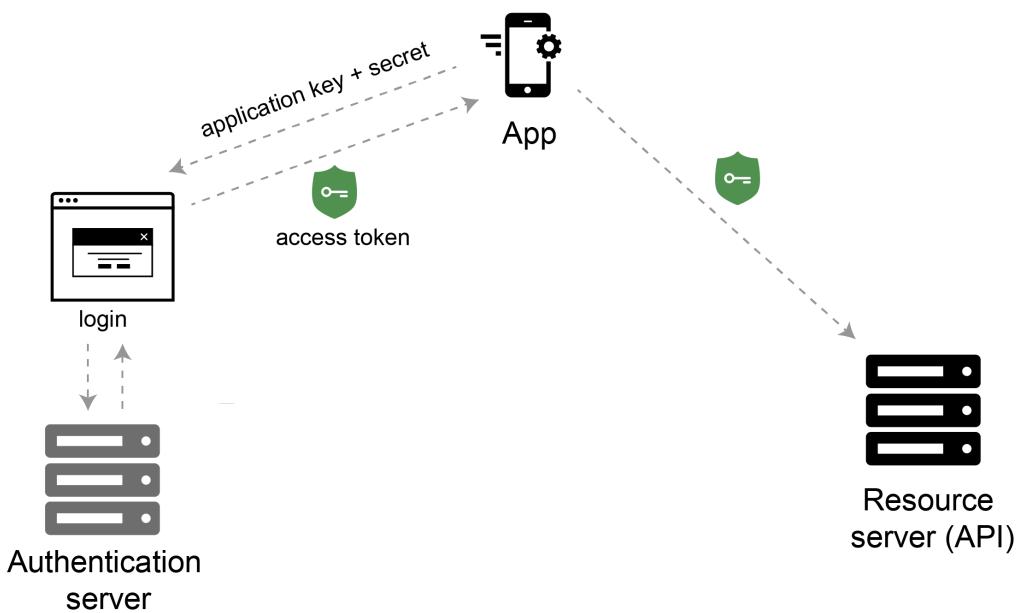
There are a few varieties of OAuth — namely, “one-legged OAuth” and “three-legged OAuth.” One-legged OAuth is used when you don’t have sensitive data to secure. This might be the case if you’re just retrieving general, read-only information (such as news articles).

In contrast, three-legged OAuth is used when you need to protect sensitive data. There are three groups 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 2.0 uses an access token from an authentication server



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, they 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 basic information such as:

- How to get API keys
- How to authenticate requests

- Error messages related to invalid authentication
- Rate limits with API requests
- Potential costs surrounding API request usage
- Token expiration times

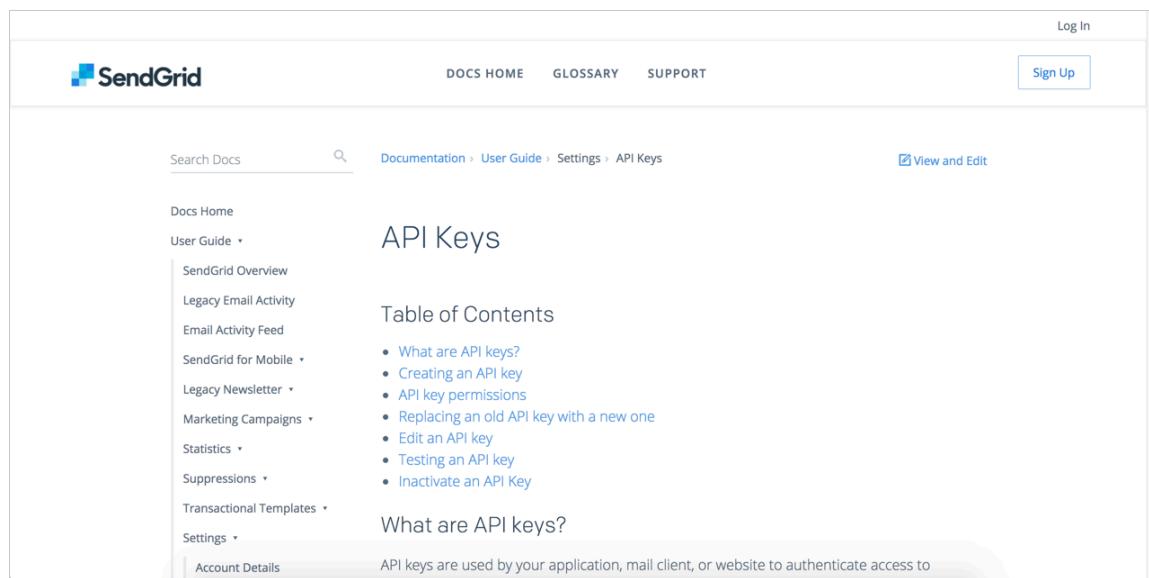
If you have public and private keys, you should explain where each key should be used, and that private keys should not be shared. If different license tiers provide different access to the API calls you can make, these licensing tiers should be explicit in your authorization section or elsewhere.

Since the API keys section is usually essential before developers can start using the API, this section needs to appear in the beginning of your help.

Samples of authorization sections

The following are a few samples of authorization sections in API documentation.

SendGrid



The screenshot shows the SendGrid documentation website. At the top, there's a navigation bar with links for 'Log In' and 'Sign Up'. Below the navigation, there's a search bar labeled 'Search Docs' and a breadcrumb trail showing the path: 'Documentation > User Guide > Settings > API Keys'. To the right of the breadcrumb trail is a 'View and Edit' button. The main content area has a title 'API Keys' and a 'Table of Contents' sidebar. The sidebar lists various sections under 'User Guide': 'Docs Home', 'SendGrid Overview', 'Legacy Email Activity', 'Email Activity Feed', 'SendGrid for Mobile', 'Legacy Newsletter', 'Marketing Campaigns', 'Statistics', 'Suppressions', 'Transactional Templates', 'Settings', and 'Account Details'. The 'API Keys' page itself contains a section titled 'What are API keys?' with a subtext explaining that API keys are used for authentication. A list of topics under 'Table of Contents' includes: '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', and 'Inactivate an API Key'.

SendGrid API keys

SendGrid offers a detailed explanation of API keys, starting out with the basics by explaining, “What are API keys?” Contextually, the topic on API keys appears with other account management topics.

Twitter

The screenshot shows the Twitter Developer Documentation website. The top navigation bar includes links for Developer, Use cases, Products, Docs, More, and a search bar. The main content area has a title "Authentication" and a "Basics" sidebar with links like Getting started, Authentication, Rate limits, Response Codes, and Security. The main content area features tabs for Overview, Guides (which is selected), and API Reference. Under the "Guides" tab, there's a "Guides contents" section with links to Authorizing a request, Percent encoding parameters, TLS, Creating a signature, Single user OAuth with examples, Access tokens from apps.twitter.com, and OAuth FAQ. Below this, a section titled "Authorizing a request" explains the purpose of the document.

Twitter authorization

With Twitter, because the OAuth 2.0 authorization requirements are a bit more involved, a detailed example is warranted and provided.

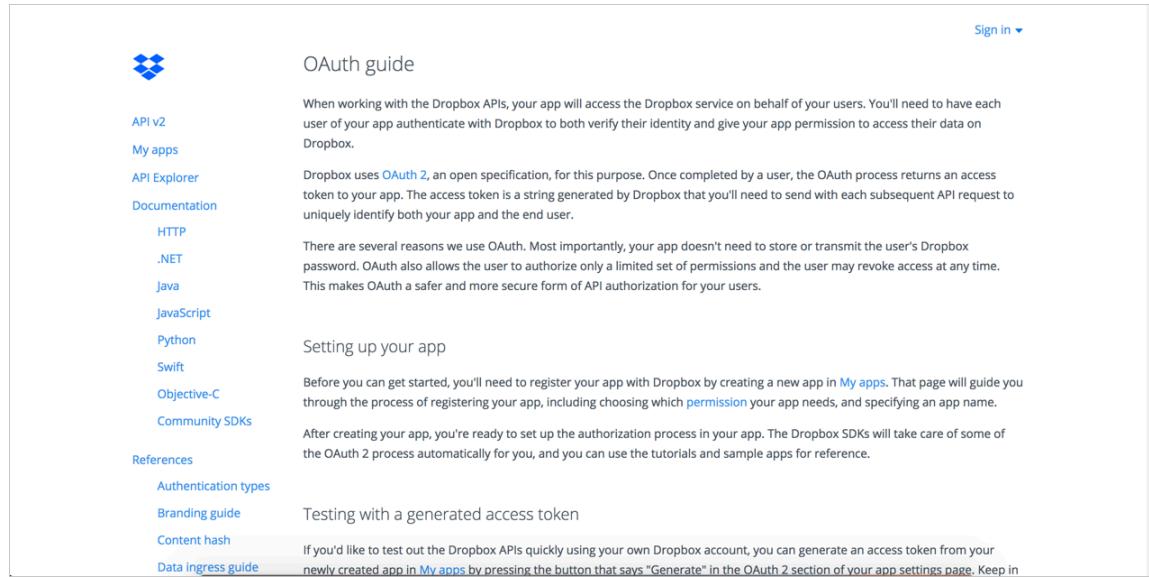
Amazon Web Services

The screenshot shows the AWS Product Advertising API documentation page. The top navigation bar includes a "Menu" icon, the AWS logo, a search bar, language selection (English), and a "Sign In to the Console" button. The main content area is titled "HMAC-SHA256 Signatures for REST Requests". It features a "Contents" sidebar with links to Welcome, Programming Guide, E-Commerce and Web Services, Product Advertising API Terminology and Basic Concepts, Visual Introduction to Product Advertising API, Organization of Items on Amazon, Requests, and Anatomy of a REST Request. The main content area describes how the API uses HMAC-SHA256 signatures for REST requests and details the "Authentication Parameters" required for REST authentication.

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



The screenshot shows the "OAuth guide" page from the Dropbox developer documentation. At the top right, there is a "Sign in ▾" button. The main content area has a heading "OAuth guide". Below it, a paragraph explains that when working with the Dropbox APIs, the app will access the Dropbox service on behalf of users. It requires each user to authenticate with Dropbox to verify their identity and grant permission to the app. The text then describes OAuth 2.0, noting it's an open specification that returns an access token to the app. The access token is a string generated by Dropbox to uniquely identify both the app and the end user. It highlights that OAuth makes the process safer and more secure by not storing or transmitting the user's Dropbox password. The "HTTP" section provides links for .NET, Java, JavaScript, Python, Swift, Objective-C, and Community SDKs. The "References" section includes links for Authentication types, Branding guide, Content hash, and Data ingress guide. The "Setting up your app" section provides instructions for registering the app with Dropbox and setting up the authorization process. The "Testing with a generated access token" section explains how to test the APIs using a generated access token from a newly created app.

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.

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 an 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 definitely 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 [an earlier lesson \(page 47\)](#)? 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=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

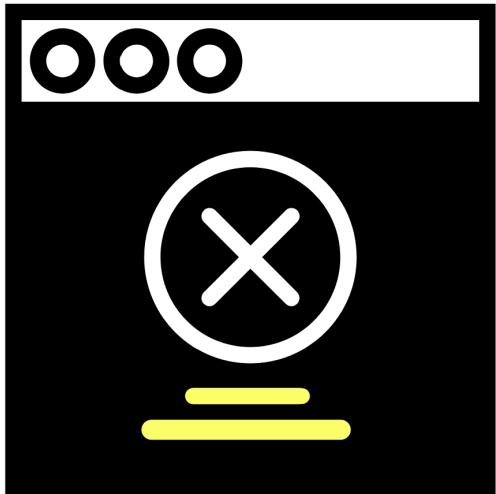
The response header looks as follows:

```
HTTP/1.1 200 OK  
Server: openresty  
Date: Mon, 02 Apr 2018 01:14:13 GMT  
Content-Type: application/json; charset=utf-8  
Content-Length: 441  
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.

```
<a target="_blank" class="noExtIcon" href="">
```



200 ok
400 bad request
404 not found
403 forbidden
500 server error

Error codes/messages should help users recover from failure.

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 will 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 \(page 253\)](#) to see all the potential error codes. For example, if you exceed the [rate limit \(page 293\)](#) 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 sort of 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

The screenshot shows the Context.io API documentation page for errors. The left sidebar has a dark background with orange links: 'GETTING STARTED', 'LITE API', '2.0 API (DEPRECATED)', 'APP LEVEL ENDPOINTS', 'ERRORS' (which is orange and underlined), 'FAQS', and 'MOBILE APPLICATIONS'. Below these are links to 'Get a free developer key!' and 'Documentation Powered by Slate'. The main content area has a light gray background with a title 'Errors'. It says: 'Here is a list of error codes you may encounter in Context.IO and what they mean for working with our API.' Below this is a table with two columns: 'Error Code' and 'Meaning'. The table rows are:

Error Code	Meaning
400	Bad Request – There is an issue in how the request was formed, you're using a wrong parameter, or passing in incorrect data in a parameter
401	Unauthorized – Your API key is wrong or you are not signing the request correctly
402	Payment Required – You've exceeded your user limit (see removing initial user limit)
403	Forbidden – You are passing a resource ID for the wrong version of the API (i.e. using 2.0 resource ID on Lite or vice-versa)
404	Not Found – The specified resource could not be found, please ensure this resource exists
409	Conflict – There is already a resource with these same criteria. (i.e. you are attempting to create a duplicate resource)
412	Pre-condition failed – Account or user is currently in DISABLED status and we cannot make calls against the mailbox (see DISABLED status)
429	Too Many Requests – You're exceeding your API rate limit (see API rate limits)
451	Unavailable for Legal Reasons – You are trying to add a user in an unsupported region

Context.io status and error codes

Clearbit not only documents the standard status codes, they describe 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

The screenshot shows the Twitter Developer API documentation for Response Codes. At the top, there's a purple navigation bar with links for Developer, Use cases, Products, Docs, More, and a search bar. Below the bar, the title "Response Codes" is displayed in a large, bold, black font. A horizontal line separates the title from the content. Under the title, the section "Basics" is shown in bold. To its right, a note states: "The standard Twitter API returns HTTP status codes in addition to JSON-based error codes and messages." Below this, the section "HTTP Status Codes" is introduced with the note: "The Twitter API attempts to return appropriate [HTTP status codes](#) for every request." A table follows, listing various status codes with their descriptions:

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 their status code documentation, Twitter not only describes the code and status, they 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

The screenshot shows the Mailchimp API Error Glossary page. It includes a sidebar with links to developer documentation, playground, and API status. The main content area has a section titled "Error Glossary" with a "Guides:" sidebar. The "400" section contains entries for **BadRequest**, **InvalidAction**, and **InvalidResource**. The "403" section contains entries for **Forbidden**, **NotAuthenticated**, **NotAuthorized**, and **RateLimitExceeded**. The "429" section contains entries for **RateLimitExceeded**.

Mailchimp status and error codes

Mailchimp provides extremely 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

The screenshot shows the Flickr REST API Error Codes page. The top navigation bar includes "Sign Up", "Explore", "Create", a search bar, and "Sign In". Below the navigation, a note states: "This is the list of new photo upload errors. This response is formatted in the REST API Response style." The main content is titled "Error Codes" and lists various error codes with their descriptions:

- 0: Video uploads are temporarily disabled**
- 2: No photo specified**
- 3: General upload failure**
- 4: Filesize was zero**
- 5: Filetype was not recognised**
- 6: User exceeded upload limit**
- 7: User exceeded video upload limit**
- 8: Filesize was too large**
- 9: Duplicate photo/video detected**
- 10: Not a valid url to upload from**
- 11: Fetch from external image failed**
- 12: Not a valid photo id to clone meta data from**
- 13: The clone photo does not belong to the uploader**
- 14: Auto Upload disabled for Non pro User/Server at capacity**

Each error code entry includes a detailed description of the error condition and its implications.

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 actually just repeated).

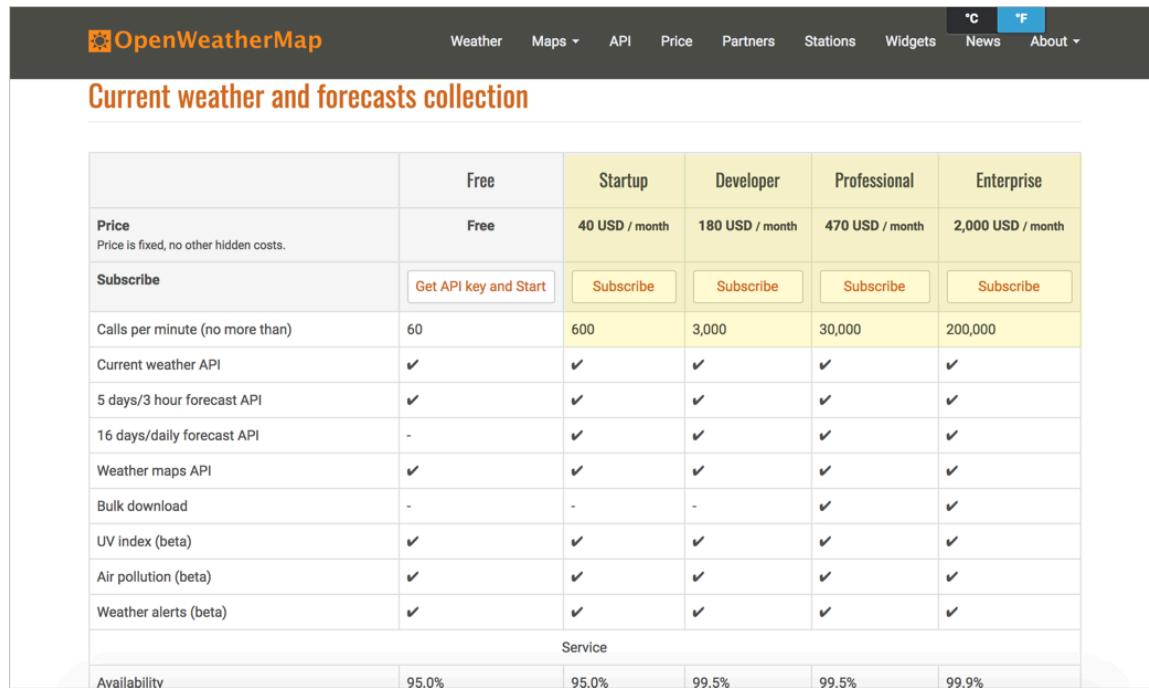
If some endpoints are prone to triggering certain status and error codes more than others, it makes sense to highlight those status and error codes on their relevant API reference pages. I recommend calling attention to any particularly relevant status or error codes on an endpoint's page, and then linking to the centralized page for full information.

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 \(page 32\)](#), you can see where the pricing tier begins:



The screenshot shows the OpenWeatherMap website's pricing section. At the top, there's a navigation bar with links for Weather, Maps, API, Price, Partners, Stations, Widgets, News, and About. Below the navigation is a heading 'Current weather and forecasts collection'. The main content is a table comparing six pricing tiers: Free, Startup, Developer, Professional, and Enterprise. The table includes columns for 'Price' (with a note about fixed costs), 'Subscribe' (with a 'Get API key and Start' button), and various service offerings like 'Calls per minute', 'Current weather API', '5 days/3 hour forecast API', etc. The table also includes a 'Service' row and an 'Availability' row at the bottom.

	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 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?
- Do the responses simply return a particular status code (if so, which one)?

Also, when developers implement the code into their applications or web pages, how are they implementing code for responses that don't provide data (due to the threshold being exceeded)? Are there conditions and checks to handle these scenarios? Does the widget (or whatever might be implementing the API) simply freeze or hang, display empty or crash?



Limits to Pro plan: 10,000 calls/day

Rate limiting might seem like a marketing topic, but actually the rate limiting policies and how they affect API calls has a significant impact on development.

Examples rate limiting sections

Here are a few examples of rate limiting sections in API documentation.

GitHub

```
<a target="_blank" class="noExtIcon" href="https://developer.github.com/v3/#rate-limiting">
```

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

GitHub rate limiting

GitHub's documentation explains rate limits for authenticated versus unauthenticated requests, the header returned and the meaning of the rate limiting titles ([X-RateLimit-Limit](#), [X-RateLimit-Remaining](#), and [X-RateLimit-Reset](#)), how to check your current usage, increasing the rate limit for a specific application, what happens when rate limits are abused, and more.

[Linkedin](#)

```
<a target="_blank" class="noExtIcon" href="https://developer.linkedin.com/docs/rest-api?u=0#">
```

Understanding request throttling

All REST API requests are throttled to prevent abuse and ensure stability. The exact number of calls that your application can make per day varies based on the type of request you are making. You will find this information along side the documentation for each specific API call.

There are three different kinds of throttles in place:

- Application Throttle — The total number of calls that your application can make in a day.
- User Throttle — The total number of calls that any unique individual member using your application can make in a day.
- Developer Throttle — The total number of calls that any user that is identified as a "developer" in the [Application's settings](#) can make in a day.

Please note that for the purposes of request throttling, the API server's "day" is defined as the 24 hour period beginning at midnight UTC and ending at midnight on the following day.

Handling Paged Responses

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](#)

[](http://dev.bitly.com/rate_limiting.html)

GETTING STARTED

API DOCUMENTATION

▼

[Transitioning From Google's URL Shortener](#)
[Authentication](#)
[Request / Response Formats](#)
[CORS \(Cross Origin Resource Sharing\)](#)
[Rate Limiting](#)
[Data APIs](#)
[Links](#)
[Link Metrics](#)
[User Info/History](#)
[User Metrics](#)
[Deeplink Metrics](#)
[Organization Metrics](#)
[Domains](#)
[Campaigns](#)
[Data Streams](#)
[Spreadsheet Support](#)
[Deprecated](#)
[API Console](#)
[BEST PRACTICES](#)

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:

- 10,000 new Bitlinks created per month
- 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 avoid 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.

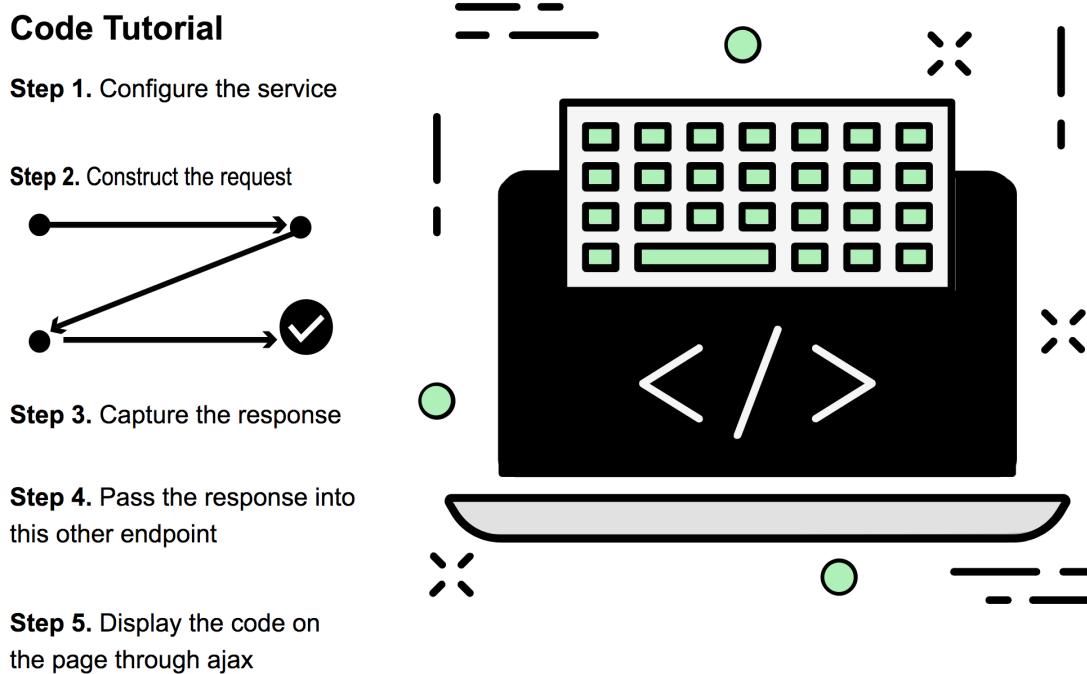
By looking at these examples, you can see that while rate limiting might seem like a simple, straightforward topic, there are layers of depth and complexity to cover. The relevance of the topic depends on your API and the rate limiting policies your company sets, but this information cannot be entirely offloaded to Marketing to handle. So much of the information around rate limiting directly affects development.

Code samples and tutorials

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 key part of your user guide.

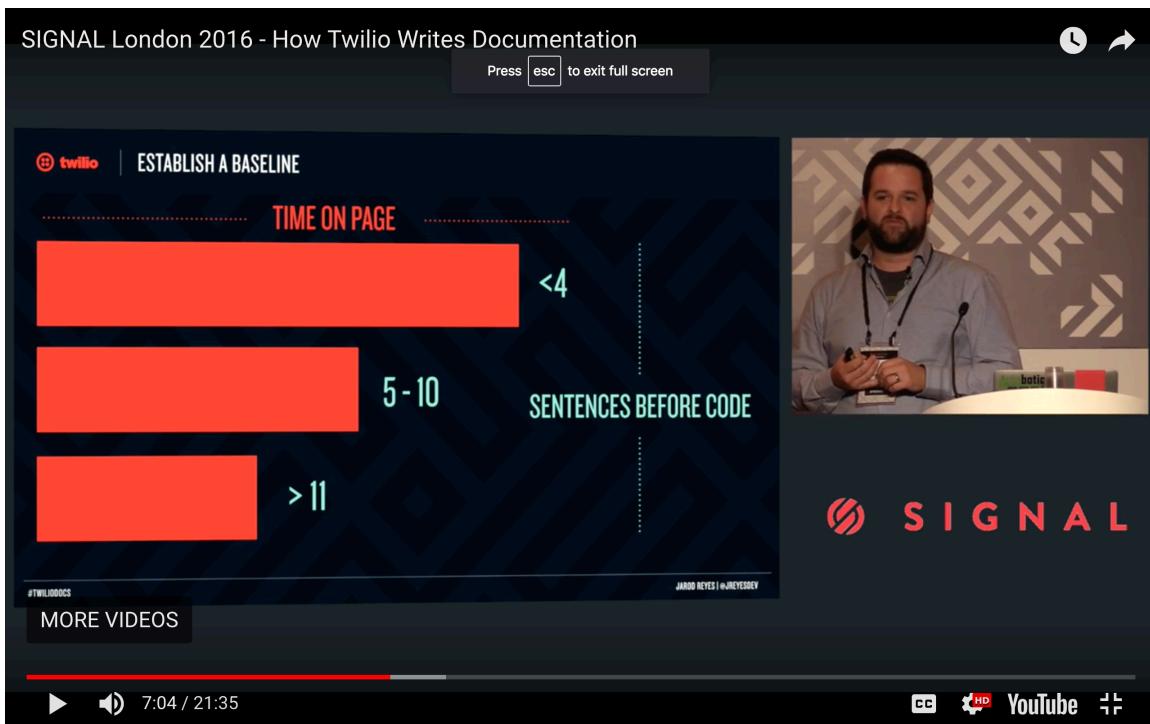
Code samples are like candy for developers

Code samples play an important 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 descriptive non-code text can't achieve.



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 developer 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.

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 or prose on a page and not a lot of code, that page didn't perform well. ([How Twilio Writes Documentation](#))

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.

Don't just provide reference docs

Sometimes engineers avoid code samples 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.

In other words, the articles and tutorials complement the reference documentation to provide more complete instruction. Code samples that show how to use the various endpoints to achieve a goal has a place in your user guide.

Not everyone agrees about the need for code samples. Developers on the product team often make the mistake of assuming that their developer audience has a skill set similar to their own, without recognizing different developer specializations. 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 skillset 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 are warranted.

Focus on the why, not the what

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 cross-origin resource sharing (CORS) for the weather response."

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.

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 complex 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 actually 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). This 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) so they could run regular tests on it. They pulled the code into their documentation where appropriate.

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 a [Postman collection \(page 39\)](#) or an [OpenAPI specification file \(page 143\)](#) — 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

The screenshot shows the Weather Underground API Documentation page. At the top, there's a navigation bar with links for Weather, Maps & Radar, Severe Weather, Photos & Video, Community, News, Climate, and a Sign In button. Below the navigation is a blue header bar with the word "DOCUMENTATION". Underneath, there's a menu bar with links for API Home, Pricing, Featured Applications, Documentation (which is currently selected), and Forums. On the left, there's a sidebar titled "API Table of Contents" under "Weather API" which lists various services like WunderMap Layers, Data Features, and Forecast. The main content area has a title "Code Samples" and two sections: "PHP" and "Ruby". Each section contains a code snippet. The "Page Contents" sidebar on the right lists "Code Samples" and links for PHP, Ruby, Python, ColdFusion, and JavaScript with jQuery.

```

<?php
$json_string =
file_get_contents("http://api.wunderground.com/api/Your_Key/geolookup/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

The screenshot shows the Eventful API documentation website. The header includes the Eventful logo, a sign-in/register link, and navigation tabs for Overview, Tools, and Technical Reference. The main content area is titled "Tutorial: Search". It contains sections on searching by what, where, and when, with examples and URLs. A sidebar on the left lists various documentation links under categories like Overview, Tools, and Technical Reference.

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 in a more narrative way with examples. It shows more of the difference between reference and tutorial information.

Twilio

```
<a target="_blank" class="noExtIcon" href="https://www.twilio.com/docs/quickstart">
```

The screenshot shows the Twilio Docs website with a dark blue header. The header includes a logo, navigation links for 'Docs', 'Quickstart', 'Tutorials', 'API Reference', and 'SDKs', and a 'LOG IN' button. Below the header, a breadcrumb trail reads 'Twilio Docs > Quickstarts'. On the left, there's a sidebar with two sections: 'PRODUCT' and 'LANGUAGE'. Under 'PRODUCT', buttons are available for 'ALL', 'SMS', 'VOICE', 'VIDEO', 'TOOLS', 'TASKROUTER', 'CHAT', 'SYNC', 'AUTHY', and 'FUNCTIONS'. Under 'LANGUAGE', buttons are available for 'ALL', 'C#', 'JAVA', 'NODE.JS', 'PHP', 'PYTHON', 'RUBY', 'JAVASCRIPT', 'JAVA (ANDROID)', 'OBJECTIVE C', and 'SWIFT'. The main content area displays the 'Programmable SMS Quickstart' page, which includes a brief description, a list of supported languages (PHP, C#, Java, Node.js, Python, Ruby), and a link to the 'Programmable Voice Quickstart' page.

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 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 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 marvel that I haven't seen anywhere else.

Mailchimp

Guides:

- Getting Started
- Intro to REST
- About the Playground
- Best Practices
- Manage Subscribers**
- E-Commerce API Guide
- Batch Operations
- Changelog
- How to Use OAuth2
- Error Glossary
- About Webhooks
- Export API

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 an easier formatting, and you can still preface each section heading with “Step 1”, “Step 2”, and so on.

IBM Watson

The screenshot shows a web browser window for 'IBM Cloud Docs'. The left sidebar has a 'Watson Assistant' icon and sections for 'LEARN' (Getting started tutorial, About, Video: Tool overview for Watson Assistant, Demo: Car Dashboard, Additional resources) and 'HOW TO' (Configuring a Watson Assistant workspace, Defining intents and entities, Building a dialog, Deploying, Improving understanding). The main content area shows a search bar with 'watson-assistant' and a result for 'Tutorial: Building a complex dialog'. The tutorial title is 'Tutorial: Building a complex dialog' with a link to 'Edit in GitHub'. It was last updated on 2018-03-23. The content describes creating a dialog for a cognitive dashboard. It includes 'Learning objectives' (Define entities, Plan a dialog, Use node and response conditions in a dialog), 'Duration' (approximately 2 to 3 hours), and a note about prerequisites.

IBM Watson code samples

The IBM Watson tutorial does a nice 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 difficult 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 \(page 79\)](#) 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 shows 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>
var settings = {
    "async": true,
    "crossDomain": true,
    "url": "https://api.openweathermap.org/surfreport/25&days=1",
    "method": "GET"
}

$.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 in order 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.

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.

For a deeper dive into how to document code samples, see my presentation on [Creating code samples for API/SDK documentation](#).

SDKs and sample apps

SDKs (software development kits) and sample apps are similar to [code samples and tutorials \(page 298\)](#) but are much more extensive and usually involve a whole collection of files that work together as a package or app. The SDK might include libraries that you download and incorporate into your application, and can include tools, sample apps, and other code. Sample apps are usually self-contained applications that implement the API for a specific scenario in a specific programming language.

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 \(page 47\)](#), 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 \(page 109\)](#)). 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.

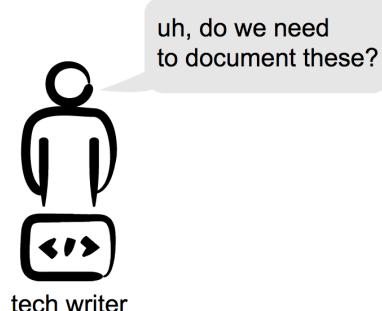
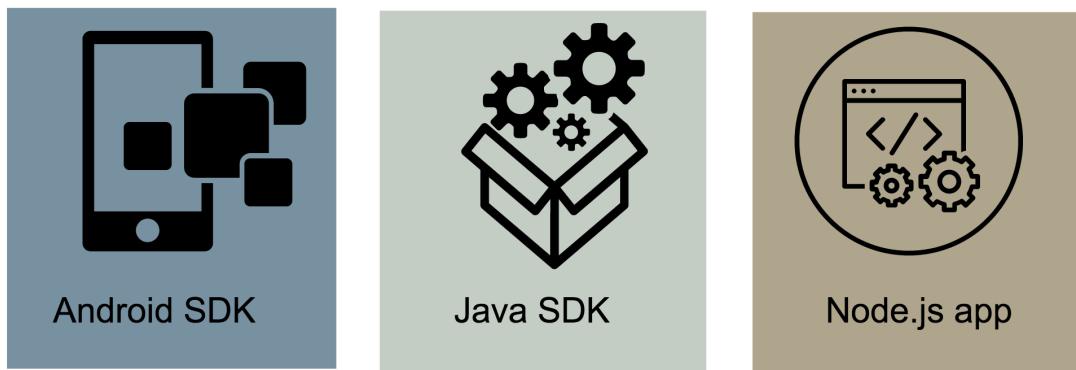
An SDK or devkit functions in much the same way, providing a set of tools, libraries, relevant documentation, code samples, processes, and or guides that allow developers to create software applications on a specific platform. If an API is a set of building blocks that allows for the creation of something, 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 (later in this course), I show how to [auto-generate client SDKs \(page 228\)](#) 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 explored the question of [how much code you need to know \(page 443\)](#) in another topic, 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 other 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 them 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 topic \(page 298\)](#), 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 sort of requires you to understand Java.)

Sample SDKs and sample apps

The following examples show documentation for some sample SDKs and sample apps.

OpenWeatherMap API

The screenshot shows the OpenWeatherMap homepage. At the top, there's a navigation bar with links for Weather, Maps, API, Price, Partners, Stations, Widgets, News, and About. Below the navigation, there are two main sections. The first section is titled "Google Weather-Based Campaign Management with OpenWeatherMap API" and discusses how demand for products varies by weather. It includes a link to the source code on Google Developers. The second section is titled "Google Maps JavaScript API based on OpenWeatherMap API" and notes that Google has closed their Weather and Cloud solutions, recommending OpenWeatherMap as an alternative. It also includes a GitHub link.

- 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 make a request to 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.

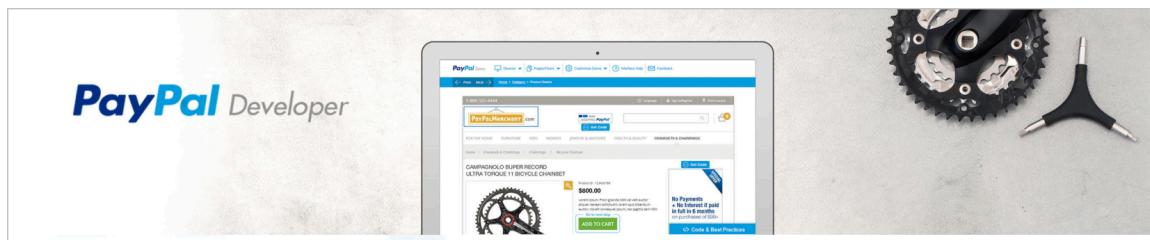
If you can put your sample apps and SDKs on GitHub, it's usually a good idea to do so. Storing code on GitHub accomplishes two purposes: First, it usually puts the burden on engineering to maintain and test the code samples as well as respond to issues users might log against the project. Second, it 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

The screenshot shows the PayPal Developer website's REST APIs section. The left sidebar has a navigation menu with 'REST APIs' selected. Under 'GET STARTED', there is a 'Get Started' link. Under 'API REFERENCE', there are links for 'Activities API', 'Billing Agreements API', 'Billing Plans API', 'Customer Disputes API', 'Identity API', 'Invoicing API', 'Managed Accounts API', and 'Orders API'. The main content area is titled 'REST SDK Quickstart' and contains a brief introduction: '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.' Below this is a section titled 'How to use this guide' with two numbered steps: 'To set up your environment, complete the [required setup steps](#)' and 'Modify the sample code to [issue standard payments](#)'. A small note at the bottom says '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	

 [Github](#)  [PayPal API Reference](#)  [Report Issues](#)

Paypal PHP SDK

Heroku SDK

[](https://devcenter.heroku.com/articles/pubnub)

The screenshot shows the Heroku Dev Center interface. At the top, there's a purple header with the Heroku logo, a 'Get Started' button, and links for 'Documentation', 'Changelog', and 'More'. On the right, there's a search bar, a 'Log in' button, and a 'Sign up' button. Below the header is a sidebar titled 'CATEGORIES' with links like 'Heroku Architecture', 'Deployment', 'Command Line', etc., and a section for 'Add-ons' which is currently selected. The main content area is titled 'PubNub' and contains a sub-header 'This add-on is operated by PubNub' with the tagline 'Realtime Communication for IoT, Mobile and Web'. Below this, there's a breadcrumb navigation 'Add-ons > All Add-ons > PubNub'. The main content is titled 'PubNub' and includes a 'Last updated 04 May 2016' timestamp. It features a 'Table of Contents' with several items listed.

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 lingo and concepts.

If engineers tell you that users should know X, don't simply 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 then 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.

Amazon SDK

[](https://aws.amazon.com/tools/#sdk)

The screenshot shows the AWS SDKs page. At the top, there's a navigation bar with 'Menu', the AWS logo, 'Contact Sales', 'Products', 'Solutions', 'More', 'English', 'My Account', and a 'Sign In to the Console' button. On the left, there's a sidebar with 'RESOURCES' and links for 'Tools' (selected), 'AWS ElasticWolf Client Console', and 'RELATED LINKS' (Documentation, AWS Code Services). Below the sidebar are buttons for 'Manage Your Resources' and 'Sign In to the Console'. The main content area is titled 'SDKs' and contains a sub-header: 'Simplify using AWS services in your applications with an API tailored to your programming language or platform.' It lists several programming languages with their respective SDKs: Java (.NET, Node.js), PHP, Python, Ruby, Browser (Go), C++, and AWS IoT Device SDK. Each language entry includes 'Install' and 'Documentation' links.

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 annotation syntax and document generation tools. The annotation syntax (which programmers use directly in the code — see [Javadoc tags \(page 0\)](#) for an example of Javadoc tags) differs by language and tool but is largely 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 common 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

The screenshot shows the AWS SDKs page. At the top, there's a navigation bar with 'Menu', the AWS logo, 'Contact Sales', 'Products', 'Solutions', 'More', 'English', 'My Account', and a 'Sign In to the Console' button. On the left, a sidebar titled 'RESOURCES' has sections for 'Tools' (selected), 'AWS ElasticWolf Client Console', and 'RELATED LINKS' (Documentation, AWS Code Services). Below the sidebar are two buttons: 'Manage Your Resources' and 'Sign In to the Console'. The main content area is titled 'SDKs' and contains a sub-header: 'Simplify using AWS services in your applications with an API tailored to your programming language or platform.' It lists nine programming languages with their respective 'Install' and 'Documentation' links:

Language	Install	Documentation
Java	Install »	Documentation »
.NET	Install »	Documentation »
Node.js	Install »	Documentation »
PHP	Install »	Documentation »
Python	Install »	Documentation »
Ruby	Install »	Documentation »
Browser	Install »	Documentation »
Go	Install »	Documentation »
C++	Install »	Documentation »
AWS IoT Device SDK	Install »	

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.

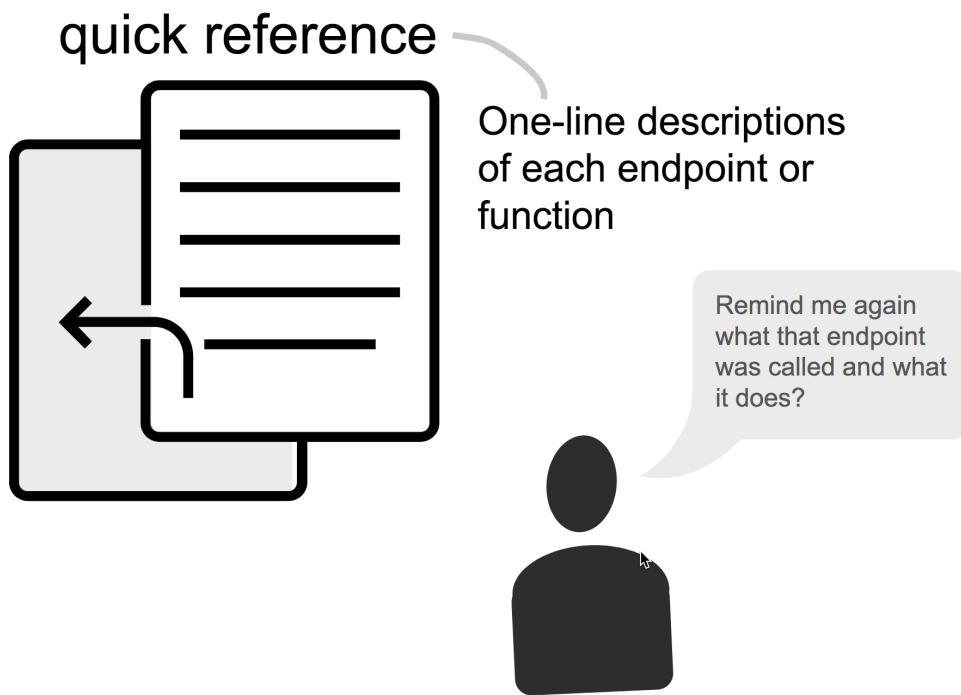
Looking at the Google Cloud SDK versus the Amazon SDK shows some of the breadth and variety of technologies you might have to document in SDK territory. These SDKs are specific to a particular programming language, operating system, or other framework, and as such, it can be daunting to try to ramp up in order to document this category of tools. For SDK documentation, you'll need to work closely with engineers and listen to feedback from users.

Quick reference guide

Quick reference guides serve a different function than [getting started tutorials \(page 269\)](#). While the getting started tutorial helps beginners get oriented by providing an beginning-to-end instruction to make a simple API request, with API documentation 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

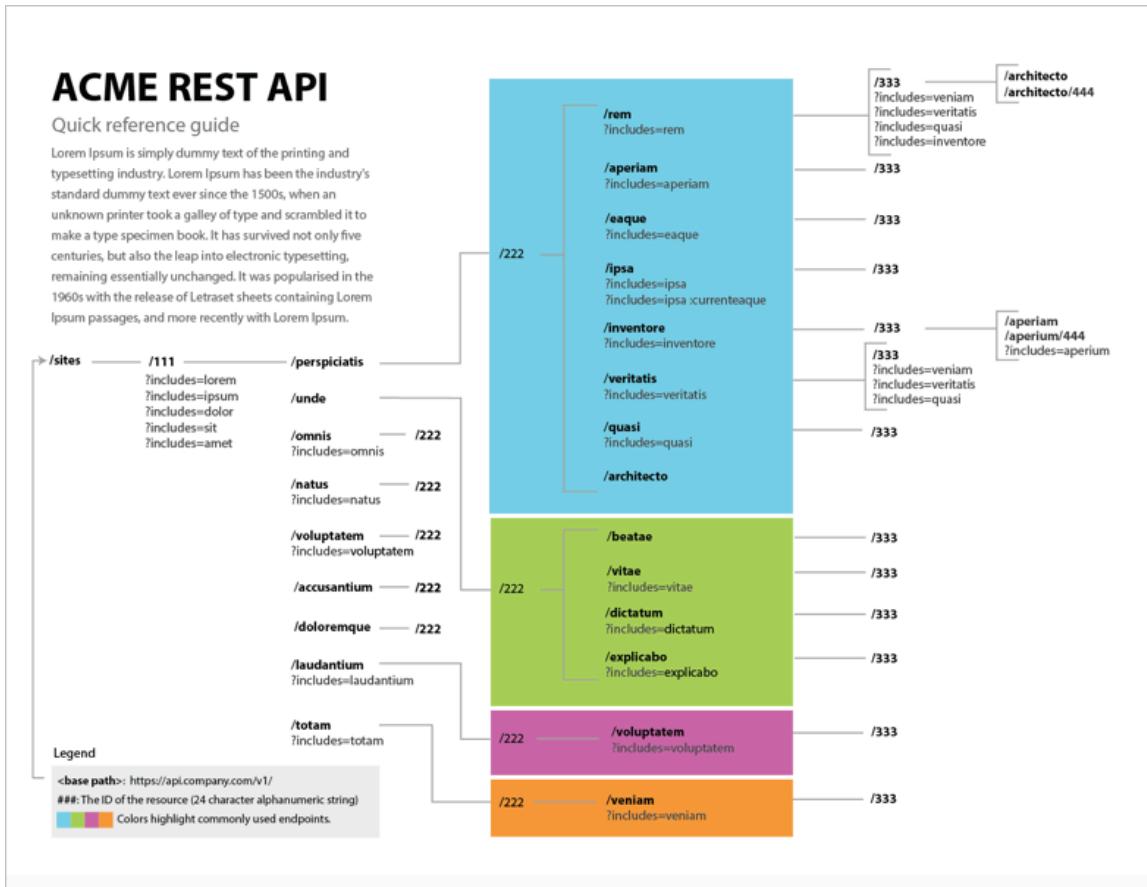
Whether for end-user documentation or developer documentation, the quick reference guide provides a 1-2-page guide that provides a brief summary of the core tasks and features in the system.



Quick reference 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 times with APIs, the endpoints have relationships with each other that can be depicted visually. Here's an API diagram I created at a previous company:

[images/sample_api_diagram.pdf](#)



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 a logical harmony.

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:

```
<a target="_blank class="noExtIcon" href="images/sample_qrg.png">
```



Lorum Ipsum
Quick Reference Guide

Im alit aliquatet, consecte
minciduissim at nons ating er incin ulla adip
essequi pismod exeros augiat velenis augit
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:

- duis acinim volorper alit iliquat umsandigna facil
dolor at, sectetum venisi.
2. Lorem ipsum dolor solaritarut est tudat
repuslare in gratidueni la palabrate entotum.



Quatumsan Venisi

Schedulare Voloper Est Tudat

- Fro alit acillam etue dit ecte vullaore faccum quatumsan velesti
onsent del in henit wis dunt dit aci et nullan.
- Olesse dipismo doloboreet ipsummy nibh elit non henim colummym
num zzrit ate faccum zzriliquam, quam quip erostis del er sed.
- Ex et, sumsandigna facil dolor at, sectetum venisi.

Raesequis eu feugiate

- Famoconsequi blaorem nulla ad tisit dolessi exeraessi.
- Camconsequi blaorem nulla ad tisit dolessi exeraessi:
✓ Dio eugimocons nos esto: dio eugiamcons nos esto
consequis amconsequi blaorem nulla ad tisit dolessi exeraessi.
- █ Faciduiscint ea com num: consequis amconsequi blaorem
nulla ad tisit dolessi ex. consequis amconsequi blaorem nulla 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

- d tinit wis diat velenit
er ipit non hent augait,
quis ex etue fe.
- Cd tinit wis diat velenit
er ipit non hent augait,
quis ex et
- is diat velenit er ipit
non hent augait, quis
ex etue feum iustrud te
feugait accum colummo
lorercilis
- Strud te feugait accum
colummo lorercilis do
odipsus tionum ius.

Diat velenit er ipit

- an hendip ea coreetue tie
velenis alit acillam etue
- wis dunt dit aci et nullan
volesse
- dunt dit aci et nullan
volesse dipismo
doloboreet ipsum

Selenit er ipit

- dunt dit aci et nullan
volesse dipismo
doloboreet ipsum
- quatumsan velesti onsent
del in henit wis dunt dit
aci et nullan volesse
dipismo doloboreet
- ipsummy nibh elit non
henim colummym num
zzrit ate
- 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
colummo lorercilis do
odipsus tionum ius
- Quics ex etue feum iustrud
te feugait accum colummo
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](#). 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

[Sign in | Register](#)

[Overview](#) [Tools](#) **Technical Reference**

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/cal**

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 descriptive definition is “Given an event ID, returns the event data associated with that event. See <http://eventful.com/events/E0-001-000278174-6> (page 0) for an example interface.”

There’s a certain value 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

The screenshot shows a "Quick Reference" page for Parse. On the left, there's a sidebar with a search bar and a list of API categories. The main content area has several sections of text about API access, configuration, and specific API details like Objects API.

Search...

Your Configuration
Getting Started
Quick Reference

- Objects API
- Users API
- Sessions API
- Roles API
- Files API
- Analytics API
- Push Notifications API
- Installations API
- Cloud Functions API
- Schemas API
- Function Hooks API
- Trigger Hooks API
- Request Format
- Response Format
- Calling From Client Apps
- Objects
- Queries
- Users
- Sessions
- Roles

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 **YOURPARSE-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

The screenshot shows the Shopify Cheat Sheet interface. At the top, there are links for "Timber framework", "Docs", "Snap Flyers", and a button to "Create an online store". Below these are four main sections:

- Liquid**: This section contains a list of Liquid tags under the heading "Logic". It includes tags like `comment %`, `raw %`, `if %`, `unless %`, `case %`, `cycle %`, `for %`, `table-row %`, `assign %`, `increment %` (marked as NEW), `decrement %` (marked as NEW), `capture %`, and `include %`. Below this is a section for "Operators" with symbols like `==`, `!=`, `>`, `<`, `>=`, `<=`, `or`, and `and contains`.
- Liquid Filters**: This section lists various Liquid filters. Examples include `escape(input)`, `append(input)`, `prepend(input)`, `size(input)`, `join(input, separator = '')`, `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)`, and `Split`.
- Template variables**: This section shows variables available in `blog.liquid`. These include `blog['the-handle'].variable`, `blog.id`, `blog.handle`, `blog.title`, `blog.articles`, `blog.articles_count`, `blog.comments_enabled?`, `blog.moderated?`, `blog.next_article`, `blog.previous_article`, `blog.all_tags`, and `blog.tags`.
- Template variables**: This section shows variables available in `article.liquid`. These include `article.id`, `article.title`, `article.author`, `article.content`, and `article.created_at`.

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 let's you know all the functions that exist.

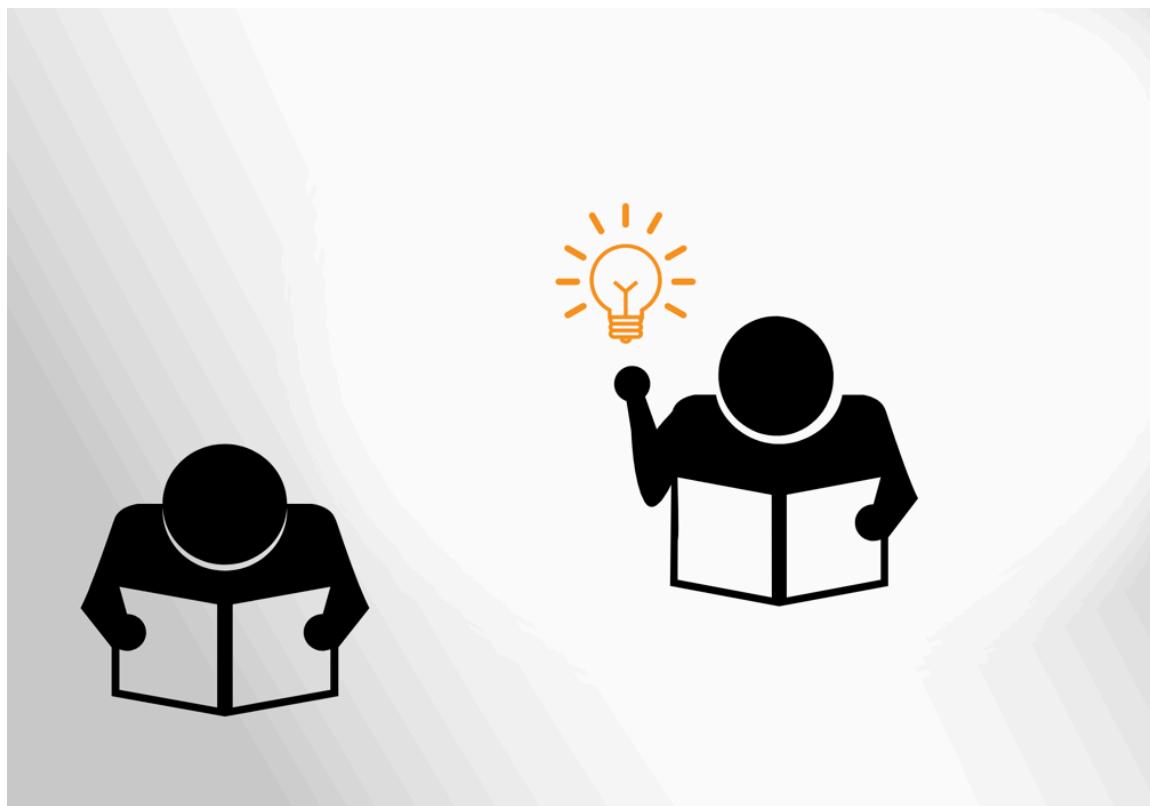
API best practices

API best practices can refer to any general advice your product team wants to communicate to developers about working with the API. There's no set number of topics or content that is 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

<a target="_blank" class="noExtIcon" href="http://developer.mailchimp.com/documentation/mailchimp/

guides/mailchimp-api-best-practices/">

Guides:

- Getting Started
- Intro to REST
- About the Playground
- Best Practices**
 - Manage Subscribers
 - E-Commerce API Guide
 - Batch Operations
 - Changelog
 - How to Use OAuth2
 - Error Glossary
 - About Webhooks
 - Export API

MailChimp API 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

[](https://developers.coinbase.com/api/v2#pagination)

The screenshot shows the Coinbase Developers API documentation. The left sidebar has a navigation menu with sections like Introduction, Authentication, Interacting with the API, Scopes, Pagination (which is currently selected), Errors, Versioning, Rate limiting, Changelog, API Client Libraries, Expanding resources, Notifications, Wallet Endpoints, Users, Accounts, Addresses, Transactions, Buys, Sells, and Documentation. The main content area is titled "Pagination". It contains a detailed description of cursor-based pagination, a curl command for making a GET request to the accounts endpoint, and an example response in JSON format.

Mailchimp best practices

Coinbase doesn't specifically refer to these topics as best practices; instead the navigation just shows a laundry list of topics. Pagination is one of these topics worth expanding on here. What is pagination in relation to an API? 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 10 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 \(page 28\)](#), or “Hypermedia as the Engine of Application State.” Links in responses that return more results is one example.

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.

Glossary

The glossary defines all the terms that might be unique to your company or API. 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.

Define any words you invest

Unlike most other professional writing disciplines, tech docs are notorious for the amount of specialized terms in their content. Not only do we have unique terms related to our products, industry jargon and company-specific terms make their way into docs, driving up their complexity.

API evangelist Kin Lane recently noted his frustration with an API's language when he encountered an undefined acronym — "DEG". 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](#))

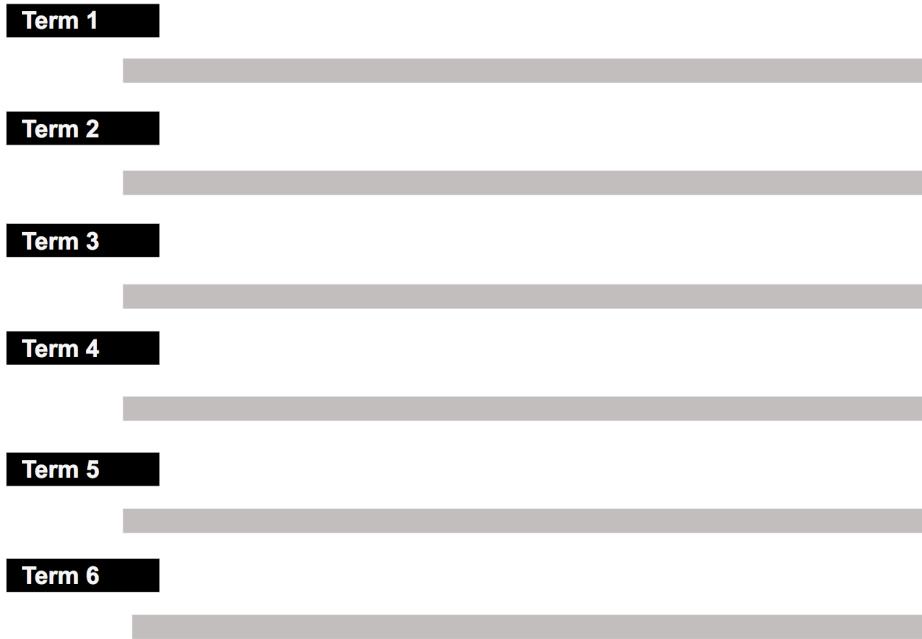
In this case, DEG must have been an acronym specific to the developer's API. In these cases, there's really no excuse for not defining your own pet acronyms and unique vocabulary. You can't simply make up an acronym and not define it for users (without frustrating them).

But many times, acronyms and unfamiliar terms are 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 or substitute in more friendly names. You have to teach the user the right language so they can participate in the conversation. One commonsense approach for teaching users how to speak this language involves defining unfamiliar words for the user by way of a glossary.

Should you provide a glossary?

Usually, providing a glossary in your documentation seems like going above and beyond the call of duty. However, simply defining these terms has tremendous benefit for the writer too, not just the users. Defining terms helps you recognize the special 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).



Glossaries not only provide clarity about terms for users, they also help the writers think more clearly and consistently about the topic

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 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 judgement 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 anyone who develops with 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.

The glossary is really where so many of the challenges around developer documentation come into focus. 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 the first time it's used on each page in your docs. However, on subsequent references, it's probably easiest to link to the glossary definition rather than continually incorporate tooltips. Let users make their own way to the glossary when they need help with a term. If the glossary page is in a clearly visible space, you won't have to go to great lengths to link terms to their glossary definitions with each usage.

Sample glossary pages

The following are sample glossary pages in various APIs.

Lyft

<https://developer.lyft.com/docs/glossary>

The screenshot shows the Lyft developer documentation website. At the top, there's a navigation bar with the Lyft logo, links for Overview, Documentation, Manage Apps, and Community, and a Log In button. Below the navigation is a secondary header with a Search bar and a 'SUGGEST EDITS' button. The main content area has a left sidebar with sections like Deeplinking, Recipes (with items like 'Add Lyft Rides to Your Customer E...'), Development (with items like 'API Changelog'), and Legal (with items like 'Brand Guidelines'). The 'Glossary' section is highlighted with a dark blue background. The main content area title is 'Glossary' with a subtitle 'Definitions of common terms used with the Lyft API.' It includes a 'TABLE OF CONTENTS' sidebar on the right listing terms such as Driver Rating, Lyft, Lyft Line, etc. The central content area contains a detailed description of the 'Driver Rating' term.

Glossary
Definitions of common terms used with the Lyft API.

TABLE OF CONTENTS

- Driver Rating
- Lyft
- Lyft Line
- Lyft Lux
- Lyft Lux SUV
- Lyft Plus
- Lyft Premier
- Passenger Rating
- Prime Time
- Trust & Service Fee

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 actually define terms that might normally 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 special terms and vocabulary.

Yext

Glossary

- **Bios** A Yext-exclusive data type that highlights a location's professionals and their expertise. Bios can appear on listings, in widgets, and on sites built with Yext Pages.
- **Calendar** A Yext-exclusive data type that describes events taking place at a location. Calendars can appear on listings, in widgets, and on sites built with Yext Pages.
- **Category** The location's business type (e.g., "Contractors > Carpet & Flooring > Carpet Retailers"). Many of the available category names in our system mention specific kinds of products or services (e.g., "Computers & Software > Software > Software Support," "Pets > Pet Food & Supplies"), making it possible for a location to have several categories.

Yext glossary

Note how in the Yext glossary, when industry standard terms such as “SSO” are defined, the definitions cover the company’s specific use of SSO; the general term SSO isn’t simply defined. Yext explains, “Single Sign On, which is a feature available to enterprise clients and partners....” And then goes on to explain more details around partners’ versus employees of enterprise clients and how they can each use SSO.

When viewing the glossary, the terms also appear in the sidebar, replacing any doc pages listed there. I’m not a fan of this approach because it causes users to lose the context of their documentation sidebar. If the glossary is common to multiple sets of documentation, one would hope the glossary definitions are single-sourced through the authoring tools.

One challenge this Yext example brings up is how to distribute glossaries across multiple doc sets. Suppose only some of these terms are relevant to one doc set, and other terms are relevant to another doc set. Do you partition the glossary into multiple specialized glossaries by product, or just have one general glossary across all products?

Use the strategy that makes sense for your docs. For the most part, since users just jump down to one specific glossary entry, it probably doesn’t matter if other terms (that aren’t specific to the doc set in context) are also included. On the other hand, filtering the definitions to a relevant subset would also encourage readers to actually browse through the definitions.

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 URLs, all of which share a common base path. To make it easier to manage your APIs, Apigee augments these raw URLs 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).

Activity: Identify non-reference content

Unlike the activities in earlier sections, it's harder to practice a skill related to non-reference content. For this activity, instead of exploring a particular tool or technique, you'll analyze content from some doc sites. Why? Sometimes the best way to learn how to create API documentation is to carefully observe how it's done on sites you admire. Following common practices in the industry helps you create more predictable, easy-to-follow patterns in your own documentation.

Activity 6a: Assess the non-reference content in 3 API doc sites

In this activity, you'll evaluate several API docs to see which non-reference topics might be missing.

1. Identify 3 API doc sets to analyze. The API doc sets could be documentation projects could be any of the following:
 - API documentation projects at your current work
 - [An open-source API documentation project \(page 137\)](#)
 - API documentation from the [list of 100 API doc sites \(page 341\)](#)
 - API documentation sites from your favorite tools or companies
2. Look through the documentation and identify whether the API documentation site contains each of the following.

Non-reference topic	API doc site 1	API doc site 1	API doc site 1
API overview (page 265)			
Getting started (page 269)			
Authentication and authorization (page 277)			
Status and error codes (page 286)			
Rate limiting and thresholds (page 293)			
Code samples and tutorials (page 298)			
SDKs and sample apps (page 308)			

Non-reference topic	API doc site 1	API doc site 1	API doc site 1
Quick reference guides (page 316)			
API best practices (page 323)			
Glossary (page 326)			

Recreate this table in your favorite word processing application and populate the columns with Xs if they contain the topic.

3. Do you have any noteworthy observations from your analysis?

Publishing your API documentation

Overview for publishing API docs.....	335
List of 100 API doc sites.....	341
Design patterns with API doc sites.....	345
Docs-as-code tools	356
More about Markdown	360
Version control systems (such as Git)	366
Static site generators.....	369
Hosting and deployment options	380
Headless CMS options.....	385
Which tool to choose for API docs — my recommendations	390
Activity: Manage content in a GitHub wiki.....	394
Activity: Use the GitHub Desktop Client.....	402
Pull request workflows through GitHub.....	412
Jekyll and CloudCannon continuous deployment.....	417
Case study: Switching tools to docs-as-code	425

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. So even though tools typically falls outside of content development, you might find it to be more relevant with API doc contexts.

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:

```
<a target="_blank" class="noExtIcon" href="https://ieeexplore.ieee.org/document/7140676/">
```

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
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
<small>* E is the number of examples that mentioned a problem; D is the number of developers who reported a problem.</small>				

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 isn't 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 isn't 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 is why being a doc tools expert is particularly relevant in API documentation contexts.

Using tools your SME authors want to use to collaborate

Another aspect of doc tools is using tools that your SME authors want to use to collaborate. 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, primarily [those listed on Programmableweb.com](#), which maintains the largest directory of web APIs, I didn't find many DITA-based API doc sites. In fact, it turns out that almost none of the API doc sites listed on Programmable Web even use 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 (at least those listed on Programmableweb.com) 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, as with [Photobucket's API](#), but they're rare. And it's even more rare 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 on Programmableweb.

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 \(page 356\)](#), committing it to [version control \(page 366\)](#), 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).

Even if you could get them using 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

Also, if most developers use Macs but you use a PC (to accommodate your HAT), you might struggle to install developer tools or to follow internal tutorials to get set up and test out content.

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 and Doxygen for the past 20 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 a OpenAPI spec through [Swagger Codegen](#)), but it's not something that HATs can do.

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 endpoints \(page 78\)](#).)

If you have a lot of endpoints, you need a system for pushing the content into standard templates. Ideally, you should separate out 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 \(page 143\)](#) 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 \(page 354\)](#) 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 \(page 214\)](#).) 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 output from HATs look dated and old. They look like a relic of the pre-2000 Internet era. (See [Tripwire help and PDF files: past their prime?](#) from Robert Desprez.)

For example, here's a sample help output from Flare for the Photobucket API:

The screenshot shows a web-based API documentation page for Photobucket. The left sidebar contains a table of contents (TOC) with sections like 'Photobucket API Help', 'Using the Help', 'What's New', 'Getting Started', 'Examples', 'Methods', and 'Albums'. Under 'Albums', 'Create New Album' is highlighted. The main content area has a header 'Create New Album' and sub-sections for 'User Login Required', 'HTTP Method' (set to POST), 'REST Path' (/album/{identifier}), and 'Parameters'. A table defines parameters: 'identifier' is optional and type String; 'name' is optional and type String, with a note that it must be between 2 and 50 characters, containing letters, numbers, underscores, hyphens, and spaces. Below these are 'Request Example' and 'Response Example' sections.

With API documentation, often times 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 toward the developers using 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 \(page 417\)](#). 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 3 authors and a minimal amount of legacy content. I wasn't burdened by a ton of documentation debt or heavy processes, so I could innovate.

Jekyll is just one documentation publishing option in the API doc space. I enjoy working with its [code-based approach \(page 356\)](#), but there are [many different tools \(page 369\)](#) and [publishing options \(page 380\)](#) to explore.

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:

This content doesn't embed well in print, as 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, we're going to begin with some concrete examples and move towards the formulation of general principles. The following are about 100 openly accessible REST APIs that you can browse as a way to look at patterns and examples. Many of these REST API links are available from programmableweb.com.

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 wide variety of publishing tools and approaches, as well as terminology, in API docs. 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. [Zomato API docs](#)
27. [Yahoo Social API docs](#)
28. [Google Analytics Management API docs](#)
29. [Yelp API docs](#)
30. [Lyft API docs](#)
31. [Facebook API docs](#)
32. [Eventful API docs](#)
33. [Concur API docs](#)
34. [Paypal API docs](#)
35. [Bitly API docs](#)

36. Callfire API docs
37. Reddit API docs
38. Netvibes API docs
39. Rhapsody API docs
40. Donors Choose docs
41. Sendgrid API docs
42. Photobucket API docs
43. Mailchimp docs
44. Basecamp API docs
45. Smugmug API docs
46. NYTimes API docs
47. USPS API docs
48. NWS API docs
49. Evernote API docs
50. Stripe API docs
51. Parse API docs
52. Opensecrets API docs
53. CNN API docs
54. CTA Train Tracker API
55. Amazon API docs
56. Revit API docs
57. Citygrid API docs
58. Mapbox API docs
59. Groupon API docs
60. AddThis Data API docs
61. Yahoo Weather API docs
62. Glassdoor Jobs API docs
63. Crunchbase API docs
64. Zendesk API docs
65. Validic API docs
66. Ninja Blocks API docs
67. Pushover API docs
68. Pusher Client API docs
69. Pingdom API docs
70. Daily Mile API docs
71. Jive docs
72. IBM Watson docs
73. HipChat API docs
74. Stores API docs
75. Alchemy API docs
76. Indivo API 2.0 docs
77. Socrata API docs
78. Github API docs
79. Mailgun API docs
80. RiotGames API docs
81. Basecamp API docs
82. ESPN API docs
83. Snap API docs
84. SwiftType API docs
85. Snipcart API docs
86. VHX API docs
87. Polldaddy API docs

88. [Gumroad API docs](#)
89. [Formstack API docs](#)
90. [Livefyre API docs](#)
91. [Salesforce Chatter REST API docs](#)
92. [The Movie Database API docs](#)
93. [Free Music Archive API docs](#)
94. [Context.io docs](#)
95. [CouchDB docs](#)
96. [Smart Home API \(Amazon Alexa\) docs](#)
97. [Coinbase docs](#)
98. [Shopify API docs](#)
99. [Authorize.net docs](#)
100. [Trip Advisor docs](#)
101. [Pinterest docs](#)
102. [Uber docs](#)
103. [Spotify API](#)
104. [Trello API](#)
105. [Yext API](#)
106. [Threat Stack API docs](#)
107. [Strava API](#)
108. [Plaid API](#)
109. [Paymill API](#)

Tip: I last checked these links in July 2018. Given how fast the technology landscape changes, some links may be out of date. However, if you simply 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`.

Programmableweb.com: A directory of API doc sites on the open web

For a directory of API documentation sites on the open web, see the [Programmableweb.com docs](#). You can browse thousands of web API docs in a variety of categories.

Note that Programmableweb lists only open web APIs, meaning APIs that you can access on the web (which also means it's usually a REST API). They don't list the countless internal, firewalled APIs that many companies provide at a cost to paying customers. There are many more thousands of private APIs out there that most of us will never know about.

Activity 7a: Look for common patterns in API doc sites

In this activity, you'll identify common patterns in API documentation sites.

1. Go to the list of about 100 API documentation sites ([page 0](#)).
2. Select about 5 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. Be prepared to share your findings.

Design patterns with API doc sites

In the previous topic, we browsed through a long [survey of API doc sites \(page 341\)](#) and looked for similar patterns in their designs. “Design patterns” are common approaches or techniques in the way something is designed. 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 [endpoint documentation \(page 78\)](#).

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 exact 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 div tags and other style classes.

Different authoring tools have different ways of processing templates. With [Jekyll \(page 417\)](#), a static site generator, you can create values in a [YAML file \(page 156\)](#) and loop through them using Liquid to access the values.

Here's how you might go about it. In the frontmatter of a page, 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. For example, 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](#). In my [Jekyll tutorial \(page 0\)](#) later in the course, I include an activity where you add a new weatherdata endpoint to the Aviator theme, using the same frontmatter templating designed by the theme author.

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="endpoi  
t {{ doc.type }}></span>  
                        {% endif %}  
                    </a>  
                </h3>  
                {% if doc.description %}  
                    <p class="description">{{doc.descrip  
tion}}</p>  
                {% endif %}  
  
                {{ doc.content | replace: "<dl>", "<h6>Param  
eters</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 `api` collection and pushes the content into the HTML styles of the template. The result looks like this:

The screenshot shows the Aviator interface for the /books endpoint. On the left sidebar, there's a search bar and links for Documentation (Getting Started, Authentication, Errors), APIs (with /books listed under GET), and a Template by CloudCannon link. The main content area has a header for '/books GET' and a sub-header 'List all books'. It includes sections for 'Parameters' (offset, limit) and a note that the call will return a maximum of 100 books. Below this is a description of what the endpoint lists: photos with access. There are tabs for jQuery, Python, Node.js, and Curl, each with a snippet of code. To the right, there's a 'Response' tab showing a JSON array of book objects, and an 'Error' tab which is currently empty.

```

[{"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 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.

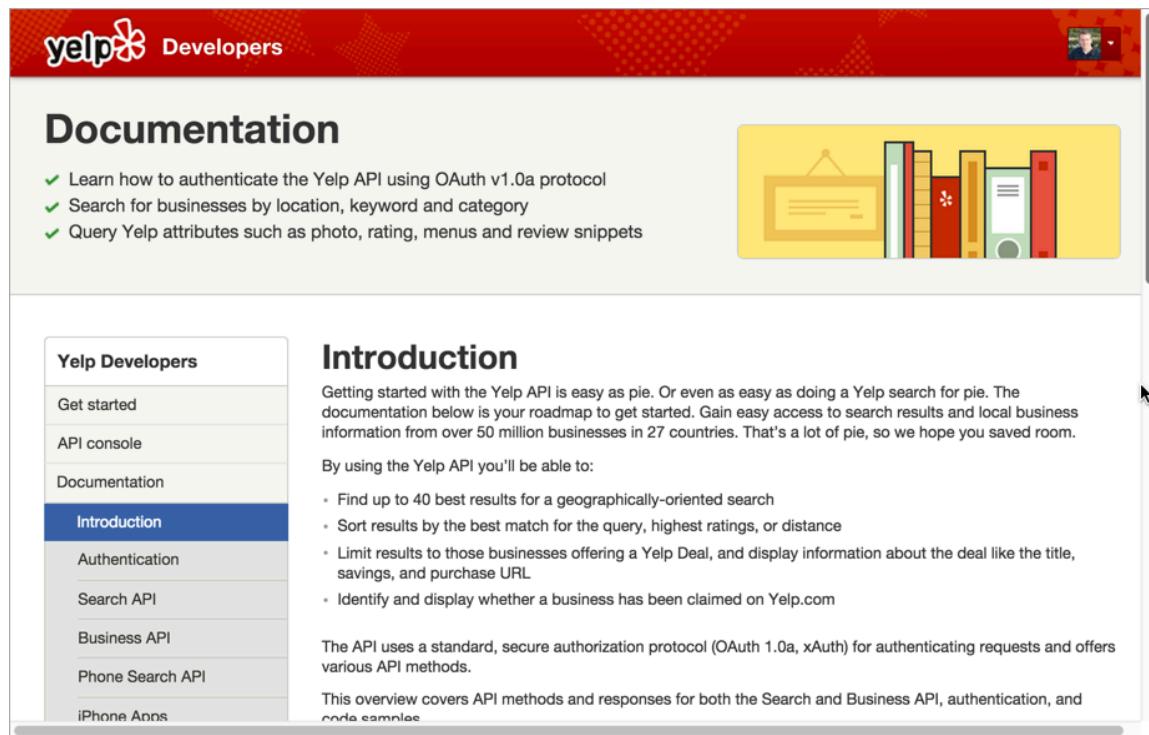
Also note that you don't have to create your own templates to display API documentation. You can probably already see problems related to custom templates. The templates are entirely arbitrary, with terms and structure based on the designer's perceived needs and styles. If you write documentation to fit a specific template, what happens when you want to switch themes? 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 \(page 142\)](#), and then launched into an extensive tutorial for the [OpenAPI specification \(page 160\)](#). I provided a tutorial for reading the OpenAPI specification using [Swagger UI \(page 214\)](#), along with an activity to [create your own Swagger UI \(page 219\)](#).

My point here is that you shouldn't be daunted 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:



The screenshot shows the Yelp Developers Documentation page. At the top, there's a red header bar with the Yelp logo and the word "Developers". Below the header, the main title "Documentation" is displayed. To the right of the title is a decorative graphic of books and a pie chart. On the left, there's a sidebar titled "Yelp Developers" containing links: "Get started", "API console", "Documentation" (which is highlighted in blue), "Introduction" (also highlighted in blue), "Authentication", "Search API", "Business API", "Phone Search API", and "iPhone Apps". The main content area starts with an "Introduction" section. It includes a brief overview of what the API can do, a list of capabilities (such as finding up to 40 results, sorting by match, rating, or distance, limiting results to deals, and identifying claimed businesses), and a note about the standard OAuth protocol. There's also a link to "code samples".

I hinted at this earlier, 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 \(page 143\)](#) or some other document generator 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 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 your [HAT \(page 0\)](#); 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 common website tooling.

Pattern 3: Abundant code samples

More than anything else, developers love [code examples \(page 298\)](#), 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:

The screenshot shows the Evernote Developers API documentation for "Sharing (and Un-Sharing) Notes". The main content area is titled "Single Note Sharing" and contains text explaining how to share a note via a public URL or email. It lists the required information: GUID, Shard ID, and authentication token. Below this is a code block for Python, Objective-C, Ruby, PHP, Java, and Node.js. The Python code is as follows:

```

1 def getUserId(authToken, userStore):
2     """
3         Get the User from userStore and return the user's shard ID
4     """
5
6     try:
7         user = userStore.getUser(authToken)
8     except (Errors.EDAMUserException, Errors.EDAMSystemException), e:
9         print "Exception while getting user's shardID:"
10        print type(e), e
11        return None

```

The right sidebar contains links to API Reference, iOS SDK Reference, and Android SDK Reference. It also includes sections for Quick-start Guides (Android, JavaScript, Python, Ruby, iOS), Core Concepts (Overview, Data Structure, Error Handling, The Sandbox), Authentication (Developer Tokens, OAuth, Permissions, Revocation and Expiration), Rate Limits (Overview, Best Practices), and Notes (Creating Notes, Sharing Notes, Reminders, Read-only Notes).

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 numerous 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 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. Although computers can read minified code, users usually can't or won't want to look at minified code. 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 a [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 \(page 298\)](#) in another topic.

Pattern 4: Lengthy pages

One of the most stark differences between regular end-user documentation and developer documentation is that developer doc pages tend to be much longer. In [How API Documentation Fails](#) (published in [IEEE Software](#)), researchers looked at common reasons why developers had severe problems with API documentation. In addition to incompleteness, ambiguity, and other reasons, Martin Robillard and Gias Uddin 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 Uddin 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:

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:

The screenshot shows the Backbone.js 1.0.0 documentation page. On the left, there's a sidebar with navigation links: [Backbone.js \(1.0.0\)](#), [» GitHub Repository](#), and [» Annotated Source](#). Below these are sections for [Introduction](#), [Upgrading](#), [Events](#) (with a list of methods like on, off, trigger, once, listenTo, stopListening, listenToOnce, and Catalog of Built-in Events), [Model](#) (with a list of methods like extend, constructor / initialize, get, set, escape, has, unset, clear, id, idAttribute, cid, and attributes), and [Catalog of Built-in Events](#). The main content area features the Backbone.js logo (a blue stylized 'B') and the title "BACKBONE.JS". A large paragraph explains Backbone.js's purpose: "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." Below this, another paragraph provides links to the project's GitHub repository, annotated source code, online test suite, example application, list of tutorials, long list of real-world projects, and MIT software license. At the bottom, there's a note about reporting bugs and discussing features on GitHub issues, Freenode IRC, Google Group, wiki, and Twitter. A footer section titled "Downloads & Dependencies" includes a note: "(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. In fact, 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 just 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. Sometimes it makes sense to keep all related information in one place, with "everything at your fingertips."
- **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, readme.io, 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 \(page 143\)](#) to provide the interactivity.

The screenshot shows a web-based API explorer interface. At the top, it says "Response Content Type: application/json". Below that is a table titled "Parameters" with the following columns: Parameter, Value, Description, Parameter Type, and Data Type. The rows are:

Parameter	Value	Description	Parameter Type	Data Type
apikey	<input type="text"/>	Your API key	query	string
html	<input type="text"/> (required)	HTML content (must be URL encoded)	query	string
outputMode	<input type="button" value="dropdown"/>	Desired response format (default XML)	query	string
url	<input type="text"/>	Input here will appear in the url field in the response	query	string
jsonp	<input type="text"/>	JSONP callback (requires outputMode to be set to json)	query	string

At the bottom left is a button labeled "Try it out!" with a double-headed arrow pointing to the right.

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 \(page 47\)](#) or [Postman \(page 39\)](#) (particularly the [Postman Run Button \(page 0\)](#)) 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 exactly 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, clearly 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 is a feature provided with sites like [Readme.io \(page 387\)](#).

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 have users logging in and dynamically populating the explorer with their API keys, you'll probably need a front-end designer and web developer to create this experience.

Some non-patterns in API doc sites

Finally, I'd like to briefly mention some non-patterns in API documentation. In the [Survey of API doc sites \(page 341\)](#), you rarely see any of the following:

- Video tutorials
- PDFs
- Commenting features
- Localization
- Single sourced outputs for different roles

By non-patterns, it's not to say 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 biggest 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 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 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

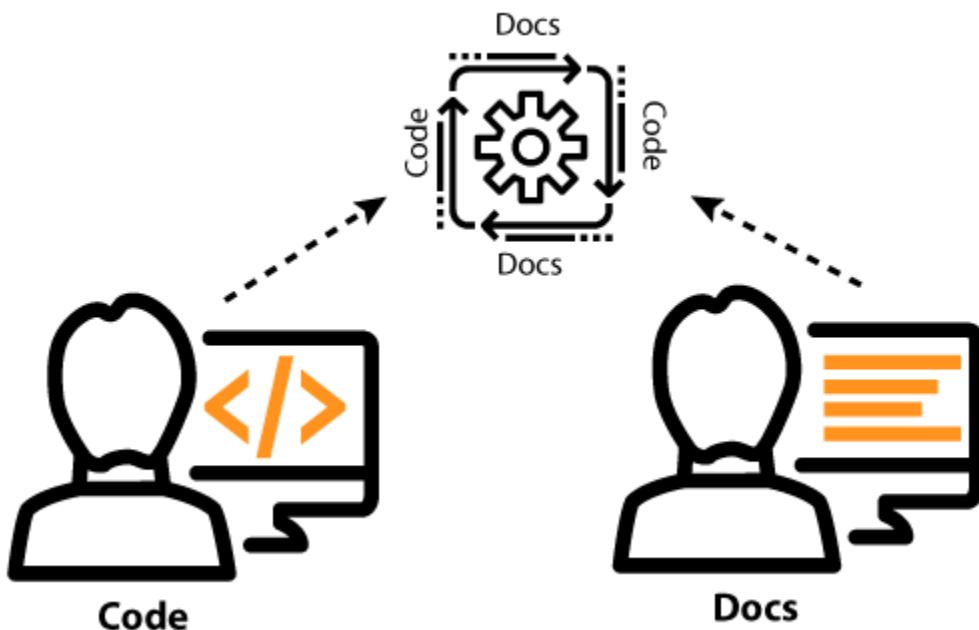
Having developers write or contribute to documentation should inform your tool choice with API documentation. If you plan to involve developers in writing and editing, you'll naturally choose more of a [docs-as-code tools \(page 356\)](#) approach. Docs-as-code means to embrace tools that treat docs just like developers treat software code. To treat docs like code generally means doing some of the following:

- **Working in plain text files** (rather than binary file formats like FrameMaker or Word).
- **Using an open-source static site generator** like [Sphinx \(page 374\)](#), [Jekyll \(page 370\)](#), or [Hugo \(page 372\)](#) to build the files locally through the command line (rather than using a commercial program such as FrameMaker or Microsoft 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 and GitHub 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).

In short, treating docs like code means to use the same systems, processes, and workflows with docs as you do with programming 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 the technical writers' 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 to just 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. It's hard to say this, but sometimes it might be more efficient than trying to transmit the information to a technical writer.

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 \(page 78\)](#), while technical writers focus more on the [non-reference documentation \(page 263\)](#). 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 changes and triggers a build and publishing job. This greatly simplifies the act of publishing. You can make edits across a number of docs 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.

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.

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 certain layout in different 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 can be especially important if you're integrating your docs into a website rather than generating a standalone output.

The docs-as-code tools are as flexible and 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 certain 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 \(page 425\)](#).

Dealing with more challenging factors

A lot of the docs-as-code solutions aren't built with robust technical documentation needs in mind. For example, you might have to deal with some of the following:

- 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 single 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. You could also use [Pandoc](#) to fill simpler PDF requirements.

My point is that you can handle these more challenging factors with non-traditional tools, but it requires more expertise.

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 is why I've dedicated an entire section to publishing in this course on API documentation.

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 just use asterisks (`*`). Instead of using `h2` tags, you use hashes (`##`). There's a Markdown tag for most of the common HTML elements, but not all of them.

Sample Markdown syntax

Here's a sample to get a sense of the Markdown syntax:

```
## 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.

If a system accepts Markdown, it converts the Markdown into HTML so the browser can read it.

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](#), 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 by developers) for a number of reasons:

- You can work in text-file format 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>
        <cmand>Go to <menucascade>
            <uicontrol>File</uicontrol><uicontrol>Print</uicontrol>
        </menucascade></cmand>
    </step>
    <step>
        <cmand>Click the <uicontrol>Print</uicontrol> button.</cmand>
    </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 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, the end result, if generated out to HTML, will probably look 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 great thing about 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, you can 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 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 commonly 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 7b: 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 \(page 0\)](#) 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 any 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 element, the content will be processed as either an inline `span` or a block `div` element. See [Syntax](#) in the kramdown documentation for more details.

What about reStructuredText and Asciidoc?

If you're using lightweight markup, you might 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 syntax for notes or warnings, and for 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 rST, 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 consistently process Markdown.

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](#).

In many ways, my blog requires tool support for kramdown-flavored Markdown and rouge syntax highlighting. However, 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. Despite the many Markdown flavors, GitHub-flavored Markdown is probably the most common. 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 and API Blueprint were dwarfed by the OpenAPI spec.

Further, the [OpenAPI spec \(page 143\)](#) actually 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 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.

Lightweight DITA

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 — OxygenXML already [provides support for Markdown](#)).

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)

Pretty much every IT shop uses some form of version control with their 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. Version control is such an important element to learn, we'll dive more deeply into it here. To provide variety from the [GitHub wikis tutorial \(page 394\)](#), in this section we'll use the GitHub Desktop client instead.

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 to you.

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 common anymore, since 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. However, Git is used most of the time, so we'll focus on it exclusively.

The screenshot shows the GitHub Bootcamp interface. At the top, there's a search bar labeled "Search GitHub" and navigation links for "Pull requests", "Issues", and "Gist". Below the header is a large banner titled "GitHub Bootcamp" with four numbered steps: 1. Set up Git (a cat with a box), 2. Create repositories (two cats in a cage), 3. Fork repositories (two cats shaking hands), and 4. Work together (two cats at laptops). Each step has a brief description. Below the banner is a user profile for "tomjohnson1492". Underneath are two comments from users "envygeeks" and "kenold" on an issue. To the right, there's a notification for "Easier feeds for GitHub Pages" and a sidebar titled "Repositories you contribute to" listing "jekyll/jekyll-help" and "slashdotdash/jekyll-lunr-js-search".

Github is a site that built tooling around Git."

Note that GitHub provides online repositories and tools for Git. However, Git and GitHub aren't the same. GitHub is simply a platform for managing Git projects.

[Bitbucket](#) is Altlassian's version of GitHub. Bitbucket lets you use either Git or Mercurial, but most of the Bitbucket projects use Git.

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 simply 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. This is the basic idea of 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. 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 special doc branch. When you're done, you'll create a pull request to have developers merge the doc branch back into the master.

With that brief introduction to docs-as-code and version control, let's jump into a host of authoring tools for developer documentation, starting with [static site generators \(page 369\)](#).

Static site generators

In the developer documentation space, you have many tool options for creating and publishing documentation, and there's no clear industry standard. Different tools may better suit different environments, skill sets, products, and requirements. On this page, I've listed the most common authoring tools related to the developer documentation space.

I've sorted these tools into several main groups:

- **Static site generators:** Used to author content and build the web output.
- **Hosting and deployment options (page 380):** Used to build, deploy, and host the web output.
- **CMS platforms (mostly headless CMSs) (page 385):** Provides an online GUI for authoring/publishing. In many cases, content is stored in plain text files and pulled in from GitHub.

As explained in [Docs-as-code tools \(page 356\)](#), I'm primarily focusing on static site generators and hosting/deployment options rather than traditional help authoring tools (HATs). See [Why focus on publishing API docs? \(page 335\)](#) for more background.

The tools below are particularly useful for writing and deploying the [non-reference content \(page 263\)](#) in your project. For tools specifically related to the [OpenAPI specification \(page 143\)](#), see [Ultimate Guide to 30+ API Documentation Solutions](#) from Nordic APIs and [Tools and Integrations](#) from Smartbear.

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, a configuration file, and your 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), and writes out the HTML pages from the Markdown sources. Each page usually has the sidebar and other navigation included directly into it, 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. This allows them to be leveraged in continuous delivery processes that are triggered from a particular trigger, 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 actually 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 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 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 in Jekyll, I used WordPress (and was even a WordPress consultant for 5 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're usually given a web server preview (such as `http://127.0.0.1:4000/`). 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, 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 in to the application code. It's also incredibly easy to work with custom code, such as special JavaScript libraries or 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. It's simply much easier and more flexible to do what you want.

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 directly on your page.

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 pipeline or platform builds and deploys it for you.

Although there are hundreds of static site generators (you can view a full list at [Staticgen.com](#), only a handful of are probably relevant for documentation. I'll consider these three in this article:

- [Jekyll \(page 370\)](#)
- [Hugo \(page 372\)](#)
- [Sphinx \(page 374\)](#)

One could discuss many more — Hexo, 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 \(page 417\)](#) 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.

The screenshot shows the Jekyll website homepage. At the top, there's a navigation bar with links for HOME, DOCS, NEWS, HELP, a search bar, and version information (V3.7.0, GITHUB). The main headline reads "Transform your plain text into static websites and blogs." Below this, there are three sections: "Simple" (describing content-only sites), "Static" (describing sites built from Markdown or Liquid), and "Blog-aware" (describing permalinks, categories, etc.). Each section has a link: "How Jekyll works →", "Jekyll template guide →", and "Migrate your blog →". A "Quick-start Instructions" box contains terminal commands: `~ $ gem install jekyll bundler`, `~ $ jekyll new my-awesome-site`, `~ $ cd my-awesome-site`, `~/my-awesome-site $ bundle exec jekyll serve`, and `# => Now browse to http://localhost:4000`. Below this, there's a "Get up and running in seconds." section featuring the GitHub logo and a "Free hosting with GitHub Pages" section with a GitHub Pages icon.

There are several compelling reasons to use Jekyll:

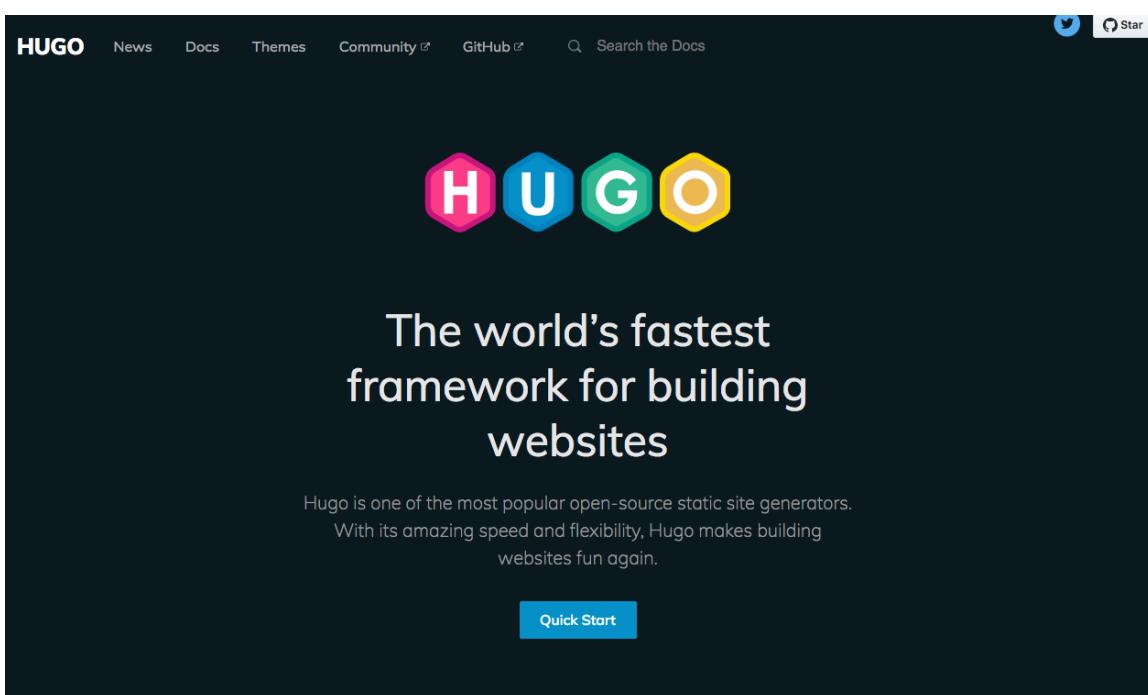
- **Large community.** The community, arguably the largest among static site generator communities, includes web developers, not just documentation-oriented groups. This broader focus attracts more developer attention and helps propel greater usage.
- **Control.** Jekyll provides a lot of powerful features (often through [Liquid](#), a scripting language) that allow you to do almost anything with the platform. This 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 use with 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 used 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 \(page 0\)](#) 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.

If you're looking for a documentation theme, see my [Documentation theme for Jekyll](#).

Hugo

[Hugo](#) is a static site generator that is rapidly growing in popularity. Based on the Go language, Hugo builds your site faster than most other static site generators, including Jekyll. There's an impressive number of [themes](#), including some designed for [documentation](#). Specifically, see the [Learn theme](#) and this [Multilingual API documentation theme](#).



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.

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 fast. This speed factor might be enough to overcome other issues.

Comparing speed with Hugo with Jekyll

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.

Speed may not be immediately apparent when you first start evaluating static site generators. 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.

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 difficult 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, much faster, it offers a serious advantage in the choice of static site generators. Given that major web development sites like [Smashing Magazine chose Hugo](#) for their static site generator, this is evidence of Hugo's emerging potential among the static site generators.

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:

I have been doing extended research on this topic and in the end chose to use Jekyll. I have done a huge project: <https://docs.mendix.com>, where we have made the complete website Open Source on Github.

Fun project where I ended up moving quite some stuff from Jekyll to Node. For example generating Sitemaps tended to be faster when doing it in Node instead of Jekyll.

But... Here's the downside. Our documentation is about 2700 pages (I'll have to lookup the real number). 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.

So if I am able to transfer the work that's done by plugins to Hugo/Node, I am going to refactor this to Hugo, because of the speed.

I might end up writing a similar blog about this project, it's long overdue.

In sum, generating a 2,700 page document site in Jekyll took 90 seconds; with Hugo, it took 0.5 seconds. This is a serious speed advantage that will allow you to scale your documentation site in robust ways. The author (whose docs are here: <https://docs.mendix.com>) did later make the switch from Jekyll to Hugo (see the [doc overview in GitHub](#)). This 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 \(page 425\)](#). 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 my doc projects at work (where we have probably 1,500 pages or so across many different doc sets), we implemented clever build shortcuts. By cascading configuration files, we can limit the builds to one particular doc directory. I

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 I'm working with, e.g., `acme`). When I'm working with that `acme` doc directory, I 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. As a result, Jekyll builds lightning fast. This method tends to work quite well and is used by others with large Jekyll projects as well. We have continuous delivery configured with the server, so 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.

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 added custom frontmatter in your Markdown pages that gets processed in unique ways by the layouts, 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 Golang. Or if working with another platform, you might need to change your Golang 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 more commercial platforms are used to process and build the outputs.

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.

The screenshot shows the Sphinx documentation website. At the top, there's a dark blue header with the Sphinx logo (an eye icon) and the word "SPHINX" in large letters, followed by "Python Documentation Generator". To the right of the logo are navigation links: Home, Get it, Docs, and Extend/Develop. Below the header, the main content area has a light blue background. On the left, a "Welcome" section introduces Sphinx as a tool for creating intelligent and beautiful documentation. It mentions its original creation for Python documentation and its features like output formats (HTML, LaTeX, etc.) and hierarchical structures. A "What users say:" box contains a quote: "Cheers for a great tool that actually makes programmers **want** to write documentation!". In the center, there's a "Documentation" section with links to "First steps with Sphinx" and "Search page". On the right, there's a sidebar titled "A project" featuring a horse icon, sections for "Download" (with links to Git repo and Python Package Index), "Questions?", "Suggestions?", and a mailing list sign-up form. Below the sidebar, there's a note about opening issues at the tracker.

Because Sphinx was designed from the ground up as a documentation tool, not just as 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? \(page 363\)](#) for more details around rST compared to Markdown.)

Sphinx can be used with the [Read the Docs \(page 0\)](#) platform. Overall, Sphinx has a passionate fan base among those who use it, especially among the Python community. However, because Sphinx is specifically designed as a documentation tool, the community might not be as large as some of the other static site generator communities.

As of July 2018, [Staticgen.com](#) shows the number of stars, forks, and issues as follows:

```
<a target="_blank" class="noExtIcon" href="https://www.staticgen.com/">
```

The screenshot shows a comparison table for three static site generators: Jekyll, Next, and Hugo. Each row contains a summary card with metrics and a detailed description.

Jekyll	Next	Hugo
<p>★ 34800 +611</p> <p>⌚ 139 +19</p> <p>⚡ 7654 +119</p> <p>🐦 6413 +90</p> <p>A simple, blog-aware, static site generator. Languages: Ruby Templates: Liquid License: MIT</p>	<p>★ 27145 +2121</p> <p>⌚ 291 +40</p> <p>⚡ 2697 +326</p> <p>🐦 N/A</p> <p>A framework for statically-exported React apps Languages: JavaScript Templates: JavaScript License: MIT</p>	<p>★ 27137 +1682</p> <p>⌚ 278 +22</p> <p>⚡ 3203 +109</p> <p>🐦 4207 +565</p> <p>A Fast and Flexible Static Site Generator. Languages: Go Templates: Go License: Apache 2.0</p>
<p>Get started with one click!</p> <p>For generators with the "Deploy to Netlify" button, you can deploy a new site from a template with one click. Get HTTPS, continuous delivery, and bring a custom domain, free of charge.</p> <p>Want your own Deploy to Netlify button? Learn more here.</p>		

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 forks that exist registered on their platform (GitHub, etc). The bug icon represents the number of open issues logged against the project. The green numbers indicate trends with these numbers.

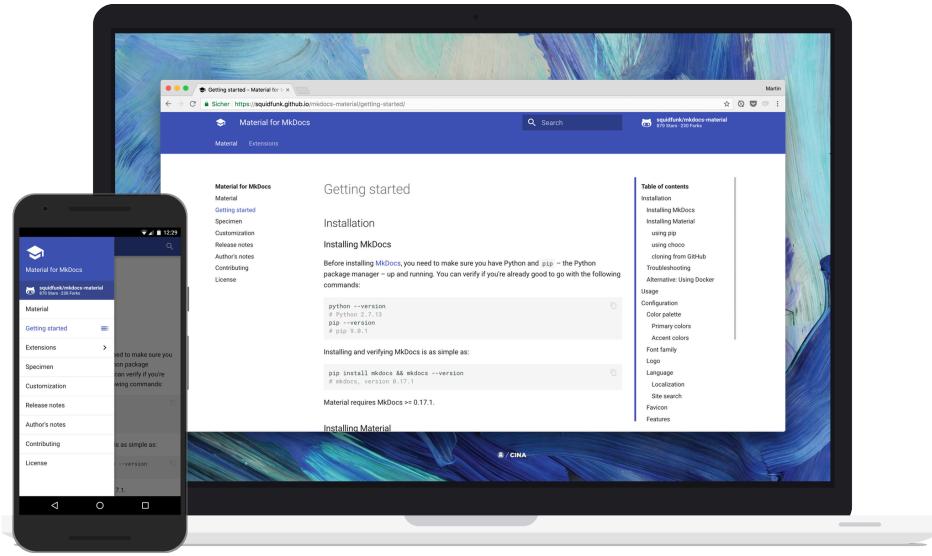
Jekyll, Next, and Hugo are the most common static site generators. If you look at [Staticgen.com](#), you’ll see that between Hugo and Sphinx, there are about 20 other static site generators (Hexo, Gatsby, GitBook, Nuxt, Pelican, Metalsmith, Brunch, Middleman, MkDocs, Harp, Expose, Assemble, Wintersmith, Cactus, React Static, Docpad, hubPress, Phenomic, Lektor, Hakyll, Nanoc, Octopress, and then Sphinx). But I called out Sphinx here because of its popularity among documentation groups and for its integration with [Read the Docs \(page 0\)](#).

Others

Besides Jekyll, Hugo, and Sphinx, there are some other popular static site generators worth noting here: MkDocs and Slate.

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.



Some [other themes](#) are also available. MkDocs uses [Jinja templating](#), provides [template variables](#) for custom theming, and more.

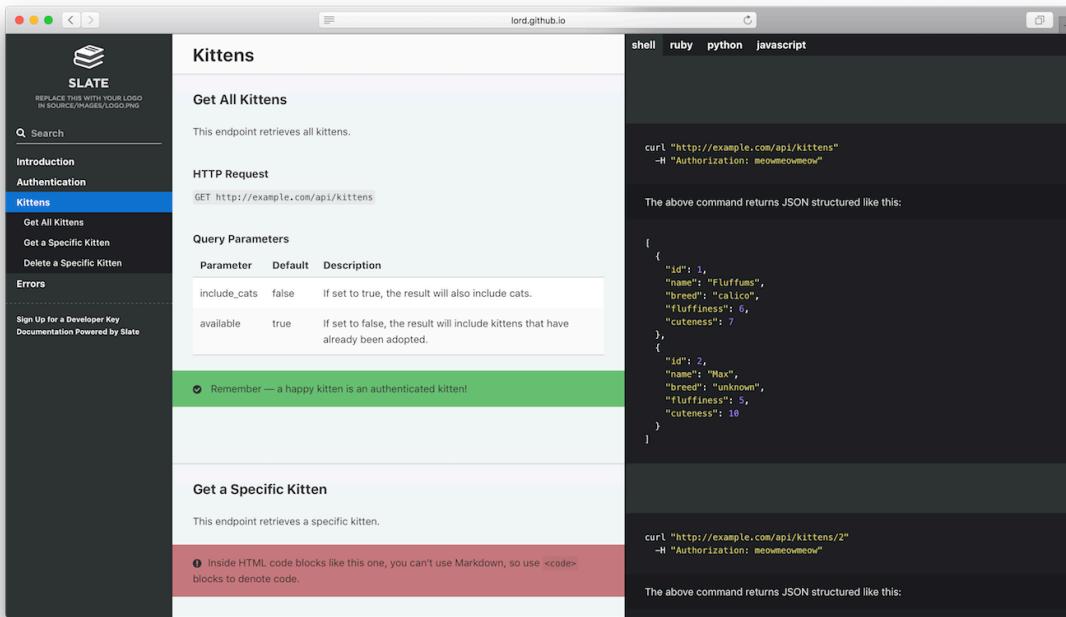
Although there are many static site generators with similar features, MkDocs is one more specifically oriented towards documentation. For example, it does include search.

However, 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.

Slate

[Slate](#) (based on [Middleman](#), a Ruby-based static site generator that is popular) is a common static site generator for API documentation that follows the three-column design made popular by [Stripe](#).



With Slate, you write in Markdown, build from the command line, and deploy your site similar to other static site generators. All your content appears on one page, with navigation that lets users easily move down to the sections they need.

Unlike with other static site generators mentioned here, Slate is more focused on API documentation than other types of content.

Miscellaneous

The list of other doc-oriented static site generator possibilities is quite extensive. Although probably not worth using due to the small community and limited platform, you might also explore [Asciidoctor](#), [Dexy](#), [Nanoc](#), [API Documentation Platform](#), [Apidoco](#), and more.

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.

What about ...[x]?

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??!!*

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. (This page is already 5,000+ words long.) If you feel strongly that I missed an essential tool for docs here, feel free to [contact me](#). Additionally, the tools landscape for developer docs is robust, complex, and seemingly endless.

Also, recognize that I'm only recommending what I perceive to be the most popular options. The developer tool landscape is diverse and constantly changing, and what may be relevant one day might be passé the next. This 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 \(page 390\)](#). The tool you choose can massively affect both your productivity and capability, so it tends to be an important choice.

Hosting and deployment options

Static site generators handle content development, but not hosting and deployment. For this, you have another category of tools to consider. 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. These platforms often read content stored on GitHub, sync it to their platform, and initiate build and publishing processes when they detect a change in a particular branch (such as gamma or prod).

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.

Quite a few doc sites use GitHub and Jekyll. For example, [Bootstrap](#) uses it:

```
<a target="_blank" class="noExtIcon" href="https://pages.github.com/">
```

The screenshot shows the GitHub Pages settings interface for a repository. At the top, a green success message box says "✓ Your site is published at <http://idratherbewriting.com/learnapidoc/>". Below this, under "Source", it says "Your GitHub Pages site is currently being built from the master branch." with a "Learn more" link. There are "master branch" and "Save" buttons. Under "Theme Chooser", there's a "Choose a theme" button. Under "Custom domain", there's a text input field and a "Save" button. At the bottom, there's a checkbox for "Enforce HTTPS" which is unchecked, with a note explaining it's unavailable because the domain is not properly configured to support HTTPS. It also links to troubleshooting custom domains and mentions HTTPS provides encryption.

```
</a>
```

GitHub Pages integration with GitHub repositories

In your GitHub repo, click **Settings** and scroll down to **GitHub Pages**. This 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.](#)

[master branch](#) [Save](#)

Theme Chooser
Select a theme to publish your site with a Jekyll theme. [Learn more.](#)

[Choose a theme](#)

Custom domain
Custom domains allow you to serve your site from a domain other than `idratherbewriting.com`. [Learn more.](#)

[Save](#)

Enforce HTTPS — Unavailable for your site because you have a custom domain configured (`idratherbewriting.com`)
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 user name 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 actually separate projects in separate repos. The fact that each repo is simply available in a subdirectory like this off of my main domain 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.

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:

Read the Docs hosts documentation, making it fully searchable and easy to find. You can import your docs using any major version control system, including Mercurial, Git, Subversion, and Bazaar. We support webhooks so your docs get built when you commit code. There's also support for versioning so you can build docs from tags and branches of your code in your repository.

Read the Docs provides both an open-source, free version (readthedocs.org) and a commercial version (readthedocs.com). This allows you to level-up your project when your needs mature but also doesn'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? \(page 363\)](#) for more details, or see [Why You Shouldn't Use “Markdown” for Documentation](#) for a more impassioned argument for rST.

The [Read the Docs documentation](#) shows a sample output.

The screenshot shows the Read the Docs website. On the left is a sidebar with a search bar and a navigation menu under 'USER DOCUMENTATION'. The main content area shows the 'Getting Started' page with the title 'Getting Started'. It includes a brief introduction, a note about existing Sphinx or Markdown users, and a section titled 'Write Your Docs' with two bullet points: 'In reStructuredText' and 'In Markdown'. Below this is a section titled 'In reStructuredText' with a note about a screencast and a paragraph about Sphinx. At the bottom of the page is a command line instruction: '\$ pip install sphinx sphinx-autobuild'.

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 50,000 projects and received 252 million page views and 56 million unique visitors). You can [view their 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 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 \(page 0\)](#), a headless CMS for your content (which I discuss in more detail later on).

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 publish a number of 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](#)

Headless CMS options

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. This category of tools is called “headless CMSs.”

List of headless CMS options

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).

The screenshot shows the homepage of the Headless CMS Options website. The header features a large orange-to-red gradient background with the word "headless" in white script and "CMS" in a black speech bubble. Below the title is a subheader: "A List of Content Management Systems for JAMstack Sites". A "SHARE" button is visible. The main content area contains eight cards, each representing a different headless CMS:

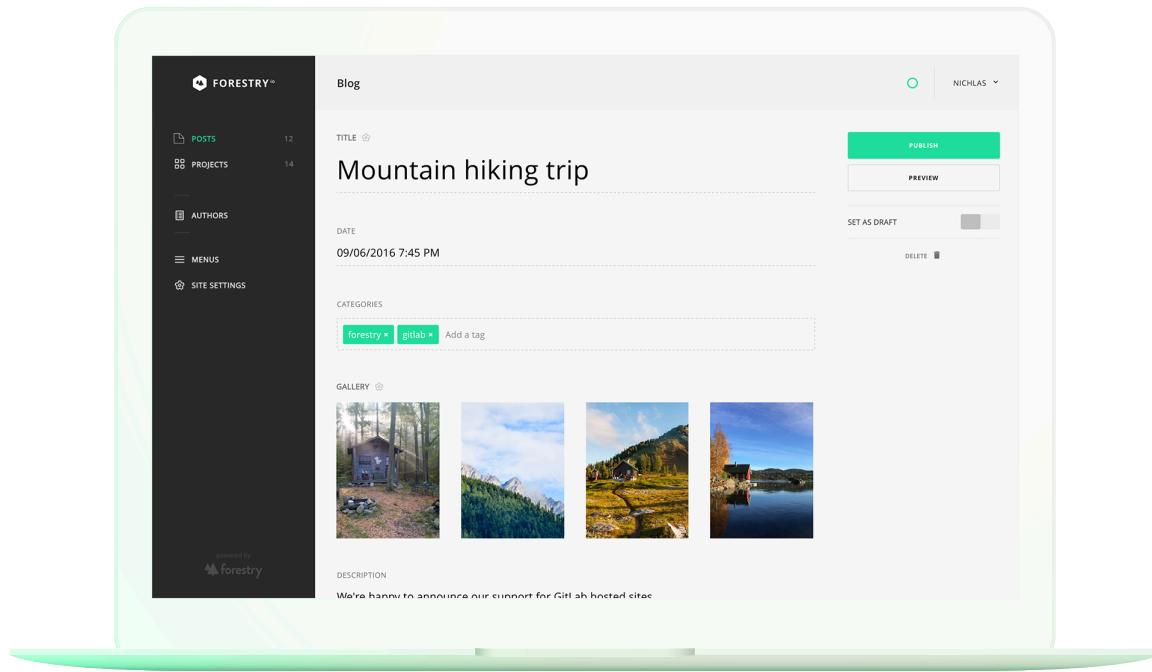
- Appernetic**: Type: Git-based. Supported Site Generators: Hugo.
- BowTie**: Type: Git-based. Supported Site Generators: Jekyll.
- Built.io**: Type: API Driven. Supported Site Generators: All.
- Butter CMS**: Type: API Driven. Supported Site Generators: All.
- CannerIO CMS**: Type: Git-based. Supported Site Generators: Custom.
- Cloud CMS**: Type: API Driven. Supported Site Generators: All.
- CloudCannon**: Type: Git-based. Supported Site Generators: Jekyll.

To the right of the cards, a sidebar reads: "Also visit [staticgen.com](#) for a ranked list of open-source site generators!"

Headless CMSs often combine both the authoring and the hosting/deployment in the same tool. Updates you make are built automatically on the platform. But unlike WordPress, the solution does not involve storing your doc content in a database and dynamically retrieving that content from the database when readers visit your page. Many times you can store your content on GitHub, and the headless CMS will read/pull it in a seamless way. (The platform probably will contain a database of some kind for your profile and other CMS features, but your content is not stored and retrieved there.)

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 for 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. The idea 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.

Unlike CloudCannon, Forestry 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. This reduces the chance that authors might use the wrong frontmatter tag in their pages (for example, `intro_blurb` or `IntroBlurb` or `introBlurb`) and instead just provides a box for this.

The screenshot shows the Netlify CMS interface for creating a new post. On the left, the 'Writing in Post collection' screen displays fields for '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' (an image of a Chemex coffee maker on a kitchen counter), and 'BODY' (Rich text editor with a preview of the text 'This week we'll take a look at all the steps required to make astonishing')). On the right, the published post is shown with the title 'A beginners' guide to brewing with Chemex', the date 'Wed, Jan 4, 2017', and a 'Read in x minutes' button. The post content includes the intro blurb and the image, followed by a summary: '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 \(page 0\)](#), 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.io

[Readme.io](#) is an online CMS for docs that offers one of the most robust, full-featured interfaces for developer docs available. Readme.io isn't a headless CMS, meaning you don't just point to your GitHub repo to pull in the content. Instead, I believe Readme.io stores content in a database (though this detail isn't mentioned on their site).

Readme.io's emphasis is on providing an interface that helps you more easily write documentation based on best practices and designs. Readme.io provides a number of wizard-like screens to move you through documentation process, prompting you with forms to complete.

Add An API

API Name: Awesome New API

Base URL: https://..

Authentication:

- API Keys
- Basic Auth
- OAuth 2.0

Mimetypes:

Consumes: application/json

Produces: application/json

API Headers:

You don't have any headers. [Add one.](#)

Additional Options:

- Enable API Explorer
- Enable API Proxy
- Enable auto code samples

[Save API](#)

Most importantly, Readme.io 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 \(page 143\)](#). You can experiment by choosing one from the [OpenAPI examples](#), such as [this one](#). Readme.io'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 \(page 241\)](#).)

Overall, Readme.io 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:

The screenshot shows a 'Documentation' page with a 'Try It Out' section. It includes input fields for 'lat*' (37.3708905) and 'lng*' (-121.9675525), a 'GET' button, a URL field containing 'https://simple-weather.p.mashape.com/weatherdata', and a 'Try It!' button. Below this is a response preview titled '200 OK' showing JSON data:

```
{ "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.io. 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.io 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.io, 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 constantly growing with new features, and there are many high-profile companies with their docs on Readme. If you consider how much time it actually takes to build and deploy your own doc solution, going with a site like Readme.io 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.io:

- [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 \(page 390\)](#).

Which tool to choose for API docs — my recommendations

I described a smattering of tools in [static site generators \(page 369\)](#). Which solution should you choose? It's a complicated decision that will invariably involve tradeoffs. The decision depends on your skillset, 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 3rd-party hosting and deployment option?
- How many authors will be authoring directly as contributors in the system?
- Do you have a 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 minimize the custom coding of a theme than it is to have complete control and flexibility over the solution. 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 \(page 372\)](#), [Sphinx \(page 374\)](#), or [Jekyll \(page 370\)](#). 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 theme or other code development, choose a solution such as [Readme.io \(page 387\)](#) or [Netlify CMS \(page 386\)](#) (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 customizing your theme and working on other doc tooling needs. 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? Is flexibility and control so important 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. There are not easy workarounds for these features that you can simply hack in during a lazy afternoon.

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 \(page 363\)](#). 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 use also 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 \(page 0\)](#). If you've decided on Jekyll, then explore [GitHub Pages \(page 380\)](#), [CloudCannon \(page 382\)](#), [Netlify \(page 384\)](#), or [Aerobatic \(page 384\)](#). If you've decided on Hugo, then explore [Netlify \(page 384\)](#) or [Aerobatic \(page 384\)](#). 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 huge 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 unimaginable ways. 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 \(page 425\)](#) for my experience going down this route.

If you don't have 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 \(page 143\)](#) could also be an important factor in your consideration of tools. How will you display all the [endpoint reference documentation \(page 78\)](#)? Rather than [creating your own template \(page 345\)](#) 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 \(page 241\)](#) (yet), so perhaps the best alternative might be to either link to or embed [Swagger UI \(page 214\)](#) 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 some day 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 my tutorial [Convert an HTML site to Jekyll](#).

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 shoes. 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.”

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.

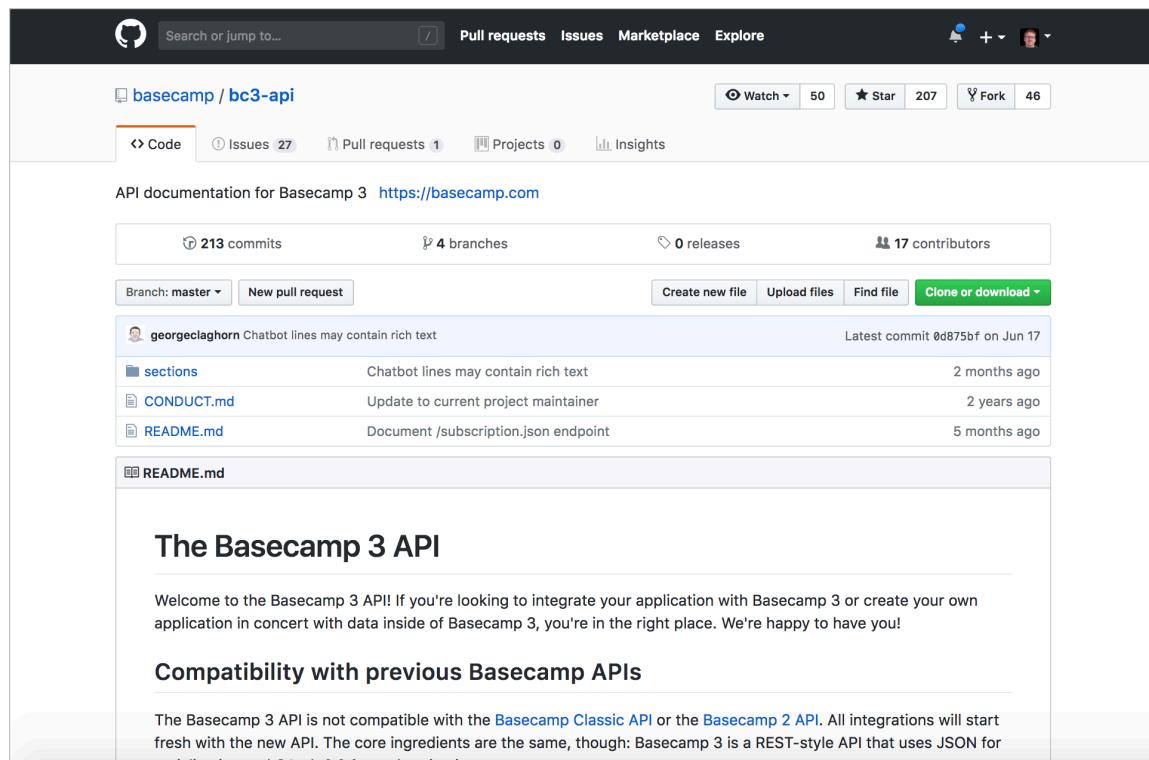
Activity: Manage content in a GitHub wiki

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 \(page 366\)](#).

Learning GitHub will allow you to become familiar with the version control workflows that are common with many [docs-as-code tools \(page 356\)](#). For this reason, I have a detailed tutorial for using GitHub in this course. Regardless of whether you actually 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.



The screenshot shows the GitHub repository page for `basecamp / bc3-api`. The repository has 213 commits, 4 branches, 0 releases, and 17 contributors. The `Code` tab is selected. The repository contains files like `sections`, `CONDUCT.md`, and `README.md`. The `README.md` file is expanded, showing its content:

```
## 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 communication and OAuth 2.0 for authentication.
```


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.

One of the neat things about using a GitHub repository is that you treat the [docs as code \(page 356\)](#), editing it in a text editor, committing it to a repository, and packaging it up into the same area as the rest of the source code. Because the content resides in a separate repository, technical writers can work in the documentation right alongside project code without getting merge conflicts.

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. This means you can handle all the reuse, 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 (that is, non-binary) 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.

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 basically 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 \(page 417\)](#).

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).

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

```
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](#). Note that when you configure your username, use your GitHub username, which will be something like `tomjohht` instead of `Tom Johnson`.
- [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.

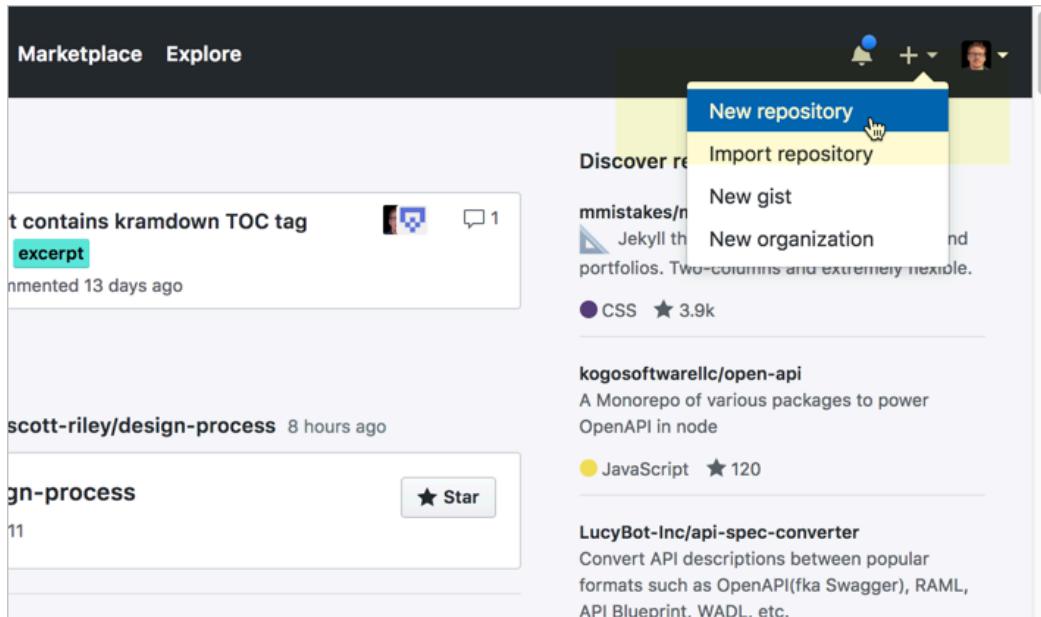
GitHub and Git are not the same. Git provides [distributed version control \(page 366\)](#). GitHub is a platform that helps you manage Git projects. GitHub also provides a GUI interface that allows you to execute a lot of Git commands, such as pull requests.

Activity 7c: 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**.

```
<a target="_blank" class="noExtIcon" href="https://github.com/new">
```



```
</a>
```

Creating a new GitHub repository

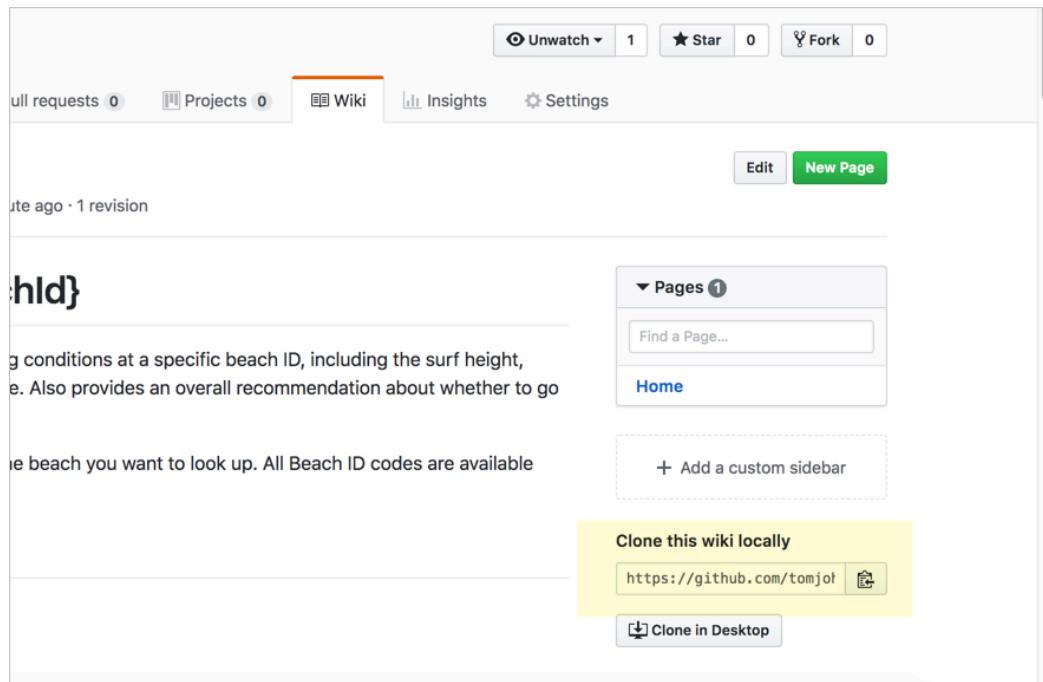
3. Give the repo a **Repository name**, a short **Description**, select **Public**, select **Initialize the repo with a README**, and then click **Create repository**. (Don't worry about selecting the license or gitignore settings for this activity.)
4. Click the **Wiki** tab on the top navigation bar of your new repository.
5. Click **Create the first page**. (Or if your wiki already has pages, click **New 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 7d: 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.

1. If you don't already have Git installed, set it up on your computer. (You can check by typing `git --version` in your terminal window. See [Install Git \(page 0\)](#) for more information on installation.)
2. While viewing your GitHub wiki in your browser, look for the section that says **Clone this wiki locally**. Click the clipboard button. (This copies the clone URL to your clipboard.)



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.

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
```

Cloning the wiki gives you a copy of the content on your local machine. Git is *distributed* version control software, so everyone has his or her 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 will 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.

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 `weatherapi.wiki`.

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` 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). If you can view invisible files on your machine (for instructions on making hidden files visible, see one of the following: [Windows](#) or [Mac](#)), you will also see a git folder.

Activity 7e: 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.
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`. Then use `cd {directory name}` to drill into the folder, or `cd ../` to move up a level.

4. Add all the files to your staging area:

```
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. As a sports analogy, the staging area is like your on-deck circle. These files are ready to be committed when you run `git commit`.

5. See the changes set in your staging area:

```
git status
```

Git responds with a message indicating which files are on-deck to be committed.

```bash Changes to be committed: (use "git reset HEAD ..." 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 by typing `git add .`, you might have 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.

1. 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 just type `git commit`, you'll be prompted with another window to describe the change. On Windows, this new window will 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. ("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.

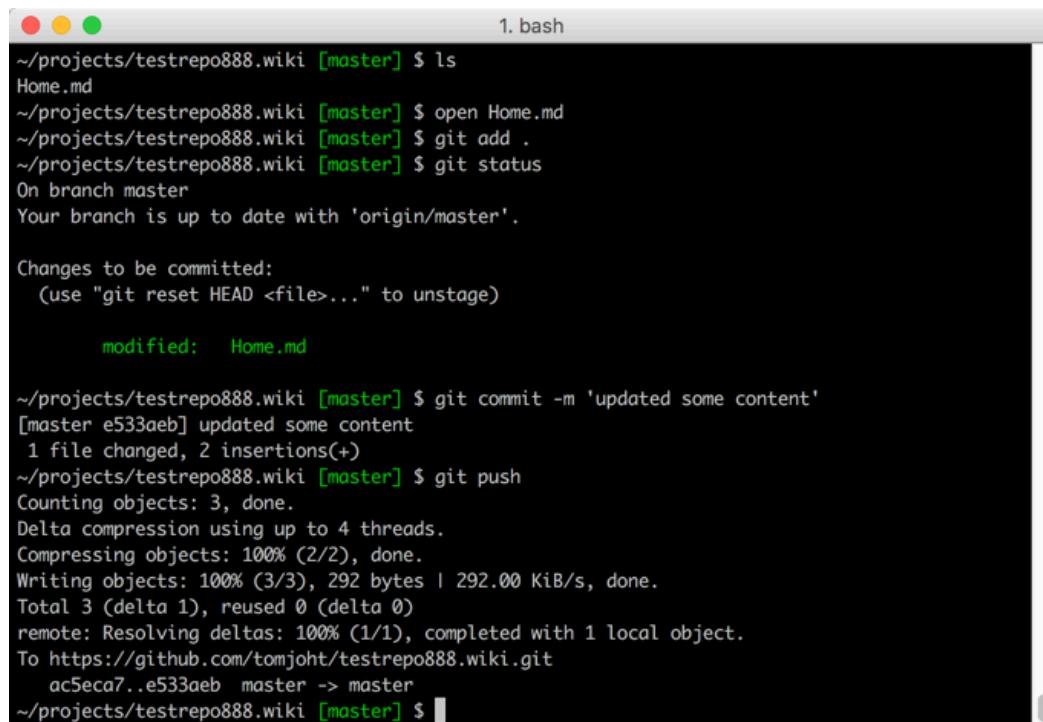
- Push the changes to your repository:

```
git push
```

If you didn't [set up automatic GitHub authentication \(page 396\)](#), you will be prompted for your GitHub user name and password.

Note that 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 `master`. Thus the command actually passed is `git push origin master` (or, push these changes to the remote repository, in the `master` 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:



The screenshot shows a terminal window titled "1. bash". The session starts with the command `ls`, which lists a single file named `Home.md`. The developer then runs `open Home.md` to open the file in a viewer. Next, they run `git add .` to stage all changes. They check the status with `git status` and see that they are on the `master` branch, and their branch is up to date with `'origin/master'`. The developer then adds a commit message "updated some content" and performs a `git commit -m 'updated some content'`. After committing, they run `git push` to push the changes to the remote repository. The output shows the commit hash `[master e533aeb]`, the number of files changed (1), and the total size of the objects pushed. Finally, they verify the push with `git log` and see the commit message "updated some content" along with the commit hash `ac5eca7..e533aeb`.

```
~/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/tomjoh/t/repo888.wiki.git
  ac5eca7..e533aeb  master -> master
~/projects/testrepo888.wiki [master] $
```

Terminal window with git commands

- Now verify that your changes took effect. Browse to your GitHub wiki repository and look to see the changes.

GitHub workflows for online and local edits

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](#).

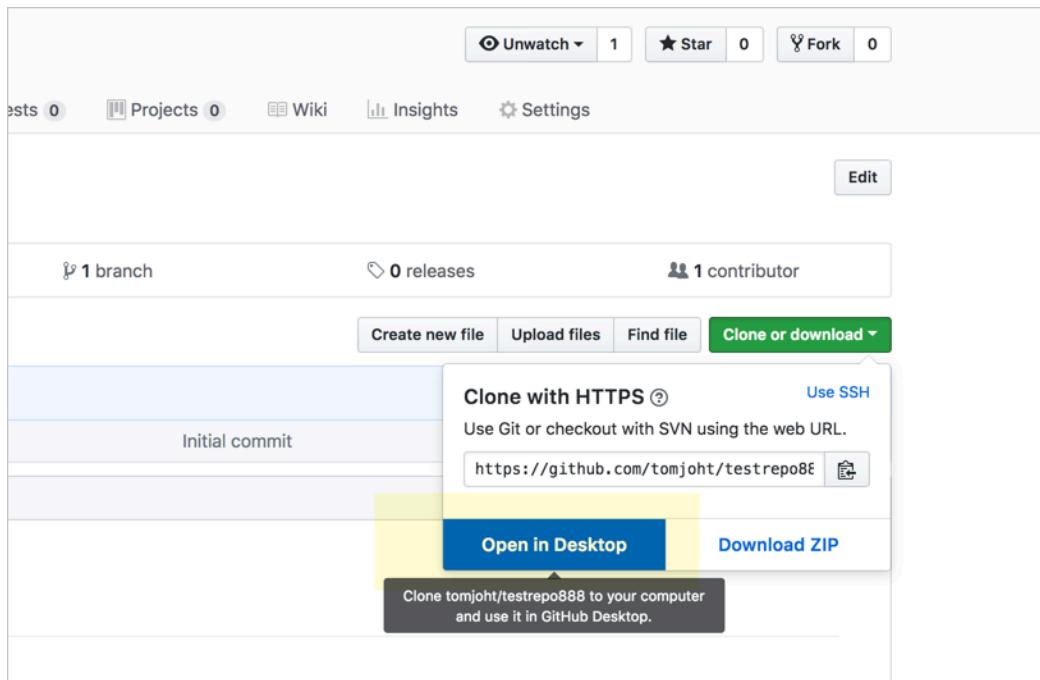
Activity: Use the GitHub Desktop client

Although most developers use the command line when working with version control systems, there are many GUI clients available that 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 easily highlight and indicate the changes taking place.

Follow a typical workflow with a GitHub project using GitHub Desktop

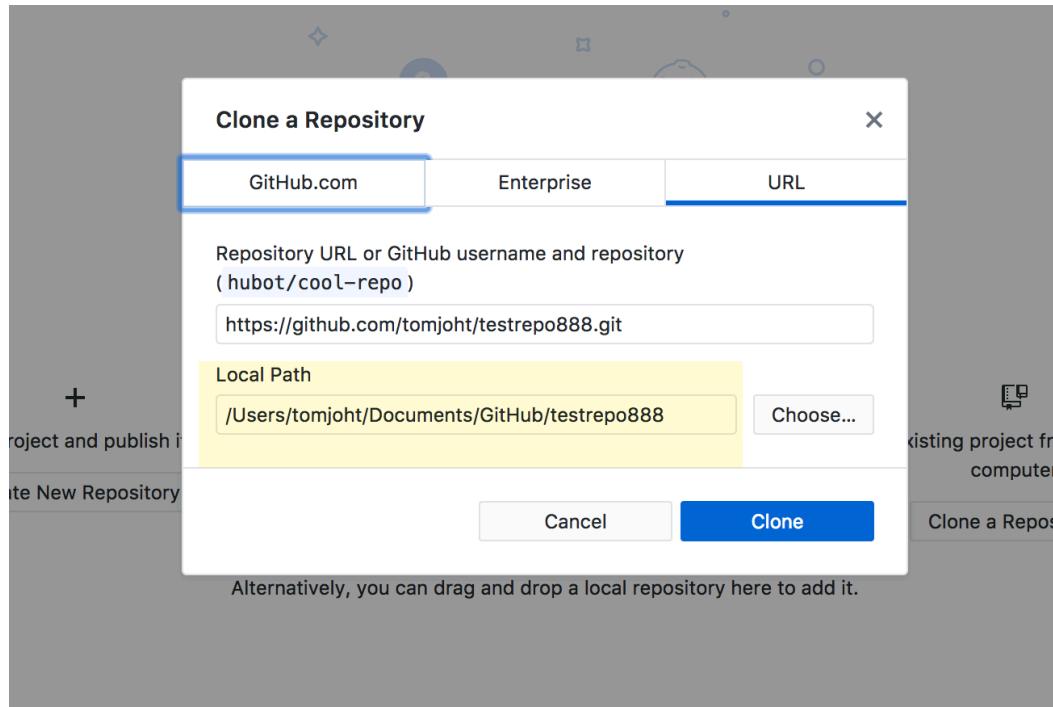
In this tutorial, you'll use GitHub Desktop to manage the Git workflow.

1. First, download and install [GitHub Desktop](#). Launch the app and sign in. (You should already have a GitHub account from [previous tutorials \(page 394\)](#), but if not, create one.)
2. Go to [Github.com](#) and browse to the wiki repository you created in the [GitHub tutorial \(page 394\)](#). (If you didn't do the previous activity, just create a new repository.)
3. While viewing your GitHub repo in the browser, click **Clone** and select **Open in Desktop**.



Open in GitHub Desktop

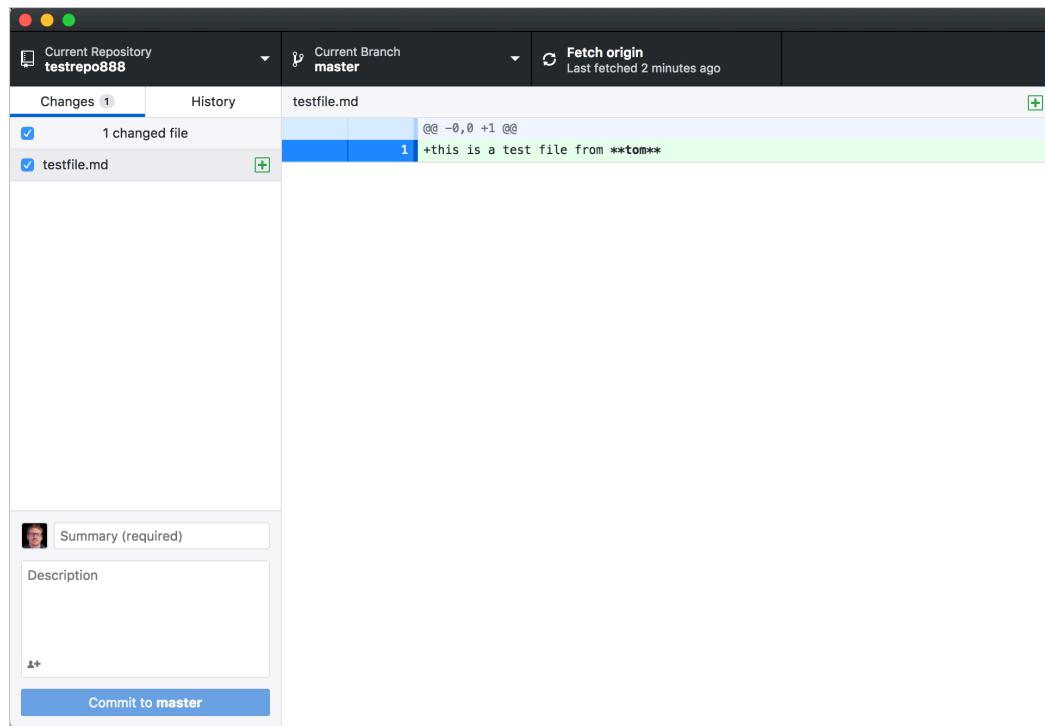
4. In the confirmation dialog, select **Open GitHub Desktop.app**. GitHub Desktop should launch with a “Clone a Repository” dialog box about where to clone the repository. If desired, you can change the Local Path.



Choosing the path for GitHub clone

(If you're using the same repo from the [previous GitHub tutorial \(page 394\)](#), you might want to delete the test repo you already installed in that directory.)

5. Click **Clone**.
6. Go into the repository where GitHub Desktop cloned the repo (use your Finder or browsing folders normally) and add a simple text file with some content. Or make a change to an existing file.
7. Go back to GitHub Desktop. You'll see the new file you added in the list of uncommitted changes on the left.



Uncommitted changes shown

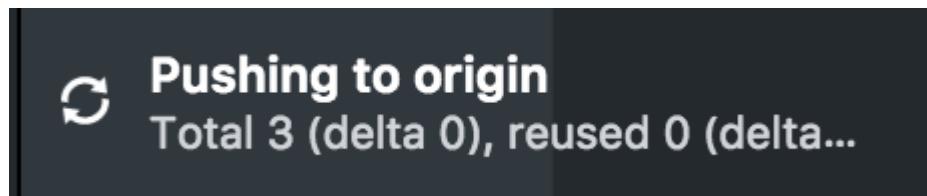
In the list of changed files, the green + means you've added a new file. A yellow circle means you've modified an existing file.

8. 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.

9. Click **Push origin** at the top.

You'll see GitHub Desktop show that it's "Pushing to origin."



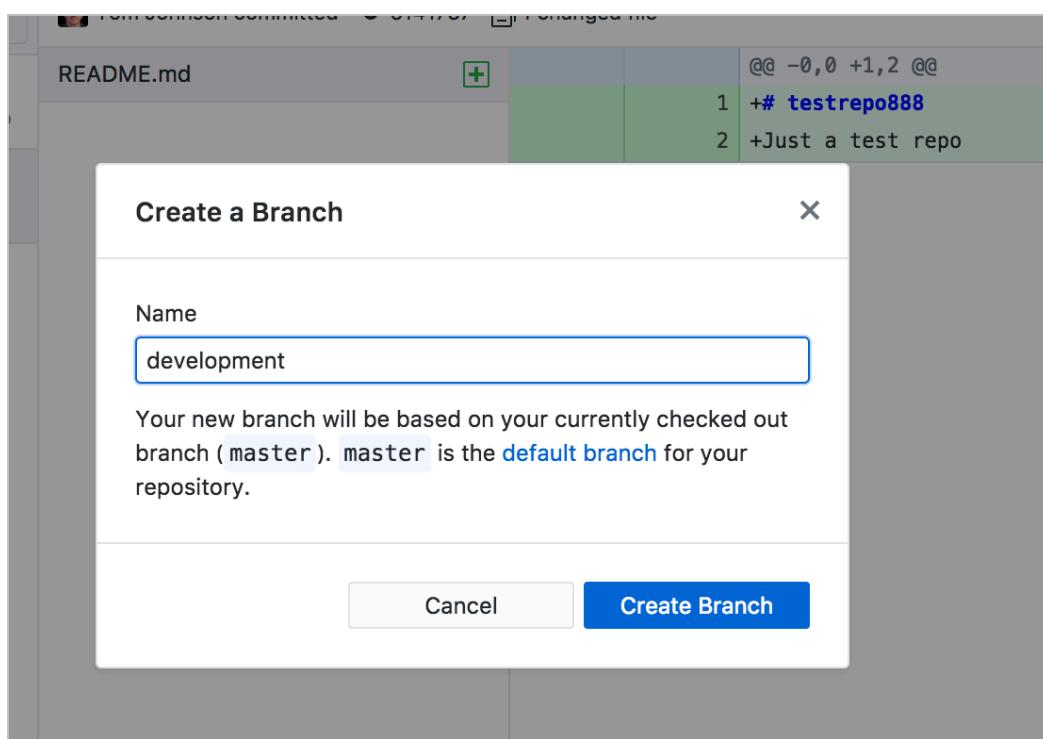
If you view your repository online (by going to **Repository > View on GitHub**), you'll see that the change you made has been pushed to the master branch on origin. You can also click the **History** tab in the GitHub Desktop client (instead of the **Changes** tab), or go to **View > Show History** 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

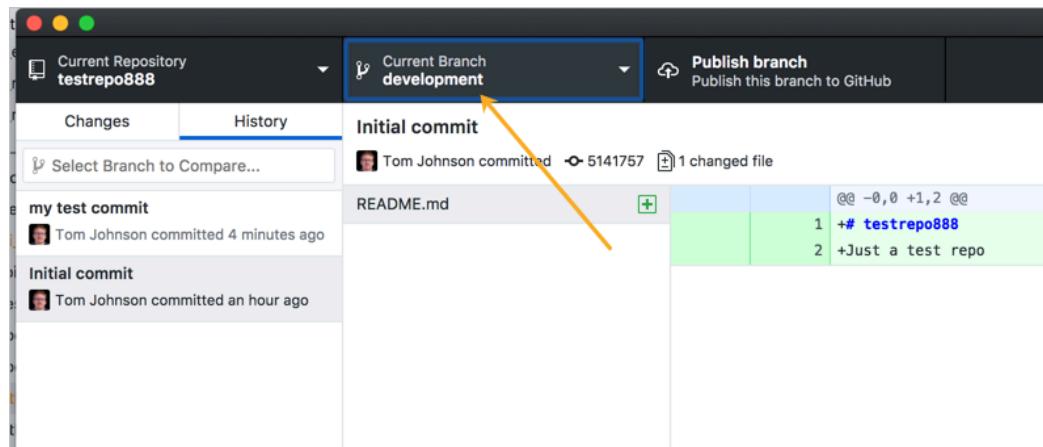
Now let's create a branch, make some changes, and see how the changes are specific to that branch.

1. Go to **Branch > New Branch** and create a new branch. Call it “development” branch, and click **Create Branch**.



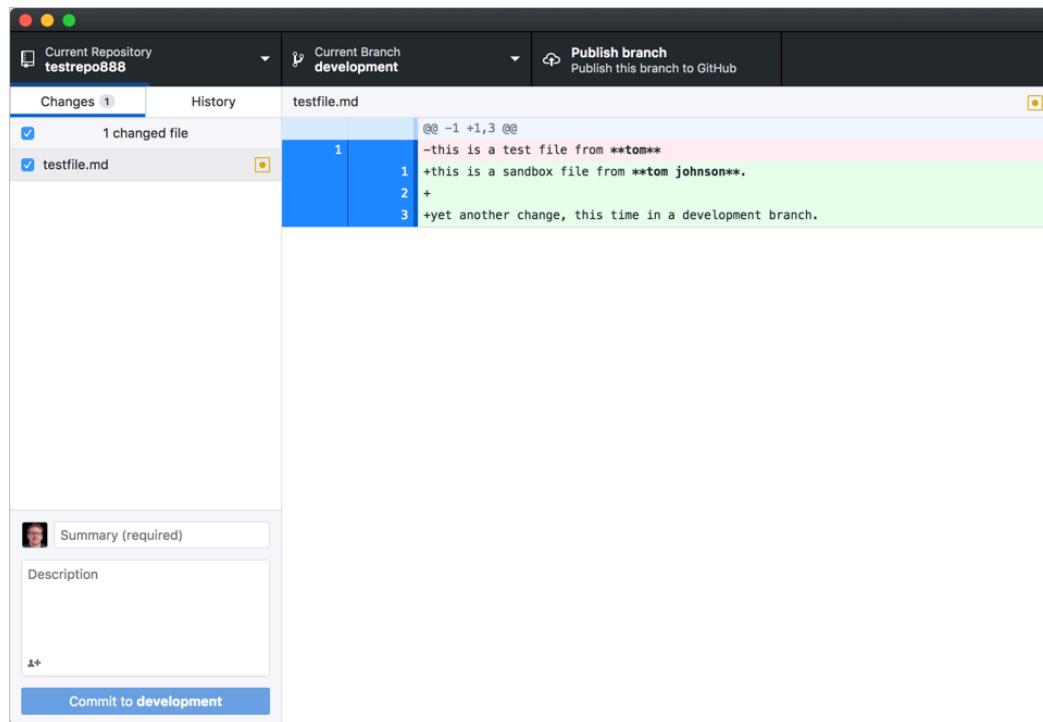
Creating a new branch

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.



New files modified

The file changes shows 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.

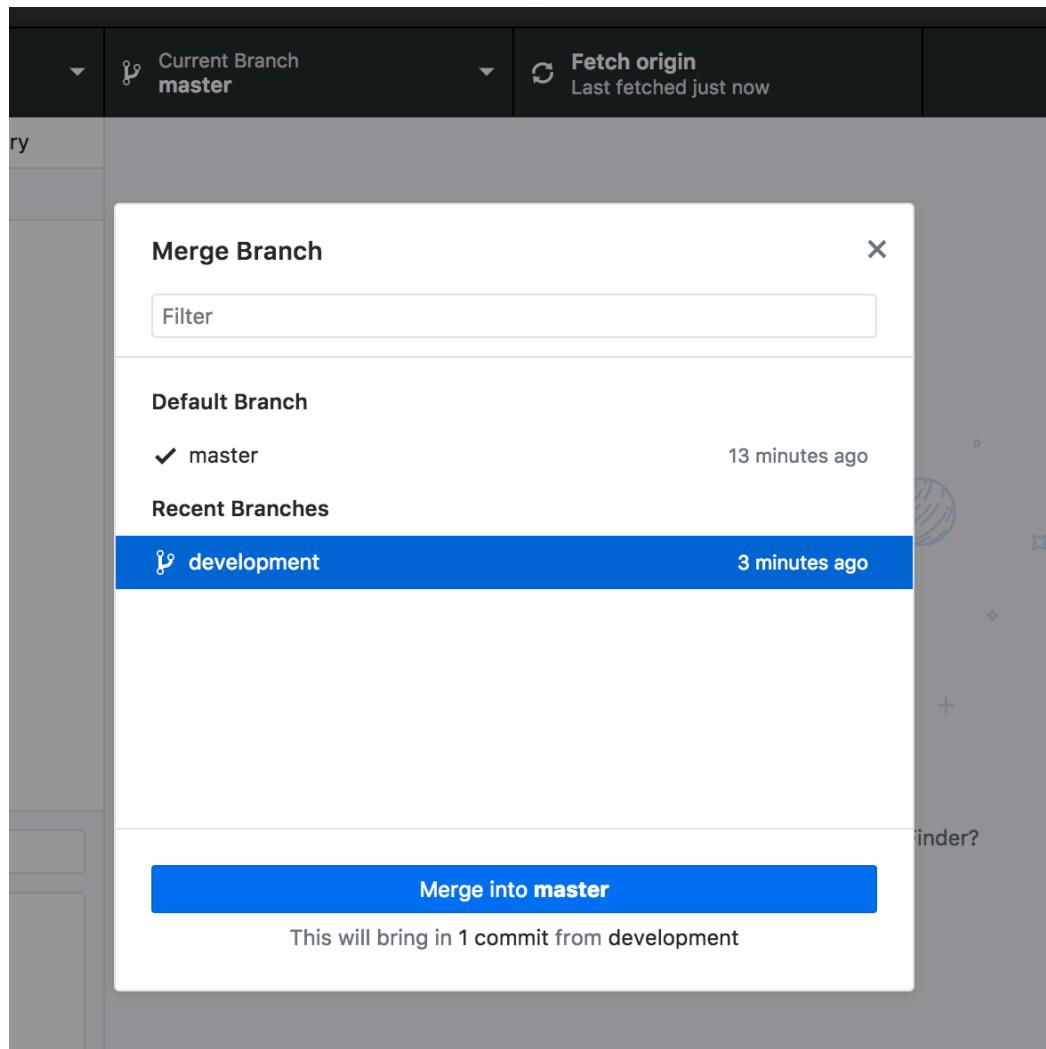
Usually, you create a new branch when you're making extensive changes to your content. For example, suppose you want to revamp a section ("Section X") in your docs. However, you might want to publish other updates before publishing the extensive changes in Section X. If you were working in the same branch, 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 master branch.
2. Go to **Branch > Merge into Current Branch**.
3. In the merge window, select the **development** branch, and then click **Merge into master**.



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

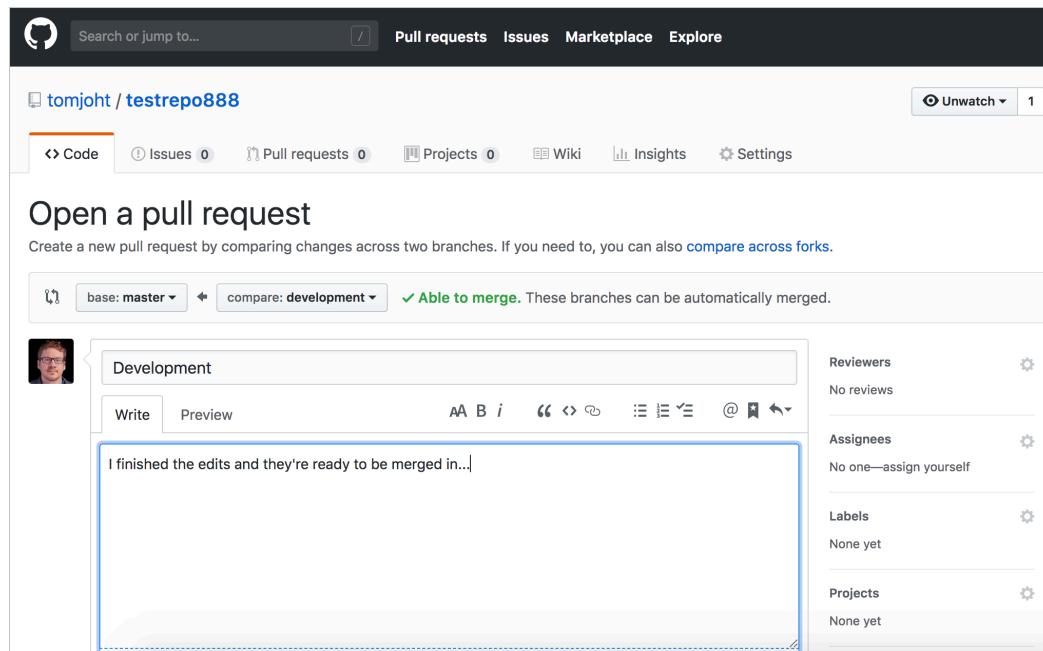
Now let's merge the development branch into the master using a pull request workflow. We'll pretend that we've cloned a repo managed by engineers, and we want to have the engineers merge in the development branch. (In other words, we might not have rights to merge branches into the master.) To do this, 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.

```
<a target="_blank" class="noExtIcon" href="">
```



```
</a>
```

Pull request

The left-facing arrow from the development branch towards the master indicates that the pull request (“PR”) wants to merge development into master.

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 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.

```
<a target="_blank" class="noExtIcon" href="">
```


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 master 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.

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.
>>>>> c163ead57f6793678dd41b5efeeef372d9bd81035
```

(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 leverage for a variety of scenarios. But most likely the scenarios you’ll actually use are somewhat limited 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 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 \(page 402\)](#), 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 important 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.

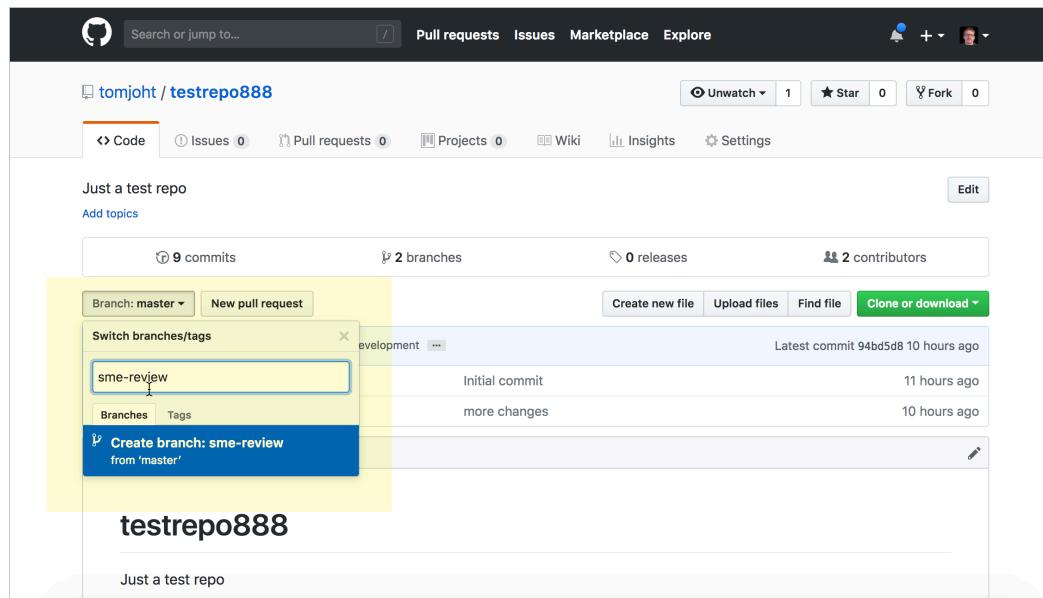
Make edits in a separate branch

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. This might be helpful if you have less technical people making edits to your content.

To make edits in a separate branch on GitHub:

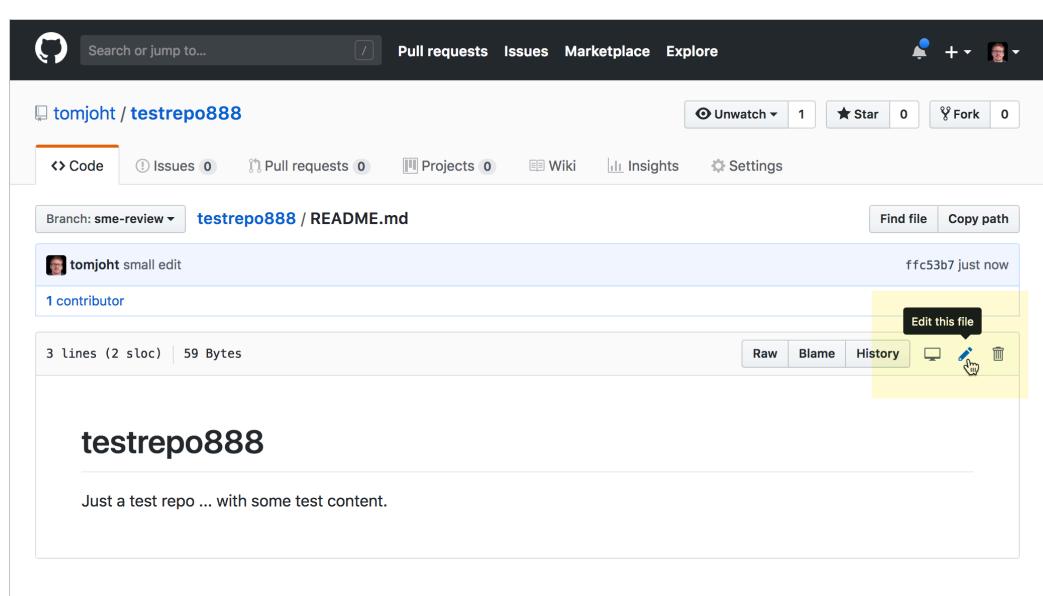
1. Pretend you're a SME-reviewer. Go to the same GitHub repo you created earlier (or create a new repo). Create a new branch by selecting the branch drop-down menu and typing a new branch name, such as "sme-review." Then press your **Enter** key.



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.



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 that the review process is complete, 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.

tomijoht / testrepo888

Code Issues Pull requests Projects Wiki Insights Settings

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](#)

Filters is:pr is:open Labels Milestones New pull request

0 Open 3 Closed Author Labels Projects Milestones Reviews Assignee Sort

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.

tomijoht / testrepo888

Code Issues Pull requests Projects Wiki Insights Settings

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](#).

base: master compare: sme-review Able to merge. These branches can be automatically merged.

Create pull request Discuss and review the changes in this comparison with others.

2 commits 1 file changed 0 commit comments 1

Commits on Aug 11, 2018

- tomijoht small edit
- tomijoht small edits

Showing 1 changed file with 1 addition and 1 deletion.

Compare to

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.

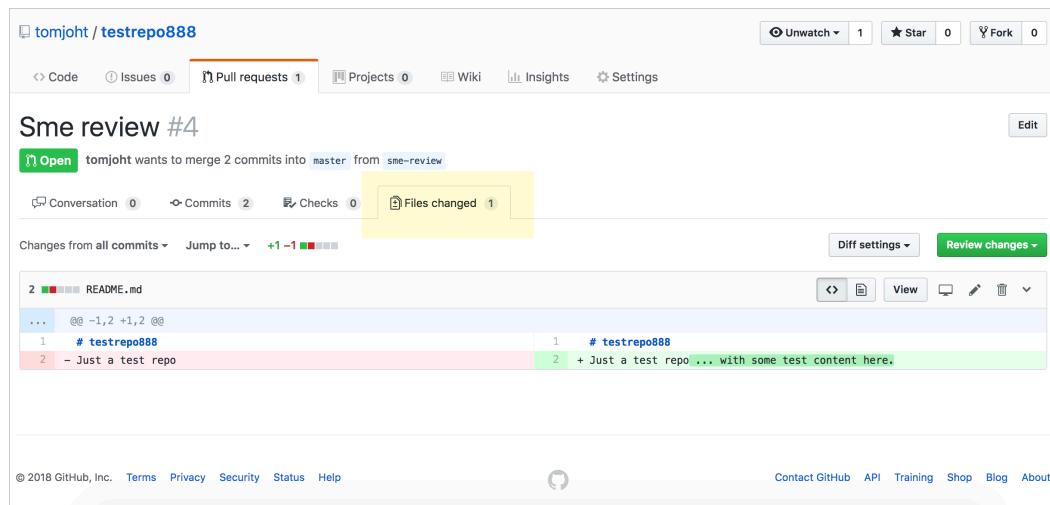
4. Click **Create pull request**.
5. Describe the pull request, and then click **Create pull request**.

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.



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** and select the **Request changes** option, 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 user names of those you want to have access in the Collaborator area.
4. Click **Add collaborator**.

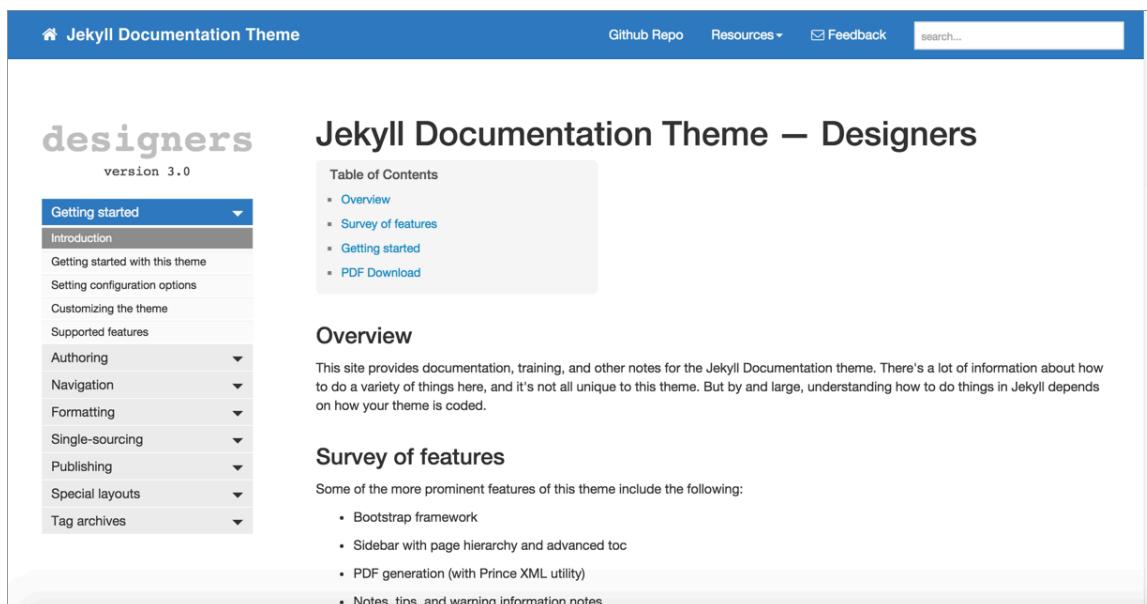
Jekyll and CloudCannon continuous deployment

Static site generators are a breed of website compilers that package up a group of files (usually written in Markdown) and make them into a fully deployable website. There are more than 350 different static site generators. You can browse them at [staticgen.com](#). One of the most popular static site generators (based on 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 \(page 370\)](#), 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:

A screenshot of a Jekyll documentation website titled "Jekyll Documentation Theme — Designers". The page features a sidebar on the left with a navigation menu for "designers" version 3.0. The menu includes sections like "Getting started" (which is currently selected), "Introduction", "Setting configuration options", "Customizing the theme", "Supported features", and several dropdown menus for "Authoring", "Navigation", "Formatting", "Single-sourcing", "Publishing", "Special layouts", and "Tag archives". The main content area on the right has a "Table of Contents" sidebar with links to "Overview", "Survey of features", "Getting started", and "PDF Download". Below this is the "Overview" section, which contains text about the theme's purpose and some of its features. The "Survey of features" section lists items such as "Bootstrap framework", "Sidebar with page hierarchy and advanced toc", "PDF generation (with Prince XML utility)", and "Notes, tips, and warning information notes".

The screenshot shows a clean, modern design with a blue header bar containing the site title and navigation links for "Github Repo", "Resources", "Feedback", and a search bar.

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:

The screenshot shows the Atom text editor interface. On the left is a tree view of a Jekyll project named 'Project'. The 'pubapis_jekyll.md' file is selected and open in the main editor area. The content of the file includes code comments and a preview of the rendered page. Below the editor is a terminal window showing the regeneration process of the site.

```

Project
├── docapis.yml
├── pubapis_switchin...
└── pubapis_jekyll.md
    ├── pubapis_github...
    ├── pubapis_overvie...
    ├── pubapis_github...
    └── pubapis_github...
    └── pubapis_which_t...
37  * Customize the look and feel of the output
38  * Insert JavaScript and other code directly on the page
39
40  ## Developing content in Jekyll
41
42  One of the questions people ask about authoring content with static site generators is how you see the
43  output and formatting given that you're working strictly in text. For example, how do you see images,
44  lists, or other formatting if you're authoring in text?
45
46  When you're authoring a Jekyll site, you open up a preview server that continuously builds your site
47  with each change you save. If you use Atom editor, you can use a terminal built directly into Atom (I
48  use the [PlatformIO IDE Terminal](https://atom.io/packages/platformio-ide-terminal) package).
49
50  Here's what my current view looks like:
51
52
53  
54
55  This setup works fairly well. Granted, I do have a Mac Thunderbolt 21-inch monitor, so it gives me
56  more real estate. On a small screen, you might have to switch back and forth between screens to see
57  the output.
58
59
60  Regenerating: 1 file(s) changed at 2018-08-11 23:11:25 ...done in 22.117034 seconds.
61  Regenerating: 1 file(s) changed at 2018-08-11 23:12:03 ...done in 18.740058 seconds.
62  Regenerating: 1 file(s) changed at 2018-08-11 23:12:22 ...done in 19.192513 seconds.
63  Regenerating: 1 file(s) changed at 2018-08-11 23:13:10 ...done in 19.546416 seconds.
64  Regenerating: 1 file(s) changed at 2018-08-11 23:13:50 ...done in 18.34229 seconds.
65  Regenerating: 1 file(s) changed at 2018-08-11 23:15:57 ...done in 17.880909 seconds.
66  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:

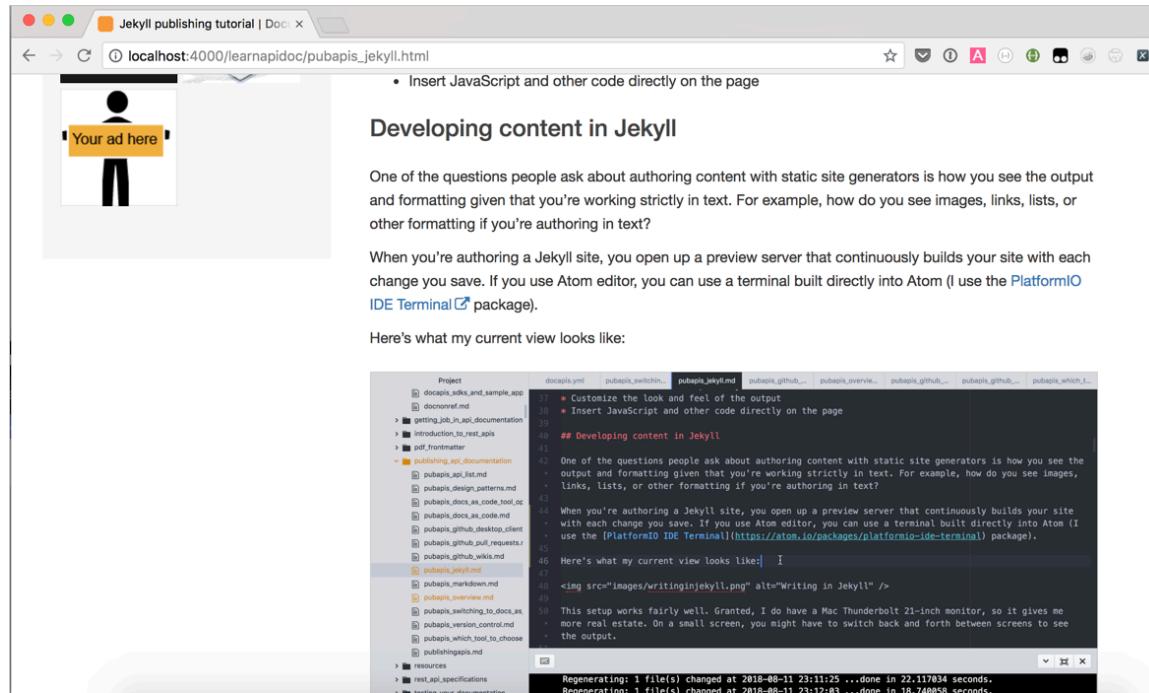
```
bundle exec jekyll serve
```

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 looks right:



Jekyll preview server

This setup works fairly 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 \(page 380\)](#) 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 calls itself as the “The Cloud CMS for Jekyll.”

In this activity, we'll publish to CloudCannon 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 simply 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.

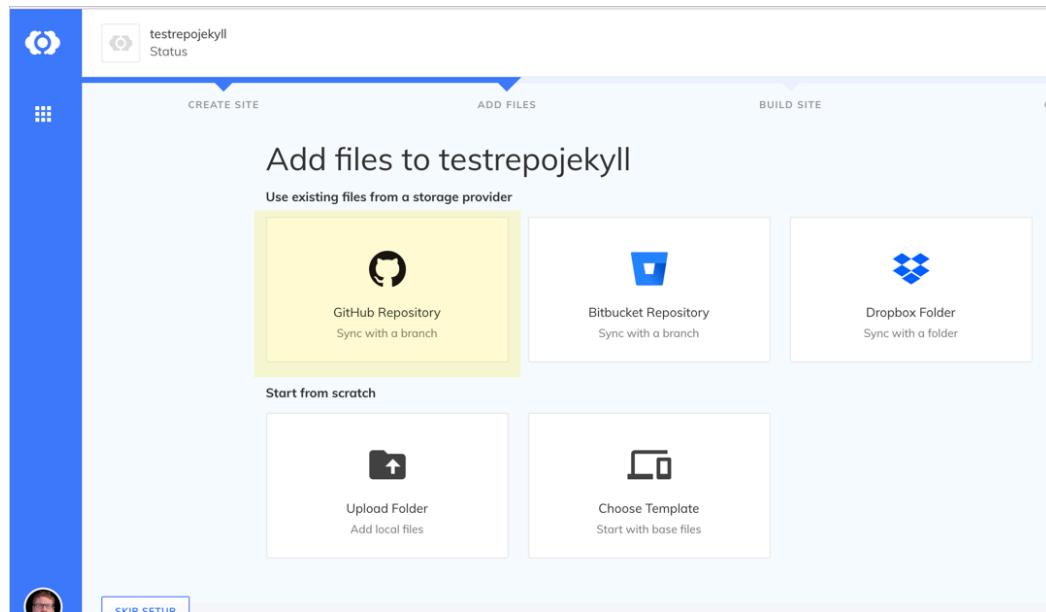
Set up your doc theme on Github

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.

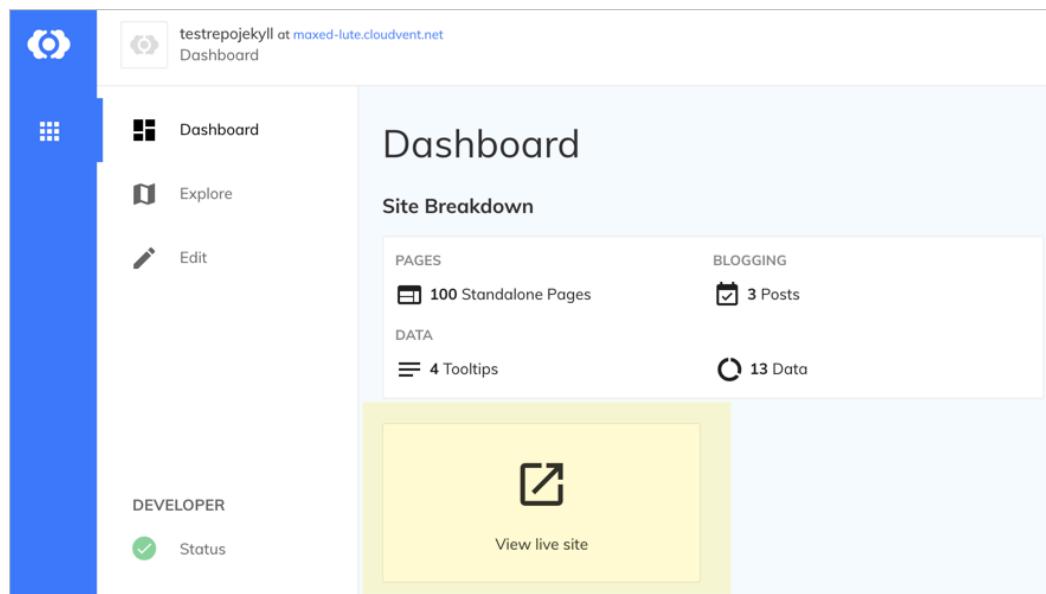
Sometimes people fork repositories to make changes and then propose pull requests of the fork to the original repo. Other times people fork repositories to create a starting point for a splinter project from the original. Github is all about social coding — one person's ending point is another person's 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.
4. Click the **GitHub Repository: Sync with a branch** 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.
7. When the sync finishes, click **Start Site Build** to have CloudCannon build Jekyll from the server.
When it finishes building, you see “Your first build is complete!”.
8. Click **Go to Editor Dashboard**.
9. From the Dashboard, click the **View Live Site** box:



[View Live Site](#)

The preview URL will be something random, such as <https://doted-lily.cloudvent.net/>. 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.

I have to say, 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.

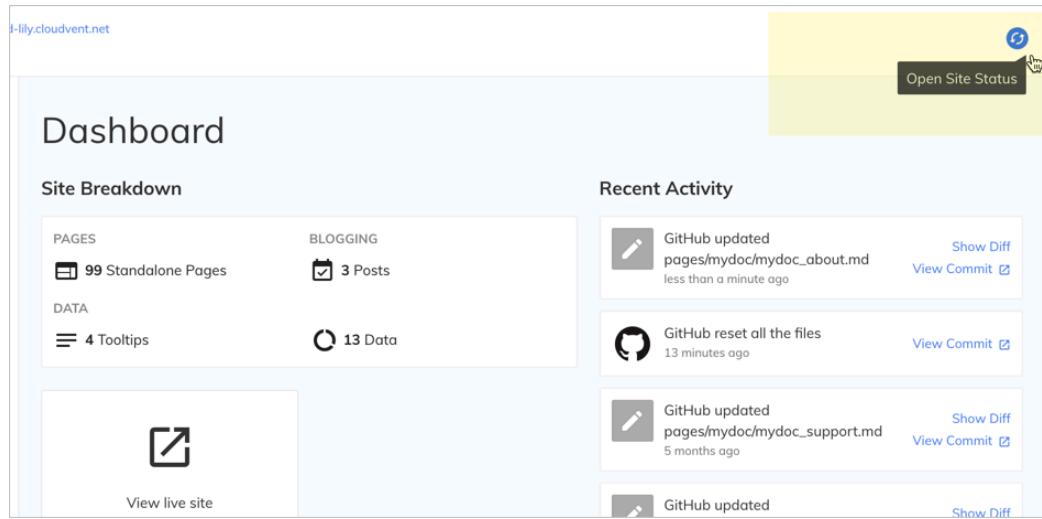
1. In your browser, go to your GitHub repository that you forked and make a change to a file.

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.



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 is the essential characteristic of a [docs-as-code publishing workflow \(page 356\)](#).

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 others Jekyll themes. I also wrote a tutorial for [creating your own Jekyll theme](#), likely using your company's site as the template.

The screenshot shows a Jekyll site using the CloudCannon Aviator theme. The left sidebar has sections for Search, Documentation (Getting Started, Authentication, Errors), APIs (with links to /books, /books:id, /books:id PUT, /books:id DELETE), and a Template by CloudCannon. The main content area shows the API endpoint '/books' with a GET method. It describes listing all books, parameters for offset and limit, and a note that it returns a maximum of 100 books. Below this is a code example for jQuery:

```
$.get("http://api.myapp.com/books/", { "token": "YOUR_APP_KEY"}, function(data) {
  alert(data);
});
```

To the right, there's a 'Response' tab showing JSON data for two book entries:

```
[{"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 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.

Additionally, you will likely need to make this change outside your normal 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 \(page 356\)](#). 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 designed specifically 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 a number of 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 halfway with our docs-as-code approach and still had room 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 in order 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 a larger 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. 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 our team's processes. We felt like a huge 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 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 whereas I had in mind a doghouse, they had in mind a skyscraper. So their processes were probably more or less scaled and scoped to the business needs and requirements.

Still, we lived in different worlds, and we had to constantly 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 main 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 really 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 tool 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 just 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 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, 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. This 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 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 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 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 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 difficult. We started out translating the Markdown source. Our translation vendor affirmed they 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 could even be used. 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 individual 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. This 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 value was 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 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 special 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, actually the greater challenge was becoming familiar with Git-based workflows for doc content. This 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 main 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 had to be able to scale in an enterprise way. Because the content used Markdown as the format, anyone could easily 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.

Also significant is that no licenses or seats were required to scale out the authoring. A writer just uses 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 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, there wasn't a huge value or time allotment for creating 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 easier to follow.

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 needs to be done, our priorities are always focused on creating documentation content. No one wants to acknowledge the time and energy required to support your own tooling process. While much of this hassle could have simply 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 their development and maintenance).

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 simply 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 \(page 380\)](#).) 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 extremely important *when you're implementing your solution*. After you're finished, you no longer need all that knowledge, and I find it somewhat fading from my awareness. This is probably why so many consultants specialize in tools — they swoop in, 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.

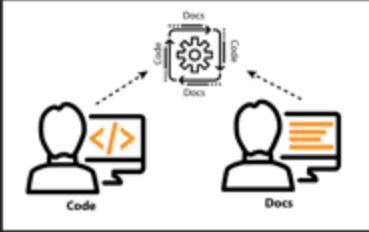
And yet, platforms and tools are rarely static for any duration of time. Even Jekyll continually releases new versions, responding to changes in the Internet landscape and trending technology needs. So maybe in a few years, we'll go through this whole process again. Even so, I have a propensity and facility 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



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:



It was also republished in Anne Gentle's [Docs Like Code: Case Studies](#):

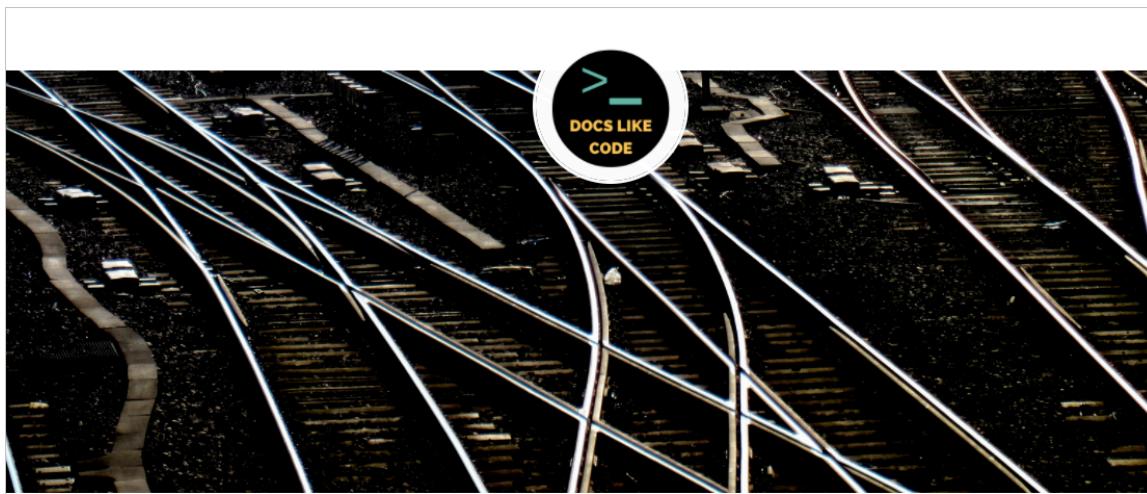


Image credit: Flickr laurencehorton

ENGINEERING • SECURITY • GITHUB • GIT • JEKYLL • STATIC SITE • CHANGE • CULTURE • TEAMWORK • COLLABORATION
• DEVELOPMENT

Case study: Switching tools to docs-as-code



BY TOM JOHNSON

FEBRUARY 12, 2018

FEBRUARY 12, 2018

TWEET

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.

To Learn more about docs as code, see Anne Gentle's book [Docs Like Code](#).

Video recording

I recently [gave a presentation on Docs-as-code tools and workflows](#) to the STC Rocky Mountain and WTD Denver group.

This content doesn't embed well in print, as it contains YouTube videos. Please go to https://idratherbewriting.com/learnapidoc/pubapis_overview.html to view the content.

Blog posts about docs-as-code tools

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](#)

Getting a job in API documentation

The job market for API technical writers.....	439
How much code do you need to know?	443
Locations for API doc writer jobs	453

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. In this section of my API documentation course, I'll dive 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

This is why in this course I've focused on activities that will actually 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.

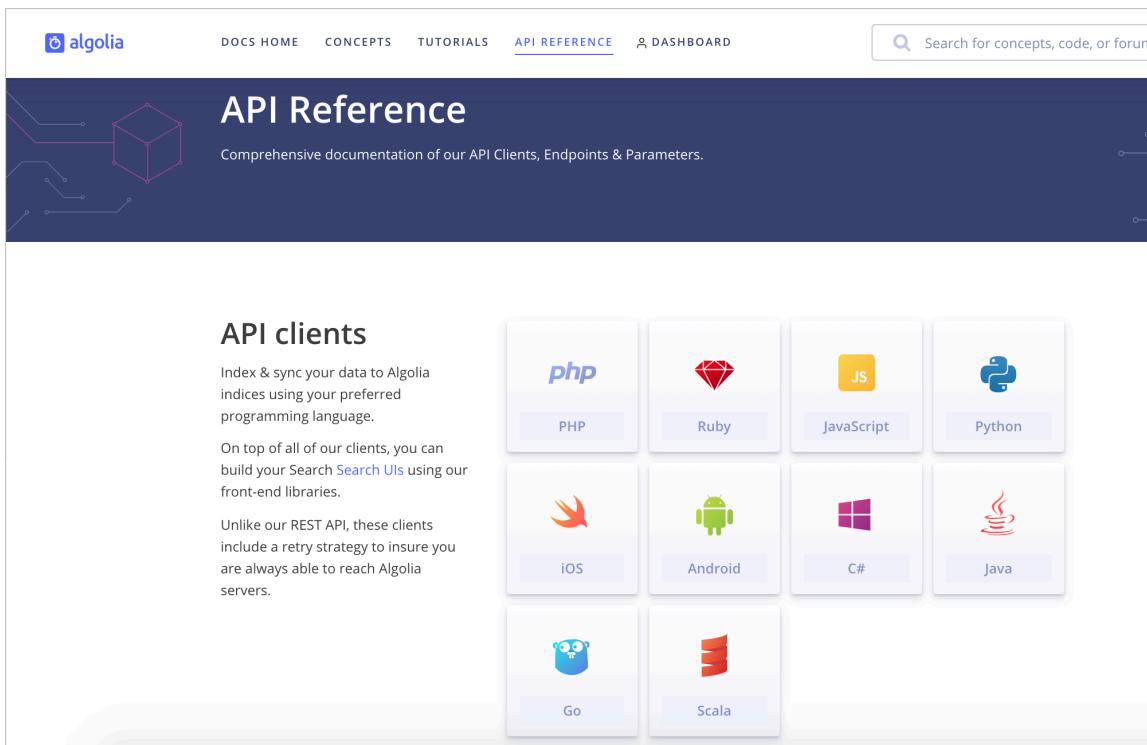
Why employers look for candidates who can read programming languages

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 actually *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 in various languages for the REST API.

Take a look at Algolia's API for an example. 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.



The screenshot shows the Algolia API Reference homepage. At the top, there's a navigation bar with links to 'DOCS HOME', 'CONCEPTS', 'TUTORIALS', 'API REFERENCE' (which is underlined in blue), and 'DASHBOARD'. To the right of the navigation is a search bar with the placeholder 'Search for concepts, code, or forum'. Below the navigation, there's a large header section with the title 'API Reference' and a subtitle 'Comprehensive documentation of our API Clients, Endpoints & Parameters.' On the left side of this section, there's a decorative graphic of a cube with glowing lines and nodes. To the right, there's a grid of icons representing different client languages and platforms:

 PHP	 Ruby	 JavaScript	 Python
 iOS	 Android	 C#	 Java
 Go	 Scala		

Although users could construct their own code when using the REST endpoints, most developers would rather leverage existing code and just 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 just use the pre-built widgets.

Remember that developers are typically using a REST API as a *third-party* service. The developer's main focus is his or her own company's code; they're just leveraging your REST API as an additional, extra service. As such, the developer wants to just get in, get the code, and get out. This is why companies need to provide multiple client SDKs in as many languages as possible — these client implementations make it easy for developers to implement the API.

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 JavaScript-only shop where all you need to know is JavaScript and nothing more. If that's the case, you'll need to develop a deeper knowledge of JavaScript 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 \(page 308\)](#).

Providing value without deep 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 – 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 deeper knowledge of the subject matter.

Still, this lack of more technical knowledge will likely remove some of the value from your role. In [How API Documentation Fails](#) (published in *IEEE Software*, 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 deep, authoritative knowledge of the API, it will be difficult to complete, clarify, and correct errors in the content.

The balance between generalist and specialist roles is an ongoing challenge that I’ll dive into more in the next topic: [How much code do you need to know? \(page 443\)](#) But in short, if you want to solve the biggest problem with API documentation, you’ll need to develop more technical expertise in the subject area.

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. 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 simply 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? \(page 443\)](#), 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 a deep technical knowledge of Java, Python, C++, .NET, Ruby, in addition to mastering docs tools 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 regular 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 mindset may better understand you, but bad because anyone who doesn't inhabit that perspective and terminology already will be somewhat lost.

Writers who learned programming

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 I'll 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 often 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.

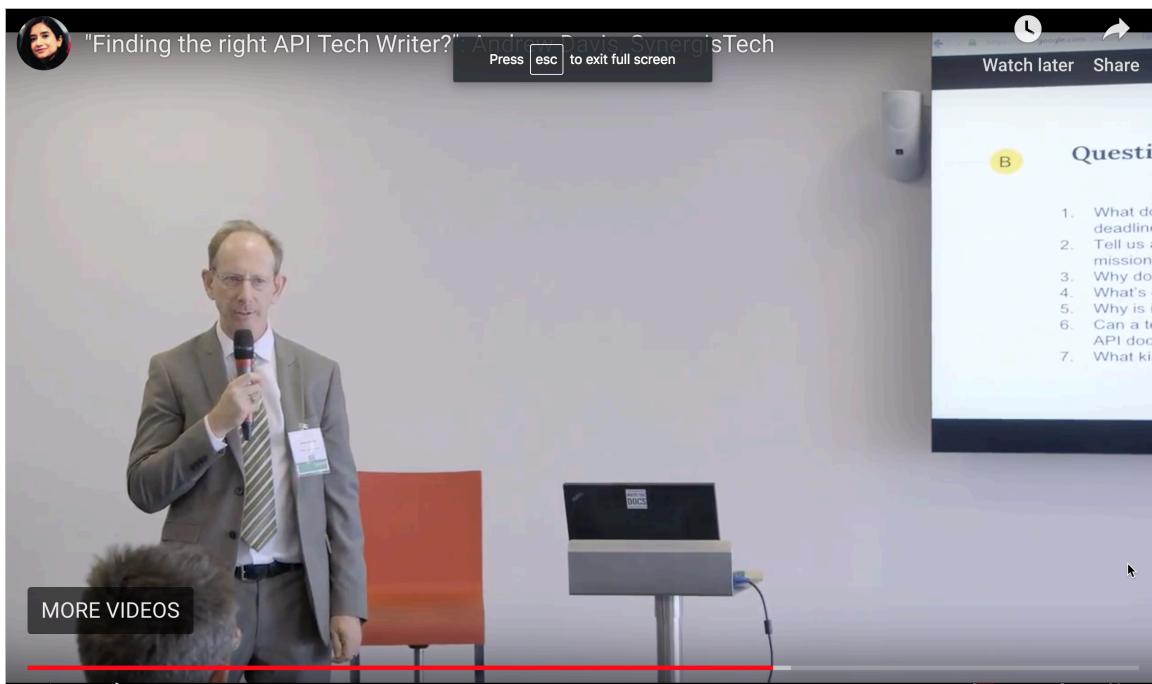
Additionally, while engineers can fill in the deep technical knowledge needed, no one will provide the tech comm tasks in place of a tech writer. As evidence, just look at any corporate wiki. Corporate wikis are prime examples of what happens when engineers (or other non-tech writers) write and publish 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 writing

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 more 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:



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 (which 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 simply send the tech writer code snippets 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 go 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 engineer's 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 myriad other languages tends to be more of the programmer's responsibility, not the tech writer's. Then again, maybe Neiman's role as a consultant and his background in engineering allows him to perform tasks that I cannot (and because I cannot, I have not been asked to).

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 properly run the code. 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 \(page 247\)](#), you have to be able to test the requests and test the 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 in the review process.

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 India. 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, only specialists can create it), then shouldn't companies prioritize writing abilities over (waning) technical abilities?

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 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 they 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. Then again, a combination of the two skills tends to be a knockout punch in the job market.

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 a 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 really easy to pull off, though. 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.

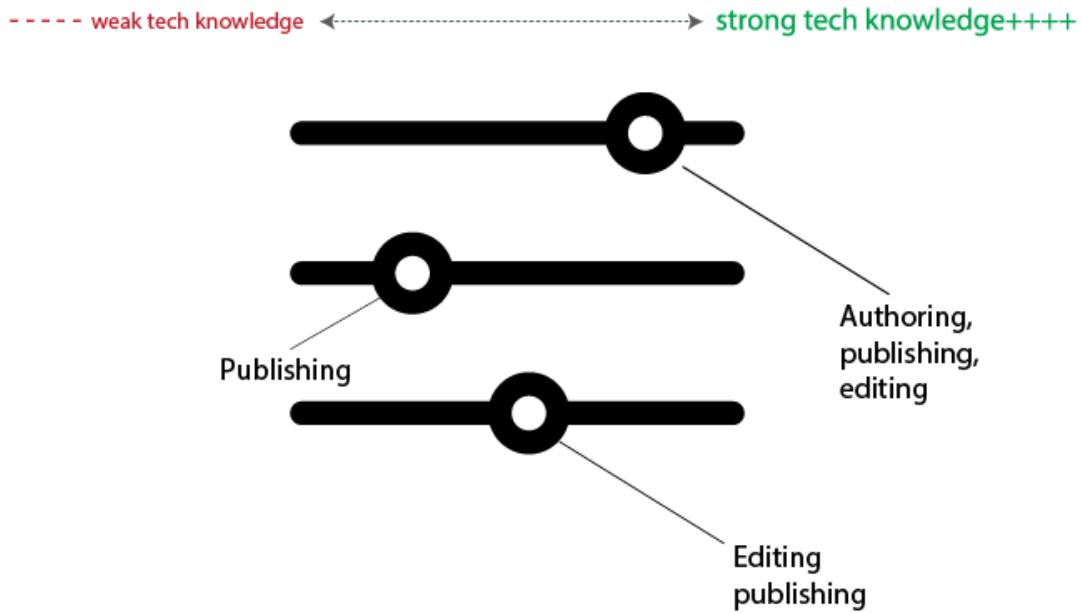
Undoubtedly, technical writers are at a disadvantage when it comes to learning programming. Full immersion is the only way to become fluent in a language, whether referring to programming languages or spoken languages. To get fully immersed, you might consider diving deep into one core programming language (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



The degree to which you can publish, edit, or 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 (even 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, the 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 information interviews. The details of *what* will either be apparent in the code or can be minimized. (See [Code samples and tutorials \(page 298\)](#) for details.)

Just 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 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.

Not an easy problem to solve

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 at least a regular regimen of programming study — you always have to be learning to survive in this field.

Fortunately, there are many helpful resources (my favorite being [Safari Books Online](#)). If you can work in a couple of hours a day, you'll be surprised at the progress you can make.

Some of the principles that are fundamental to programming, like variables, loops, and try-catch statements, will begin to feel second nature, since these techniques are common across almost all programming languages. You'll also be equipped with a confidence that you can learn what you need to learn on your own (the hallmark of a good education).

But in discussions with hiring managers looking to fill contracts for technical writers already familiar with their programming environment, it will be a hard sell to persuade the manager that “you can learn anything.”

The truth is that you can learn anything, but it may take a long time to do so. It can take years to learn Java programming, and you'll never get the kind of project experience that would give you the understanding that a full-time developer possesses.

Techniques for learning code

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?

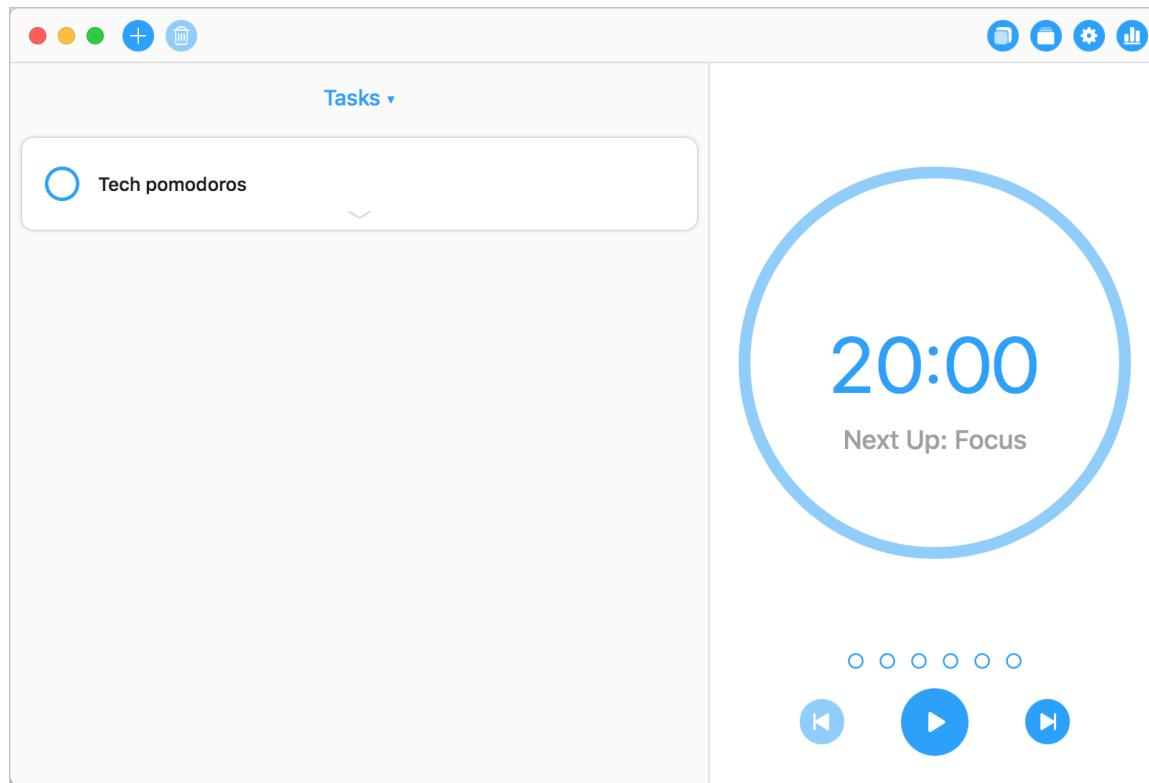
For example, do you dedicate 2 hours a day to simply learning to code in the particular language of the product you're documenting? Should you carve this time out of your employer's time, or your own, or both? How do you get other doc work done, given that meetings and miscellaneous tasks usually eat up another 2 hours of work time? What strategies should you implement to actually learn code in a way that sticks? What if what you're 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. 60 minutes is more appropriate, but really, if you want to make progress, you'll need to devote about twice this time. Finding 1-2 hours of time 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. The information I need to document in the present moment is usually too advanced to simply learn from watching tutorials on SafariBooks 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. And yet, I have a busy home life as well (4 daughters, all at home!), so I have to find time between work and home to get some learning done. (Did I also mention that I blog a lot?)

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.

<figure>



<figcaption>The Focus app lets you define and track pomodoros</figcaption></figure>

Even so, this technique 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. These explanations should focus on the *why* more than the *how*.
- You should be able to test code from engineers in order to identify assumptions that engineers are often blind to.
- To thrive in an API documentation career, you have to incorporate a regimen of continual learning.

Is being a generalist disappointing from a career standpoint?

Technical writers will likely be generalists with the code, not really 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 in a thorough way, right?

It might seem depressing to realize that your coding knowledge will usually be kindergartner-like in comparison to developers. This 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 awesome *documentation*. Creating awesome 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 direction 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 and sample apps \(page 308\)](#)
- [Code samples and tutorials \(page 298\)](#)

Locations for API doc writer 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

Undoubtedly, Silicon Valley (the San Francisco Bay area of California) is where many API documentation jobs are located. However, you will also find API documentation jobs for technical writers in other tech hub cities, such as Seattle, New York, Portland, Boston, Dallas, Atlanta, Austin, and more (basically West Coast, East Coast, and Texas). As far as statistics or percentages detailing the number of **API tech writing jobs by state**, that research hasn't been done, so we have to triangulate a bit.

Let's start with what we know. According to the 2016 STC Salary Database (which is based on the Bureau of Labor Statistics data), the "10 largest metro areas by employment of tech writers" are as follows:

1. New York-Newark-Jersey City, NY-NJ-PA
2. Los Angeles-Long Beach-Anaheim, CA
3. Chicago-Naperville-Elgin, IL-IN-WI
4. San Jose-Sunnyvale-Santa Clara, CA
5. Minneapolis-St. Paul-Bloomington, MN-WI
6. Houston-The Woodlands-Sugar Land, TX
7. Baltimore-Columbia-Towson, MD
8. Denver-Aurora-Lakewood, CO
9. Phoenix-Mesa-Scottsdale, AZ
10. San Diego-Carlsbad, CA

In other words, these areas have the most technical writers. Much of the employment data in the STC Salary Database focuses on "metropolitan statistical areas," but it hard to filter jobs based on these metro areas (rather than states) using job tools such as [Indeed.com](#).

To gather some quick data, I did a search on Indeed.com for the term "[API](#)" filtered by state, another search for "[technical writer](#)" filtered by state, and then listed the employment of technical writers by state as indicated by the STC Salary Database (2016 is the latest version). My searches on Indeed.com were done on August 12, 2018.

Whether searches for these terms yields meaningful results is not certain, which is why I also list the more reliable information (though unrelated to APIs) from the STC Salary Database. Also, I'm by no means versed in statistics. Even if California has the largest number of mentions of "API" and also the largest mentions of "technical writer," this data has to be contextualized by the size of the state and the number of employed technical writers in the area. A high number of jobs doesn't necessarily mean more jobs will be available if there are simply more technical writers competing for those jobs. Thus, it's hard to say for sure where the best locations are for API doc jobs.

For example, suppose the state is small and has only about 500 tech writers in the area, but the number of jobs for technical writers is high as well as the mentions of "API." Is that a better area in which to find a job? Maybe. It depends on how many writers are competing for the jobs. 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 jobs 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.

All right, here's the data from my quick search:

State	Instances of "API" on Indeed	Instances of "technical writer" on Indeed	Employed technical writers (STC Salary DB)
California	4,983	1,152	6,590
Texas	2,450	480	3,930
New York	1,735	403	2,390
District of Columbia	1,381	951	800
Washington State	1,283	333	1,190
Virginia	1,092	625	2,590
Massachusetts	1,080	310	2,700
Illinois	949	238	1,680
Georgia	879	227	990
New Jersey	800	209	1,860
Pennsylvania	800	277	1,600
Florida	722	314	2,130
Colorado	635	183	1,270
North Carolina	631	198	1,570
Maryland	567	453	1,980

State	Instances of "API" on Indeed	Instances of "technical writer" on Indeed	Employed technical writers (STC Salary DB)
Ohio	466	157	1730
Arizona	404	153	970
Minnesota	389	100	1,200
Michigan	351	146	1,000
Missouri	322	98	1,030
Utah	275	88	680
Oregon	271	77	600
Connecticut	229	51	690
Wisconsin	225	104	1,120
Tennessee	212	85	670
Indiana	204	73	730
Oklahoma	169	50	720
Louisiana	167	28	220
Iowa	130	47	500
Alabama	115	77	660
South Carolina	110	68	480

State	Instances of "API" on Indeed	Instances of "technical writer" on Indeed	Employed technical writers (STC Salary DB)
Nebraska	102	23	300
Kansas	101	62	540
Kentucky	95	43	290
Rhode Island	73	22	170
Nevada	69	46	220
Delaware	65	27	140
Arkansas	56	21	170
New Hampshire	51	137	220
North Dakota	46	17	70
West Virginia	37	10	120
New Mexico	36	42	300
Idaho	35	21	210
Alaska	31	13	70
South Dakota	25	16	100
Maine	20	194	100
Montana	17	8	70

State	Instances of "API" on Indeed	Instances of "technical writer" on Indeed	Employed technical writers (STC Salary DB)
Hawaii	15	8	70
Wyoming	12	4	0
Vermont	8	8	120
Mississippi	7	18	90
Puerto Rico	4	2	150

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, New York, D.C., Washington State, Virginia, Massachusetts, Illinois, and Georgia are hot areas for API jobs for tech writers.

To provide more of a percentage relative to the area size, I divided the API number by the number of tech writers employed in the state. With this weighting, the top 10 states were as follows:

1. Massachusetts
2. DC
3. Washington State
4. Georgia
5. Louisiana
6. California
7. New York
8. North Dakota
9. Texas
10. Illinois

Presumably, this weighting suggests that API jobs make up a higher percentage of jobs in the area. Weighting the states like this might be meaningless. Again, maybe Maine has a real shortage to fill a dozen open positions for tech writers where applicants are largely absent, whereas California might have 200 positions but many hundreds more applicants competing for them.

I'll leave the statistics and number-crunching to academics (who have more time to research and analyze this topic). Overall, I recommend locating yourself in any of these states:

- California
- Texas
- New York
- District of Columbia
- Washington
- Virginia
- Massachusetts
- Illinois

- Georgia

Should you move to California?

Since I'm currently in California, 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. However, California has a number of drawbacks, mainly with housing and traffic.



Should you move to California (or stay here if you're already in California)?

To live in Silicon Valley, you 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. 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. In fact, the STC Salary Database indicates that "San Jose-Sunnyvale-Santa Clara" had the second largest reported loss of tech writers during 2016 (second to Houston-The Woodlands-Sugar Lands, TX). This might just be anecdotal, but 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 there.

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, 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 (where the jobs flow like milk and honey). 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 5 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 on 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).

Before I moved into 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. Seattle also has an abundance of jobs, and the real estate market slightly better 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 gets fewer and fewer as companies grow. Many say that Silicon Valley's days are over, since 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 more thoughts on API tech writer jobs in Silicon Valley, see [Do you have to relocate to an urban tech hub to find a technical writing job?](#).)

How to conduct a simple test

If you're curious about the job opportunities in your area, there's a simple way to gauge how many technical writing jobs a city provides. 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 month.

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 a 2-3 times more than in Seattle.

In this case, I didn't run comparisons for more cities, nor did I add "API" into the job title. I'm willing to bet that API technical writing jobs will always be a fraction of normal technical writing jobs (maybe 15%?).

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, New York, Texas, D.C. Washington State, Massachusetts, Virginia, Georgia, and Illinois. On the other hand, if you want to start a farm or ranch, move to Wyoming.

Activity 8a: 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 will likely need to dedicate more time to the [open-source documentation project \(page 137\)](#) that you identified earlier.

Glossary

API Glossary	462
--------------------	-----

API glossary

API

Application Programming Interface. Enables different systems to interact with each other programmatically. Two types of APIs are web services and library-based APIs. See [What is a REST API? \(page 22\)](#).

API Console

Renders an interactive display for the RAML spec. Similar to Swagger UI, but for [RAML \(page 0\)](#). 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 \(page 0\)](#) 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/>.

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](#).

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 back 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 \(page 47\)](#) and [Understand curl more \(page 49\)](#).

endpoint

The endpoints indicate how you access the resource, and the method used with the endpoint indicates the allowed interactions (such as GET, POST, or DELETE) with the resource. The endpoint shows the end path of a resource URL only, not the base path common to all endpoints.

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. See [Endpoints and methods \(page 89\)](#).

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 \(page 394\)](#) in this course as well as on [pages.github.com/](#).

git repo

A tool for consolidating and managing many smaller repos with one system. See [git-repo](#).

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 tend to use [Docs as code tools \(page 356\)](#) more than [HATs \(page 0\)](#).

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 \(page 369\)](#).

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 \(page 59\)](#) in this course and <http://www.json.org/>

Mercurial

An 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 \(page 89\)](#).

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 \(page 0\)](#). See <https://www.mulesoft.com/>.

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. See <https://github.com/OAI/OpenAPI-Specification>. Now under the Open API Initiative with the Linux Foundation, the OpenAPI specification aims to be vendor neutral. For details in this course, see [Introduction to the OpenAPI specification and Swagger \(page 143\)](#).

OpenAPI contract:

Synonym for OpenAPI specification document.

OpenAPI specification document

The specification document, usually created manually, that defines the blueprints that developers should code the API to. The contract aligns with a “spec-first” or “spec-driven development” philosophy. The contract essentially acts like the API requirements for developers. See [this blog post/podcast](#) for details on spec-first development.

OpenAPI Initiative

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

parameter

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 four types of parameters: header parameters, path parameters, query string parameters, and request body parameters.

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 \(page 95\)](#) for more details.

Path parameters

Parameters that appear within the path of the endpoint, before the query string (`?`). These are usually set off within curly braces. See [Parameters \(page 95\)](#) 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 \(page 369\)](#).

Perforce

Revision control system often used before Git became popular. Often configured as 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 own 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 has a great 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>.

Query string parameters

Parameters that appear in the query string of the endpoint, after the `?`. See [Parameters \(page 95\)](#) for more details.

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 \(page 0\)](#) 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

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 (but not with binary files).

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 \(page 104\)](#) for more details.

request body parameters

Parameters that are included in the request body. Usually submitted as JSON. See [Parameters \(page 95\)](#) for more details.

request example

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. See [Request example \(page 104\)](#) for more 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 provides details about the information returned in each resource.

The resource description is brief (1-3 sentences) and usually starts with a verb. Resources usually have a number of endpoints to access the resource, and multiple methods for each endpoint. Thus, 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 \(page 84\)](#) 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 \(page 116\)](#) for details.

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. See [Response example and schema \(page 116\)](#) 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? \(page 22\)](#) for more details.

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 and sample apps \(page 308\)](#) 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 \(page 374\)](#).

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 \(page 417\)](#) in this course for a deep-dive into the most popular static site generator, or [Staticgen](#) for a list of all static site generators. Also see [Static site generators \(page 369\)](#) 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. Also see [Stoplight – visual modeling tools for creating your OpenAPI spec \(page 233\)](#).

Swagger

Refers to general API tooling to support OpenAPI specifications. See [swagger.io/](#). See [Swagger UI tutorial \(page 214\)](#) for details.

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 and sample apps \(page 308\)](#).

Swagger Editor

An online editor that validates your OpenAPI document against the rules of the spec, showing validation errors as found. See [Swagger editor](#).

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 \(page 160\)](#).

Swagger UI

The most common display framework for parsing an OpenAPI specification document and producing the interactive documentation as shown in the [Petstore demo site](#). See [Swagger-UI](#)

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 \(page 224\)](#).

VCS

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

version control

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. Also see [Version Control Systems \(page 366\)](#).

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 \(page 156\)](#).