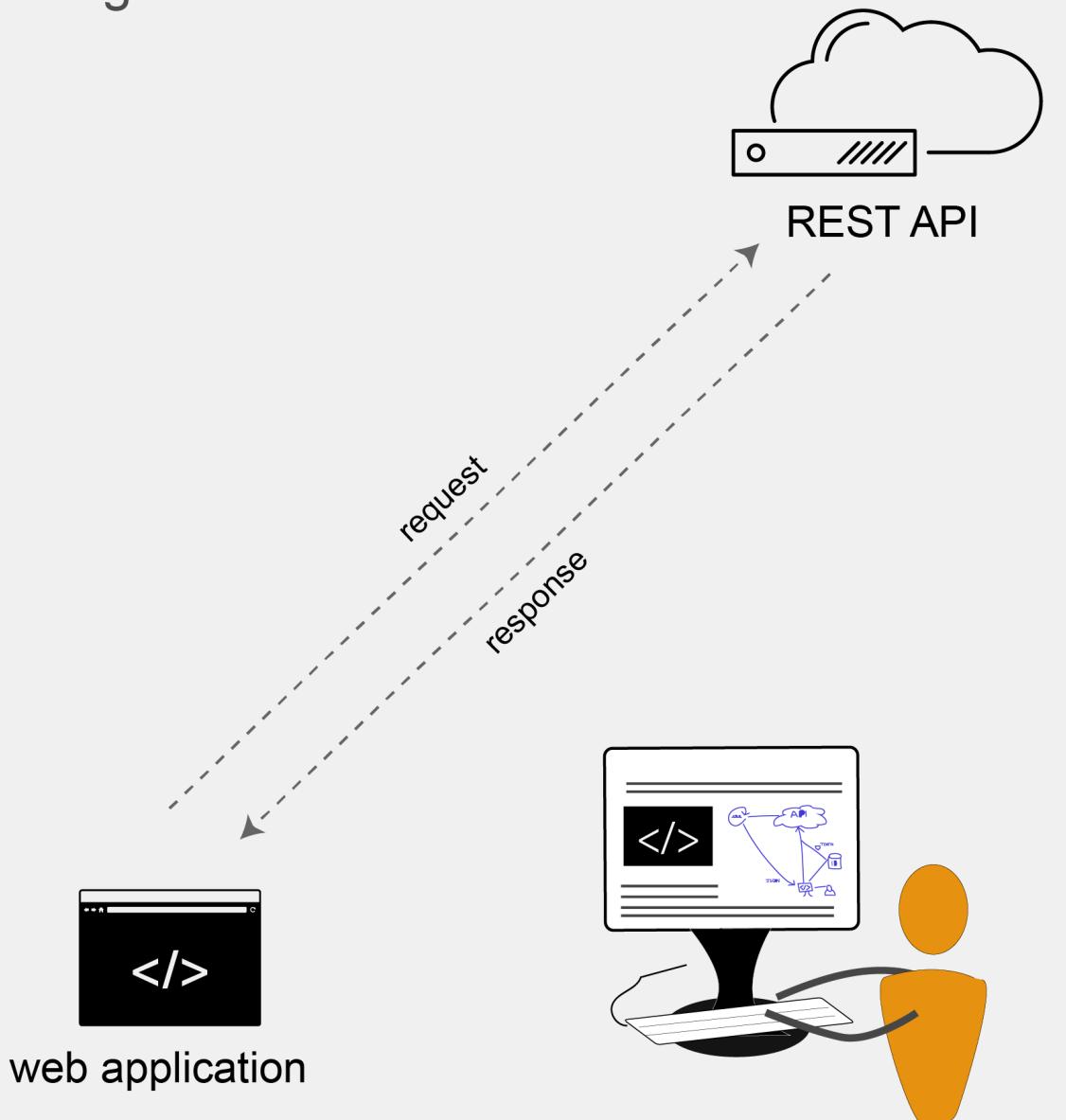


Documenting REST APIs

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

Introduction to REST APIs	6
Course Overview.....	7
Recorded video presentations.....	11
Course Background	12
About the author	14
The market for REST API documentation.....	16
What is a REST API?.....	23
Activity: Find an Open Source Project.....	31
Workshop activities.....	35
Using a REST API as a developer	39
Scenario for using a weather API.....	40
Get authorization keys	44
Submit requests through Postman	46
curl intro and installation.....	52
Make a curl call.....	54
Understand curl more	56
Use methods with curl	60
Analyze the JSON response	65
Use the JSON from the response payload	69
Access and print a specific JSON value.....	74
Dive into dot notation.....	78
Documenting API reference topics.....	83
A new endpoint to document	84
API reference tutorial overview	87
Step 1: Resource description (API ref tutorial).....	89
Step 2: Endpoints and methods (API ref tutorial)	95
Step 3: Parameters (API ref tutorial)	101
Step 4: Request example (API ref tutorial).....	110
Step 5: Response example and schema (API ref tutorial)	121
Putting it all together.....	137
Activity: Critique or create an API reference topic.....	142
Testing your API documentation.....	144
Overview to testing your docs	145
Set up a test environment.....	146
Test all instructions yourself	149
Test your assumptions.....	154
Activity: Test your project's documentation	158

Documenting non-reference sections	159
User guide tasks	160
Documenting the API overview	165
Documenting the getting started section	166
Documenting authentication and authorization requirements	170
Documenting status and error codes	177
Documenting code samples and tutorials	184
How to create the quick reference guide	188
Next phase of course	190
Publishing your API documentation.....	191
Overview for publishing API docs	192
List of 100 API doc sites	195
Design patterns with API doc sites	199
Docs-as-code tools	209
More about Markdown	215
Version control systems.....	224
Publishing tool options for developer docs	227
Which tool to choose for API docs — my recommendations.....	249
Activity: Manage content in a GitHub wiki	253
Activity: Use the GitHub Desktop Client	259
Pull request workflows through GitHub	268
Jekyll publishing tutorial	274
Case study: Switching tools to docs-as-code.....	282
OpenAPI specification and Swagger	295
Overview of REST API specification formats	296
Introduction to the OpenAPI spec and Swagger	298
OpenAPI tutorial overview	310
Step 1: openapi object (OpenAPI tutorial)	314
Step 2: info object (OpenAPI tutorial)	318
Step 3: servers object (OpenAPI tutorial).....	320
Step 4: paths object (OpenAPI tutorial)	322
Step 5: components object (OpenAPI tutorial)	331
Step 6: security object (OpenAPI tutorial).....	345
Step 7: tags object (OpenAPI tutorial)	349
Step 8: externaldocs object (OpenAPI tutorial).....	351
Activity: Create an OpenAPI specification document	354
Swagger UI tutorial	355
Activity: Create your own Swagger UI display	363
Swagger UI Demo	365
Integrating Swagger UI with the rest of your docs	366
SwaggerHub introduction and tutorial	372
Other tools to parse and display OpenAPI specs.....	382

More about YAML	384
RAML tutorial	388
API Blueprint tutorial	402
Documenting native library APIs.....	414
Overview of native library APIs	415
Get the Java source.....	418
Java crash course.....	423
Activity: Generate a Javadoc from a sample project.....	429
Javadoc tags.....	434
Explore the Javadoc output.....	442
Make edits to Javadoc tags.....	445
Doxygen, a document generator mainly for C++.....	446
Create non-ref docs with native library APIs	448
Getting a Job in API Documentation	449
The job market for API technical writers.....	450
How much code do you need to know?	455
Resources and glossary.....	458
Glossary	459
Overview for exploring other REST APIs	466
EventBrite example: Get event information	467
Flickr example: Retrieve a Flickr gallery	474
Klout example: Retrieve Klout influencers	483
Aeris Weather Example: Get wind speed	493
API documentation survey	499

Introduction to REST APIs

Course Overview	7
Recorded video presentations	11
Course Background.....	12
About the author.....	14
The market for REST API documentation.....	16
What is a REST API?	23
Activity: Find an Open Source Project.....	31
Workshop activities	35

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.

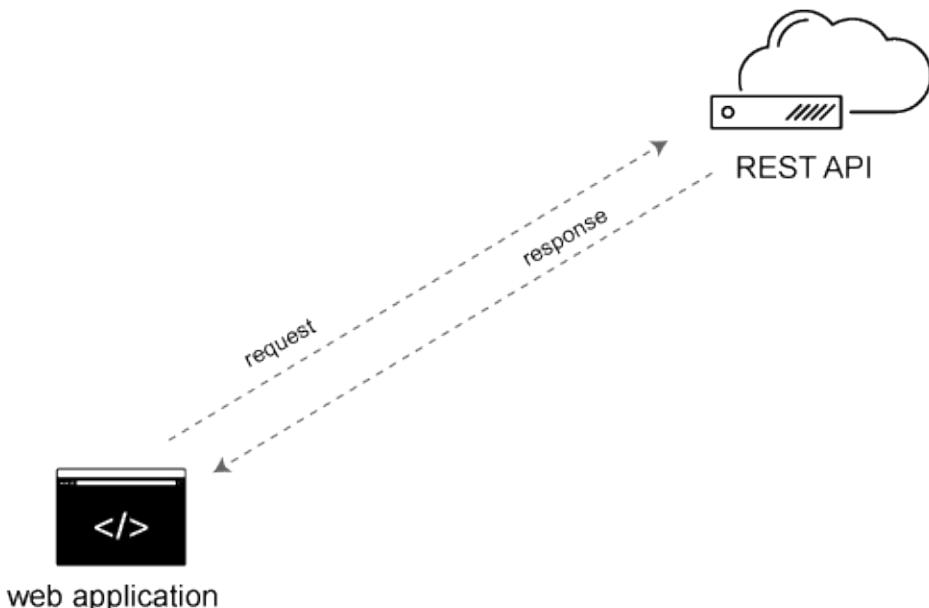
You'll learn about API documentation in the context of using some simple weather APIs to put a weather forecast on your site.

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 details 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? \(page 23\)](#). 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 \(page 89\)](#)
2. [Endpoints and methods \(page 95\)](#)
3. [Parameters \(page 101\)](#)
4. [Request example \(page 110\)](#)
5. [Response example \(page 121\)](#)

Diving into these sections will give you a solid understanding of how to document REST APIs. You'll also learn how to document the [non-reference sections for an API \(page 159\)](#), such as the [getting started \(page 166\)](#), [status and error codes \(page 177\)](#), and request [authorization \(page 170\)](#) sections.

Finally, you'll dive into different ways to [publish REST API documentation \(page 191\)](#), exploring tools and specifications such as [GitHub \(page 253\)](#), [Jekyll \(page 274\)](#), and other [Docs-as-code approaches \(page 209\)](#). 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 \(page 224\)](#).

I also dive into specifications such as the [OpenAPI specification \(page 310\)](#) and [Swagger \(page 298\)](#), which provides tooling for the OpenAPI specification. Additionally, I included a section on [documenting native library APIs \(page 414\)](#) and generating [Javadoc \(page 429\)](#). 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 \(page 39\)](#)
- [Documenting endpoints \(page 83\)](#)
- [Testing your API documentation \(page 144\)](#)
- [Documenting non-reference sections \(page 159\)](#)
- [Publishing your API documentation \(page 191\)](#)
- [OpenAPI specification and Swagger \(page 295\)](#)
- [Documenting native library APIs \(page 414\)](#)
- [Getting a job in API documentation \(page 449\)](#)
- [Resources and glossary \(page 458\)](#)

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 \(page 39\)](#), and the section on [documenting endpoints \(page 83\)](#)) 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 include an activity graphic like this:

ACTIVITY



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. However, this content is just as much a book or website as it is a course.

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](#).

What you’ll need

Here are a few things 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 this browser. [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](#).
- **Git.** [Git](#) is a version control tool developers often use to collaborate on code. See [Set Up Git](#) for more details.
- **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 App.** [Stoplight](#) provides visual modeling tools for working with the OpenAPI specification. Create a Stoplight account and download the [Desktop app](#) [here](#).

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’ve got to do all the activities outlined in each section. These activities are key to building experience and credibility with a portfolio. These in-depth activities are identified with the word “Activity” in the topic title. There is usually one main activity for each section:

- Activity: Find an Open Source Project (page 31)
- Activity: Critique or create an API reference topic (page 142)
- Activity: Test the docs in your Open Source project (page 158)
- Activity: Manage content in a GitHub wiki (page 253)
- Activity: Create an OpenAPI specification document (page 354)
- Activity: Create your own Swagger UI display (page 363)
- Activity: Generate a Javadoc from a sample project (page 429)

Video recordings

For video recordings of this course, see the [Recorded Video Presentations \(page 11\)](#).

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

(By the way, these slides are all hosted on GitHub at <https://github.com/tomjoht/>. I use [RevealJS](#) for slides, which lets me create the slide content in HTML.)

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 at <https://tinyletter.com/tomjoht>.

Recorded video presentations

I've given numerous presentations on API documentation, and most of them I've recorded. The most recent presentations are available below. Keep in mind that my API content evolves, so some older presentations might no longer match the course content.

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

Course Background

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 API doc.

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.

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. 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, through this feedback I helped clarify and strengthen the overall content.

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 a digital publication, 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.

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. I work at [Amazon Lab126](#), which builds the devices for many of Amazon's products (such as the Kindle, Echo, Fire TV, and more).

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 nonfiction creative writing, 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 boring), I found that I actually liked technical writing — a lot more than copywriting. Technical writing combined my love for writing with my fascination for technology. I got to play with tools and handle all aspects of content production from designing web styles to configuring 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 into 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 don'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.



That's probably why you're taking this course — because you want to develop your skills and knowledge to increase your capabilities at work, enhance your skillset's marketability, and 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. If you have feedback during any part of the course, feel free to drop me a line or comment on one of the pages.

Tom Johnson

tom@idratherbewriting.com

San Francisco Bay area, California, USA

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, because the library has become part of 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.

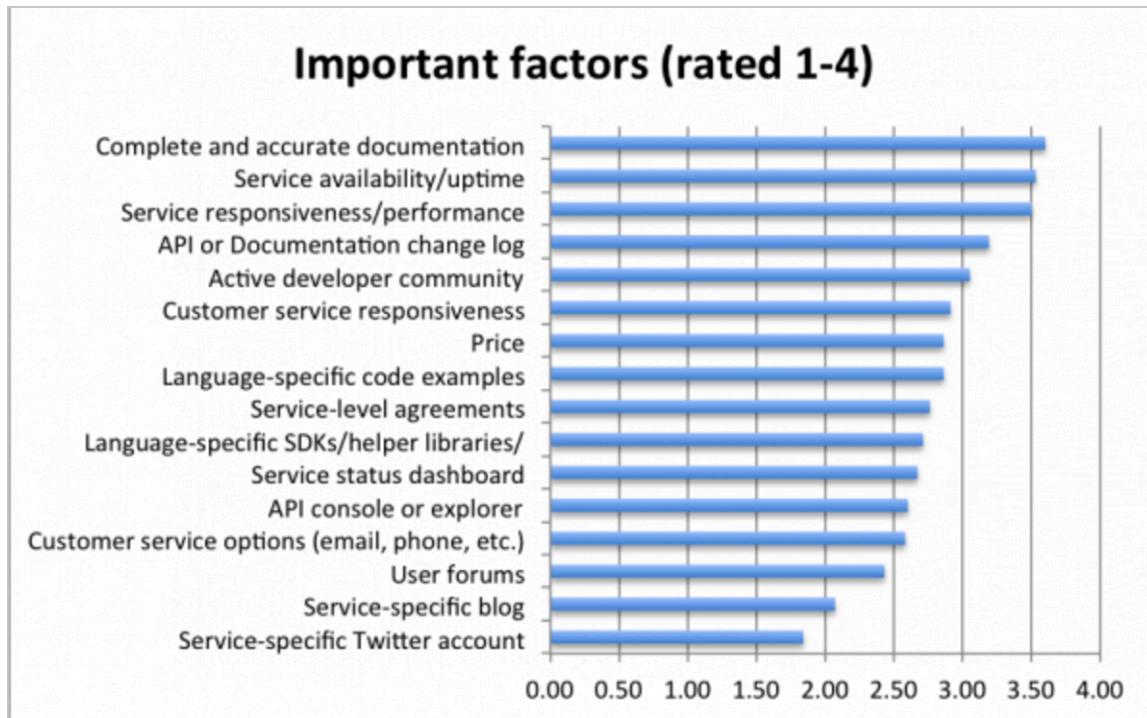
REST APIs follow the same protocol as the web. When you open a browser and type a website URL (such as <http://idratherbewriting.com>), you're actually making a GET request for a resource on a server. The server responds with the content and the browser makes the content visible.

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 \(page 415\)](#).)

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.

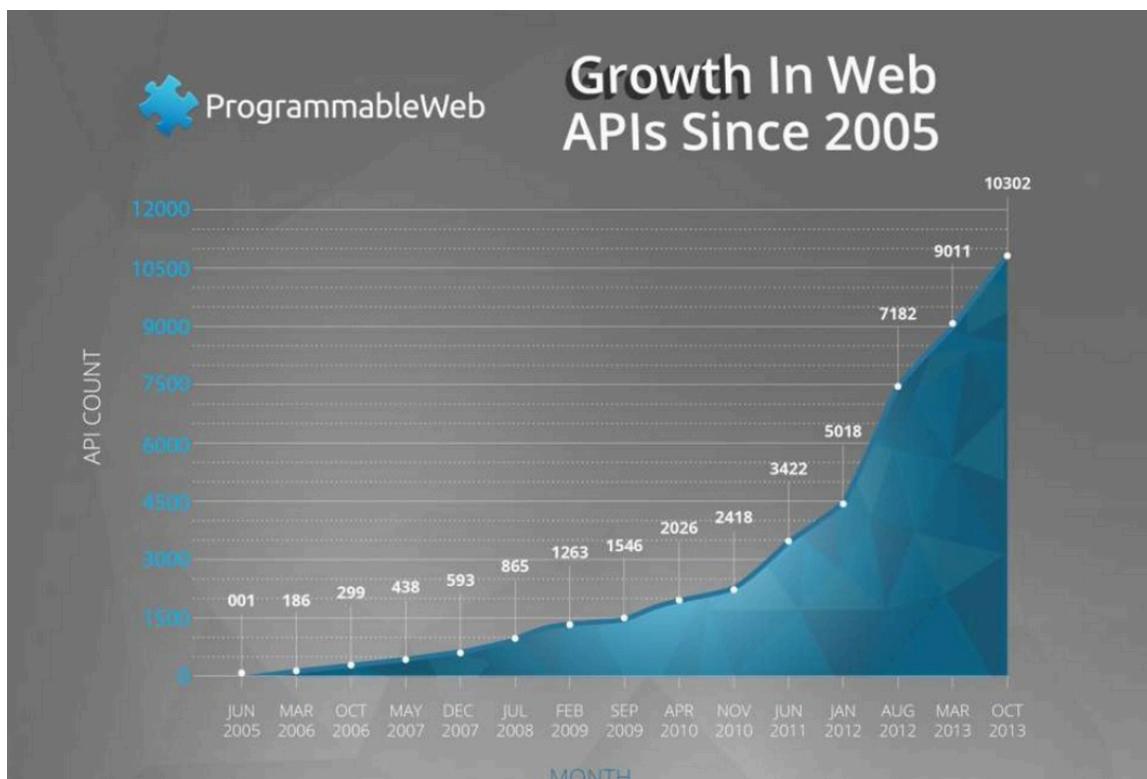
In a [2013 survey by Programmableweb.com](#) (which is a site that tracks and lists web APIs), about 250 developers were asked to rank the most important factors in an API. "Complete and accurate documentation" ranked as #1.



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. 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) enforced a specific message format for sending requests and returning responses. As an XML message format, SOAP was very specific and had a WSDL file (web service description language) that described 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 web page titled "Services". Below the title, a sub-section titled "Newsletters" is shown with the subtext "Capture email addresses and send periodic newsletters." Five service icons are displayed in a row: AWeber (blue icon), Campaign Monitor (white icon with blue envelope), MailChimp (icon of a smiling monkey wearing a cap), MailerLite (white icon with green "lite" text), and Sendicate (orange icon with a white "S" and wavy lines). Below this section is another titled "Analytics".

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 is a grid of three boxes: 'Read our docs' (describing language support 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 is one of its strengths. 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 \(page 0\)](#).)

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 did.

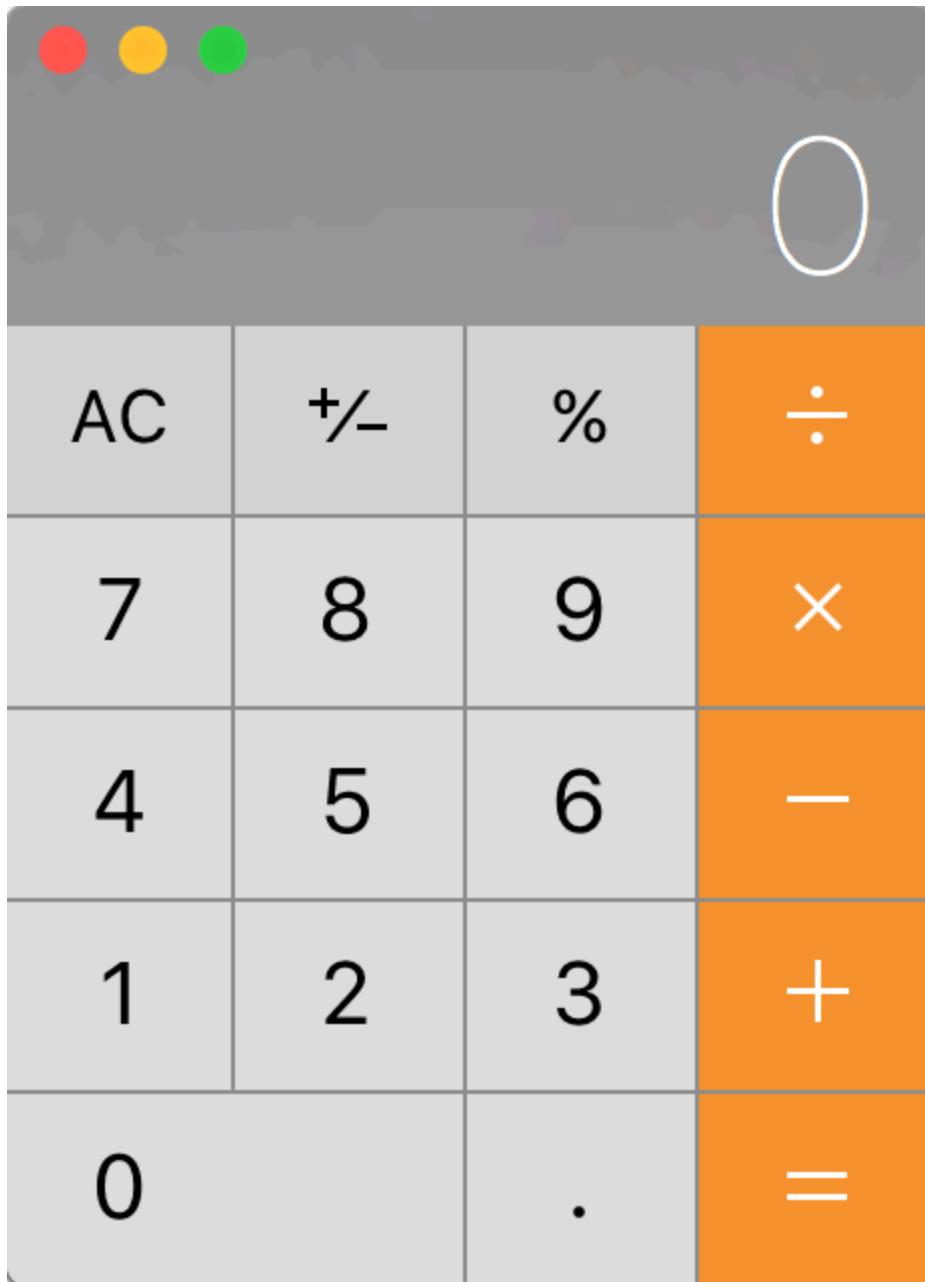
What is an API?

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 also 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 used 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 file (Web Services Description Language).

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.

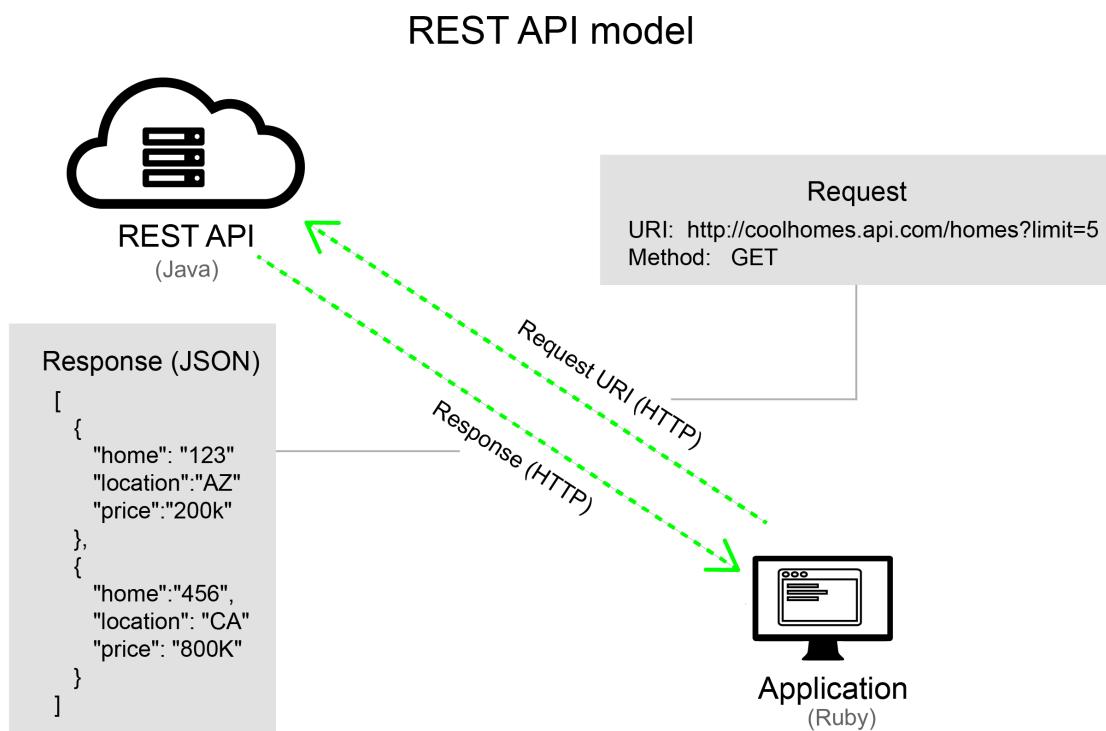
REST is a style, not a standard

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, we 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.

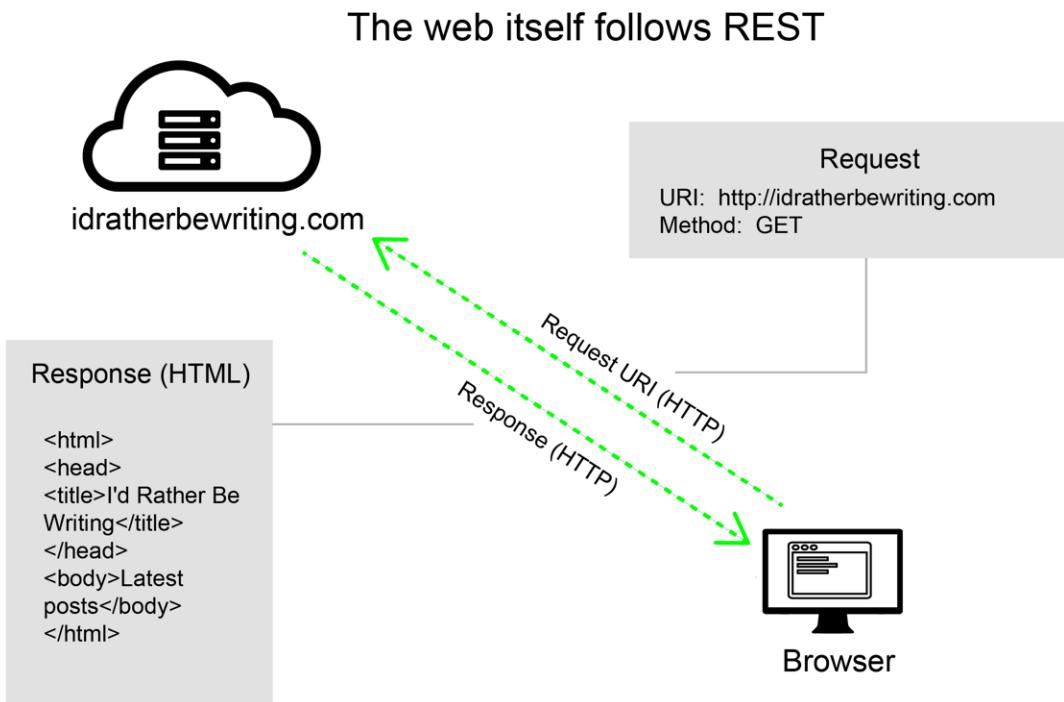
The relationship between resources and methods is often described in terms of “nouns” and “verbs.” The resource is the noun because it is an object or thing. The verb is what you're doing with that noun. Combining nouns with verbs is how you form the language in REST.

We'll explore endpoints in much more depth in the sections to come. 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 really 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 <http://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 if you open a terminal prompt and type `curl http://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 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 writers' role extremely important with REST APIs.

Some formal specifications — for example, such [OpenAPI \(page 298\)](#) and [RAML \(page 388\)](#) — have been developed to describe REST APIs. When you describe your API using the OpenAPI or RAML specification, tools that can read those specifications (like [Swagger UI \(page 355\)](#) or the [RAML API Console \(page 0\)](#)) 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 \(page 355\)](#) 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 \(page 170\)](#), 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 of the total needed documentation.

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: Find an Open Source Project

In this course, you'll do a variety of activities to build your skills, develop content, and gain experience. Many of the activities in this course will require you to work on documentation related to an open-source project that you find.

Without this project, it will be difficult to do many of the later activities in this course.

Reaching your larger goals

You're probably going through this course for one or more of the following reasons:

- You want to break into the field of API documentation.
- You want to develop your API doc skills for a future job.
- You have a new API to document at work and need information on how to do it.
- You're a developer who needs to document your API.

For the first two scenarios, you need to start thinking about API documentation samples in your portfolio. Your portfolio is key to [getting a job in API documentation \(page 449\)](#). Without a portfolio with compelling API documentation samples, it will be extremely difficult to get a job in API documentation.

How will you generate API doc samples for your portfolio, without having a job developing API documentation? 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 doc 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](#).)

Criteria for the open-source project

The ideal open-source API project should meet the following criteria. The project should:

- Involve a REST API (not a [library-based API \(page 415\)](#) 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.

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. See [GitHub Survey: Open Source Is Popular, Plagued by Poor Docs and Rude People](#). A [2017 GitHub Survey](#) found that

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](#) as well. The article also 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.

Enough pep talk, let's find a project that fits your needs.

Search for an open-source project with API doc needs

To find an open-source project with API doc needs:

1. Go to the [GitHub Advanced Search](#).
2. In the "With the labels" section, 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).
3. Scroll to the top and notice that `label: "help wanted"` automatically populates in the field. In this Advanced Search box at the top, add some additional keywords (such as `api`, `docs`, and

documentation) as well:

The screenshot shows the GitHub Advanced search interface. The search bar contains the query "api documentation docs|label:'help wanted'". Below the search bar, there are several filter options under "Advanced options": "From these owners" set to "github, joyent", "In these repositories" set to "twbs/bootstrap, rails/rails", "Created on the dates" set to ">YYYY-MM-DD, YYYY-MM-DD", and "Written in this language" set to "Any Language".

1. Click **Search** and browse [the results \(page 0\)](#).

Are there any projects that meet the [needed criteria \(page 31\)](#)? If so, great. If not, adjust some of the keywords and keep looking.

2. 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](#)

Programmableweb.com has the largest index of API documentation projects. Many of the projects may not need documentation nor provide open-source GitHub projects for working on the documentation. However, with an index of nearly 20,000 APIs, you might be able to find a project that might align with your interests, background, and other needs.

3. After selecting a project, make notes on the following:

- Identify how the project tags documentation related issues.
- Identify project members who worked on previous documentation issues (which are now closed).
- Identify any currently stated documentation needs.

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 \(page 268\)](#). 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 this now. You can learn these skills later when you have content you're ready to contribute.

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](#)

Workshop activities

The following are activities used in the [live workshops](#).

Part I: Intro to API documentation

Slides: [Intro to API documentation](#)

Activity: Explore an API

Explore the basic sections in API reference documentation in these 2 weather APIs:

- [OpenWeatherMap API](#).
- [Aeris Weather API](#)

Activity: Make a curl request

```
curl -I -X GET "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cu  
s&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

Activity: Postman client

1. Download [Postman](#).
2. Populate Postman from the Run in Postman button or from [this link](#):
3. Make requests in Postman.
4. Generate code snippets in JavaScript (AJAX) from Postman.
5. Create a Run in Postman button for your requests. In the Collections pane, click the < arrow to expand the pane and click **Share**.

Part II: OpenAPI and Swagger

Slides: [OpenAPI and Swagger](#)

Activity: OpenAPI with Stoplight

1. Open [v3 next Stoplight app](#). (You can also use the [Desktop app](#))
2. From main.oas, open the Code tab and paste in content for this 2.0 JSON Open API definition:
http://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_weather20.json
(page 0)
3. Edit, explore Basics, Requests, Responses sections.
4. In the Responses area, click **Generate from JSON**, paste in complex JSON snippet into the Responses area, then click **Generate!**

```
{  
  "query": {  
    "count": 1,  
    "created": "6/14/2017 2:30:14 PM",  
    "lang": "en-US",  
    "results": {  
      "channel": {  
        "units": {  
          "distance": "km",  
          "pressure": "mb",  
          "speed": "km/h",  
          "temperature": "C"  
        },  
        "title": "Yahoo! Weather - Singapore, South West, SG",  
        "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Country__Country/*https://weather.yahoo.com/country/state/city-91792352/",  
        "description": "Yahoo! Weather for Singapore, South West, SG",  
        "language": "en-us",  
        "lastBuildDate": "Wed, 14 Jun 2017 10:30 PM SGT",  
        "ttl": 60,  
        "location": {  
          "city": "Singapore",  
          "country": "Singapore",  
          "region": "South West"  
        },  
        "wind": {  
          "chill": 81,  
          "direction": 158,  
          "speed": 0  
        },  
        "atmosphere": {  
          "humidity": 87,  
          "pressure": 0,  
          "rising": 0,  
          "visibility": 0  
        },  
        "astronomy": {  
          "sunrise": "6:59 am",  
          "sunset": "7:11 pm"  
        },  
        "image": {  
          "title": "Yahoo! Weather",  
          "width": 142,  
          "height": 18,  
          "link": "http://weather.yahoo.com",  
          "url": "http://l.yimg.com/a/i/brand/purplelogo//uh/us/news-w  
ea.gif"  
        },  
        "item": {  
          "title": "Conditions for Singapore, South West, SG at 09:00  
PM SGT",  
        }  
      }  
    }  
  }  
}
```

```
    "lat": 1.33464,
    "long": 103.726471,
    "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Country__Country/*https://weather.yahoo.com/country/state/city-91792352/",
    "pubDate": "Wed, 14 Jun 2017 09:00 PM SGT",
    "condition": {
        "code": 33,
        "date": "Wed, 14 Jun 2017 09:00 PM SGT",
        "temp": 26,
        "text": "Mostly Clear"
    },
    "forecast": [
        {
            "code": 4,
            "date": "14 Jun 2017",
            "day": "Wed",
            "high": 28,
            "low": 25,
            "text": "Thunderstorms"
        }
    ],
    "description": "<![CDATA[<img src=\"http://l.yimg.com/a/i/us/we/52/4.gif\"/>\n<BR />\n<b>Current Conditions:</b>\n<BR />Thunderstorms\n<BR />\n<b>Forecast:</b>\n<BR /> Fri - Thunderstorms.\nHigh: 30Low: 25\n<BR /> Sat - Thunderstorms. High: 28Low: 25\n<BR /> Sun - Thunderstorms. High: 28Low: 25\n<BR /> Mon - Thunderstorms. High: 28Low: 25\n<BR /> Tue - Thunderstorms. High: 28Low: 25\n<BR />\n<a href=\"http://us.rd.yahoo.com/dailynews/rss/weather/Country__Country/*https://weather.yahoo.com/country/state/city-91792352/\">Full Forecast at Yahoo! Weather</a>\n<BR />\n(provided by <a href=\"http://www.weather.com\">The Weather Channel</a>)\n<BR />\n]]>",
        "guid": "string"
    }
}
}
```

Activity: Swagger Editor

1. Paste [this YAML file \(page 0\)](#) into [Swagger Editor](#) and make updates.
2. Go to [this SwaggerHub API](#). Observe Generate Client SDK options.

Activity: Swagger UI

1. Download [Swagger UI](#).
2. Uncompress and pull out the **dist** folder.
3. Save this file locally: [http://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_weather.yml \(page 0\)](http://idratherbewriting.com/learnapidoc/docs/rest_api_specifications/openapi_weather.yml (page 0)) into the **dist** folder.
4. Reference **openapi_weather.yml** in place of the default **url** value.

5. Open in Firefox.

Part III: Non-reference content in API docs

Slides: [Non-reference content in API docs](#)

Activity: GitHub workflow

1. Create new repo and initialize with readme. Clone repo locally using `git clone`.
2. Make update to readme file and push back into repo:

```
git add .
git commit -m "made update to readme"
git pull
git push
```

Using a REST API like a developer

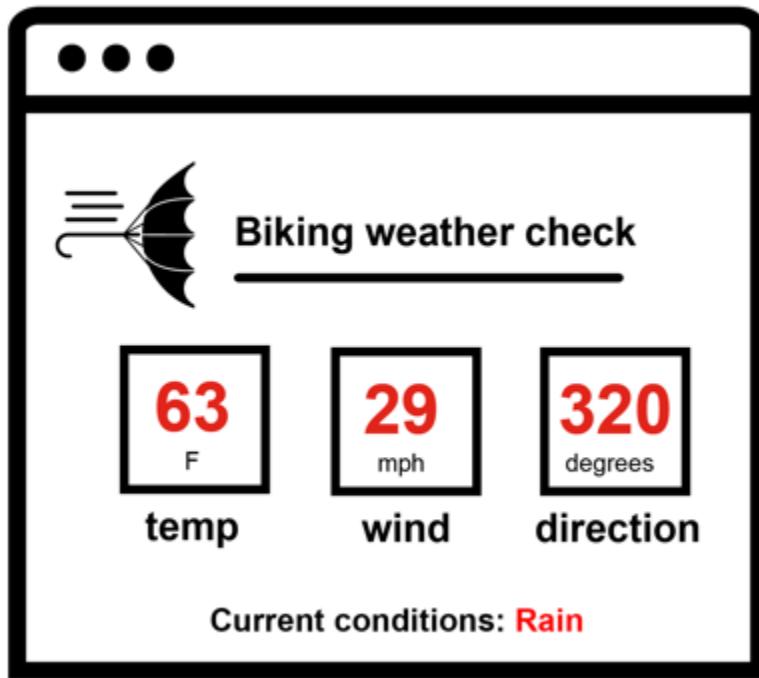
Scenario for using a weather API	40
Get authorization keys.....	44
Submit requests through Postman.....	46
curl intro and installation	52
Make a curl call.....	54
Understand curl more.....	56
Use methods with curl.....	60
Analyze the JSON response.....	65
Use the JSON from the response payload.....	69
Access and print a specific JSON value	74
Dive into dot notation	78

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: [wind-weatherbit.html \(page 0\)](#). 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

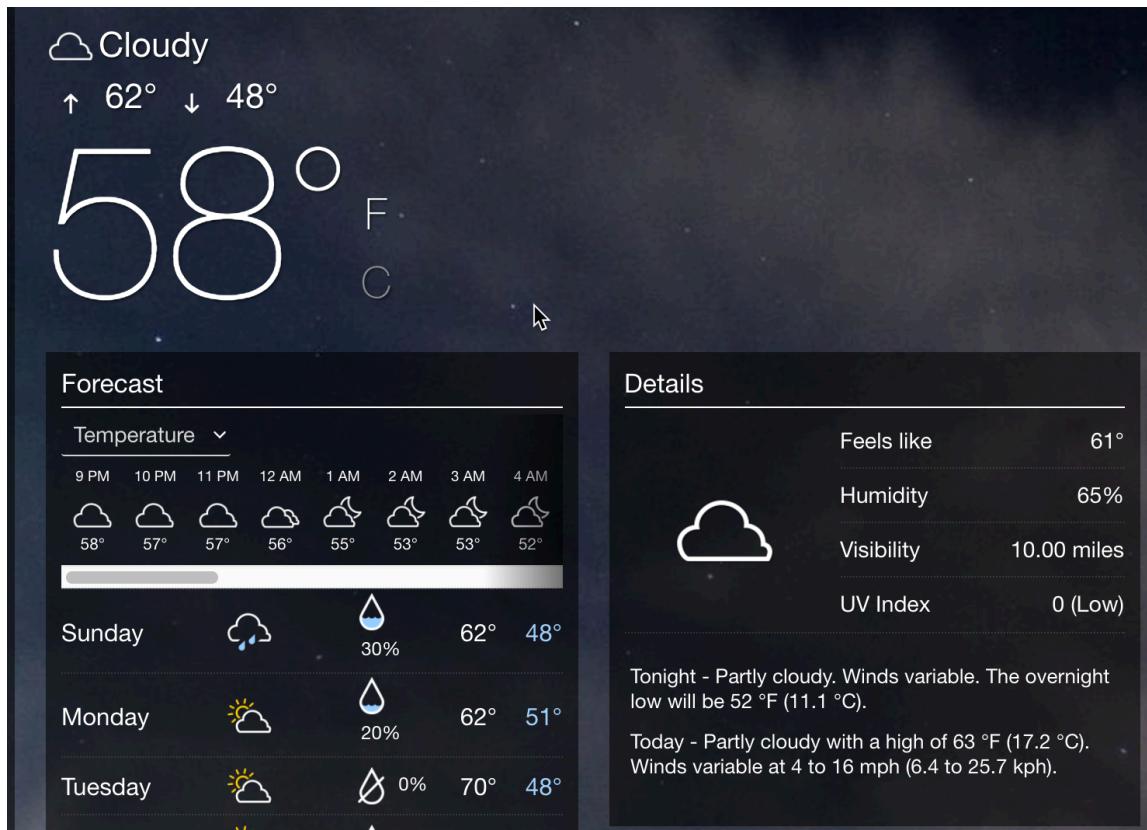
Temperature:

Wind speed:

Wind direction:

Current conditions:

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



The concept and general techniques are more or less the same.

Explore the OpenWeatherMap API

Let's find a simple weather API that we can use for some exercises. There are [many good weather API options for developers](#). Let's use [OpenWeatherMap](#), because their service is easy to use, free, and stable.

ACTIVITY



Explore the information available on OpenWeatherMap:

1. Go to <https://openweathermap.org> and click **API**.
2. Explore the information available in the **Current Weather Data** by clicking **API Doc** in that section.

As you explore the site, get a sense of the variety and services that 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. As you explore the Current Weather Data API, see if it contains information about temperature and wind conditions relevant to our coding scenario.

Explore the Aeris Weather API

Now 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 **Developer** on the top navigation.
3. Under **Aeris Weather API**, click **Documentation**.
4. Under **Reference** in the left sidebar, click **Endpoints**.

The screenshot shows the Aeris Weather API documentation. The top navigation bar includes links for APPS, DEVELOPER, INDUSTRIES, SUPPORT, and CASE STUDIES. Below the navigation is a breadcrumb trail: HELP CENTER / DOCS / AERIS WEATHER API / REFERENCE / ENDPOINTS. On the left, there's a sidebar with a logo and links for Aeris Weather API, Getting Started, Reference, and Downloads. The main content area has a title 'API Endpoints'. It says: 'Endpoints refer to the types of data to request, such as a place, observation, forecast or advisory, etc. Any request made to the API. The following endpoints are currently supported within Aeris.' Below this is a note: 'To query multiple endpoints with a single API request review the [Batch Requests](#) feature.' The 'advisories' endpoint is detailed, showing its description and data coverage (Continental US, Alaska, Hawaii). The 'advisories/summary' endpoint is also mentioned.

5. In the list of endpoints, click **observations**.
6. Browse the type of information that is available through this endpoint.

Here's the Aeris weather forecast API in action making mostly the same calls as I showed earlier with OpenWeatherMap: </learnapidoc/assets/files/wind-aeris.html>.

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 \(page 27\)](#), REST APIs are an architectural style following common characteristics and principles; they don't all follow the same standard or specification. You really have to read the documentation to understand how to use the APIs.

Answer some questions about the APIs

ACTIVITY



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.

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

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 \(page 170\)](#) in more detail.

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 personal passwords and not given out or published openly on a web page. (Even so, if you want to borrow my API keys, you can view them [here \(page 0\)](#).)

Get an OpenWeatherMap API key

To get an authorization key to use the OpenWeatherMap API:

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 API key from the developer dashboard.
3. Copy the key into a place you can easily find it.

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 a weather icon (57°), a search bar ('Weather for...'), and account links ('ACCOUNT', 'LOG OUT'). Below the header is the Aeris Weather logo. The main navigation menu includes 'DASHBOARD', 'USAGE', 'APPS' (which is underlined, indicating it's the active page), 'MAP BUILDER', 'LEGEND GENERATOR', 'ADD SUBSCRIPTION', 'DOCS', 'SUPPORT', and 'BILLING'. Under the 'APPS' menu, there's a 'PROFILE' link. The main content area is titled 'Registered Apps'. It displays a list of registered applications, starting with 'API testing'. Each application entry includes fields for 'ID' (ByruDorHEne2JB64BhP1k) and 'SECRET' (uBDNO535gYHULH8mqx3skZmU13EV4nlf4GvB6jhY), along with edit and delete icons. A red arrow points to a 'NEW APPLICATION' button in the top right corner of the app list area.

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 text file, since you'll need it to make requests.

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 usingTextEdit 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. You can also use [curl \(page 52\)](#) (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

Using 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 things now, like what a GET method is or a endpoint, hang in there. When we deep dive into these points later, things will be a lot clearer.

Make a request in Postman

In this exercise, you'll make a REST call using OpenWeatherMap's [current weather data API endpoint](#).

1. If you haven't already done so, download and install the Postman app at <http://www.getpostman.com>. If you're on a Mac, choose the Mac app. If you're on Windows, choose the Windows app.
2. Start the Postman app.
3. Select **GET** for the method. (This is the default.)
4. Insert the endpoint into the box next to GET: `http://api.openweathermap.org/data/2.5/weather`.
5. Click the **Params** button (to the right of the box where you inserted the endpoint) 3 parameters:
 - `zip: 95050`
 - `units: imperial`
 - `appid: {your api key}`

Customize the `zip` and `appid` values to your own zip code and [OpenWeatherMap API key \(page 0\)](#). Your Postman should look like this:

The screenshot shows the Postman interface with a GET request to `http://api.openweathermap.org/data/2.5/weather`. The parameters section contains three entries: `zip` with value `95050`, `appid` with value `fd4698c940c6d1da602a70ac34f0b147`, and `units` with value `imperial`. The `key` column lists the parameter names, and the `value` column lists their corresponding values. Below the parameters are tabs for Authorization, Headers, Body, Pre-request Script, Tests, Cookies, and Code. The Authorization tab is selected, showing "No Auth".

When you add these parameters, they will dynamically be added as a query string to the endpoint URL. For example, your endpoint will now look like this:

`http://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 (below the GET button) and insert the required key-value pairs in the header.

6. Click **Send**.

The response appears in the lower window. For example:

The screenshot shows the Postman response window with the **Body** tab selected. The status bar indicates `Status: 200 OK Time: 237 ms Size: 780 B`. The response body is a JSON object:

```

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  }

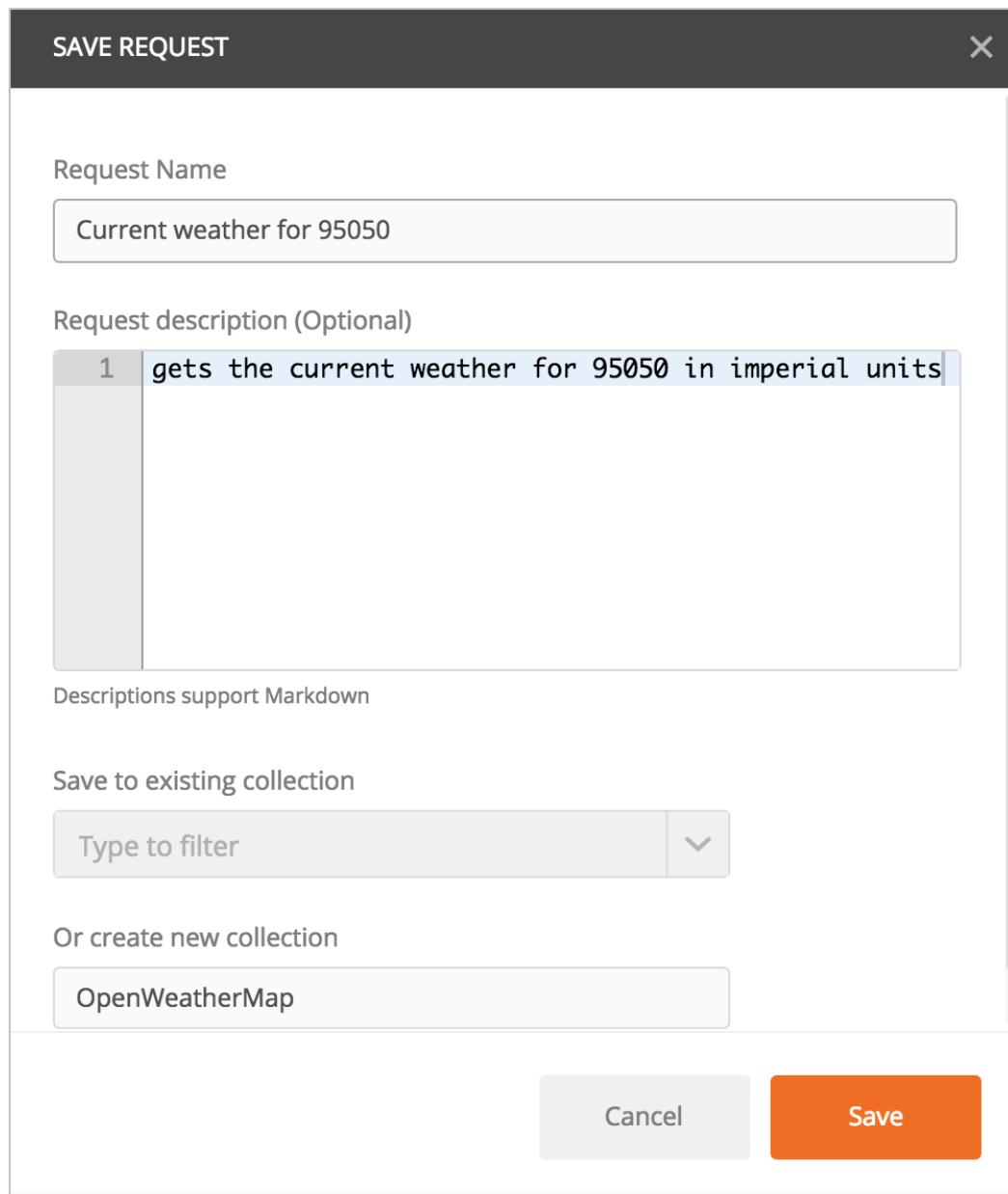
```

Save the request

- In Postman, click the **Save** button (next to Send).

2. In the Save Request dialog box, in the **Request Name** box at the top of the dialog box, type a friendly name for the request, such as “Current weather for 95050.”
3. 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.

Make a request for the OpenWeatherMap 5 day forecast

Enter details into Postman for the [5 day forecast](#). You can click a new tab, or click the arrow next to Save and choose **Save As**. Then choose your collection and request name.

The 5 day forecast request looks like this:

```
http://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

For the sake of variety with GUI clients, here's the same call made in [Paw \(for Mac\)](#):

The screenshot shows the Paw application window. On the left, there's a sidebar with 'Sessions' and 'Environments'. A main panel shows a 'Request' for 'GET /data/2.5/weather'. Below it, a table lists 'URL Parameter' and 'Value' for 'zip', 'appid', and 'units'. To the right, the 'Response' pane shows a JSON tree with expanded nodes for 'coord', 'weather', 'Index 0', 'main', and 'clouds', along with numerical values for coordinates and weather parameters. At the bottom, a terminal-like interface shows the raw HTTP request and response.

```
1 GET /data/2.5/weather?
2 zip=95050,us&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial
3 Host: api.openweathermap.org
4 Connection: close
5 User-Agent: Paw/3.1.5 (Macintosh; OS X/10.12.6) GCDHTTPRequest
6
7
8
```

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 \(page 40\)](#). 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 requested them in [Get the authorization keys \(page 44\)](#)). If you don't have a client ID or secret, you can use my keys [here \(page 0\)](#).

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.

OpenWeatherMap API collection

If this button doesn't work for you, copy this [import link](#).

Aeris Weather API collection.

If this button doesn't work for you, copy this [import link](#).

Clicking the Run in Postman buttons should automatically prompt you to import the content 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](#), focusing on the documentation features within Postman.

curl intro and installation

While Postman 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 it's 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: AsynchDNS GSS-Negotiate IPv6 Largefile NTLM NTLM_WB SSL libz
```

Make a test API call

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

```
curl -X GET "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&a  
ppid=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

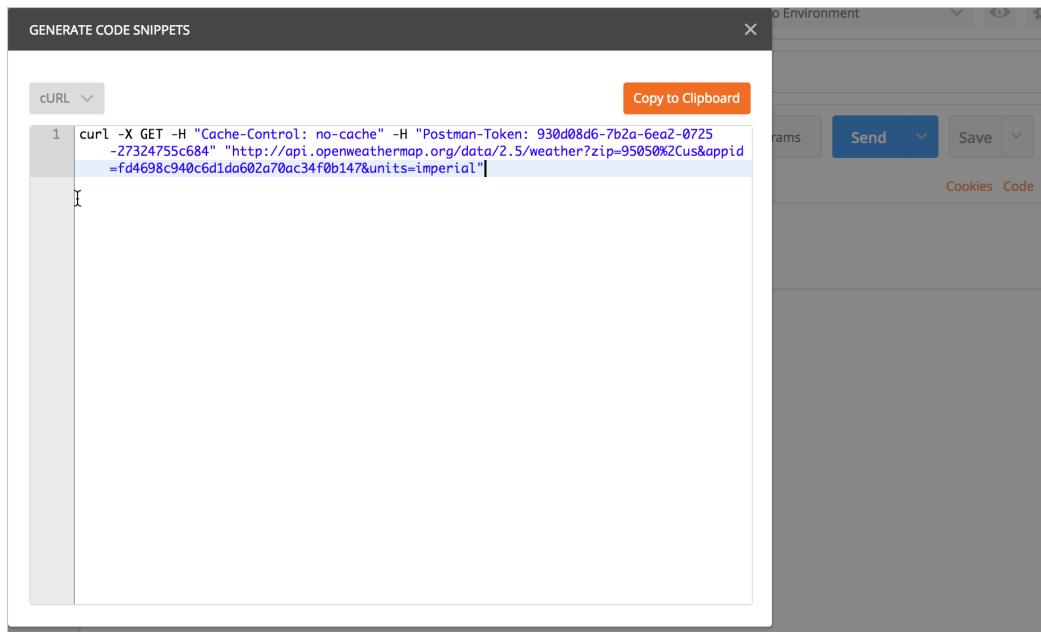
- Use double quotes in the Windows command line. (Windows doesn't support single quotes.)
- Don't use backslashes (\) to separate lines. (This is for readability only and doesn't affect the call on Macs.)
- By adding `-k` in the curl command, you can bypass curl's security certificate, which may or may 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 \(page 52\)](#) first.

Prepare the weather request in curl format

1. Assuming you completed the exercises in the [Postman tutorial \(page 46\)](#), go back into Postman.
2. On any call you've configured, right 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 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" "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

For whatever reason, Postman adds some of its own header information (designated with `-H`). Remove these extra header tags.

```
curl -X GET "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

3. If you're on a Mac, open 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.

Mac users: Instead of using Terminal, download and use [iTerm](#) instead. It will give you the ability to open multiple tabs, save profiles, and more.

4. 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.

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.

Make curl requests for each of the Postman requests

ACTIVITY



Make a curl request for each of the requests you entered 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.

REST APIs follow the same model of the web

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.

Try using curl to GET a web page

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.

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

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

The response header is as follows:

```
HTTP/1.1 200 OK
Content-Encoding: gzip
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Date: Sat, 25 Mar 2017 16:24:59 GMT
Etag: "359670651"
Expires: Sat, 01 Apr 2017 16:24:59 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (rhv/81A7)
X-Cache: HIT
Content-Length: 606
```

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.

- 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 \(page 54\)](#):

```
curl -X GET -H "Cache-Control: no-cache" -H "Postman-Token: 930d08d6-7b2a-6ea2-0725-27324755c684" "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&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`. (`GET` is the default, so if it's not specified, it's automatically used. 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%2Cus&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>	Include the response headers in the response.	<code>curl -i http://www.example.com</code>
<code>-d</code> or <code>--data</code>	Include 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>	Submit the request header to the resource. This is very common with REST API requests because the authorization is usually included here.	<code>curl -H "key:12345" http://www.example.com</code>

curl command	Description	Example
<code>-X POST</code>	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 GET is the default method used.	<code>curl -X POST -d "resource-to-update" http://www.example.com</code>
<code>@filename</code>	Load 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.

ACTIVITY



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 \(page 40\)](#) 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 \(page 298\)](#)). 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.

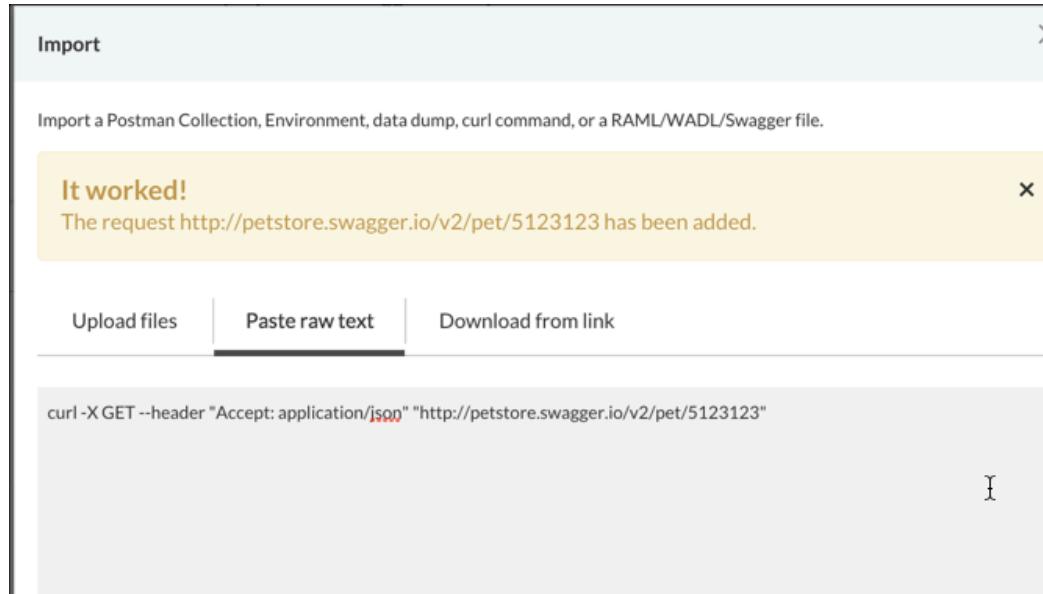
Although Postman is probably easier to use, curl lends itself to power-level usage. Quality assurance teams often construct advanced test scenarios that iterate through a lot of curl requests.

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 export Postman to curl by doing the following:

1. In Postman, click the **Generate Code** button.

Generating code snippets

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 (except for the last pair).

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.

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

```
[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 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.

ACTIVITY



Look at the response from the `weather` endpoint of the OpenWeatherMap weather API. Where are the objects? Where are the arrays?

It's common for arrays to contain lists of objects, and for objects to contain arrays.

More information

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

Use 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.

Display part of the REST JSON response on a web page

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

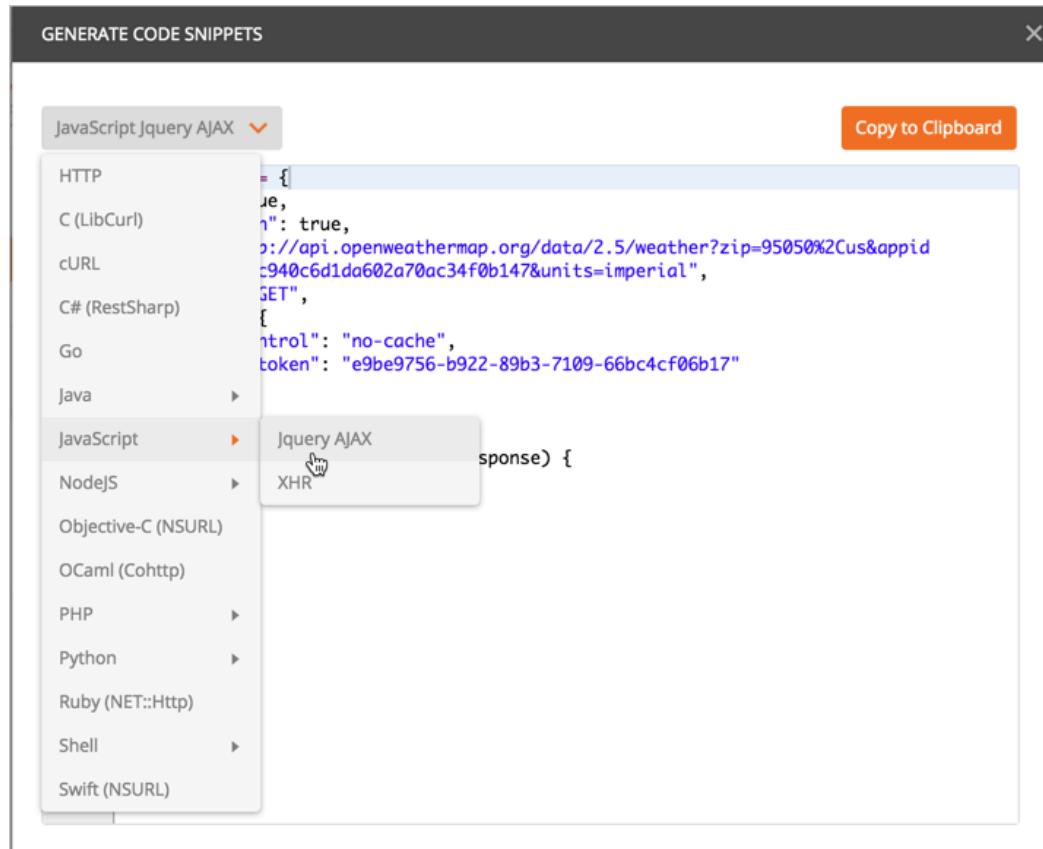
1. Start with a basic HTML template with jQuery referenced, like this:

```
<html>
<meta charset="UTF-8">
<head>
    <title>Sample page</title>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.1
1.1/jquery.min.js"></script>
</head>
<body>
    <h1>Sample page</h2>

</body>
</html>
```

Save your file with a name such as weather.html.

2. Open Postman and click the request to the `weather` endpoint that you [configured earlier](#) (page 46).
3. Click the **Code** link (below the Save button), and then select **JavaScript > jQuery AJAX**.



4. Copy **Copy to Clipboard** to copy the code sample.

```
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&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. In the same template you started building in step 1, add a pair of `<script></script>` tags below the jQuery script, and then insert the Postman code inside the new `script` tags.

You can put the script in the `head` section if you want — just make sure you add it after the jQuery reference.

6. In the jQuery code, remove the headers that Postman inserts:

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

Your final code should look like this:

```
<!DOCTYPE html>  
<html>  
  <meta charset="UTF-8">  
  <head>  
    <meta charset="UTF-8">  
    <script src="http://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": "http://api.openweathermap.org/data/2.5/weather?zip=95050%2C  
us&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

7. Start **Firefox** and open the Web Console by going to **Tools > Web Developer > Web Console**. (Normally, I would recommend using Chrome. But Chrome's security settings block JavaScript on local files, so use Firefox instead.)
8. Open the **weather.html** file in the browser by going to **File > Open File**.

The page body will be blank, but the weather response should be logged to the web console. You can inspect the payload by expanding the object. Here's what it looks like expanded in Firefox:

The screenshot shows the Firefox Developer Tools interface with the 'Console' tab selected. A JSON object is expanded, revealing its structure. The object includes properties like 'base', 'clouds', 'cod', 'coord', 'dt', 'id', 'main', 'name', 'sys', 'visibility', 'weather', 'wind', and '__proto__'. The 'name' property is specifically highlighted with the value 'Santa Clara'.

```
{
  base: "stations",
  clouds: Object { all: 1 },
  cod: 200,
  coord: Object { lon: -121.96, lat: 37.35 },
  dt: 1522622100,
  id: 420006397,
  main: Object { temp: 69.76, pressure: 1012, humidity: 53, ... },
  name: "Santa Clara",
  sys: Object { type: 1, id: 399, message: 0.0053, ... },
  visibility: 16093,
  weather: Array [ ... ],
  wind: Object { speed: 19.46, deg: 250 },
  __proto__: Object { ... }
}
```

If you want to use Chrome instead, go to the web location the file [here](#) and open the JS Console by going to **View > Developer > JavaScript Console**. If you expand the object returned to the console, it will look as follows:

The screenshot shows the Google Chrome DevTools interface with the 'Console' tab selected. A JSON object is expanded, showing the same structure as the Firefox example. The 'name' property is again highlighted with the value 'Santa Clara'. The browser's developer tools also show tabs for Elements, Sources, Network, Performance, Memory, Application, and Security.

```
{
  coord: {...}, weather: Array(1), base: "stations", main: {...}, visibility: 16093, ...
  base: "stations",
  clouds: {all: 1},
  cod: 200,
  coord: {lon: -121.96, lat: 37.35},
  dt: 1522622100,
  id: 420006397,
  main: {temp: 69.76, pressure: 1012, humidity: 53, temp_min: 62.6, temp_max: 75.2},
  name: "Santa Clara",
  sys: {type: 1, id: 399, message: 0.168, country: "US", sunrise: 1522590706, ...},
  visibility: 16093,
  weather: [{}],
  wind: {speed: 19.46, deg: 250},
  __proto__: Object
}
```

Note that Chrome tells you whether each expandable section is an object or an array. Knowing this is important to accessing the value through JavaScript dot notation.

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

The AJAX method from jQuery

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. 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`. In the preceding 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.

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.

ACTIVITY



Inspect [the payload](#) by expanding each of the sections returned in the JSON console object. Find the section that appears here: `object > query > results > channel > item > description`. Based on the information here, what's the forecast for today?

In the next section, [Access and print a specific JSON value \(page 74\)](#), we'll pull out this value and print it to the page.

Access and print a specific JSON value

This tutorial continues from the previous topic, [Use the JSON from the response payload \(page 69\)](#). In the [sample page \(page 0\)](#) 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 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.

This section will use a little bit of JavaScript. You probably wouldn't use this code very much for documentation, but it's important to know anyway.

Getting a specific property from a JSON response 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
```

JSON wouldn't be very useful if you had to always print out the entire response. Instead, you select the exact element 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.

To pull out the wind speed element from the JSON response and display it on the page, add this to your code sample (which you created in the [previous tutorial \(page 65\)](#)), 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. This involves a little bit of JavaScript or jQuery (to make it easier).

I'm assuming you're starting with the [same code \(page 0\)](#) from the [previous tutorial — Use the JSON from the response payload \(page 69\)](#). That code looks like this:

```
<html>
<meta charset="UTF-8">
<head>
    <meta charset="UTF-8">
<script src="http://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": "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&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:

```
<html>
<meta charset="UTF-8">
<head>
    <meta charset="UTF-8">
<script src="http://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": "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&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: </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.

Dive into dot notation

In the previous topic, [Access and print a specific JSON value \(page 74\)](#), 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

ACTIVITY



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="http://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: <http://idratherbewriting.com/learnapidoc/assets/files/dot-notation-practice.html> (page 0). 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 \(page 40\)](#), I showed an example of [embedding the wind speed \(page 40\)](#) 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 page:

```
<html>
<head>
<script src="http://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": "http://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="mai
```

```
n_temp_unit"></span><br/>
<span><b>Wind speed: </b></span><span id="wind_speed"></span> <span id="win
d_speed_unit"></span><br/>
<span><b>Wind direction: </b></span><span id="wind_direction"></span><span i
d="wind_degree_unit"></span><br/>
<span><b>Current conditions: </b></span><span id="weather_conditions"></spa
n>
</body>
</html>
```

A few things are different here, but it's essentially the same code:

- 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:

The screenshot shows a web page with a red button labeled "Check wind conditions". Below the button, the title "Wind conditions for Santa Clara" is displayed in large, bold, dark gray font. Underneath the title, there are three lines of text in red, each preceded by a bold label: "Wind chill: 82 C", "Wind speed: 11.27 km/h", and "Wind direction: 295 degrees".

You can view the file idratherbewriting.com/learnapidoc/assets/files/wind-weatherbit.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 \(page 83\)](#), you'll assume the task of [documenting a new endpoint \(page 84\)](#) that was added to the weather API. You'll learn the 8 essential sections in endpoint reference documentation, the terminology to use, and formatting conventions for the reference information.

Documenting API reference topics

A new endpoint to document	84
API reference tutorial overview	87
Step 1: Resource description (API ref tutorial)	89
Step 2: Endpoints and methods (API ref tutorial)	95
Step 3: Parameters (API ref tutorial)	101
Step 4: Request example (API ref tutorial).....	110
Step 5: Response example and schema (API ref tutorial)	121
Putting it all together	137
Activity: Critique or create an API reference topic	142

A new endpoint to document

Until this point, you've been [acting as a developer \(page 39\)](#) 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:

```
http://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 \(page 87\)](#) 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.

For context, we'll structure the information from the [sample new endpoint to document \(page 84\)](#) into these five sections.

Five common sections in REST API docs

Almost all API reference topics include these five sections:

[1. Resource description \(page 89\)](#)

"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.

[2. Endpoints and methods \(page 95\)](#)

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 variant information about the same resource. Endpoints usually have brief descriptions similar to the overall resource description but shorter.

[3. Parameters \(page 101\)](#)

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. Not all endpoints contain each type of parameter.

[4. Request example \(page 110\)](#)

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.

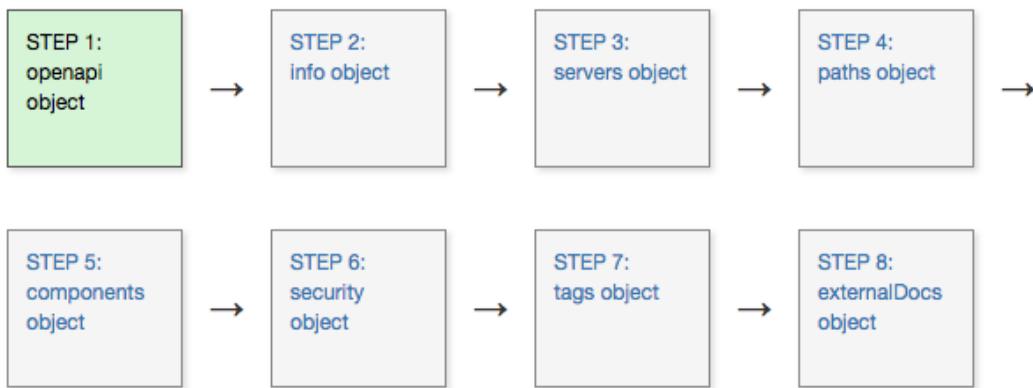
[5. Response example and schema \(page 121\)](#)

The response example shows a sample response from the request example. 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.

Tutorial workflow map

The topics include a workflow map to help guide and orient you each step of the way.



(page 89)

After the tutorial

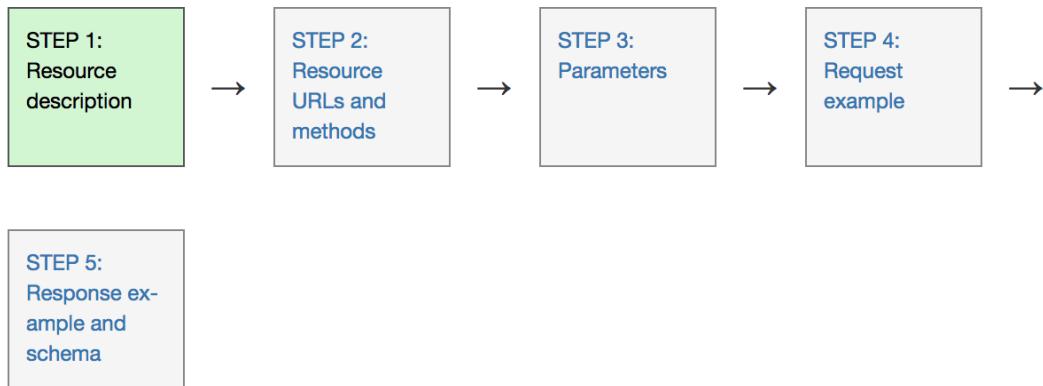
When we're finished, the end result will look [like this \(page 137\)](#). You'll then have an opportunity to [create an API reference topic \(page 142\)](#) with your own [open-source API project \(page 31\)](#).

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 \(page 191\)](#), 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 \(page 89\)](#).

Step 1: Resource description (API reference tutorial)



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.

Example of a resource description

Here's an example of a resource description from the Mailchimp API's [Campaigns resource](#):

The screenshot shows the MailChimp API documentation for the 'Campaigns' resource. At the top, there's a navigation bar with links for 'Developer', 'Documentation', 'Playground', and 'API Status'. Below the navigation, 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' (which is currently selected). To the right of the sidebar, the main content area has a title 'Campaigns' and a yellow callout box stating: 'Campaigns are how you send emails to your MailChimp list. Use the Campaigns API calls to manage campaigns in your MailChimp account.' Below this, there's a section titled 'Subresources' with three links: 'Content', 'Feedback', and 'Send Checklist'. The 'Available methods' section lists five actions: Create, Read, Edit, Delete, and Action. Under 'Read', there are two entries: a blue button labeled 'GET /campaigns' with the text 'Get all campaigns' and another blue button labeled 'GET /campaigns/{campaign_id}' with the text 'Get information about a specific campaign'.

Typically, an API will have a number of endpoints grouped under the same resource. Normally, 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 detailed section for Membership Object. The Membership Object section is highlighted with a yellow background and lists endpoints: GET Get Membership, POST Create Membership, PUT Update Membership, DELETE Delete Membership, GET Get Memberships for Group, GET Get Memberships for User, and GET Get Collaborations for Group. To the right, the main content area has a title "Membership Object" and a "SUGGEST EDITS" button. A yellow callout box contains the text: "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, a table details the fields of a 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. Below this is a search bar for documentation and a breadcrumb trail: Eventbrite APIv3 Documentation > Events. The main content area has a title "Events" with a subtitle "1". It shows a "GET /events/search/" endpoint. A yellow callout box contains the text: "Allows you to retrieve a paginated response of public event objects from across Eventbrite's directory, regardless of which user owns the event." Below this is a "Parameters" table:

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 several 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 is shown for the first endpoint: "GET /users/self/follows". It includes the URL (`https://api.instagram.com/v1/users/self/follows?access_token=ACCESS-TOKEN`), a "RESPONSE" dropdown, a description ("Get the list of users this user follows."), requirements ("Scope: follower_list"), parameters ("ACCESS_TOKEN: A valid access token."), and a note ("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 \(page 188\)](#).

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.

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

ACTIVITY



Let's review the [surf report wiki page \(page 84\)](#) (which contains the information about the resource) and try to describe the resource in 1-3 sentences. Here's my approach:

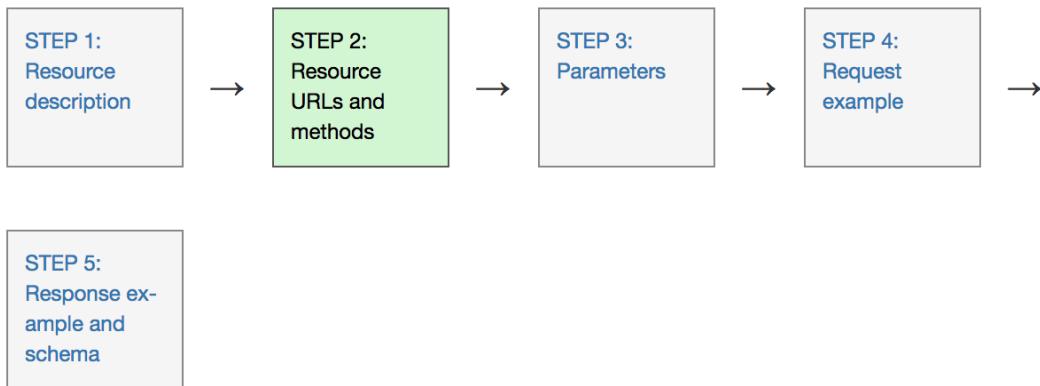
Surfreport

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 \(page 95\)](#) for the resource.

Step 2: Endpoints and methods (API reference tutorial)



Endpoints and methods: 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 variant 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 lists various API endpoints under categories like Overview, Authentication, Login Permissions, etc. The main content area shows two specific endpoints:

- GET /users/self/follows**: Description: Get the list of users this user follows. Requirements: Scope: follower_list. Parameters: ACCESS_TOKEN (A valid access token).
- GET /users/self/followed-by**: Description: Get the list of users this user is followed by. Requirements: Scope: follower_list. Parameters: ACCESS_TOKEN (A valid access token).

The endpoint is usually set off in a stylized way that gives it more visual attention. Much of the documentation is built around the endpoint, so it might make sense to give each endpoint more visual weight in your documentation.

The endpoint is arguably the most important aspect of API documentation, since this is what developers will implement to make their requests.

Represent path parameters with curly braces

If you have path parameters 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 \(page 101\)](#) 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 endpoints under 'Graph API'. 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'. There are also tabs for 'HTTP', 'PHP SDK', 'JavaScript SDK', 'Android SDK', 'iOS SDK', and 'Graph API Explorer'.

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: Retrieve a resource
- POST: Create a resource
- PUT: Update or create within an existing resource
- PATCH: Partially modify an existing resource
- DELETE: Remove 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 'Data Formats' section with a 'Requesting data from the APIs' heading. It states that unless otherwise specified, all LinkedIn APIs return XML data format. Below this is a yellow box for a 'GET' request to `https://api.linkedin.com/v1/people/~`. At the bottom, there's a 'sample response' box containing XML code for a person named Frodo Baggins.

```

<?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>
</person>
  
```

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 `http://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 \(page 170\)](#), 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 \(page 89\)](#), 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 had three GET endpoints and one POST endpoint, all of which are related 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 pretty 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 \(page 199\)](#), I explore how [long pages \(page 205\)](#) are common pattern with developer docs, in part because they make content easily findable for developers using Ctrl + F.

How to refer to endpoints

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:

```
// get all rewards  
/rewards  
  
// get a specific reward  
/rewards/{rewardId}  
  
// get all rewards for a specific user  
/users/{userId}/rewards  
  
// get a specific reward for a specific user  
/users/{userId}/rewards/{rewardId}
```

And rewards in context of Missions looked like this:

```
// get the rewards for a specific mission related to a specific user  
/users/{userId}/rewards/{missionId}  
  
// get the rewards available for a specific mission  
/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, "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 \(page 159\)](#) sections of your documentation. However, brief and clear references to the endpoints are sometimes challenging.

Endpoint for surfreport API

Let's create the Endpoints section for our [fictitious surfreport API \(page 84\)](#). Here's my approach:

Endpoints

GET `surfreport/{beachId}`

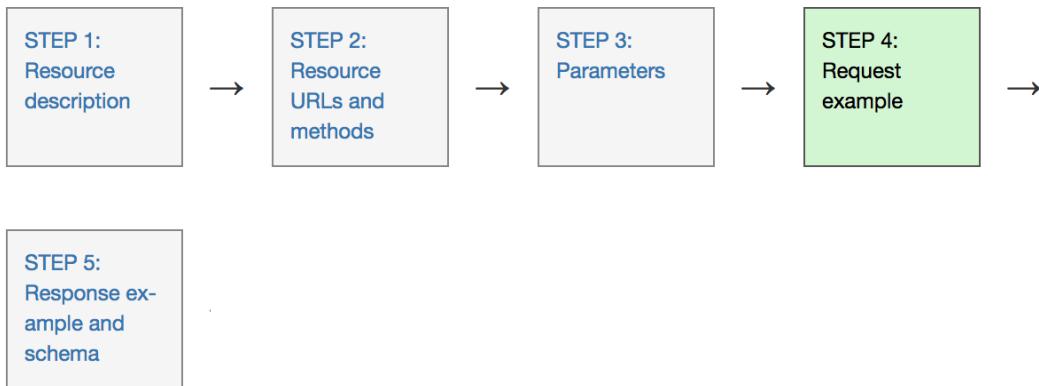
Gets the surf conditions for a specific beach ID.

(There's not much to see here – endpoints look best when styled attractively with CSS.)

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 \(page 101\)](#).

Step 3: Parameters (API reference tutorial)



Parameters: Parameters are options you can pass with the endpoint (such as specifying the response format or the amount returned) to influence the response. There are 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. Not all endpoints contain each type of parameter.

Example of parameters

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

Update Collaboration

Update a collaboration.

SUGGEST EDITS

PUT https://api.box.com/2.0/collaborations/{collab_id}

PATH PARAMS

collab_id	REQUIRED string	
------------------	--------------------	--

QUERY PARAMS

fields	Comma-separated list of fields to include in the response string	
---------------	---	--

BODY PARAMS

role	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> .	
status	The status of the collaboration invitation. Can be <code>accepted</code> , <code>pending</code> , or <code>rejected</code> . string	
can_view_path	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.

Listing parameters in a table

Many times parameters are listed in a simple 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 104)**: Parameters that are included in the request header, usually related to authorization.
- **Path parameters (page 0)**: Parameters that appear within the path of the endpoint, before the query string (`?`). These are usually set off within curly braces.
- **Query string parameters (page 105)**: Parameters that appear in the query string of the endpoint, after the `?` .
- **Request body parameters (page 106)**: Parameters that are included in the request body. Usually submitted as JSON.

The terms for each of these parameter types comes from the [OpenAPI specification \(page 0\)](#), which defines a formal specification that includes descriptions of each parameter type (see the [Path object tutorial \(page 0\)](#)). 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 \(page 104\)](#)
- [Max and min values \(page 104\)](#)

Data types for parameters

Because APIs may not process the parameter correctly if it's the wrong data type or wrong format, 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 probably 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 \(page 177\)](#). I get deeper with the importance of testing in [Testing your docs \(page 144\)](#).

Header parameters

Header parameters are included in the request header. Usually, the header just includes authorization parameters that are common across all endpoints and thus is not documented with each endpoint. Instead, the authorization parameters are documented in the [authorization requirements section \(page 170\)](#).

However, if your endpoint requires specific 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 \(page 56\)](#) 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 in contrast with the other parts of the endpoint.

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), then 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.

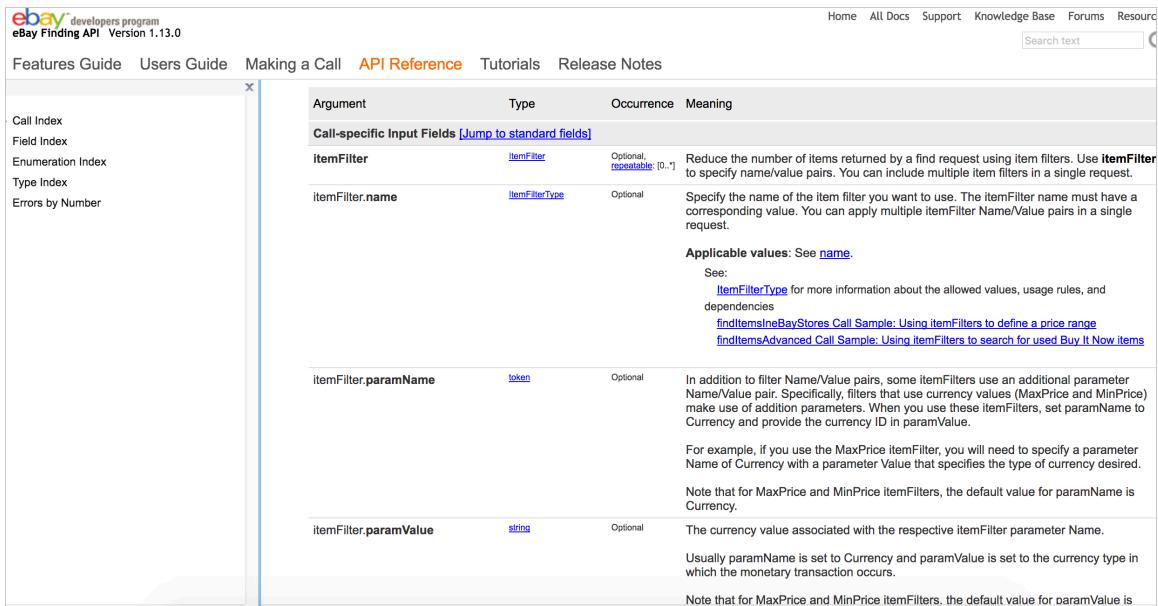
By all means, if the JSON object is relatively small, a table is probably your best option. But there are other approaches that designers have taken as well.

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 panel title is 'Input'. Below the title, a note states: 'The box below lists all fields that could be included in the call request. To learn more about an individual field or its type, click its name in the box (or down to find it in the table below the box).'. It also says 'See also [Samples](#)'. The main content area displays the XML schema for the 'findItemsByProduct' request body:

```
<?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>
  </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 a detailed description of the parameter's purpose and usage.

Argument	Type	Occurrence	Meaning
Jump to standard fields			
itemFilter	ItemFilter	Optional, repeatable: [0..1]	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 organization approach 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 \(page 355\)](#), which we explore later and also [demo \(page 365\)](#), 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

Name	Description
body * required (body)	Pet object that needs to be added to the store

Example Value | **Model**

```

Pet ▾ {
  id           integer($int64)
  category    Category ▾ {
    id           integer($int64)
    name         string
  }
  name*        string
  example: doggie
  photoUrls*   ▾ [
    xml: OrderedMap { "name": "photoUrl", "wrapped": true }string
  ]
  tags         ▾ [
    xml: OrderedMap { "name": "tag", "wrapped": true }Tag ▾ {
      id           integer($int64)
      name         string
    }
  ]
  status        string
}
pet status in the store

```

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 that are 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 \(page 298\)](#).

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 \(page 121\)](#).

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
units	Optional	Options are either <code>imperial</code> or <code>metric</code> . Whether to return the values in imperial or metric measurements. Imperial will use feet, knots, and fahrenheit. Metric will use centimeters, kilometers per hour, and celsius. <code>metric</code> is the default.	String
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 \(page 110\)](#) for the resource.

Step 4: Request example (API reference tutorial)

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.

Example of a request

The following example shows a sample request from the [Callfire API](#):

The screenshot shows a three-column layout. The left column contains a sidebar with links like LOGIN, SIGN UP, LEARN, DOCS, Introduction, Authentication, Methods, Errors, Pagination, Partial Response, Calls, Campaigns, Contacts, and Keywords. The middle column has a heading "Pagination" with a detailed description of how to use the limit and offset parameters for paginating results. Below the description is a JSON snippet showing the response structure. An orange arrow points from the "Example Request" section in the right column to the "Pagination" section in the middle column. The right column contains sections for "Example Request" and "Example Response". The "Example Request" section shows a curl command to query 50 text records starting from 200. The "Example Response" section shows a partial JSON response with items and totalCount fields.

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 \(page 0\)](#).

```
curl -u "username:password" -H "Content-Type:application/json" -X GET "http://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

Search...

Your Configuration
Getting Started
Quick Reference

Objects

- Object Format
- Creating Objects
- Retrieving Objects
- Updating Objects**
- Counters
- Arrays
- Relations

Deleting 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 \(page 0\)](#), Windows has trouble with backslashes).

Other API doc sites might use the full resource URL, such as this plain example from Twitter:

Developer
Use cases
Products
Docs
More
Apply

skip_status	optional	When set to either <i>true</i> , <i>1</i> or <i>0</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 [literarily 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	
Usage	URL
Find movie theaters in zip code 90045	https://api.citygridmedia.com/content/places/v2/search/where? type=movie theater&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 \(page 0\)](#), 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 a section titled "MAKING CALLS" under the "Twilio Voice API > Making Calls" heading. On the left, there's a rating section with five stars and a "Rate this page" button. Below that, there's a brief description of what the API can do. In the center, there's a "Make an Outbound Call" section with a dropdown menu set to "JAVA". To the right, there's a code editor window showing Java code for making an outbound call. The code uses the Twilio Java helper library and includes imports for URISyntaxException, Twilio, rest.api.v2010.account.Call, and type.PhoneNumber. The code then creates an Account object with ACCOUNT_SID and AUTH_TOKEN, initializes it, and creates a new Call object to "+14155551212" using the Twilio REST API. Finally, it prints the call's SID to the console. The code editor has tabs for C#, CURL, JAVA, NODE.JS, PHP, PYTHON, and RUBY, with JAVA currently selected. There are also buttons for "CODE", "OUTPUT", and various sharing options. A note at the top right says "SDK Version: 6.x 7x".

```
SDK Version: 6.x 7x
/a helper library from twilio.com/docs/java/install
URI;
URIException;
.Twilio;
.rest.api.v2010.account.Call;
.type.PhoneNumber;

public class Example {
10 // Find your Account Sid and Token at twilio.com/user/account
11 public static final String ACCOUNT_SID = "ACXXXXXXXXXXXXXXXXXXXXXX";
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());
20 }
```

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:

Combined API



A common use-case is looking up a person and company simultaneously based on a email address. To save you making two requests to do this, we offer a combined lookup API.

This endpoint expects an email address, and will return an object containing both the person and company (if found). A call to the combined lookup will only count as one API call.

HTTP REQUEST

```
GET https://person.clearbit.com/v2/combined/find?email=:email
```

(Where :email is the person's email address)

HTTP GET PARAMS

Alongside the email address you may also provide any additional attributes you have about the person, such as their given and family names. Including more detail will help us be more accurate when searching.

The supported parameters are:

param	Description
email	string (required)
The email address to look up.	

To lookup both a company and person based on an email address:

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

clearbit.Enrichment.find({email: 'alex@alexmacaw.com', s
tream: true})
  .then(function (response) {
    var person = response.person;
    var company = response.company;

    console.log('Name: ', person && person.name.fullName
);
  })
  .catch(function (err) {
    console.error(err);
});
```

The `stream` option ensures that the request blocks until Clearbit has found some data on both the person & company. For cached information this will return in the region of 300 milliseconds, for uncached requests 2-4 seconds. If speed is key, you can omit the `stream` option and try the request again later (if you receive a pending response). Alternatively you can use our [webhook](#) API.

You can see the request in Shell (curl), Ruby, Node, or Python. Developers can easily copy the needed code into their applications, rather than figuring out how to make the translate the curl request into a particular programming language.

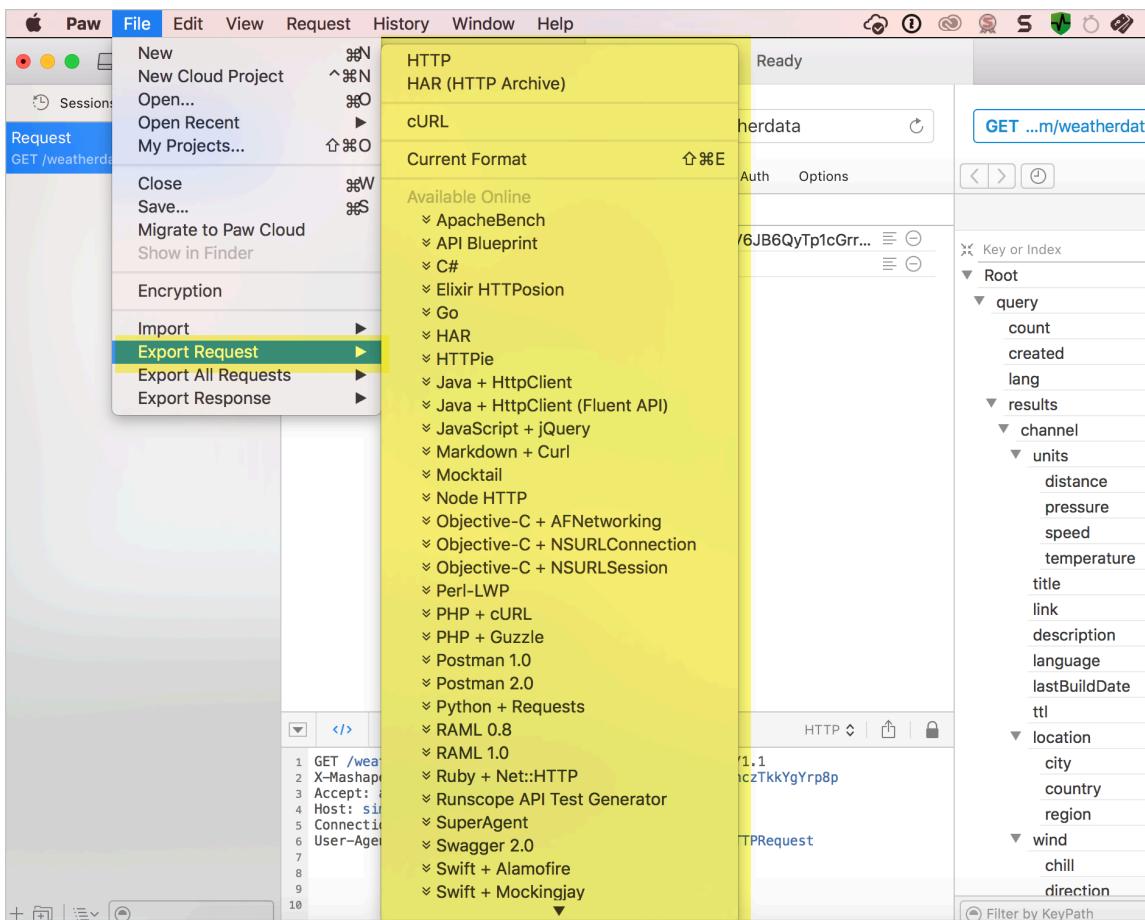
Providing a variety of requests like this, often displayed through [tabs](#), helps make your API easier to implement. It's even better if you can automatically populate the API keys with the actual user's API keys based on their logged-in profile.

However, don't feel so intimidated by this smorgasbord of code samples. Some API doc tools (such as [Readme.io \(page 245\)](#) or [SwaggerHub \(page 372\)](#)) can actually 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 \(page 46\)](#)), 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 \(page 69\)](#). In Postman, after you configure your request, click the **Code** link.

Generate code snippet

Then select the language you want, such as jQuery:

JavaScript Ajax code snippet

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 [Use the JSON from the response payload \(page 69\)](#).)

SDKs provide tooling for APIs

A lot of times, developers will create an SDK (software development kit) 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 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, a sidebar lists various endpoints under categories like 'Interactive Console', 'Albums', 'Artists', 'Tracks', 'Search', and 'Playlists'. The main content area is titled 'Get an Album'. It shows the following details for the 'Get an Album' endpoint:

Description	Get an Album docs
Endpoint	https://api.spotify.com/v1/albums/{id}
HTTP Method	GET
OAuth	Required

Below this, there are input fields for 'Spotify Album ID *' (containing '4aawyAB9vmqN3uQ7FjRGTy'), 'Market' (containing 'ES'), and 'OAuth Token' (containing 'OAuth Access Token'). A yellow button labeled 'GET OAUTH TOKEN' is next to the token field. At the bottom, there are two buttons: 'TRY IT' and 'FILL SAMPLE DATA'. To the right of the input fields, there's a 'cURL Command' section with the following code:

```
curl -X GET "https://api.spotify.com/v1/albums/" -H "Accept: application/json"
```

A 'COPY' button is located to the right of the command.

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 two main sections: "Arguments" and "Useful Values".

Arguments:

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

Useful Values:

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 documentation page. At the top, it says "Article Search API" and "Source: [Swagger 2.0]". There are tabs for "README", "Documentation", and "Console". Below that, there's a search bar with "All" and "GET /articlesearch.json" selected, and a "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 "Parameters" section with fields for "q" (containing "writing"), "fq", "begin_date", and "end_date".

The main content area shows a JSON response for the GET request:

```
{
  "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-mcphee-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 Explorer. Being able to see your own data maybe 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, 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=Z1JPN4QNO23USRXRWFHJCYI3UU2XVLRMKRIRJDNFWEZDYP4&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 example:

```
{
  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 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 \(page 310\)](#)) 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 \(page 355\)](#).

Request example for the `surfreport` endpoint

Let's return to the `surfreport/{beachId}` endpoint in our [sample scenario \(page 84\)](#) and create a request example for it. Here's my approach:

Sample request

```
curl -I -X GET "http://api.openweathermap.org/data/2.5/surfreport?zip=95050%2Cus&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 \(page 121\)](#) 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)

Response example and schema: The response example shows a sample response from the request example. 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

200	Schema	Example	✖ collapse all
		<pre>{ "date": "2016-02-01", "stats": [{ "first_name": "John", "last_name": "Doe", "metrics": { "blocks": 0, "bounce_drops": 0, "bounces": 0, "clicks": 5, "deferred": 0, "delivered": 0, "invalid_emails": 0, "opens": 10, "processed": 10, "requests": 10, "spam_report_drops": 0, "spam_reports": 0, "unique_clicks": 0, "uniqueOpens": 0, "unsubscribe_drops": 0, "unsubscribes": 0 }, "name": "user1", "type": "subuser" }] }</pre>	

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

Responses application/json			
200	Schema	Example	x collapse all
	object		
	date	string	The date the statistics were gathered.
	stats	array[object]	The list of statistics.
	first_name	string	The first name of the subuser.
	last_name	string	The last name of the subuser.
	metrics	object	
	blocks	integer	The number of emails that were not allowed to be delivered by ISPs.
	bounce_drops	integer	The number of emails that were dropped because of a bounce.
	bounces	integer	The number of emails that bounced instead of being delivered.
	clicks	integer	The number of links that were clicked in your emails.
	deferred	integer	The number of emails that temporarily could not be delivered.
	delivered	integer	The number of emails SendGrid was able to confirm were actually delivered to a recipient.
	invalid_emails	integer	The number of recipients who had malformed email addresses or whose mail provider reported the address as invalid.
	opens	integer	The total number of times your emails were opened by recipients.

The definition of the response is called the schema or the 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 created for the [SwaggerUI activity \(page 0\)](#), you can see a distinction between the response example and the response schema. Here's the **Example Value**:

Responses

Code	Description
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>

The example response should correspond with the example request. And 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:

The screenshot shows a JSON schema for a response object. The schema includes properties for various times of day (sunrise, sunset, solar_noon, day_length) and twilight periods (civil_twilight_begin, civil_twilight_end, nautical_twilight_begin, nautical_twilight_end, astronomical_twilight_begin, astronomical_twilight_end). It also includes a status property. The schema is annotated with descriptions and examples for each field.

```

{
  "Response": {
    "results": {
      "sunrise": {
        "type": "string",
        "example": "7:27:02 AM",
        "description": "time of sunrise"
      },
      "sunset": {
        "type": "string",
        "example": "5:05:55 PM",
        "description": "time of sunset"
      },
      "solar_noon": {
        "type": "string",
        "example": "12:16:28 PM",
        "description": "time when sun is at apex"
      },
      "day_length": {
        "type": "string",
        "example": "9:38:53",
        "description": "length of the day"
      },
      "civil_twilight_begin": {
        "type": "string",
        "example": "6:58:14 AM",
        "description": "time when civil twilight begins"
      },
      "civil_twilight_end": {
        "type": "string",
        "example": "5:34:43 PM",
        "description": "time when civil twilight ends"
      },
      "nautical_twilight_begin": {
        "type": "string",
        "example": "6:25:47 AM",
        "description": "time when nautical twilight begins"
      },
      "nautical_twilight_end": {
        "type": "string",
        "example": "6:07:10 PM",
        "description": "time when nautical twilight ends"
      },
      "astronomical_twilight_begin": {
        "type": "string",
        "example": "5:54:14 AM",
        "description": "time when astronomical twilight begins"
      },
      "astronomical_twilight_end": {
        "type": "string",
        "example": "6:38:43 PM",
        "description": "time when astronomical twilight ends"
      }
    },
    "status": {
      "type": "string",
      "example": "OK"
    }
  }
}

```

Four possible statuses:

- OK: No errors occurred.
- INVALID_REQUEST: Either lat or lng parameters are missing or invalid.
- INVALID_DATE: The date parameter is missing or invalid.
- UNKNOWN_ERROR: The request could not be processed due to a server error. The request may succeed if you try again.

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 \(page 177\)](#)), 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 \(page 274\)](#) or markdown syntax with [GitHub \(page 253\)](#), 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 could 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](#), he 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.

RETURNS User account information.

Sample JSON response

```
{
    "uid": 12345678,
    "display_name": "John User",
    "name_details": {
        "familiar_name": "John",
        "given_name": "John",
        "surname": "User"
    },
    "referral_link": "https://www.dropbox.com/referrals/r1a2n3d4m5s6t7",
    "country": "US",
    "locale": "en",
    "is_paired": false,
    "team": {
        "name": "Acme Inc.",
        "team_id": "dbtid:1234abcd"
    },
    "quota_info": {
        "shared": 253738410565,
        "quota": 107374182400000,
        "normal": 680031877871
    }
}
```

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).
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.
team/name	The name of the team the user belongs to.
team/team_id	The ID of the team the user belongs to.
quota_info/normal	The user's used quota outside of shared folders (bytes).
quota_info/shared	The user's used quota in shared folders (bytes). If the user belongs to a team, this includes all usage contributed to the team's quota outside of the user's own used quota (bytes).
quota_info/quota	The user's total quota allocation (bytes). If the user belongs to a team, the team's total quota allocation (bytes).

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,  
            "user": "somebitlyuser",  
            "user_ts": 1337892044  
          }  
        ]  
      }  
    ]  
  }  
}
```

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) </OriginalRetailP
        <PricingTreatment> PricingTreatmentCodeType </PricingTreatme
        <SoldOffeBay> boolean </SoldOffeBay>
        <SoldOneBay> boolean </SoldOneBay>
      </DiscountPriceInfo>
      <EndTime> dateTime </EndTime>
```

Here's the response documentation:

Field	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 an item listing qualifies it to be listed as a MAP item (PricingTreatment returns MAP), the item price cannot be directly displayed on the page containing the item. When listing a MAP item, the seller stipulates how they want the buyer to view the price of the item by setting this field to either PreCheckout or DuringCheckout. If this field is not set for a MAP item, the treatment defaults to PreCheckout. If this field is set to PreCheckout, the buyer must click a link (or button) to view the item price on a different page (such as in a pop-up window). If this field is set to DuringCheckout, the StartPrice must be shown only when the buyer in the eBay checkout flow. MAP items are supported only on the eBay US site.

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 other endpoints such as 'Introduction', 'Authentication', 'Errors', 'Pagination', 'Versioning', 'Expanding objec...', 'Metadata', 'Idempotent req...', 'METHODS', 'Charges', 'The charge obj...', 'Create a charge', 'Retrieve a char...', 'Update a charge', 'Capture a char...', 'List all charges', and 'Refunds'. The main content area has three columns. The first column contains the JSON schema for 'Create a charge': 'currency' (REQUIRED, 3-letter ISO code for currency), 'customer' (optional, either customer or source is required, The ID of an existing customer that will be charged in this request), and 'source' (optional, either source or customer is required, A payment source to be charged, such as a credit card. If you also pass a customer ID, the source must be the ID of a source belonging to the customer. Otherwise, if you do not pass a customer ID, the source you provide must either be a token, like the ones returned by [Stripe.js](#), or a dictionary containing a user's credit card details, with the options described below. Although not all information is required, the extra info helps prevent fraud.). The second column shows the JSON response example with a 'Hide child attributes' button. The third column displays code samples in curl and various programming languages (Ruby, Python, PHP, Java, Node, Go).

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)		"UID" : "eb043b43-1d66-472b-a6ee-ad48def81b96",
Name	String (30)		"Name" : "Business Bank Account #2",
DisplayID	String (6)		"DisplayID" : "1-1120",
Classification	Description	Classification	"Classification" : "Asset",
Type	Account code format includes separator ie 1-1100	CountType	"Type" : "Bank",
Number	Integer (4)	Integer (4)	"Number" : 1120,
Description	String (255)		"Description" : "Bank account clearwtr",
ParentAccount			"ParentAccount" : {
UID	Guid (36)		"UID" : "f5cc9506-3472-4227-8c45-7a95c322c38b",
Name	String (6)		"Name" : "Bank Accounts",
DisplayID	String (6)		"DisplayID" : "1-1100",
URI	String		"URI" : "
IsActive	Boolean		(domain)/{cf_guid}/GeneralLedger/Account/f5cc9506-3472-4227-8c45-7a95c322c38b"
TaxCode			},
UID	Guid (36)		"IsActive" : true,
			"TaxCode" : {
			"UID" : "94966872-b140-4da2-bc43-5dd74f33a09",
			"Code" : "N-T",
			"URI" : "
			(domain)/{cf_guid}/GeneralLedger/TaxCode/94966872-b140-4da2-bc43-5dd74f33a09"
			},
			"Level" : 4,
			"OpeningBalance" : 100000.00,
			"CurrentBalance" : 5000.00.

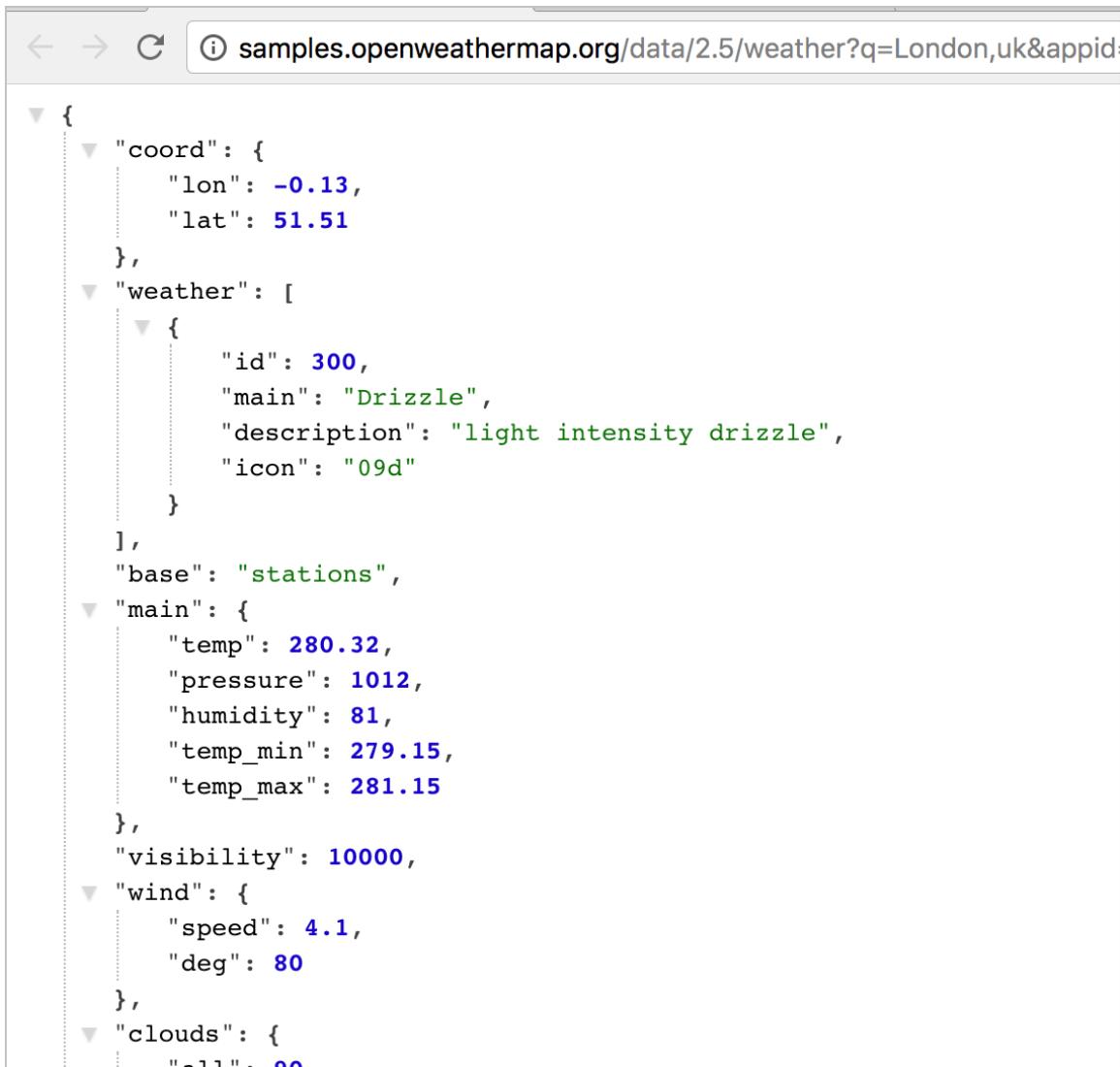
This approach facilitates scanning, and the popover + collapsible approach allows you to compress the table so you can jump to the part you're interested in.

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 [Open Weather API](#). 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=080c10e04cb0bdaa155f5f2f45a1c44. 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 \(page 110\)](#), 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 \(page 177\)](#).

Response example and schema for the surfreport endpoint

For the `surfreport/{beachId}` endpoint that we've been exploring in our [sample API scenario \(page 84\)](#), 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 \(page 137\)](#).

Putting it all together

ACTIVITY



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

Query string parameter	Required / optional	Description	Type
<code>units</code>	Optional	Options are either <code>imperial</code> or <code>metric</code> . Whether to return the values in imperial or metric measurements. Imperial will use feet, knots, and fahrenheit. Metric will use centimeters, kilometers per hour, and celsius. <code>metric</code> is the default.	String
<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 "http://api.openweathermap.org/data/2.5/surfreport?zip=95050%2Cus&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

Response item	Description	Data type
{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 upcoming sections, [Publishing API Documentation \(page 191\)](#).

Next steps

Now that you've completed the [API reference tutorial \(page 87\)](#), 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 \(page 142\)](#).

Activity: Critique or create an API reference topic

After completing the [API reference tutorial \(page 87\)](#), 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 \(page 31\)](#).

Critique the API reference documentation

If you've found an open source project, great. If you don't have a project but you still want to do the activity, select one of the API from the [list here \(page 195\)](#).

To critique an API reference topic:

1. Locate one of the reference topics for a resource in the API.
2. Identify each of the sections in the existing documentation:
 - Resource description
 - Endpoints
 - Parameters
 - Request example
 - Response example and schema

The section names may differ, but they usually are easily recognizable.

3. Evaluate each of these sections and assess whether the documentation is complete. Are there areas for improvement? Critique one or more of the API reference topics.

Here are some questions to look at:

Resource description: Is the description action-oriented? Is it a brief 1-3 sentence summary? Is it clear? Does it link to more information somewhere else?

Endpoints and methods: Does the endpoint list the methods available? Are any path parameters in the endpoint easy to identify? If there are multiple endpoints, are they logically grouped?

Parameters: Is each parameter described? Are the parameters separated out into different sections by parameter type? If it's a request body parameter, are the data types indicated? Are max and min values identified? Any unsupported values noted?

Request example: Does the sample request work (with the right authorization)? Does it include a representative number of parameters? Is it formatted correctly in curl? Are any other languages shown with the sample request? Is the code syntax highlighted?

Response example: Does the sample response match the sample request? Is it formatted and highlighted correctly? Is each element in the response described, along with the data type? Does the response documentation separate the example from the description, or combine the two? How are nested objects portrayed? Are any status and error codes listed?

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 \(page 268\)](#) 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. Go to [Overview to testing your docs \(page 145\)](#).

Testing your API documentation

Overview to testing your docs.....	145
Set up a test environment	146
Test all instructions yourself	149
Test your assumptions	154
Activity: Test your project's documentation.....	158

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.

I believe so strongly in testing API doc content that I've created an entire section devoted to this topic. This section includes three topics:

- Set up a test environment (page 146)
- Test all instructions yourself (page 149)
- Test your assumptions (page 154)



Photo from Flickr (<https://flic.kr/p/6Grete>). 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 from a test server \(page 146\)](#)
- [Testing local builds \(page 146\)](#)
- [Testing sample apps in specific programming languages \(page 147\)](#)
- [Testing hardware products \(page 148\)](#)

Testing on a test servers

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.

With the test system, you’ll need to get the appropriate URLs, login IDs, roles, etc., from QA. Ask if there’s anything you shouldn’t alter or delete because sometimes the same testing environment is shared among groups. 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 asking around for help.

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.

Testing local builds

Many times, developers work on local instances of the system, meaning 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 or to access a test server, ask an engineer to install the local system on your machine. Tell him or her that, in order to write good documentation — documentation that is accurate, complete, and doesn't assume anything — you need access to these test systems.

If the setup to build locally is complex, you may need to ask a developer for help. 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.

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.

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 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. If the app is in PHP, you may need to install PHP. Or you may need to download Android Studio and connect it to an 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 included in certain 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. 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 doing anything more than just transcribing and relaying the information given to them. 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.

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. 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 \(page 149\)](#).

Test all instructions yourself

After setting up your [test environment \(page 146\)](#), the next step is to test your instructions. This will likely involve testing API endpoints with various parameters along with testing other configurations. Testing all your docs can be challenging, but it's where you'll get the most value in 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, 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 \(page 54\)](#), [Postman \(page 46\)](#), 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.

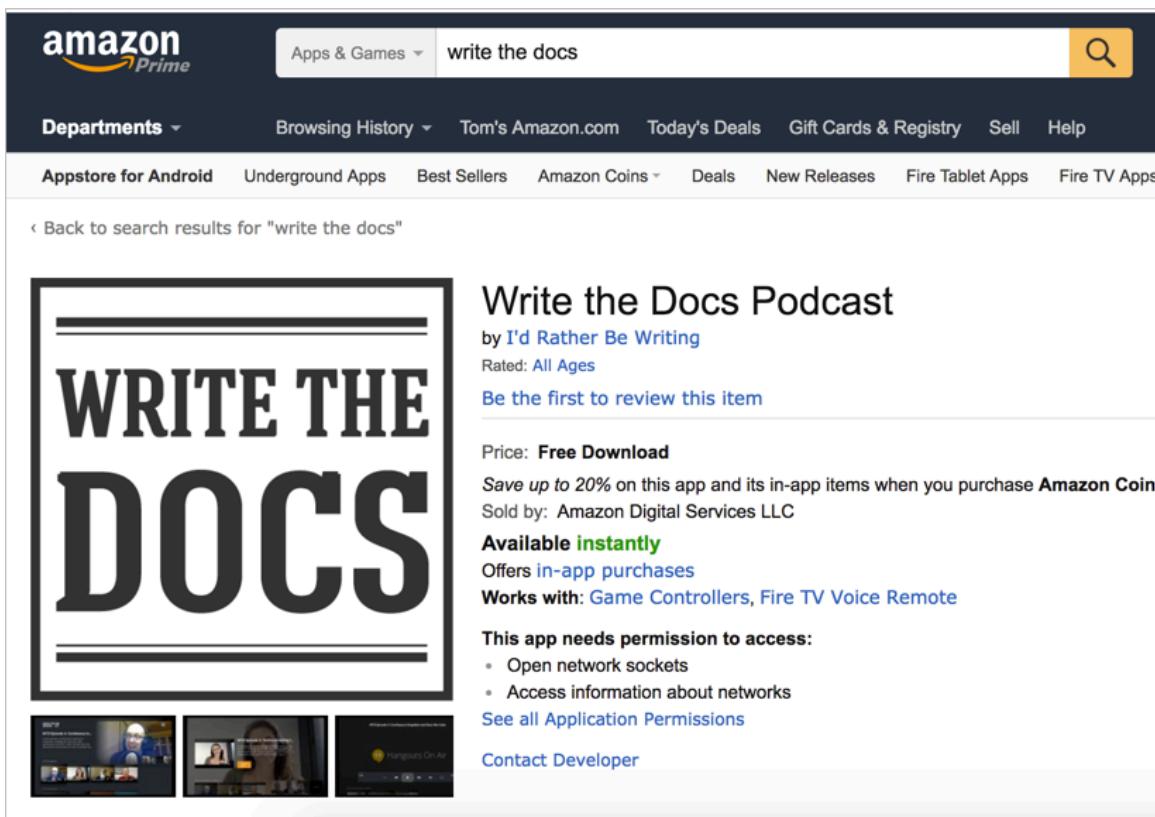
To build my sample app, I had to first figure out how to get content for my app. I decided to take the video recordings of podcasts and meetups that we had through the [Write the Docs podcast](#) and various WTD meetups and use that media for the app.

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.

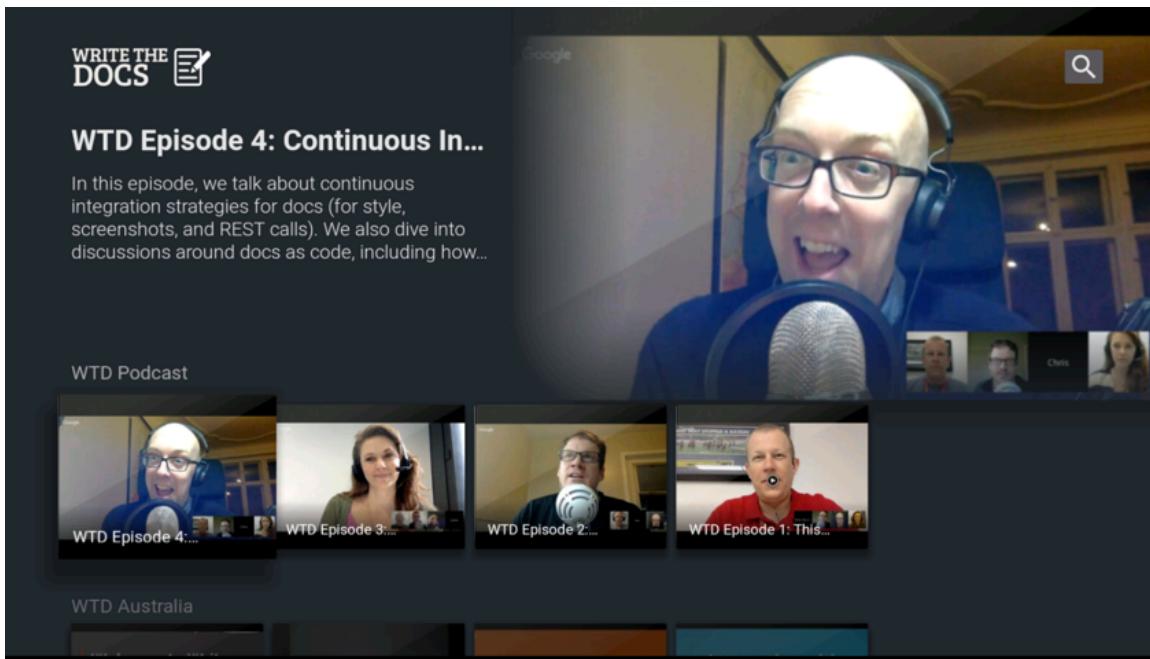
I used Jekyll to build my feed. (You can view my JSON-based feed here: podcast.writethedocs.org/fab.json.) The most difficult part in setting up this feed was configuring the `recommendations` object. Each video has some `tags`. The `recommendations` object needs to show other videos that have the same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

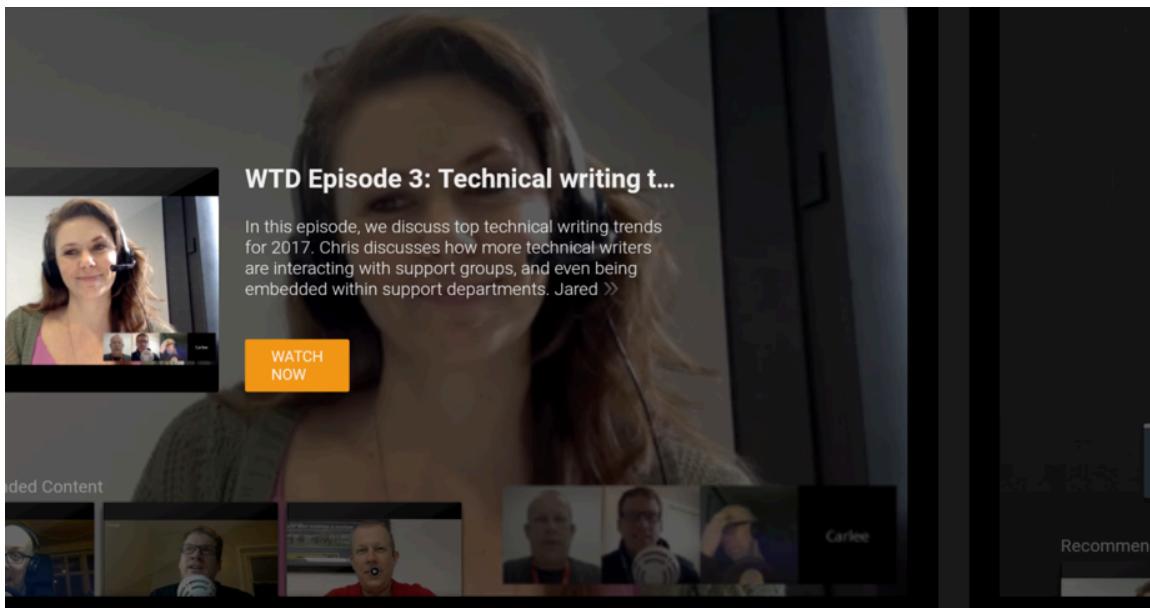
Submitting the app into the Appstore was fun and illuminated parts of the developer's workflow that I hadn't previously understood. You can view the Write the Docs podcast app in the Amazon Appstore website [here](#).



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 contains 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.

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 the team.

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 they 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 info you send in the display. 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 get the initial foundation going, 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 listen to what engineers say, write up notes, send it to them for review, and 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 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, your doc is done. You can always go back over your existing docs and improve them. Consider the first release a kind of "Day 1" for your docs. It's the first iteration. Your docs 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 it for the upcoming release.

With the first release, if you can capture feedback as your docs get used (feedback from forums, contact email, logs, and other means), you can improve your docs and see gaps that you likely missed. In some ways, each time users consult your docs they are testing them.

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 that may or may not be shared with your audience. While [testing your documentation \(page 149\)](#), recognize that what may seem clear to you may be confusing to your users. Learn to identify these assumptions that can interfere with your audience's ability to follow the instructions in your docs.

Assumptions about terminology

You might assume that your audience already know how to SSH onto a server, create authorizations in REST headers, use curl to submit calls, 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 throw them in the trash.

For example, my 10-year-old daughter is starting to cook. She feels confident that if the 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, 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 *JRE* 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.

Solutions for addressing different audiences

The solution to addressing different audiences doesn't involve writing entirely different sets of documentation. 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 as developers. There are some cases where tech docs are "tested" by QA, but when I do get feedback from QA, the reviewers rarely assess clarity, organization, or other communication. They just highlight any errors they find.

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 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. Strangely, few IT shops actually have a consistent quality assurance process for documentation. You 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. 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 the old “Oh, users will know that.” The problem is that we’re usually so disconnected with the actual user experience — we rarely see users trying out docs — we can’t recognize the “users-will-know-how-to-do-that” statement for the fallacy that it is.

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 to try out the instructions.

Using your colleagues as test subjects

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).

Additionally, 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 button, 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.

What might come out of a user test from a colleague is more than highlighting shortcomings about poor instruction. 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? Hmm.*)

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 in the software cycle you catch an error, 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 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 \(page 31\)](#), find one of the following:

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

Test the content. 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 tasks	160
Documenting the API overview	165
Documenting the getting started section	166
Documenting authentication and authorization requirements.....	170
Documenting status and error codes	177
Documenting code samples and tutorials.....	184
How to create the quick reference guide	188
Next phase of course	190

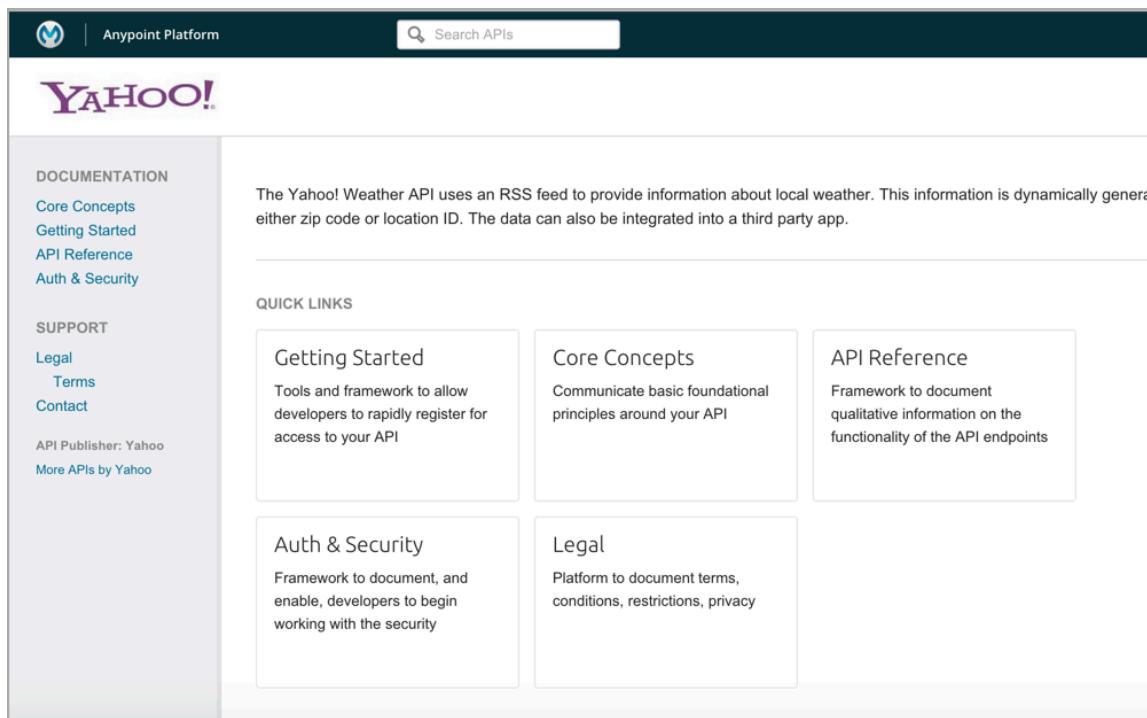
User guide tasks

Up until this point, we've been focusing on the endpoint (or reference) documentation aspect of user guides. The endpoint documentation is only one part (albeit a significant one) in API documentation. You also need to create a user guide and tutorials.

User guide overview

Whereas the endpoint documentation explains how to use each of the endpoints, you also need to explain how to use the API overall. There are other sections common to API documentation that you must also include.

In Mulesoft's API tooling, you can see some other sections common to API documentation:

A screenshot of the Yahoo Weather API page on the Anypoint Platform. The page has a dark header with the Mule logo, "Anypoint Platform", and a search bar. Below the header is the Yahoo logo. On the left, there's a sidebar with "DOCUMENTATION" (Core Concepts, Getting Started, API Reference, Auth & Security), "SUPPORT" (Legal, Terms, Contact), and links for "API Publisher: Yahoo" and "More APIs by Yahoo". The main content area starts with a brief description of the Yahoo Weather API: "The Yahoo! Weather API uses an RSS feed to provide information about local weather. This information is dynamically generated either zip code or location ID. The data can also be integrated into a third party app." Below this is a section titled "QUICK LINKS" with four boxes: "Getting Started" (Tools and framework to allow developers to rapidly register for access to your API), "Core Concepts" (Communicate basic foundational principles around your API), "Auth & Security" (Framework to document, and enable, developers to begin working with the security), and "Legal" (Platform to document terms, conditions, restrictions, privacy).

Although this is the Yahoo Weather API page, all APIs using the Mulesoft platform have this same template.

Essential sections in a user guide

Some of these other sections to include in your documentation include the following:

- Overview
- Getting started
- Authorization keys
- Code samples/tutorials
- Response and error codes
- Quick reference

Since the content of these sections varies a lot based on your API, it's not practical to explore each of these sections using the same API like we did with the API endpoint reference documentation. But I'll briefly touch upon some of these sections.

[Sendgrid's documentation](#) has a good example of these other user-guide sections essential to API documentation. It does a good job showing how API documentation is more than just a collection of endpoints.

Also include the usual user guide stuff

Beyond the sections outlined above, you should include the usual stuff that you put in user guides. By the usual stuff, I mean you list out the common tasks you expect your users to do. What are their real business scenarios for which they'll use your API?

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

Provide how-to's for these tasks 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 — ask what will users want to do with this product, what can they do, and how do they do it.

Breaking down API doc into guides, tutorials, and reference

Perhaps no other genre of technical documentation has such variety in the outputs as API documentation. Almost every API documentation site looks unique. REST APIs are as diverse as different sites on the web, each with their own branding, navigation, terminology, and style.

Despite the wide variety of APIs, there is some commonality among them. The common ground is primarily in the endpoint documentation. But user guides have common themes as well.

In a [blog post by the writers at Parse](#), they break down API doc content into three main types:

Reference: This is the listing of all the functionality in excruciating detail. This includes all datatype and function specs. Your advanced developers will leave this open in a tab all day long.

Guides: This is somewhere between the reference and tutorials. It should be like your reference with prose that explains how to use your API.

Tutorials: These teach your users specific things that they can do with your API, and are usually tightly focused on just a few pieces of functionality. Bonus points if you include working sample code.

This division of content represents the API documentation genre well and serves as a good guide as you develop strategies for publishing API documentation.

In Mulesoft's API platform, you can see many of these sections in their standard template for API documentation:

The screenshot shows the Anypoint Platform interface for the Yahoo! Weather API. At the top, there's a navigation bar with the Mule logo, 'Anypoint Platform', and a search bar labeled 'Search APIs'. Below the header, the title 'YAHOO!' is prominently displayed. On the left side, there's a sidebar with two main sections: 'DOCUMENTATION' and 'SUPPORT'. Under 'DOCUMENTATION', links include 'Core Concepts', 'Getting Started', 'API Reference', and 'Auth & Security'. Under 'SUPPORT', links include 'Legal', 'Terms', and 'Contact'. Below the sidebar, a main content area starts with a heading 'The Yahoo! Weather API uses an RSS feed to provide information about local weather. This information is dynamically generated either zip code or location ID. The data can also be integrated into a third party app.' To the right of this text, there are five boxes under the heading 'QUICK LINKS': 'Getting Started' (Tools and framework to allow developers to rapidly register for access to your API), 'Core Concepts' (Communicate basic foundational principles around your API), 'API Reference' (Framework to document qualitative information on the functionality of the API endpoints), 'Auth & Security' (Framework to document, and enable, developers to begin working with the security), and 'Legal' (Platform to document terms, conditions, restrictions, privacy).

I won't get into too much detail about each of these sections. In previous sections of this course, I explored the content development aspect of API documentation in depth. Here I'll just list the salient points.

Guides

In most API guide articles, you'll find the following recurring themes in the guides section:

- API Overview
- Authorization keys
- Core concepts
- Rate limits
- Code samples
- Quick reference
- Glossary

Guide articles aren't auto-generated, and they vary a lot from product to product. When technical writers are involved in API documentation, they're almost always in charge of this content.

Tutorials

The second genre of content is tutorial articles. Whether it's called Getting Started, Hello World Tutorial, First Steps, or something else, the point of the tutorial articles is to guide a new developer into creating something simple and immediate with the API.

By showing the developer how to create something from beginning to end, you provide an overall picture of the workflow and necessary steps to getting output with the API. Once a developer can pass the authorization keys correctly, construct a request, and get a response, he or she can start using practically any endpoint in the API.

Here's a list of tutorials from Parse:



Some tutorials can even serve as reference implementations, showing full-scale code that shows how to implement something in a detailed way. This kind of code is highly appealing to developers because it usually helps clarify all the coding details.

Reference

Finally, reference documentation is probably the most characteristic part of API documentation. Reference documentation is highly structured and templated. Reference documentation follows common patterns when it comes to describing endpoints.

In most endpoint documentation, you'll see the following sections:

- Resource description
- Endpoint
- Methods
- Parameters
- Request submission example
- Request response example
- Status and error codes
- Code samples

If engineers write anything, it's usually the endpoint reference material.

Note that the endpoint documentation is never meant to be a starting point. The information is meant to be browsed, and a new developer will need some background information about authorization keys and more to use the endpoints.

Here's a sample page showing endpoints from Instagram's API:

The screenshot shows the Instagram API documentation interface. The left sidebar has a search bar and links to Overview, Authentication, Secure API Requests, Real-time, Mobile Sharing, API Console, and a detailed Endpoints section. The Endpoints section is expanded to show sub-links for Users, Relationships, Media, Comments, Likes, Tags, Locations, and Geographies. The main content area is titled "Relationship Endpoints" and describes relationships using terms like "outgoing_status" and "incoming_status". It lists several API endpoints with their descriptions:

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

Below these, three specific examples are shown in boxes:

- GET /users/ user-id /follows**
https://api.instagram.com/v1/users/{user-id}/follows?access_token=ACCESS-TOKEN RESPONSE
Get the list of users this user follows.
- GET /users/ user-id /followed-by**
https://api.instagram.com/v1/users/{user-id}/followed-by?access_token=ACCESS-TOKEN RESPONSE
Get the list of users this user is followed by.
- GET /users/self/requested-by**

Documenting the API overview

The overview explains what you can do with the API (high-level business goals), and who the API is for. 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.

Sample overview

The SendGrid API does a good job at providing an overview:

The screenshot shows the SendGrid User Guide's "SendGrid Overview" page. The left sidebar has a dark background with white text. It includes a search bar, a navigation menu with "Docs Home", "User Guide" (which is expanded), and several sub-sections under "User Guide": "SendGrid Overview" (selected, highlighted in blue), "Dashboard", "Email Activity", "About the User Guide", "Setting Up Your Server", "SendGrid for Mobile", and "Marketing Emails". The main content area has a light background. At the top, it shows the breadcrumb "Documentation > User Guide > SendGrid Overview". Below that is the title "SendGrid Overview". Under the title is a section titled "What is SendGrid?". The text describes SendGrid as a cloud-based SMTP provider that manages email servers, scaling infrastructure, ISP outreach, and reputation monitoring. It also mentions real-time analytics. Another section, "Who is SendGrid for?", describes SendGrid for anyone needing reliable, scalable email for transactional or marketing emails, handling billions of emails monthly.

Common business scenarios

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 me?

Where to put the overview

Your overview should probably go on the homepage of the API, or be a link from the homepage. This is really where you define your audience as well, since the degree to which you explain what the API does depends on how you perceive the audience.

Documenting getting started section

Following the Overview section, you usually have a “Getting started” section that details the first steps users need to start using the API.

Common topics in getting started

The “Getting started” section should explain the first steps users must take to start using the API. Some of these steps might involve the following:

- Signing up for an account
- Getting API keys
- Making a request
- Reviewing the endpoints available
- Calling a specific endpoint

Show the general pattern for requests

When you start listing out the endpoints for your resources, you just list the “end point” part of the URL. You don’t list the full resource URL that users will need to make the request. Listing out the full resource URL with each endpoint would be tedious and take up a lot of space.

You generally list the full resource URL in a Getting Started section that shows how to make a call to the API.

For example, you might explain that the domain root for making a request is this:

```
http://myapi.com/v2/
```

And when you combine the domain root with a sample endpoint, it looks like this:

```
http://myapi.com/v2/homes/{id}
```

Once users know the domain root, they can easily add any endpoint to that domain root to construct a request.

Sample Getting Started sections

Here’s the Getting Started section from the Alchemy API:

Getting Started with AlchemyAPI

Here are some short, simple instructions that walk through the basic steps to integrate AlchemyAPI's text and image analysis tools in your application. If you have any questions, please [contact support](#).

How to use AlchemyAPI

4 simple steps:

1. Get API Key - use a key you already have or [register for a free key](#).
2. Download an SDK - visit our [SDK page](#) and pick out the SDK in your favorite programming language.
3. Select a function - Do you want keywords? Entities? Sentiment analysis? Do you want to analyze a URL or a block of text? The SDKs will provide easy access to each API function.
4. Parse the response data and utilize in your application.

Getting Started Tutorials

Will you be using AlchemyAPI in an application coded in Python, PHP, Ruby or Node.js? If so, here's a Getting Started Tutorial that guides you through the process. Select your programming language below:

- [Using AlchemyAPI with Python](#)
- [Using AlchemyAPI with PHP](#)
- [Using AlchemyAPI with Ruby](#)
- [Using AlchemyAPI with Node.js](#)

[Back to Top](#)

Here's a Getting Started tutorial from the HipChat API:

The screenshot shows the HipChat API Documentation homepage. The main navigation bar includes links for Overview, Authentication, Webhooks, Title expansion, Rate limiting, Response codes, Integrations, and various API endpoints like Capabilities and Emoticons. The 'Getting started' section is highlighted, featuring a 'Supported Platforms' list (HipChat.com, HipChat Server), a 'Build your own integrations' guide, and a 'Develop an independent add-on' section. A sidebar on the right provides an 'API changelog' with a link to the HipChat page on API Changelog.

Here's a Getting Started section from the Aeris Weather API:

The screenshot shows the Aeris Weather API documentation website. At the top, there's a dark blue header with the Aeris Weather logo on the left and four navigation links: CONSUME, DEVELOP, VISUALIZE, and MANAGE. Below the header, a breadcrumb trail reads HELP CENTER / DOCS / AERIS WEATHER API / GETTING STARTED. On the left, a sidebar has a blue header "Aeris Weather API" with sub-links: Getting Started, Reference, and Downloads. The main content area has a large title "Getting Started". To its right, a text block says "You'll need an active Aeris API subscription and your application register access ID and secret key." Below this, four steps are listed: Step 1: Sign up for an Aeris API subscription service. We offer a free dev drive. Step 2: Log in to your account to register your application for an API access key. Step 3: Find the endpoints and actions that provide you with the data you need. Step 4: Review our weather toolkits to speed up your weather integration.

Here's another example of a Getting Started tutorial from Smugmug's API:

The screenshot shows the SmugMug API documentation website. At the top, there's a header with the SmugMug logo and a "PHOTO SHARING" link. The main navigation menu on the left includes "SmugMug API" (selected), "BETA" (highlighted in green), "Guidelines for Beta Users", "Release Log", "Known Issues", "Contact Us", "Legal", and "Legal (Beta)". Other sections in the menu are "Tutorial", "API Concepts", and "Advanced Topics". The main content area has a title "Welcome to the SmugMug API!". It includes a beta announcement: "The SmugMug API v2 beta is here! As of December 2nd, 2014, anyone is now welcome to join our beta program. Read the [announcement](#) on our blog." Below this is a list of steps: "Request a new API key", "Get beta program access for your existing API keys", "Documentation for the stable API v1.3.0", and "API v2 tutorial". A section titled "What is the SmugMug API?" explains the API's purpose and features. Another section, "Learn the SmugMug API", contains a button labeled "First step: Getting an API Key".

I like how, right from the start, Smugmug tries to hold your hand to get you started. In this case, the tutorial for getting started is integrated directly in with the main documentation.

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.

Hello World tutorials

In developer documentation, one common topic type is a Hello World tutorial. The Hello World tutorial holds a user's hand from start to finish in producing the simplest possible output with the system. The simplest output might just be a message that says "Hello World."

Although you don't usually write Hello World messages with the API, the concept is the same. You want to show a user how to use your API to get the simplest and easiest result, so they get a sense of how it works and feel productive. That's what the Getting Started section is all about.

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.

Documenting 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 just 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 what you should focus on in documentation.

Defining terms

First, a brief definition of terms:

- **Authentication:** Proving correct identity
- **Authorization:** Allowing a certain action

An API might authenticate you but not authorize you to make a certain request.

Consequences if an API lacks security

There are many different ways to enforce authentication and authorization with the API requests. Enforcing this authentication and authorization is vital. Consider the following scenarios 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, making a revenue model associated with your API difficult.
- You couldn't track who is using your API, or what endpoints are most used
- Someone could possibly make malicious DELETE requests on another person's data through API calls
- The wrong person could intercept or access private information and steal it

Clearly, API developers must think about ways to make APIs secure. There are quite a few different methods. I'll explain a few of the most common ones here.

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 in a 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 is associated with a specific app that you register.

The company producing the API might use the API key for any of the following:

- 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
- Apply different permission levels to different users

Sometimes APIs will 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.

In some API documentation, when you're logged into the site, your API key automatically gets populated into the sample code and API Explorer. (Flickr's API does this, for example.)

Basic Auth

One 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 header:

```
Authorization: Basic bG9s0nNlY3VzZQ==
```

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 like this:

1. Click the **Authorization** tab.
2. Type the **username** and **password** on the right of the colon on each row.
3. Click **Update Request**.

The Headers tab now contains 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.

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).

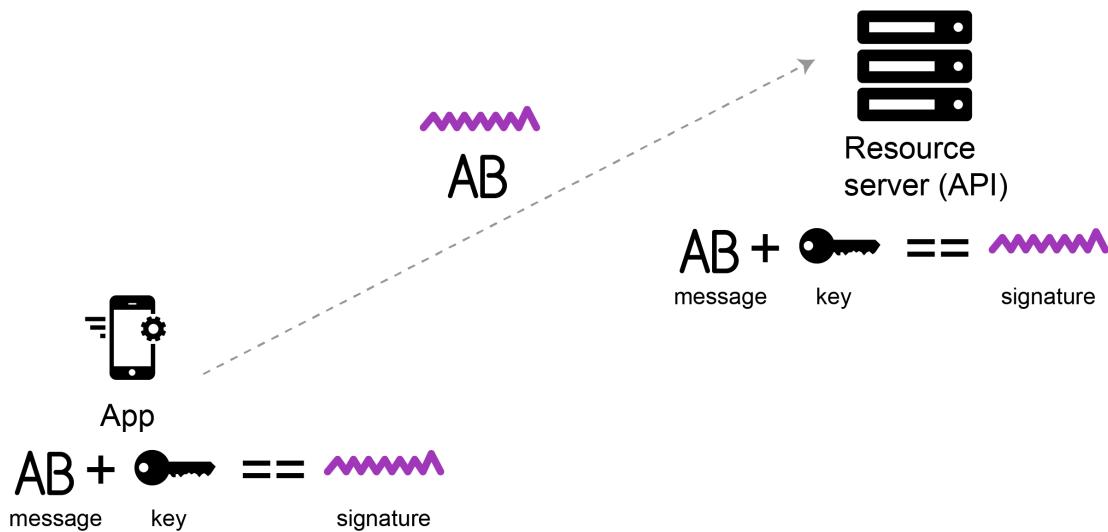
The message is then encoded by the secret key and passed through a secure hashing algorithm (SHA). (A hash is a scramble of a string based on an algorithm.) The resulting value, referred to as a signature, is placed in the request header.

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

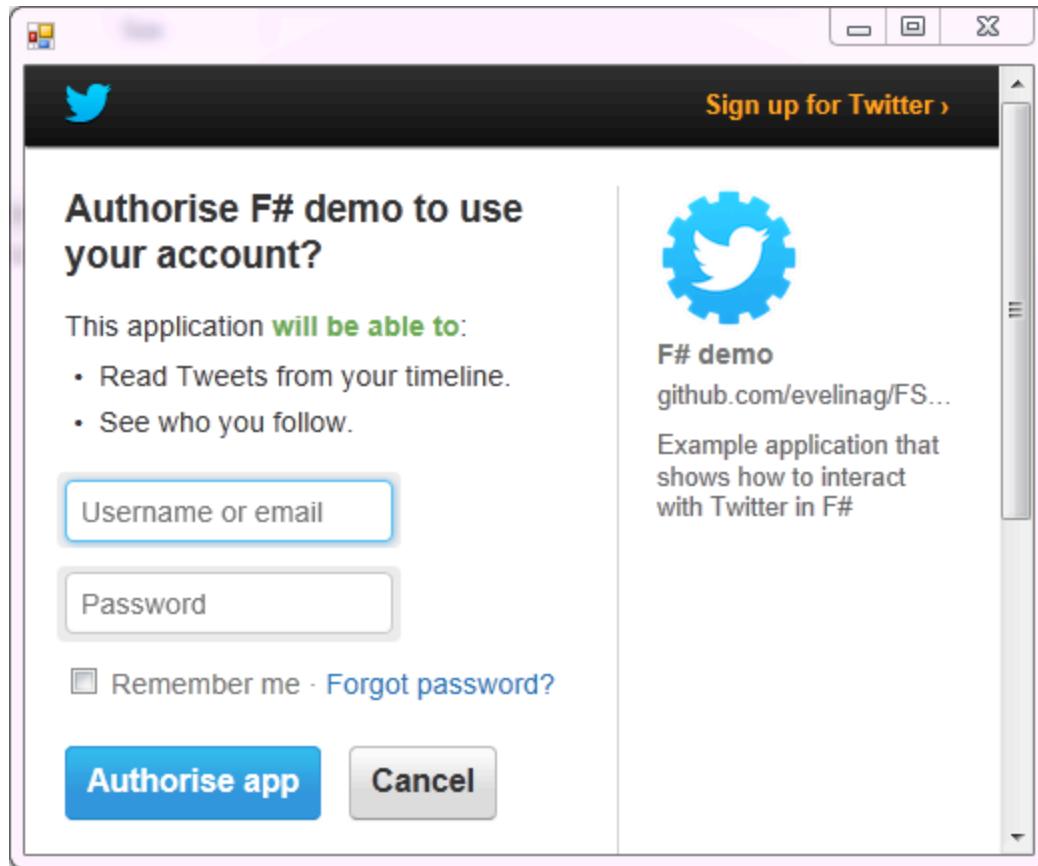


The important point is that the secret key (critical to reconstructing the hash) is known only to the sender and receiver. The secret key is not included in the request.

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 to use OAuth 2.0. This approach relies upon 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.



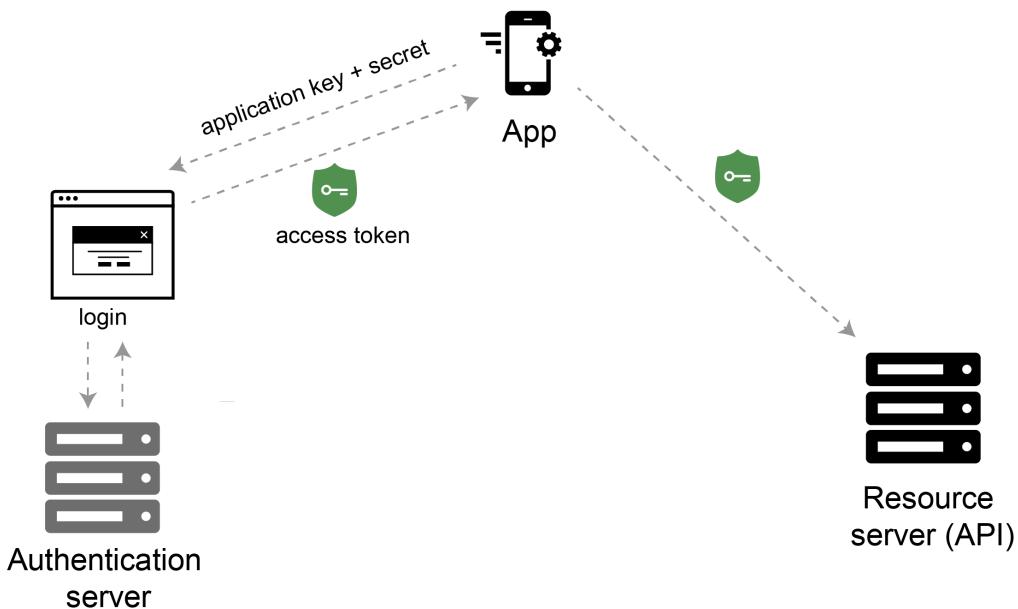
There are a few varieties of OAuth — namely, “one-legged OAuth” and “three-legged OAuth.” One-legged OAuth is used when you don’t have sensitive data to secure. This might be the case if you’re just retrieving general, read-only information (such as news articles).

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



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:

- Peter Udemy's course [API technical writing on Udemy](#)
- [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.

Where to list the API keys section in documentation

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.

Here's a screenshot from SendGrid's documentation on API keys:

The screenshot shows the SendGrid documentation website. The top navigation bar includes links for DOCS HOME, USER GUIDE, CODE WORKSHOP, GLOSSARY, SUPPORT, and a Contact Us button. A search bar is also present. The left sidebar has a 'User Guide' section expanded, showing options like SendGrid Overview, Dashboard, Email Activity, About the User Guide, Setting Up Your Server, SendGrid for Mobile, and Marketing Emails. The main content area is titled 'API Keys'. It explains that API Keys are used for authentication and provides definitions for Name, API Key ID, and Action. A 'Create an API Key' section is shown with a note about naming the key. The URL in the browser's address bar is Documentation > User Guide > Settings > API Keys.

Include information on rate limits

Whether in the authorization keys or another section, you should list any applicable rate limits to the API calls. Rate limits determine how frequently you can call a particular endpoint. Different tiers and licenses may have different capabilities or rate limits.

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.

Here's a great example of the rate limits section from the Github API:

Rate Limiting

For requests using Basic Authentication or OAuth, you can make up to 5,000 requests per hour. For unauthenticated requests, the rate limit allows you to make up to 60 requests per hour. Unauthenticated requests are associated with your IP address, and not the user making requests. Note that [the Search API has custom rate limit rules](#).

You can check the returned HTTP headers of any API request to see your current rate limit status:

```
$ curl -i https://api.github.com/users/whatever
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
```

The headers tell you everything you need to know about your current rate limit status:

Header Name	Description
X-RateLimit-Limit	The maximum number of requests that the consumer is permitted to make per hour.
X-RateLimit-Remaining	The number of requests remaining in the current rate limit window.
X-RateLimit-Reset	The time at which the current rate limit window resets in UTC epoch seconds .

If you need the time in a different format, any modern programming language can get the job done.

Documenting status and error codes

Status and error codes show 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 need documentation as much as custom status and error codes specific to the API. Error codes in particular help with troubleshooting bad requests.

Example of status and error codes

Here's an example of a status and error codes section in Flickr's API:

The screenshot shows a portion of the Flickr API documentation. At the top, there are navigation links: Sign Up, Explore, Create, Upload, and Photos. Below these, a search bar is visible. The main content area has a dark background with white text. A large block of text is partially visible at the top, mentioning photo responses and gallery creators. Below this, a section titled "Error Codes" is shown in pink text. This section lists several error codes with their descriptions:

- 100: Invalid API Key**
The API key passed was not valid or has expired.
- 105: Service currently unavailable**
The requested service is temporarily unavailable.
- 106: Write operation failed**
The requested operation failed due to a temporary issue.
- 111: Format "xxx" not found**
The requested response format was not found.
- 112: Method "xxx" not found**
The requested method was not found.
- 114: Invalid SOAP envelope**
The SOAP envelope send in the request could not be parsed.
- 115: Invalid XML-RPC Method Call**
The XML-RPC request document could not be parsed.
- 116: Bad URL found**
One or more arguments contained a URL that has been used for abuse on Flickr.

At the bottom of the visible section, there is a link to "API Explorer" and a specific endpoint: "API Explorer : flickr.galleries.getPhotos".

Here you can see that mostly the status and error codes are not the standard codes common to all requests (like 200), but are rather codes that are specific to Flickr's API.

Here's an example from OpenSecret's API:

Status Codes

101:	Switching Protocols
200:	OK
201:	Created
202:	Accepted
203:	Non-Authoritative Information
204:	No Content (may not contain a message body)
205:	Reset Content (may not contain a message body)
206:	Partial Content
300:	Multiple Choices
301:	Moved Permanently
302:	Found
303:	See Other
304:	Not Modified
305:	Use Proxy
307:	Temporary Redirect
400:	Bad Request (invalid syntax do not repeat request)
401:	Unauthorized
402:	Payment Required
403:	Forbidden
404:	Not Found
405:	Method Not Allowed
406:	Not Acceptable

Honestly, I've never seen so many status and error codes listed. But I think it's great to document this information if it's relevant to the API. (Obviously, if users would rarely encounter a particular status code, don't include it.)

The status and error codes section of an API is often the same across most resources. As a result, this section might appear outside the reference topics. But as a convenience, the status and error codes are often embedded in each reference topic as well, with the assumption that developers need quick access to these codes as they're working with the endpoints.

In the Clearbit API, the error codes appears as its own topic:

HTTP Status codes			
	Code	Title	Description
API Reference	200	OK	The request was successful.
Authentication	201	Created	The resource was successfully created.
Errors			
HTTP Status codes	202	Async created	The resource was asynchronously created
Error types	400	Bad request	Bad request
Rate limiting	401	Unauthorized	Your API key is invalid.
Versioning	402	Over quota	Over plan quota on this endpoint.
Webhooks	404	Not found	The resource does not exist.
Streaming	422	Validation error	A validation error occurred.
OAuth			
Enrichment API	50X	Internal Server Error	An error occurred with our API.
Discovery API			

Again, although many codes are standard, some unique codes specific to the Clearbit API are highlighted, such as 402: “Over plan quota on this endpoint.”

Finally, here’s an example of status and error codes from Dropbox’s API:

OAuth 1.0	Errors are returned using standard HTTP error code syntax. Any additional info is included in the body of the return call, JSON-formatted. Error codes not listed here are in the API methods listed below.	
/request_token	Standard API errors	
/authorize		
/access_token		
OAuth 2.0		
/authorize	Code	Description
/token	400	Bad input parameter. Error message should indicate which one and why.
/token_from_oauth1	401	Bad or expired token. This can happen if the user or Dropbox revoked or expired an access token. To fix, you should re-authenticate the user.
Access tokens	403	Bad OAuth request (wrong consumer key, bad nonce, expired timestamp...). Unfortunately, re-authenticating the user won't help here.
/disable_access_token	404	File or folder not found at the specified path.
Dropbox accounts	405	Request method not expected (generally should be GET or POST).
/account/info	429	Your app is making too many requests and is being rate limited. 429s can trigger on a per-app or per-user basis.
Files and metadata	503	If the response includes the Retry-After header, this means your OAuth 1.0 app is being rate limited. Otherwise, this indicates a transient server error, and your app should retry its request.
/files (GET)	507	User is over Dropbox storage quota.
/files_put	5xx	Server error. Check DropboxOps .
OAuth 1.0		

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 we submitted the curl call back in [an earlier lesson \(page 54\)](#)? In order to get the response header, we need to add `--include` or `-i` to the curl request. If we want *only the response header returned*, we capitalize it: `-I`.

```
curl -I -X GET "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&appid=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%2Cus
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. (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.

Common status codes follow standard protocols

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.

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. In addition to the Clearbit example mentioned above, Twitter's API also has a good example of the possible status and error codes you will receive when making requests:

The screenshot shows the Twitter Developers Documentation Overview page. The left sidebar contains links to Documentation, Best Practices, API Overview, Object: Users, Object: Tweets, Object: Entities, Object: Entities in Objects, Object: Places, Twitter IDs, Connecting to Twitter API using SSL, Using cursors to navigate collections, Error Codes & Responses, and Twitter Libraries. The main content area is titled "Error Codes & Responses" and "HTTP Status Codes". It states: "The Twitter API attempts to return appropriate [HTTP status codes](#) for every request." A table lists 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. In API v1.1, requests without authentication are considered invalid and will yield this response.

At the bottom of the page, there is a link to "Authentication credentials".

This Response Codes page stands alone rather than appearing embedded within each API reference topic. However, either location has merits. A standalone page allows you to expand on each code with more detail without crowding out the other documentation. It also reduces redundancy and the appearance of a heavy amount of information (information which is actually just repeated).

On the other hand, if some resources are prone to triggering certain status and error codes more than others, it makes sense to highlight those status and error codes on the relevant API reference pages. Perhaps you could call attention to any particularly relevant status or error codes, and then link to the centralized 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 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 to see all the potential error codes.

For example, if you exceed the rate limit 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

Your list of status codes can be done 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 codes aren't readily visible

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. All too often, status codes are uninformative, poorly written, and communicate little or no helpful information to the user to overcome the error.

Status/error codes can assist in troubleshooting

Status and error codes can be particularly helpful when it comes to troubleshooting. Therefore, 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. Document what happens when users get off the happy path and start stumbling around in the dark forest.

A section on troubleshooting could list possible error messages users get when they do any of the following:

- 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.

Documenting code samples and tutorials

As you write documentation for developers, you'll start to include more and more code samples. You might not include these more detailed code samples 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 tasks, you'll end up leveraging different endpoints and showing how to address a variety of scenarios.

The following sections list some best practices around code samples.

Code samples are like candy for developers

Code samples play an important role in helping developers use an API. No matter how much you try to explain and narrate *how*, it's only when you *show* something in action that developers truly get it.

Recognize that, as a technical writer rather than a developer, you aren't your audience. Developers aren't newbies when it comes to code. But different developers have different specializations. Someone who is a database programmer will have a different skill set from a Java developer who will have a different skillset from a JavaScript developer, and so on.

Developers often make the mistake of assuming that their developer audience has a skill set similar to their own, without recognizing different developer specializations. Developers will often say, "If the user doesn't understand this code, he or she shouldn't be using our API."

It might be important to 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.

Focus on the why, not the what

In any code sample, you should 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.

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 resources. In the request, you submit the authorization through the header rather than including the API key directly in the endpoint path.

Explain your company's code, not general coding

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. You shouldn't write your own version of another service's documentation. 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).

Keep code samples simple

Code samples should be stripped down and as simple as possible. Providing code for an entire HTML page is probably unnecessary. But including it doesn't hurt anyone, and for newbies it can help them see the big picture.

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.

When developers take the code and integrate it into a production environment, they will make a lot of changes to account for scaling, threading, and efficiency, and other production-level factors.

Add both code comments and before-and-after explanations

Your documentation regarding the code should mix code comments with some explanation either after or before the code sample. Brief code comments are set off with forward slashes `//` in the code; longer comments are set off between slashes with 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 later.

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.

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.

Make sure that the code works. When I first used some sample `ajax` code, 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. This allows you to spot errors, understand whether all the parameters are complete and valid, and more. Usually you just need a sample like this to get started, and then you can use the same pattern to plug in different endpoints and parameters. You don't need to come up with new code like this every time.

Provide a sample in your target language

With REST APIs, developers can use pretty much any programming language to make the request. Should you show code samples that span across various languages?

Providing code samples is almost always a good thing, so if you have the bandwidth, follow the examples from Evernote and Twilio. However, providing just one code example in your audience's target language is probably enough, if needed at all. You could also skip the code samples altogether, since the approach for submitting an endpoint follows a general pattern across 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.

Code samples require heavy maintenance with new releases

Getting into code samples leads us more toward user guide tasks than reference tasks. However, keep in mind that code samples are a bear to maintain. When your API pushes out a new release, will you check all the code samples to make sure the code doesn't break with the new API (this is called regression testing by QA).

What happens if new features require you to change your code examples? The more code examples you have, the more maintenance they require.

General code samples

Although you could provide general code samples for every language with every call, it's usually not done. Instead, there's often a page that shows how to work with the code in various languages. For example, with the Wunderground Weather API, they have a page that shows general code samples:

The screenshot shows a web browser displaying the Wunderground Weather API documentation. The top navigation bar includes links for Weather, Maps & Radar, Severe Weather, Photos & Video, Community, News, and Climate, along with a sign-in link. Below the navigation is a blue header bar with the word "DOCUMENTATION". Underneath is a navigation menu with links for API Home, Pricing, Featured Applications, Documentation (which is currently selected), and Forums. On the left, there's an "API Table of Contents" sidebar listing categories like Weather API, Data Features, and Forecast. The main content area has a large title "Code Samples". Under "PHP", there is a code block:

```
<?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";
?>
```

Under "Ruby", there is another code block:

```
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
```

To the right of the main content area is a "Page Contents" sidebar with links for Code Samples, PHP, Ruby, Python, ColdFusion, and JavaScript with jQuery.

The OpenWeatherMap API has a number of [example integrations](#) that provide more advanced user interfaces with weather.

Create a code sample for the surfreport endpoint

As a technical writer, add a code sample to the `surfreport/{beachId}` endpoint that you're documenting. Use the same code as above, and add a short description about why the code is doing what it's doing.

Here's my approach. (You might not include a detailed code sample like this for just one endpoint, but including some kind of code sample is almost always helpful.)

Code example

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="http://code.jquery.com/jquery-2.1.1.min.js"></script>
<script>
var settings = {
    "async": true,
    "crossDomain": true,
    "url": "http://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.

How to create the quick reference guide

For those power users who just want to glance at the content to understand it, provide a quick reference guide. The quick reference guide serves a different function from the getting started guide. The getting started guide helps beginners get oriented; the quick reference guide helps advanced users quickly find details about endpoints and other API details.

Sample quick reference guide

Here's a quick reference guide from Eventful's API:

Technical Reference	All API Methods
API documentation	Events
XML feed dictionary	/events/new Add a new event record.
Interface libraries	/events/get Get an event record.
Output format options	/events/modify Modify an event record.
FAQ	/events/withdraw Withdraw (delete, remove) an event.
Wise words	/events/restore Restore a withdrawn event.
"The only way of discovering the limits of the possible is to venture a little way past them into the impossible." - Arthur C. Clarke	/events/search Search for events.
	/events/reindex Update the search index for an event record.
	/events/ical Get events in iCalendar format.
	/events/rss Get events in RSS format.
	/events/tags/list List all tags attached to an event.
	/events-going/list List all users going to an event.
	/events/tags/new Add tags to an event.
	/events-tags/remove Remove tags from an event.
	/events/comments/new Add a comment to an event.
	/events/comments/modify

An online quick reference guide can serve as a great entry point into the documentation. Here's a quick reference from Shopify about using Liquid:

The screenshot shows the Shopify Cheat Sheet interface. At the top, there's a header with the title "Shopify Cheat Sheet" and a "follow on twitter" button. Below the header, there are three main sections:

- Liquid** (left column):
 - Logic**: Includes logic operators like {%- comment %}, {%- raw %}, {%- if %}, {%- unless %}, {%- case %}, {%- cycle %}, {%- for %}, {%- tablerow %}, {%- assign %}, {%- capture %}, and {%- include %}.
 - Operators**: Shows operators like ==, !=, >, <, >=, <=, or, and, contains.
- Liquid Filters** (middle column):
 - Includes filters like 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), and remove(input, substring).
- Template variables** (right column):
 - blog.liquid**: blog['the-handle'].variable, blog.id, blog.handle, blog.title, blog.articles, blog.articles_count, blog.url, blog.comments_enabled?, blog.moderated?, blog.next_article, blog.previous_article, blog.all_tags, blog.tags.
 - article.liquid**: article.id, article.title, article.author.

Visual quick reference guides

You can also make a visual illustration showing the API endpoints and how they relate to one another. I once created a one page endpoint diagram at Badgeville, and I found it so useful I ended up taping it on my wall. Although I can't include it here for privacy reasons, the diagram depicted the various endpoints and methods available to each of the resources (remember that one resource can have many endpoints).

Next phase of course

Congratulations, you finished the documenting REST APIs section of the course. You've learned the core of documenting REST APIs. We haven't covered publishing tools or strategies. Instead, this part of the course has focused on the creating content, which should always be the first consideration.

Summary of what you learned

During this part of the course, you learned the core tasks involved in documenting REST APIs. First, as a developer, you did the following:

- How to make calls to an API using curl and Postman
- How to pass parameters to API calls
- How to inspect the objects in the JSON payload
- How to use dot notation to access the JSON values you want
- How to integrate the information into your site

Then you switched perspectives and approached APIs from a technical writer's point of view. As a technical writer, you documented each of the main components of a REST API:

- Resource description
- Endpoints
- Parameters
- Request example
- Response example
- Code example
- Status codes

Although the technology landscape is broad, and there are many different technology platforms, languages, and code bases, most REST APIs have these same sections in common.

More practice with API requests and responses

If you'd like to get more practice making requests to APIs and doing something with the response (even just printing it to the page), check out the additional tutorials in the [Resources \(page 458\)](#) section:

- [Overview for exploring other REST APIs \(page 466\)](#)
- [EventBrite example: Get event information \(page 467\)](#)
- [Flickr example: Retrieve a Flickr gallery \(page 474\)](#)
- [Klout example: Retrieve Klout influencers \(page 483\)](#)
- [Aeris Weather Example: Get wind speed \(page 493\)](#)

The next part of the course

Now that you've got the content down, the next step is to focus on [publishing strategies for API documentation \(page 192\)](#). This is the focus of the next part of the course.

Publishing your API documentation

Overview for publishing API docs.....	192
List of 100 API doc sites.....	195
Design patterns with API doc sites.....	199
Docs-as-code tools	209
More about Markdown	215
Version control systems	224
Publishing tool options for developer docs.....	227
Which tool to choose for API docs — my recommendations	249
Activity: Manage content in a GitHub wiki.....	253
Activity: Use the GitHub Desktop Client.....	259
Pull request workflows through GitHub.....	268
Jekyll publishing tutorial	274
Case study: Switching tools to docs-as-code	282

Overview for publishing API docs

In earlier parts of this course, I used a [Weather API from OpenWeatherMap](#) to demonstrate how to use a REST API. Now I'll explore various tools to publish API documentation, also using this same OpenWeatherMap API as an example.

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](<http://idratherbewriting.com/2018/01/19/wtd-south-bay-publish-api-documentation-presentation/>) on this topic, you can view it here:

(For more details about this post, see my [writeup here](#).)

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?

This is a valid question that I want to answer by telling a story. 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 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 (so far, [CouchDB](#) is the only one I've come across).

I asked a recruiter (who specializes in API documentation jobs in the Bay area) whether it was more advantageous to become adept with DITA or to learn a tool such as a static site generator, which is more common in the API space.

My recruiter friend knows the market — especially the Silicon Valley market — extremely well. Without hesitation, he urged me to pursue the static site generator route. 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. Here are 5 reasons:

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 \(page 209\)](#), committing it to [version control \(page 224\)](#), 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.

Why are engineers writing documentation in the first place, you might ask? Well, sometimes you really need engineers to contribute because the content is so technical, it's beyond the domain of non-specialists. If you want engineers to get involved, especially to write, you need to use developer tooling.

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 [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 \(page 83\)](#).)

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 \(page 298\)](#) 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 \(page 207\)](#) 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 \(page 355\)](#).) 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 [Tripahe help and PDF files: past their prime?](#) from Robert Desprez.)

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 awesome.

Most tripahe 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.

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 \(page 274\)](#). 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 legacy content 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 \(page 209\)](#), but there are [many different options and routes \(page 195\)](#) 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.

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](#)

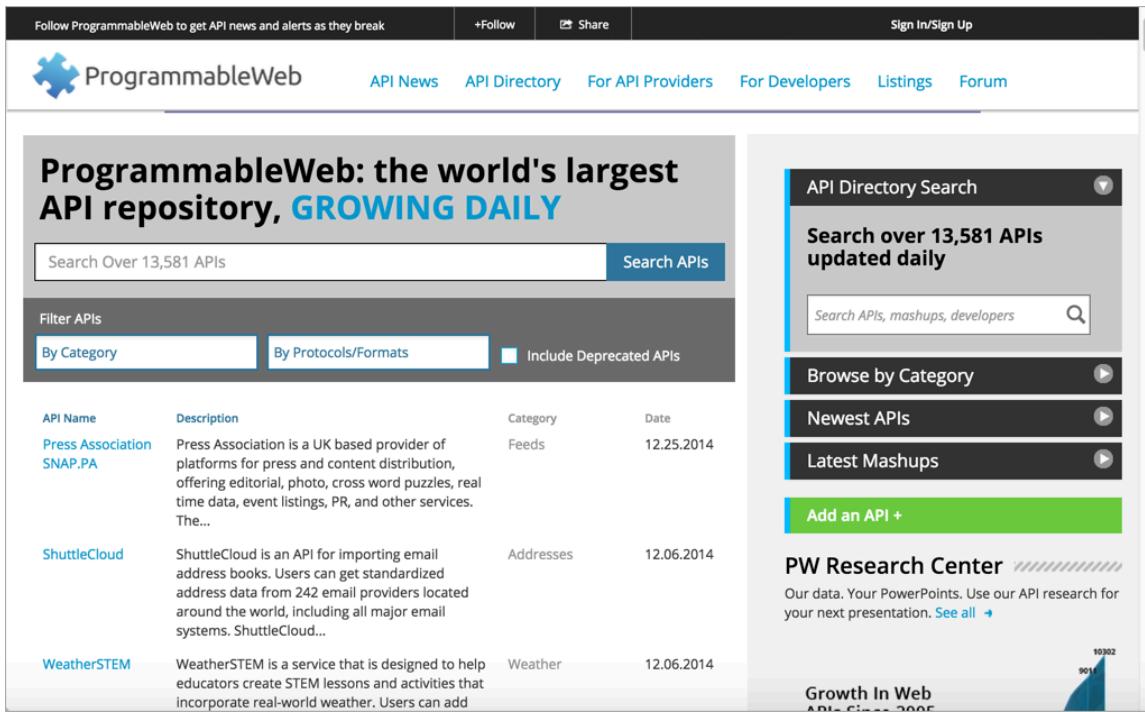
36. Callfire API docs
37. Reddit API docs
38. Netvibes API docs
39. Rhapsody API docs
40. Donors Choose docs
41. Sendgrid API docs
42. Photobucket API docs
43. Mailchimp docs
44. Basecamp API docs
45. Smugmug API docs
46. NYTimes API docs
47. USPS API docs
48. NWS API docs
49. Evernote API docs
50. Stripe 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](#)

Tip: I last checked these links in January 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.



The screenshot shows the ProgrammableWeb homepage. At the top, there's a navigation bar with links for "Follow ProgrammableWeb to get API news and alerts as they break", "+Follow", "Share", and "Sign In/Sign Up". Below the navigation is the ProgrammableWeb logo and a menu with links for "API News", "API Directory", "For API Providers", "For Developers", "Listings", and "Forum".

ProgrammableWeb: the world's largest API repository, GROWING DAILY

Search Over 13,581 APIs

Filter APIs Include Deprecated APIs

API Name	Description	Category	Date
Press Association	Press Association is a UK based provider of platforms for press and content distribution, offering editorial, photo, cross word puzzles, real time data, event listings, PR, and other services. The...	Feeds	12.25.2014
SNAP.PA			
ShuttleCloud	ShuttleCloud is an API for importing email address books. Users can get standardized address data from 242 email providers located around the world, including all major email systems. ShuttleCloud...	Addresses	12.06.2014
WeatherSTEM	WeatherSTEM is a service that is designed to help educators create STEM lessons and activities that incorporate real-world weather. Users can add	Weather	12.06.2014

API Directory Search

Search over 13,581 APIs updated daily

Search APIs, mashups, developers

Browse by Category

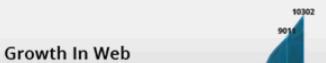
Newest APIs

Latest Mashups

Add an API +

PW Research Center 

Our data. Your PowerPoints. Use our API research for your next presentation. [See all →](#)

Growth In Web APIs since 2005 

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



Look at about 5 different APIs (choose any of those listed on the page). Look for one thing that the APIs have in common. I provide a list of patterns in the next topic: [Design patterns with API doc sites \(page 199\)](#).

Design patterns with API doc sites

In the previous topic, we browsed through a long [survey of API doc sites \(page 195\)](#) and looked for similar patterns in their design. “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, website platforms, 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 \(page 83\)](#). The [non-reference topics \(page 159\)](#) also share similar topics, which I touched upon earlier as well.

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. In [Jekyll \(page 274\)](#), a static site generator, you can create values in a [YAML file \(page 384\)](#) 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 \(page 278\)](#) 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 `````. `{: .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="endpo
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:

```

[{"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 imported into Drupal. The import process populated various fields in the 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.

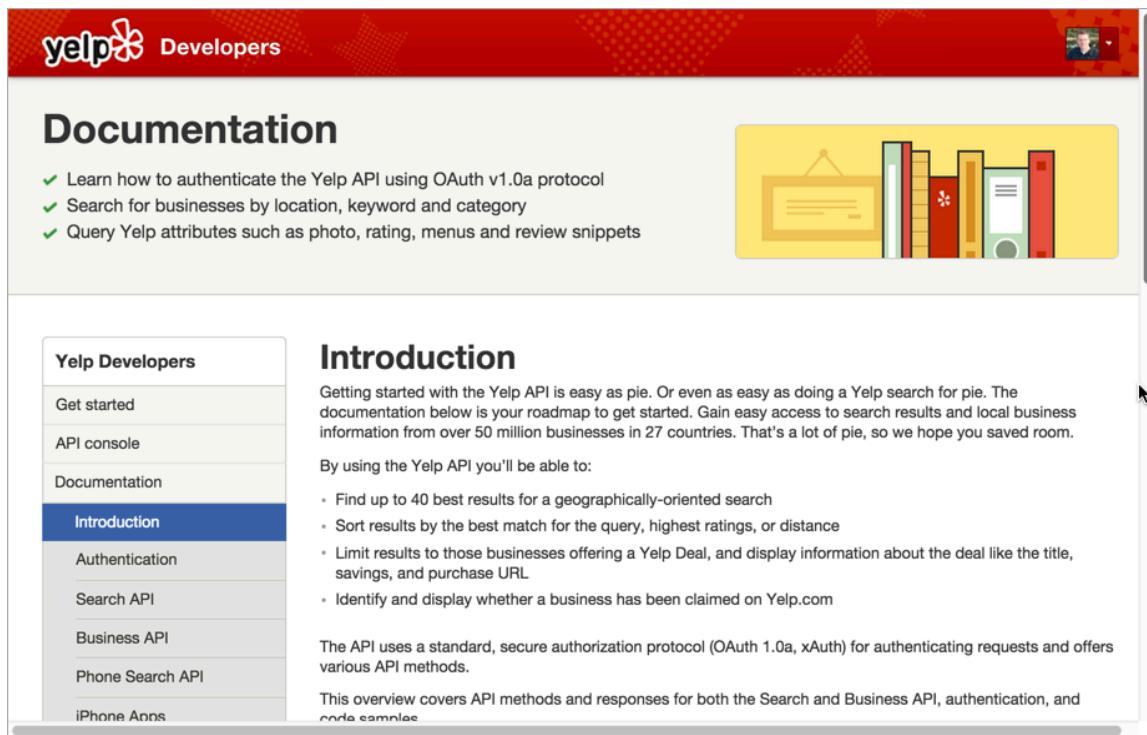
Also note that you don't have to create your own templates to display API documentation. You can probably already see problems related to custom templates. The templates are entirely arbitrary, with terms and structure based on the designer's perceived needs and styles. If you write documentation to 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. Later in this course, I explain several [REST API description formats \(page 296\)](#), and then launch into an extensive tutorial for the [OpenAPI specification \(page 310\)](#). Afterwards, I provide a tutorial for reading the OpenAPI specification using [Swagger UI \(page 355\)](#), along with an activity to [create your own Swagger UI \(page 363\)](#).

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.

Pattern 2: A website platform

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:

A screenshot of the Yelp Developers Documentation page. The header features the Yelp logo and 'Developers'. On the left is a sidebar with a navigation menu:

- Get started
- API console
- Documentation
- Introduction** (selected)
- Authentication
- Search API
- Business API
- Phone Search API
- iPhone Apps

The main content area has a title 'Introduction' with a sub-section 'Getting started with the Yelp API is easy as pie. Or even as easy as doing a Yelp search for pie. The documentation below is your roadmap to get started. Gain easy access to search results and local business information from over 50 million businesses in 27 countries. That's a lot of pie, so we hope you saved room.' Below this is a list of what you can do with the API, followed by a note about OAuth authentication and a summary of the API methods covered.

I hinted at this earlier, but with API documentation, there isn't an application 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 \(page 298\)](#) 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 \(page 212\)](#); 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 blessing and a challenge. A blessing 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 \(page 0\)](#), 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:

```

1 def getUserShardId(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 sidebar on the right includes links to API Reference, iOS SDK Reference, and Android SDK Reference. It also features 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).

The writers at Parse emphasize the importance of code samples in docs, giving 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.

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 compare the code samples against the guidelines in it. I dive [more into code samples \(page 0\)](#) 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 a [post on designing great API docs](#), the writers at Parse explain that short pages frustrate developers:

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.

The Backbone.js documentation takes this length to an extreme, publishing everything on one page:

The screenshot shows the Backbone.js API documentation. On the left, there's a sidebar with navigation links like "Backbone.js (1.0.0)", "GitHub Repository", "Annotated Source", "Introduction", "Upgrading", "Events", "Model", and a "Catalog of Built-in Events". The main content area features the Backbone.js logo and a brief description of what it does. It also includes links to GitHub, source code, test suites, examples, tutorials, and real-world projects, along with a note about the MIT license. Below that, there's information on reporting bugs and discussing features. A section at the bottom right says "Backbone is an open-source component of DocumentCloud." At the very bottom, there's a "Downloads & Dependencies" section with a "Right-click, and use 'Save As'" note.

For another example of a long page, see the Reddit API:

The screenshot shows the Reddit API documentation. The top navigation bar includes "MY SUBREDDITS", "FRONT", "ALL", "RANDOM", "FUNNY", "PICS", "AWW", "TODAYILEARNED", "GAMING", "VIDEOS", "GIFS", "NEWS", "ASKREDDIT", "WORLDNEWS", "SHOWERTHOUGHTS", "MILDLYINTEREST", and "EDIT". The main content area features the Reddit logo and a message about automatically-generated documentation. It includes a call to action for respecting API access rules. Below this, there's an "overview" section and a "listings" section. The "listings" section contains detailed information about how endpoints use pagination and filtering, mentioning "after", "before", "limit", and "count" parameters. There's also a note about "flair" and "captcha" methods.

Why do API doc sites tend to have such lengthy pages? Here are a few reasons:

- **Provides the big picture:** As the Parse writers indicate, 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 \(page 298\)](#) to provide the interactivity.

The screenshot shows a web-based API explorer. 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="↓"/>	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 \(page 54\)](#) or [Postman \(page 46\)](#) (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 \(page 245\)](#).

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 \(page 195\)](#), you rarely see any of the following:

- Video tutorials
- PDFs
- Commenting features
- Translated sites
- 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.

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 tool may not matter as much. But if developers will be contributing to the docs, you should 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 the 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

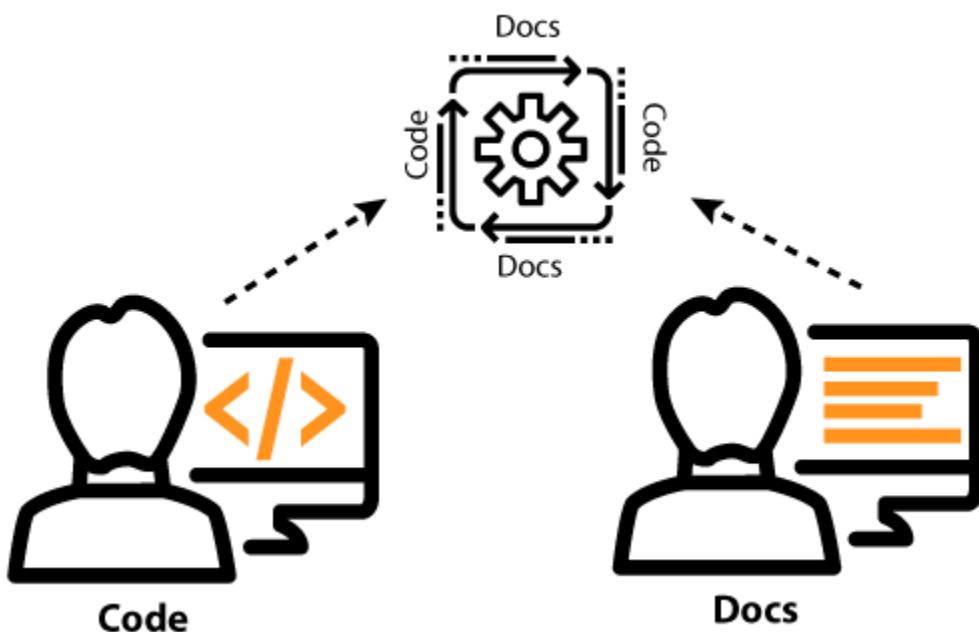
Having developers write or contribute to documentation should inform your tool choice with API documentation. If you plan to involve developers in writing and editing, you'll naturally choose more of a [docs-as-code tools \(page 209\)](#) approach. Docs-as-code means to embrace tools that treat docs 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 \(page 232\)](#), [Jekyll \(page 232\)](#), or [Hugo \(page 232\)](#) 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 an IDE** such as Atom, Sublime, or another text editor (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 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.

Granted, engineers who write documentation often fall prey to the [curse of knowledge](#). That is, the more you know about a topic, the more assumptions and background information you have that get 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 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 \(page 83\)](#), while technical writers focus more on the [non-reference documentation \(page 159\)](#). 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. 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 coded 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 \(page 282\)](#).

What about help authoring tools (HATS)

What about help authoring tools? Did I dismiss them too easily?

Help authoring tools (HATs) refer to the common toolset often used by technical writers. Common HATs include MadCap Flare, Adobe Robohelp, Author-it, and more. You can, in fact, use these tools to create API documentation, but almost no one does.

Some advantages of using a HAT might include the following:

- **Comfortable authoring environment for writers.** If writers will be creating and publishing the documentation, using a tool technical writers are familiar with is a good idea.
- **Handles the toughest tech comm scenarios.** When you have to deal with versioning, translation, content re-use, conditional filtering, authoring workflows, and PDF output, you're going to struggle to make this work with the other tools mentioned in this course.

However, with developer docs, HATs have more disadvantages and are much less common than docs-as-code tools. Here are a few reasons why.

Dated UI won't help sell the product

The output from a help authoring tool usually looks dated. Here's a sample help output from Flare for the Photobucket API:

The screenshot shows a web-based API documentation interface. On the left, there's a sidebar with a 'TOC' button and several links: 'Using the Help', 'What's New', 'Getting Started', 'Examples', 'Methods', 'Ping Photobucket', 'Get Timestamp', 'Web User Login', 'Non-Web User Login', 'Get Site Login Credentials', 'Albums' (which is expanded to show 'Upload Media to an Album', 'Get Album URL', 'Get Album', 'Create New Album', 'Rename Album', 'Delete Album', 'Get Album Organization', 'Set Album Organization', 'Follow an Album', 'Stop Following an Album', 'Get Album Following Stats', 'Get Album Privacy Setting', 'Update Album Privacy Setting', 'Get Album Vanity URL', 'Get Album Theme', and 'Share an Album'). Below this is another 'TOC' button, 'Index', 'Search', and 'Glossary'. The main content area has a header 'Create New Album' with a sub-header 'Create a new sub-album.' It says 'User Login Required' and provides a link to 'End-User Authentication'. It specifies the 'HTTP Method' as 'POST'. The 'REST Path' is '/album/{identifier}'. It includes a note about 'Object Identifiers'. A 'Parameters' table lists two columns: 'Parameter' (identifier, name) and 'Optional' (N, N). The 'Description' column contains 'Album identifier.' and 'Name of result. Must be between 2 and 50 characters. Valid characters are letters, numbers, underscore (_), hyphen (-), and space.'. The 'Variable' column shows 'String' for both. Below this is a 'Request Example' section with a code snippet:

```
POST http://api123.photobucket.com/album/pbapi
format=xml&name=thisisnewalbum&oauth_consumer_key=0000000000&oauth_nonce=ff84a82e65a6a517e753f5d34846c940&c
SHA1&oauth_timestamp=1236627874&oauth_token=20.000000_1236627874&oauth_version=1.0
```

There's also a 'Response Example' section.

The problem with the dated tripane look and feel is that API documentation *is* the product interface that users navigate. There isn't a separate GUI interface that the help opens up next to. The help is front and center as the information product that users purchase or use.

If you want to promote the idea that your API is modern and awesome, you want a website that looks modern and awesome. In fact, you might have a UX developer create the website itself. If you lead with an outdated tripane site that loads frames, developers may not be as excited to use your API.

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.

Additionally, many of the API doc sites are single-website experiences, or at least skinned to have similar branding as the main company site. The API docs are usually integrated as part of the main website, not a link that opens in its own window and frame, separate from the other content. If you split and divide the user into separate sites (with the HAT output looking notably different and dated), you're following a less common pattern with API doc sites. Because of the tight integration, skinning the output with the same modern web framework ranks as a top priority.

Removes authoring capability from developers

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. HATs are tools for writers, not developers. I covered this point earlier in [Collaboration with developers \(page 210\)](#), so I won't repeat the argument here. But in developer doc spaces, you collaborate heavily with engineers. As such, you need to use tools that engineers understand and can easily plug into.

Additionally, almost no HAT runs on a Mac. Many developers and designers prefer Macs because they have a much better development platform (the command line is much friendlier and functional, for example).

If most developers use Macs but you use a PC (to accommodate your HAT), you may struggle to install developer tools or to follow internal tutorials to get set up and test out content.

Additionally, HATs often have steep license restrictions, whereas docs-as-code tools are often open source and can therefore scale across the company without budgetary funding and approval.

Dealing with more challenging factors

A lot of the solutions we've looked at tend to break down when you start applying more difficult requirements in your tech comm scenario. You may have to resort to more traditional tech comm tooling if you have to deal with some of the following:

- Translation
- 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 will require a higher degree of technical skill and coding.

At one company where I used Jekyll, we had requirements around both PDF output and versioning. We singled sourced the content into about 8 different outputs (for different product lines and programming languages). It was double that number if you included PDF output for the same content.

Jekyll provides a templating language called Liquid that allows you to do conditional filtering, content re-use, variables, and more, so you can fill these more robust requirements. I used this advanced logic to single source the output without duplicating the content. Other static site generators (like Hugo or Sphinx) have similar templating and scripting logic that lets you accomplish advanced tasks.

To handle PDF with Jekyll, I integrated a tool called [Prince](#), which converts a list of HTML pages into a PDF document, complete with running headers and footers, page numbering, and other print styling. 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

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.

Sample syntax

Here's a sample to get a sense of the 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, fireStructuredText created Markdown (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](#)) 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?

Although you can also work with DITA in a text editor, it's a lot harder to read the code with all the XML tag syntax. For example, look at the tags required by DITA for a simple instruction about printing a page:

```
<task id="task_mhs_zjk_pp">
    <title>Printing a page</title>
    <taskbody>
        <steps>
            <stepsection>To print a page:</stepsection>
            <step>
                <cmd>Go to <menucascade>
                    <uicontrol>File</uicontrol><uicontrol>Print</uicontrol>
                </menucascade></cmd>
            </step>
            <step>
                <cmd>Click the <uicontrol>Print</uicontrol> button.</cmd>
            </step>
        </steps>
    </taskbody>
</task>
```

Now compare the same syntax with Markdown:

```
## Print a page

1. Go to **File > Print**.
2. Click the **Print** button.
```

I wrote about this 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 aren't 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.
- Most developers don't want to spend a lot of time in documentation, 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 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.

Drawbacks of Markdown

Markdown has 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 still render as valid even if the Markdown conversion to HTML is problematic. It can be difficult to catch all the errors.

Markdown and complexity

If you need more complexity than Markdown offers, a lot of tools will leverage other templating languages, such as [Liquid](#) or [Coffeescript](#). Many times these other processing languages 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

Analyzing a Markdown sample

Take a look at the following Github-Flavored-Markdown content. Try to identify the various Markdown syntax used.

```
# surfreport/{beachId}
```

Returns information about surfing conditions at a specific beach ID, including the surf height, water temperature, wind, and tide. Also provides an overall recommendation about whether to go surfing.

`{beachId}` refers to the ID for the beach you want to look up. All Beach ID codes are available from our site.

```
## Endpoint definition
```

```
'surfreport/{beachId}'
```

```
## HTTP method
```

```
<span class="label label-primary">GET</span>
```

```
## Parameters
```

Parameter	Description	Data Type
days *Optional*. The number of days to include in the response. Default is 3. integer		
units *Optional*. Whether to return the values in imperial or metric measurements. Imperial will use feet, knots, and fahrenheit. Metric will use centimeters, kilometers per hour, and celsius. string		
time *Optional*. If you include the time, then only the current hour will be returned in the response. integer. Unix format (ms since 1970) in UTC.		

```
## Sample request
```

```
```
```

```
curl -I -X GET "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

```
```
```

```
## Sample response
```

```
```json
```

```
{
```

```
 "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."
 }
 }
}
```
}

The following table describes each item in the response.
```

| Response item | Description |
|-----------------------------|--|
| **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. |
| **{day}** | The day of the week selected. A maximum of 3 days get returned in the response. |
| **{time}** | The time for the conditions. This item is only included if you include a time parameter in the request. |
| **{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. |
| **{day}/{time}/wind** | The wind speed at the beach, measured in knots per hour or kilometers per hour depending on the units you specify. Wind affects the surf height and general wave conditions. Wind speeds of more than 15 knots per hour make surf conditions undesirable, since the wind creates white caps and choppy waters. |
| **{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. |
| **{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 f |

```
eet is the minimum size needed for surfing. If the surf height exceeds 10 feet, it is not safe to surf. |
| **{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. |
```

Error and status codes

The following table lists the status and error codes related to this request.

| Status code | Meaning |
|-------------|--|
| 200 | Successful response |
| 400 | Bad request -- one or more of the parameters was rejected. |
| 4112 | The beach ID was not found in the lookup. |

Code example

The following code samples shows how to use the surfreport endpoint to get the surf conditions for a specific beach. In this case, the code is just showing the overall recommendation about whether to go surfing.

```
```html
```

# Sample Page

Surf report conditions:

```
...
```

In this example, the `ajax` method from jQuery is used because it allows cro

ss-origin resource sharing (CORS) for the weather resources. In the request, you submit the authorization through the header rather than directly in the endpoint path. The endpoint limits the days returned to 1 in order to increase the download speed.

For simple demo purposes, the response is assigned to the `data` argument of the success method, and then written out to the `output` tag on the page. We're just getting the surfing recommendation, but there's a lot of other data you could choose to display.

Look at about 5 different APIs (choose any of those listed on the page). Look for one thing that the APIs have in common.

## ACTIVITY



On your Github wiki page, edit the page and create the following:

- a numbered list
- a bulleted list
- a bold word
- a code sample with html highlighting
- a level 2 heading

## Limitations in Markdown

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.

If you're using a static site generator, see the markdown processor used to convert the Markdown into HTML. With Jekyll, the default Markdown processor 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, doesn't have a dozen Markdown flavors, 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. (redcarpet and pygments are the engines that render Markdown to HTML.) It was supposed to be a seamless backend switch that wouldn't require any adjustment to existing Markdown. However, I found that kramdown imposed different requirements around spacing that broke a lot of my content, particularly around lists. I wrote about this issue here: [Updating from redcarpet and Pygments to Kramdown and Rouge on Github Pages](#).

In many ways, my content 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, usually 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 are dwarfed by the OpenAPI spec.

Further, the OpenAPI spec 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 introduced a proposal for [Lightweight DITA](#), which would allow you to use Markdown and HTML in your DITA projects.

Right now, the proposal is still in the process of being considered and accepted. After its acceptance, tool vendors will need to build in the support. (OxygenXML already [provides support for Markdown](#).) In a few years, it will likely be more common for DITA systems to process Markdown. What flavor of Markdown will they adopt? GitHub-flavored Markdown.

# Version control systems

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 invariably 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, in this section, you'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. If 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 systems are probably Git and Mercurial. Largely due to the fact that GitHub provides repositories for free on the web, Git is the most common version control repository for web and open source projects, so we'll be focusing on it more. However, these two systems share the same concepts and workflows.

The screenshot shows the GitHub web interface. At the top, there's a search bar labeled "Search GitHub" and navigation links for "Pull requests", "Issues", and "Gist". Below the search bar is a "GitHub Bootcamp" guide with four numbered steps: 1. Set up Git, 2. Create repositories, 3. Fork repositories, and 4. Work together. Each step has a cartoon illustration and a brief description. To the right of the bootcamp is a notification box for "Easier feeds for GitHub Pages" with a link to "View 56 new broadcasts". Below the bootcamp are two recent comments from users "envygeeks" and "kenold" on an issue titled "jekyll/jekyll#3260". On the far right, there's a sidebar titled "Repositories you contribute to" listing "jekyll/jekyll-help" and "slashdotdash/jekyll-lunr-js-search".

Github's distributed version control system allows for a phenomenon called "social coding."

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 Altassian'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 that file 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.

# Publishing tool options for developer docs

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 three main groups:

- **Static site generators (page 227)**: Used to author content and build the web output.
- **Hosting and deployment options (page 237)**: Used to build, deploy, and host the web output.
- **CMS platforms (mostly headless CMSs) (page 242)**: Provides an online GUI for authoring/publishing. In many cases, content is stored in plain text files and pulled in from GitHub.

Note that the tools below are particularly useful for writing and deploying the [non-reference content \(page 159\)](#) in your project. For tools that will read an [OpenAPI specification document \(page 298\)](#) and generate interactive reference documentation, see [Other tools to parse and display OpenAPI specs \(page 382\)](#).

As explained in [Docs-as-code tools \(page 209\)](#), 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? \(page 192\)](#) for more background.

## Static site generators

### What are static site generators?

Static site generators (you can view a full list at [Staticgen.com](#)) are applications that run on the command line and compile a website. 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.

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. The entire website is fully built when the user arrives; nothing changes dynamically based on the user's profile (unless done with client-side JS).

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, only a handful of are probably relevant for documentation. I'll consider these three in this article:

- [Jekyll \(page 228\)](#)
- [Hugo \(page 230\)](#)
- [Sphinx \(page 232\)](#)

One could discuss many more — Hexo, Middleman, Gitbook, Pelican, and so on. But the reality is that these other static site generators aren't used that frequently for documentation projects.

## Jekyll

I devote an entire topic to [Jekyll \(page 274\)](#) 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 heading is "Transform your plain text into static websites and blogs." Below this, there are three columns: "Simple" (describing content-only sites), "Static" (describing sites built from Markdown or Liquid templates), and "Blog-aware" (describing sites with permalinks, categories, and custom layouts). Each column has a corresponding link: "How Jekyll works →", "Jekyll template guide →", and "Migrate your blog →". A "Quick-start Instructions" box contains the command-line steps: `~ $ 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 section titled "Free hosting with GitHub Pages" featuring the GitHub logo and a cartoon cat holding a smartphone. It explains that GitHub Pages are powered by Jekyll and offers free hosting with a custom domain. A yellow arrow points from the "Free hosting with GitHub Pages" section towards the bottom right of the page.

There are several compelling reasons to use Jekyll:

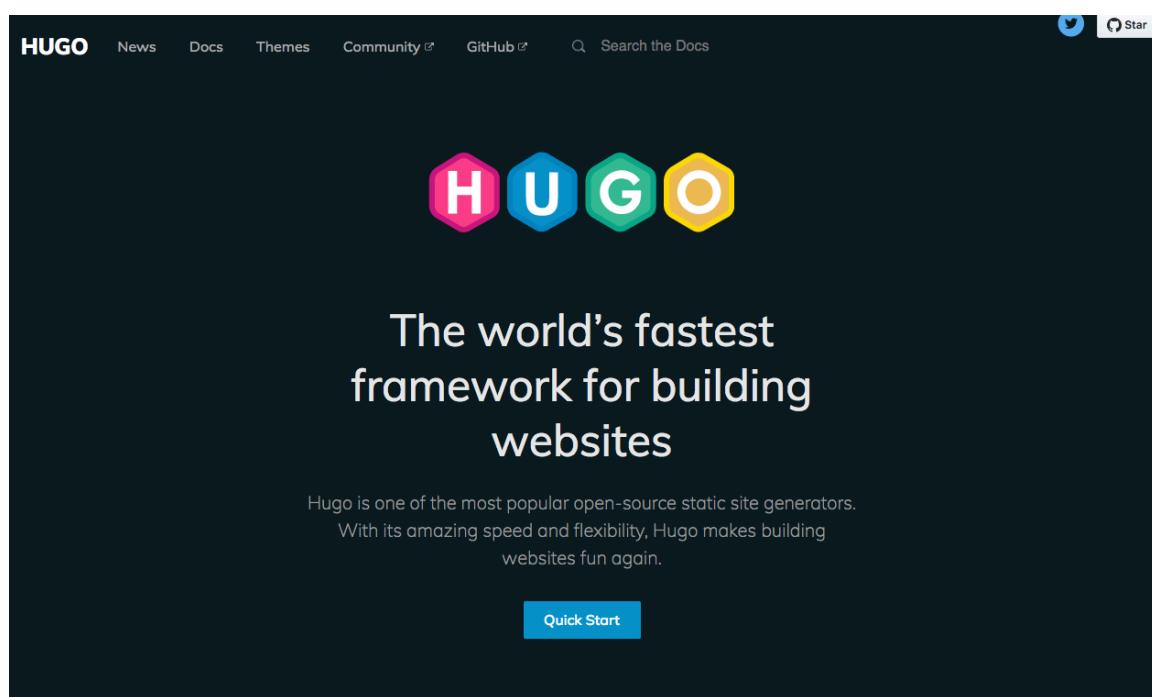
- **Large community.** The community, arguably the largest among static site generator communities, includes web developers, not just documentation-oriented groups. This broader focus attracts more developer attention and helps propel greater usage.
- **Control.** Jekyll provides a lot of powerful features (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 \(page 237\)](#) 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.

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 superiority 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.

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 \(page 282\)](#). I concluded that having a single, massive project is superior because it allows easier content re-use, onboarding, validation, and error checking, deployment management, and more.

If starting from scratch, I might use Hugo instead of Jekyll. However, I have a lot of custom scripting in Jekyll already (such as the ability to generate Kindle books and PDF), not to mention a publishing pipeline with Jekyll already integrated at the server level. However, given that content is largely in the same format (Markdown with YAML frontmatter), switching between the two platforms shouldn't be too difficult — though admittedly, I haven't tried it.

Also, 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:

```
jekeyll 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 highlights its use for Python projects and lists features like output formats (HTML, LaTeX, etc.), extensive cross-references, hierarchical structure, and automatic indices. A sidebar on the right contains a "What users say:" box with a quote, a "Download" section with links to Git and PyPI, a "Questions?" section with a mailing list link, and a "Suggestions?" section with a subscribe form.

**Welcome**

Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license.

It was originally created for [the Python documentation](#), and it has excellent facilities for the documentation of software projects in a range of languages. Of course, this site is also created from reStructuredText sources using Sphinx!

The following features should be highlighted:

- **Output formats:** HTML (including Windows HTML Help), LaTeX (for printable PDF versions), ePub, Texinfo, manual pages, plain text
- **Extensive cross-references:** semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information
- **Hierarchical structure:** easy definition of a document tree, with automatic links to siblings, parents and children
- **Automatic indices:** general index as well as a language-specific module indices
- **Code handling:** automatic highlighting using the [Pygments](#) highlighter
- **Extensions:** automatic testing of code snippets, inclusion of docstrings from Python modules (API docs), and [more](#)
- **Contributed extensions:** more than 50 extensions [contributed by users](#) in a second repository; most of them installable from PyPI

Sphinx uses [reStructuredText](#) as its markup language, and many of its strengths come from the power and straightforwardness of reStructuredText and its parsing and translating suite, the [Docutils](#).

**Documentation**

[First steps with Sphinx](#)  
overview of basic tasks

[Search page](#)  
search the documentation

**What users say:**

"Cheers for a great tool that actually makes programmers **want** to write documentation!"

**Download**

This documentation is for version [1.6.7+](#), which is not released yet.

You can use it from the [Git repo](#) or look for released versions in the [Python Package Index](#).

**Questions?**  
**Suggestions?**

Join the [sphinx-users](#) mailing list on Google Groups:

or come to the [#sphinx-doc](#) channel on FreeNode.

You can also open an issue at the [tracker](#).

Because Sphinx was designed from the ground up as a documentation tool, not just as 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? \(page 221\)](#) for more details around rST compared to Markdown.)

Sphinx can be used with the [Read the Docs \(page 240\)](#) platform and 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 generators.

As of January 2018, [Staticgen.com](#) shows the number of stars, forks, and issues as follows:

Jekyll			Hugo		
jekyllrb.com			gohugo.io/		
★	▶	⚡	★	▶	⚡
32768	7247	136	22463	2893	218
99 ▲	22 ▲	-2 ▼	183 ▲	16 ▲	11 ▲
A simple, blog-aware, static site generator.			A Fast and Flexible Static Site Generator.		
Language: Ruby Templates: Liquid License: MIT			Language: Go Templates: Go Templates License: APL 2.0		
 Deploy to netlify			 Deploy to netlify		

Sphinx		
www.sphinx-doc.org/		
★	▶	⚡
1567	684	711
17 ▲	2 ▲	-10 ▼
A tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl.		
Language: Python Templates: Jinja2 License: BSD		

The star icon represents the number of users who have "starred" the project (basically followed its activity).

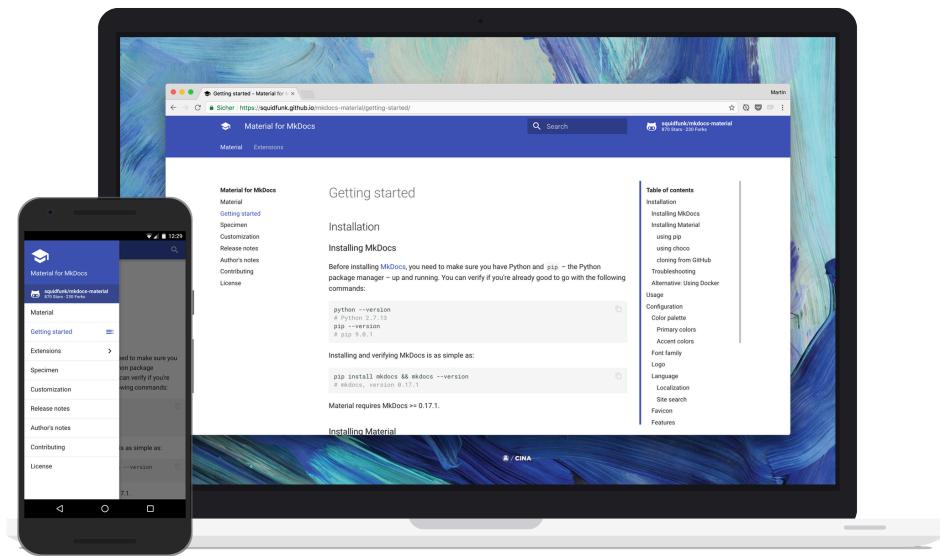
The branch icon represents the number of times the project has been branched (copied into another GitHub account). The bug icon represents the number of open issues logged against the project. The green numbers indicate trends with these numbers.

Jekyll and Hugo are the most common static site generators. Although I positioned the three graphics together here, if you look at [Staticgen.com](https://staticgen.com), you'll see that between Hugo and Sphinx, there are 22 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 \(page 240\)](#).

## Others

### MkDocs

**MkDocs** is a static site generator based on Python and designed for documentation projects. Similar to Jekyll, with MkDocs you write in Markdown, store page navigation in YAML files, and 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.

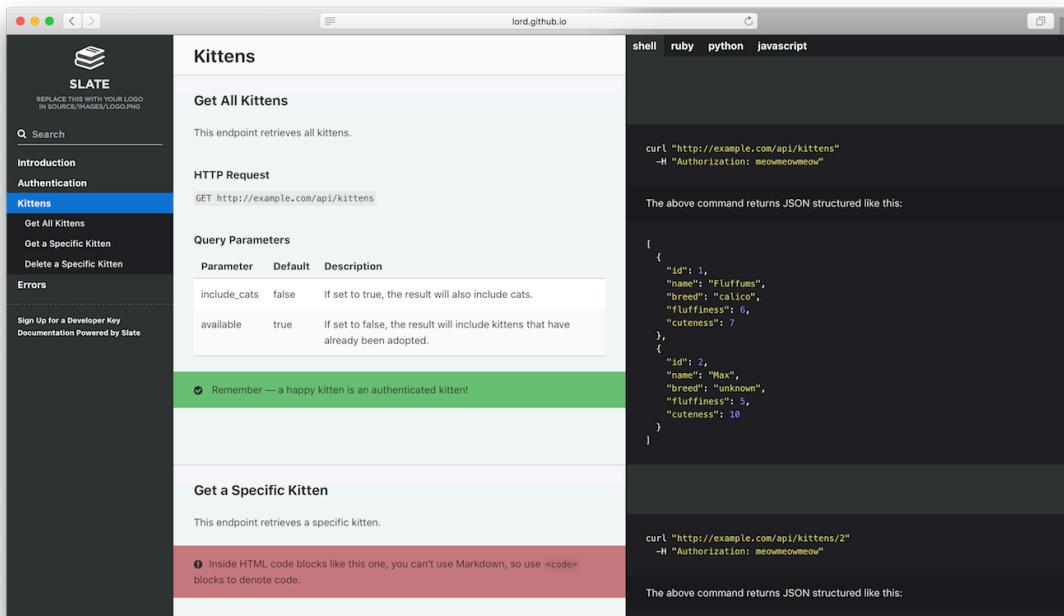
Although there are many static site generators with similar features, MkDocs is one more specifically oriented towards documentation. For example, it does include search.

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.

Right now there are probably many readers who are clenching their fist 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 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 \(page 0\)](#). 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 [though I offer more specific advice and recommendations here \(page 249\)](#). 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.”

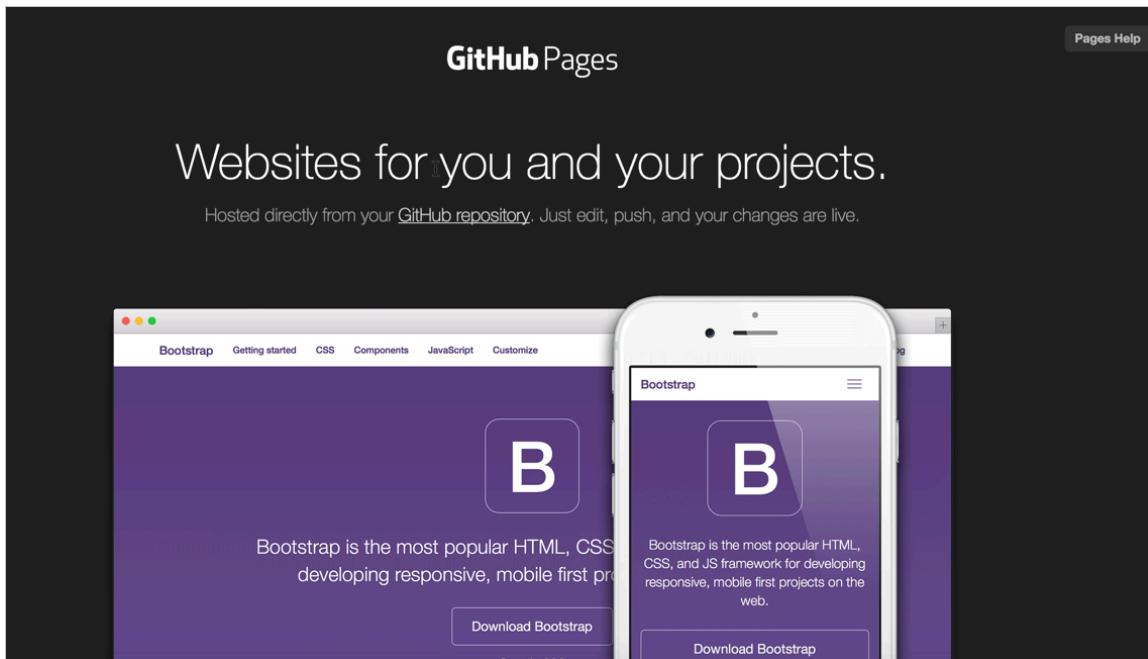
Theoretically, you could publish a static website on any web server (e.g., AWS S3, Bluehost, and more). But continuous delivery hosting platforms do something more — they automatically build your output when you commit a change to a repo. These platforms often read content stored on GitHub, sync it to 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:



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 sites are subject to the following usage limits:

- 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](https://tomjoht.github.io), named after my GitHub user name but published using a custom domain [idratherbewriting.com](http://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 <http://idratherbewriting.com/learnapidoc>. They seem like the same site, but they are actually separate projects in separate repos.

## 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](#)). Whereas GitHub Pages is based on Jekyll, Read the Docs is based on Sphinx.

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 has both an open-source, free version ([readthedocs.org](https://readthedocs.org)) and a commercial version ([readthedocs.com](https://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 [here \(page 221\)](#) 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 options: 'In reStructuredText' and 'In Markdown'. Below this is a section titled 'In reStructuredText' with a note about a screencast and a link to install Sphinx via pip.

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 include:

- 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
- Internationalization support. (— [An introduction to Sphinx and Read the Docs for Technical Writers](#))

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 \(page 244\)](#), a headless CMS for your content.

### 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 CMSSes

Finally, 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 CMSSes.”

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 a website titled "headless CMS". The page features a large orange header with the title "headless CMS" and a subtitle "A List of Content Management Systems for JAMstack Sites". Below the header is a dark navigation bar with links for "About", "Contribute", "What is JAMstack?", and "Contact". A "SHARE" button is located in the top right corner of the header area. The main content area contains a grid of eight cards, each representing a different CMS:

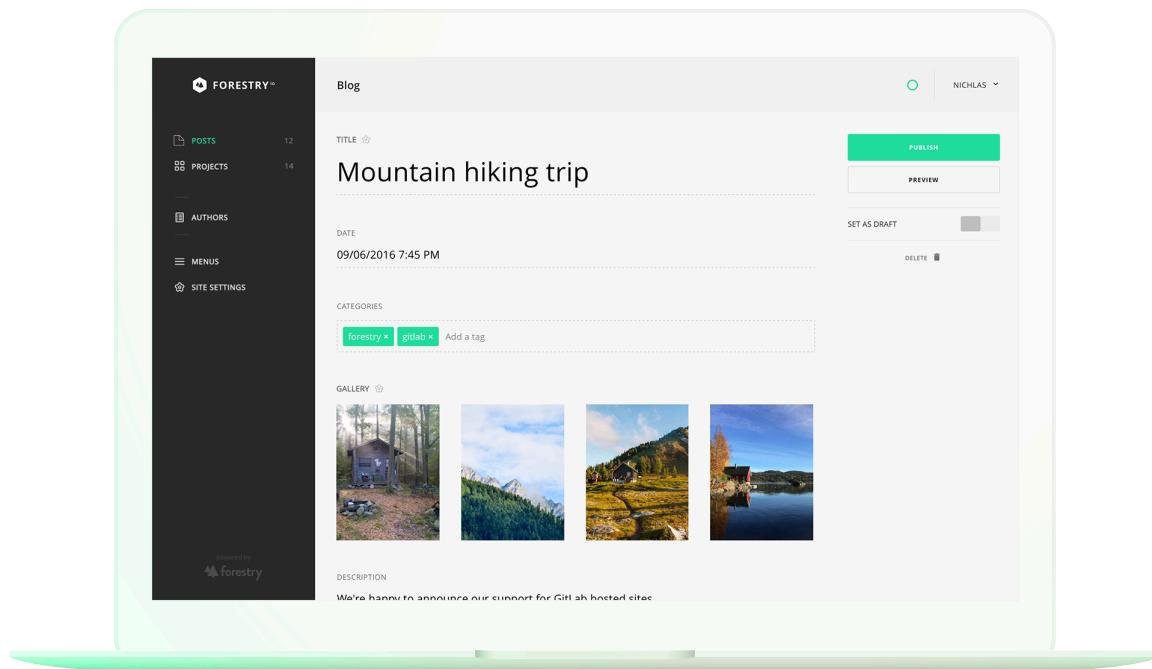
- Appernetic**: Type: Git-based. Supported Site Generators: Hugo.
- BowTie**: Type: Git-based. Supported Site Generators: Jekyll.
- Built.io**: Type: API Driven. Supported Site Generators: All.
- Butter CMS**: Type: API Driven. Supported Site Generators: All.
- CannerIO CMS**: CMS platform for developers, build any websites in only one schema! Canner help you de...
- Cloud CMS**: An API-first approach, built around JSON and a high performance cloud architecture.
- CloudCannon**: The Cloud CMS for Jekyll - build Jekyll or static websites, make updates inline.

On the right side of the grid, there is a sidebar with the text "Also visit [staticgen.com](#) for a ranked list of open-source site generators!" and a "Fork me on GitHub" button.

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 editing a blog post. On the left, the 'Writing in Post collection' section 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 preview shows the title 'A beginners' guide to brewing with Chemex', the date 'Wed, Jan 4, 2017', the reading time 'Read in x minutes', the intro blurb, the image, and the body text.

Their site says, "The web-based app includes rich-text editing, real-time preview, and drag-and-drop media uploads. ... Writers and editors can easily manage content from draft to review to publish across any number of custom content types."

Your content source can be stored in GitHub, GitLab, or BitBucket. Netlify CMS also integrates with [Netlify \(page 241\)](#), 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. 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:

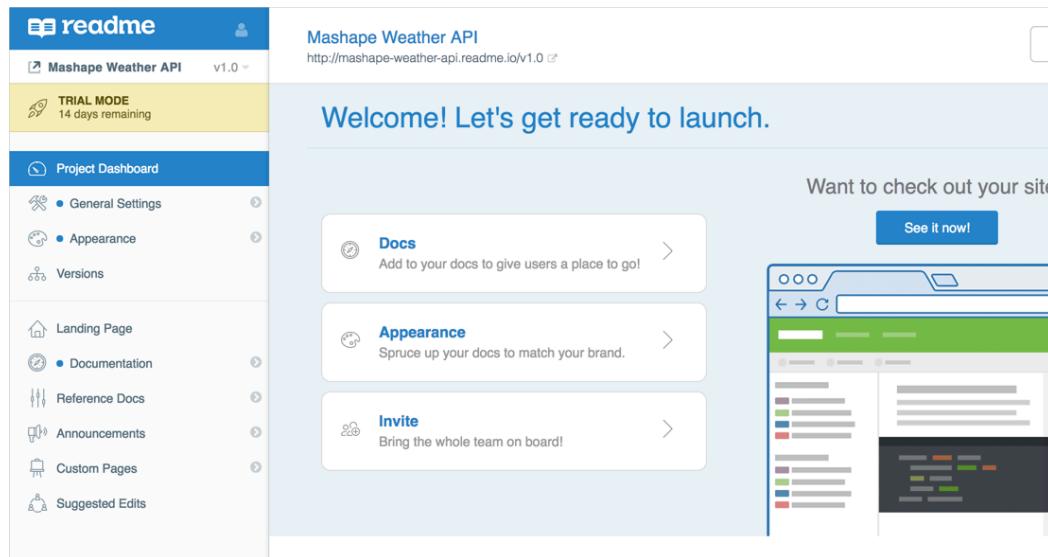
- Enable API Explorer
- Enable API Proxy
- Enable auto code samples

**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 \(page 298\)](#). 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 \(page 366\)](#).)

To explore Readme.io:

1. Go to [readme.io](#).
2. Click the **Sign Up** button in the upper-right corner and sign up for an account.
3. Click **+Add Project**. Then add a Project Name (e.g., Weather API), Subdomain (e.g., weather-api), and Project Logo. Then click **Create**.



- Now check out the API doc configuration section. In the left sidebar, click **Reference Docs**, and then click **API**.

For a demo of the sample weather API (that we've been using in this course) published on Readme.io, see [apitest.readme.io/docs](https://apitest.readme.io/docs).

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:

```

{
 "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
http://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 the platform. Readme.io 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 probably can't just hack it into your code. 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 \(page 249\)](#).

# Which tool to choose for API docs — my recommendations

I described a smattering of tools in [Docs-as-code tool options for developer docs \(page 227\)](#). Which solution should you choose? It's a complicated decision that will invariably involve tradeoffs. As I said earlier, 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? 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 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*.

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 \(page 230\)](#), [Sphinx \(page 232\)](#), or [Jekyll \(page 0\)](#). If you have serious doc needs (or 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 \(page 0\)](#) or [Netlify CMS \(page 244\)](#) (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, a special TOC for series items, 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 \(page 221\)](#). But if engineers insist on Markdown, then maybe Jekyll or Hugo will be a better choice. You can use also Markdown with Sphinx, but when you do, some other Sphinx features become limited.

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.

## 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 \(page 240\)](#). If you've decided on Jekyll, then explore [GitHub Pages \(page 0\)](#), [CloudCannon \(page 240\)](#), [Netlify \(page 241\)](#), or [Aerobatic \(page 242\)](#). If you've decided on Hugo, then explore [Netlify \(page 241\)](#) or [Aerobatic \(page 242\)](#).

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.

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. Even so, after you get used to continuous delivery publishing by simply committing to your repo, it's hard to consider publishing any other way.

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 \(page 282\)](#) for my experience going down this route.

## Parsing the OpenAPI specification

At this point in the course, I haven't yet explored the [OpenAPI specification \(page 298\)](#), but it could be an important factor in your consideration of tools. How will you display all the [endpoint reference documentation \(page 83\)](#)? Rather than [creating your own template \(page 199\)](#) 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 \(page 366\)](#) (yet), so perhaps the best alternative might be to [embed Swagger UI \(page 355\)](#) (assuming more deep integration isn't possible) into your doc platform.

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 even notice 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

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 is probably not a tool you would use for publishing your docs, understanding how to interact with it can be important for understanding [version control \(page 224\)](#).

Learning GitHub will allow you to become familiar with the version control workflows that are common with many docs-as-code tools. 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 a GitHub repository page for `basecamp/bcx-api`. The repository has 253 commits, 5 branches, 0 releases, and 23 contributors. The master branch is selected. A merge pull request #177 from `tjschuck/patch-1` is shown, authored by `georgeclaghorn` on Apr 9. The README.md file contains documentation for the new Basecamp API, mentioning it's REST-style and uses JSON serialization and OAuth 2 authentication. The sidebar shows issues (6), pull requests (0), and a pulse. Clone options are available at the bottom right.

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 \(page 209\)](#), 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 then commit them to your GitHub repository.

GitHub wikis have some limitations:

- **Limited branding.** All GitHub wikis 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 \(page 274\)](#).

## Set up Git and GitHub authentication

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).

1. If you don't already have Git, set it up on your computer.

You can check to see if you have Git already installed by opening a terminal and typing `git --version`.

On Windows, it might be easiest to install Git by [installing GitHub Desktop](#). Installing GitHub Desktop will include all the Git software as well. However, I recommend using the command line rather than the GitHub Desktop GUI tool.

If you're installing the Windows version of GitHub Desktop, after you install GitHub, you'll get a special GitHub Shell shortcut that you can use to work on the command line. Use that special GitHub Shell rather than the usual command line prompt. When you use that GitHub Shell, you can also use more typical Unix commands, such as `pwd` for present working directory instead of `dir` (though both commands will work).

You can also install Git on Windows by following the instructions here: [Installing on Windows](#).

To install Git on a Mac, see [Installing on Mac](#). On a Mac, however, you don't need a special Git Shell. You can open Terminal by doing to **Applications > Utilities > Terminal**. Or install [iTerm](#), or use [PlatformIO IDE Terminal](#) in [Atom](#) (my preferred method).

2. Create a GitHub account by going to [GitHub.com](#).

GitHub and Git are not the same. Git provides [distributed version control \(page 0\)](#). 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.

3. Configure Git with GitHub authorization. This will allow you to push changes without entering your username and password each time. See the following topics to set this up:

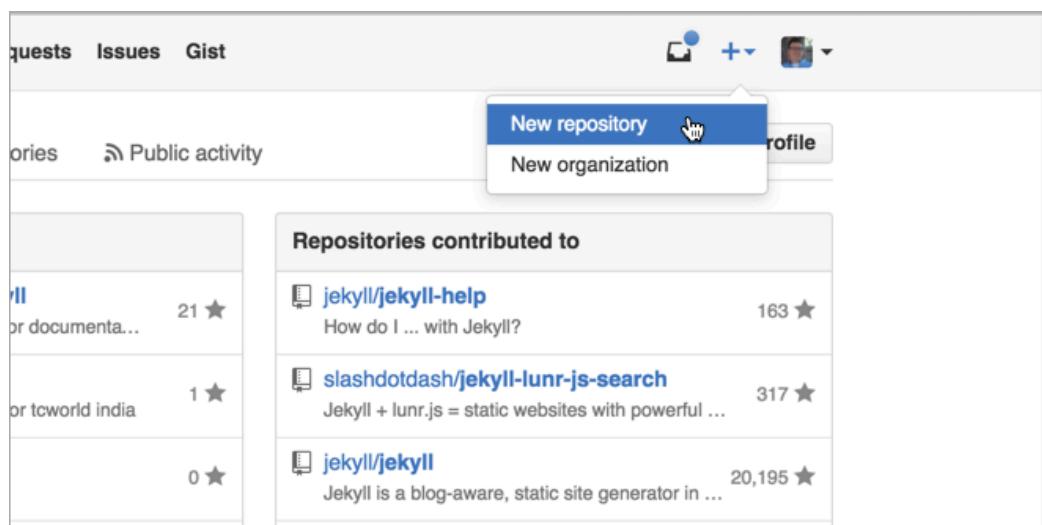
- [Set up Git](#). Note that when you configure your username, use your GitHub username, which will be something like `tomjohht` instead of `Tom Johnson`.
- [Generating a new SSH key and adding it to the ssh-agent](#)
- [Adding a new SSH key to your GitHub account](#)
- [Associating text editors with Git](#)

After you make these configurations, close and re-open your terminal.

## 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. Go to [GitHub](#) and sign in. After you're signed in, click the + button in the upper-right corner and select **New repository**.



2. Give the repository a name, description, select **Public**, select **Initialize the repo with a README**, and then click **Create repository**.
3. Click the **Wiki** link at the top of the repository.
4. Click **Create the first page**.
5. 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.
6. In the **Edit Message** box, type a description of what you updated (your commit message).
7. Click **Save page**.

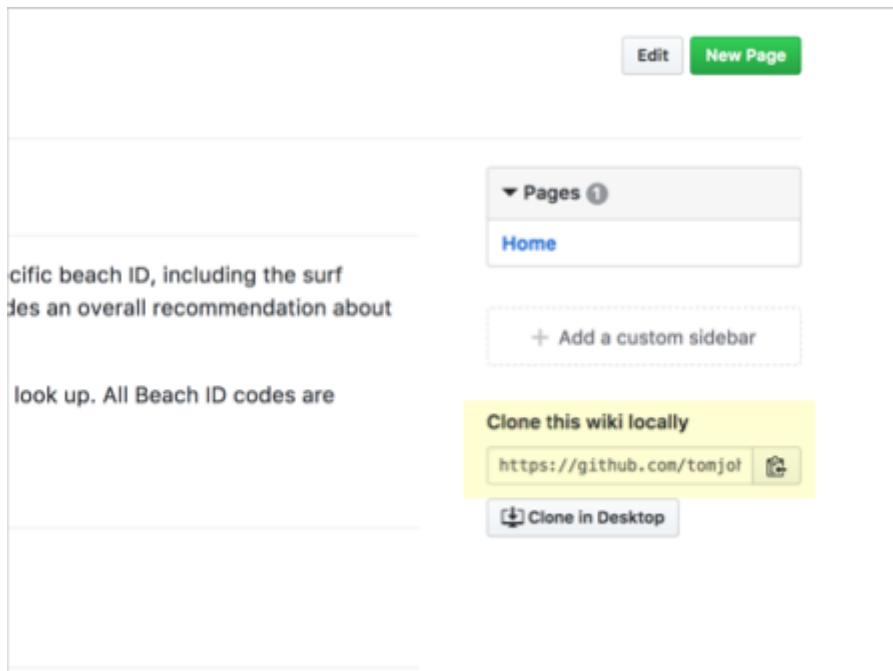
Notice how GitHub automatically converts the Markdown syntax into HTML and styles it in a readable way.

You could use this GitHub wiki in an entirely browser-based way for multiple people to collaborate and edit content. However, you can also take all the content offline and edit locally, and then commit your changes and push them online.

## Save the GitHub repository locally

So far you've been working with GitHub in the browser. Now we'll take the same content and work with it locally.

1. While viewing your GitHub wiki in your browser, look to the right to the section that says **Clone this wiki locally**. Click the clipboard button. (This copies the clone URL to your clipboard.)



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 “Clone this wiki locally,” the “Clone in Desktop” option 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.

2. If you're a Windows user, open the **Git Shell**, which should be a shortcut on your Desktop or should be available in your list of programs. (This shell gets installed when you installed GitHub Desktop — see [Set Up Git and GitHub authentication \(page 254\)](#) above). On a Mac, open a terminal.
3. In your terminal, either use the default directory or browse (`cd`) to a directory where you want to download your repository.
4. 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 like this:

```
git clone https://github.com/tomjoh/weatherapi.wiki.git
```

To paste content into the Git Shell on Windows, right-click and select **Paste**.

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. You will clone this wiki on your local machine; the version in the cloud on GitHub is referred to as “origin.”

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 of the online repository (origin) pulled down to your local copy. You can also `commit` your changes and then `push` your changes back up to the origin repository if you’re entitled as a collaborator for the project.

5. Navigate to the directory (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.

If you can view invisible files on your machine ([Windows](#), [Mac](#)), you will also see a git folder.

## Make a change locally, commit it, and push the commit to the GitHub repository

1. In a text editor, open the Markdown file you downloaded in the GitHub repository.

On a Mac, you can type `open Home.md` in your Terminal, and the file will open in the default text editor.

2. Make a small change 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 the invisible Git folder has been initialized. Git tracks modifications only for the files that have been “added” to Git. By selecting `.` or `--all`, you’re adding all the files in the folder to Git. You could also type a specific file name here instead, such as `git add Home.md`, to just add one file to Git’s tracking.

Use Git only to track text (non-binary) files. Don’t start tracking large binary files, especially audio or video files. Version control systems really can’t handle that kind of format well. 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.

5. See the changes set in your staging area:

```
git status
```

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. You can always back out a change to a file by typing `git checkout Home.md`, where `Home.md` is the file you in which want to undo changes and Git staging.

6. 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.) To use this mode, you have to know a few simple Unix commands:

- **Arrow keys:** You use your arrow keys to move around. You don't use your mouse.
- **Insert mode:** If you start typing, vi enters the Insert mode.
- **Escaping out of Insert Mode:** To escape out of Insert mode, press your `Escape` key.
- **Saving** To save your edits, you need to do a "write quit." Press `Escape` to exit Insert mode. Then Press `Ctrl + :.` Then type `wq` for "write quit." If you made changes but don't want to save them, type `q!` for "quit override."

You can also use [other vim commands](#). Overall, I don't like the Vi[m] editor and prefer using Sublime Text as my default editor associated with Git. See [Associating text editors with Git](#) for details.

7. Push the changes to your repository:

```
git push
```

If you didn't [set up GitHub authentication \(page 254\)](#), you may 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 `git pull origin master`. Some developers prefer to specify the repository and branch to ensure they are interacting with the right repositories and branches.

8. 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](#).

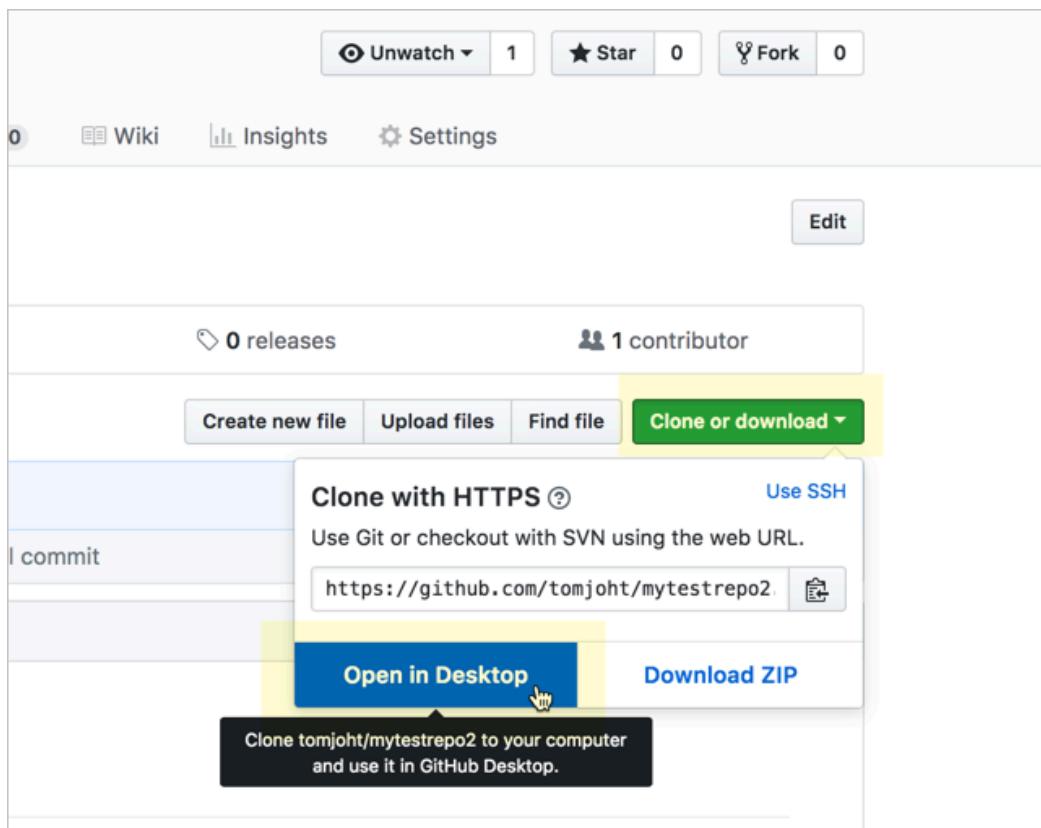
# Activity: Use the GitHub Desktop client

Although most developers use the command line when working with version control systems, there are many GUI clients available that may simplify the whole 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

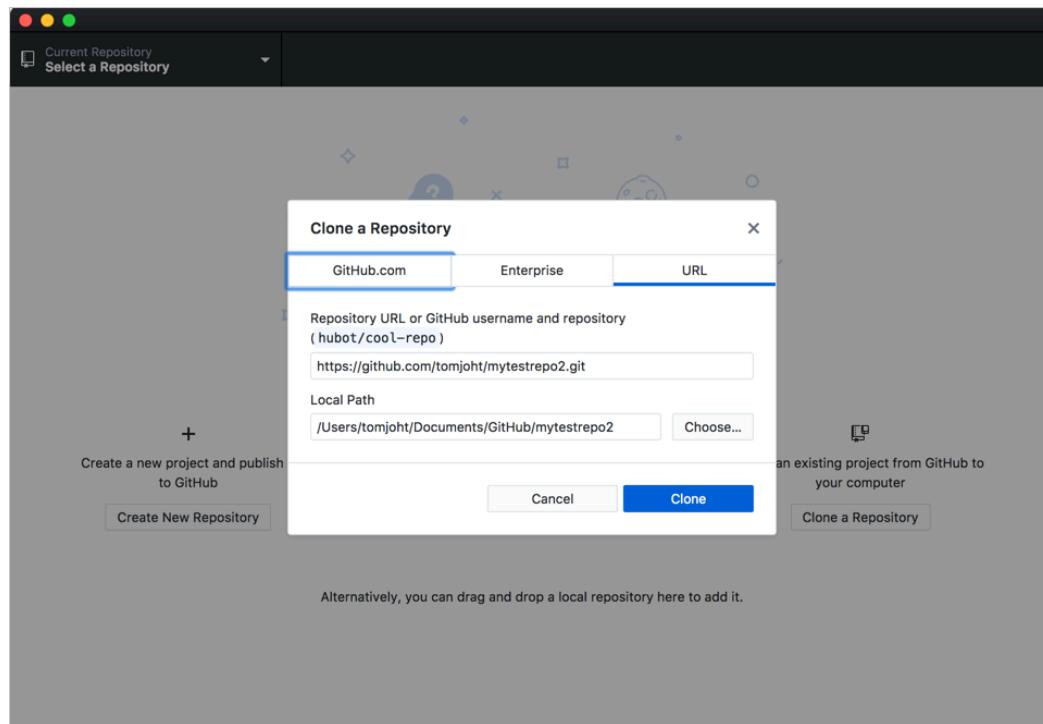
In this tutorial, you'll use GitHub Desktop to manage the Git workflow.

1. First, download and install [GitHub Desktop](#). Launch the app and sign in. (You should already have a GitHub account from [previous tutorials \(page 253\)](#), but if not, create one.)
2. Go to [Github.com](#) and browse to the wiki repository you created in the [GitHub tutorial \(page 253\)](#). (If you didn't do the previous activity, just create a new repository.)
3. Click **Clone** and select **Open in Desktop**. (In the confirmation dialog, select **Open in GitHub Desktop.app**.)

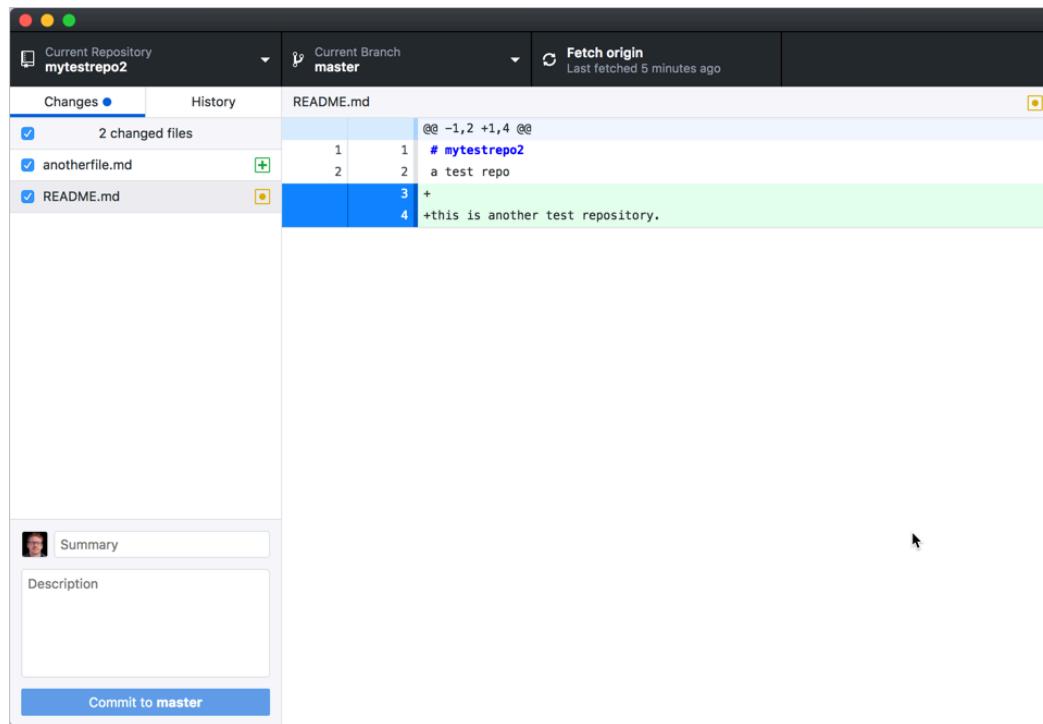


GitHub Desktop should launch with a “Clone a Repository” dialog box about where to clone the repository. If desired, you can change the Local Path.

4. Click **Clone**.



5. 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.
6. Go back to GitHub Desktop. You'll see the new file you added in the list of uncommitted changes on the left.



In the list of changed files, the green + means you've added a new file. The yellow circle means you've modified an existing file.

7. 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.

8. Click **Push origin** at the top.

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.

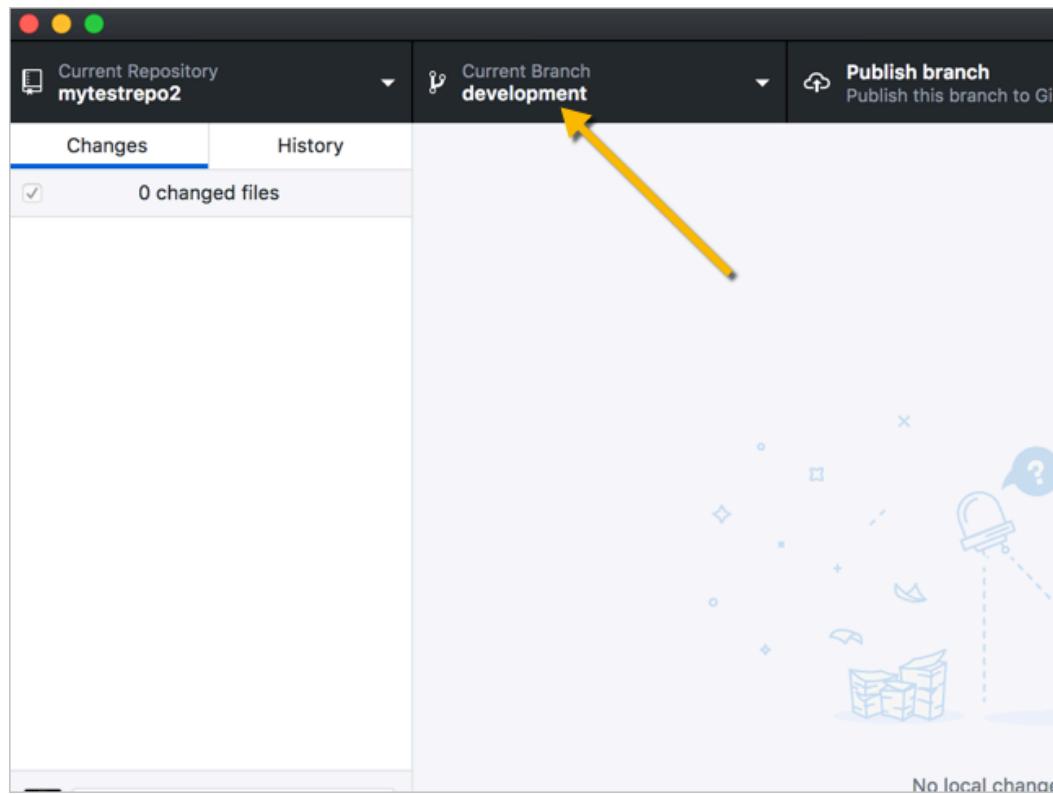
Although I prefer to use the terminal instead of the GitHub Desktop GUI, the GUI gives you an easier visual experience to see the changes made to a repository. You can use both the command line and Desktop client in combination, if you want.

## Create a branch

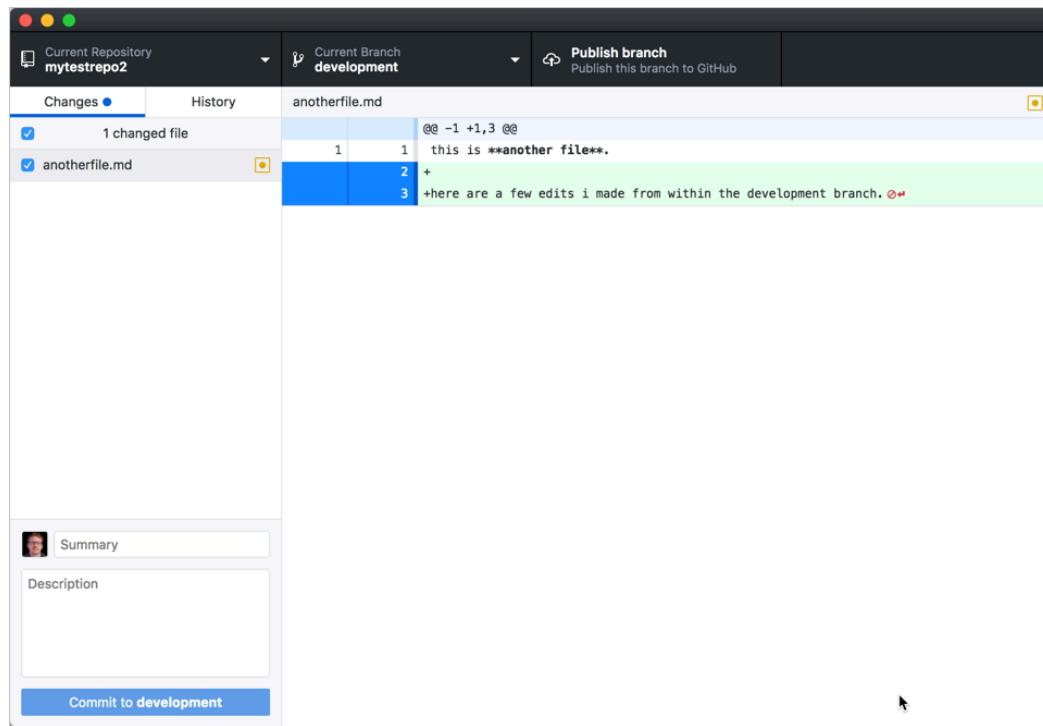
Now let's create a branch, make some changes, and see how the changes are specific to that branch.

1. Go to **Branch > New Branch** and create a new branch. Call it “development” branch.

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).



2. In your new branch, 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.



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** to make the local branch also available on origin (GitHub). (There are usually two versions of a branch — the local version and the remote version.)
6. Switch back to your master branch and note how the changes you made while editing in the development branch don't appear in your master branch.

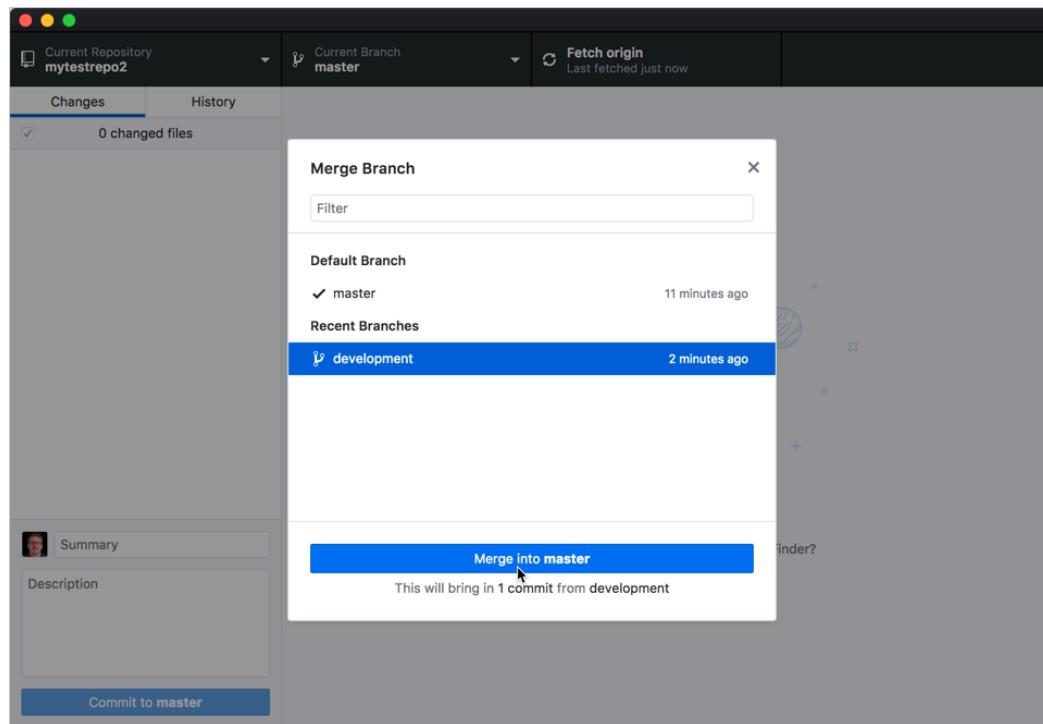
Usually, you create a new branch when you're making extensive changes to your content. For example, suppose you want to revamp a section ("Section X") in your docs. However, you might want to publish other updates before publishing the extensive changes in Section X. If you were working in the same branch, 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, select the development branch. Make a change to a file, and then commit the changes.
2. Switch to the master branch.
3. Go to **Branch > Merge into Current Branch**.
4. In the merge window, select the **development** branch, and then click **Merge into master**.



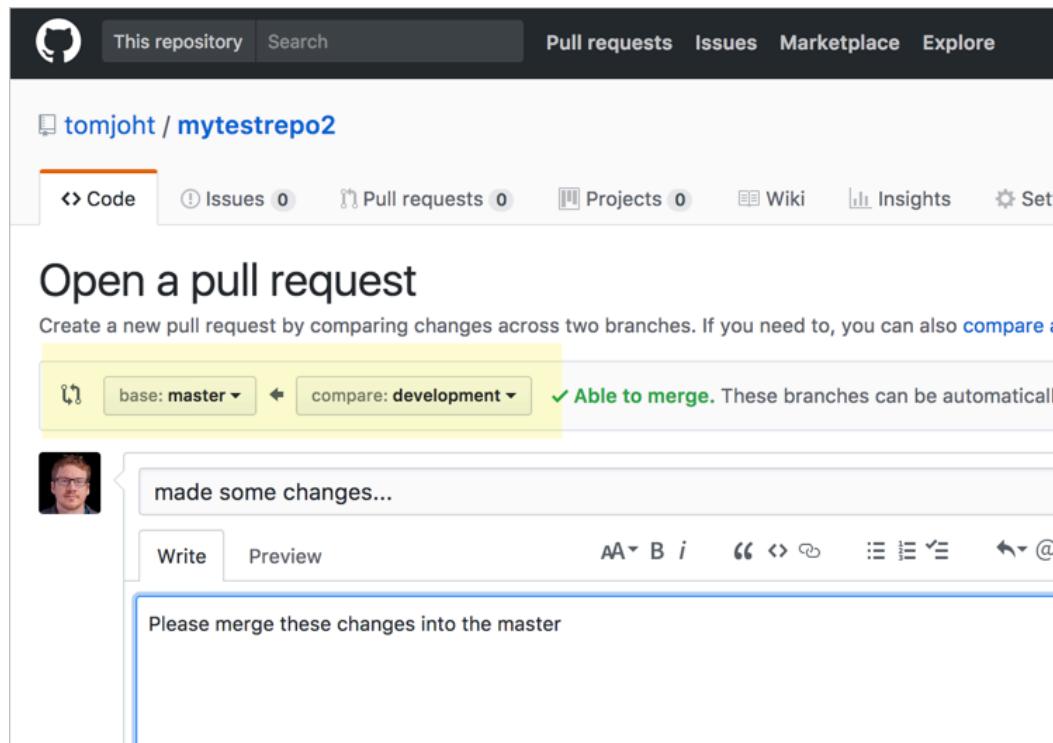
If you look at your changed file, you should see the changes in the master branch.

## 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. As before, 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**.

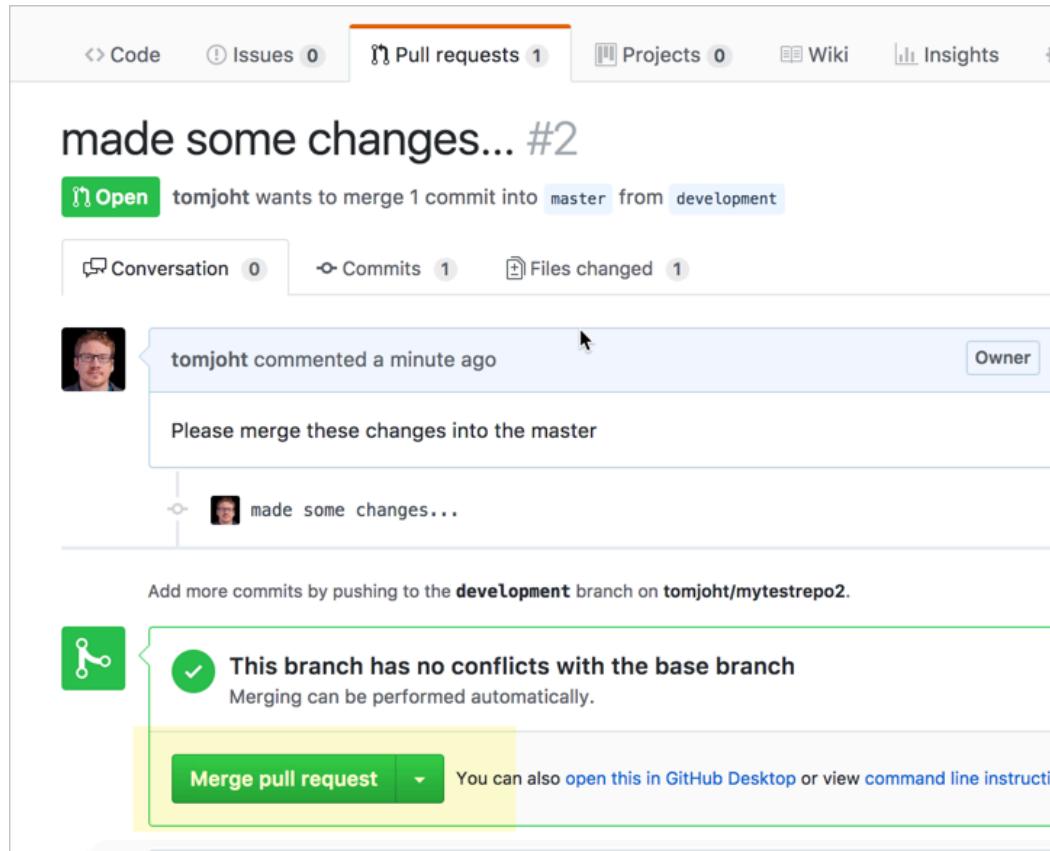
GitHub opens in the browser with the Pull Request form opened.



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 **Send Pull Request**.

At this point, engineers would get an email request asking for them to merge in the edits. Play the part of the engineer by confirming the merge request. As long as the merge request doesn’t pose any conflicts, you’ll see a **Merge Pull Request** button.



To see what changes you're merging into master, you can click the **Files changed** tab. Then click **Merge Pull Request** to merge in the branch, and click **Confirm Merge** to complete the merge.

4. 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**.

5. 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.

# Pull request workflows through GitHub

In the previous step, [Activity: Use the GitHub Desktop Client \(page 259\)](#), 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.

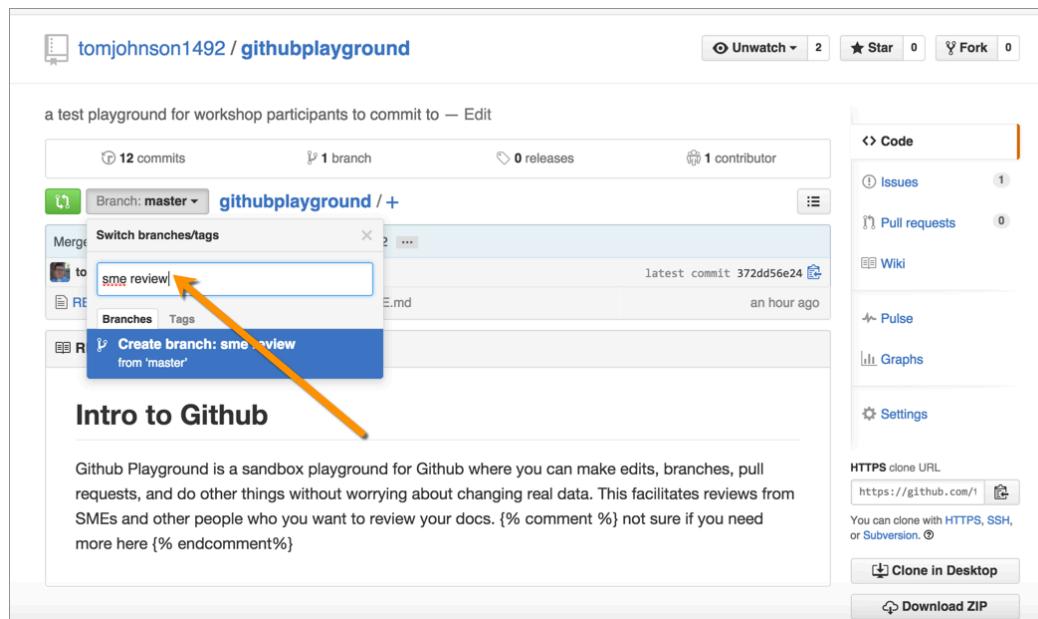
## Make edits in a separate branch

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.

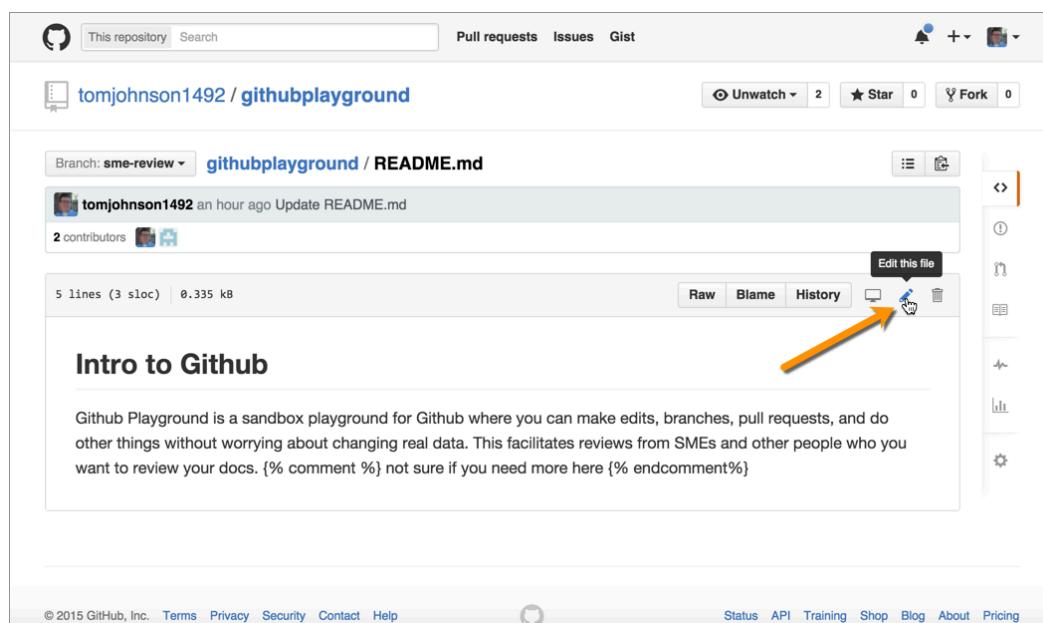
To make edits in a separate branch on GitHub:

1. Pretend you're a SME reviewer. Go to the GitHub repo and 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**.



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.



3. Make some changes to the content, and then scroll down and click **Commit Changes**. Explain the reason for the changes and commit the changes to your sme review branch by clicking **Commit Changes**.

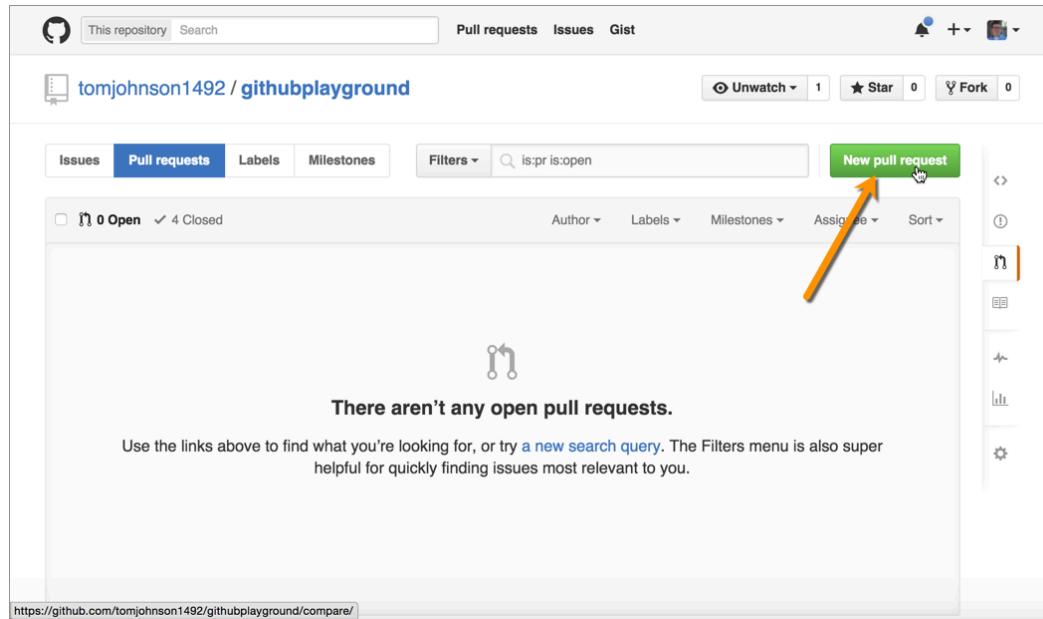
Reviewers could continue making edits this way until they have finished reviewing all of the documentation. All of the changes are made on a branch, not the master.

## Create a pull request

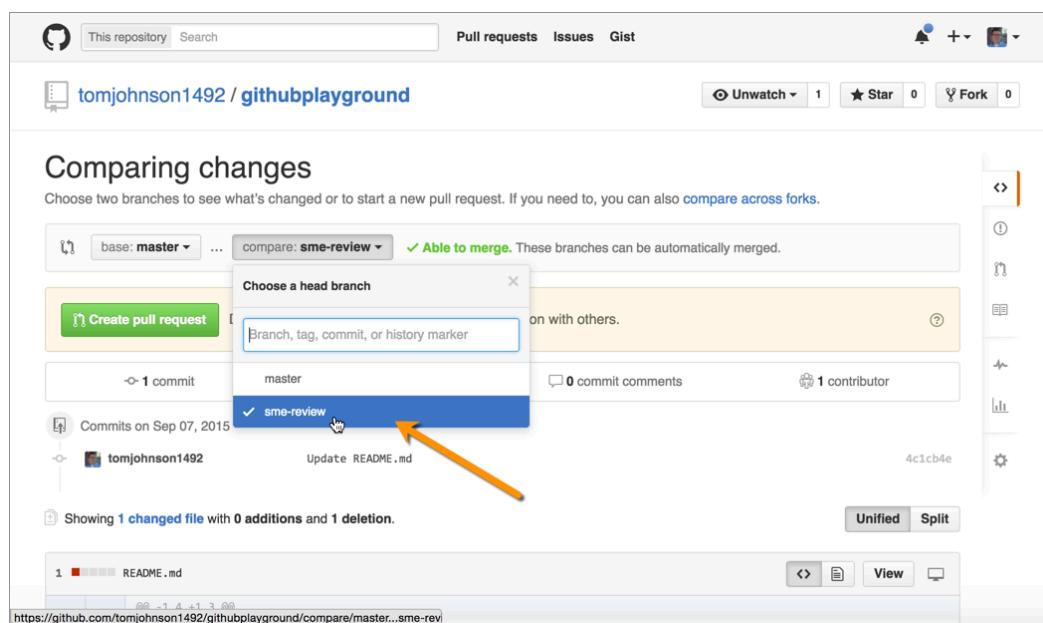
Now that the review process is complete, it's time to merge the branch into the master. You merge the branch into the master through a pull request. Any “collaborator” on the team with write access can initiate and complete the pull request. You can add collaborators through Settings > Collaborators.

To create a pull request:

1. View the repository and click the **Pull requests** button on the right.
2. Click the **New pull request** button.



3. Select the branch ("sme review") that you want to compare against the master.



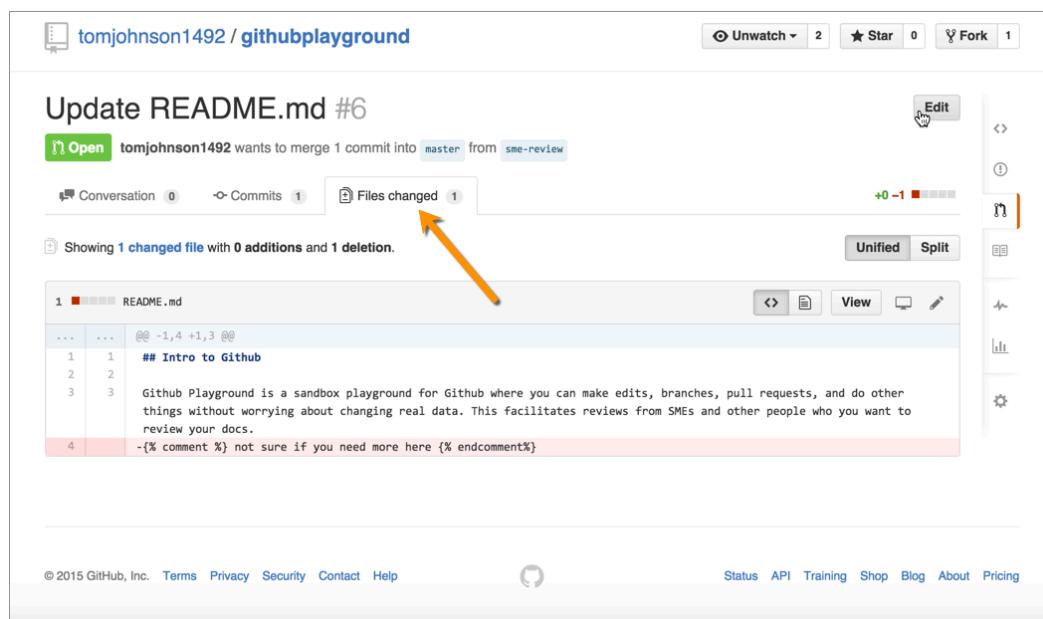
When you compare the branch against the master, you can see a list of all the changes. You can view the changes through two viewing modes: Unified or Split (these are tabs shown on the right of the content). Unified shows the edits together in the same content area, whereas split shows the two files side by side.

4. Click **Create pull request**.
5. Describe the pull request, and then click **Create pull request**.

## 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** button to see the pending pull requests.
2. Click the pull request and view the changes by clicking the **Files changed** tab.



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.

The screenshot shows the GitHub interface for the repository 'tomjohnson1492/githubplayground'. The 'Overview' tab is selected. The 'Default branch' section shows the 'master' branch. The 'Your branches' section lists the 'sme-review' branch, which has a status of 'Merged'. A yellow arrow points to the 'Delete this branch' button next to the merged branch. The 'Active branches' section also lists the 'sme-review' branch with a 'Merged' status. The bottom of the screen shows the GitHub footer with links to Status, API, Training, Shop, Blog, About, and Pricing.

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** button (gear icon).
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**.

This screenshot shows the GitHub repository settings page for the repository `tomjohson1492/githubplayground`. The left sidebar has tabs for Options, Collaborators (which is selected), Branches, Webhooks & services, and Deploy keys. The main area is titled "Collaborators" and includes a note: "This repository doesn't have any collaborators yet. Use the form below to add a collaborator." Below this is a search bar with the placeholder "Search by username, full name or email address" and the input "saphira1410". To the right of the search bar is a blue "Add collaborator" button. An orange arrow points from the bottom-left towards this button. On the far right, there is a vertical sidebar with icons for file operations like copy, paste, and refresh.

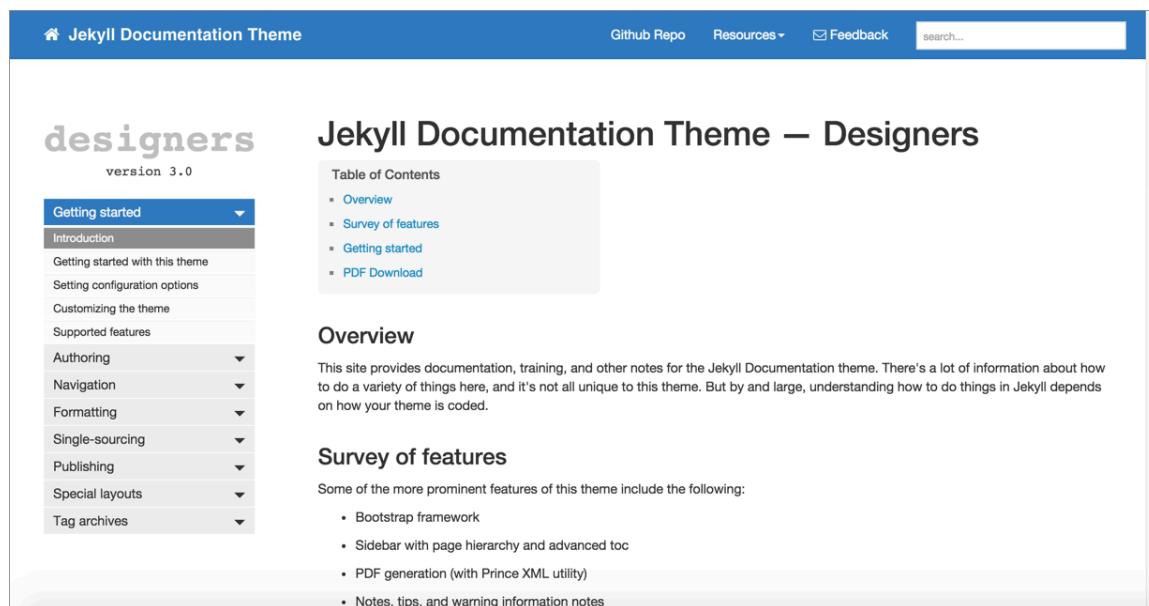
# Jekyll publishing tutorial

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](http://staticgen.com). The most popular static site generator (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.

## Jekyll

Jekyll is a Ruby-based static site generator, meaning it uses Ruby as the underlying programming language to compile the website. This site, my [blog](#), and the documentation for my past two jobs were created using 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 web browser displaying the "Jekyll Documentation Theme — Designers" page. The page has a blue header bar with the title and navigation links. The main content area features a sidebar on the left with a "Table of Contents" section containing links to "Overview", "Survey of features", "Getting started", and "PDF Download". The main content area contains sections for "Overview" and "Survey of features", each with descriptive text and bullet-point lists. The overall design is clean and modern, utilizing a light gray background and white text.

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, translation, PDF generation, and more.

## Static site generators give you a flexible web platform

Static site generators give you a lot of flexibility. They're a good choice if you need a lot of flexibility and control over your site. You're not just plugging into an existing API documentation framework or architecture. You define your own templates and structure things however you want.

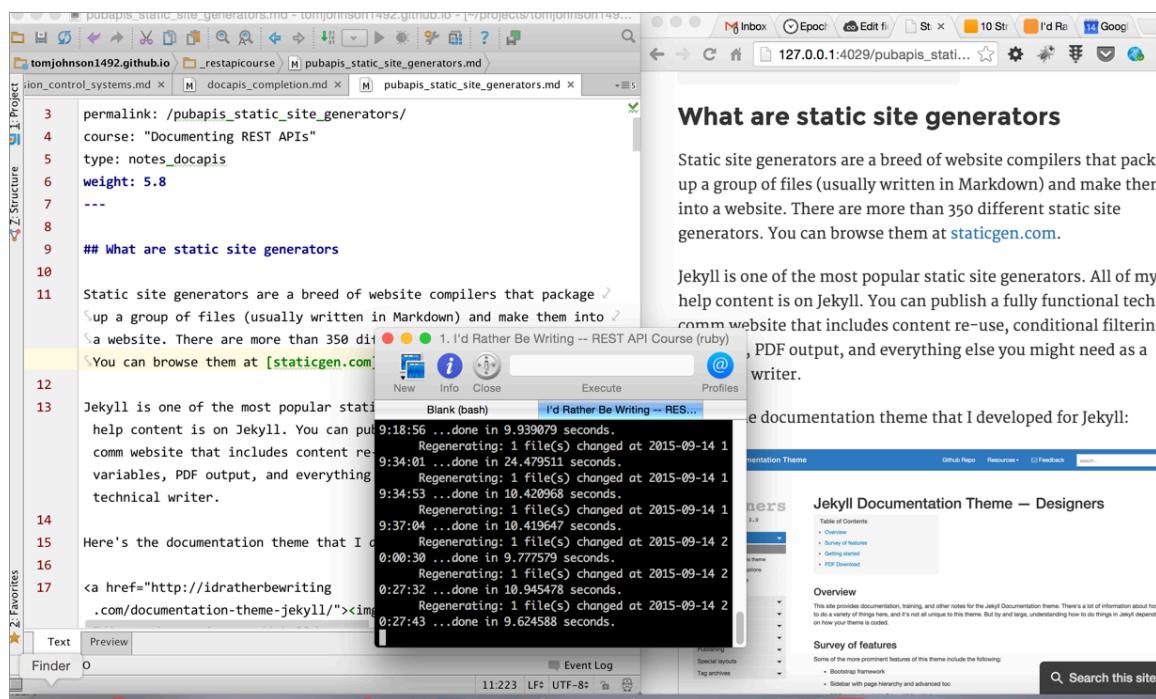
With static site generators, you can do the following:

- Write in a text editor
- 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?

When you're authoring a Jekyll site, you open up a preview server that continuously builds your site with each change you save. I open up my text editor on the left, and the auto-generating site on the right. On a third monitor, I usually put the Terminal window so I can see when a new build is done (it takes about 10 seconds for my doc sites).



This setup works fairly well. Granted, I do have a Mac Thunderbolt 21-inch monitor, so it gives me more real estate. On a small screen, you might have to switch back and forth between screens to see the output.

Admittedly, the Markdown format is easy to use but also susceptible to error, especially if you have complicated list formatting. But for 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

Let's do an example in publishing in [CloudCannon](#), "The Cloud CMS for Jekyll," 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. 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.

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>.

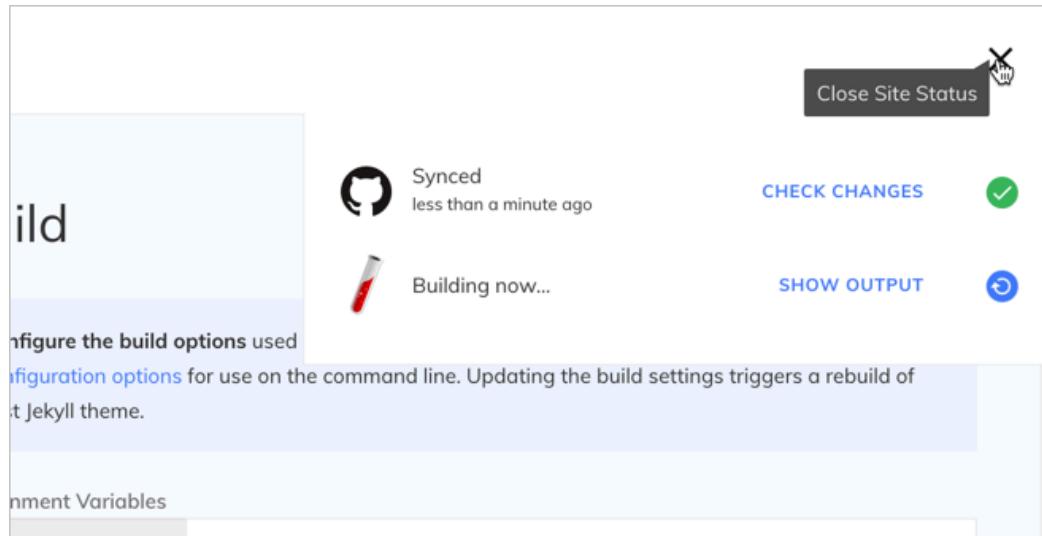
Sometimes people fork repositories to make changes and then propose pull requests of the fork to the original repo. Other times people fork repositories to create a starting point for a splinter project from the original. Github is all about social coding — one person's ending point is another person's 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](#).
3. Once you sign in, click **Create Site** and then give the new site a name.
4. While viewing your site, in the left sidebar, click **Settings**.
5. While still under Settings, click **File Syncing**, and then next to **Github**, click **Connect**. You may be taken to Github to authorize CloudCannon's access to your Github repository.
6. When asked which repository to authorize, select the **Documentation theme for Jekyll** repository (the gh-pages branch), and then click **Connect**.
7. In the left sidebar, click **Files**. You should see the files from the repo you synced from GitHub.
8. Click **Settings**, select **Build**, and do the following:

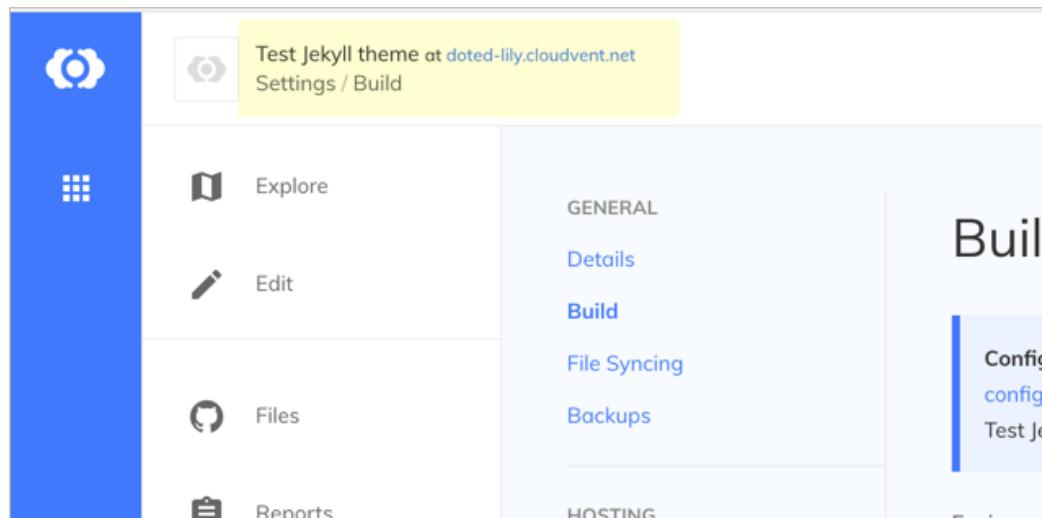
- Clear the **Minify and serve assets from CDN** check box.
- Select **Use Experimental Build Server**.

Then click **Update Build Details** at the bottom.

In the upper-right corner, you'll see a small icon showing the build process. Click the rotating circle to see the build status.



- View your CloudCannon site at the preview URL in the upper-left corner.



It should look just like the [Documentation theme for Jekyll here](#).

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. Then of course 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.

## Make an update to your Github repo

Remember your Github files are syncing from Github to CloudCannon. Let's see that workflow in action.

1. In your browser, go to your Github repository that you forked and make a change.

For example, browse to the index.md file, click the **Edit this file** button (pencil icon), make an update, and then commit the update.

2. Return to CloudCannon and observe that your site automatically starts rebuilding. Wait until the build finishes, 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](#).

## Publish the endpoint in the Jekyll Aviator theme

Let's say you want to use a theme that provides ready-made templates for REST API documentation. Although my Jekyll documentation theme could work, it doesn't have templates coded for API endpoint reference documentation. In this activity, you'll publish the weatherdata endpoints that we worked on earlier; we'll use the [Aviator API documentation theme](#) by CloudCannon, which is designed for REST APIs.

The screenshot shows the Aviator API documentation theme. On the left, there's a sidebar with a green header containing the theme name. Below it are sections for Search, Documentation (with links to Getting Started, Authentication, and Errors), and APIs (listing /books with GET, POST, PUT, and DELETE methods). A note says "Template by CloudCannon". The main content area has a header for "/books GET". It describes the endpoint as listing all books and provides parameters for offset (to offset results) and limit (to limit the number of books returned). It also notes that the call will return a maximum of 100 books. Below this, it says it lists all photos you have access to and provides pagination instructions. At the bottom of this section, there are links for jQuery, Python, Node.js, and Curl, followed by a code snippet for jQuery:

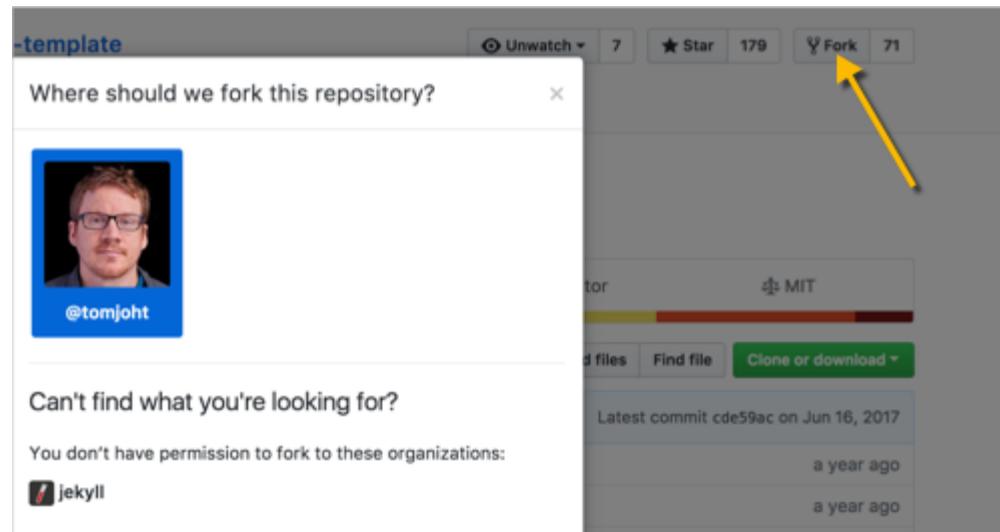
```
$.get("http://api.myapp.com/books/", { "token": "YOUR_APP_KEY"}, function(data) {
 alert(data);
});
```

To the right, there's a "Response" tab showing a JSON array with two book entries. Each entry includes an id, title, score, and dateadded.

```
[
 {
 "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"
 }
]
```

### a. Fork the Jekyll Aviator theme, clone it, and open it in Atom

1. Go to [Aviator API documentation theme](#) and click the **Fork** button in the upper-right. Confirm the GitHub account where you want to fork the project.



Instead of cloning the project, this time we're forking it. You clone a project when you want to regularly pull from and push to the same repo. But we don't own the Aviator repo, and we want to customize it with our own content. You fork a project when you want to make a copy of the project in another GitHub repo.

2. After forking the project, go to the project in your repo and copy the Clone URI. Then clone the repo on your machine. The `git clone` command and URI will look something like this:

```
git clone https://github.com/tomjoht/aviator-jekyll-template.git
```

Only instead of `tomjoht`, you will see your own GitHub username.

3. Download [Atom](#), install it, and open it.
4. Open the **aviator-jekyll-template** folder in Atom so you can see all the files in the sidebar.
5. Go to [CloudCannon](#) and click **Create Site** to create a new site. Give it a name, such as “Aviator demo.”
6. Go to **Settings**, and then click **File Syncing**.
7. Next to **GitHub**, click **Connect**, then next to **aviator-jekyll-template** repo, click **Connect Repo**. The files from the GitHub repo sync over to CloudCannon. In CloudCannon, in the upper-right corner, a green check icon indicates a successful build.
8. Click the preview link in the upper-left and preview the built site.

### b. Add the weatherdata endpoint doc to the theme

1. In Atom, browse to the `_api` folder, duplicate the `books_add.md` file, and name the duplicated file `weatherdata.md`. Then open up `weatherdata.md`.

In every Jekyll page, there's some “frontmatter” at the top. The frontmatter section has three dashes before and after it.

The frontmatter is formatted in a syntax called YML. YML is similar to JSON but uses spaces and hyphens instead of curly braces. This makes it more human readable. (See [More About YAML \(page 384\)](#) for details.)

The key-value pairs in the YAML frontmatter are entirely arbitrary and are designed here to suit the API doc theme the author created. If we were to look at the code in the theme, we'd see Liquid `for` loops that iterate over the YAML values and populate the content into a template. Jekyll will access these values and push this content into the template (which you can see by going to `_layouts/default.md`). The author has separated the content from the format so that we don't have to manually wrap all the values in style tags.

2. Replace the contents of `weatherdata.md` with the following content: [aviator-weatherdata.txt \(page 0\)](#).

Normally, you would build the Jekyll project locally to make sure it looks right. But since that's beyond the scope of this tutorial, we'll just commit it to GitHub and have CloudCannon build it.

3. Then add the files to Git tracking, commit, and push to origin:

```
add all files to Git tracking
git add .

commit the added files to Git's staging
git commit -m "added weatherdata file"

push the files to origin
git push
```

CloudCannon is listening to changes in the repo and will immediately sync and build the project.

4. In CloudCannon, click the preview URL in the upper-left corner and view the built site. The built site should include the `weatherdata` endpoint and look as follows:

The screenshot shows the CloudCannon interface with the Aviator theme. On the left, there's a sidebar with links to Search, Documentation (Getting Started, Authentication, Errors), APIs (books, books POST, books/:id GET, books/:id PUT, books/:id DELETE, weatherdata GET), and a Template by CloudCannon footer. The main content area shows the `/weatherdata` endpoint details. It includes a description: "Gets the full weather data details, including temperature, wind, astronomy, and more.", parameters (`lat` and `lon`), and a note that these values will be added as query parameters. Below this is a jQuery code sample for making a request. To the right, the "Response" tab shows the JSON output for a request to `/weatherdata?lat=37.3708698&lon=-122.037593`. The response object contains a `query` with `count: 1`, `created: "2014-05-03T03:57:53Z"`, `lang: "en-US"`, and a `results` array containing a single channel object with a title ("Yahoo! Weather - Tebrau, MY"), link ([http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau\\_MY/\\*http://weather.yahoo.com/forecast/MYXX004\\_c.html](http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau_MY/*http://weather.yahoo.com/forecast/MYXX004_c.html)), description ("Yahoo! Weather for Tebrau, MY"), language ("en-us"), and lastBuildDate ("Sat, 03 May 2014 11:00 am MTT").

You can view my site here: <http://tropical-jackal.cloudvent.net>.

Each time you commit changes to your repo, CloudCannon rebuilds the Jekyll files into the site that you see.

If you don't want to use CloudCannon to build your site, you can also use [GitHub Pages](#) to achieve a similar result. GitHub Pages is also free.

## Doc Websites Using Jekyll

Here are some websites using Jekyll:

- [Bootstrap](#)
- [Beegit](#)
- [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

For an overview of the docs-as-code approach, see [Docs-as-code tools \(page 209\)](#). 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.

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.

## 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, and the tech writers loved the new system.

## 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 started 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](#). 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.

More specifically, 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, I realized it was the right 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.

## 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 do that too much adjustment 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 liked to try new approaches and techniques.

In the engineering camp, I also took some flak for changing directions too frequently. 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). 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.

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.

## Conclusion

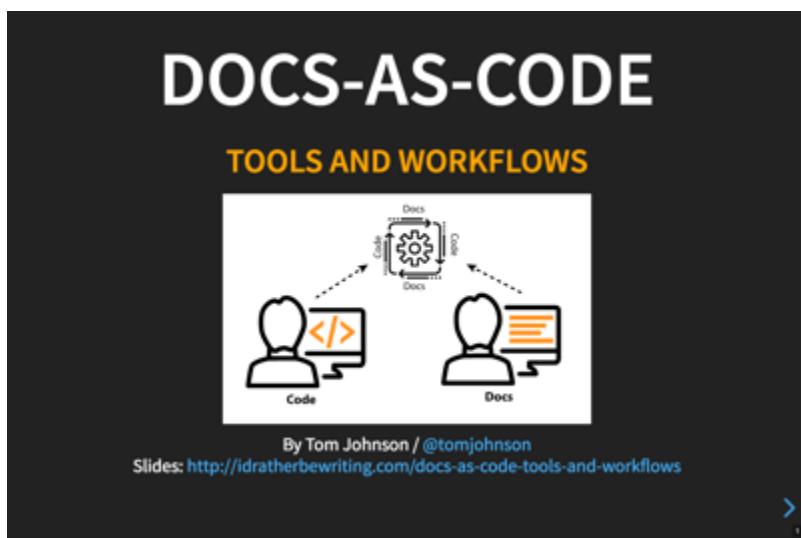
Almost everyone on the team was happy about the way our doc solution turned out. Of course, there are always areas for improvement, but the existing solution was head and shoulders above the previous processes. Perhaps most importantly, Jekyll gave us an incredible degree of flexibility to create and 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 here: [Publishing tool options for developer docs \(page 227\)](#).) 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.

## Slides and links to republished content

I have a slide presentation that covers topics listed in this article:



These slides are for an [upcoming presentation](#) I'm giving on this topic.

Additionally, this content was republished here in the [Developer Portals e-Magazine Winter 2018](#), by Pronovix:



It was also republished in Anne Gentle's [Docs Like Code: Case Studies](#):

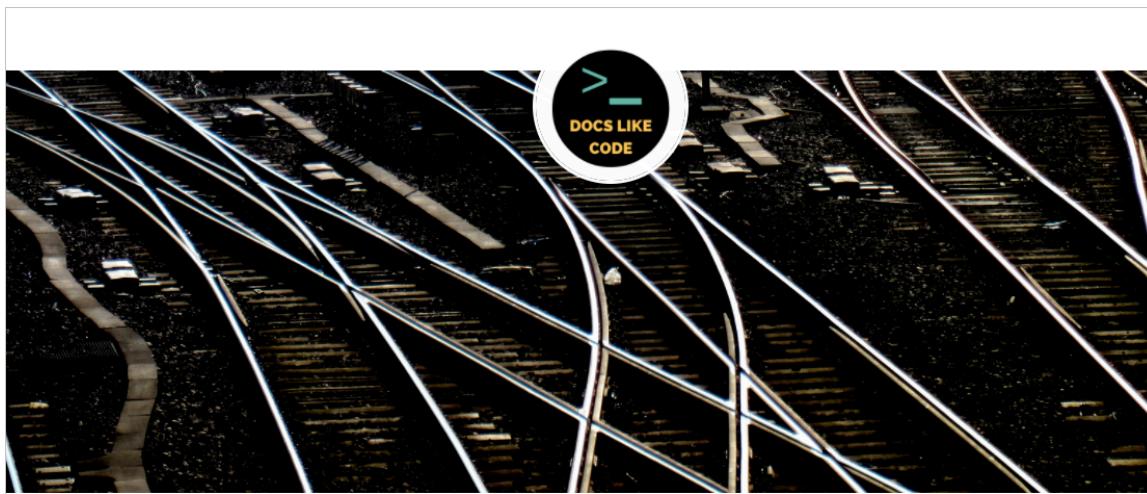


Image credit: Flickr laurencehorton

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 \(page 0\)](#) to the STC Rocky Mountain and WTD Denver group.

## 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](#)

# OpenAPI specification and Swagger

Overview of REST API specification formats.....	296
Introduction to the OpenAPI spec and Swagger.....	298
OpenAPI tutorial overview .....	310
Step 1: openapi object (OpenAPI tutorial) .....	314
Step 2: info object (OpenAPI tutorial) .....	318
Step 3: servers object (OpenAPI tutorial) .....	320
Step 4: paths object (OpenAPI tutorial) .....	322
Step 5: components object (OpenAPI tutorial).....	331
Step 6: security object (OpenAPI tutorial) .....	345
Step 7: tags object (OpenAPI tutorial) .....	349
Step 8: externaldocs object (OpenAPI tutorial) .....	351
Activity: Create an OpenAPI specification document .....	354
Swagger UI tutorial .....	355
Activity: Create your own Swagger UI display .....	363
Swagger UI Demo .....	365
Integrating Swagger UI with the rest of your docs .....	366
SwaggerHub introduction and tutorial .....	372
Other tools to parse and display OpenAPI specs .....	382
More about YAML .....	384
RAML tutorial.....	388
API Blueprint tutorial.....	402

# Overview of REST API specification formats

When I [introduced REST APIs \(page 23\)](#), 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](#).

By far, the OpenAPI specification is the most popular, with the largest community, momentum, and history. Because of this, I spend the most time on OpenAPI here. Overall, these specifications for REST APIs lead to better documentation, tooling, and structure with your API documentation.

“OpenAPI” refers to the specification, while “Swagger” refers to the API tooling that reads and displays the information in the specification.

## Should you use one of these specifications?

In a [survey on API documentation \(page 499\)](#), I asked people if they were automating their REST API documentation through one of these standards. About 30% of the people said yes.

In my opinion, these specifications should certainly be used, as they not only lead to predictable, industry-consistent experiences for users of your APIs, they also force you to standardize on API terminology and give users a way to learn by doing as they try out the endpoints with real parameters and data.

Most of all, the specifications give you a template to fill out with your API. This template makes it clear what information you need, how you organize and structure the information, and other details. This kind of template, standardized and highly valued within the API community, won’t pit you against your engineers as you negotiate which terms to use and what users really need.

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.

Keep in mind that these REST API specifications mostly describe the *reference endpoints* in an API. While the reference topics are important, you will likely have [a lot more documentation to write \(page 159\)](#) in addition to the reference endpoints.

The bulk of documentation often explains how to use the endpoints together to achieve specific goals, how to configure the services that use the endpoints, how to deploy the services, what the various resources and rules are, how to get an API key, throttling limits, and so forth. It’s hard to include all of this information into the specification alone. Nevertheless, the documentation the specification provides often constitutes the core value of your API, since it addresses the endpoints and what they return.

## Separate outputs from other docs

If you choose to automate your documentation using one of these specifications, it likely will be a separate site that showcases your endpoints and provides API interactivity. You’ll still need to write many more pages of documentation about how to actually use your API.

Having multiple documentation outputs (rather than one seamless whole) presents a challenge when creating and publishing API documentation. I explore this challenge in more depth in [Integrating Swagger UI with the rest of your docs \(page 366\)](#).

## Video presentation of this section

For a video presentation of this section of the course, see the following:

This was a presentation I gave to the STC/WTD San Diego chapter on February 18, 2018. (More details are [here](#).)

# Introduction to the OpenAPI specification and Swagger

For step-by-step tutorial on creating an OpenAPI specification document, see the [OpenAPI tutorial here \(page 310\)](#).

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.

## Backstory: experiences that prompted me toward OpenAPI

On one project some years ago, after I created documentation for a new API, the project manager wanted to demo the new functionality to some field engineers.

To prepare for the demo, the project manager summarized, in a PowerPoint presentation, the new endpoints that had been added. The request and responses from each endpoint were included as attractively as possible in a number of PowerPoint slides.

During the demo, the project manager talked through each of the slides, explaining the new endpoints, the parameters the users can configure, and the responses from the server.

How did the field engineers react to the new demo? The field engineers wanted to try out the requests and see the responses for themselves. They wanted to “push the buttons,” so to speak, and see how the API responded.

I’m not sure if they were skeptical of the API’s advertised behavior, or if they had questions the slides failed to answer. But they insisted on making actual calls themselves and seeing the responses, despite what the project manager had noted on each slide.

The field engineers’ insistence on trying out every endpoint made me rethink my API documentation. All the engineers I’ve ever known have had similar inclinations to explore and experiment on their own.

I have a mechanical engineering friend who once nearly entirely dismantled his car’s engine to change a head gasket: he simply loved to take things apart and put them back together. It’s the engineering mind. When you force engineers to passively watch a PowerPoint presentation, they quickly lose interest.

After the meeting, I wanted to make my documentation more interactive, with options for users to try out the calls themselves. I had heard of [Swagger](#). I knew that Swagger / OpenAPI was a way to make my API documentation interactive. Looking at the [Swagger Petstore demo](#), I knew I had to figure it out.

## About OpenAPI

OpenAPI is a specification for describing REST APIs. This means OpenAPI provides a set of objects, with a specific schema about their naming, order, and contents, that you use to describe each part of your API.

You can think of the OpenAPI specification like DITA but for APIs. With DITA, you have a number of elements that you use to describe your help content (for example, `task`, `step`, `cmd`). The elements have a specific order they have to appear in. The `cmd` element must appear inside a `step`, which must appear inside a `task`, and so on. The elements have to be used correctly according to the XML schema in order to be valid.

Many tools can parse valid DITA XML and transform the content into different outputs. The OpenAPI specification works similarly, only the specification is entirely different, since you're describing an API instead of a help topic.

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"
```

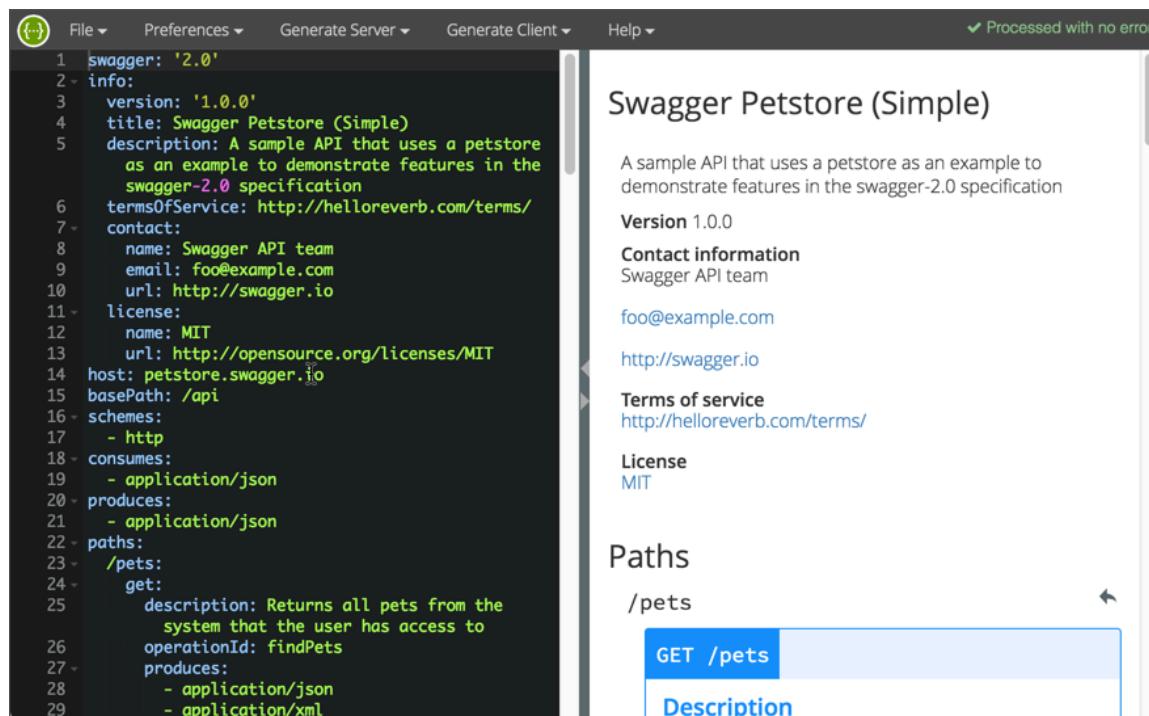
This [YAML code](#) comes from the [Swagger Petstore demo](#).

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.

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, there is a code editor displaying a YAML-based OpenAPI specification for a "Petstore" API. The specification includes details like the swagger version (2.0), info (version 1.0.0, title "Swagger Petstore (Simple)", description about using a petstore as an example, terms of service URL, contact information, license (MIT), host, basePath, schemes (HTTP), consumes (application/json), produces (application/json), and a /pets endpoint with a get operation. The right side of the interface shows the generated API documentation. It features a header for "Swagger Petstore (Simple)" with a note about it being a sample API using a petstore as an example. Below the header are 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). A large section titled "Paths" shows the "/pets" endpoint with a "GET /pets" operation highlighted in blue. A "Description" button is also visible for this operation.

```

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.

## Manual or automated?

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 approached 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 and 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. Other tools such as [REST United](#), [Restlet Studio](#), [APIMATIC](#) can also be used.

These libraries, specific to your programming language, will parse through your code's annotations and generate an OpenAPI specification document. Of course, 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 specification document. 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.

Although you can generate your specification document from code annotations, many say that this 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."

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 embraced with agile. There's nothing worse than releasing a new version of your API that invalidates endpoints or parameters used in previous releases. Documentation also becomes a nightmare.

In my conversations with [Smartbear](#), the company that makes [SwaggerHub \(page 372\)](#) (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 \(page 378\)](#) 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.

With my initial project, our developers weren't that familiar with Swagger or OpenAPI, so I simply created the OpenAPI specification document by hand. Additionally, I didn't have free access to the programming source code, and our developers spoke English as a second or third language only. They weren'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 API template (the approach fits the engineering mindset). As such, you'll probably need to take the lead to guide engineers in the needed information, the 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).

In summary, here are the pros and cons of annotating your programming code to auto-generate the specification document:

#### Pros of code annotations method:

- Reduces potential for documentation drift.
- Consolidates your doc and code in the same location.
- Enables engineers to write documentation using their existing IDE.
- Automatically creates the model definitions for requests and responses.

#### Cons of code annotation method:

- Annotation syntax differs by programming language, with some languages not supported.
- Potentially difficult to gain access to the programming source code (you'd need to integrate into the developer's version control workflow and review tooling).
- Learning curve is greater, since you have to run your app to generate the specification document.
- The specification file can only be generated *after* the API is coded, eliminating the idea of a contract or blueprint.
- None of Swagger Codegen libraries support the latest version of the [OpenAPI 3.0 spec \(page 310\)](#) (as of Nov 2017).
- Annotations clutter up the code with a lot of documentation.
- No ability to use tools like [SwaggerHub \(page 372\)](#) to collaborate (which provides inline commenting and versioning features).

## 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 example](#) I referenced earlier.

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, you basically just separate out the **dist** folder, open up the **index.html** file inside the **dist** folder (which contains the Swagger UI project build) and reference your own OpenAPI specification document in place of the default one.

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

The screenshot shows the Swagger Petstore UI interface. 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 Petstore server and links to '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 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 color-coded (blue for GET, green for POST, blue for GET, red for DELETE) and includes a lock icon indicating security.

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

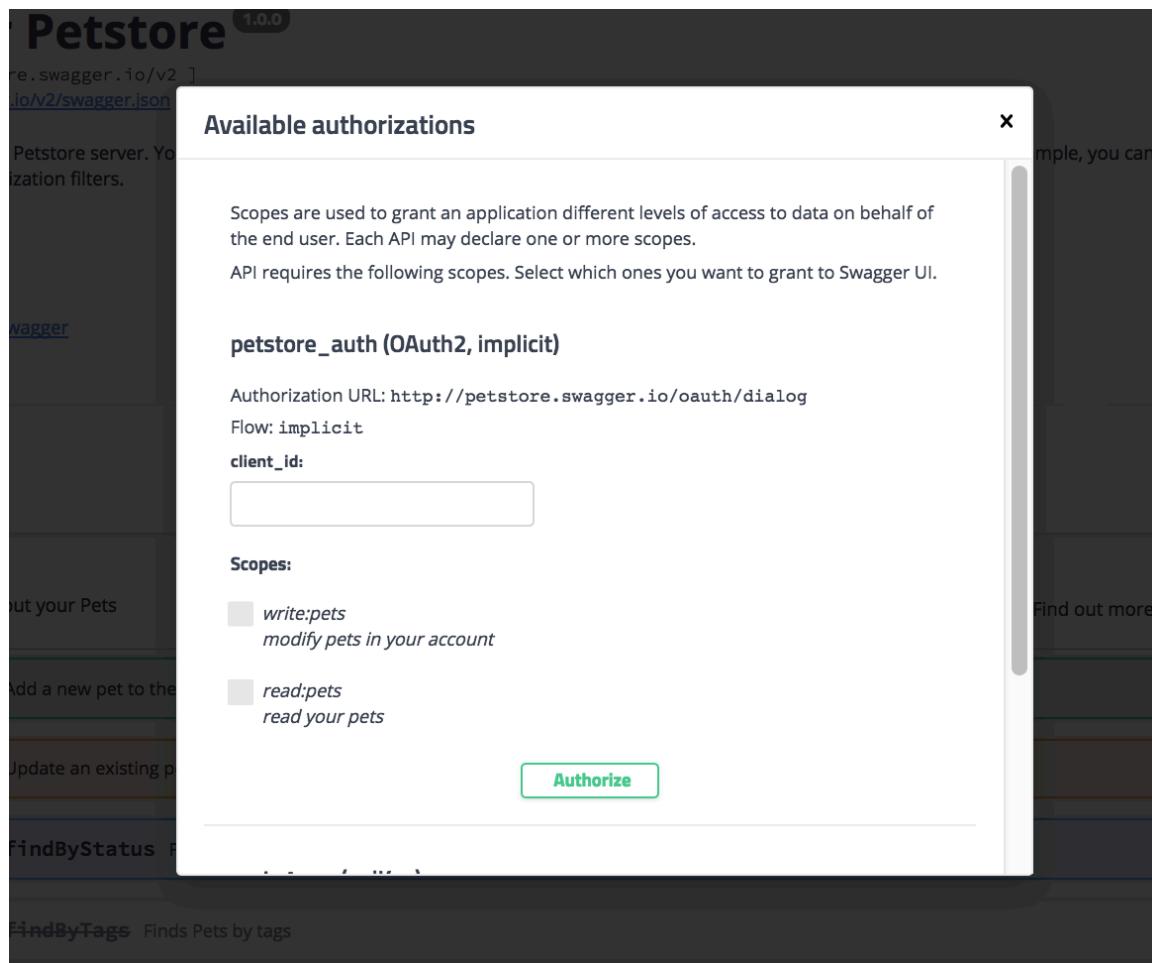
Some designers criticize Swagger UI's expandable/collapsible output as being dated. At the same time, developers find the one-page model attractive and like the ability to zoom out or in for details. By consolidating all endpoints on the same page in one view, users can take in the whole API at a glance.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

## Play with Swagger

### ACTIVITY



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.

Next, expand the **Pet** endpoint. Click **Try it out**.

The screenshot shows the Swagger UI interface for a `POST /pet` endpoint. The endpoint description is "Add a new pet to the store". The `body` parameter is marked as required and is described as a "Pet object that needs to be added to the store". An example value is provided as a JSON object:

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

The "Parameter content type" dropdown is set to "application/json". Below the request section, there are tabs 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. Change the first `id` value to an integer, such as `193844`. Change the second `name` value to something you'd recognize (your pet's name). Then click **Execute**.

**POST** /pet Add a new pet to the store

**Parameters**

Name	Description
<b>body</b> * required	Pet object that needs to be added to the store (body)

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. The Responses section shows the response. (If you select JSON rather than XML in the “Response content type” drop-down box, you can specify that JSON is returned rather than XML.)

**Responses**

Response content type application/json

**Curl**

```
curl -X POST "http://petstore.swagger.io/v2/pet" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"id\": 193844, \"category\": { \"id\": 0, \"name\": \"string\" }, \"name\": \"Bentley\", \"photoUrls\": [\"string\"], \"tags\": [{ \"id\": 0, \"name\": \"string\" }], \"status\": \"available\"}"
```

**Request URL**

http://petstore.swagger.io/v2/pet

**Server response**

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

Response headers

The Petsore 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.)

## Responses to Swagger documentation

With my OpenAPI project, I used the Swagger UI to parse my OpenAPI specification. I customized Swagger UI's colors a bit, added a logo and a few other features. I spliced in a reference to Bootstrap so that I could have pop-up modals where users could generate their authorization codes. I even added some collapse and expand features in the description element to provide necessary information to users about a sample project.

Beyond these simple modifications, however, it takes a bit of web-developer prowess to significantly alter the Swagger UI display.

When I showed the results to the project manager, he loved it. He and others quickly embraced the Swagger output in place of the PowerPoint slides 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 was a huge feather in my cap at the company, and it established an immediate credibility of my technical documentation skills, since no one else in the company knew how to deliver the Swagger output.

## A slight trough of disillusionment

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

The OpenAPI specification and Swagger UI's output are just reference documentation. 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 Hello World tutorial, information about how to get API keys, how to configure any API services, information about rate limits, or the hundred other details that go into a user guide for developers.

Even though you have this cool, interactive tool for users to explore and learn about your API, at the same time you still have to provide a user guide. Similarly, 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, and you have to either keep the two in sync, embed one in the other, or otherwise link between the two.

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. By poring through the specification, 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 \(page 355\)](#)
- [OpenAPI specification tutorial \(page 310\)](#)
- [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>.

# OpenAPI 3.0 tutorial overview

In the [Swagger tutorial \(page 355\)](#), I referenced an [OpenAPI specification](#) document without explaining much about it. You simply plugged the document into a Swagger UI project. In this section, we'll dive more deeply into the OpenAPI specification. Specifically, 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.

## General resources for learning the OpenAPI specification

Learning the [OpenAPI specification](#) will take some time. As an estimate, plan about a week 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, 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.

## 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 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, I think Swagger UI is probably the best tool to use when rendering your specification document. Swagger UI is sponsored by SmartBear, the same company that is heavily invested in the [OpenAPI initiative](#) and which develops [Swaggerhub \(page 372\)](#). Their tooling will almost always be 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

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 spec, but the spec was later changed to [OpenAPI](#) to reinforce the open, non-proprietary nature of the standard. Now Swagger refers to API tooling,

not the spec. People 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 \(page 459\)](#).)

## 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 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.

## Terminology to Describe JSON/YAML

The specification document in my OpenAPI tutorial uses YAML, 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 APIHandyman’s take on [JSON vs YAML](#)), so I’ve used YAML exclusively here. You will see that the spec 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 synonyms. 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.

## 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 \(page 314\)](#)
- [Step 2: info object \(page 318\)](#)
- [Step 3: servers object \(page 320\)](#)

- [Step 4: paths object \(page 322\)](#)
- [Step 5: components object \(page 331\)](#)
- [Step 6: security object \(page 345\)](#)
- [Step 7: tags object \(page 349\)](#)
- [Step 8: externalDocs object \(page 351\)](#)

Note that the spec 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.

You can see OpenAPI spec rendered with Swagger UI in the following links:

- [Swagger UI with OpenWeatherMap API](#)
- [Embedded Swagger with OpenWeatherMap API \(page 365\)](#)

## Migrating from OpenAPI 2.0 to 3.0

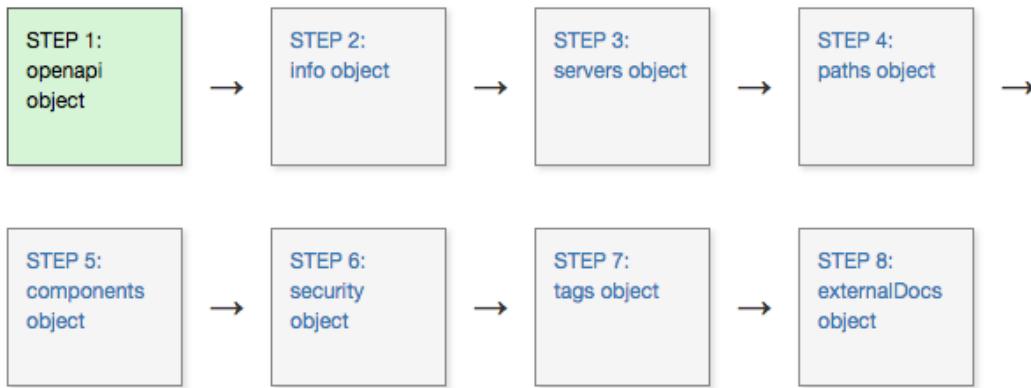
If you have an existing specification document that validates against version OpenAPI 2.0 and you want to convert it to OpenAPI 3.0, you can use [APIMATIC](#) to convert it automatically. You can also use APIMATIC to transform your specification document into a number of other outputs, such as RAML, 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](#).

# 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 \(page 310\)](#) 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 \(page 40\)](#) 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.

Note that the Swagger Editor will validate your content in real-time, and you will probably see validation errors until you finish coding the specification document.

I usually keep a local text file (using [Atom editor](#)) where I keep the specification document, but I work with the document’s content in the online [Swagger Editor](#). When I’m done, I copy and save the content back to my local file. The Swagger Editor caches the content quite well (just don’t clear your browser’s cache).

## Step 1: Add root-level objects

Start your `openapi.yml` file by adding each of these root level objects:

```
openapi:
 info:
 servers:
 paths:
 components:
 security:
 tags:
 externalDocs:
```

In the following sections, we’ll proceed through each of these objects and document the [OpenWeatherMap current API](#). Tackling each root-level object individually 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 \(page 331\)](#) to fully explain how to reference a schema in one object and add a reference pointer to the full definition in `components`.

## The `openapi` object

In the `openapi`, indicate the version of the OpenAPI spec to validate against. The latest version is [3.0.1](#).

```
openapi: "3.0.1"
```

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.

In the Swagger UI display, an "OAS3" tag appears to the right of the API name.

The screenshot shows the Swagger UI interface for the OpenWeatherMap API. The main title is "OpenWeatherMap API". To the right of the title, there are two green buttons: one labeled "2.5" and another labeled "OAS3". An arrow points from the explanatory text above to the "OAS3" button. Below the title, the URL "/learnapidoc/docs/rest\_api\_s" is visible in the address bar. The page content describes the API's capabilities and includes a note about the sample Swagger file covering the "current" endpoint. At the bottom, there are links to various resources: "Terms of service", "OpenWeatherMap API - Website", "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)", and "API Documentation".

## Validator errors

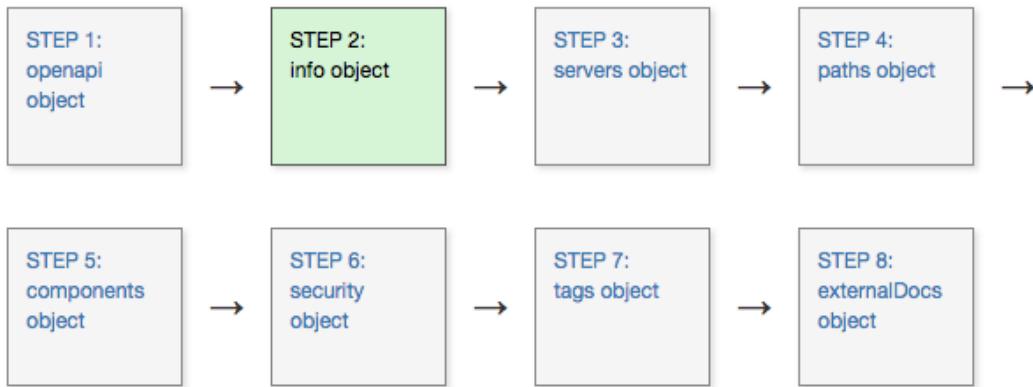
If your spec doesn't validate, the Swagger UI display often won't load the content or will show an error. For example, if you have an incorrect indentation in your YAML syntax, an error message might appear that indicates a `bad indentation of a mapping entry`. You can click the **Error** button in the lower right to see more information.

The screenshot shows the Swagger UI interface. At the top, there's a green header bar with the Swagger logo and navigation links for 'Explore' and the current URL '/learnapidoc/docs/rest\_api\_specifications/openapi\_weather.yml'. Below the header, a red-bordered box contains the word 'Errors' and a detailed parser error message: 'Parser error on line 7: bad indentation of a mapping entry'. A 'Hide' button is located in the top right corner of this error box. Below the error box, the main content area displays the message 'No operations defined in spec!'. In the bottom right corner of this area, there is a red button labeled 'ERROR' with a three-dot icon.

Clicking this error button takes you to [https://online.swagger.io/validator/debug?url=/learnapidoc/docs/rest\\_api\\_specifications/openapi\\_weather.yml](https://online.swagger.io/validator/debug?url=/learnapidoc/docs/rest_api_specifications/openapi_weather.yml), showing you which document the online Swagger validator is attempting to validate and the error. You can also open up the JS console to get a little more debugging information (such as the column where the error occurs).

The online Swagger Editor provides these messages in the UI, so you probably won't need to use Swagger UI's error validation messaging to troubleshoot errors.

# 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:

```
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.

 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.

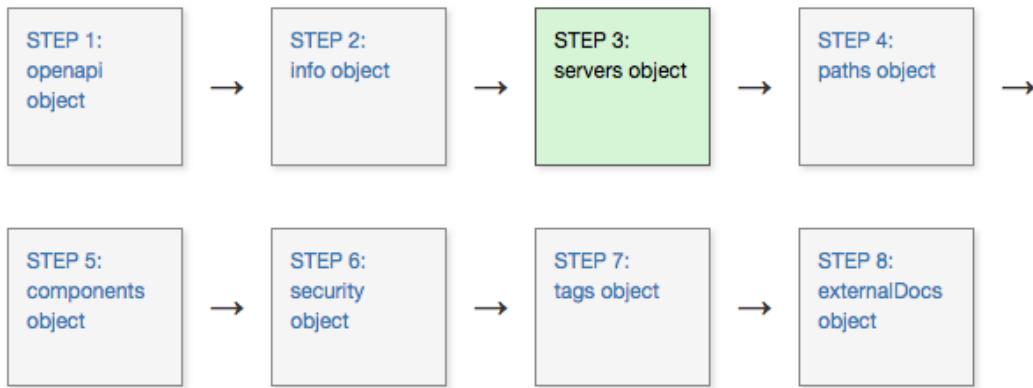
## Appearance in Swagger UI

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

The screenshot shows the Swagger UI interface for the OpenWeatherMap API. At the top, there is a green header bar with the "swagger" logo, a URL input field containing "/learnapidoc/docs/rest\_api\_s|", and a "Explore" button. Below the header, the title "OpenWeatherMap API" is displayed, along with a "2.5 OAS3" badge. A link to the API specification file, "/learnapidoc/docs/rest\_api\_specifications/openapi\_openweathermap.yml", is shown below the title. The main content area contains a brief description of the API: "Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. Note: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API." A 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." At the bottom of the content area, there are links to "Terms of service", "OpenWeatherMap API - Website", "CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)", and "API Documentation".

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: http://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 it to `{server URL}{path}` or `http://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: http://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:

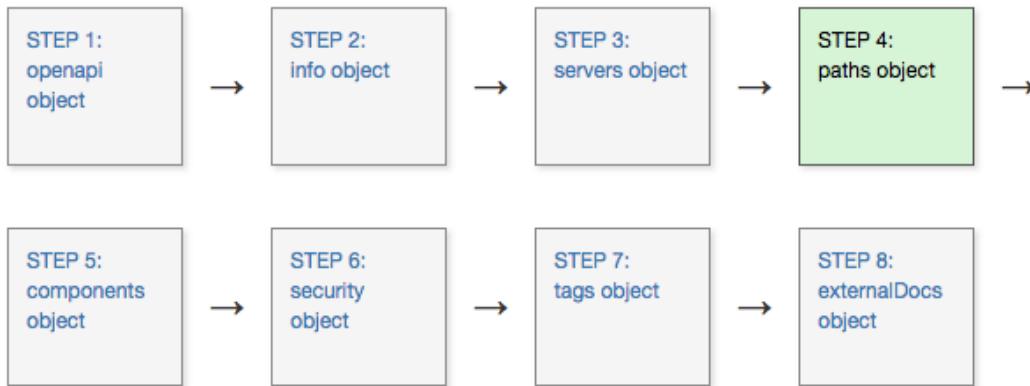
The screenshot shows the Swagger UI interface. At the top right, there is a green button labeled "Authorize" with a lock icon. Below the header, there is a section titled "Server" which contains a dropdown menu with the URL "http://api.openweathermap.org/data/2.5/". A black arrow points from the text "If you have just one URL, you still see a drop-down box but with just one option." to this dropdown menu. Below the server section, there is a detailed description of the "Current Weather Data" endpoint. It includes a "GET" method, a path "/weather", a description "Call current weather data for one location", and a lock icon. The description text 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."

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 322) 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.

# 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.

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.

## Start by listing the paths

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

```
paths:
/weather:
 get:
```

## Operation Objects

Each path item object contains an `operation object`. Operations are the GET, POST, PUT, and DELETE methods we explored in the [Endpoints section \(page 95\)](#) of the API reference tutorial methods. The operation object 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 \(page 331\)](#)).
- `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 \(page 331\)](#)).
- `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. Only include this object at the path level 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 345)`.
- `servers` (object): A servers object that might differ for this path than the global `servers` object ([page 320](#)).

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.

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:

- 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 \(page 320\)](#) at the root level.
- There’s no need to include `security` because all the paths use the same `security` object, which we defined globally at the root (see [step 6 \(page 345\)](#)).
- 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 fields to concern ourselves with:

```
paths:
 /weather:
 get:
 tags:
 summary:
 description:
 operationId:
 externalDocs:
 parameters:
 responses:
```

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.

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.

## Parameters object

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

`parameters`:

- `name` : Parameter name.
- `in` : Where the parameter appears. Possible values: `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.

Here's the `operation` object defined for the OpenWeatherMap API:

```
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 — 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,us*. Please note if country is not specified then the search works for USA as a default."
 schema:
 type: string
```

```
schema:
 type: string
 default: "94040,us"
example:
 "94040,us"

- name: units
 in: query
 description: '**Units**. *Example: imperial*. Possible values: `metric`, `imperial`. When you do not use units parameter, format is `standard` by default.'
 schema:
 type: string
 enum: [standard, metric, imperial]
 default: "standard"
 example: "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"
 example: "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"
 example: "json"
```

## 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.

## Single-sourcing definitions across objects

Instead of defining all your parameters and schemas in the `paths` object, if you re-use the same parameters or schemas across multiple endpoints, you can store these definitions in `components` and then reference them using `$ref` pointers (`$ref` stands for `reference object`). I explain more about using `$ref` in [step 5 \(page 331\)](#).

In this example, I'm just documenting one endpoint in the OpenWeatherMap API. But suppose some parameters such as `lat` and `lon` are re-used across many endpoints. In that case, you would want to store each of these parameters in `components` so that you can re-use them. Copying and pasting information multiple times is inefficient and can lead to inconsistency. The OpenAPI spec allows you to single source the parameter information from a common definition.

The code below shows how you reference a common definition stored in `components`. See how `parameters` simply contains a `$ref: '#/components/parameters/q'`, which is defined in `components`. For brevity, I included just 2 parameters in this example.

```

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'
 ...
 ...

components:
 parameters:
 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:
 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 — 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.*"
 schema:
 type: string
 ...

```

See [Storing re-used parameters in components \(page 0\)](#) for more details. Also see [Describing Parameters](#) in Swagger's OpenAPI documentation.

## Responses

One property of the operation object that we haven't yet defined is the `responses` object. `responses` is at the same level as `parameters`. 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 \(page 331\)](#).

For now, just add a `$ref` pointer to it:

```
paths:
 /weather:
 get:
 tags:
 summary: Get current weather data
 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: WeatherGet
 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.
 schema:
 type: string

 - 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 -- just separate them by commas. The limit of locations is 20. NOTE: A single ID counts as one API call. So, if you have city IDs. it's treated as 3 API calls."
 schema:
 type: string

 ...
 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
```

We'll define the details for `$ref: '#/components/schemas/WeatherGetResponse'` in the next topic ([page 0](#)).

## Appearance of paths in Swagger UI

Swagger UI displays the `paths` object like this:

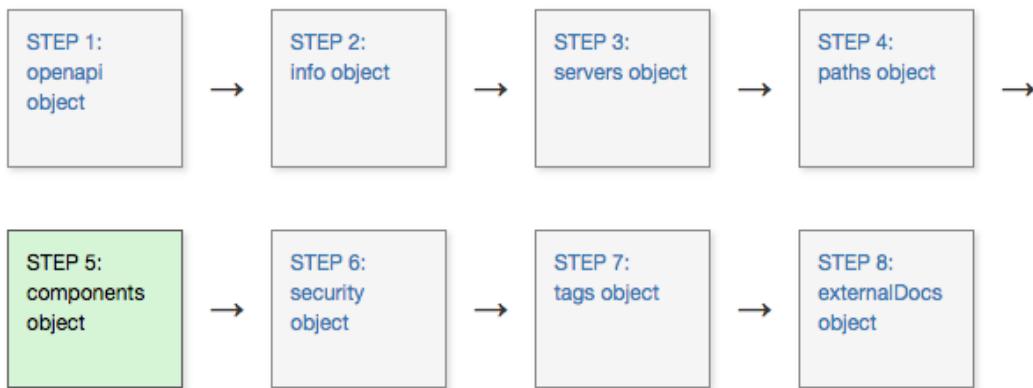
Name	Description
q string (query)	<b>City name.</b> Example: London. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by comma; use ISO 3166 country codes.
id string (query)	<b>City ID.</b> Example: 2172797. You can call by city ID. API responds with exact result. The List of city IDs can be downloaded <a href="#">here</a> . You can include multiple cities in parameter — just separate them by commas. The limit of locations is 20. Note: A single ID counts as a one API call. So, if you have city IDs, it's treated as 3 API calls.
lat	<b>Latitude.</b> Example: 35. The latitude coordinate of the location of your

When you click **Try it out**, the `example` value populates the parameters field.

Each path is collapsed by default, but 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](#) that control its display. If you don't want the `Models` section to appear, add the parameter `defaultModelsExpandDepth: -1,` in your Swagger UI file.

# Step 5: The components object (OpenAPI tutorial)



In the [step 4 \(page 322\)](#), when we described the `responses` object object, 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?](#) for more details). It's common to use a reference pointer (`$ref`) for the `schema` object that points to more details in the `components` object.

## Reasons to use the components object

Describing the schema of complex responses can be one of the more challenging aspects of the OpenAPI spec. Although you can define the schema directly in the `requestBody` or `responses` object, you typically don't list it there for two reasons:

- You might want to re-use parts of the schema in other requests or responses. It's common to have the same object, such as `units` or `days`, appear 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 request 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 an 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.

## Re-using response objects

In the previous topic, I explained how to [re-use parameter definitions in components \(page 327\)](#), so I won't re-explain the approach. Here we'll cover how to re-use the schema definitions in the `responses` objects.

Although we don't need to reuse the `weather` response in this exercise (because we're only documenting one endpoint), it'll be better to organize this schema definition under `components` anyway because it reduces the detail under the `paths` object. If you recall in the previous step ([OpenAPI tutorial step 4: The paths object \(page 322\)](#)), the `responses` object for the `weather` endpoint looked like this:

```
paths:
 /current:
 get:
 parameters:

 ...

 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 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:

```
components:
 schemas:
 200:
 title: Successful response
 type: object
 properties:
 coord:
 $ref: '#/components/schemas/Coord'
 weather:
 type: array
 items:
 $ref: '#/components/schemas/Weather'
 description: (more info Weather condition codes)
 base:
 type: string
 description: Internal parameter
 example: cmc stations
 main:
 $ref: '#/components/schemas/Main'
 visibility:
 type: integer
 description: Visibility, meter
 example: 16093
 wind:
 $ref: '#/components/schemas/Wind'
 clouds:
 $ref: '#/components/schemas/Clouds'
 rain:
 $ref: '#/components/schemas/Rain'
 snow:
 $ref: '#/components/schemas/Snow'
 dt:
 type: integer
 description: Time of data calculation, unix, UTC
 format: int32
 example: 1435658272
 sys:
 $ref: '#/components/schemas/Sys'
 id:
 type: integer
 description: City ID
 format: int32
 example: 2172797
 name:
 type: string
 example: Cairns
 cod:
 type: integer
 description: Internal parameter
 format: int32
 example: 200
```

```
Coord:
 title: Coord
 type: object
 properties:
 lon:
 type: number
 description: City geo location, longitude
 example: 145.77000000000001
 lat:
 type: number
 description: City geo location, latitude
 example: -16.92000000000002
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 from current temp that is possible for large cities and megalopolises geographically expanded (use these parameter optionally). Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.'
 example: 289.8199999999999
 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: 5.099999999999996
 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
```

As you can see, not only can you use `$ref` properties in other parts of your spec, you can use it within `components` too.

## Automatically generate the schema from JSON using Stoplight

Describing a JSON response can be complicated and confusing. Fortunately, there's a somewhat easy workaround. Download [Stoplight](#). 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:

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.

## Automating the examples

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. It's one of the neat things about Swagger UI. This way, your schema documentation and example remain consistent.

## Appearance of components in Swagger UI

By default, Swagger UI displays each object in `components` in a section called `Models` at the end of your Swagger UI display. If you consolidate all schemas into a single object, without using the `$ref` property to point to new objects, you will see just one object in Models. If you split out the objects, then you see each object listed separately, including the object that contains all the references.

Because I want to re-use objects, I'm going to define each object in `components` separately. As a result, the Models section looks like this:

The screenshot shows the 'Models' section of the Swagger UI. It contains five entries, each represented by a light gray rectangular button with a dark gray border and a small downward arrow in the top right corner:

- 200 >**
- Coord >**
- Weather >**
- Main >**
- Wind >**

A vertical scroll bar is visible on the right side of the section.

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

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:

Code	Description	Links
200	<p><b>Successful response</b></p> <p><b>application/json</b></p> <p>Controls <code>Accept</code> header.</p> <p>Example Value   <b>Model</b> ←</p> <pre> <b>Successful response</b> ▾ {     coord          Coord &gt; {...}     weather        &gt; [...]     base           string                   example: cmc stations                   Internal parameter      main           Main &gt; {...}     visibility     integer                   example: 16093                   Visibility, meter      wind           Wind &gt; {...}     clouds         Clouds &gt; {...}     rain           Rain &gt; {...}     snow           Snow &gt; {...}   </pre>	N

You might confuse users by including the Models section. To hide Models, simply add the `defaultModelsExpandDepth: -1` parameter in your Swagger UI project.

## 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 keywords:

- `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`

I suggest you start by looking in the OpenAPI's [schema object](#), and then consult the [JSON Schema](#) if something isn't covered.

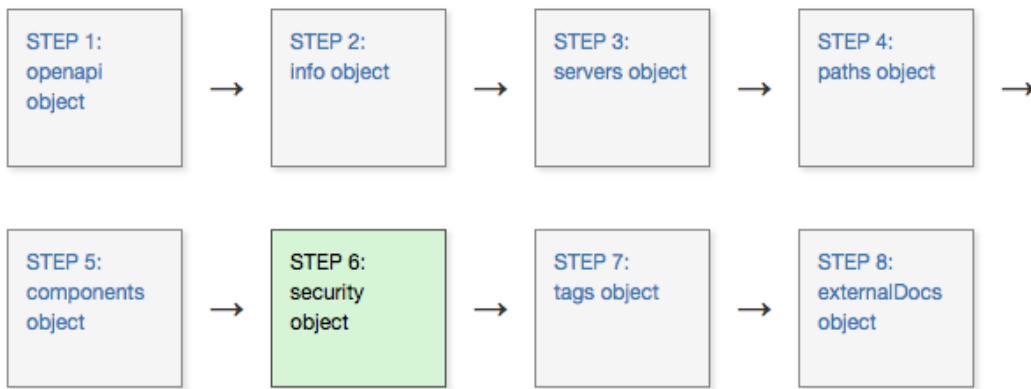
Additionally, look at some example schemas. You can view [3.0 examples here](#). I usually find a spec that resembles what I'm trying to represent and mimic the same properties and structure.

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 lot has changed from 2.0 to 3.0 in the spec.)

## 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 \(page 345\)](#).

# 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 \(page 170\)](#). Swagger UI supports four authorization schemes:

- API key
- HTTP
- OAuth 2.0
- Open ID Connect

In this tutorial, I'll explain the API key method, as it's the most common and it's what I'm most familiar with. If your API uses [OAuth 2.0 \(page 172\)](#) 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.

## Security object

At the root level of your OpenAPI document, add a `security` object that defines the global method we're using for 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 \(page 322\)](#). 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 \(page 331\)](#), 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
```

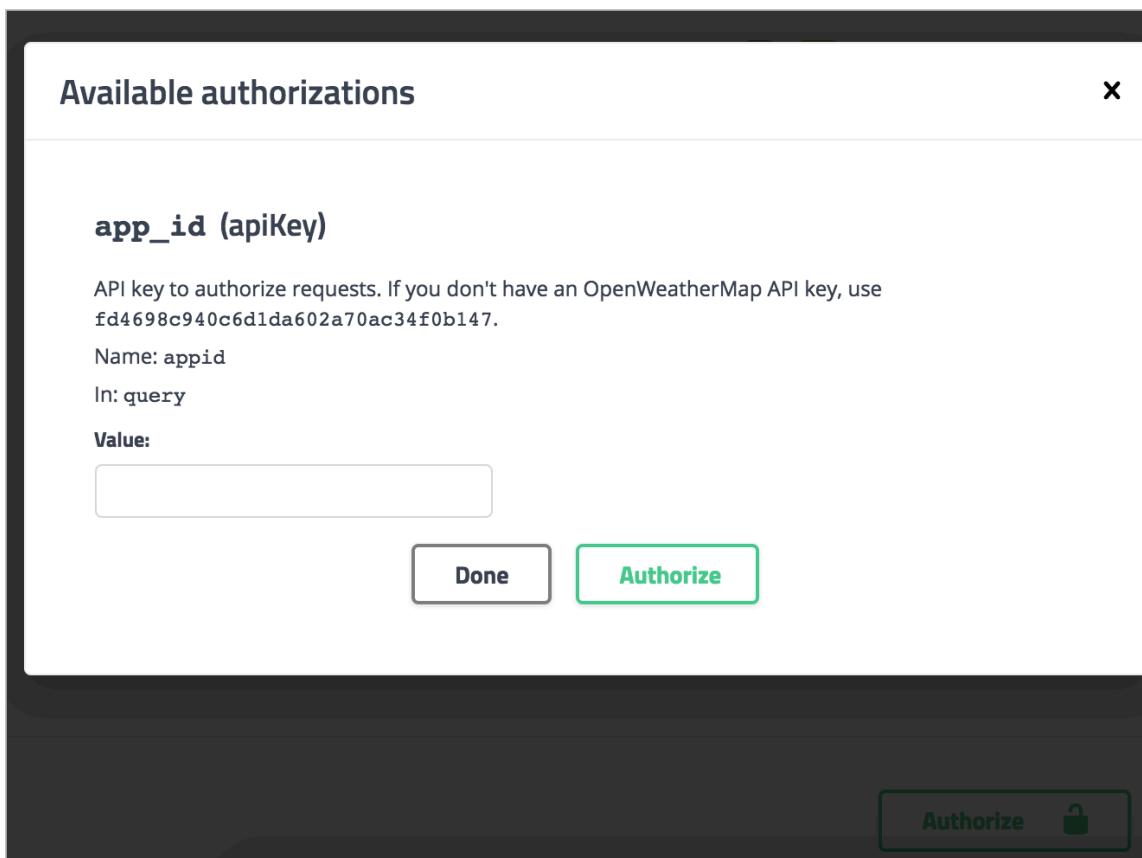
Properties you can use in the `securitySchemes` object include the following:

- `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.

## Swagger UI appearance

In the Swagger UI, you see the `description` and other security details in the Authorization modal (which appears when you click the Authorization button):



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.

## Checking to see if authorization works

When you submit a request, Swagger UI shows you the curl request that is submitted. For example, after executing a weather request, the curl is as follows:

```
curl -X GET "http://api.openweathermap.org/data/2.5/weather?zip=95050%2Cus&appid=fd4698c940c6d1da602a70ac34f0b147&units=imperial"
```

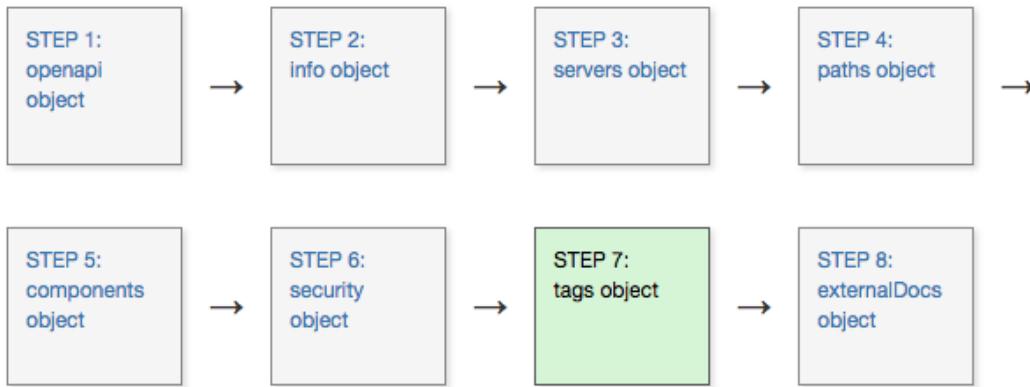
The `&appid=fd4698c940c6d1da602a70ac34f0b147` indicates that the API key is being included in the query string. (For more on curl, see [Make a curl call \(page 54\)](#).)

## Troubleshooting 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.

# 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 \(page 322\)](#)).

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 used the same tag `Weather`:

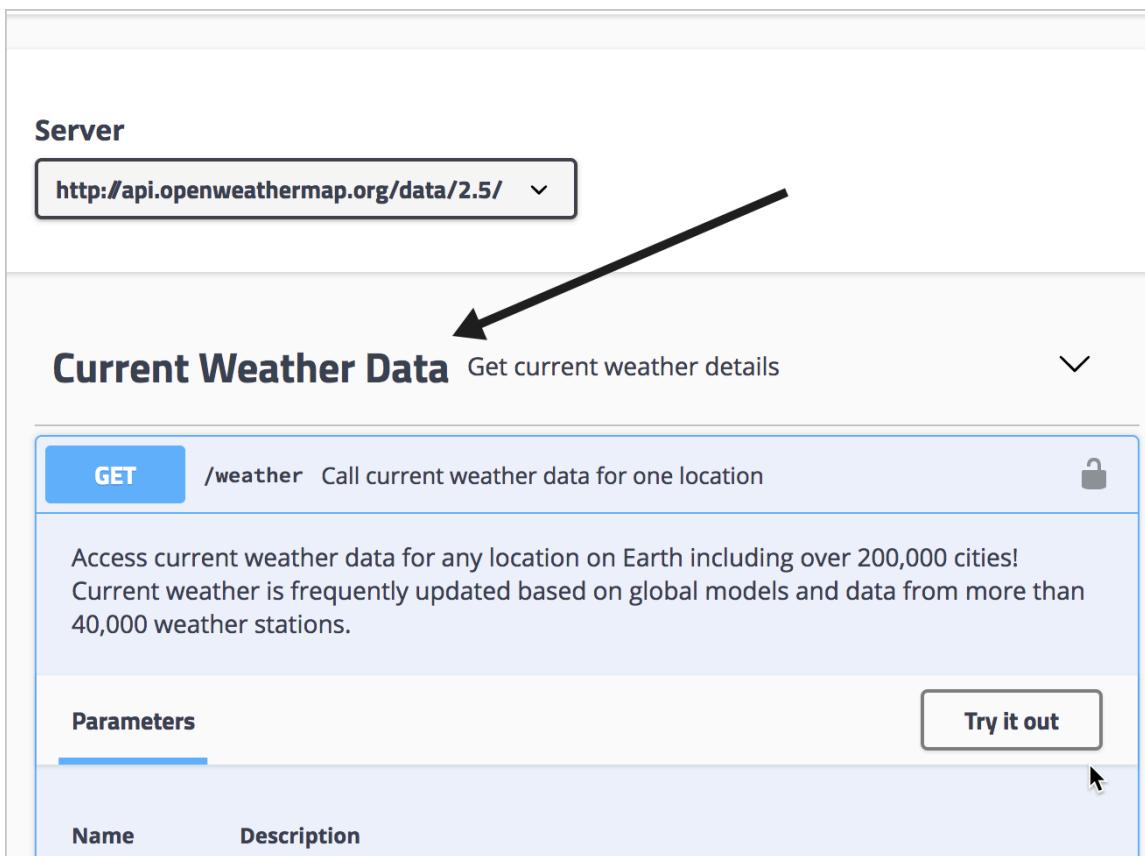
```

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

```

## How tags appear in Swagger UI

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

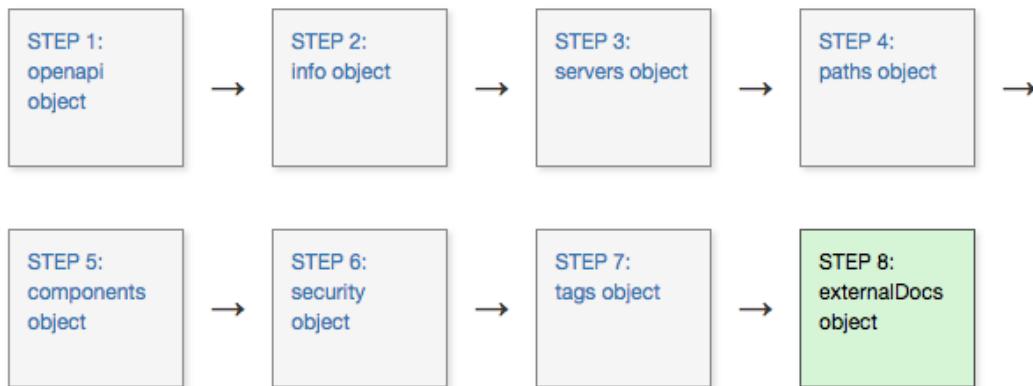


The screenshot shows the Swagger UI interface for a weather API. At the top, there's a 'Server' section with a dropdown menu containing the URL `http://api.openweathermap.org/data/2.5/`. Below this, a section titled 'Current Weather Data' is expanded, indicated by a downward arrow icon. This section contains a 'GET /weather' operation description: 'Call current weather data for one location'. To the right of the description is a lock icon. Below the operation, there's 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.' On the left side of this description, there's a 'Parameters' heading. To the right of the description is a 'Try it out' button with a cursor hovering over it. At the bottom of the screenshot, there's a table header with columns 'Name' and '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 display 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
```

In the Swagger UI, this link appears after the API description along with other info about the API.

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 a search bar containing the URL '/learnapidoc/docs/rest\_api\_s...'. To the right of the search bar is a 'Explore' button. Below the header, the title 'OpenWeatherMap API' is displayed in large, bold, dark letters, accompanied by a '2.5' badge and an 'OAS3' badge. Underneath the title is a blue link: '/learnapidoc/docs/rest api specifications/openapi\_openweathermap.yml'. A descriptive paragraph follows, explaining the API's capabilities: '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.' It includes a note: 'Note: This sample Swagger file covers the **current** endpoint only from the OpenWeatherMap API.' Below this, another note 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.' At the bottom of the page, there are several links: 'Terms of service', 'OpenWeatherMap API - Website', 'CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)', and 'API Documentation' with a black arrow pointing left.

See the related topic, [Integrating Swagger UI with the rest of your docs \(page 366\)](#) 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 document. You can see the end result here: [docs/rest\\_api\\_specifications/openapi\\_weather.yml](#).

Here's the specification document embedded in Swagger UI:

The screenshot shows the Swagger UI interface for the OpenWeatherMap API. At the top, there's a green header bar with the 'swagger' logo, a search bar containing the URL '/learnapidoc/docs/rest\_api\_specifications/openapi\_openweathermap.yml', and an 'Explore' button. Below the header, the title 'OpenWeatherMap API' is displayed with a '2.5 OAS3' badge. 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. Note: This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.' There are also links for 'Terms of service', 'OpenWeatherMap API - Website', 'CC Attribution-ShareAlike 4.0 (CC BY-SA 4.0)', and 'API Documentation'. On the right side of the main content area, there's a 'Authorize' button with a lock icon. In the bottom left corner of the main content area, there's a 'Server' dropdown menu set to 'http://api.openweathermap.org/data/2.5/'. Below this, under the heading 'Current Weather Data', there's a section for 'GET /weather Call current weather data for one location'. The entire interface is contained within a light gray frame.

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.

# Activity: Create an OpenAPI specification document

The [OpenAPI tutorial \(page 310\)](#) 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.

## Identify an API

In an earlier activity, you [found an open-source API project \(page 31\)](#) with some documentation needs. Create an OpenAPI specification for this API.

If you don't want to use that API (maybe it already has an OpenAPI specification document, or there are other complications), you can use this simple [Sunrise and sunset times API](#). This Sunrise and sunset times API doesn't require authentication with requests, so it removes some of the more complicated authentication workflows.

Depending on the API you choose to work with, you could potentially use this specification document as part of your portfolio.

## Follow the OpenAPI tutorial

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

- [Step 1: openapi object \(page 314\)](#)
- [Step 2: info object \(page 318\)](#)
- [Step 3: servers object \(page 320\)](#)
- [Step 4: paths object \(page 322\)](#)
- [Step 5: components object \(page 331\)](#)
- [Step 6: security object \(page 345\)](#)
- [Step 7: tags object \(page 349\)](#)
- [Step 8: externalDocs object \(page 351\)](#)

## Make sure your spec validates

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

## Check your spec against mine

If you get stuck or want to compare your spec with mine, see [openapi\\_sunrise\\_sunset.yml \(page 0\)](#).

Note that the Sunrise and sunset times API doesn't require authorization, so you can skip [Step 6: security object \(page 345\)](#).

You can use this OpenAPI specification document when working through the [Swagger UI activity \(page 363\)](#).

# Swagger UI tutorial

Swagger UI provides a display framework that reads the [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 \(page 298\)](#).

For step-by-step tutorial on creating an OpenAPI specification document, see the [OpenAPI tutorial \(page 310\)](#).

## Terminology notes

First, let's clarify a few terms:

- [Swagger](#) was the original name of the spec, but the spec was later changed to [OpenAPI](#) to reinforce the open, non-proprietary nature of the standard. People sometimes refer to both names interchangeably (esp. on older web pages), but “OpenAPI” is how the spec should be referred to.
- The OpenAPI spec is driven by the [OpenAPI initiative](#), backed by the Linux Foundation and steered by [many companies and organizations](#). The Swagger YAML file that you create to describe your API is called the “OpenAPI specification document.”
- 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](#).
- [SwaggerHub](#) is the more fully featured, commercial version of the open-source Swagger UI. See [Swagger UI on Swagger.io](#) for a feature comparison.

For more details, see [What Is the Difference Between Swagger and OpenAPI?](#) and the [API Glossary \(page 459\)](#).

This tutorial focuses on Swagger UI. For a deep dive into the OpenAPI spec, see my [OpenAPI tutorial here \(page 310\)](#).

## 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 to Swagger UI works and how to integrate an OpenAPI specification document into it.

## 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](#).

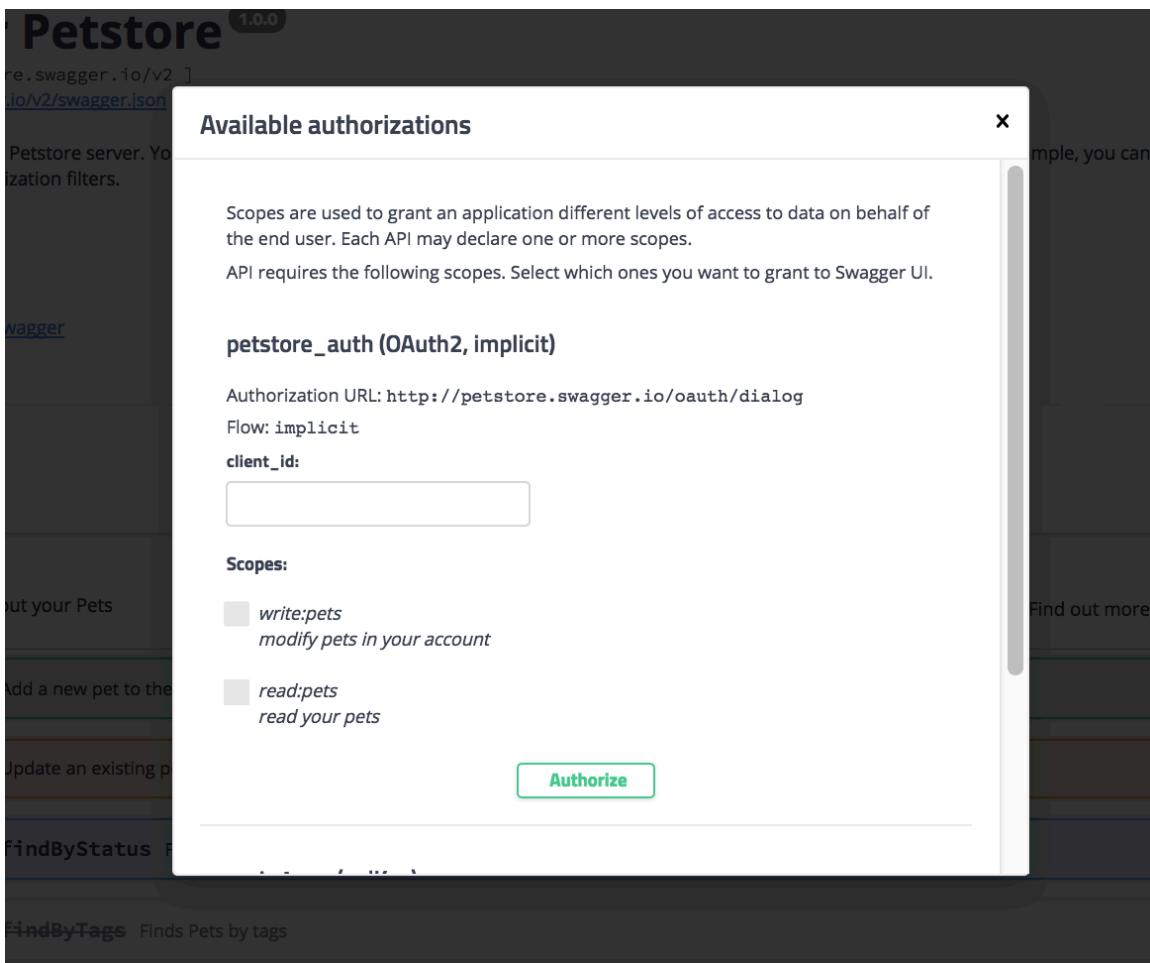
The screenshot shows the Swagger Petstore interface. At the top, there's a navigation bar with the Swagger logo, the URL <http://petstore.swagger.io/v2/swagger.json>, and an **Explore** button. Below the navigation is a header for the **Swagger Petstore 1.0.0** API. It includes links for Terms of service, Contact the developer, Apache 2.0, and Find out more about Swagger. A note 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." On the left, there's a dropdown for Schemes with "HTTP" selected. To the right is an **Authorize** button with a lock icon. The main content area shows a section for the **pet** tag, which is described as "Everything about your Pets". It lists four endpoints: a green **POST** button for "/pet" (Add a new pet to the store), an orange **PUT** button for "/pet" (Update an existing pet), a blue **GET** button for "/pet/findByStatus" (Finds Pets by status), and another blue **GET** button for "/pet/findByTags" (Finds Pets by tags). Each endpoint has a lock icon to its right.

The endpoints are grouped into three tags:

- [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 demo 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**

<b>body</b> <small>* required</small>	Pet object that needs to be added to the store <i>(body)</i>
---------------------------------------	-----------------------------------------------------------------

**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**

<b>body</b> <small>* required</small>	Pet object that needs to be added to the store <i>(body)</i>
---------------------------------------	-----------------------------------------------------------------

**Example Value Model**

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

**Parameter content type**  
application/json

**Execute**

Swagger UI submits the request and shows the [curl that was submitted \(page 54\)](#). The Responses section shows the [response \(page 121\)](#). (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 operation. At the top, there's a 'Responses' tab and a dropdown for 'Response content type' set to 'application/json'. Below that is a 'Curl' section containing a curl command to post a pet. Under 'Request URL', the address 'http://petstore.swagger.io/v2/pet' is listed. In the 'Server response' section, it shows a code block for status 200 with the following JSON response body:

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

**Important:** The Petstore is a functioning API, and you have actually created a pet. You now need to take responsibility for your pet and begin feeding and caring for it! All joking aside, most users don't realize they're playing with real data when they execute responses in an API (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.

## Create a Swagger UI display with an OpenAPI spec document

In this activity, you'll create a Swagger UI display for the weather endpoint in this [OpenWeatherMap API](#). (If you're jumping around in the documentation, this is a simple API that we used in earlier parts of the course.) You can see a demo of what we'll build [here](#).

The screenshot shows the Swagger UI interface for the Weather API from Mashape. At the top, there's a green header bar with the 'swagger' logo, a search bar containing 'swagger\_weather.yml', and a 'Explore' button. Below the header, the title 'Weather API from Mashape' is displayed with a '2.0.0' badge. A note indicates the base URL is simple-weather.p.mashape.com and the file is swagger\_weather.yml. A descriptive text explains the API displays forecast data by latitude and longitude. Below this, there are links to 'Terms of service', 'Contact the developer', 'Apache 2.0', and 'Find out more about this Weather API'. On the left, a 'Schemes' dropdown is set to 'HTTPS'. On the right, there's an 'Authorize' button with a lock icon. The main content area is divided into sections: 'Air Quality' and 'Weather Forecast'. The 'Air Quality' section contains a single endpoint: GET /aqi Gets the air quality index. The 'Weather Forecast' section contains two endpoints: GET /weather Gets the weather forecast in abbreviated form and GET /weatherdata Get weather forecast with lots of details.

You can also follow instructions for working with Swagger UI [here in the Swagger.io docs](#).

### To integrate your OpenAPI spec into Swagger UI:

1. If you don't already have an OpenAPI specification document, follow the [OpenAPI tutorial here \(page 310\)](#) to create one. The tutorial here focuses on Swagger UI, so for convenience, copy [this sample OpenAPI file](#) by right-clicking the link and saving the file ("openapi\_weather.yml") to your desktop.

If you want to preview what your Swagger UI implementation will look like ahead of time, copy the content from the OpenAPI specification document you just downloaded into the [Swagger online editor](#). The view on the right of the Swagger Editor shows a fully functional Swagger UI display.

2. Go to the [Swagger UI GitHub project](#).
3. Click **Clone or download**, and then click **Download ZIP** button. Download the files to a convenient location on your computer and extract the files.

The only folder you'll be working with here is the **dist** folder (short for distribution). Everything else is used only if you're regenerating the files, which is beyond the scope of this tutorial.

4. Drag the **dist** folder out of the swagger-ui-master folder so that it stands alone. Then delete the swagger-ui-master folder and zip file.
5. Inside your **dist** folder, open **index.html** in a text editor such as Atom or Sublime Text.
6. Look for the following code:

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

7. Change the **url** value from `http://petstore.swagger.io/v2/swagger.json` to the following:

```
url: "openapi_weather.yml",
```

Save the file.

8. Drag the **openapi\_weather.yml** file that you downloaded earlier into the same directory as the **index.html** file you just edited. Your file structure should look as follows:

```
└── swagger
 ├── favicon-16x16.png
 ├── favicon-32x32.png
 ├── index.html
 ├── oauth2-redirect.html
 ├── swagger-ui-bundle.js
 ├── swagger-ui-bundle.js.map
 ├── swagger-ui-standalone-preset.js
 ├── swagger-ui-standalone-preset.js.map
 ├── swagger-ui.css
 ├── swagger-ui.css.map
 ├── swagger-ui.js
 ├── swagger-ui.js.map
 └── swagger30.yml
 └── openapi_weather.yml
```

9. Upload the folder to a web server and go to the folder path. 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.)

You can also view the file locally in your browser. It's also common to run a local web server to view the Swagger UI site. To run a local web server on your computer, you can use a [simple Python http server](#) or a more robust local server such as [XAMPP](#).

Swagger UI provides a number of [parameters](#) you can use to customize the 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, and more.

## Challenges with Swagger UI

As you explore Swagger UI, you may notice a few limitations with the approach:

- 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 \(page 322\)](#) and the [info object \(page 318\)](#).
- The Swagger UI looks mostly the same for each output. You can modify the source files and regenerate the output, but doing so requires more advanced coding 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 \(page 366\)](#) for more details on workarounds. You can [customize Swagger UI](#) with your own branding, but it will require some deeper UX skills.

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

For other tools and libraries, see [Swagger services and tools](#) and [Open Source Integrations](#).

## Sorting out the various Swagger tools

As I explained earlier, [Swagger](#) refers to the various API tools built around the [OpenAPI spec](#). Here's a quick summary of what Swagger tools are available:

- **Swagger editor:** The Swagger Editor is an online editor that validates your OpenAPI document against the rules of the OpenAPI spec. You'll need to be familiar with OpenAPI specification to be successful here. The Swagger editor will flag errors and give you formatting tips. See my [OpenAPI tutorial \(page 310\)](#) for a step-by-step walkthrough.
- **Swagger-UI:** The Swagger UI is an HTML/CSS/JS framework 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:** This tool 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.
- **Swaggerhub:** The commercial version of the open-source Swagger UI project.

For more tools, see [Swagger Tools](#).

# Activity: Create your own Swagger UI display

In the [Swagger tutorial \(page 355\)](#), you explored the [Swagger Petstore site](#), learned how the [Swagger Editor](#) works, and how to download and integrate your [OpenAPI specification document \(page 310\)](#) into [Swagger UI](#). Now it's time to put this learning into practice by integrating the specification document into your own Swagger UI site.

## Integrate your OpenAPI spec into Swagger UI

1. If you completed the [OpenAPI specification document activity \(page 354\)](#), you should have a functional specification document from an API. If you don't, you can use the [openapi\\_sunrise\\_sunset.yml \(page 0\)](#). This spec's information comes from the [Sunset and sunrise times API](#). Download the file to your computer.
2. Go to the [Swagger UI GitHub project](#) and click **Clone or download**, then click **Download ZIP**.
3. Uncompress the downloaded swagger-ui-master.zip and move the **dist** folder into another directory on your computer. Give the dist folder a more meaningful name, such as "swagger."
4. Move your OpenAPI specification document (e.g., openapi\_sunrise\_sunset.yml) into the folder.
5. Open the **index.html** file in the folder and update the reference to swagger.json to your own specification document.

Change this:

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

to this:

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

6. To run Swagger UI locally, some browsers (such as Chrome or Safari) may block the local JavaScript. Try opening the index.html file in Firefox. If it doesn't load without error in Firefox, either upload the project to a web server or [run a local web server \(page 363\)](#) in that folder.

## Run a Local Web Server

A local web server simulates a web server on your local machine. To run a simple Python web server in the directory, you can follow use this [simple HTTP server](#).

1. Check that you have Python installed by opening a terminal prompt and typing `Python -V`. If you don't have Python, [download and install it](#).
2. Using your terminal, `cd` into the same directory as your Swagger UI project.
3. Do one of the following:

If you have Python 2.x, run the following command:

```
python -m SimpleHTTPServer 7800
```

If you have Python 3.x, run the following command:

```
python -m http.server 7800
```

4. Open a browser (such as Chrome), and go to <http://localhost:7800/>. The web server serves up the index.html file in that directory by default. You should see your Swagger UI project load the specification document.

The screenshot shows the Swagger UI interface for a "Sunset and sunrise times API". The top navigation bar includes a logo, the title "swagger", a file selector "openapi\_sunrise\_sunset.yml", and a "Explore" button. Below the title, the API name "Sunset and sunrise times API" is displayed with an "1.0 OAS3" badge. A link to "openapi\_sunrise\_sunset.yml" is shown. A descriptive text states: "We offer a free API that provides sunset and sunrise times for a given latitude and longitude." Below this, links to "API team - Website" and "Documentation and main site" are provided. A cursor arrow points towards the "Documentation and main site" link. In the middle section, there's a "Servers" dropdown set to "https://api.sunrise-sunset.org". Under the "default" endpoint, a "GET /json Get sunrise/sunset times" operation is listed. At the bottom, a "Models" dropdown is shown. The entire interface is contained within a light gray frame.

## Test your Swagger UI project

Make a request with the Swagger UI display to make sure it's working. If you're using the [Sunset and sunrise times API example \(page 0\)](#), you can use these values for latitude and longitude:

- lat: [37.3710062](#)
- lng: [-122.0375932](#)

# Swagger UI Demo

The following is an embedded instance of the [Swagger UI](#) showing the OpenAPI file for the Mashape weather API.

This page can only be viewed online in your computer's web browser. Go to [http://idratherbewriting.com/learnapidoc/pubapis\\_swagger\\_demo.html](http://idratherbewriting.com/learnapidoc/pubapis_swagger_demo.html) to view it.

# 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 while remaining hard for engineers or other non-writers to grasp. 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 \(page 160\)](#) part of the guide. You have sections such as the following:

- getting started
- Hello World tutorial
- demos
- authentication and authorization
- response and error codes
- code samples and tutorials
- quick reference guide
- troubleshooting
- glossary

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 \(page 355\)](#) 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 `420006397` 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 2 \(embedded\) \(page 365\)](#). 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 Models section, particularly the weatherdata response, 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 \(page 365\)](#) and check out the second `<style>` block, you can see the styles I haphazardly added.

I like the embedded option, because it means 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 \(page 310\)](#), which is something many tools don't yet support. Additionally, Swagger UI has the familiar interface that API developers are probably already familiar with.

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

You can try to put all information into your spec. 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.

 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> Tip: 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> Note: 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">
 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. (Reference the needed JavaScript in the header or footer of the same index.html file where you referenced your Swagger.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, I recommend trying to put 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 with zero 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 \(page 372\)](#), 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 read the OpenAPI specification document. It 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 is probably the best way to single source the content.

## 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 the [MTool options for developer docs \(page 245\)](#). 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 content from it. Although Ellis is still working on this approach, if he's successful it could be a huge win at integrating tech comm tools with API specification formats.

## 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.

The truth is that programmers have been operating this way for years with [Javadocs \(page 429\)](#), [Doxygen \(page 446\)](#), and other document-generator tools that generate documentation from Java, C++, or C# files. 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.

# SwaggerHub introduction and tutorial

Previously, I explored using the open-source [Swagger UI project \(page 355\)](#) as a way to render your [OpenAPI specification document \(page 310\)](#). **SwaggerHub** is the commercial version of Swagger UI. You can see a comparison of features [here](#).

You can see a demo of the [sample OpenWeatherMap API on SwaggerHub here](#).

## Advantages of SwaggerHub

While the open-source Swagger 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](#) — the same company that maintains and develops the open source Swagger tooling ([Swagger Editor](#), [Swagger UI](#), [Swagger Codegen](#), and others), and that formed the [OpenAPI Initiative](#) which leads the evolution of the [Swagger \(OpenAPI\) specification](#) — 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 like back, forward, and search. A top bar shows user information: "tomjohit-idbw" with a gear icon. The main area displays a list of APIs:

- open-weather\_map\_api**  
PUBLIC | PUBLISHED  
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,  
[API] [OAS3]
- MashapeWeatherAPI**  
PUBLIC | UNPUBLISHED  
MashapeWeatherAPI - [2.3]  
[API]
- sample**  
PUBLIC | UNPUBLISHED  
This is an example of using OAuth2 Access Code Flow in a specification to describe security to your API.  
[API]

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

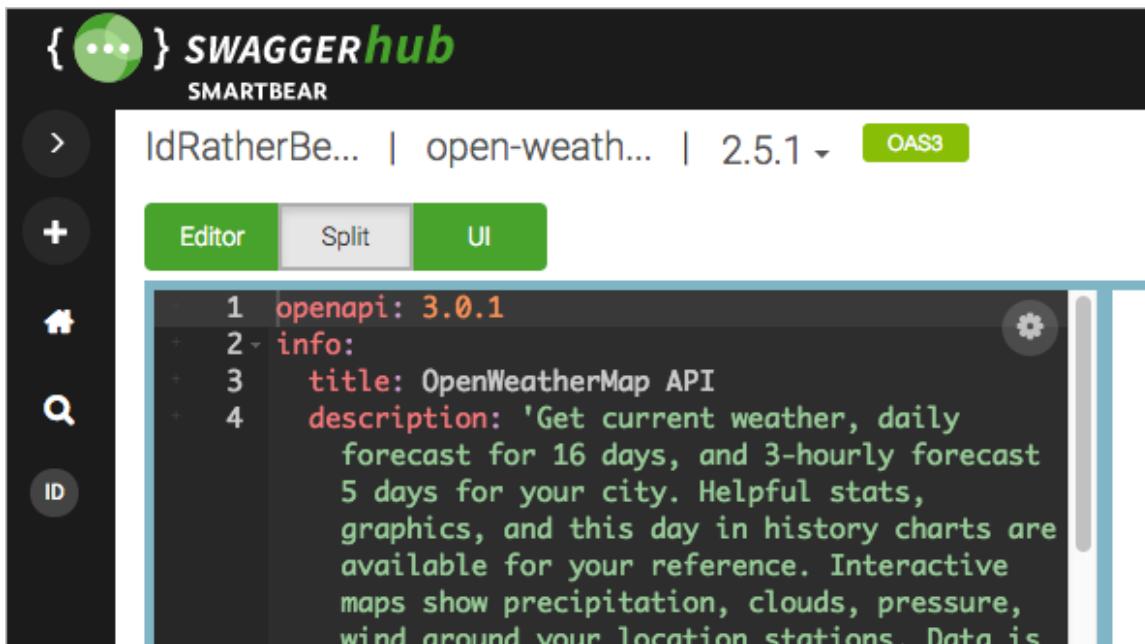
The dashboard shows a list of the APIs you've created. In this example, you see the [OpenWeatherMap API \(page 40\)](#) that I've been using throughout this course.

## 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)

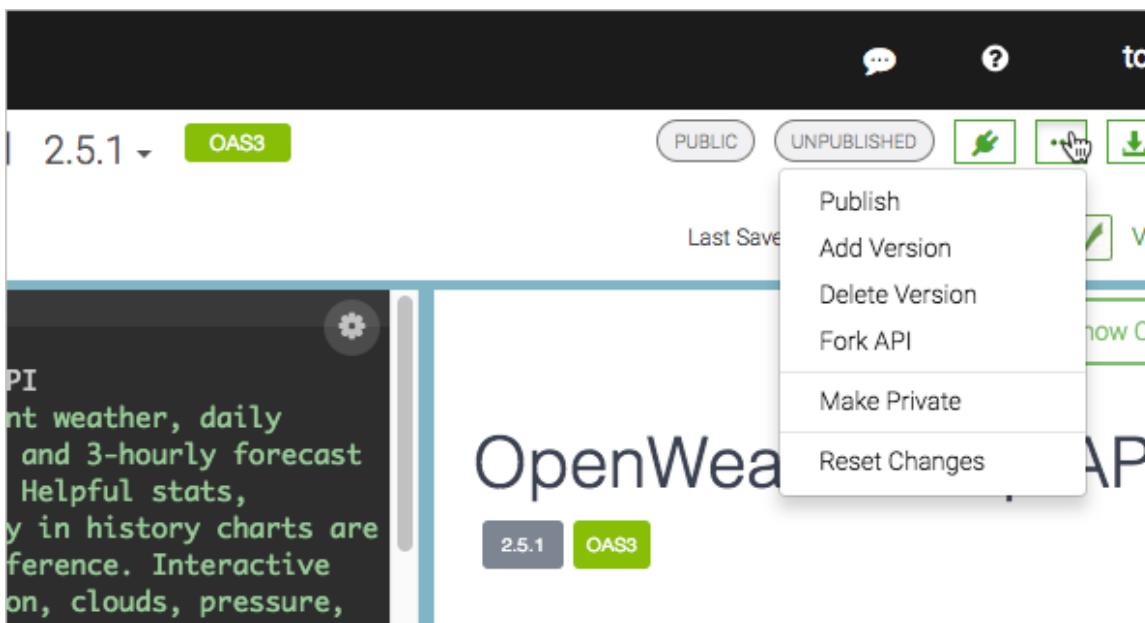


Most importantly, as you're working in the Editor, SwaggerHub allows you to save *your work*. With the free Swagger Editor, your content is just kept in the browser cache, with no ability to save it. When you clear your cache, your content is gone. As a result, if you use the standalone Swagger Editor, 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 3.0 of the spec: [https://app.swaggerhub.com/apis/IdRatherBeWriting/open-weather\\_map\\_api/2.5](https://app.swaggerhub.com/apis/IdRatherBeWriting/open-weather_map_api/2.5).

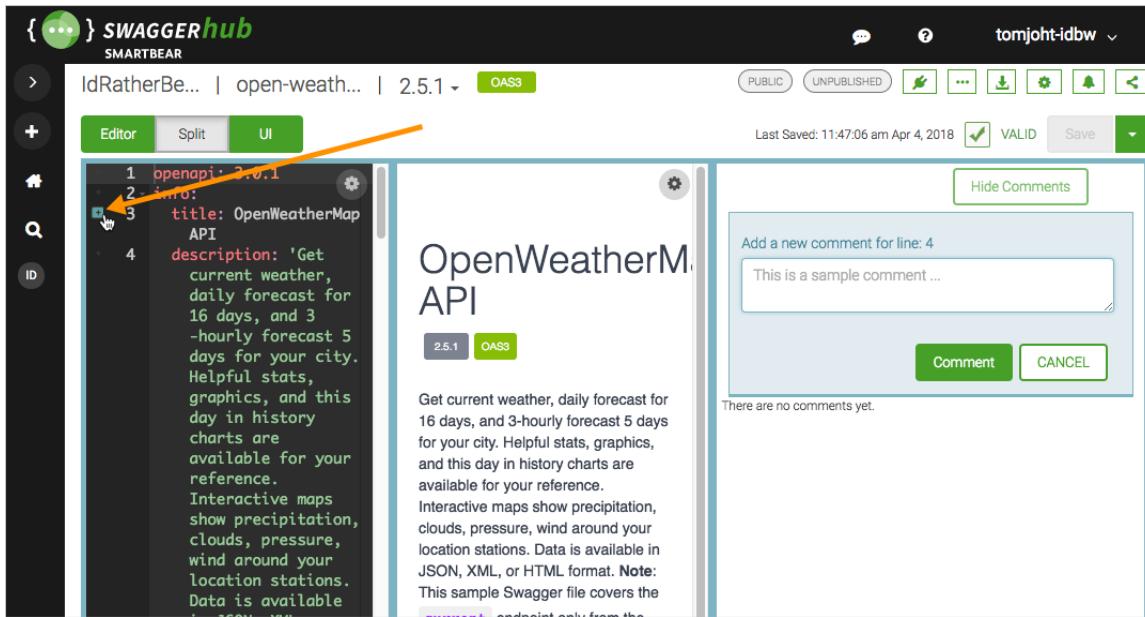
You can also send users to the default version when they go they don't include the version at the end. For example, if you go to [https://app.swaggerhub.com/apis/IdRatherBeWriting/open-weather\\_map\\_api/](https://app.swaggerhub.com/apis/IdRatherBeWriting/open-weather_map_api/), you get forwarded to the latest published version of the spec automatically.

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, the base URL remain as a subdirectory on app.swaggerhub.com. 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.



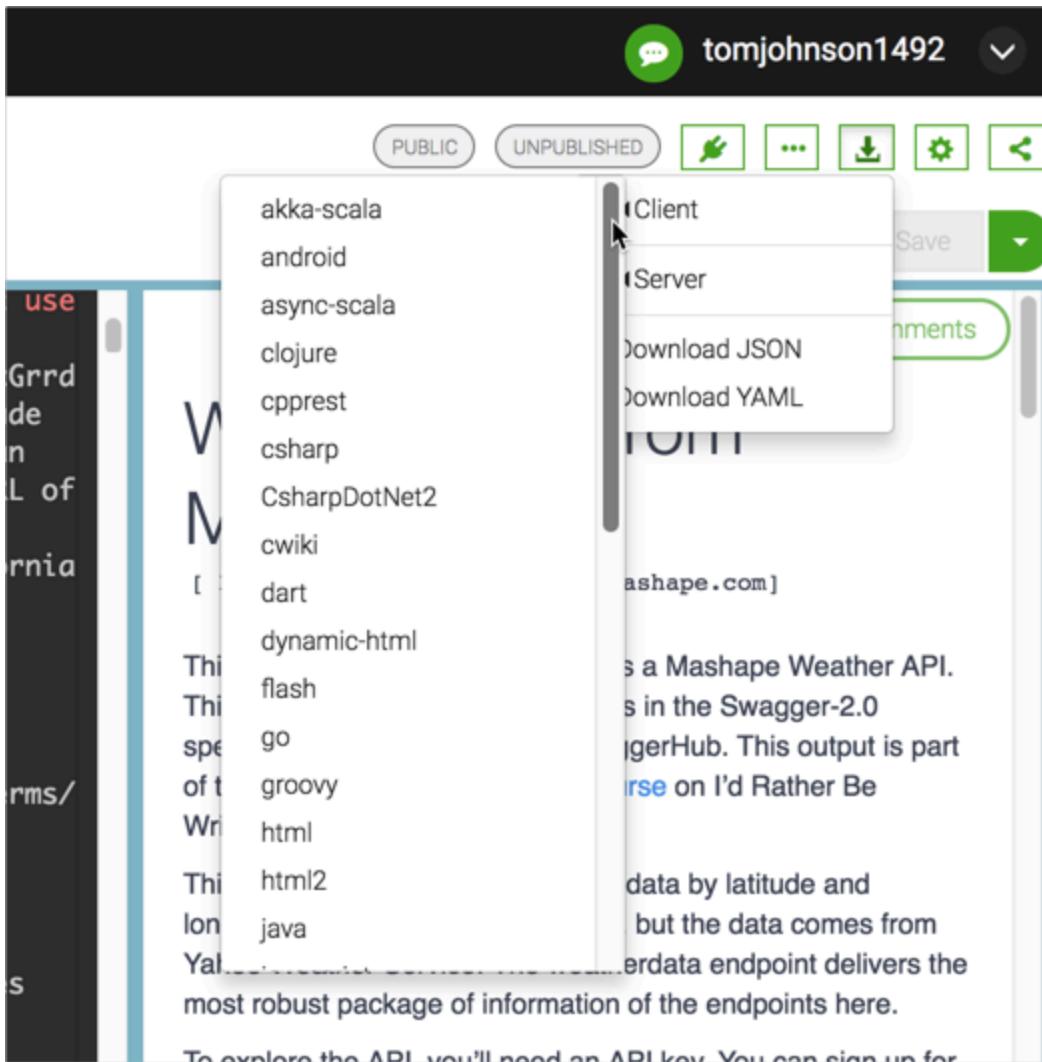
When you click the plus sign, a comment pane appears on the right where you can elaborate on comments, and where others can reply. Users can edit, delete, or resolve the comments. This commenting feature helps facilitate the review process in a way that tightly integrates with your content. You can also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

One of the options for export 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 \(page 366\)](#).

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 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](#)) 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.

## 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.

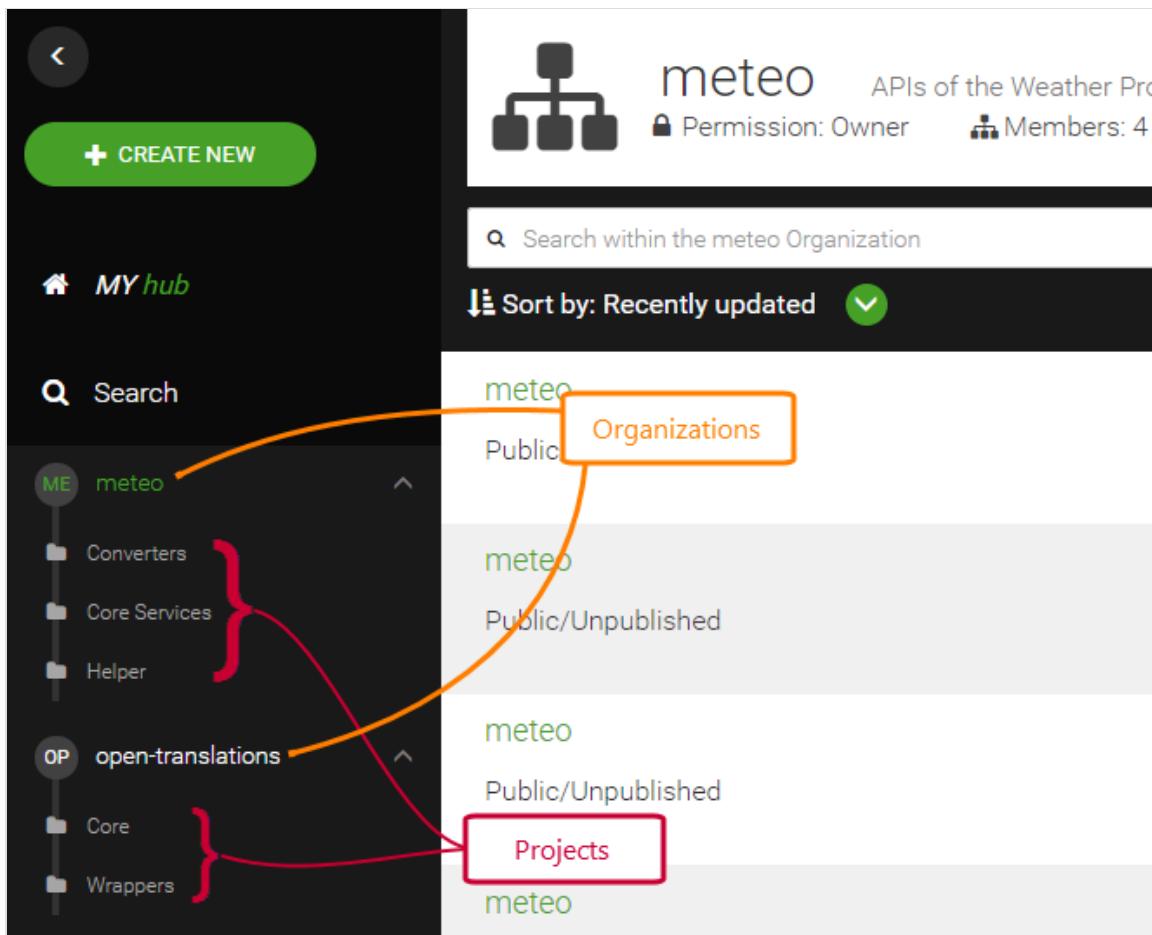
When you create definitions for your requests and responses, you may find yourself re-using the same code over and over. Rather than duplicating this code, you can save it as a domain. When you want to re-use the code, you select this domain.

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 biggest 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 when you consider how you'll scale and grow as you have dozens of APIs and multiple teams, organizations and projects 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.](#)

*Note that SwaggerHub is one of the sponsors of this site.*

# Other tools to parse and display OpenAPI specs

In this course, I've emphasized using [Swagger UI \(page 355\)](#) to parse and display the [OpenAPI specification \(page 298\)](#) as interactive documentation. However, there are many other tools that can read OpenAPI specification documents. That's the whole idea of a standard — when you create a standard way of describing APIs, many tools can predictably read the description and generate documentation (and other tooling) based on it.

## List of Commercial Tools

Swagger has a detailed list of tools that can read the OpenAPI spec here: [Commercial Tools](#). You can see the many [open-source tools here](#).

Below I'll expand on a couple of others, and add more over time.

I'm currently developing out the content here. Currently, the list tools and info is anemic.

## Spectacle

[Spectacle](#) is an open-source Github project that builds an output from an OpenAPI specification file. The display provides a three-pane output similar to the Stripe or Slate docs. After you download the project files, you can build the display using Node simply by referencing your OpenAPI spec file.

Here's a [demo output](#) using the sample OpenWeatherMap API:

The screenshot shows the Spectacle application interface with three main panes:

- TOPICS** pane (left): A sidebar listing categories: Introduction, Authentication, PATHS, Call current weather data for..., SCHEMA DEFINITIONS, Units, Lang, Mode, SuccessfulResponse, Coord, Weather, Main, Wind, Clouds, Rain, Snow, Sys.
- OpenWeatherMap API** pane (center):
  - Introduction**: Describes the API's purpose and data availability.
  - Note**: Mentions the `current` endpoint.
  - Authentication**: Details the `appid` authentication method.
  - API key**: Provides instructions for using an OpenWeatherMap API key.
- API Endpoint** pane (right):
  - API Endpoint: `http://api.openweathermap.org/data/2.5/`
  - Request Content-Type: `application/json`
  - Response Content-Type: `application/json`
  - Schemes: `http`
  - Version: `2.5`

You can also see their [cheesestore demo](#).

With almost no needed setup or configuration, Spectacle gives you a world-class output and site for your API reference docs. As long as the [OpenAPI spec \(page 310\)](#) that you integrate is fully detailed, the generated Spectacle site will be attractive and full-featured. (Spectacle doesn't allow you to add custom pages for other tutorials or conceptual docs.)

You can also build the Spectacle site without the framed layout so you can embed it into another site. However, in playing with this embed option, I found that to do this, I'd have to create my own styles. If using the default styles in the full-site output, they most likely will overwrite or interfere with your host site's appearance.

## Run in Postman button

Postman is a REST API GUI client that we explored earlier in [Submit requests through Postman \(page 46\)](#). The [Run in Postman button](#) provides a button (labeled "Run in Postman") that, when clicked, imports your API info into Postman so users can run calls using the Postman client. As such, this isn't a full-fledged authoring tool but rather a way to import the interactive, try-it-out API explorer for your endpoints into a web page.

To try out Run in Postman, first [import your 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](#).

For a demo of Run in Postman using the sample OpenWeatherMap API, go to [idratherbewriting.com/learnapidoc/pubapis\\_docs\\_as\\_code\\_tool\\_options.html#postman](http://idratherbewriting.com/learnapidoc/pubapis_docs_as_code_tool_options.html#postman).

Postman 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 you don't have the in-browser experience to try out calls, in many ways the Postman client is more useful. This is what 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 integrate your API into a client they can use. It gives them 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.

# More about YAML

When you created the [OpenAPI specification \(page 310\)](#), you usually use a syntax called YAML. The file extension can be either “.yaml” or “.yml.” (You can also use [JSON \(page 65\)](#), 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/paste/

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

YAML and JSON are practically 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 they're non-standard). Instead, you space twice.

Each level can contain either a single key-value pair (also referred to as a dictionary) 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 or not. If your value has something like a colon or quotation mark in it, you'll want to enclose it in quotation marks. And if there's a double quotation mark, enclose the value in single quotation marks, or vice versa.

## 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
```

These key-value pairs are also called dictionaries.

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

Here's the same array containing objects converted to YAML:

```
-
 name: Tom
 age: 39
-
 name: Shannon
 age: 37
```

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.

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`.

For details on other differences, see [Learn YAML in Minutes](#). To learn more about YAML, see this [YAML tutorial](#).

# RAML tutorial

RAML stands for REST API Modeling Language and is similar to the [OpenAPI specification \(page 310\)](#).

RAML is backed by [Mulesoft](#), a full-service, end-to-end API company, and uses a more YAML-based syntax in the specification.

Unless you're publishing your docs with Mulesoft or another platform that specifically requires RAML, I recommend using the [OpenAPI specification \(page 310\)](#) instead. However, Mulesoft offers enterprise-grade API design, management, and deployment capabilities. If you're using Mulesoft for your API, use RAML for your documentation specification.

## RAML overview

Similar to OpenAPI, after you create a RAML file that describes your API, it can be consumed by different platforms to parse and display the information in attractive outputs. The RAML format, which uses YML syntax, is human-readable, efficient, and simple. Here's the OpenWeatherMap API file described in RAML file and rendered with the API Console:

The screenshot shows the API Console interface for the OpenWeatherMap API. The left sidebar has a tree view with nodes like 'Documentation', 'Types', and 'Resources'. Under 'Resources', the '/weather' node is expanded, showing a 'Request' section with a 'GET' method and its URL. To the right, there's a detailed table for the '/weather' endpoint's parameters:

Parameter	Type	Description
q	string	<b>City name.</b> Example: London. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by comma; use ISO 3166 country codes.
id	string	<b>City ID.</b> Example: 2172797. You can call by city ID. API responds with exact result. The List of city IDs can be downloaded <a href="#">here</a> . You can include multiple cities in parameter – just separate them by commas. The limit of locations is 20. Note: A single ID counts as a one API call. So, if you have city IDs, it's treated as 3 API calls.
lat	string	<b>Latitude.</b> Example: 35. The latitude coordinate of the location of your interest. Must use with lon.
lon	string	<b>Longitude.</b> Example: 139. Longitude coordinate of the location of your interest. Must use with lat.
zip	string	<b>Zip code.</b> Search by zip code. Example: 95050,us. Please note if country is not specified then the search works for USA as a default. Default value: 94040,us Example value: 94040,us
units	string (enum)	<b>Units.</b> Example: imperial. Possible values: metric, imperial. When you do not use units parameter, format is standard by default.

At the bottom left, it says 'Powered by MuleSoft'.

This is a sample RAML output in something called API Console

## Auto-generating client SDK code

It's important to note that with these REST API specifications (not just RAML), you're not just describing an API to generate a nifty doc output with an interactive console. There are tools that can also generate client SDKs and other code from the spec into a library that you can integrate into your project. This can help developers to more easily make requests to your API and receive responses.

Additionally, the interactive console can provide a way to test out your API before developers code it. Mulesoft offers a “mocking service” for your API that simulates calls at a different baseURI. The idea of the mocking service is to design your API the right way from the start, without iterating with different versions as you try to get the endpoints right.

## Sample spec for OpenWeatherMap API

To understand the proper syntax and format for RAML, you need to read the [RAML spec](#) and look at some examples. See also [this RAML tutorial](#) and this [video tutorial](#).

Here's the OpenWeatherMap API we've been using in this course formatted in the RAML spec. (I actually just used [API Transformer](#) to convert my OpenAPI 3.0 spec to RAML.) It's highly similar to the OpenAPI spec.

```
#%RAML 1.0
title: OpenWeatherMap API
version: 2.5
baseUri: http://api.openweathermap.org/data/2.5/
baseUriParameters: {}
documentation:
- title: OpenWeatherMap API
 content: 'Get current weather, daily forecast for 16 days, and 3-hourly fo
recast 5 days for your city. Helpful stats, graphics, and this day in histor
y charts are available for your reference. Interactive maps show precipitati
on, 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: Al
l parameters are optional, but you must select at least one parameter. Calli
ng the API by city ID (using the `id` parameter) will provide the most preci
se location results.'
securitySchemes:
 auth:
 type: Pass Through
 describedBy:
 queryParameters:
 appid:
 required: true
 displayName: appid
 description: API key to authorize requests. If you don't have an O
penWeatherMap API key, use `fd4698c940c6d1da602a70ac34f0b147`.
 type: string
types:
 SuccessfulResponse:
 displayName: Successful response
 type: object
 properties:
 coord:
 required: false
 displayName: coord
 type: Coord
 weather:
 required: false
 displayName: weather
 description: (more info Weather condition codes)
 type: array
 items:
 type: Weather
 base:
 required: false
 displayName: base
 description: Internal parameter
 type: string
 main:
 required: false
 displayName: main
```

```
 type: Main
 visibility:
 required: false
 displayName: visibility
 description: Visibility, meter
 type: integer
 format: int32
 wind:
 required: false
 displayName: wind
 type: Wind
 clouds:
 required: false
 displayName: clouds
 type: Clouds
 rain:
 required: false
 displayName: rain
 type: Rain
 snow:
 required: false
 displayName: snow
 type: Snow
 dt:
 required: false
 displayName: dt
 description: Time of data calculation, unix, UTC
 type: integer
 format: int32
 sys:
 required: false
 displayName: sys
 type: Sys
 id:
 required: false
 displayName: id
 description: City ID
 type: integer
 format: int32
 name:
 required: false
 displayName: name
 type: string
 cod:
 required: false
 displayName: cod
 description: Internal parameter
 type: integer
 format: int32
 Coord:
 displayName: Coord
```

```
type: object
properties:
 lon:
 required: false
 displayName: lon
 description: City geo location, longitude
 type: number
 format: double
 lat:
 required: false
 displayName: lat
 description: City geo location, latitude
 type: number
 format: double
 Weather:
 displayName: Weather
 type: object
 properties:
 id:
 required: false
 displayName: id
 description: Weather condition id
 type: integer
 format: int32
 main:
 required: false
 displayName: main
 description: Group of weather parameters (Rain, Snow, Extreme etc.)
 type: string
 description:
 required: false
 displayName: description
 description: Weather condition within the group
 type: string
 icon:
 required: false
 displayName: icon
 description: Weather icon id
 type: string
 Main:
 displayName: Main
 type: object
 properties:
 temp:
 required: false
 displayName: temp
 description: 'Temperature. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.'
 type: number
 format: double
 pressure:
```

```
 required: false
 displayName: pressure
 description: Atmospheric pressure (on the sea level, if there is no
sea_level or grnd_level data), hPa
 type: integer
 format: int32
 humidity:
 required: false
 displayName: humidity
 description: Humidity, %
 type: integer
 format: int32
 temp_min:
 required: false
 displayName: temp_min
 description: 'Minimum temperature at the moment. This is deviation f
rom current temp that is possible for large cities and megalopolises geograph
ically expanded (use these parameter optionally). Unit Default: Kelvin, Met
ric: Celsius, Imperial: Fahrenheit.'
 type: number
 format: double
 temp_max:
 required: false
 displayName: temp_max
 description: 'Maximum temperature at the moment. This is deviation f
rom current temp that is possible for large cities and megalopolises geograph
ically expanded (use these parameter optionally). Unit Default: Kelvin, Met
ric: Celsius, Imperial: Fahrenheit.'
 type: number
 format: double
 sea_level:
 required: false
 displayName: sea_level
 description: Atmospheric pressure on the sea level, hPa
 type: number
 format: double
 grnd_level:
 required: false
 displayName: grnd_level
 description: Atmospheric pressure on the ground level, hPa
 type: number
 format: double
 Wind:
 displayName: Wind
 type: object
 properties:
 speed:
 required: false
 displayName: speed
 description: 'Wind speed. Unit Default: meter/sec, Metric: meter/se
c, Imperial: miles/hour.'
```

```
 type: number
 format: double
deg:
 required: false
 displayName: deg
 description: Wind direction, degrees (meteorological)
 type: integer
 format: int32
Clouds:
 displayName: Clouds
 type: object
 properties:
 all:
 required: false
 displayName: all
 description: Cloudiness, %
 type: integer
 format: int32
Rain:
 displayName: Rain
 type: object
 properties:
 3h:
 required: false
 displayName: 3h
 description: Rain volume for the last 3 hours
 type: integer
 format: int32
Snow:
 displayName: Snow
 type: object
 properties:
 3h:
 required: false
 displayName: 3h
 description: Snow volume for the last 3 hours
 type: number
 format: double
Sys:
 displayName: Sys
 type: object
 properties:
 type:
 required: false
 displayName: type
 description: Internal parameter
 type: integer
 format: int32
 id:
 required: false
 displayName: id
```

```
 description: Internal parameter
 type: integer
 format: int32
 message:
 required: false
 displayName: message
 description: Internal parameter
 type: number
 format: double
 country:
 required: false
 displayName: country
 description: Country code (GB, JP etc.)
 type: string
 sunrise:
 required: false
 displayName: sunrise
 description: Sunrise time, unix, UTC
 type: integer
 format: int32
 sunset:
 required: false
 displayName: sunset
 description: Sunset time, unix, UTC
 type: integer
 format: int32
/weather:
 get:
 displayName: Call current weather data for one location
 description: Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.
 securedBy:
 - auth
 queryParameters:
 q:
 required: false
 displayName: q
 description: '**City name**. *Example: London*. You can call by 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.
 type: string
 id:
 required: false
 displayName: id
 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 — just separate them by commas. The limit of locations is 50.
 type: string
```

```
ons is 20. *Note: A single ID counts as a one API call. So, if you have cit
y IDs. it's treated as 3 API calls.*"
 type: string
 lat:
 required: false
 displayName: lat
 description: '**Latitude**. *Example: 35*. The latitude cordinate o
f the location of your interest. Must use with `lon`.'
 type: string
 lon:
 required: false
 displayName: lon
 description: '**Longitude**. *Example: 139*. Longitude cordinate of
the location of your interest. Must use with `lat`.'
 type: string
 zip:
 required: false
 default: 94040,us
 example:
 value: 94040,us
 displayName: zip
 description: '**Zip code**. Search by zip code. *Example: 95050,u
s*. Please note if country is not specified then the search works for USA a
s a default.'
 type: string
 units:
 required: false
 default: standard
 example:
 value: imperial
 displayName: units
 description: '**Units**. *Example: imperial*. Possible values: `metr
ic`, `imperial`. When you do not use units parameter, format is `standard` b
y default.'
 type: string
 enum:
 - standard
 - metric
 - imperial
 lang:
 required: false
 default: en
 example:
 value: en
 displayName: lang
 description: '**Language**. *Example: en*. You can use lang paramete
r to get the output in your language. We support the following languages tha
t you can use with the corresponded lang values: Arabic - `ar`, Bulgarian -
`bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English -
`en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `g
l`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Kore
```

```
an - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `n
l`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedi
sh - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`,
Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh_cn`, Chinese T
raditional - `zh_tw`.
 type: string
 enum:
 - ar
 - bg
 - ca
 - cz
 - de
 - el
 - en
 - fa
 - fi
 - fr
 - gl
 - hr
 - hu
 - it
 - ja
 - kr
 - la
 - lt
 - mk
 - nl
 - pl
 - pt
 - ro
 - ru
 - se
 - sk
 - sl
 - es
 - tr
 - ua
 - vi
 - zh_cn
 - zh_tw
 Mode:
 required: false
 default: json
 example:
 value: json
 displayName: Mode
 description: '**Mode**. *Example: html*. Determines format of respon
se. Possible values are `xml` and `html`. If mode parameter is empty the for
mat is `json` by default.
 type: string
 enum:
```

```
- json
- xml
- html
responses:
 200:
 description: Successful response
 body:
 application/json:
 displayName: response
 description: Successful response
 type: SuccessfulResponse
 404:
 description: Not found response
 body: {}
```

## Outputs

You can generate outputs using the RAML spec from a variety of platforms. Here are three ways:

- [Developer Portal on Anypoint platform \(page 398\)](#). Choose this option if you are developing and delivering your API on Mulesoft's Anypoint platform.
- [API Console output \(page 400\)](#). Choose this option if you want a standalone API Console output delivered on your own server. (Note that this option also allows you to embed the console in an iframe.)
- [RAML2HTML project \(page 401\)](#). Choose this option if you want a simple HTML output of your API documentation. No interactive console is included.

## Deliver doc through Mulesoft's Anypoint Platform and Exchange Portal

Anypoint is the API development platform on Mulesoft. APIs you develop with Mulesoft can be shared and viewed on their [Anypoint Exchange portal](#).

The Anypoint console has a dashboard where you can work on your RAML definition, add other documentation pages (outside the spec), configure mocking services, and more. If your company already uses Mulesoft for other API development services, it would make sense to use their documentation and portal features as well.

The screenshot shows the Anypoint platform interface. On the left, there's a code editor window titled "openweathermap.raml" containing RAML 1.0 code. The code defines the OpenWeatherMap API with endpoints for current weather and forecasts. On the right, there's a "Mocking service" panel showing the API summary and documentation. The documentation page includes sections for Types, Resources, and /weather, with a prominent "GET /weather" button.

```

1 #@RAML 1.0
2 title: OpenWeatherMap API
3 version: 2.5
4 baseUri: http://api.openweathermap.org/data/2.5/
5 baseUriParameters: {}
6 documentation:
7 - title: OpenWeatherMap API
8 | content: 'Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **Note:** This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API.

 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.'
9 securitySchemes:
10 auth:
11 type: Pass Through
12 describedBy:
13 queryParameters:
14 appid:
15 required: true
16 displayName: appid
17 description: API key to authorize requests. If you don't have an OpenWeatherMap API key, use `fd4698c940c6d1da02a70ac34f0b147`.
18 type: string
19 types:

```

The UI when working with RAML definitions in the Anypoint platform

Note that you can add additional pages to your documentation in Anypoint. (Kudos to the Mulesoft team for recognizing that API documentation is more than just a set of reference endpoints.)

Here's what the sample OpenWeatherMap API looks like in Anypoint Exchange:

The screenshot shows the Anypoint Exchange interface. On the left, there's a sidebar with "Assets list" and "OpenWeatherMap" selected. The main area shows the "OpenWeatherMap" API listing, which includes a logo, a 5-star rating (0 reviews), and the version 2.5. Public. The listing page provides a brief description of the API and links to its documentation, types, resources, and /weather endpoint. The "/weather" endpoint is detailed with a "Request" section showing the GET method and URL, and a "Parameters" section showing a table for "Query parameters".

Parameter	Type	Description
q	string	City name. Exam...

One of the unique options Mulesoft is called [API Notebook](#). This is a unique tool designed by Mulesoft that allows you to provide interactive code examples that leverage your RAML spec.

## Deliver doc through the API Console Project

You can download the same code that generates the output on the Anypoint Platform and create your own API Console. The standalone project (on GitHub) is called [API Console](#). Here's a [demo](#). Instructions are available on the [api-console](#) project on GitHub. Basically, do the following:

1. Install the api-console CLI:

```
sudo npm install -g api-console-cli
```

2. Create a directory where you want to build the tool.
3. Switch to sudo on your computer (`sudo su`).
4. Build your RAML file (or convert your existing OpenAPI spec using [API Transformer](#)).
5. Put your RAML file onto a web server (for example, <http://idratherassets.com/raml/weather.raml>).
6. Run the api-console build tool:

```
api-console build http://idratherassets.com/raml/weather.raml
```

It takes a few minutes for the build to complete. (If you run into errors, make sure you're in sudo mode.)

7. Upload the output to your web server, and then go to the index.html file in your browser.

The screenshot shows the API console interface for the OpenWeatherMap API. The left sidebar has a tree view with nodes for API summary, Documentation (expanded), Types (under Documentation), Resources (under Documentation), and a /weather node under Resources. The /weather node is expanded, showing a 'GET Call current weather' button and a 'Parameters' table. The table has columns for Parameter, Type, and Description. It lists five parameters: 'q' (string, City name), 'id' (string, City ID), 'lat' (string, Latitude), 'lon' (string, Longitude), and 'zip' (string, Zip code). The 'units' parameter is also listed at the bottom, with a note that it's standard by default. A 'TRY IT' button is visible above the parameters table. The right side of the interface shows the 'OpenWeatherMap API' title and a detailed description of the /weather endpoint: '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.' Below this is a 'Request' section with a 'GET' link to 'http://api.openweathermap.org/data/2.5/weather'.

Parameter	Type	Description
q	string	<b>City name.</b> Example: London. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by comma; use ISO 3166 country codes.
id	string	<b>City ID.</b> Example: 2172797. You can call by city ID. API responds with exact result. The List of city IDs can be downloaded <a href="#">here</a> . You can include multiple cities in parameter – just separate them by commas. The limit of locations is 20. Note: A single ID counts as a one API call. So, if you have city IDs, it's treated as 3 API calls.
lat	string	<b>Latitude.</b> Example: 35. The latitude coordinate of the location of your interest. Must use with lon.
lon	string	<b>Longitude.</b> Example: 139. Longitude coordinate of the location of your interest. Must use with lat.
zip	string	<b>Zip code.</b> Search by zip code. Example: 95050,us. Please note if country is not specified then the search works for USA as a default.
		Default value: 94040,us Example value: 94040,us
units	string (enum)	<b>Units.</b> Example: imperial. Possible values: metric, imperial. When you do not use units parameter, format is standard by default.

Powered by MuleSoft

This is a sample RAML output in something called API Console

You can also [embed the API console as an HTML element](#).

## Deliver doc through the RAML2HTML Utility

Finally, you can also use a tool called [RAML2HTML project](#) to generate HTML documentation from a RAML spec. Here's [an example](#) of what the RAML2HTML output looks like. It's a static HTML output without any interactivity. See the [RAML2HTML documentation](#) for instructions on generating this output.

## Conclusions

Exploring Mulesoft in more depth is beyond the scope of this tutorial, but hopefully I've introduced you to RAML and Mulesoft. Overall, large platforms that process and display your API documentation can only do so if your documentation aligns with a spec their tools can parse. RAML provides this standard spec for the Mulesoft ecosystem of tools. Their enterprise-level API tools provide powerful capabilities for your API.

# API Blueprint tutorial

Just as Swagger defines a spec for describing a REST API, [API Blueprint](#) is another specification for describing REST APIs. If you describe your API with this blueprint, tools that support API Blueprint can read and display the information.

Unless you're using a platform that specifically requires API Blueprint, I recommend using the [OpenAPI specification \(page 310\)](#) instead.

## What is API Blueprint

The API Blueprint spec is written in a Markdown-flavored syntax. It's not normal Markdown, but it has a lot of the same, familiar Markdown syntax. However, the blueprint is clearly a very specific schema that is either valid or not valid based on the element names, order, spacing, and other details. In this way, it's not nearly as flexible or forgiving as pure Markdown. But it may be preferable to YAML.

## Sample blueprint

Here's a sample blueprint to give you an idea of the syntax:

```
FORMAT: 1A
HOST: http://polls.apiblueprint.org/

test

Polls is a simple API allowing consumers to view polls and vote in them.

Polls API Root [/]

This resource does not have any attributes. Instead it offers the initial
API affordances in the form of the links in the JSON body.

It is recommend to follow the "url" link values,
[Link](https://tools.ietf.org/html/rfc5988) or Location headers where
applicable to retrieve resources. Instead of constructing your own URLs,
to keep your client decoupled from implementation details.

Retrieve the Entry Point [GET]

+ Response 200 (application/json)

{
 "questions_url": "/questions"
}

Group Question

Resources related to questions in the API.

Question [/questions/{question_id}]

A Question object has the following attributes:

+ question
+ published_at - An ISO8601 date when the question was published.
+ url
+ choices - An array of Choice objects.

+ Parameters
 + question_id: 1 (required, number) - ID of the Question in form of an i
nteger

View a Questions Detail [GET]

+ Response 200 (application/json)

{
 "question": "Favourite programming language?",
 "published_at": "2014-11-11T08:40:51.620Z",
 "url": "/questions/1",
 "choices": [
```

```
{
 "choice": "Swift",
 "url": "/questions/1/choices/1",
 "votes": 2048
}, {
 "choice": "Python",
 "url": "/questions/1/choices/2",
 "votes": 1024
}, {
 "choice": "Objective-C",
 "url": "/questions/1/choices/3",
 "votes": 512
}, {
 "choice": "Ruby",
 "url": "/questions/1/choices/4",
 "votes": 256
}
]
}

Choice [/questions/{question_id}/choices/{choice_id}]

+ Parameters
 + question_id: 1 (required, number) - ID of the Question in form of an integer
 + choice_id: 1 (required, number) - ID of the Choice in form of an integer

Vote on a Choice [POST]

This action allows you to vote on a question's choice.

+ Response 201

 + Headers

 Location: /questions/1

Questions Collection [/questions{?page}]

+ Parameters
 + page: 1 (optional, number) - The page of questions to return

List All Questions [GET]

+ Response 200 (application/json)

 + Headers

 Link: </questions?page=2>; rel="next"
```

**+ Body**

```
[
 {
 "question": "Favourite programming language?",
 "published_at": "2014-11-11T08:40:51.620Z",
 "url": "/questions/1",
 "choices": [
 {
 "choice": "Swift",
 "url": "/questions/1/choices/1",
 "votes": 2048
 }, {
 "choice": "Python",
 "url": "/questions/1/choices/2",
 "votes": 1024
 }, {
 "choice": "Objective-C",
 "url": "/questions/1/choices/3",
 "votes": 512
 }, {
 "choice": "Ruby",
 "url": "/questions/1/choices/4",
 "votes": 256
 }
]
 }
]
```

**### Create a New Question [POST]**

You may create your own question using this action. It takes a JSON object containing a question and a collection of answers in the form of choices.

+ question (string) – The question  
+ choices (array[string]) – A collection of choices.

+ Request (application/json)

```
{
 "question": "Favourite programming language?",
 "choices": [
 "Swift",
 "Python",
 "Objective-C",
 "Ruby"
]
}
```

+ Response 201 (application/json)

#### + Headers

Location: /questions/2

#### + Body

```
{
 "question": "Favourite programming language?",
 "published_at": "2014-11-11T08:40:51.620Z",
 "url": "/questions/2",
 "choices": [
 {
 "choice": "Swift",
 "url": "/questions/2/choices/1",
 "votes": 0
 }, {
 "choice": "Python",
 "url": "/questions/2/choices/2",
 "votes": 0
 }, {
 "choice": "Objective-C",
 "url": "/questions/2/choices/3",
 "votes": 0
 }, {
 "choice": "Ruby",
 "url": "/questions/2/choices/4",
 "votes": 0
 }
]
}
```

For a tutorial on the blueprint syntax, see this [Apiary tutorial](#) or [this tutorial on Github](#).

You can find [examples of different blueprints here](#). The examples can often clarify different aspects of the spec.

## Parsing the blueprint

There are many tools that can parse an API blueprint. [Drafter](#) is one of the main parsers of the Blueprint. Many other tools build on Drafter and generate static HTML outputs of the blueprint. For example, [aglio](#) can parse a blueprint and generate static HTML files.

For a more comprehensive list of tools, see the [Tooling](#) section on apiblueprint.org. (Some of these tools require quite a few prerequisites, so I omitted the tutorial steps here for generating the output on your own machine.)

## Create a sample HTML output using API Blueprint and Apiary

For this tutorial, we'll use a platform called Apiary to read and display the API Blueprint. Apiary is just a hosted platform that will remove the need for installing local libraries and utilities to generate the output.

## a. Create a new Apiary project

1. Go to [apiary.io](#) and click **Quick start with Github**. Sign in with your Github account. (If you don't have a Github account, create one first.)
2. Sign up for a free hacker account and create a new project.

You'll be placed in the API Blueprint editor.

The screenshot shows the Apiary Blueprint editor interface. At the top, there's a header with the project name "test" and a dropdown for "Yours \* test5395". To the right are links for "Documentation", "Inspector", and "Editor", along with user icons and a "Try 'Apiary for Teams'" button. Below the header, the main area has tabs for "FORMAT" (selected), "A" (API Blueprint Syntax Tutorial), and "Reference". A status bar indicates "Valid API Blueprint" and "Preview On". The left panel displays the API Blueprint code for the Polls API, which includes sections for the root resource and a "questions" resource. The right panel shows the generated API documentation, starting with the "Polls" section under "INTRODUCTION". The "REFERENCE" section is also visible.

By default the Polls blueprint is loaded so you can see how it looks. This blueprint gives you an example of the required format for the Apiary tool to parse and display the content. You can also see the [raw file here](#).

3. At this point, you would start describing your API using the blueprint syntax in the editor. When you make a mistake, error flags indicate what's wrong.

You can [read the Apiary tutorial](#) and structure your documentation in the blueprint format. The syntax seems to accommodate different methods applied to the same resources.

For this tutorial, you'll integrate the OpenWeatherMap weather API information info formatted in the blueprint format.

4. Copy the following code, which aligns with the API Blueprint spec, and paste it into the Apiary blueprint editor.

FORMAT: 1A

HOST: <http://api.openweathermap.org/data/2.5/>

# OpenWeatherMap API

Get current weather, daily forecast for 16 days, and 3-hourly forecast 5 days for your city. Helpful stats, graphics, and this day in history charts are available for your reference. Interactive maps show precipitation, clouds, pressure, wind around your location stations. Data is available in JSON, XML, or HTML format. **\*\*Note\*\*:** This sample Swagger file covers the `current` endpoint only from the OpenWeatherMap API. <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.

## Authentication

This API uses Custom Query Parameter for its authentication.

The parameters that are needed to be sent for this type of authentication are as follows:

+ `appid` – API key to authorize requests. If you don't have an OpenWeatherMap API key, use `fd4698c940c6d1da602a70ac34f0b147`.

# Group Current Weather Data

## Weather [/weather{?q,id,lat,lon,zip,units,lang,Mode}]

### Call current weather data for one location [GET]

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations.

+ Parameters

+ q (string, optional)

**\*\*City name\*\*.** \*Example: London\*. You can call by city name, or by city name and country code. The API responds with a list of results that match a searching word. For the query value, type the city name and optionally the country code divided by comma; use ISO 3166 country codes.

+ id (string, optional)

**\*\*City ID\*\*.** \*Example: `2172797`\*. You can call by city ID. A PI 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 — just separate them by commas. The limit of locations is 20. \*Note: A single ID counts as a one API call. So, if you have city IDs. it's treated as 3 API calls.\*

+ lat (string, optional)

**\*\*Latitude\*\*.** \*Example: 35\*. The latitude coordinate of the lo

cation of your interest. Must use with `lon`.

+ lon (string, optional)

\*\*Longitude\*\*. \*Example: 139\*. Longitude coordinate of the location of your interest. Must use with `lat`.

+ zip (string, optional) -

\*\*Zip code\*\*. Search by zip code. \*Example: 95050,us\*. Please note if country is not specified then the search works for USA as a default.

+ Default: 94040,us

+ Sample: 94040,us

+ units (enum[string], optional)

\*\*Units\*\*. \*Example: imperial\*. Possible values: `metric`, `imperial`. When you do not use units parameter, format is `standard` by default.

+ Default: standard

+ Sample: imperial

+ Members

+ `standard`

+ `metric`

+ `imperial`

+ lang (enum[string], optional)

\*\*Language\*\*. \*Example: en\*. You can use lang parameter to get the output in your language. We support the following languages that you can use with the corresponded lang values: Arabic - `ar`, Bulgarian - `bg`, Catalan - `ca`, Czech - `cz`, German - `de`, Greek - `el`, English - `en`, Persian (Farsi) - `fa`, Finnish - `fi`, French - `fr`, Galician - `gl`, Croatian - `hr`, Hungarian - `hu`, Italian - `it`, Japanese - `ja`, Korean - `kr`, Latvian - `la`, Lithuanian - `lt`, Macedonian - `mk`, Dutch - `nl`, Polish - `pl`, Portuguese - `pt`, Romanian - `ro`, Russian - `ru`, Swedish - `se`, Slovak - `sk`, Slovenian - `sl`, Spanish - `es`, Turkish - `tr`, Ukrainian - `ua`, Vietnamese - `vi`, Chinese Simplified - `zh\_cn`, Chinese Traditional - `zh\_tw`.

+ Default: en

+ Sample: en

+ Members

+ `ar`

+ `bg`

+ `ca`

+ `cz`

+ `de`

```
+ `el`
+ `en`
+ `fa`
+ `fi`
+ `fr`
+ `gl`
+ `hr`
+ `hu`
+ `it`
+ `ja`
+ `kr`
+ `la`
+ `lt`
+ `mk`
+ `nl`
+ `pl`
+ `pt`
+ `ro`
+ `ru`
+ `se`
+ `sk`
+ `sl`
+ `es`
+ `tr`
+ `ua`
+ `vi`
+ `zh_cn`
+ `zh_tw`

+ Mode (enum[string], optional)

 Mode. *Example: html*. Determines format of response. Possible values are `xml` and `html`. If mode parameter is empty the format is `json` by default.

 + Default: json
 + Sample: json
 + Members
 + `json`
 + `xml`
 + `html`

+ Response 200 (application/json)

 Successful response

 + Attributes (Successful response)

+ Response 404
```

### Not found response

```
Data Structures

Successful response (object)

Properties
+ `coord` (Coord, optional)
+ `weather` (array[Weather], optional) – (more info Weather condition codes)
+ `base`: `cmc stations` (string, optional) – Internal parameter
+ `main`: `cmc stations` (Main, optional)
+ `visibility`: `16093` (number, optional) – Visibility, meter
+ `wind`: `16093` (Wind, optional)
+ `clouds`: `16093` (Clouds, optional)
+ `rain`: `16093` (Rain, optional)
+ `snow`: `16093` (Snow, optional)
+ `dt`: `1435658272` (number, optional) – Time of data calculation, unix, UTC
+ `sys`: `1435658272` (Sys, optional)
+ `id`: `2172797` (number, optional) – City ID
+ `name`: `Cairns` (string, optional)
+ `cod`: `200` (number, optional) – Internal parameter

Coord (object)

Properties
+ `lon`: `145.77` (number, optional) – City geo location, longitude
+ `lat`: `-16.92` (number, optional) – City geo location, latitude

Weather (object)

Properties
+ `id`: `803` (number, optional) – Weather condition id
+ `main`: `Clouds` (string, optional) – Group of weather parameters (Rain, Snow, Extreme etc.)
+ `description`: `broken clouds` (string, optional) – Weather condition within the group
+ `icon`: `04n` (string, optional) – Weather icon id

Main (object)

Properties
```

```
+ `temp`: `293.25` (number, optional) – Temperature. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
+ `pressure`: `1019` (number, optional) – Atmospheric pressure (on the sea level, if there is no sea_level or grnd_level data), hPa
+ `humidity`: `83` (number, optional) – Humidity, %
+ `temp_min`: `289.82` (number, optional) – Minimum temperature at the moment. This is 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.
+ `temp_max`: `295.37` (number, optional) – 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.
+ `sea_level`: `984` (number, optional) – Atmospheric pressure on the sea level, hPa
+ `grnd_level`: `990` (number, optional) – Atmospheric pressure on the ground level, hPa

Wind (object)

Properties
+ `speed`: `5.1` (number, optional) – Wind speed. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour.
+ `deg`: `150` (number, optional) – Wind direction, degrees (meteorological)

Clouds (object)

Properties
+ `all`: `75` (number, optional) – Cloudiness, %

Rain (object)

Properties
+ `3h`: `3` (number, optional) – Rain volume for the last 3 hours

Snow (object)

Properties
+ `3h`: `6` (number, optional) – Snow volume for the last 3 hours
```

```
Sys (object)

Properties
+ `type`: `1` (number, optional) – Internal parameter
+ `id`: `8166` (number, optional) – Internal parameter
+ `message`: `0.0166` (number, optional) – Internal parameter
+ `country`: `AU` (string, optional) – Country code (GB, JP etc.)
+ `sunrise`: `1435610796` (number, optional) – Sunrise time, unix, UTC
+ `sunset`: `1435650870` (number, optional) – Sunset time, unix, UTC
```

If the code isn't easy to copy and paste, you can [view and download the file here](#).

5. Click **Save and Publish**.

### b. Interact with the API on Apiary

In the Apiary's top navigation, click **Documentation**. Then interact with the API on Apiary by clicking **Switch to Console**. Call the resources and view the responses.

You can switch between an Example and a Console view in the documentation. The Example view shows pre-built responses. The Console view allows you to enter your own values and generate dynamic responses based on your own API key. This dual display — both the Example and the Console views — might align better with user needs:

- For users who might not have good data or might not want to make requests that would affect their data, they can view the Example.
- For users who want to see how the API specifically returns their data, or certain parameters, they can use the Console view.

# Documenting native library APIs

Overview of native library APIs .....	415
Get the Java source .....	418
Java crash course .....	423
Activity: Generate a Javadoc from a sample project.....	429
Javadoc tags .....	434
Explore the Javadoc output .....	442
Make edits to Javadoc tags .....	445
Doxygen, a document generator mainly for C++ .....	446
Create non-ref docs with native library APIs.....	448

# Overview of native library APIs

In previous parts of the course, we focused exclusively on REST APIs. Now let's explore native library APIs, which are more common when building native apps.

## Characteristics of native library APIs

Native library APIs (also called class-based APIs or just APIs) are notably different in the following ways:

- **Installed locally.** Native library APIs are installed locally, compiled into the programmer's code as an additional library. The programmer can then use the classes, methods, or other functions available in the library. (The API part refers to the *public* classes the users use to access the functions in the library. There are probably lots of helper and utility classes in the Java library that aren't public. The *public* functions that the developer audience uses form the API, since this is how people make use of the library.)
- **No requests and responses.** The classes in the native library API don't use HTTP protocol, nor are there request and responses sent across the web. The native library API is simply a collection of functions that enhance the existing code with more capabilities. It's entirely on-premises.
- **Language specific.** Native library APIs are language specific. There are as many different types of APIs as there are programming languages, more or less. You can have a Java API, C++ API, C# or .NET API, JavaScript API, and so on.
- **Requires some programming knowledge to document.** To understand how the API works, you need to have a general understanding of the programming language the API is written for. You don't need to be a programmer, but you should be familiar with the nuts and bolts of the programming language, the correct terms, how the different parts fit together, and how developers will use the API.

We will focus this section on Java APIs, since they're probably one of the most common. However, many of the concepts and code conventions mentioned here will apply to the other languages, with minor differences.

```

1 package acme;
2
3 import java.io.IOException;
4
5 /**
6 * Works like a regular smartphone but also tracks roadrunners.
7 * <p>
8 * The ACME Smartphone can perform similar functions as other smartphones, such
9 * as making phone calls, sending text messages, and browsing the web. However,
10 * the ACME Smartphone also enables GPS tracking on roadrunners. You can monitor
11 * the location of all roadrunners within a 20 mile radius using the RoadRunner
12 * Tracker app.
13 * <p>
14 * Note that the RoadRunner Tracker app requires you to be connected to wifi. It
15 * will not work on cellular data.
16 *
17 * @author Tom Johnson
18 * @version 2.0
19 * @since 1.3
20 * @see Dynamite
21 */
22 public class ACMESmartphone {
23
24 /**
25 * The coordinates where the nearest roadrunner is located.
26 */
27
28 public String LongLat = "Longitude = 39.2334, Latitude = 41.4899"; // hard-coded for s
29
30
31 double model;
32 String license;

```

## Do you have to be a programmer to document native library APIs?

Because native library APIs are so dependent on a specific programming language, the documentation is usually written or driven by engineers rather than generalist technical writers. This is one area where it helps to be a former software engineer when doing documentation.

Even so, you don't need to be a programmer. You just need a minimal understanding of the language. Technical writers can contribute a lot here in terms of style, consistency, clarity, tagging, and overall professionalism.

You know what happens when engineers write — the content is cryptic and often incomplete. Usually the developer assumes everyone is as knowledgeable as he or she is, and any kind of extra explanatory detail, examples, cross-references, glossaries, or other helpful information is omitted.

## My approach to teaching native library API doc

There are many books and online resources you can consult to learn a specific programming language. This section of the course will not try to [teach you Java \(page 423\)](#). However, to understand a bit about Java API documentation (which uses a document generator called [Javadoc](#)), you will need some understanding of Java.

To keep the focus on API documentation, we'll take a documentation-centric approach to understanding Java. You'll learn the various parts of Java by looking at a specific Javadoc file and sorting through the main components.

## What you need to install

For this part of the course, you need to install the following:

- **Java Development Kit (JDK).** You can [download the JDK here](#). Click the Java button on the left

(not Netbeans) and then select the appropriate download for your machine.

- **Eclipse IDE for Java Developers.** Use the [Eclipse Installer to download Eclipse](#).

The screenshot shows the Eclipse Installer website interface. At the top, there's a search bar labeled "type filter text" and a magnifying glass icon. Below the search bar, there are four main sections, each with an icon and a title. The first section is "Eclipse IDE for Java Developers" with an icon of a gear and a "T" (representing Java). It describes the essential tools for Java developers, including a Java IDE, Git client, XML Editor, Mylyn, Maven, and Gradle integration. The second section is "Eclipse IDE for Java EE Developers" with an icon of a gear and a "J" (representing Java EE). It describes tools for Java developers creating Java EE and Web applications, including a Java IDE, tools for Java EE, JPA, JSF, Mylyn, EGit, and others. The third section is "Eclipse IDE for C/C++ Developers" with an icon of a gear and a "C++" symbol. It describes an IDE for C/C++ developers with Mylyn integration. The fourth section is "Eclipse IDE for JavaScript and Web Developers" with an icon of a globe and a "JS" symbol. It describes the essential tools for JavaScript developers, including JavaScript, HTML, CSS, XML languages support, Git client, and Mylyn.

To make sure you have Java installed, you can do the following:

- On Mac: Open Terminal and type `java -version`.
- On Windows: Open a Command Prompt and type `where java`.

Also, start Eclipse and make sure it doesn't complain that you don't have the JDK.

(Since we'll just be using Java within the context of Eclipse, Windows users don't need to add Java to their class path. But if you want to be able to compile Java from the command line, you can do this.)

# Get the Java source

In order to understand documentation for Java APIs, it helps to have a context of some sort. As such, I created a simple little Java application (called [sample-java-project](#)) to demonstrate how the various tags get rendered into the Javadoc.

## Sample Java Project

The [sample Java project](#) is a little application about different tools that a coyote will use to capture a roadrunner. There are two classes (ACMESmartphone and Dynamite) and another class file called App that references the classes.

This program doesn't really do anything except print little messages to the console, but it's hopefully simple enough to be instructive in its purpose. The purpose of the app is to demonstrate different doc tags, their placement, and how they get rendered in the Javadoc.

## Download the project

One of your immediate challenges to editing Javadoc will be to get the source code into your IDE. The sample java project is [here on Github](#).

First clone the source using version control. We covered some version control basics [earlier in the course \(page 224\)](#).

You can clone the source in a couple of ways:

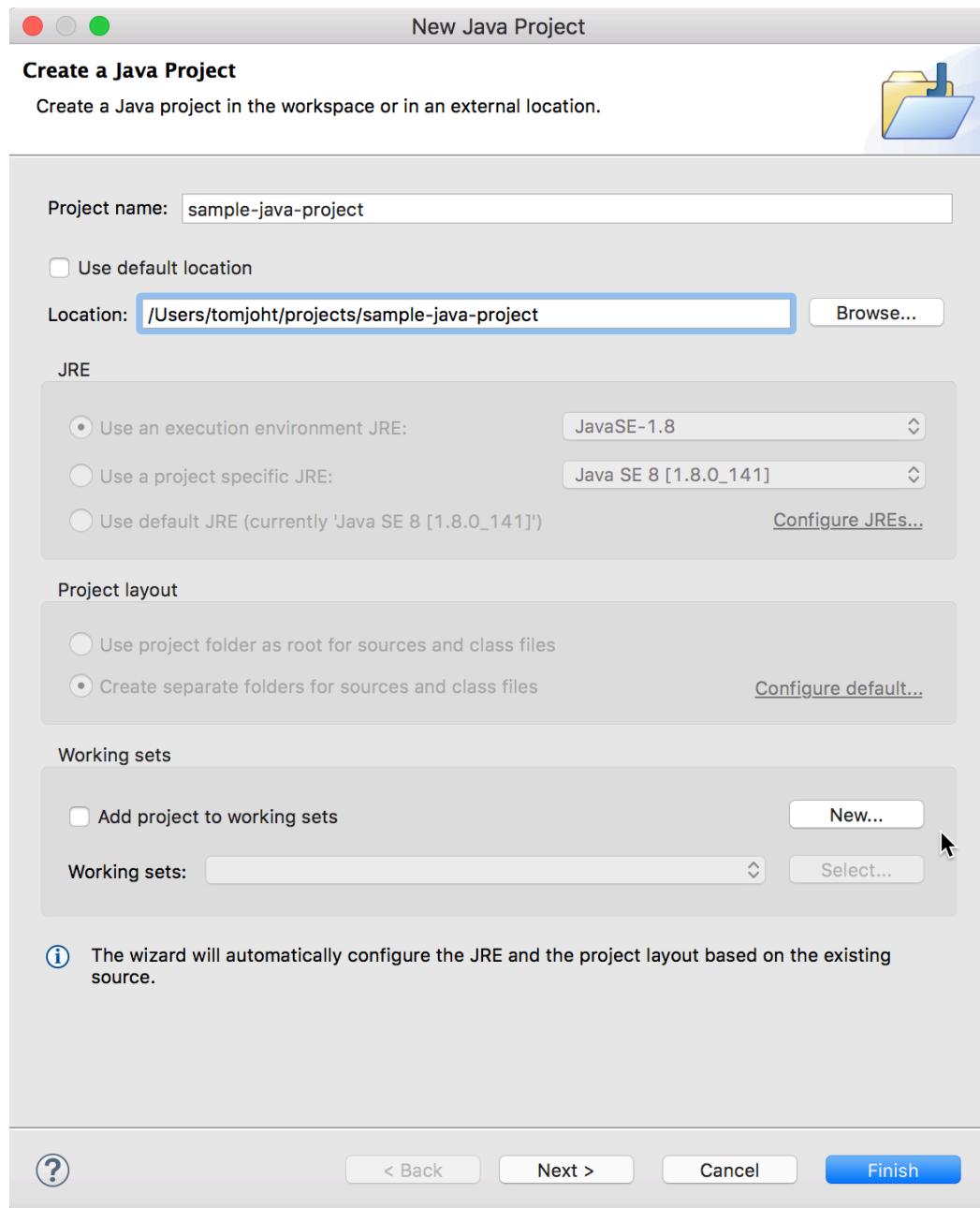
```
git clone https://github.com/tomjoht/sample-java-project
```

Or click **Clone in Desktop** and navigate to the right path in Github Desktop.

(If you don't want to clone the source, you could click **Download ZIP** and download the content manually.)

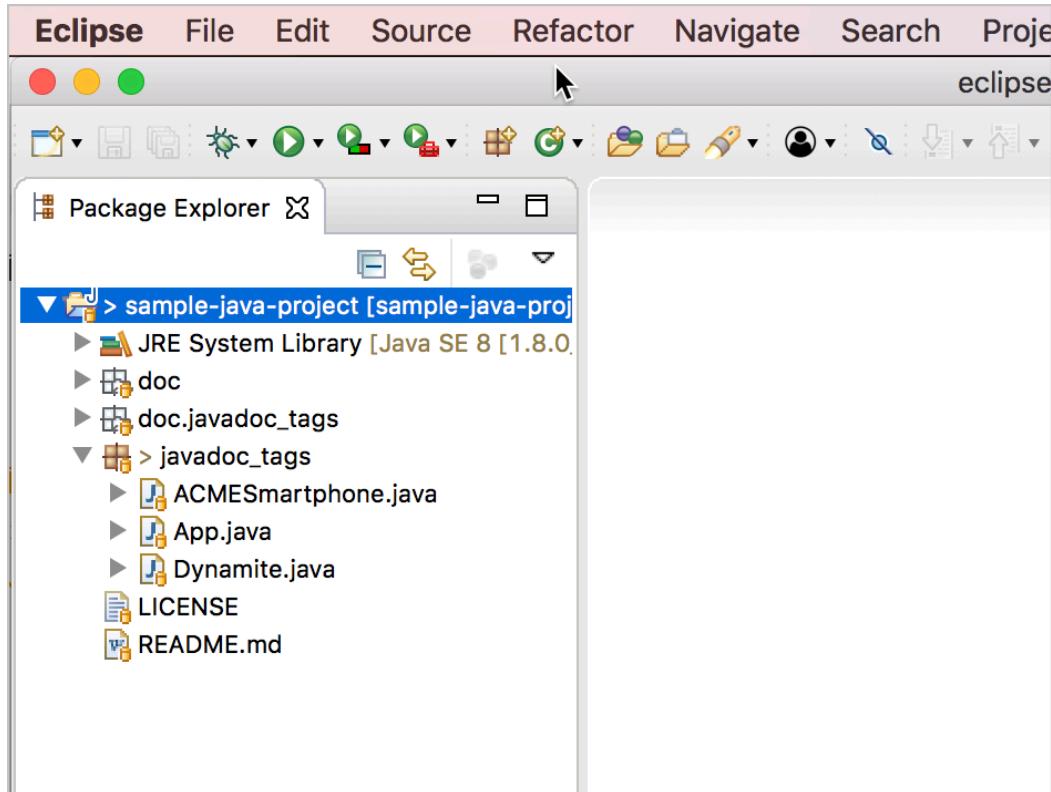
## Open the project in Eclipse

1. If you haven't already [downloaded and installed Eclipse \(page 416\)](#), do so.
2. After you've cloned or downloaded the Java project, open Eclipse. (If prompted to select a workspace, select the default location and click **Launch**. Also, close the Welcome screen if it appears.)
3. In Eclipse, go to **File > New > Java Project**.
4. Clear the **Use default location** check box, and then click **Browse** and go to the folder where you cloned the Github project.



5. Click **Finish**.

The Java files should be visible in the left pane (Package Explorer) within your Eclipse IDE.



## Side Note about Maven projects

Java projects often have a lot of dependencies on packages that are third-party libraries or at least non-standard Java utilities. Rather than requiring users to download these additional packages and add them to their class manually, developers frequently use Maven to manage the packages.

Maven projects use a pom.xml file that defines the dependencies. Eclipse ships with Maven already installed, so when you import a Maven project and install it, the Eclipse Maven plugin will retrieve all of the project dependencies and add them to your project.

The sample project doesn't use Maven, but I want to add a note about Maven here anyway. Chances are if you're getting a Java project from developers, you won't import it in the way I previously described. Instead, you'll import it as an existing Maven project.

(By the way, to import a Maven project into Eclipse, you would go to **File > Import > Maven > Existing Maven Projects** and click **Next**. In the Root Directory field, you would click **Browse** and browse to the Java project folder (which contains the Maven pom.xml file) and then click **Open**. Then you would click **Finish** in the dialog box. In the Project Explorer pane in Eclipse, you would right-click the Java folder and select **Run as Maven Install**. Maven retrieves the necessary packages and builds the project. If the build is successful, you will see a "BUILD SUCCESS" message in the console. You would then use the source code in the built project.)

## Play with the Sample Java Project

### ACTIVITY



This Java app doesn't do much. It's main purpose is to create some classes where I can add some Javadoc annotations. But for fun, you can run the app.

In the Package Explorer, expand **javadoc\_tags** and select the **App.java**. Then click the **Run App** button



The main method runs these functions:

```
public static void main(String[] args) throws IOException {

 // First initialize your smartphone using the model number and license key.
 ACMESmartphone myACMESmartphone = new ACMESmartphone(2.0, "398978fdskj");

 // Locate the roadrunner.
 myACMESmartphone.findRoadRunner("Santa Clara", "California");

 // Zap the roadrunner with the amount of voltage you want.
 myACMESmartphone.zapRoadRunner(40);

}
```

You can view the details of each function in the ACMESmartphone.java and Dynamite.java classes.

Then app prints this text to the console:

```
model2.0 now initialized for license 398978fdskj
location: Santa Clara, California
getting geocoordinates of roadrunner....
roadrunner located at Longitude = 39.2334, Latitude = 41.4899
Backfire!!! zapping coyote with 1,000,000 volts!!!!
```

The screenshot shows the Eclipse IDE interface with the following details:

- Menu Bar:** Eclipse, File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help.
- Title Bar:** eclipse-workspace - sample-java-project/javadoc\_tags/App.java - Eclipse
- Toolbar:** Standard Eclipse icons for file operations, search, and run.
- Package Explorer View:** Shows the project structure:
  - sample-java-project [sample-java-pr...]
    - JRE System Library [Java SE 8 [1.8.0\_181]]
    - doc
    - doc.javadoc\_tags
    - javadoc\_tags
      - ACMESmartphone.java
      - ACMESmartphone
    - App.java
      - App
      - main(String[]): void
    - Dynamite.java
    - LICENSE
    - README.md
- Editor View:** Displays the code for `ACMESmartphone.java`. The code includes imports, class definitions, and several multi-line comments. The line numbers 1 through 21 are visible on the left.
- Console View:** Shows the output of a Java application named `App`. The output text is:

```
<terminated> App [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_201/lib/jre/lib/server/libjvm.dylib
model2.0 now initialized for license 398978fdskj
location: Santa Clara, California
getting geocoordinates of roadrunner....
roadrunner located at Longitude = 39.2334, Latitude = 41.4899
Backfire!!! zapping coyote with 1,000,000 volts!!!!
```

# Java crash course

To understand the different components of a Javadoc, you have to first understand a bit about Java. Just being familiar with the names of the different components of Java will allow you to enter conversations and understand code at a high level. When you describe different aspects of sample code, knowing when to call something a class, method, parameter, or enum can be critical to your documentation's credibility.

I'll run you through a brief crash course in the basics. For more detail about learning Java, I recommend consulting [lynda.com](#) and [safaribooksonline](#). Below I'll focus on some basic concepts in Java that will be important in understanding the Javadoc tags and elements.

## About Java

Java is one of the most common languages used because of its flexibility. Java isn't tied to a specific language platform because Java code compiles into byte code. The platform you deploy your code on contains a Java Virtual Machine (JVM) that interprets the byte code. Hence through JVMs, different platforms can interpret and run Java code. This gives Java more flexibility with different platforms.

## Classes

Classes are templates or blueprints that drive pretty much everything in Java. It's easiest to understand classes through an example. Think of a class like a general blueprint of a "bicycle." There are many different types of bicycles (Trek bikes, Specialized bikes, Giant bikes, Raleigh bikes, etc.). But they're all just different instances of the general class of a bicycle.

In Java, you start out by defining classes. Each class is its own file, and begins with a capital letter. The file name matches the class name, which means you have just one class per file.

Each class can contain a number of fields (variables for the class) and methods (subroutines the class can do).

Before the class name, an access modifier indicates how the class can be accessed. Several options for access modifiers are:

- `public` : Anyone can access
- `private` : Only other packages can access
- `static` : No one can change the class
- `abstract` : The class can't be instantiated, only sub-classed.

Here's an example of a class:

```
public class Bicycle{
 //code...
}
```

You mostly need to focus on `public` classes, since these are the classes that will be used by your audience. The `public` classes are the API of the library.

## Methods

Methods are subroutines or actions that the class can do. For example, with a bicycle you can pedal, brake, and turn. A class can have as many methods as it needs.

Methods can take arguments, so there are parentheses `()` after the method name. The arguments are variables that are used within the code for that method. For example:

```
add(a, b) {
 sum = a + b;
}
```

Methods can return values. When a method finishes, the value can be returned to the caller of the method.

Before the method name, the method indicates what type of data it returns. If the method doesn't return anything, `void` is listed. Other options are `String` or `int`.

Here's an example of some methods for our Bicycle class:

```
class Bicycle {

 void turn() {
 // code ...
 }
 void pedal(int rotations) {
 System.out.println("Your speed is " + rotations + " per minute");
 }

 int brake(int force, int weight) {
 torque == force * weight;
 return torque;
 }
}
```

See how the `brake` method accepts two arguments — `force` and `weight`? These arguments are integers, so Java expects whole numbers here. (You must put the data type before the parameters in the method.) The arguments passed into this method get used to calculate the `torque`. The `torque` is then returned to the caller.

In Javadoc outputs, you'll see methods divided into two groups:

- **Instance methods:** Means that you can use the method in the instance of an object. If the method isn't an instance method, it's considered a static method. Static methods can be used directly from the class without instantiating an object first. Static methods can't be changed by any object or subclass.
- **Concrete methods:** These are methods that can be used when you instantiate an object. If a method isn't concrete, it's called an “abstract method.” The only way you use an abstract method is by creating a subclass for the method.

Somewhere in your Java application, users will have something called a `main` method that looks like this:

```
public static void main(String[] args) {
}
```

Inside the main method is where you add your code to make your program run. This is where the Java Virtual Machine will look to execute the code.

## Fields

Fields are variables available within the class. A variable is a placeholder that is populated with a different value depending on what the user wants.

Fields indicate their data types, because all data in Java is “statically typed” (meaning, its format/length is defined) so that the data doesn’t take up more space than it needs. Some data types include `byte`, `short`, `long`, `int`, `float`, or `double`. Basically these are numbers or decimals of different sizes. You can also specify a `char`, `string`, or `boolean`.

Here’s an example of some fields in class:

```
class Bicycle {
 String brand;
 int size;
}
```

Many times fields are “encapsulated” with getter-setter methods, which means their values are set in a protected way. Users call one method to set the field’s value, and another method to get the field’s value. This way you can avoid having users set improper values or incorrect data types for the fields.

Fields that are constant throughout the Java project are called ENUMS. Alternatively, the fields are given `public static final` modifiers.

## Objects

Objects are instances of classes. They are the Treks, Raleighs, Specialized, etc., of the Bicycle class.

If I wanted to use the `Bicycle` class, I would create an instance of the class. The instance of the class is called an object. Here’s what it looks like when you “instantiate” the class:

```
Bicycle myBicycle = new Bicycle();
```

You write the class name followed by the object name for the class. Then assign the object to be a new instance of the class. Now you’ve got `myBicycle` to work with.

The object inherits all of the fields and methods available to the class.

You can access fields and methods for the object using dot notation, like this:

```
myBicycle Bicycle = new Bicycle();

myBicycle.brand = "Trek";
myBicycle.pedal();
```

You probably won't see many objects in the native library. Instead, the developers who implement the API will create objects. However, if you have a reference implementation or sample code on how to implement the API, you will see a lot of objects.

## Constructors

Constructors are methods used to create new instances of the class. The default constructor for the class looks like the one above, with `new Bicycle()`.

The constructor uses the same name as the class and is followed by parentheses (because constructors are methods).

Often classes have constructors that initialize the object with specific values passed in to the constructor.

For example, suppose we had a constructor that initialized the object with the brand and size:

```
public class Bicycle{

 public Bicycle(String brand, int size) {
 this.brand = model;
 this.size = size;
 }

}
```

Now I use this constructor when creating a new Bicycle object:

```
Bicycle myBicycle = new Bicycle ("Trek", 22);
```

It's a best practice to include a constructor even if it's just the default.

## Packages

Classes are organized into different packages. Packages are like folders or directories where the classes are stored. Putting classes into packages helps avoid naming conflicts.

When you create your class, if it's in a package called `vehicles`, you list this package at the top of the class:

```
package vehicles

public class Bicycle {

}
```

Classes also set boundaries on access based on the package. If the access modifier did not say `public`, the class would only be accessible to members of the same package. If the access modifier were `protected`, the class would only be accessible to the class, package, and subclasses.

When you want to instantiate the class (and your file is outside the package), you need to import the package into your class, like this:

```
import vehicles

public static void main(String[] args) {

}
```

When packages are contained inside other packages, you access the inner packages with a dot, like this:

```
import transportation.motorless.vehicles
```

Here I would have a transportation package containing a package called motorless containing a package called vehicles. Package naming conventions are like URLs in reverse (com > yoursitename > subdomain).

Maven handles package management for Java projects. Maven will automatically go out and get all the package dependencies for a project when you install a Maven project.

## Exceptions

In order to avoid broken code, developers anticipate potential problems with exception handling. Exceptions basically say, if there's an issue here, flag the error with this exception and then continue on through the code.

Different types of errors throw different exceptions. By identifying the type of exception thrown, you can more easily troubleshoot problems when code breaks because you know the specific error that's happening.

You can identify a specific exception the class throws in the class name after the keyword `throws` :

```
public class Bicycle throws IOException {

}
```

When you indicate the exception here, you list the type of exception using a specific Javadoc tag (explained later).

## Inheritance

Some classes can extend other classes. This means a class inherits the properties of another class. When one class extends another class, you'll see a note like this:

```
public class Bicycle extends Vehicle {

}
```

This means that `Bicycle` inherits all of the properties of `Vehicle` and then can add to them.

## Interfaces

An interface is a class that has methods with no code inside the method. Interfaces are intended to be implemented by another class that will insert their own values for the methods. Interfaces are a way of formalizing a class that will have a lot of subclasses, when you want all the subclasses to standardize on common strings and methods.

## JAR files and WAR files

The file extension for a class is .java, but when compiled by the Java Development Kit into the Java program, the file becomes .class. The .class file is binary code, which means only computers (in this case, the Java Virtual Machine, or JVM) can read it.

Developers often package up java files into a JAR file, which is like a zip file for Java projects. When you distribute your Java files, you'll likely provide a JAR file that the developer audience will add to their Java projects.

Developers will add their JAR to their class path to make the classes available to their project. To do this, you right-click your project and select **Properties**. In the dialog box, select **Java Build Path**, then click the **Libraries** tab. Then click **Add JARs** and browse to the JAR.

When you deliver a JAR file, developers can use the classes and methods available in the JAR. However, the JAR will not show them the source code, that is, the raw Java files. For this, users will consult the Javadoc.

If you're distributing a reference implementation that consists of a collection of Java source files (so that developers can see how to integrate your product in Java), you'll probably just send them a zip file containing the project.

A WAR file is a web application archive. A WAR is a compiled application that developers deploy on a server to run an application. Whereas the JAR is integrated into a Java project while the developers are actively building the application, the WAR is the deployed program that you run from your server.

That's probably enough Java to understand the different components of a Javadoc.

## Summary

Here's a quick summary of the concepts we talked about:

- **Class**: blueprints for something
- **Object**: an instance of a class
- **Methods**: what the object/class can do
- **Fields**: variables in the object/class
- **Constructor**: a method to create an object for a class
- **Package**: a folder that groups classes
- **Access modifier** (e.g. public): the scope at which a thing can be accessed
- **Interface**: a skeleton class with empty methods (used for standardizing)
- **Enum**: a data type offering predefined constants
- **Subclass**: a class that inherits the fields + methods of another class
- **JAR file**: a zip-like file containing Java classes
- **WAR file**: a compiled Java web application to be deployed on a server

# Activity: Generate a Javadoc from a sample project

Javadoc is the standard output for Java APIs, and it's really easy to build a Javadoc. The Javadoc is generated through something called a "doclet." Different doclets can parse the Java annotations in different ways and produce different outputs. But by and large, almost every Java documentation uses Javadoc. It's standard and familiar to Java developers.

## Characteristics of Javadoc

Here are some other characteristics of Javadoc:

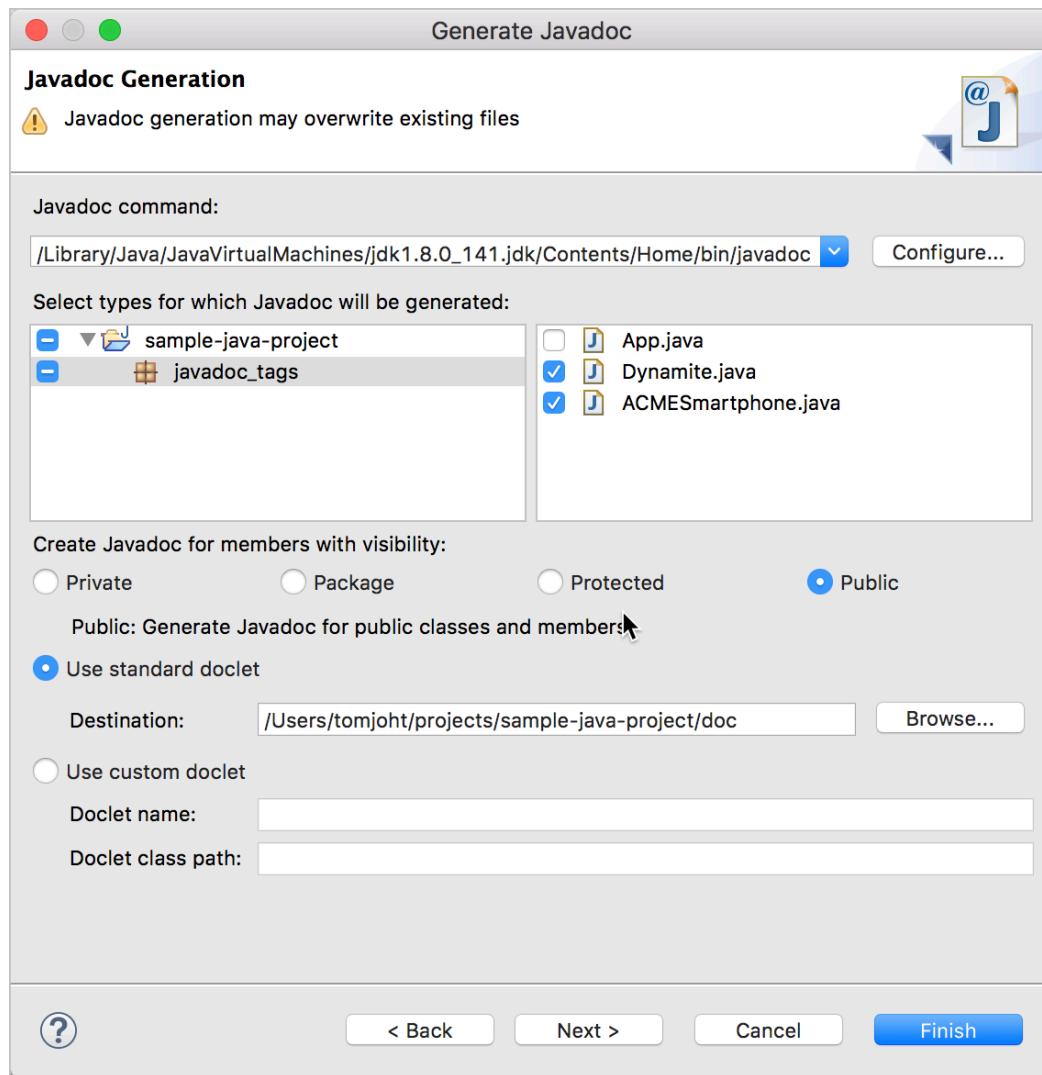
- Javadoc is supported by Oracle
- Javadoc's output integrates directly into IDEs that developers use.
- The Javadoc output is skinnable, but you can't add non-ref files to it.
- The Javadoc comment style is highly similar to most other document generators.

## Generate a Javadoc

### ACTIVITY



1. In Eclipse, go to **File > Export**.
2. Expand **Java** and select **Javadoc**. Then click **Next**.
3. Select your project and package. Then in the right pane, select the classes you want included in the Javadoc. Don't select the class that contains your main method. In this sample project, the main method is included in App.java.



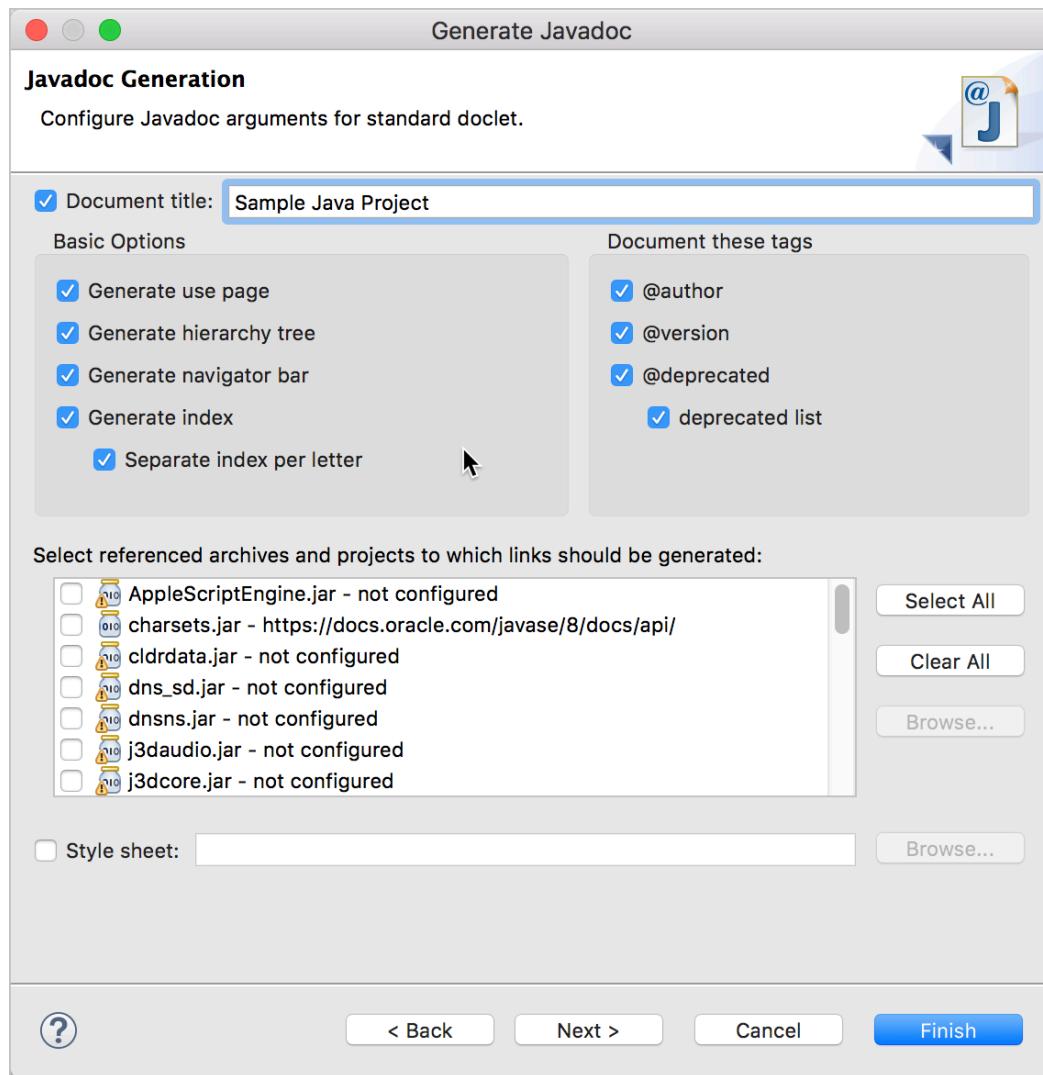
4. Select which visibility option you want: Private, Package, Protected, or Public. Generally you select **Public**.

Your API probably has a lot of helper or utility classes used on the backend, but only a select number of classes will actually be used by your developer audience. These classes are made public. It's the public classes that your developer audience will use that form the API aspect of the class library.

5. Make sure the **Use standard doclet** radio button is selected (it's selected by default).
6. Click the **Browse** button and select the output location where you want the Javadoc generated. By default, it will be generated in the same project folder as your code, but in a subfolder called **doc**. This way you can browse the Javadoc directly within your Eclipse IDE.

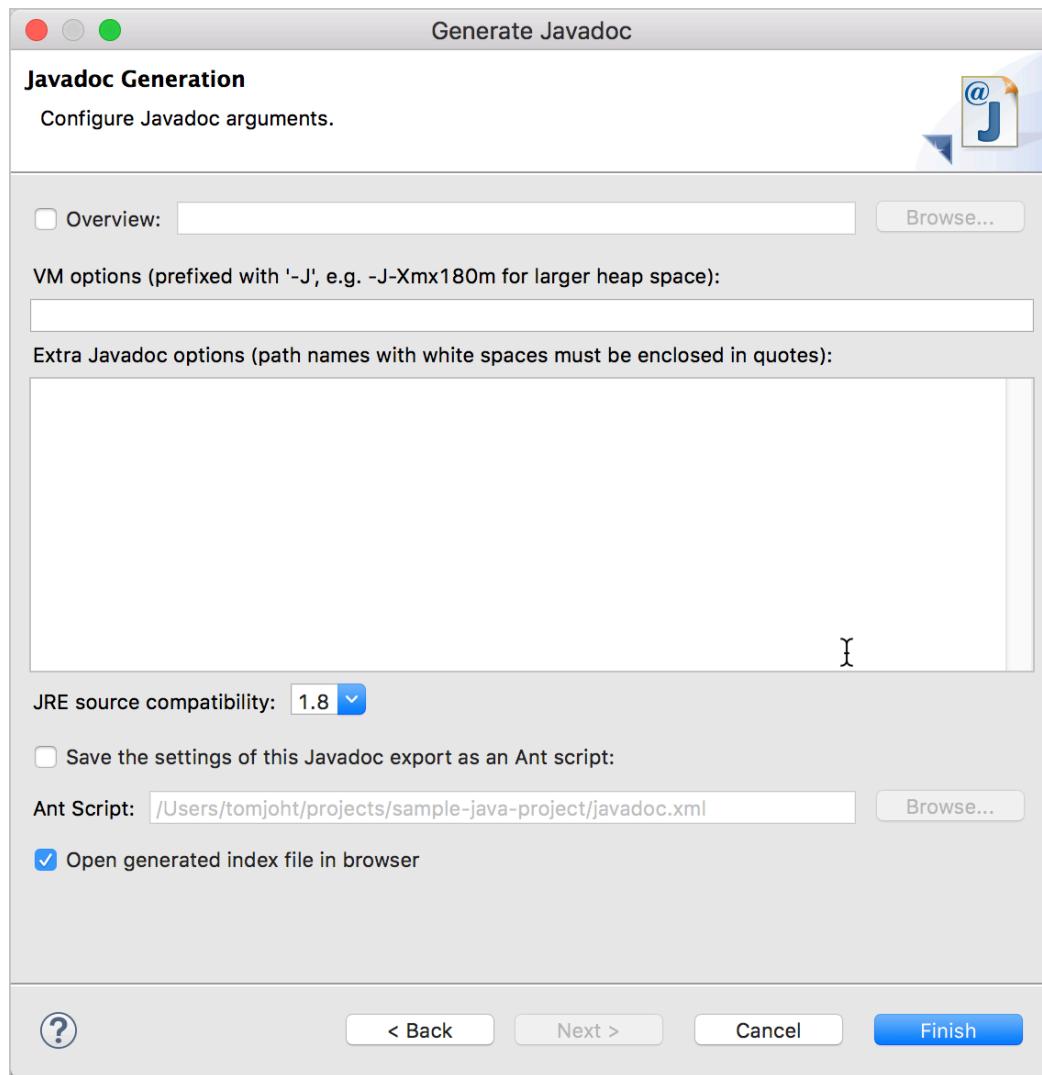
For this activity, choose a different output location (such as a folder on your desktop or in your documents) other than the default. Reason being, the project already has the generated Javadoc in a docs folder, so you might not even realize that you've generated a Javadoc file because your new output will just overwrite the existing doc files.

7. Click **Next**.



Here you can select if you want to omit some tags, such as @author and @deprecated. Generally you don't include the @author tag, since it may only be important internally, not externally. You can also select different options in the Javadoc frame. If you have a custom stylesheet, you can select it here. Most likely you would only make superficial style changes such as with colors.

8. Click **Next**.



Here you can select an HTML page that you want to be your [overview page in the Javadoc](#). You can select any HTML page and it will be included in the index.

9. Click **Finish**.

If prompted to update the Javadoc location (which likely differs from your Eclipse workspace location), do so by clicking **Yes to all**.

Browse to the destination location and open the index.html file in your browser to view the files.

## Javadoc and error checking

Javadoc also checks your tags against the actual code. If you have parameters, exceptions, or returns that don't match up with the parameters, exceptions, or returns in your actual code, then Javadoc will show some warnings.

The screenshot shows the Eclipse IDE interface. On the left, there's a tree view labeled "All Classes" containing "ACMESmartphone", "App", and "Dynamite". The main area has tabs for "OVERVIEW", "PACKAGE", "CLASS", "USE", "TREE", "DEPRECATED", "INDEX", and "HELP", with "OVERVIEW" selected. Below the tabs are links for "PREV", "NEXT", "FRAMES", and "NO FRAMES". The title "Acme Project" is displayed, followed by the text "This is my overview page." and a link "See: Description". A "Packages" tab is also visible. At the bottom, a "Problems" view shows the output of the Javadoc generation process:

```
<terminated> Javadoc Generation
Loading source files for package acme...
Constructing Javadoc information...
Standard Doclet version 1.8.0_05
Building tree for all the packages and classes...
Generating /Users/tjohnson/projects/acmeproject/doc/javadoc/acme/ACMESmartphone.html...
/Users/tjohnson/projects/acmeproject/acme/ACMESmartphone.java:41: error: reference not found
 * @exception Throws NullPointerException if model or license is null
 ^
/Users/tjohnson/projects/acmeproject/acme/ACMESmartphone.java:43: warning: no @param for model
 public ACMESmartphone(double model, String license) {
 ^
Generating /Users/tjohnson/projects/acmeproject/doc/javadoc/acme/App.html...
/Users/tjohnson/projects/acmeproject/acme/App.java:12: warning: no description for @param
 * @param args
 ^
/Users/tjohnson/projects/acmeproject/acme/App.java:13: warning: no description for @throws
```

## ACTIVITY



Try removing a parameter from a method and generate the Javadoc again. Make sure the console window is open.

# Javadoc tags

Javadoc is a document generator that looks through your Java source files for specific annotations. It parses out the annotations into the Javadoc output. Knowing the annotations is essential, since this is how the Javadoc gets created.

## Common Javadoc tags

The following are the most common tags used in Javadoc. Each tag has a word that follows it. For example, `@param latitude` means the parameter is “latitude”.

**Tip:** To see a lengthy Javadoc tag, see this [example from Oracle](#).

The following are some common Javadoc tags:

- `@author` A person who made significant contribution to the code. Applied only at the class, package, or overview level. Not included in Javadoc output. It's not recommended to include this tag since authorship changes often.
- `@param` A parameter that the method or constructor accepts. Write the description like this:  
“`@param count` Sets the number of widgets you want included.”

`@deprecated` Lets users know the class or method is no longer used. This tag will be positioned in a prominent way in the Javadoc. Accompany it with a `@see` or `{@link}` tag as well.

- 
- `@return` What the method returns.
- `@see` Creates a see also list. Use `{@link}` for the content to be linked.
- `{@link}` Used to create links to other classes or methods. Example: `{@link Foo#bar}` links to the method `bar` that belongs to the class `Foo`. To link to the method in the same class, just include `#bar`.
- `@since 2.0` The version since the feature was added.
- `@throws` The kind of exception the method throws. Note that your code must indicate an exception thrown in order for this tag to validate. Otherwise Javadoc will produce an error.  
`@exception` is an alternative tag.
- `@Override` Used with interfaces and abstract classes. Performs a check to see if the method is an override.

## Comments versus Javadoc tags

A general comment in Java code is signaled like this:

```
// sample comment...

/*
sample comment
*/
```

Javadoc does nothing with these comments.

To include content in Javadoc, you add *two asterisks* at the start, before the class or method:

```
/**
*
*
*
*
*/
```

(In Eclipse, if you type `/*` and hit return, it autofills the rest of the syntax automatically.)

The format for adding the various elements is like this:

```
/**
* [short description]
* <p>
* [long description]
*
* [author, version, params, returns, throws, see, other tags]
* [see also]
*/
```

Here's a real example of Javadoc comments for a method.

```
/**
* Zaps the roadrunner with the amount of volts you specify.
* <p>
* Do not exceed more than 30 volts or the zap function will backfire.
* For another way to kill a roadrunner, see the {@link Dynamite#blowDynamite()} method.
*
* @exception IOException if you don't enter an data type amount for the voltage
* @param voltage the number of volts you want to send into the roadrunner's body
* @see #findRoadRunner
* @see Dynamite#blowDynamite
*/
public void zapRoadRunner(int voltage) throws IOException {
 if (voltage < 31) {
 System.out.println("Zapping roadrunner with " + voltage + " volts!!!!");
 }
 else {
 System.out.println("Backfire!!! zapping coyote with 1,000,000 volts!!!!");
 }
}
```

## Where the Javadoc tag goes

You put the Javadoc description and tags *before* the class or method (no need for any space between the description and class or method).

## What elements you add Javadoc tags to

You add Javadoc tags to classes, methods, and fields.

- For the @author and @version tags, add them only to classes and interfaces.
- The @param tags get added only to methods and constructors.
- The @return tag gets added only to methods.
- The @throws tag can be added to classes or methods.

## Public versus private modifiers and Javadoc

Javadoc only includes classes, methods, etc. marked as public. Private elements are not included. If you omit `public`, the default is that the class or method is available to the package only. In this case, it is not included in Javadoc.

## The description

There's a short and long description. Here's an example showing how the description part is formatted:

```
/**
 * Short one line description.
 * <p>
 * Longer description. If there were any, it would be
 * here.
 * <p>
 * And even more explanations to follow in consecutive
 * paragraphs separated by HTML paragraph breaks.
 *
 * @param variable Description text text text.
 * @return Description text text text.
 */
public int methodName (...) {
// method body with a return statement
}
```

(This example comes from [Wikipedia entry](#).)

The short description is the first sentence, and gets shortened as a summary for the class or method in the Javadoc. After a period, the parser moves the rest of the description into a long description. Use `<p>` to signal the start of a new paragraph. You don't need to surround the paragraphs with opening and closing `<p>` tags – the Javadoc compiler automatically adds them.

Also, you can use HTML in your descriptions, such as an unordered list, code tags, bold tags, or others.

After the descriptions, enter a blank line (for readability), and then start the tags. You can't add any more description content below the tags. Note that only methods and classes can have tags, not fields. Fields (variables) just have descriptions.

Note that the first sentence is much like the `shortdesc` element in DITA. This is supposed to be a summary of the entire class or method. If one of your words has a period in it (like `Dr. Jones`), then you must remove the space following the period by adding `Dr.&nbsp;Jones` to connect it.

Avoid using links in that first sentence. After the period, the next sentence shifts to the long paragraph, so you really have to load up that first sentence to be descriptive.

The verb tense should be present tense, such as *gets*, *puts*, *displays*, *calculates*...

What if the method is so obvious (for example, `printPage`) that your description (“prints a page”) becomes obvious and looks stupid? Oracle says in these cases, you can omit saying “prints a page” and instead try to offer some other insight:

Add description beyond the API name. The best API names are “self-documenting”, meaning they tell you basically what the API does. If the doc comment merely repeats the API name in sentence form, it is not providing more information. For example, if method description uses only the words that appear in the method name, then it is adding nothing at all to what you could infer. The ideal comment goes beyond those words and should always reward you with some bit of information that was not immediately obvious from the API name. – [How to write javadoc comments](#)

## Avoid @author

Commenting on Javadoc best practices, one person says to avoid `@author` because it easily slips out of date and the source control provides better indication of the last author. ([Javadoc coding standards](#)

## Order of tags

Oracle says the order of the tags should be as follows:

```
@author (classes and interfaces)
@author (classes and interfaces)
@param (methods and constructors)
@return (methods)
@throws (@exception is an older synonym)
@see
@since
@serial
@deprecated
```

## @param tags

`@param` tags only apply to methods and constructors, both of which take parameters.

After the `@param` tag, add the parameter name, and then a description of the parameter, in lowercase, with no period, like this:

```
@param url the web address of the site
```

The parameter description is a phrase, not a full sentence.

The order of multiple @param tags should mirror their order in the method or constructor.

Stephen Colebourne recommends adding an extra space after the parameter name to increase readability (and I agree).

As far as including the data type in the parameter description, Oracle says:

By convention, the first noun in the description is the data type of the parameter.  
(Articles like “a”, “an”, and “the” can precede the noun.) An exception is made for the primitive int, where the data type is usually omitted. – [How to write doc comments using Javadoc](#)

The example they give is as follows:

```
@param ch the character to be tested
```

However, the data type is visible from the parameters in the method. So even if you don't include the data types, it will be easy for users to see what they are.

Note that you can have multiple spaces after the parameter name so that your parameter definitions all line up.

@param tags must be provided for every parameter in a method or constructor. Failure to do so will create an error and warning when you render Javadoc.

Note that usually classes don't have parameters. There is one exception: Generics. Generic classes are classes that work with different type of objects. The object is specified as a parameter in the class in diamond brackets: `<>`. Although the Javadoc guidance from Oracle doesn't mention them, you can add a @param tag for a generic class to note the parameters for the generic class. See this [StackOverflow post](#) for details. Here's an example from that page:

```
/**
 * @param <T> This describes my type parameter
 */
class MyClass<T>{

}
```

## @return tag

Only methods return values, so only methods would receive a @return tag. If a method has `void` as a modifier, then it doesn't return anything. If it doesn't say `void`, then you must include a @return tag to avoid an error when you compile Javadoc.

## @throws tag

You add @throws tags to methods or classes only if the method or class throws a particular kind of error.

Here's an example:

```
@throws IOException if your input format is invalid
```

Stephen Colebourne recommends starting the description of the throws tag with an “if” clause for readability.

The @throws feature should normally be followed by “if” and the rest of the phrase describing the condition. For example, “@throws if the file could not be found”. This aids readability in source code and when generated.

If you have multiple throws tag, arrange them alphabetically.

## Doc comments for constructors

It's a best practice to include a constructor in a class. However, if the constructor is omitted, Javadoc automatically creates a constructor in the Javadoc but omits any description of the constructor.

Constructors have @param tags but not @return tags. Everything else is similar to methods.

## Doc comments for fields

Fields have descriptions only. You would only add doc comments to a field if the field were something a user would use.

## Cases where you don't need to add doc comments

Oracle says there are 3 scenarios where the doc comments get inherited, so you don't need to type them:

When a method in a class overrides a method in a superclass  
When a method in an interface overrides a method in a superinterface  
When a method in a class implements a method in an interface – [How to write Javadoc comments](#)

## @see tags

The @see tag provides a see also reference. There are various ways to denote what you're linking to in order to create the link. If you're linking to a field, constructor, or method within the same field, use `#`.

If you're linking to another class, put that class name first followed by the `#` and the constructor, method, or field name.

If you're linking to a class in another package, put the package name first, then the class, and so on. See this sample from Oracle:

```
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id)
@see Class#method(Type, Type,...)
@see Class#method(Type id, Type id,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id)
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type id, Type, id)
```

#### – How to write Javadoc comments

## Links

You can create links to other classes and methods using the `{@link}` tag.

Here's an example from [Javadoc coding standards](#) on making links:

```
/**
 * First paragraph.
 * <p>
 * Link to a class named 'Foo': {@link Foo}.
 * Link to a method 'bar' on a class named 'Foo': {@link Foo#bar}.
 * Link to a method 'baz' on this class: {@link #baz}.
 * Link specifying text of the hyperlink after a space: {@link Foo the Foo class}.
 * Link to a method handling method overload {@link Foo#bar(String,int)}.
 */
public ...
```

To link to another method within the same class, use this format: `{@link #baz}`. To link to a method in another class, use this format: `{@link Foo#baz}`. However, don't over hyperlink. When referring to other classes, you can use `<code>` tags.

To change the linked text, put a word after `#baz` like this: `@see #baz Baz method`.

## Previewing Javadoc comments

In Eclipse, you can use the Javadoc tab at the bottom of the screen to preview the Javadoc information included for the class you're viewing.

The screenshot shows a code editor with Java code and its corresponding Javadoc documentation. The code is as follows:

```
50 if (voltage < 31) {
51 System.out.println("Zapping roadrunner with " + voltage + " "
52 } else {
53 System.out.println("Backfire!!! zapping coyote with 1,000,0
54 }
55 }
56
57 }
```

The Javadoc documentation is displayed below the code:

- void javadoc\_tags.ACME\_Smartphone.zapRoadRunner(int voltage) throws IOException**
- Zaps the roadrunner with the amount of volts you specify.
- Do not exceed more than 30 volts or the zap function will backfire. For another way to kill a roadrunner, see [Dynamite.blowDynamite](#).
- Parameters:**
  - voltage the number of volts you want to send into the roadrunner's body
- Throws:**
  - [IOException](#) – if you don't enter an data type amount for the voltage
- See Also:**
  - [findRoadRunner](#)
  - [Dynamite.blowDynamite](#)

## More information

- Oracle's explanation of Javadoc tags
- Javadoc

# Explore the Javadoc output

The Javadoc output hasn't changed much in the past 20 years, so in some sense it's predictable and familiar. On the other hand, the output is dated and lacks some critical features, like search, or the ability to add more pages. Anyway, it is what it is.

## Class summary

The class summary page shows a short version of each of the classes. The description you write for each class (up to the period) appears here. It's kind of like a quick reference guide for the API.

The screenshot shows a Javadoc interface with a sidebar on the left containing a list of 'All Classes' including 'ACMESmartphone' and 'Dynamite'. The main content area has a header 'PACKAGE CLASS USE TREE DEPRECATED INDEX HELP' and navigation links 'PREV PACKAGE' and 'NEXT PACKAGE' along with 'FRAMES' and 'NO FRAMES' options. Below this, the title 'Package javadoc\_tags' is displayed. A 'Class Summary' section is highlighted with an orange background, containing a table with two rows. The first row has columns 'Class' and 'Description'. The second row lists 'ACMESmartphone' with the description 'Works like a regular smartphone but also tracks roadrunners.' and 'Dynamite' with the description 'Provides ways to explode dynamite to blow up roadrunners.' At the bottom of the main content area is another set of navigation links: 'PACKAGE CLASS USE TREE DEPRECATED INDEX HELP' and 'PREV PACKAGE' and 'NEXT PACKAGE' along with 'FRAMES' and 'NO FRAMES'.

Class	Description
ACMESmartphone	Works like a regular smartphone but also tracks roadrunners.
Dynamite	Provides ways to explode dynamite to blow up roadrunners.

You click a class name to dive into the details.

## Class details

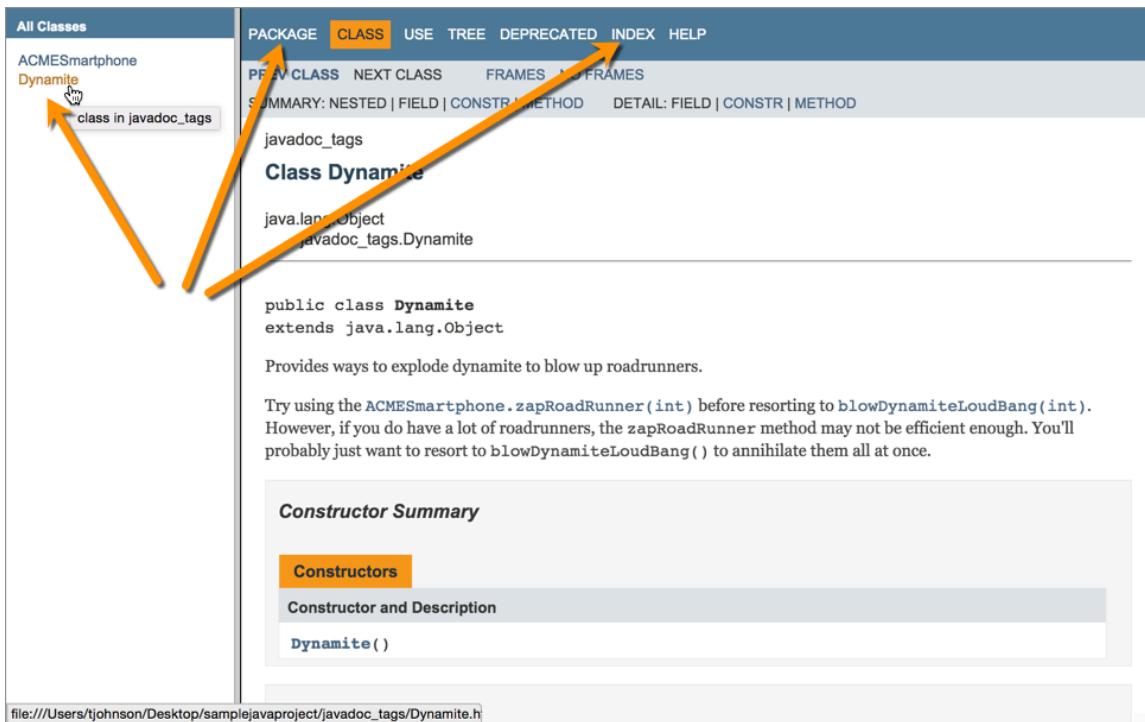
When you view a class page, you're presented with a brief summary of the fields, constructors, and methods for the class. Again this is just an overview. When you scroll down, you can see the full details about each of them.

<b>All Classes</b>  ACMESmartphone Dynamite	<p>PACKAGE <b>CLASS</b> USE TREE DEPRECATED INDEX HELP</p> <p>PREV CLASS NEXT CLASS FRAMES NO FRAMES</p> <p>SUMMARY: NESTED   FIELD   CONSTR   METHOD DETAIL: FIELD   CONSTR   METHOD</p> <p>javadoc_tags</p> <p><b>Class ACMESmartphone</b></p> <p>java.lang.Object javadoc_tags.ACMESmartphone</p> <hr/> <p>public class <b>ACMESmartphone</b> extends java.lang.Object</p> <p>Works like a regular smartphone but also tracks roadrunners.</p> <p>The ACME Smartphone can perform similar functions as other smartphones, such as making phone calls, sending text messages, and browsing the web. However, the ACME Smartphone also enables GPS tracking on roadrunners. You can monitor the location of all roadrunners within a 20 mile radius using the RoadRunner Tracker app.</p> <p>Note that the RoadRunner Tracker app requires you to be connected to wifi. It will not work on cellular data.</p> <p><b>Since:</b> 1.3</p> <p><b>Version:</b> 2.0</p> <p><b>Author:</b> Tom Johnson</p> <p><b>See Also:</b></p>
------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<http://docs.oracle.com/javase/7/docs/api/>

## Other navigation

If you click **Package** at the top, you can also browse the classes by package. Or you can go to the classes by clicking the class name in the left column. You can also browse everything by clicking the **Index** link.



For more information about how the Javadoc is organized, click the **Help** button.

# Make edits to Javadoc tags

It's pretty common for developers to add Javadoc tags and brief comments as they're creating Java code. In fact, if they don't add it, the IDE will usually produce a warning error.

However, the comments that developers add are usually poor, incomplete, or incomprehensible. A tech writer's job with Javadoc is often to edit the content that's already there, providing more clarity, structure, inserting the right tags, and more.

## What to look for when editing Javadoc content

When you make edits to Javadoc content, look for the following:

- **Missing doc.** Lots of Javadoc is incomplete. Look for missing documentation.
- **Consistent style.** See if the existing tags follow Java's style conventions.
- **Clarity.** Some descriptions are unintelligible due to the curse of knowledge, but it's hard to judge without a stronger grasp of Java.

## ACTIVITY



Make some edits to a class and method. Then regenerate the Javadoc and find your changes. See how they get rendered in the output.

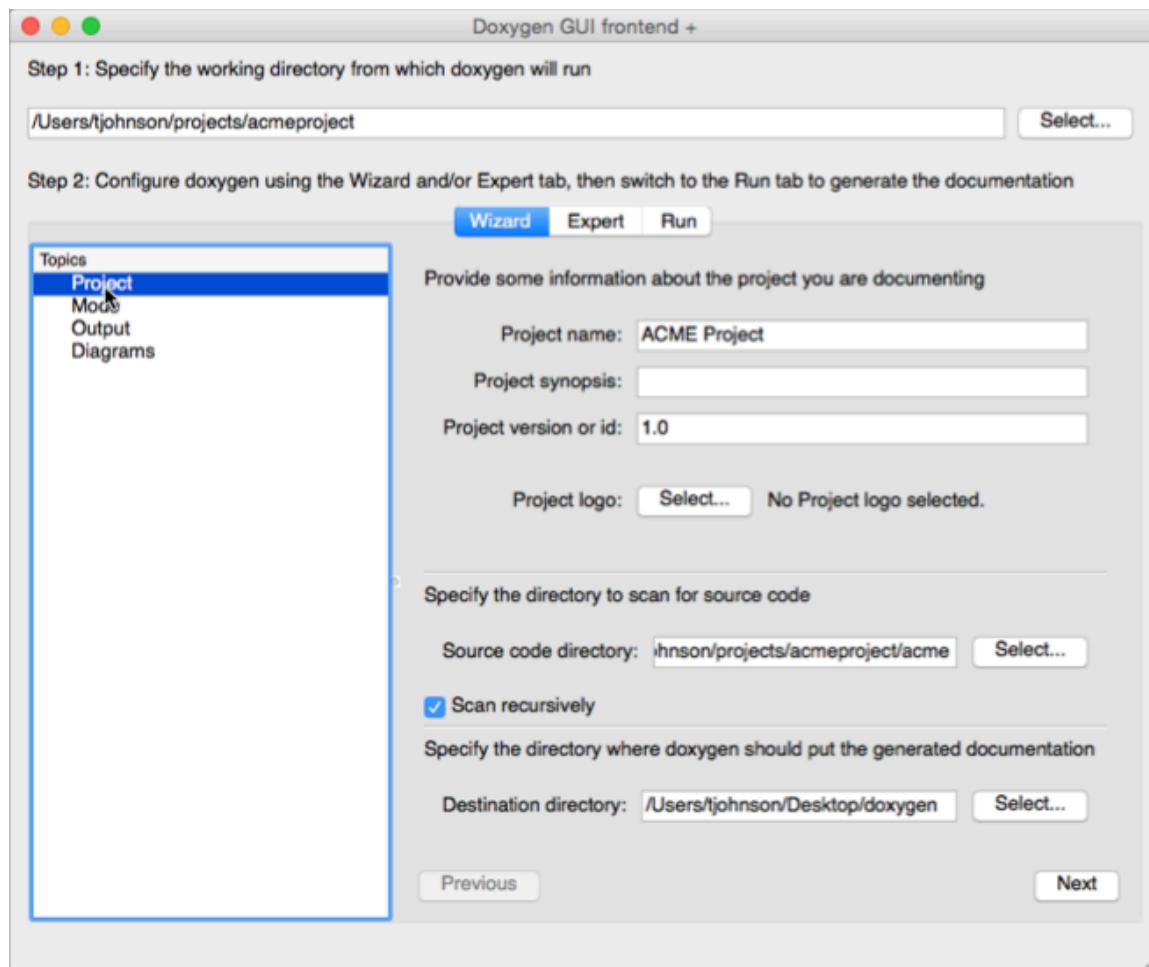
# Doxygen, a document generator mainly for C++

An alternative to Javadoc is Doxygen. Doxygen works highly similarly to Javadoc, except that you can process more languages (Java, C++, C#, and more) with it. Doxygen is most commonly used with C++. Additionally, there's a GUI tool (called Doxywizard) that makes it really easy to generate the file.

## Download Doxywizard

You can download the Doxywizard tool when you install Doxygen. See the [Doxygen](#) download page for more information.

Here's Doxygen's front-end GUI generator (Doxywizard):



Here's the Doxygen output:

The screenshot shows the Doxygen interface for the 'ACME Project 1.0' documentation. The left sidebar has a tree view of class hierarchies. The main panel shows the 'acme.ACMESmartphone Class Reference' with sections for 'Public Member Functions' and 'Public Attributes'. The 'Detailed Description' section contains a note about the phone tracking roadrunners.

By the way, you don't need to use the wizard. You can also just generate Doxygen through a configuration file. This is how developers typically run Doxygen builds from a server.

In contrast to Javadoc, Doxygen also allows you to incorporate external files written in Markdown. And Doxygen provides a search feature. These are two features that Javadoc lacks.

Doxygen is maintained by a single developer and, like Javadoc, hasn't changed much over the years. In my opinion, the interface is highly dated and kind of confusing.

## Integrating builds automatically

In a lot of developer shops, document generators are integrated into the software build process automatically. Doxygen allows you to create a configuration file that can be run from the command line (rather than using the frontend GUI). This means when developers build the software, the reference documentation is automatically built and included in the output.

## Other document generators

You don't need to limit yourself to either Javadoc or Doxygen. There are dozens of different document generators for a variety of languages. Just search for "document generator + {programming language}" and you'll find plenty. However, don't get very excited about this genre of tools. Document generators are somewhat old, produce static front-ends that look dated, are often written by engineers for other engineers, and not very flexible.

Perhaps the biggest frustration of document generators is that you can't really integrate the rest of your documentation with them. You're mostly stuck with the reference doc output. You'll need to also generate your how-to guides and other tutorials, and then link to the reference doc output. As such, you won't end up with a single integrated experience of documentation. Additionally, it will be hard to create links between the two outputs.

# Create non-ref docs with native library APIs

Although much attention tends to be given to the reference documentation with APIs, actually the bulk of what technical writers usually do with native library API docs is provide non-reference documentation. This is the stuff that engineers rarely write.

Engineers will throw a quick description of a class in a file and generate a Javadoc, and they'll give that Javadoc to the user as if it represents a complete set of documentation, but reference docs don't tell even half the story.

## Reference docs can be an illusion for real doc

Jacob Kaplan Moss says that reference docs can be an illusion:

... auto-generated documentation is worse than useless: it lets maintainers fool themselves into thinking they have documentation, thus putting off actually writing good reference by hand. If you don't have documentation just admit to it. Maybe a volunteer will offer to write some! But don't lie and give me that auto-documentation crap. – Jacob Kaplan Moss

Other people seem to have [similar opinions](#):

Auto-generated documentation that documents each API end-point directly from source code have their place (e.g., it's great for team that built the API and it's great for a reference document) but hand-crafted quality documentation that walks you through a use case for the API is invaluable. It should tell you about the key end-points that are needed for solving a particular problem and it should provide you with code samples.”

In general, document generators don't tell you a whole lot more than you would discover by browsing the source code itself. Some people even refer to auto-generated docs as a glorified source-code browser.

## Reference docs are feature-based, not task-based

One of the main problems with reference documentation is that it's feature based rather than task based. It's the equivalent of going tab-by-tab through an interface and describing what's on each tab, what's in each menu, and so on. We know that's a really poor way to approach documentation, since users organize their mental model by the tasks they want to perform.

When you write API documentation, consider the tasks that users will want to do, and then organize your information that way. Reference the endpoints as you explain how to accomplish the tasks. Users will refer to the reference docs as they look for the right parameters, data types, and other class details. But the reference docs won't guide them through tasks alone.

# Getting a job in API documentation

The job market for API technical writers.....	450
How much code do you need to know? .....	455

# 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.

## Ability to 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. Here's the most common scenario. The company has a REST API for interacting with their services. However, 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 want to follow the documentation for your specific platform.

The screenshot shows the Algolia Documentation homepage. The left sidebar includes links for Documentation, How It Works, API Documentation, FAQ, and REST API. The main content area features a "Documentation" header with a "Get started with Basics Tutorial" link. Below this is a section titled "API Clients & Integrations" containing a grid of icons for various programming languages and frameworks: Ruby, Rails, Python, PHP, Symfony, Node.js, JavaScript, Go, Java, cURL, Objective-C, Swift, Android, C#, WordPress, Mage..., Laravel, WooCommerce, Parse, and Jekyll. At the bottom right is a "Need help?" button with a question mark icon.

Although users could construct their own code when using the REST endpoints, most developers would rather leverage existing code to just copy and paste what they need.

When I worked at Badgeville, we developed a collection of JavaScript widgets that code developers could easily copy and paste into their web pages, making a few adjustments as needed. Sure developers could have created their own JavaScript widget code 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 language.

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 references 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.”

## Technical writers who are former programmers

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, and more 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 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.

This 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 embrace the terminology already will be somewhat lost.

Remember that the terminology and model will vary from one language and platform to the next. One user may speak fluently in Ruby, but that language may not connect with somebody who is a .NET developer. Consequently speaking “geek” can both connect with some developers and backfire with other developers.

## Wide, not deep understanding of programming

Although you may have client implementations in a variety of programming languages, the implementations will be brief. The core documentation needed will most likely be for the REST API, and then 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 document them. You're probably just scratching the surface with each of them.

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 6 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 like Spanish. I studied Spanish for 3 years in high school, but it wasn't until I lived in Venezuela and interacted with locals for 6 months continuously speaking Spanish that the language finally clicked for me.

As such, you might consider diving deep into one core programming language (like Java) and 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 it. It's best if you can make learning programming one of your "hobbies."

## Diverse technical landscape

The technical landscape is diverse, so the generalizations I'm providing here may not hold true in all companies. You may be in a Java or JavaScript shop where all you need to know is Java/JavaScript. If that's the case, you'll need to develop a deeper knowledge of the programming language so you can provide more depth.

However, with the proliferation of REST APIs, this scenario is much less common. Companies can't afford to cater only to one programming language. Doing so drastically reduces their audience and limits their revenue. The advantages of providing a universally accessible API using any language platform usually outweigh the specifics you get from a native library API.

The company I currently work for has a Java, .NET, and C++ API that each do the same thing but in different languages. Maintaining the same functionality across three separate platforms is a serious challenge for developers. Not only is it difficult to find skill sets for developers across these three platforms, having multiple code bases makes it harder to test and maintain the code. It's 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's almost impossible to get users to upgrade to the latest version of your API. You have 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 can take 6 months or more.

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 backwards compatibility), and developers can simply continue using their same deployment code.

The more you can facilitate implementation in the user's desired language, the higher your chances of implementation — which means greater product adoption, revenue, and success.

## Consolations for technical writers

This proliferation of code and platforms creates more pressure on the multi-lingual capabilities of technical writers. Here's one consolation, though. If you can understand what's going on in one programming language, then 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 may refer to "functions" instead of "classes," and so on. Even so, getting all the language right can be a serious challenge, which is why it's so hard to find technical writers who have skills for producing developer documentation.

With this scenario of having multiple client implementations, you'll face other challenges, such as maintaining consistency across the various platforms. As you try to single source your explanations for various languages, your documentation code will become complex and difficult to maintain.

Additionally, product managers may want you to push out separate outputs within each programming language channel to keep things simple for the users. Can you imagine pushing out a dozen different outputs across different languages for content that follows highly similar patterns and has common explanations but differs in just enough ways to make single sourcing from the same core content an act of sorcery? Here is where you have to put your technical writing wizard hat on and pull off level 9 incantations.

## 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 likely need to incorporate at least a regular regimen of programming study.

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 (this is the hallmark of a good education).

But in discussions with hiring managers looking to fill 6-month 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 developer possesses.

## Strategies to get by

When you work in developer documentation environments, one strategy is to interview engineers about what's going on in the code, 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 what the code is doing.

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 strategy behind the code.

Just remember that even though someone is a developer, it doesn't mean he or she is an expert with all code. 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 have to use a progressive-disclosure approach so that expert users aren't bogged down with novice-level detail; at the same time, you have to make this additional detail available for those who need it. 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.

# 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.

## Adequate versus Deep Technical Knowledge

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 is often enough to perform the tasks needed for documentation.

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](#))

## Tech comm work that gets neglected

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 may 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) publish documentation. Pages may be rich with technical detail, but the degree of ROT (redundant, outdated, trivial content) gets compounded, navigation suffers, clarity gets muddled, and no one can find anything.

Just today I spent a good chunk of time trying to find information on a corporate wiki, only to be met with mountains of poorly written pages, abandoned content, impossible to navigate spaces, and other issues. It was a completely frustrating experience.

Anyone who has a wiki at their company usually has a similar experience. Because no one really cares about internal wikis, companies let them degenerate into content junkyards.

## Times when deeper tech knowledge is needed

In contrast to Wood and Rhea, [James Neiman](#), an experienced API technical writer, says that tech writers need an engineering background, such as a computer science degree, 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.

Neiman also says you may 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 than Wood and Rhea suggest.

James Neiman and [Andrew Davis](#) recently gave a presentation titled [Finding the right API Technical Writer](#) at a API conference in London last October.

See [this video recording on YouTube](#) (around the 23 minute mark) for the highlights.

Clearly, Neiman argues 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. If you're in a situation where the code is over your head, developers may send you chunks of code to add to the documentation.

Without the technical acumen to fully understand, test, and integrate the code in meaningful ways, you will be at the mercy of engineers and their terse explanations or cryptic inline comments. Your role will be reduced more to scribe than writer.

Neiman says in one company, he tested out the code from engineers and found that much of the code 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.

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.

## 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 it?

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?

There are a lot of questions to answer about just how to learn code. But a few conclusions are clear:

- Developer documentation requires some familiarity with code.
- You have to understand explanations from engineers, including the terms used.
- You should be able to test code from engineers.
- Learning code will require a constant effort.
- Learning to code badly may be enough to create good documentation.

## It's okay to be a bad coder

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 knowledge somewhat disappointing. After all, if you want to excel in your career, usually this means mastering something in a thorough way.

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, I've since realized that this mindset is misguided. My role as a technical writer is not to code nor even to develop code. My role is 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 disappointed in being a bad coder.

# Resources and glossary

Glossary.....	459
Overview for exploring other REST APIs .....	466
EventBrite example: Get event information.....	467
Flickr example: Retrieve a Flickr gallery .....	474
Klout example: Retrieve Klout influencers.....	483
Aeris Weather Example: Get wind speed .....	493
API documentation survey .....	499

# 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? \(page 23\)](#).

## 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](https://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 the [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](https://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. See [apiary.io](https://apiary.io).

## API Blueprint

The API Blueprint spec is an alternative to OpenAPI or RAML. API Blueprint is written in a Markdown-flavored syntax. See [API Blueprint \(page 402\)](#) 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](https://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 the 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, a clone is a copy of the repository. 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 \(page 54\)](#) and [Understand curl more \(page 56\)](#).

### 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 variant information about the same resource. Endpoints usually have brief descriptions similar to the overall resource description but shorter. See [Endpoints and methods \(page 95\)](#).

### 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 \(page 253\)](#) in this course as well as on [pages.github.com/](https://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 \(page 209\)](#) more than [HATs \(page 212\)](#).

**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/>.

**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 \(page 65\)](#) 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.

**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 \(page 388\)](#). 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 elements 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.

### 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.

### 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. Not all endpoints contain each type of parameter. See [Documenting parameters \(page 101\)](#) for more.

### Path parameters

Parameters that appear within the path of the endpoint, before the query string ( [?](#) ). These are usually set off within curly braces.

## Pelican

A static site generator based on Python. See <https://github.com/getpelican/pelican>.

## 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, 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 <https://git-scm.com/docs/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 make `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 `?`.

## RAML

Stands 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 \(page 388\)](#) 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.

## request body parameters

Parameters that are included in the request body. Usually submitted as JSON.

## 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.

## 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.

### **response**

The information returned by an API after a request is made. Responses are usually in either JSON or XML format.

### **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.

## **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? \(page 23\)](#).

### **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.

### **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 <https://www.staticgen.com/sphinx>.

### **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 \(page 274\)](#) in this course for a deep-dive into the most popular static site generator, or [Staticgen](#) for a list of all static site generators.

### **Swagger**

Refers to general API tooling to support OpenAPI specifications. See [swagger.io/](#).

### **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](#).

### **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.

### **Swagger UI**

A display framework. The most common way to parse a OpenAPI specification document and produce 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 \(page 372\)](#).

### **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](#) for details.

### **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 \(page 384\)](#).

# Overview for exploring other REST APIs

In this resources section, I explore some other REST APIs and code for some specific scenarios. This experience will give you more exposure to different REST APIs, how they're organized, the complexities and interdependency of endpoints, and more.

## Attack the challenge first, then read the answer

There are several examples with different APIs. A challenge is listed for each exercise. First, try to solve the challenge on your own. Then follow along in the sections below to see how I approached it.

In these examples, I usually printed the code to a web page to visualize the response. However, that part is not required in the challenge. (It mostly makes the exercise more fun to me.)

## Exercises

The following exercises are available:

- [EventBrite example: Get event information \(page 467\)](#)
- [Flickr example: Retrieve a Flickr gallery \(page 474\)](#)
- [Klout example: Retrieve Klout influencers \(page 483\)](#)
- [Aeris Weather Example: Get wind speed \(page 493\)](#)

## Shortcuts for API keys

Each API requires you to use an API key, token, or some other form of authentication. You can register for your own API keys, or you can [use my keys here](#).

## Swap out APIKEY in code samples

I never insert API keys in code samples for a few reasons:

- API keys expire
- API keys posted online get abused
- Customizing the code sample is a good thing

When you see `APIKEY` in a code sample, remember to swap in an API key there. For example, if the API key was `123`, you would delete `APIKEY` and use `123`.

# Eventbrite example: Get event information

<https://www.eventbrite.com/myevent?eid=17920884849>

IO6EB7MM6TSCIL2TIOHC

Use the [Eventbrite API](#) to get the event title and description of an event.

## About Eventbrite

Eventbrite is an event management tool, and you can interact with it through an API to pull out the event information you want. In this example, you'll use the Eventbrite API to print a description of an event to your page.

### 1. Get an anonymous OAuth token

To make any kind of requests, you'll need a token, which you can learn about in the [Authentication section](#).

If you want to sign up for your own token, create and register your app [here](#). Then click **Show Client Secret and OAuth Token** and copy the “Anonymous access OAuth token.”

### 2. Determine the resource and endpoint you need

The Eventbrite API documentation is here: [developer.eventbrite.com](https://developer.eventbrite.com). Look through the endpoints available (listed under Endpoints in the sidebar). Which endpoint should we use?

To get event information, we'll use the [events](#) object.

[Eventbrite APIv3 Documentation](#) > Events

## Events

### GET /events/search/

Allows you to retrieve a paginated response of public **event** objects from across Eventbrite's directory, regardless of which user owns the event.

#### Parameters

NAME	TYPE	REQUIRED	DESCRIPTION
q	<b>string</b>	No	Return events matching the given keywords.
since_id	<b>string</b>	No	Return events after this Event ID.
popular	<b>boolean</b>	No	Boolean for whether or not you want to only return popular results.
sort_by	<b>string</b>	No	Parameter you want to sort by - options are "id", "date", "name", "city", "distance" and "best". Prefix with a hyphen to reverse the order, e.g. "-date"

Instead of calling them "resources," the Eventbrite API uses the term "objects."

The events object allows us to "retrieve a paginated response of public event objects from across Eventbrite's directory, regardless of which user owns the event."

The events object has a lot of different endpoints available. However, the GET `events/:id/` URL, described [here](#) seems to provide what we need.

The Eventbrite docs convention is to use `:id` instead of `{id}` to represent values you pass into the endpoint. I don't recommend this convention, as it seems non-standard. The convention for "path parameters," as they're called, with [OpenAPI specs \(page 310\)](#) would be to use `{id}`.

### 3. Construct the request

Reading the [quick start page](#), the sample request format is here:

```
https://www.eventbriteapi.com/v3/users/me/?token=MYTOKEN
```

This is for a users object endpoint, though. For events, we would change it to this:

```
https://www.eventbriteapi.com/v3/events/:id/?token={your api key}
```

Find an ID of an event you want to use, such as [this event](#):

The screenshot shows the Eventbrite Event Dashboard for a completed event titled "An Aggressive Approach to Concise Writing, with Joe Welinske". The event is a Webinar (online through gotomeeting) scheduled for Thursday, September 24, 2015, from 12:00 PM to 1:00 PM (PDT). The dashboard includes sections for Order Options, Invite & Promote, Analyze, Manage Attendees, and Extensions. A prominent feature is a "Completed" section with a green ribbon icon and the message "Congratulations on completing your event!". It also displays ticket sales statistics: 23 Tickets Sold / 24, with a progress bar showing 96% completion. Below this, there's a circular progress bar for "Tickets sold All time" which is mostly green. Other metrics shown include 23 Tickets sold, 12 Page views, and 12 Invites.

(You have to sign in to Eventbrite to see this event page.)

The event ID appears in the URL. Now populate the request with the ID of this event:

```
https://www.eventbriteapi.com/v3/events/17920884849/?token={your api key}
```

(If you need some sample API keys, you can use the [API keys listed here](#).)

## 4. Make a request and analyze the response

Now that you have an endpoint and API token, make the request. You can actually just go to this URL in your browser to see the response.

The response from the endpoint is as follows:

```
{
 "name": {
 "text": "An Aggressive Approach to Concise Writing, with Joe Welinske",
 "html": "An Aggressive Approach to Concise Writing, with Joe Welinske"
 },
 "description": {
 "text": "Webinar Description\r\nWriting concisely is one of the fundamental skills central to any mobile user assistance. The minimal screen real estate can\u2019t support large amounts of text and graphics without extensive gesturing by the users. Using small font sizes just makes the information unreadable unless the user pinches and stretches the text. Even outside of the mobile space, your ability to streamline your content improves the likelihood it will be effectively consumed by your target audience. This session offers a number of examples and techniques for reducing the footprint of your prose while maintaining a quality message. The examples used are in the context of mobile UA but can be applied to any technical writing situation.\r\nAbout Joe WelinskeJoe Welinske specializes in helping your software development effort through crafted communication. The best user experience features quality words and images in the user interface. The UX of a robust product is also enhanced through comprehensive user assistance. This includes Help, wizards, FAQs, videos and much more. For over twenty-five years, Joe has been providing training, contracting, and consulting services for the software industry. Joe recently published the book, Developing User Assistance for Mobile Apps. He also teaches courses for Bellevue College, the University of California, and the University of Washington. Joe is an Associate Fellow of STC.",
 "html": "<P>Webinar Description</P>\r\n<P>Writing concisely is one of the fundamental skills central to any mobile user assistance. The minimal screen real estate can\u2019t support large amounts of text and graphics without extensive gesturing by the users. Using small font sizes just makes the information unreadable unless the user pinches and stretches the text.

Even outside of the mobile space, your ability to streamline your content improves the likelihood it will be effectively consumed by your target audience.

This session offers a number of examples and techniques for reducing the footprint of your prose while maintaining a quality message. The examples used are in the context of mobile UA but can be applied to any technical writing situation.</P>\r\n<P>About Joe Welinske
Joe Welinske specializes in helping your software development effort through crafted communication. The best user experience features quality words and images in the user interface. The UX of a robust product is also enhanced through comprehensive user assistance. This includes Help, wizards, FAQs, videos and much more. For over twenty-five years, Joe has been providing training, contracting, and consulting services for the software industry. Joe recently published the book, Developing User Assistance for Mobile Apps. He also teaches courses for Bellevue College, the University of California, and the University of Washington. Joe is an Associate Fellow of STC.</P>"
 },
}
```

```
"id": "17920884849",
"url": "https://www.eventbrite.com/e/an-aggressive-approach-to-concise-writing-with-joe-welinske-tickets-17920884849",
"start": {
 "timezone": "America/Los_Angeles",
 "local": "2015-09-24T12:00:00",
 "utc": "2015-09-24T19:00:00Z"
},
"end": {
 "timezone": "America/Los_Angeles",
 "local": "2015-09-24T13:00:00",
 "utc": "2015-09-24T20:00:00Z"
},
"created": "2015-07-27T15:14:49Z",
"changed": "2015-09-25T03:38:00Z",
"capacity": 24,
"capacity_is_custom": false,
"status": "completed",
"currency": "USD",
"listed": true,
"shareable": true,
"online_event": false,
"tx_time_limit": 480,
"hide_start_date": false,
"hide_end_date": false,
"locale": "en_US",
"is_locked": false,
"privacy_setting": "unlocked",
"is_series": false,
"is_series_parent": false,
"is_reserved_seating": false,
"source": "create_2.0",
"is_free": true,
"version": "3.0.0",
"logo_id": null,
"organizer_id": "7774592843",
"venue_id": "11047889",
"category_id": "102",
"subcategory_id": "2004",
"format_id": "2",
"resource_uri": "https://www.eventbriteapi.com/v3/events/17920884849/",
"logo": null
}
```

## 5. Pull out the information you need

The information has a lot more than we need. We just want to display the event's title and description on our site. To do this, we use some simple jQuery code to pull out the information and append it to a tag on our web page:

```
<html>
<body>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

<script>
 var settings = {
 "async": true,
 "crossDomain": true,
 "url": "https://www.eventbriteapi.com/v3/events/17920884849/?token=APIKEY",
 "method": "GET",
 "headers": {}
 }

 $.ajax(settings).done(function (data) {
 console.log(data);
 var content = "<h2>" + data.name.text + "</h2>" + data.description.html;
 $("#eventbrite").append(content);
 });
</script>

<div id="eventbrite"></div>

</body>
</html>
```

We covered this approach earlier in the course, so I won't go into much detail here.

My API key is hidden from the above code sample to protect it from unauthorized access.

Here's the [result](#):

**An Aggressive Approach to Concise Writing, with Joe Welinske**

**Webinar Description**

Writing concisely is one of the fundamental skills central to any mobile user assistance. The minimal screen real estate can't support large amounts of text and graphics without extensive gesturing by the users. Using small font sizes just makes the information unreadable unless the user pinches and stretches the text.

Even outside of the mobile space, your ability to streamline your content improves the likelihood it will be effectively consumed by your target audience.

This session offers a number of examples and techniques for reducing the footprint of your prose while maintaining a quality message. The examples used are in the context of mobile UA but can be applied to any technical writing situation.

**About Joe Welinske**

Joe Welinske specializes in helping your software development effort through crafted communication. The best user experience features quality words and images in the user interface. The UX of a robust product is also enhanced through comprehensive user assistance. This includes Help, wizards, FAQs, videos and much more. For over twenty-five years, Joe has been providing training, contracting, and consulting services for the software industry. Joe recently published the book, *Developing User Assistance for Mobile Apps*. He also teaches courses for Bellevue College, the University of California, and the University of Washington. Joe is an Associate Fellow of STC.

## Code explanation

The sample implementation is as plain as it can be in terms of style. But with API documentation code examples, you want to keep code examples simple. In fact, you most likely don't need a demo at all. Simply showing the payload returned in the browser is sufficient for a UI developer. However, for testing it's fun to make content actually appear on the page.

The `ajax` method from jQuery gets a payload for an endpoint URL, and then assigns it to the `data` argument. We log `data` to the console to more easily inspect its payload. To pull out the various properties of the object, we use dot notation. `data.name.text` gets the text property from the name object that is embedded inside the data object.

We then rename the content we want with a variable (`var content`) and use jQuery's `append` method to assign it to a specific tag (`eventbrite`) on the page.

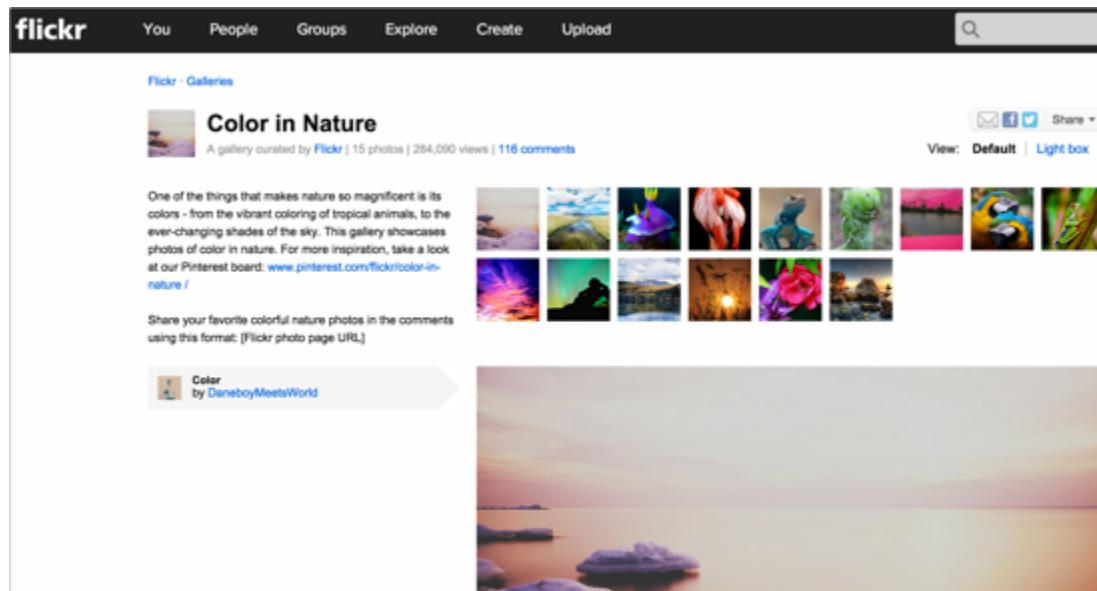
# Flickr example: Retrieve a Flickr gallery

Use the Flickr API to get photo images from [this Flickr gallery](#).

## Flickr Overview

In this Flickr API example, you'll see that our goal requires us to call several endpoints. You'll see that just having an API reference that lists the endpoints and responses isn't enough. Often one endpoint requires other endpoint responses as inputs, and so on.

In this example, we want to get all the photos from a [specific Flickr gallery](#) and display them on a web page. Here's the gallery we want:



## 1. Get an API key to make requests

Before you can make a request with the Flickr API, you'll need an API key, which you can read more about [here](#). When you register an app, you're given a key and secret.

## 2. Determine the resource and endpoint you need

From the list of [Flickr's API methods](#), the `flickr.galleries.getPhotos` endpoint, which is listed under the galleries resource, is the one that will get photos from a gallery.

The screenshot shows the Flickr API documentation for the `flickr.galleries.getPhotos` endpoint. The page has a dark header with the Flickr logo, a "Sign Up" button, and links for "Explore" and "Create". Below the header, the endpoint name is displayed in large bold letters. A description follows: "Return the list of photos for a gallery". The "Authentication" section states that no authentication is required. The "Arguments" section lists several parameters:

- api\_key** (Required): Your API application key. [See here](#) for more details.
- gallery\_id** (Required): The ID of the gallery of photos to return.
- extras** (Optional): A comma-delimited list of extra information to fetch for each returned record. Currently supported values include `date_upload`, `date_taken`, `owner_name`, `icon_server`, `original_format`, `last_update`, `views`, `media`, `path_alias`, `url_sq`, `url_t`, `url_s`, `url_q`, `url_m`, `url_n`, `url_z`, and `url_c`.
- per\_page** (Optional): Number of photos to return per page. If this argument is omitted, it defaults to 100. The maximum value is 500.
- page** (Optional): Page number to return.

One of the arguments we need for the `getPhotos` endpoint is the gallery ID. Before we can get the gallery ID, however, we have to use another endpoint to retrieve it. *Rather unintuitively, the gallery ID is not the ID that appears in the URL of the gallery.*

We use the [flickr.urls.lookupGallery](#) endpoint listed in the URLs resource section to get the gallery ID from a gallery URL:

The screenshot shows the Flickr API Explorer interface. At the top, there's a navigation bar with the Flickr logo, a 'Sign Up' button, and 'Explore' and 'Create' links. 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.urls.lookupGallery'. It features a table for 'Arguments' with one row for 'url' (Required, Send checked, Value input field). Below the table, there's a dropdown for 'Output' set to 'XML (REST)'. Two radio buttons are present: 'Sign call with no user token?' (selected) and 'Do not sign call?'. A 'Call Method...' button is at the bottom. A link 'Back to the flickr.urls.lookupGallery documentation' is also visible.

The gallery ID is [66911286-72157647277042064](#). We now have the arguments we need for the `flickr.galleries.getPhotos` endpoint.

### 3. Construct the request

We can make the request to get the list of photos for this specific gallery ID.

Flickr provides an API Explorer to simplify calls to the endpoints. If we go to the [API Explorer for the galleries.getPhotos endpoint](#), we can plug in the gallery ID and see the response, as well as get the URL syntax for the endpoint.

**The App Garden**

Create an App | API Documentation | Feeds | What is the App Garden?

## flickr.galleries.getPhotos

**Arguments**

Name	Required	Send	Value
gallery_id	required	<input checked="" type="checkbox"/>	66911286-7215764727
extras	optional	<input type="checkbox"/>	
per_page	optional	<input type="checkbox"/>	
page	optional	<input type="checkbox"/>	

Output: **JSON**

Sign call as tomhenryjohnson with full permissions?   
 Sign call with no user token?   
 Do not sign call?

**Useful Values**

Your user ID:  
86824645@N00

Your recent photo IDs:  
19271714336 -  
19111657009 -  
19110196618 -

Your recent photoset IDs:  
72157651486110150 - Auto U  
72157649733910233 - hover  
72157649565389074 - Disco

Your recent group IDs:

Your contact IDs:  
30744708@N00 - katiew  
92673622@N00 - dulcelife

[Call Method...](#)

[Back to the flickr.galleries.getPhotos documentation](#)

Insert the gallery ID, select **Do not sign call** (we're just testing here, so we don't need extra security), and then click **Call Method**.

Here's the result:

```
{
 "photos": {
 "page": 1,
 "pages": 1,
 "perpage": "500",
 "total": 15,
 "photo": [
 {
 "id": "8432423659",
 "owner": "37107167@N07",
 "secret": "dd1b834ec5",
 "server": "8187",
 "farm": 1
 },
 {
 "id": "8047948330",
 "owner": "70121902@N00",
 "secret": "b0e55d45f",
 "server": "8450",
 "farm": 1
 },
 {
 "id": "2209143676",
 "owner": "14478436@N02",
 "secret": "ae987333b5",
 "server": "2072",
 "farm": 1
 },
 {
 "id": "399296912",
 "owner": "58329132@N07",
 "secret": "6adcc29651",
 "server": "161",
 "farm": 1
 },
 {
 "id": "5812344633",
 "owner": "47690289@N02",
 "secret": "af53e53bf1",
 "server": "3277",
 "farm": 1
 },
 {
 "id": "4960822520",
 "owner": "48600090482@N01",
 "secret": "d3094b0d5",
 "server": "4090",
 "farm": 1
 },
 {
 "id": "3460002981",
 "owner": "37357417@N07",
 "secret": "9121bb0695",
 "server": "3609",
 "farm": 1
 },
 {
 "id": "3033898918",
 "owner": "44115070@N00",
 "secret": "33238aca22",
 "server": "3036",
 "farm": 1
 },
 {
 "id": "9437404307",
 "owner": "70140013@N07",
 "secret": "293b54b7d5",
 "server": "5494",
 "farm": 1
 },
 {
 "id": "8948867145",
 "owner": "91805169@N04",
 "secret": "34930c7865",
 "server": "2880",
 "farm": 1
 },
 {
 "id": "13687274945",
 "owner": "28145073@N08",
 "secret": "5102c35ca9",
 "server": "2912",
 "farm": 1
 },
 {
 "id": "13892714966",
 "owner": "36587311@N08",
 "secret": "ae06a2ee97",
 "server": "7460",
 "farm": 1
 },
 {
 "id": "9422871791",
 "owner": "52846362@N04",
 "secret": "db45e0b7ed",
 "server": "3754",
 "farm": 1
 },
 {
 "id": "14412870627",
 "owner": "85737574@N02",
 "secret": "5a469dda2a",
 "server": "3896",
 "farm": 1
 },
 {
 "id": "6231102554",
 "owner": "53760536@N07",
 "secret": "966a8675c9",
 "server": "6218",
 "farm": 1
 }
],
 "stat": "ok"
 }
}
```

URL: [https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api\\_key=acada0ff2bb2747f40e0b227675360c9&gallery\\_id=66911286-72157647277042064&format=json&nojsoncallback=1](https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api_key=acada0ff2bb2747f40e0b227675360c9&gallery_id=66911286-72157647277042064&format=json&nojsoncallback=1)

The URL below the response shows the right syntax for using this method:

```
https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api_key=APIKEY&gallery_id=66911286-72157647277042064&format=json&nojsoncallback=1
```

I have removed my API key from code samples to prevent possible abuse to my API keys. If you're following along, swap in your own API key here.

If you submit the request direct in your browser using the given URL, you can see the same response but in the browser rather than the API Explorer:

```
{
 "photos": {
 "page": 1,
 "pages": 1,
 "perpage": 500,
 "total": 15,
 "photo": [
 {
 "id": "8432423659",
 "owner": "37107167@N07",
 "secret": "dd1b834ec5",
 "server": "8187",
 "farm": 9,
 "title": "Color",
 "ispublic": 1,
 "isfriend": 0,
 "isfamily": 0,
 "is_primary": 1,
 "has_comment": 0
 },
 {
 "id": "8047948330",
 "owner": "70121902@N00",
 "secret": "b0e55d455f",
 "server": "8450",
 "farm": 9,
 "title": "Owens River and Sea Grass",
 "ispublic": 1,
 "isfriend": 0,
 "isfamily": 0
 }
]
 }
}
```

I'm using the [JSON Formatting extension for Chrome](#) to make the JSON response more readable. Without this plugin, the JSON response is compressed.

## 4. Analyze the response

All the necessary information is included in this response in order to display photos on our site, but it's not entirely intuitive how we construct the image source URLs from the response.

Note that the information a user needs to actually achieve a goal isn't explicit in the API *reference* documentation. All the reference doc explains is what gets returned in the response, not how to actually use the response.

The [Photo Source URLs](#) page in the documentation explains it:

You can construct the source URL to a photo once you know its ID, server ID, farm ID and secret, as returned by many API methods. The URL takes the following format:

```
https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}.jpg
or
https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}_{mstz}
b].jpg
or
https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{o-secre
t}_o.(jpg|gif|png)
```

Here's what an item in the JSON response looks like:

```
"photos": {
 "page": 1,
 "pages": 1,
 "perpage": 500,
 "total": 15,
 "photo": [
 {
 "id": "8432423659",
 "owner": "37107167@N07",
 "secret": "dd1b834ec5",
 "server": "8187",
 "farm": 9,
 "title": "Color",
 "ispublic": 1,
 "isfriend": 0,
 "isfamily": 0,
 "is_primary": 1,
 "has_comment": 0
 } ...
```

You access these fields through dot notation. It's a good idea to log the whole object to the console just to explore it better.

## 5. Pull out the information you need

The following code uses jQuery to loop through each of the responses and inserts the necessary components into an image tag to display each photo. Usually in documentation you don't need to be so explicit about how to use a common language like jQuery. You assume that the developer is capable in a specific programming language.

```
<html>
<style>
img {max-height:125px; margin:3px; border:1px solid #dedede;}
</style>
<body>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

<script>

var settings = {
 "async": true,
 "crossDomain": true,
 "url": "https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api_key=APIKEY&gallery_id=66911286-72157647277042064&format=json&nojsoncallback=1",
 "method": "GET",
 "headers": {}
}

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

 $("#galleryTitle").append(data.photos.photo[0].title + " Gallery");
 $.each(data.photos.photo, function(i, gp) {

 var farmId = gp.farm;
 var serverId = gp.server;
 var id = gp.id;
 var secret = gp.secret;

 console.log(farmId + ", " + serverId + ", " + id + ", " + secret);

 // https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}.jpg

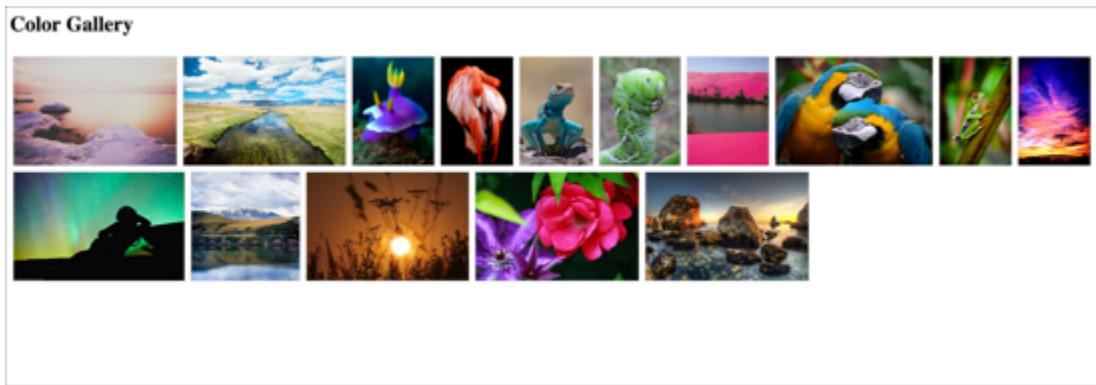
 $("#flickr").append('');
 });
});

</script>

<h2><div id="galleryTitle"></div></h2>
<div style="clear:both;">
<div id="flickr"/>
```

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

And the result looks like this:



## Code explanation

Note that this code uses JavaScript logic that is usually beyond the need to include in documentation. However, if it was a common scenario to embed a gallery of images on a web page, this kind of code and explanation would be helpful.

- In this code, the `ajax method` from jQuery gets the JSON payload. The payload is assigned to the `data` argument and then logged to the console.
- The data object contains an object called `photos`, which contains an array called `photo`. The `title` field is a property in an object in the `photo` array. The `title` is accessed through this dot notation: `data.photos.photo[0].title`.
- To get each item in the object, jQuery's `each method` loops through an object's properties. Note that jQuery `each` method is commonly used for looping through results to get values. Here's how it works. For the first argument (`data.photos.photo`), you identify the object that you want to access.
- For the `function( i, gp )` arguments, you list an index and value. You can use any names you want here. `gp` becomes a variable that refers to the `data.photos.photo` object you're looping through. `i` refers to the starting point through the object. (You don't actually need to refer to `i` beyond the mention here unless you want to begin or end the loop at a certain point.)
- To access the properties in the JSON object, we use `gp.farm` instead of `data.photos.photo[0].farm`, because `gp` is an object reference to `data.photos.photo[i]`.
- After the `each` function iterates through the response, I added some variables to make it easier to work with these components (using `serverId` instead of `gp.server`, etc.). And a `console.log` message checks to ensure we're getting values for each of the elements we need.
- This comment shows where we need to plug in each of the variables:

```
// https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{secret}.jpg
```

The final line shows how you insert those variables into the HTML:

```
$("#flickr").append('');
```

A common pattern in programming is to loop through a response. This code example used the `each` method from jQuery to look through all the items in the response and do something with each item. Sometimes you incorporate logic that loops through items and looks for certain conditions present to decide whether to take some action. Pay attention to methods for looping, as they are common scenarios in programming.

# Klout example: Retrieve Klout influencers

Use the Klout API to get your Klout score and a list of your influencers and influencees.

## About Klout

Klout is a service that gauges your online influence (your klout) by measuring tweets, retweets, likes, etc. from a variety of social networks using a sophisticated algorithm. In this tutorial, you'll use the Klout API to retrieve a Klout score for a particular Twitter handle, and then a list of your influencers.

Klout has an “interactive console” driven by Mashery I/O docs that allows you to insert parameters and go to an endpoint. The interactive console also contains brief descriptions of what each endpoint does.

The screenshot shows the Klout Interactive Console interface. At the top, it says "Interactive Console" and "Signed in as Tom Johnson". On the left sidebar, there are links for "INTERACTIVE CONSOLE", "TERMS OF SERVICE", "CODE LIBRARY", "DOCS", "- VERSION 2", and "REGISTER AN APP". The main content area has a dropdown menu set to "Partner API v2". A note says "All calls require you to log in or provide an API key." Below that, there's a field for "App/Key" containing "I'd rather be writing: urgey4a79n5x6df6xx4p64dr" with a link to "Manually provide key information". There are also links to "Toggle All Endpoints" and "Toggle All Methods". The central part of the screen is titled "Identity Methods" with "List Methods" and "Expand Methods" buttons. It lists four GET methods: "Identity(twitter\_id)", "Identity(Google+)", "Identity(Instagram)", and "Identity(twitter\_screen\_name)". The "Identity(twitter\_screen\_name)" method is expanded, showing its description: "This method allows you to retrieve a KloutID for a twitter screen\_name". Below this, there's a table with columns "Parameter", "Value", "Type", and "Description". The "screenName" parameter is set to "tomjohnson" of type "string" with the description "A twitter screen name (e.g. jtimmerlake)".

## 1. Get an API key to make requests

To use the API, you have to register an “app,” which allows you to get an API key. Go to [My API Keys](#) page to register your app and get the keys.

## 2. Make requests for the resources you need

The API is relatively simple and easy to browse.

To get your Klout score, you need to use the `score` endpoint. This endpoint requires you to pass your Klout ID.

Since you most likely don't know your Klout ID, use the `identity(twitter_screen_name)` endpoint first.

The screenshot shows the Klout API console interface. At the top, a red button labeled "GET" is next to the endpoint "Identity(twitter\_screen\_name) /identity.json/twitter". Below this, a table provides details for the "screenName" parameter:

Parameter	Value	Type	Description
screenName	tomjohnson	string	A twitter screen name (e.g. jtimmerlake)

Below the table are two buttons: "Try it!" and "Clear Results".

The "Request URI" section contains the URL: `http://api.klout.com/v2/identity.json/twitter?screenName=tomjohnson&key=u4r7nd3r7bj9ksxfx3cuy6hw`.

The "Request Headers" section shows the header: `X-Originating-Ip: 24.23.183.203`.

The "Response Status" section shows the status: `200 OK`.

The "Response Headers" section displays the following headers:

```
Cf-Ray: 21edbfff82cc1ec5-SJC
Content-Type: application/json; charset=utf-8
Date: Tue, 01 Sep 2015 03:04:49 GMT
Server: cloudflare-nginx
X-Apikey-Qps-Allotted: 10
X-Apikey-Qps-Current: 1
X-Apikey-Quota-Allotted: 20000
X-Apikey-Quota-Current: 1
X-Mashery-Responder: prod-j-worker-us-west-1b-26.mashery.com
X-Quota-Reset: Wednesday, September 2, 2015 12:00:00 AM GMT
Transfer-Encoding: chunked
Connection: keep-alive
```

The "Response Body" section shows the JSON response:

```
{
 "id": "1134760",
 "network": "ks"
}
```

Instead of using the API console, you can also submit the request via your browser by going to the request URL:

```
http://api.klout.com/v2/identity.json/twitter?screenName=tomjohnson&key={your api key}
```

In place of `{your api key}`, insert your own API key. (I initially displayed mine here only to find that bots grabbed it and made thousands of requests, which ended up disabling my API key.)"

My Klout ID is `1134760`.

Now you can use the `score` endpoint to calculate your score.

**GET Score /user.json/kloutId/score**

This method allows you to retrieve a user's Klout Score and deltas.

Parameter	Value	Type	Description
kloutId	1134760	string	A kloutId (like 635263)

**Try it! Clear Results**

**Request URI**  
`http://api.klout.com/v2/user.json/1134760/score?key=u4r7nd3r7bj9ksxfx3cuy6hw`

**Request Headers Select content**  
`X-Originating-Ip: 24.23.183.203`

**Response Status Select content**  
`200 OK`

**Response Headers Select content**  
`Cf-Ray: 21edcb54499e1ebf-SJC  
Content-Type: application/json; charset=UTF-8  
Date: Tue, 01 Sep 2015 03:12:33 GMT  
Server: cloudflare-nginx  
X-Apikey-Qps-Allotted: 10  
X-Apikey-Qps-Current: 1  
X-Apikey-Quota-Allotted: 20000  
X-Apikey-Quota-Current: 2  
X-Mashery-Responder: prod-j-worker-us-west-1c-42.mashery.com  
X-Quota-Reset: Wednesday, September 2, 2015 12:00:00 AM GMT  
Content-Length: 159  
Connection: keep-alive`

**Response Body Select content**  
`{  
 "score": 54.233149646009174,  
 "scoreDelta": {  
 "dayChange": -0.5767549117977069,  
 "weekChange": -0.5311640476663939,  
 "monthChange": -0.2578449396243201  
 },  
 "bucket": "50-59"  
}`

My score is **54**. Klout's interactive console makes it easy to get responses for API calls, but you could equally submit the request URI in your browser.

```
http://api.klout.com/v2/user.json/1134760/score?key={your api key}
```

After submitting the request, here is what you would see:

```
{
 "score": 54.233149646009174,
 "scoreDelta": {
 "dayChange": -0.5767549117977069,
 "weekChange": -0.5311640476663939,
 "monthChange": -0.2578449396243201
 },
 "bucket": "50-59"
}
```

Now suppose you want to know who you have influenced (your influencees) and who influences you (your influencers). After all, this is what Klout is all about. Influence is measured by the action you drive.

To get your influencers and influencees, you need to use the `influence` endpoint, passing in your Klout ID.

### 3. Analyze the response

And here's the influence resource's response:

```
{
 "myInfluencers": [
 {"entity": {
 "id": "441634251566461018",
 "payload": {
 "kloutId": "441634251566461018",
 "nick": "jekyllrb",
 "score": {
 "score": 50.41206120210041,
 "bucket": "50-59"
 },
 "scoreDeltas": {
 "dayChange": -0.05927708546307997,
 "weekChange": -0.739829931907181,
 "monthChange": -0.7917151139830239
 }
 }
 }
],
 {"entity": {
 "id": "33214052017370475",
 "payload": {
 "kloutId": "33214052017370475",
 "nick": "Mrtnlrssn",
 "score": {
 "score": 22.45014953758632,
 "bucket": "20-29"
 },
 "scoreDeltas": {
 "dayChange": -0.3481056157609004,
 "weekChange": -2.132213372307284,
 "monthChange": -2.315034722843535
 }
 }
 },
 {"entity": {
 "id": "177892199475207065",
 "payload": {
 "kloutId": "177892199475207065",
 "nick": "TCSpeakers",
 "score": {
 "score": 28.23034124231384,
 "bucket": "20-29"
 },
 "scoreDeltas": {
 "dayChange": 0.00154327588529668,
 "weekChange": -0.6416866188503434,
 "monthChange": -4.226666088333872
 }
 }
 }
}
```

```
 }
 },
 {
 "entity": {
 "id": "91760850663150797",
 "payload": {
 "kloutId": "91760850663150797",
 "nick": "JohnFoderaro",
 "score": {
 "score": 39.39045702175103,
 "bucket": "30-39"
 },
 "scoreDeltas": {
 "dayChange": -0.6092388403641991,
 "weekChange": -0.699356032047298,
 "monthChange": 5.34513233077341
 }
 }
 }
 },
 {
 "entity": {
 "id": "1057244",
 "payload": {
 "kloutId": "1057244",
 "nick": "peterlalonde",
 "score": {
 "score": 42.39625419500191,
 "bucket": "40-49"
 },
 "scoreDeltas": {
 "dayChange": -0.32068173129262334,
 "weekChange": 0.14276611846587173,
 "monthChange": -0.9354253686809457
 }
 }
 }
],
 "myInfluencees": [
 {
 "entity": {
 "id": "537311",
 "payload": {
 "kloutId": "537311",
 "nick": "techwritertoday",
 "score": {
 "score": 49.99313854987996,
 "bucket": "40-49"
 },
 "scoreDeltas": {
 "dayChange": -0.10510042996928348,
 "weekChange": -0.568647896457648,
 "monthChange": 0.3425617785475197
 }
 }
 }
 }
]
}
```

```
 }
 }
}, {
 "entity": {
 "id": "91760850663150797",
 "payload": {
 "kloutId": "91760850663150797",
 "nick": "JohnFoderaro",
 "score": {
 "score": 39.39045702175103,
 "bucket": "30-39"
 },
 "scoreDeltas": {
 "dayChange": -0.6092388403641991,
 "weekChange": -0.699356032047298,
 "monthChange": 5.34513233077341
 }
 }
 }
}, {
 "entity": {
 "id": "33214052017370475",
 "payload": {
 "kloutId": "33214052017370475",
 "nick": "Mrtnlrssn",
 "score": {
 "score": 22.45014953758632,
 "bucket": "20-29"
 },
 "scoreDeltas": {
 "dayChange": -0.3481056157609004,
 "weekChange": -2.132213372307284,
 "monthChange": -2.315034722843535
 }
 }
 }
}, {
 "entity": {
 "id": "45598950992256021",
 "payload": {
 "kloutId": "45598950992256021",
 "nick": "DavidEgyes",
 "score": {
 "score": 40.40572793362214,
 "bucket": "40-49"
 },
 "scoreDeltas": {
 "dayChange": 0.001934309078080787,
 "weekChange": 2.233816485488269,
 "monthChange": 1.4901401977594801
 }
 }
 }
}
```

```
 }
 },
], {
 "entity": {
 "id": "46724857496656136",
 "payload": {
 "kloutId": "46724857496656136",
 "nick": "fabi_ator",
 "score": {
 "score": 30.32498605174672,
 "bucket": "30-39"
 },
 "scoreDeltas": {
 "dayChange": -0.005890177199574964,
 "weekChange": -0.6859163242901047,
 "monthChange": -5.293301673692355
 }
 }
 },
 "myInfluencersCount": 5,
 "myInfluenceesCount": 5
}
```

The response contains an array containing 5 influencers and an array containing 5 influencees. (Remember the square brackets denote an array; the curly braces denote an object. Each array contains a list of objects.)

## 4. Pull out the information you need

Suppose you just want a short list of Twitter names with their links.

Using jQuery, you can iterate through the JSON payload and pull out the information that you want:

```
<html>
<body>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

<script>
var settings = {
 "async": true,
 "crossDomain": true,
 "url": "http://api.klout.com/v2/user.json/1134760/influence?key=APIKEY&callback=?",
 "method": "GET",
 "dataType": "jsonp",
 "headers": {}
}

$.ajax(settings).done(function (data) {
 console.log(data);
 $.each(data.myInfluencees, function(i, inf) {
 $("#kloutinfluencees").append('' + inf.entity.payload.nick + '');
 });
 $.each(data.myInfluencers, function(i, inf) {
 $("#kloutinfluencers").append('' + inf.entity.payload.nick + '');
 });
});
</script>

<h2>My influencees (people I influence)</h2>
<ul id="kloutinfluencees">

<h2>My influencers (people who influence me)</h2>
<ul id="kloutinfluencers">

</body>
</html>
```

Remember to swap in your own API key in place of `APIKEY`. The result looks like this:

The screenshot shows a local file in a browser window titled "klout.html". The page content is as follows:

## My influencees (people I influence)

- [techwritertoday](#)
- [JohnFoderaro](#)
- [Mrtnlrssn](#)
- [DavidEgyes](#)
- [fabiator](#)

## My influencers (people who influence me)

- [jekyllrb](#)
- [Mrtnlrssn](#)
- [TCSpeakers](#)
- [JohnFoderaro](#)
- [peterlalonde](#)

## Code explanation

The code uses the `ajax` method from jQuery to get a JSON payload for a specific URL. It assigns this payload to the `data` argument. The `console.log(data)` code just logs the payload to the console to make it easy to inspect.

The jQuery `each` method iterates through each property in the `data.myInfluencees` object. It renames this object `inf` (you can choose whatever names you want) and then gets the `entity.payload.nick` property (nickname) for each item in the object. It inserts this value into a link to the Twitter profile, and then appends the information to a specific tag on the page (`#kloutinfluencers`).

Pretty much the same approach is used for the `data.myInfluencers` object, but the tag the data is appended to is (`#kloutinfluencers`).

Note that in the `ajax` settings, a new attribute is included: `"dataType": "jsonp"`. If you omit this, you'll get an error message that says:

```
XMLHttpRequest cannot load http://api.klout.com/v2/user.json/876597/influenc
e?key=APIKEY&callback=?.. No 'Access-Control-Allow-Origin' header is present
on the requested resource. Origin 'null' is therefore not allowed access.
```

When you submit requests to endpoints, you're getting information from other domains and pulling the information to your own domain. For security purposes, servers block this action. The resource server has to enable something called Cross Origin Resource Sharing (CORS).

JSONP gets around CORS restricts by wrapping the JSON into script tags, which servers don't block. With JSONP, you can only use GET methods. You can [read more about JSONP here](#).

# Aeris Weather Example: Get wind speed

Use the Aeris Weather API to get the wind speed (MPH) for a specific place (your choice).

## The Aeris Weather API

Since you've been working with the weather API on OpenWeatherMap, it's probably a good idea to compare this simple weather API with another one. Check out the [Aeris Weather API here](#). This is one of the most interesting, well-documented and powerful weather APIs I've encountered.

In this example, you'll get the wind in MPH and then set an answer to display on the page based on some conditional logic.

### 1. Get the API keys

See the [Getting Started](#) page for information on how to register and get API keys. (Obviously, get the free version of the keys available to development projects.) You will need both the CLIENTID and CLIENTSECRET to make API calls.

### 2. Construct the request

Browse through the [available endpoints](#) and look for one that would give you the wind speed. The `observations` resource provides information about wind speed, as does `forecasts`. The response from `observations` looks a little simpler, so let's use that endpoint.

The screenshot shows the Aeris Weather API documentation for the 'observations' endpoint. The left sidebar has a blue header 'Aeris Weather API' with sections for 'Getting Started', 'Reference', and 'Downloads'. The main content area has a title 'Endpoint: observations'. It describes the observations data set as providing access to current and archived weather observations from a variety of stations. It mentions METAR reports are generated once an hour, and other sources like PWS. Below this is a 'Data Coverage' section labeled 'Global'. A 'Included With' section lists 'API Developer', 'API Basic', and 'API Premium'. At the bottom of the main content are navigation links for 'ACTIONS', 'PARAMETERS', 'FILTERS', 'QUERIES', 'SORTING', 'EXAMPLES', 'RESPONSE', and 'PROPERTIES'. The top right of the page has a 'Search docs' bar.

To get the forecast details for Santa Clara, California, add it after `/observations`, like this:

```
http://api.aerisapi.com/observations/santa%20clara,ca?client_id=CLIENT_ID&client_secret=CLIENT_SECRET
```

### 3. Analyze the response

Here's the response from the request:

```
{
 "success": true,
 "error": null,
 "response": {
 "id": "KSJC",
 "loc": {
 "long": -121.91666666667,
 "lat": 37.366666666667
 },
 "place": {
 "name": "san jose",
 "state": "ca",
 "country": "us"
 },
 "profile": {
 "tz": "America/Los_Angeles",
 "elevM": 24,
 "elevFT": 79
 },
 "obTimestamp": 1441083180,
 "obDateTime": "2015-08-31T21:53:00-07:00",
 "ob": {
 "timestamp": 1441083180,
 "dateTimeISO": "2015-08-31T21:53:00-07:00",
 "tempC": 18,
 "tempF": 64,
 "dewpointC": 14,
 "dewpointF": 57,
 "humidity": 78,
 "pressureMB": 1012,
 "pressureIN": 29.88,
 "spressureMB": 1009,
 "spressureIN": 29.8,
 "altimeterMB": 1012,
 "altimeterIN": 29.88,
 "windKTS": 5,
 "windKPH": 9,
 "windMPH": 6,
 "windSpeedKTS": 5,
 "windSpeedKPH": 9,
 "windSpeedMPH": 6,
 "windDirDEG": 300,
 "windDir": "WNW",
 "windGustKTS": null,
 "windGustKPH": null,
 "windGustMPH": null,
 "flightRule": "LIFR",
 "visibilityKM": 16.09344,
 "visibilityMI": 10,
 "weather": "Clear",
 "weatherShort": "Clear",
 }
 }
}
```

```
 "weatherCoded": "::CL",
 "weatherPrimary": "Clear",
 "weatherPrimaryCoded": "::CL",
 "cloudsCoded": "CL",
 "icon": "clearn.png",
 "heatindexC": 18,
 "heatindexF": 64,
 "windchillC": 18,
 "windchillF": 64,
 "feelslikeC": 18,
 "feelslikeF": 64,
 "isDay": false,
 "sunrise": 1441028278,
 "sunriseISO": "2015-08-31T06:37:58-07:00",
 "sunset": 1441075047,
 "sunsetISO": "2015-08-31T19:37:27-07:00",
 "snowDepthCM": null,
 "snowDepthIN": null,
 "precipMM": 0,
 "precipIN": 0,
 "solradWM2": null,
 "light": 0,
 "sky": 0
 },
 "raw": "KSJC 010453Z 30005KT 10SM CLR 18/14 A2989 RMK A02 SLP122 T018301
39",
 "relativeTo": {
 "lat": 37.35411,
 "long": -121.95524,
 "bearing": 68,
 "bearingENG": "ENE",
 "distanceKM": 3.684,
 "distanceMI": 2.289
 }
}
```

`windSpeedMPH` is the value we want.

## 4. Pull out the values from the response

To get the `windSpeedMPH`, you would access it through dot notation like this:

```
data.response.ob.windSpeedMPH .
```

To add a little variety to the code samples, here's one that's a bit different. We get the value for the `data.response.ob.windSpeedMPH` and assign the variable based on a condition. The variable then gets appended to the page. See if you can understand this code logic by following the if-else condition:

```
<html>
 <body>
 <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
 <script>

 jQuery.ajax({
 url: "http://api.aerisapi.com/observations/santa%20clara,ca",
 type: "GET",
 data: {
 "client_id": "CLIENTID",
 "client_secret": "CLIENTSECRET",
 },
 })
 .done(function(data, textStatus, jqXHR) {
 console.log("HTTP Request Succeeded: " + jqXHR.status);
 console.log(data);
 if (data.response.ob.windSpeedMPH > 15) {
 var windAnswer = "Yes, it's too windy.";
 }
 else {
 var windAnswer = "No, it's not that windy.";
 }
 $("#windAnswer").append(windAnswer)
 })
 .fail(function(jqXHR, textStatus, errorThrown) {
 console.log("HTTP Request Failed");
 })
 .always(function() {
 /* ... */
 });

 </script>
 <p>Is it too windy to go on a bike ride?</p>
 <div id="windAnswer" style="font-size:76px"></div>

 </body>
</html>
```

Here's the [result](#):

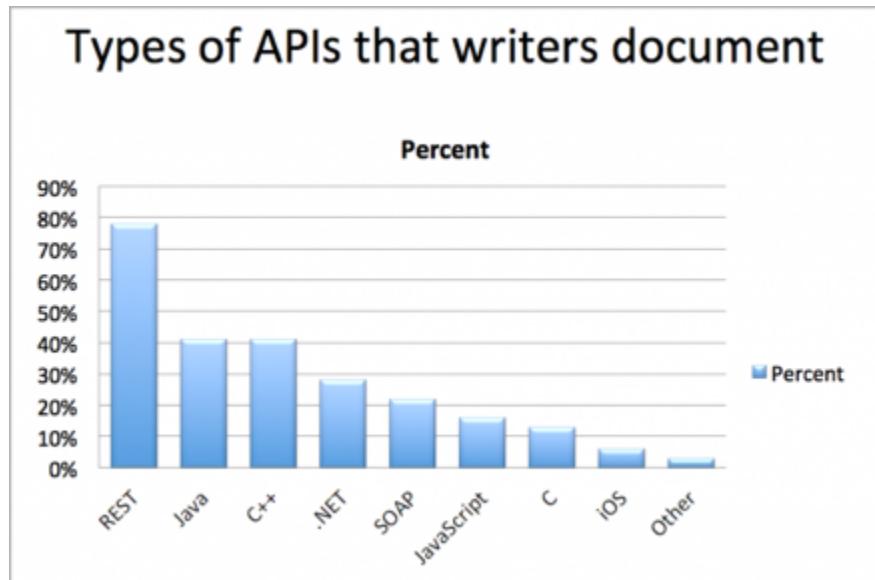


# API documentation survey overview

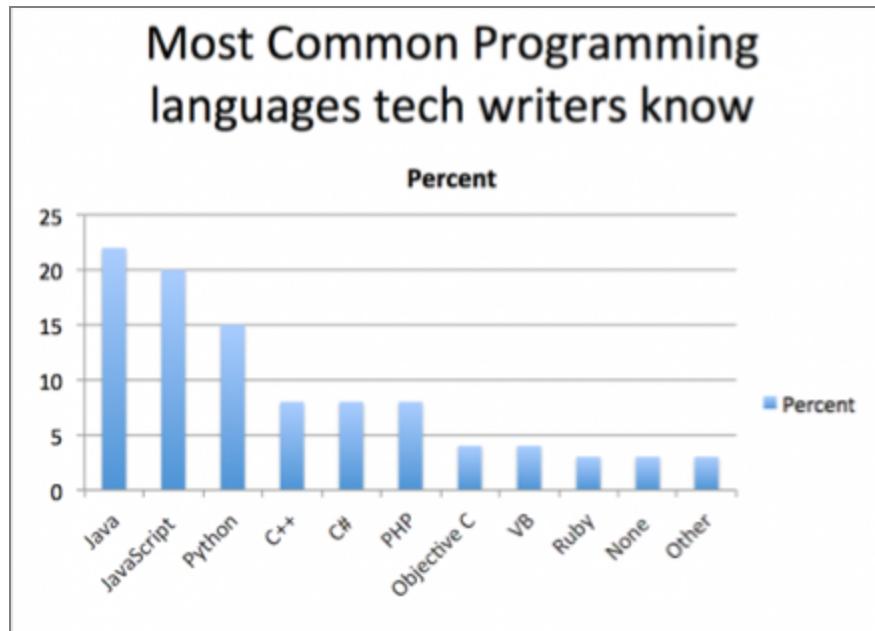
In December 2014, I [published an informal survey](#) to gather some basic information about API documentation practices. I used an open, short-answer style with 10 questions listed in a Google doc. About 43 people participated, though not everyone responded to every question.

If I were doing this survey again, I would clarify some of my questions. Still, the responses were valuable and painted a good picture of how API tech writers author and produce content.

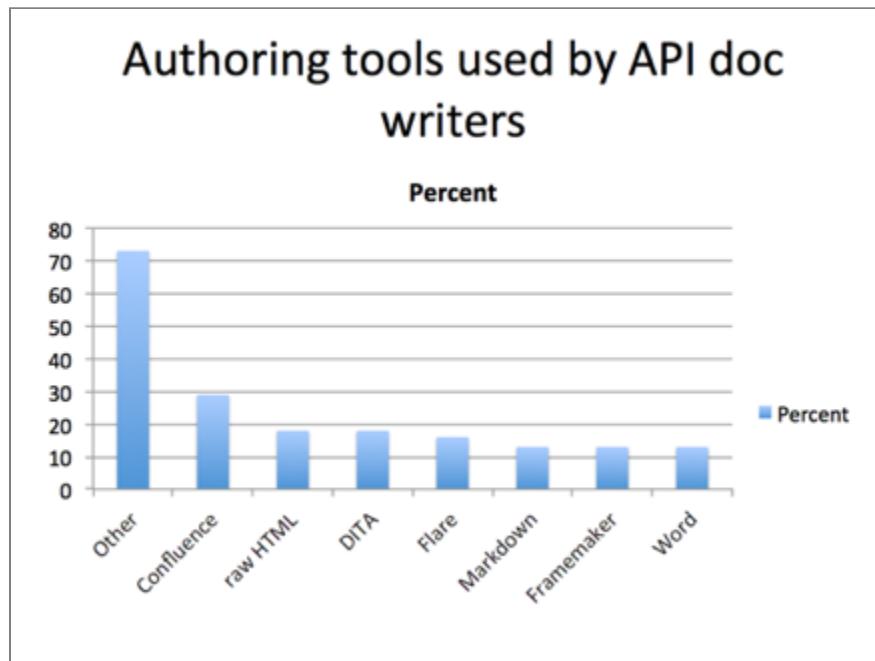
Here are the 10 questions. In a series of blog posts, I elaborated on the meaning of the responses at length. For brevity here, I've included only the screenshots of the responses.



[What types of APIs do you document?](#)

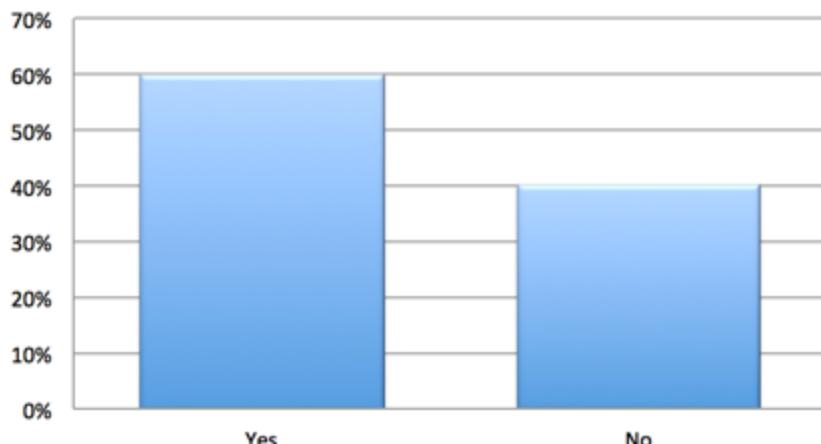


What programming languages do you know?



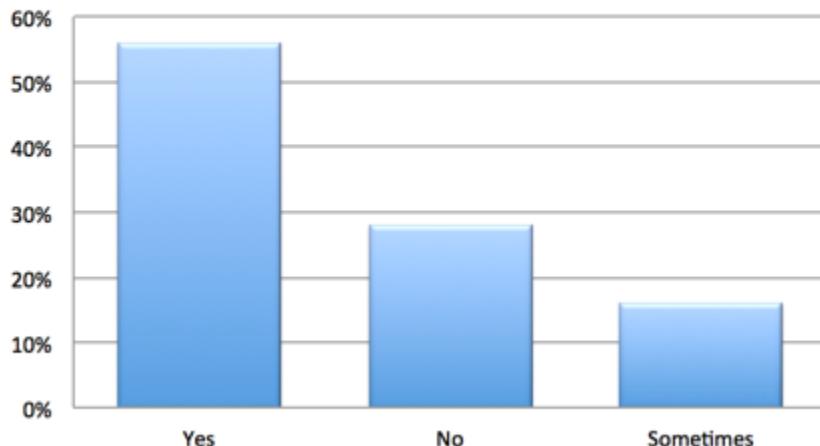
What programming languages do you know?

## Do you write doc by looking in the source code?



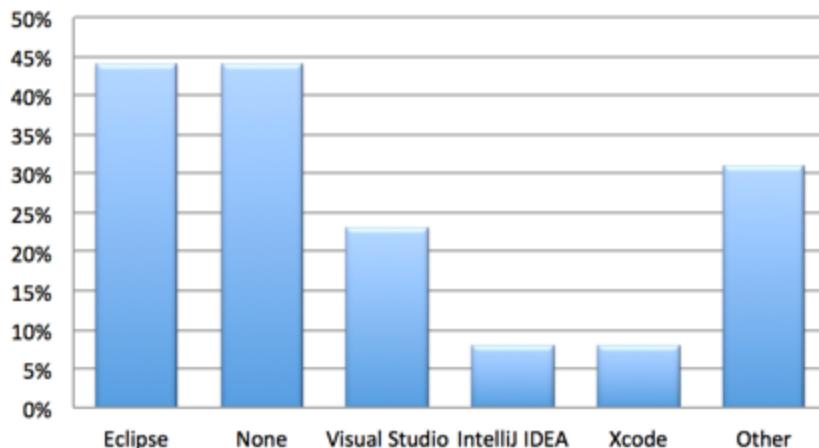
[Do you create API doc by looking at source code?](#)

## Do you test out the API calls used in your doc yourself?



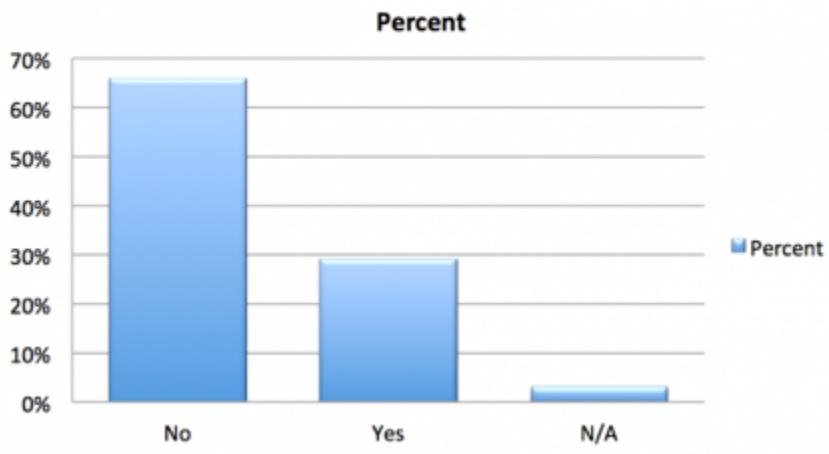
[Do you test out the API calls used in your doc yourself?](#)

## What IDE do you use?



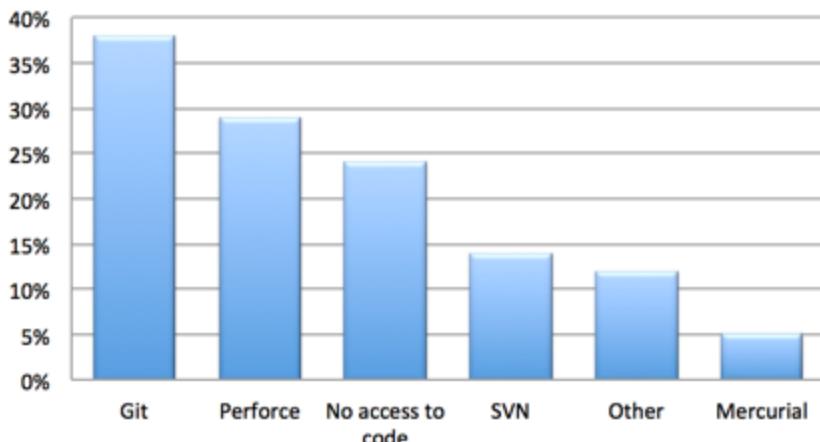
[What IDE do you use?](#)

## Are you automating REST API docs?



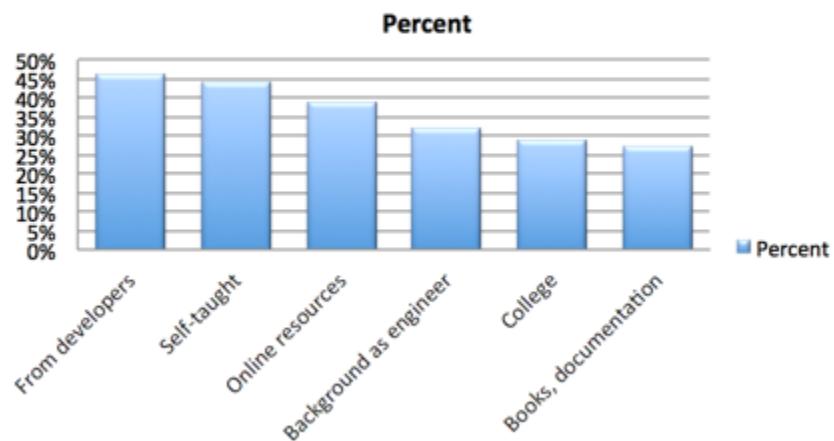
[Are API docs auto-rendered through build process?](#)

## How do you access the source code?



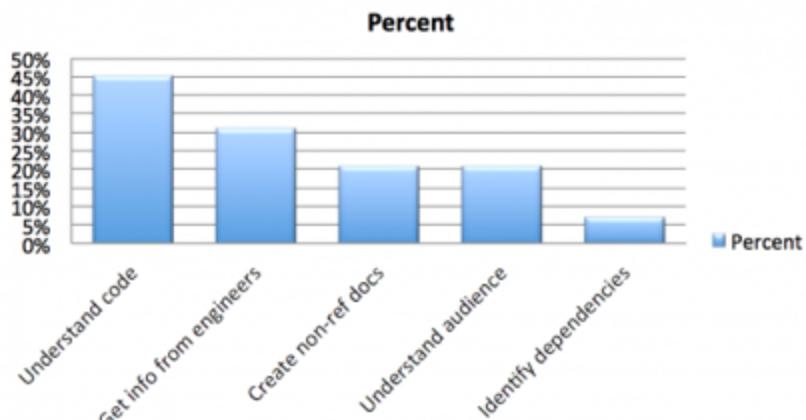
How do you get the source files that contain code comments?

## How did you learn what you needed to know?



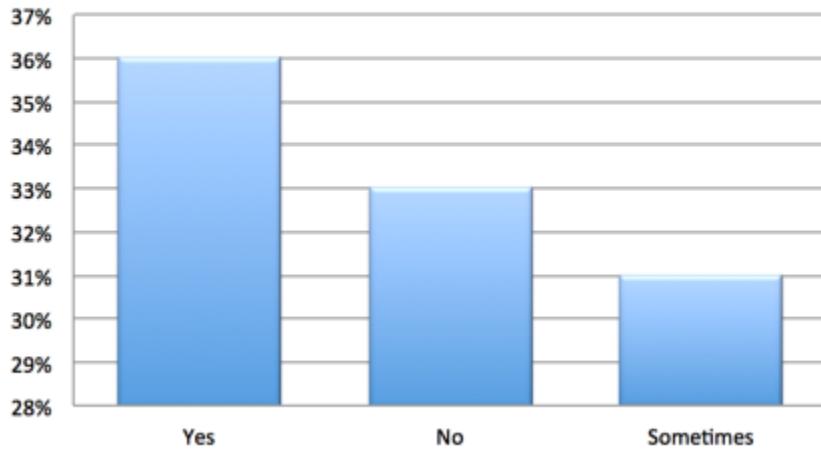
How to learn what you need to know?

## Most difficult part of API doc?



What aspect of API doc do you find most challenging?

## Do developers write the initial API documentation in the source code?



Do engineers write API doc in the source code?