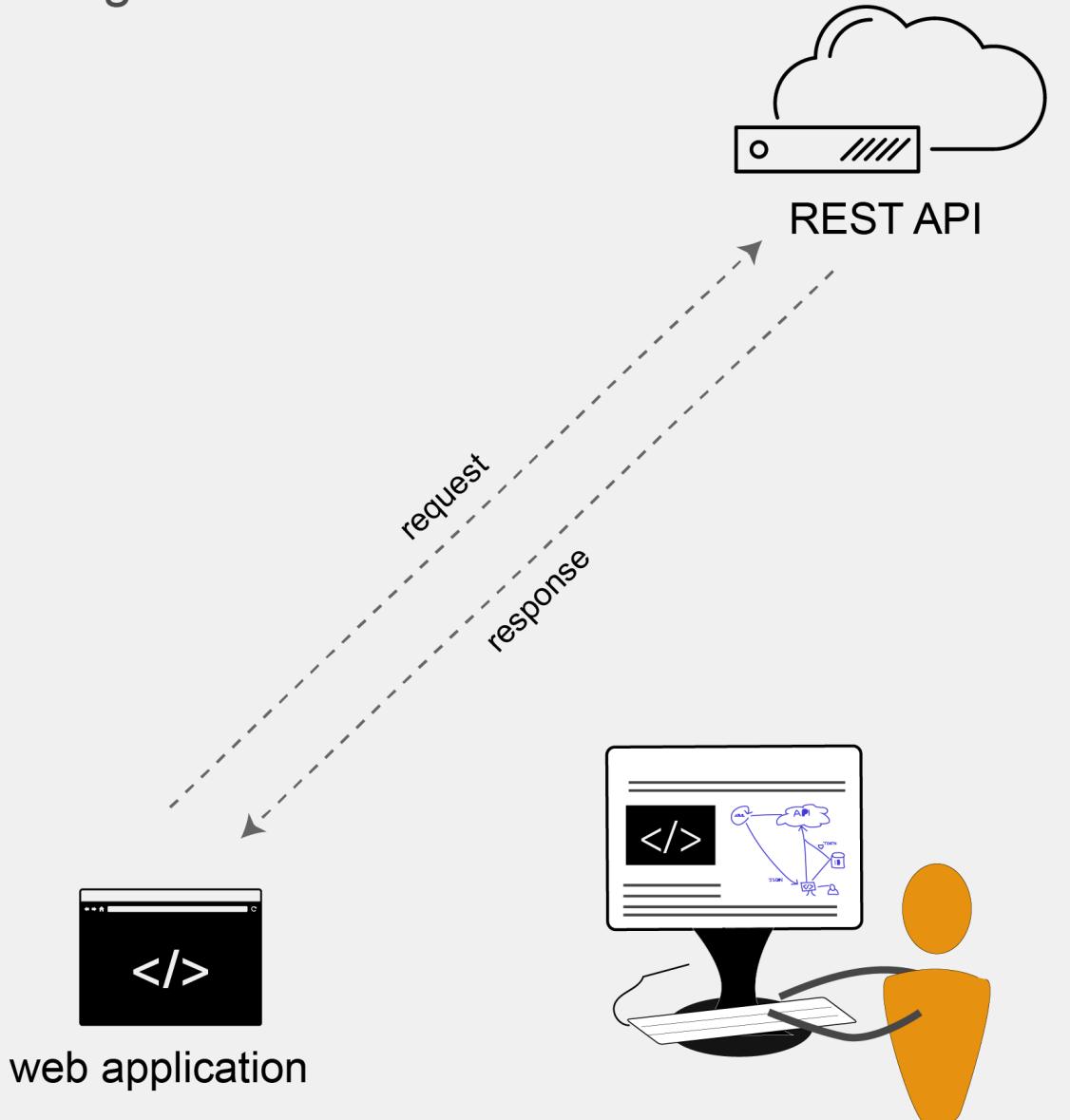


Documenting REST APIs

A guide for technical writers



web application

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
Course Background	10
About the author	12
The market for REST API documentation.....	14
What is a REST API?.....	22
Using a REST API as a developer	29
Scenario for using a weather API.....	30
Get authorization keys	35
Submit requests through Postman	38
cURL intro and installation.....	44
Make a cURL call.....	46
Understand cURL more	49
Use methods with cURL.....	54
Analyze the JSON response	60
Use the JSON from the response payload	66
Access and print a specific JSON value.....	72
Dive into dot notation.....	75
Documenting endpoints.....	81
A new endpoint to document	82
Documenting resource descriptions.....	85
Documenting endpoint definitions and methods.....	92
Documenting parameters	95
Documenting sample requests	102
Documenting sample responses	108
Documenting status and error codes	121
Documenting code samples	126
Putting it all together.....	136
Testing your API documentation.....	141
Set up a test environment.....	142
Test all instructions yourself	146
Test your assumptions.....	151
Documenting non-reference sections.....	155
User guide tasks	156
Documenting the API overview.....	161
Documenting the getting started section	162

Documenting authentication and authorization requirements	166
Documenting code samples and tutorials	173
How to create the quick reference guide.....	176
Next phase of course.....	178
Publishing your API documentation.....	179
Overview for publishing API docs	180
List of 100 API doc sites	183
Design patterns with API doc sites	187
Tool decisions: Who will write?.....	195
Docs-as-code tools	197
Github wikis	200
More about Markdown	205
Version control systems.....	213
Pull request workflows through GitHub	221
Jekyll — my favorite static site generator.....	227
Other tool options — a miscellaneous compilation	238
What about traditional help authoring tools (HATs)?.....	245
Tools versus Content.....	248
Case study: Switching tools to docs-as-code.....	249
OpenAPI specification and Swagger	257
Overview of REST API specification formats.....	258
Introduction to the OpenAPI spec and Swagger	259
OpenAPI tutorial overview	270
OpenAPI Tutorial Step 1: openapi object	274
OpenAPI Tutorial Step 2: info object	277
OpenAPI Tutorial Step 3: servers object.....	279
OpenAPI Tutorial Step 4: paths object	281
OpenAPI Tutorial Step 5: components object	292
OpenAPI Tutorial Step 6: security object.....	314
OpenAPI Tutorial Step 7: tags object	317
OpenAPI Tutorial Step 8: externaldocs object	319
OpenAPI specification activity: Create your own specification document.....	322
Swagger UI tutorial	323
Swagger UI activity: Create your own Swagger UI docs.....	331
Swagger UI Demo.....	333
Integrating Swagger UI with the rest of your docs	334
SwaggerHub introduction and tutorial.....	340
More about YAML.....	348
RAML tutorial	352
API Blueprint tutorial	363

Getting a Job in API Documentation	373
The job market for API technical writers.....	374
How much code do you need to know?.....	379
Resources and glossary.....	382
Glossary	383
Overview for exploring other REST APIs	389
EventBrite example: Get event information	390
Flickr example: Retrieve a Flickr gallery	397
Klout example: Retrieve Klout influencers	406
Aeris Weather Example: Get wind speed	416

Introduction to REST APIs

Course Overview	7
Course Background.....	10
About the author.....	12
The market for REST API documentation.....	14
What is a REST API?	22

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.

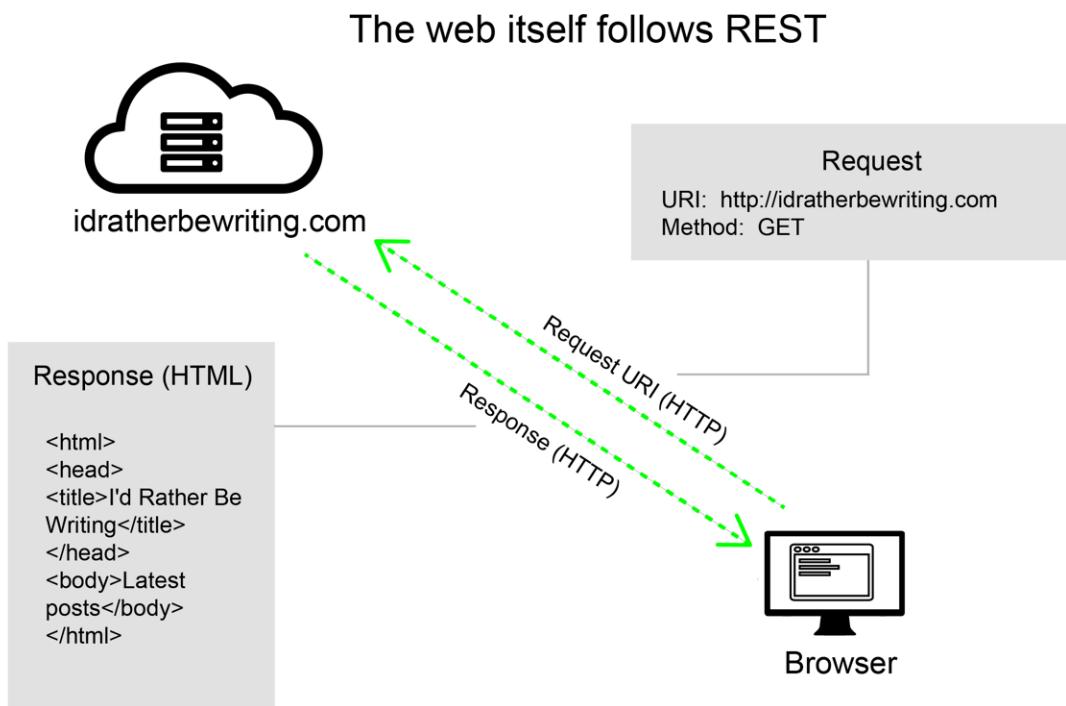
As you use the API, you'll learn about endpoints, parameters, data types, authentication, cURL, JSON, the command line, Chrome's Developer Console, JavaScript, and other details associated with REST APIs.

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.

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. REST stands for representational state transfer.

You may find REST APIs familiar because the web itself follows REST. Going to my website, idratherbewriting.com, involves submitting a GET request to my web server and getting a response (which happens to be rendered in a friendly, readable way through the browser). The protocol used to transport the data is HTTP.



REST APIs involve requests and responses over HTTP protocol

I dive more into the principles of REST in [What is a REST API? \(page 22\)](#)

From practice to documentation

After you use the sample weather API as 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:

- [Resource descriptions \(page 85\)](#)
- [Endpoint definitions and methods \(page 92\)](#)
- [Parameters \(page 95\)](#)
- [Sample requests \(page 102\)](#)
- [Sample responses \(page 108\)](#)
- [Status and error codes \(page 121\)](#)
- [Code samples \(page 126\)](#)

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 \(page 155\)](#), such as the [getting started \(page 162\)](#) and request [authorization \(page 166\)](#) sections.

Finally, you'll dive into different ways to [publish REST API documentation \(page 179\)](#), exploring tools and specifications such as [GitHub \(page 200\)](#), [Jekyll \(page 227\)](#), and other [Docs-as-code approaches \(page 197\)](#). 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 \(page 213\)](#).

I also dive into specifications such as the [OpenAPI specification \(page 270\)](#) and [Swagger \(page 259\)](#), which provides tooling for the OpenAPI specification. 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 \(page 29\)](#)
- [Documenting endpoints \(page 81\)](#)
- [Testing your API documentation \(page 141\)](#)
- [Documenting non-reference sections \(page 155\)](#)
- [Publishing your API documentation \(page 179\)](#)
- [OpenAPI specification and Swagger \(page 257\)](#)
- [Getting a job in API documentation \(page 373\)](#)
- [Resources and glossary \(page 382\)](#)

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 \(page 29\)](#), and the section on [documenting endpoints \(page 81\)](#)) 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.** ([Sublime Text](#) is a good option, it works on both Mac and Windows, but other text editors will also work.)
- **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.

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>.

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 them 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 field 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 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 stronger. Almost no content is finished on its first release. Instead, content needs to iterate over a period of time through user testing and feedback.

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. But as content strategists know, documentation is a rich information asset that draws traffic to any site. It's what people primarily search for online.

After having put 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 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....

And more. I regularly receive emails like the following:

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, from broken links to 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, 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

Hi, I'm Tom Johnson, a technical writer in the San Francisco Bay area. I work at [Amazon Lab126](#), which builds the devices for many of Amazon's products (such as the Kindle, Echo, Fire TV, and more). Here's the story of how I got into API documentation.

Like most technical writers, I stumbled into technical writing after working in other fields. After earning a BA in English and an MFA in nonfiction creative writing, I started out my career as a writing teacher. Realizing that teaching wasn't my calling, I transitioned into marketing copywriting, and then turned to technical writing to survive financially.

Despite my initial resistance to the idea of technical writing, 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 worked as a traditional technical writer for a number of years, mostly documenting applications with user interfaces until one day, my organization laid off the tech writing team.

After that, 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 the center of tech: Silicon Valley, which is the San Francisco Bay area of California.

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. Even before turning to developer documentation, I often set up or hacked the authoring tools and outputs. I like learning and experimenting with new technologies. As such, the developer documentation landscape suits me well, and I enjoy it.

Still, I'm by no means a programmer nor do I possess what I'd consider deep technical skills. As a technical writer, deep technical knowledge is helpful but not essential, as it tends to be too specialized. What matters most is your ability to learn something new, across a lot of different domains and products, even if it's challenging at first. And then to articulate your knowledge in easy-to-consume ways.

If you have a question for me, or just want to drop me a line, send your feedback to the e-mail address below.



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. To get a taste of the various types of APIs out there, check out Sarah Maddox's post about [API types](#). In addition to web service APIs (which include REST), there are web socket APIs, hardware APIs, and more.

A simple classification of APIs

Web service APIs

- SOAP
- XML-RPC and JSON-RPC
- REST

WebSocket APIs

- Library-based APIs
 - JavaScript
 - TWAIN

Class-based APIs (object orientation)

- Java API
- Android API

OS functions and routines

- Access to file system
- Access to user interface

Object remoting APIs

- CORBA
- .NET Remoting

Hardware APIs

- Video acceleration
- Hard disk drives
- PCI buses

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

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.

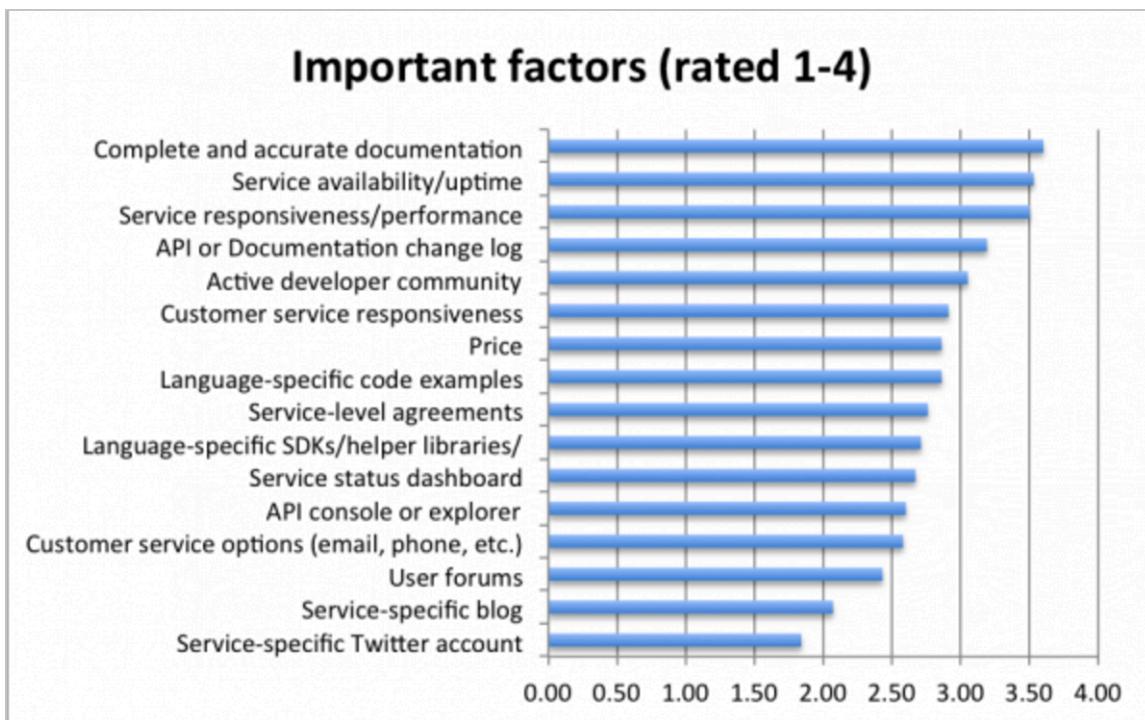
REST APIs follow the same protocol as the web. When you open a browser and type a website URL (such as <http://idratherbewriting.com>), you're actually making a GET request for a resource on a server. The server responds with the content and the browser makes the content visible.

This course focuses mostly on REST APIs because REST APIs are more accessible to technical writers, as well as more popular and in demand. You don't need to know programming to document REST APIs. And REST is becoming the most common type of API anyway.

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.”

REASON #1

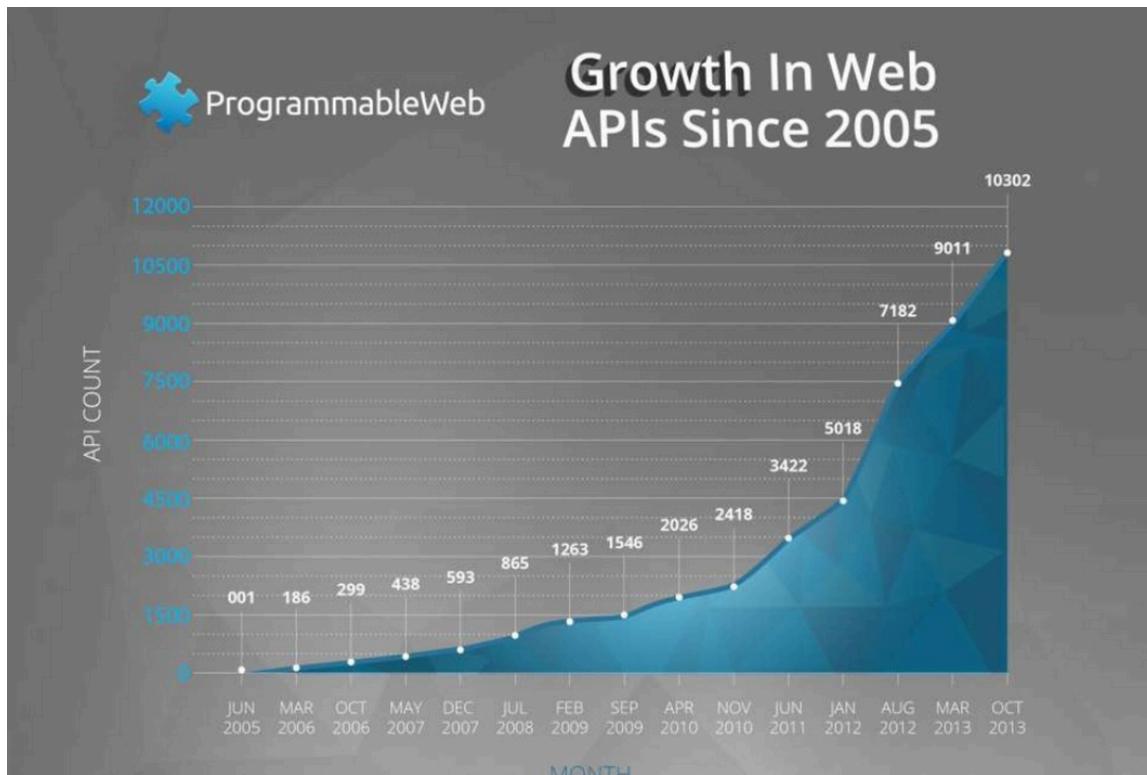
Your

documentation

sucks

Since 2005, REST APIs are taking off in a huge way

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 exploding.

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.

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 site.

CloudCannon has put together a [long list of services](#) that you can integrate into your static site.

Services

Third-party services for Jekyll websites.

Newsletters

Capture email addresses and send periodic newsletters.


AWeber


Campaign Monitor


MailChimp


MailerLite


Sendicate

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
 what: <input type="text"/> job title, keywords or company where: <input type="text"/> city, state, or zip		
<p>Contract API Tech Writer, Palo Alto Synergistech - Palo Alto, CA</p> <p>Principals only, please</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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</p>		

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 subtext 'A powerful API for apps that work with files.' On the left side, there's a sidebar with links like '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'), and 'Read our docs', 'Create your app', and 'Test your ideas'. The central area features a cartoon illustration of three people working on computers, with gears and clouds in the background. Below the illustration are three boxes: 'Read our docs' (Docs are organized by language, from .NET to Swift.), 'Create your app' (Getting started is simple and quick from the App Console.), and 'Test your ideas' (It's easy to prototype and test examples with our API Explorer.).

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.

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 seem foreign. For example, all of these aspects differ from traditional documentation:

- API doc authoring tools
- Developer audience
- Programming languages
- Endpoint reference topics
- User tasks

When you try to navigate the world of API documentation, the world probably looks as unfamiliar as Mars.

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 have asked for more training. In our Silicon Valley STC chapter, we've 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.

In 2014, the STC Summit in Columbus held its first ever API documentation track. Since then, API documentation tracks and themes have become a consistent pattern in technical writing conferences. Many technical writers choose to specialize in API documentation as a way of distinguishing themselves and their skillsets in the marketplace.

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 market.

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 standards or protocols like their predecessors did.

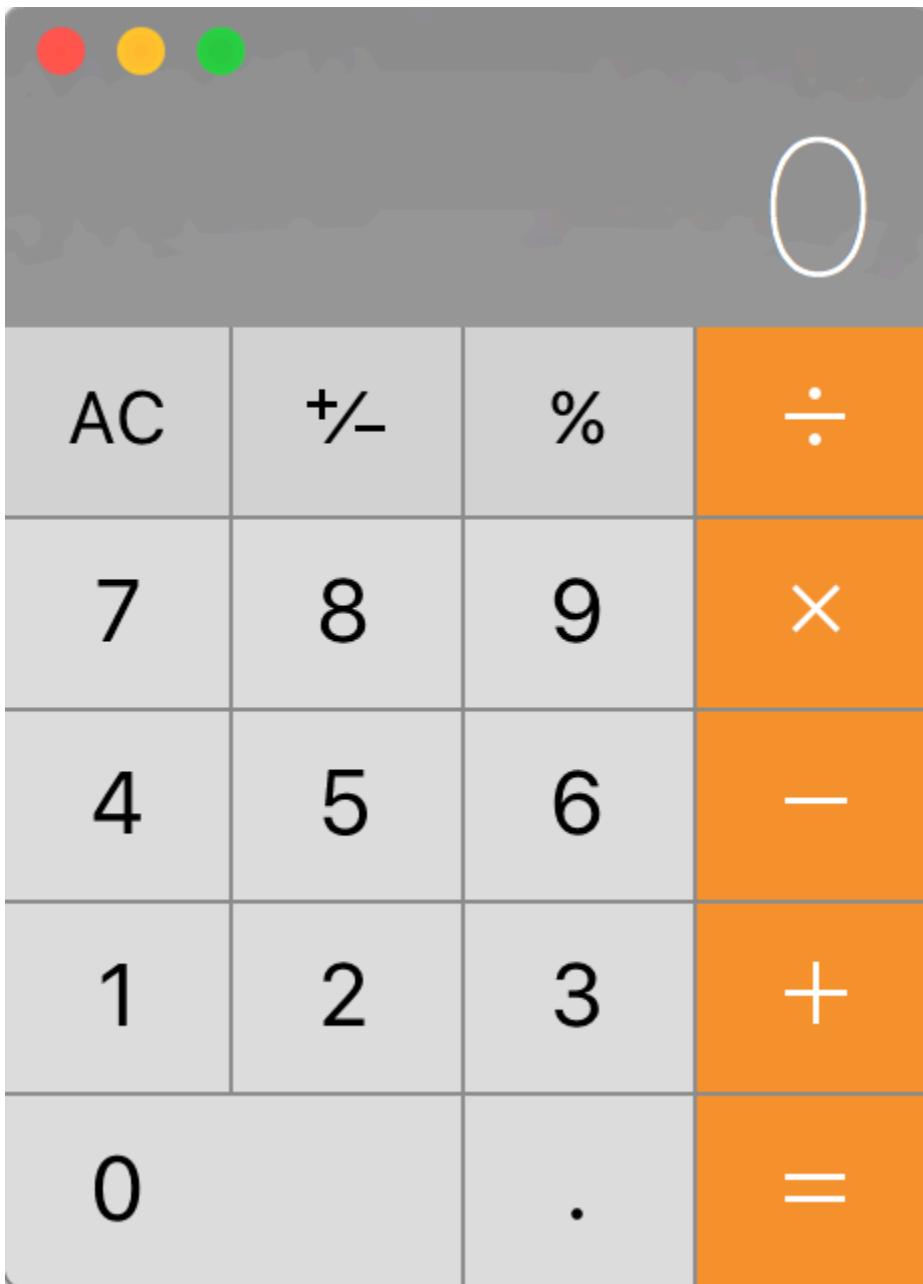
What is an API?

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 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 information 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, 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 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.

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 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. You can browse some SOAP APIs at <http://xmethods.com/ve2/index.po>.

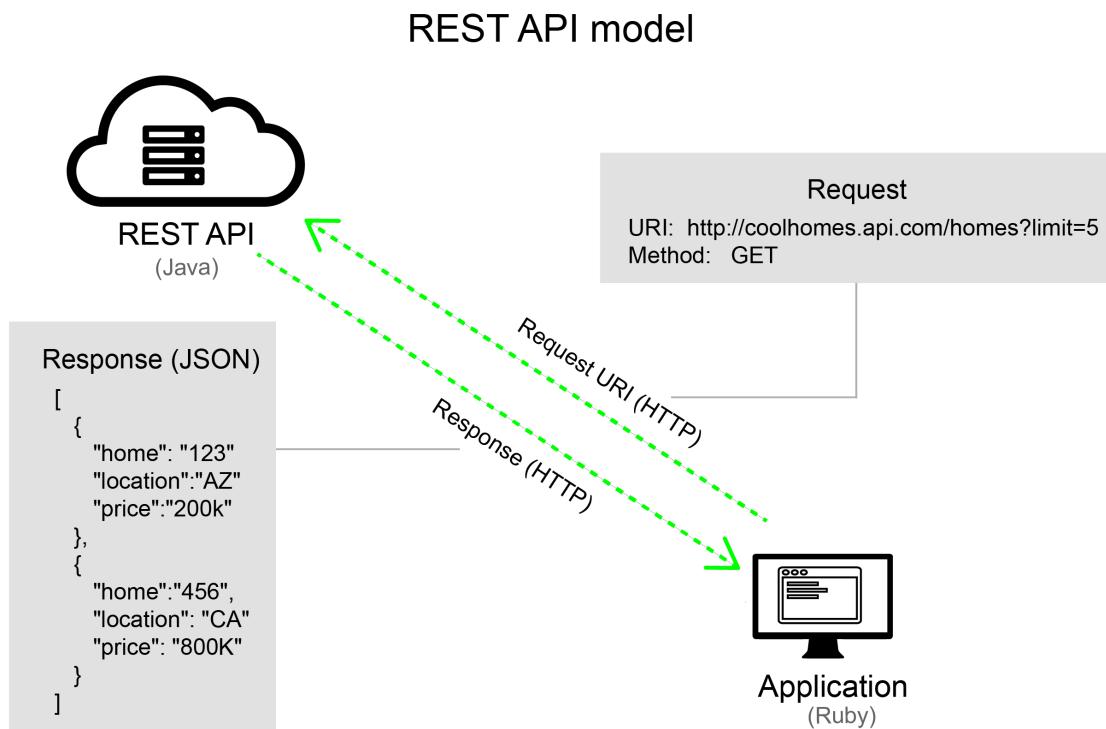
REST is a style, not a standard

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

Here’s 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 you have to use XSLT to parse and process the content.

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. You access the resources through URLs (Uniform Resource Locations). 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 URL 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.

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.

Sample URLs for a REST API

Here’s what a sample REST URI or endpoint might look like:

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

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/1234
```

This might be an endpoint that retrieves a home resource with an ID of `1234`. 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 home 1234.

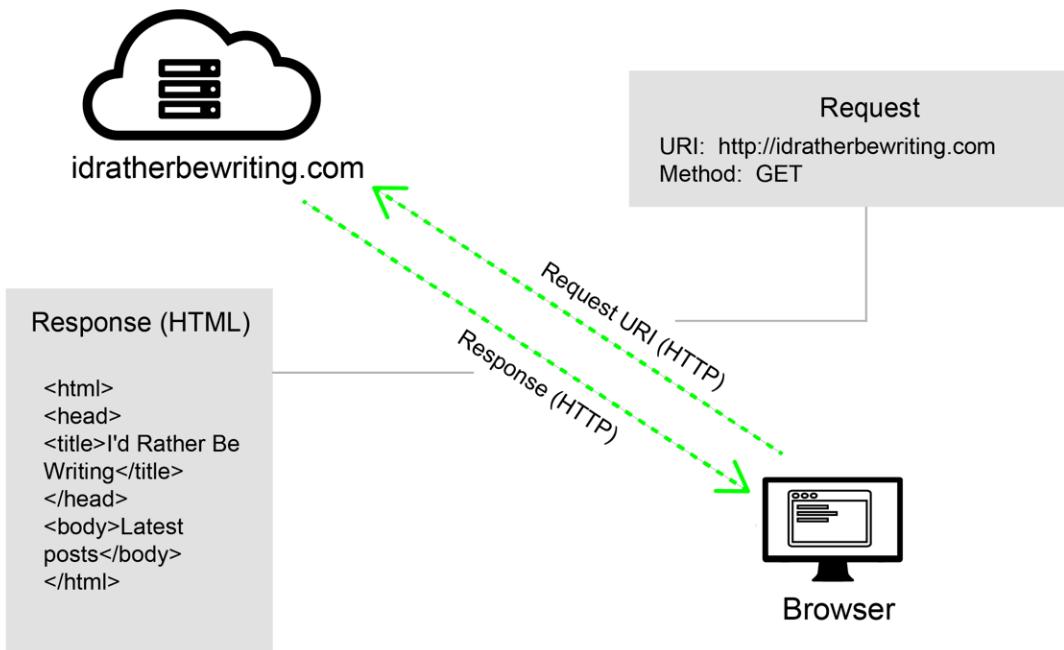
We’ll explore endpoints in much more depth in the sections to come. But I wanted to provide a brief overview here.

The web itself follows REST

The terminology of “URLs” 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.

The web itself follows REST



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

Some additional features of REST APIs are that they are 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, Swagger (also called OpenAPI) and RAML — have been developed to describe REST APIs. When you describe your API using the [Swagger \(page 323\)](#) or [RAML \(page 352\)](#) specification, tools that can read those specifications (like Swagger UI or the RAML API Console) will generate an interactive documentation output.

The Swagger or RAML output can take the place of the WSDL file that was more common with SOAP. These spec-driven outputs 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, 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, 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

Using a REST API like a developer

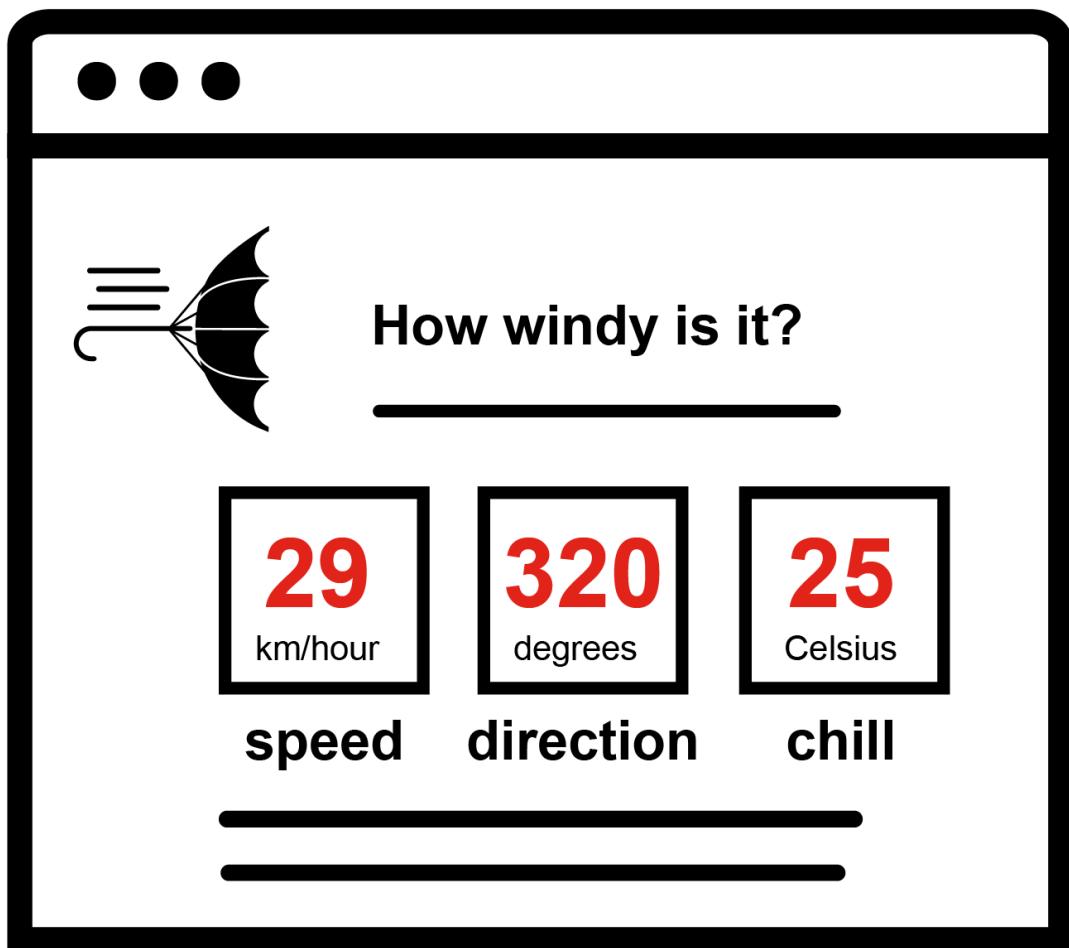
Scenario for using a weather API	30
Get authorization keys.....	35
Submit requests through Postman.....	38
cURL intro and installation	44
Make a cURL call.....	46
Understand cURL more.....	49
Use methods with cURL.....	54
Analyze the JSON response.....	60
Use the JSON from the response payload.....	66
Access and print a specific JSON value	72
Dive into dot notation	75

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 use two different 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.

Let's say that you're a web developer and you want to add a weather forecast feature to your site. Your site is for bicyclists. You want to allow users who come to your site to see what the wind conditions are for biking. You want something like this:

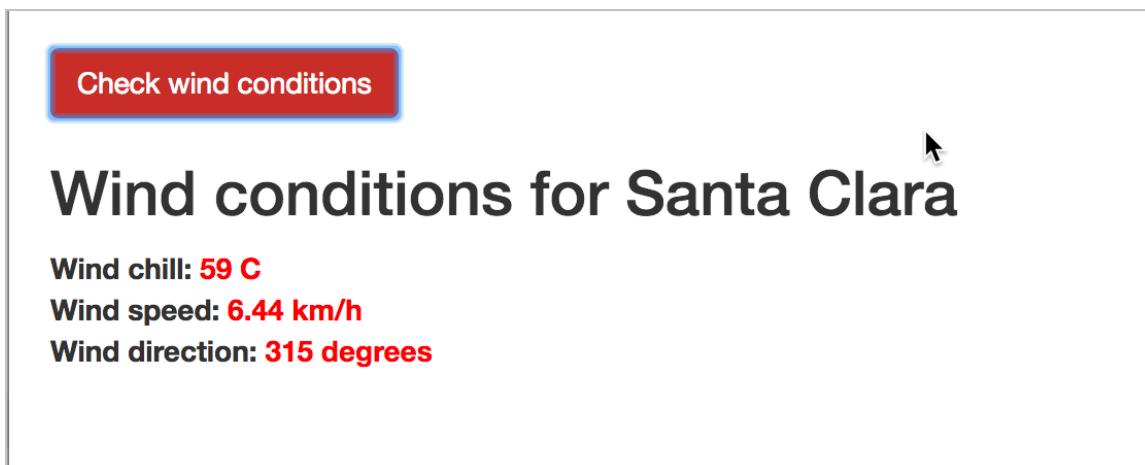


You don't have your own meteorological service, so you'll need to make some calls out to a weather service to get this information. Then you will present that information to users.

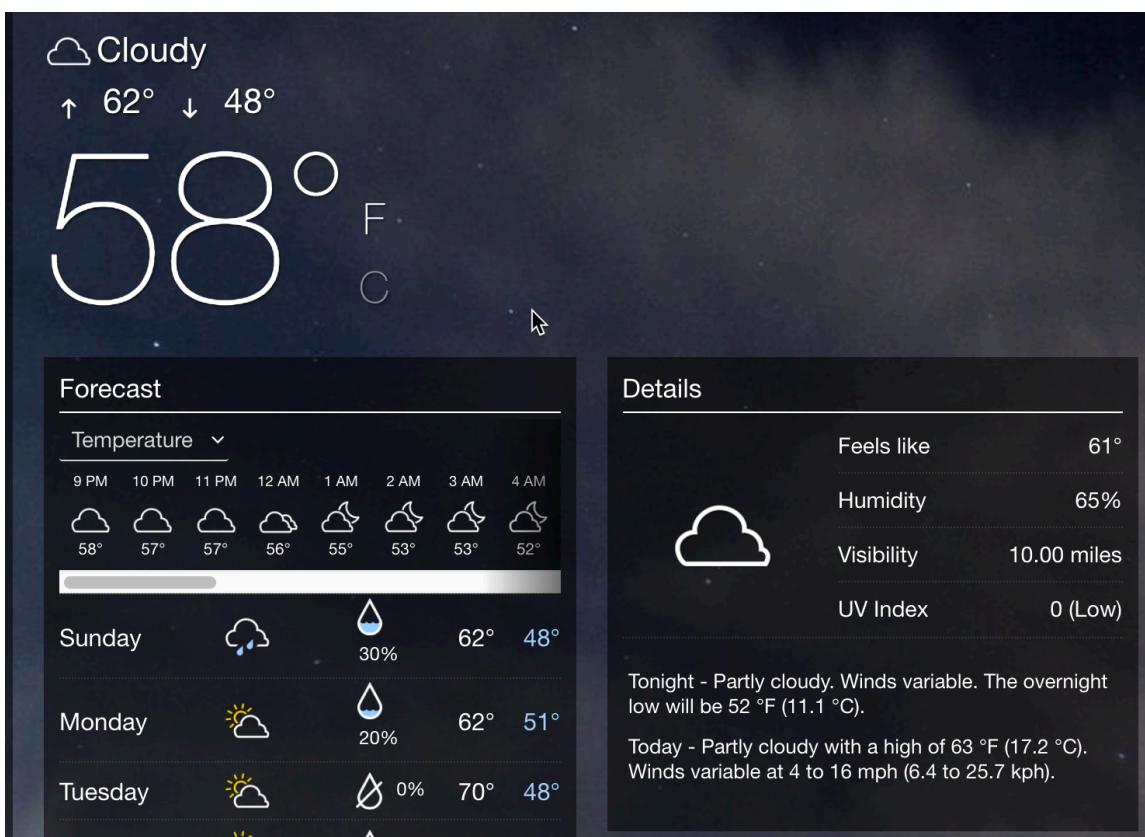
Get an idea of the end goal

To give you an idea of the end goal, here's a sample: idrathebewriting.com/learnapidoc/assets/files/wind-mashape.html. It's not necessarily styled the same as the mockup, but it answers the question, "How windy is it?"

Click the button to see wind details. When you request this data, an API goes out to a weather service, retrieves the information, and displays it to you.



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



The concept and general technique is more or less the same.

Find the Weather API by fyhao on Mashape

The Mashape Marketplace is a directory where publishers can publish their APIs, and where consumers can consume the APIs. Mashape manages the interaction between publishers and consumers by providing an interactive marketplace for APIs.

The APIs on Mashape tend to be rather simple compared to some other APIs, but this simplicity will work well to illustrate the various aspects of an API without getting too mired in other details.



You're a consumer of an API, but which one do you need to pull in weather forecasts?

ACTIVITY

Explore the APIs available on Mashape and find the weather forecast API:

1. Go to [Mashape Marketplace](#) and click **Explore APIs**.
2. Try to find an API that will allow you to retrieve the weather forecast.

As you explore the various APIs, get a sense of the variety and services that APIs provide. These APIs aren't applications themselves. They provide developers with ways to pipe information into their applications. In other words, the APIs will provide the data plumbing for the applications that developers build.

3. Search for an API called "Weather," by fyhao at <https://market.mashape.com/fyhao/weather-13>. Although there are many weather APIs, this one seems to have a lot of reviews and is free.

The screenshot shows the Aeris Weather API documentation page. At the top, there's a header with the Aeris logo, developer statistics (3930 developers, 3526 followers), and a 'REPORT BROKEN API' button. Below the header, a navigation bar includes links for DOCUMENTATION, OVERVIEW, ANNOUNCEMENTS, and API SUPPORT. The main content area is titled 'ENDPOINTS' and shows the 'aqi' endpoint. It includes a brief description ('Air Quality Index'), URL parameters ('lat' and 'lon'), and a 'REQUEST EXAMPLE' section with a curl command.

Find the Aeris Weather API

Now let's look at another weather API (this one not on Mashape). In contrast to the simple API on Mashape, the [Aeris Weather API](#) is much more robust and extensive. You can see that the Aeris Weather API is a professional grade, information-rich API that could empower an entire meteorology service.

ACTIVITY

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 page. The top navigation bar includes links for APPS, DEVELOPER, INDUSTRIES, SUPPORT, and CASE STUDIES. The main content area is titled 'API Endpoints'. On the left, there's a sidebar with links for 'Aeris Weather API' (Getting Started, Reference, Downloads), 'Getting Started' (Authentication, Responses, Advanced Queries, Sorting, Reducing Output, Batch Requests), and 'Reference' (Endpoints, Actions, Supported Places). The main content area describes endpoints and provides a table of supported endpoints with their descriptions and data coverage.

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

Here's the Aeris weather forecast API in action making the same call as I showed earlier with Mashape:
[/learnapidoc/assets/files/wind-aeris.html](#).

As you can see, both APIs contain this same information about wind, but the units differ.

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?

These are common questions developers want to know about an API.

Can you see how APIs can differ significantly? As I mentioned previously, REST APIs are an architectural style, not a specific standard that everyone follows. You really have to read the documentation to understand how to use them.

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. 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 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.

Get the Mashape authorization keys

To get the authorization keys to use the Mashape API, you must sign up for a Mashape account.

1. On market.mashape.com, click **Sign Up** in the upper-right corner and create an account. It's easiest if you first create an account on GitHub, and then just click **SIGNUP WITH GITHUB** in the Mashape login window.
2. Click **Applications** on the top navigation bar, and then select **Default Application**.
3. In the upper-right corner, click **Get the Keys**.

The screenshot shows the Mashape API Marketplace interface. At the top, there's a navigation bar with links for Search APIs, Explore APIs, Docs, Features, Applications (3), My APIs (0), and a user profile icon for 'tomjohnson1492'. Below the navigation, a banner promotes Kong, Galileo, and Gelato. The main area displays a grid of API cards. The first card, 'Simple Weather' by 'Ser Programadores', shows a 'No Data Available' message. The second card, 'Weather' by 'fyhao', includes a weather icon, a temperature of 73°, and a line graph. The third card, 'Indeed' by 'indeed', and the fourth card, 'FOOAS' by 'community', both show 'No Data Available'. In the top right corner of the 'Weather' card, there is a blue button labeled 'GET THE KEYS'. A thick orange arrow points from the text in step 3 above to this 'GET THE KEYS' button.

If you don't see the Get the Keys button, make sure you click **Applications > Default Application** on the top navigation bar first. You may have to horizontally scroll to the right (eek!) to see the Get the Keys button.

4. When the Environment Keys dialog appears, click **Copy** to copy the keys. (Choose the Testing keys, since this type allows you to make unlimited requests.)

ENVIRONMENT KEYS

Here are the keys related to the current environment. Should a sensitive key be leaked or accidentally exposed you can regenerate them at will, if you need to block them immediately you are also able to do this.

Testing	▼	EF3g83pKnzmshgoksF83V6JB6QyTp1cGrdd	COPY	REGENERATE ▾
---------	---	-------------------------------------	------	--------------

5. Open a text editor and paste the key so that you can easily access it later when you construct a call.

Get the Aeris Weather API secret and ID

Now 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**. (Note that 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 developer portal. At the top, there's a header with weather information (57°), a search bar ('Weather for...'), and account links ('ACCOUNT' and 'LOG OUT'). Below the header, there's a navigation bar with links like 'DASHBOARD', 'USAGE', 'APPS' (which is underlined), 'MAP BUILDER', 'LEGEND GENERATOR', 'ADD SUBSCRIPTION', 'DOCS', 'SUPPORT', 'CASE STUDIES', 'BLOG', and 'SIGN UP'. A 'PROFILE' link is also visible. The main content area is titled 'Registered Apps'. It contains a list of registered applications, including one named 'API testing' with an ID and secret. There are edit and delete icons next to each application entry. A prominent orange arrow points from the bottom-left towards the 'NEW APPLICATION' button, which is located in the top right corner of the 'Registered Apps' section.

6. In the dialog box, enter the following:
 - **Application Name:** My biking app (or something)
 - **Application Namespace:** localhost
7. Click **Save App**.

Once 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 using TextEdit since it adds some formatting behind the scenes that can corrupt your content.

Submit requests through Postman

When you're testing endpoints with different parameters, you can use one of the many GUI REST clients available. 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](#)
- [Advanced REST Client](#) (Chrome browser extension)
- [REST Console](#)
- [Paw](#) (Mac, \$30)

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 resource URL, hang in there. When we deep dive into these points later, things will be a lot clearer.

Make a request in Postman

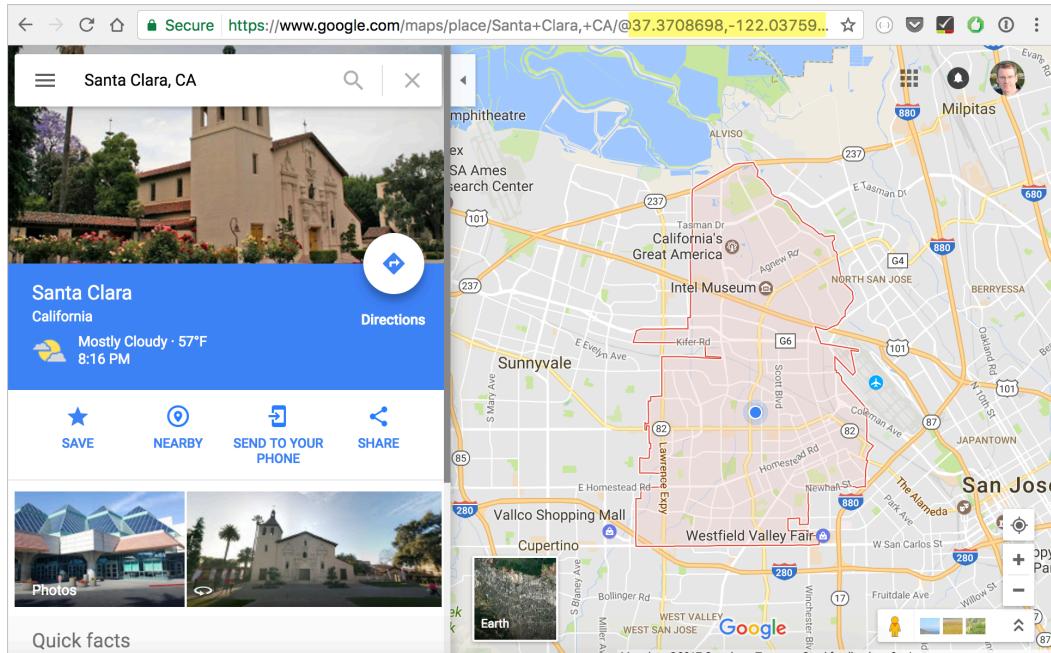
1. If you haven't already done so, download and install the Postman app at <http://www.getpostman.com>. If you're on a Mac, choose the Mac app. If you're on Windows, choose the Windows app.
2. Start the Postman app.
3. You'll make a REST call for the second endpoint (<https://simple-weather.p.mashape.com/weather>) in the Mashape Weather API. Select **GET** for the method.

You can also call the first endpoint ([aqi](#)), but the response is pretty short (2 characters), and unfortunately sometimes the API doesn't always return a response for the location. If this endpoint isn't working, you'll see "Not supported" response.

4. Insert the endpoint into the main box (next to the method, which is GET by default): <https://simple-weather.p.mashape.com/weather>
5. Click the **Params** button (to the right of the box where you inserted the endpoint) and insert [lat](#) and [lng](#) parameters with specific values (other than [1](#)).

Only some countries are supported in this weather API — specifically the United States, Singapore, Malaysia, Europe, and Australia. If the country isn't supported, you'll see "Not supported" in the API response. The latitude and longitude coordinates for Santa Clara, California

are `lat: 37.3710062` and `lng: -122.0375935`. For Singapore, they're `lat: 1.3321256` and `lng: 103.7373503`. You can find latitude and longitude values from the URL in Google Maps when you go to a specific location. The latitude appears first.

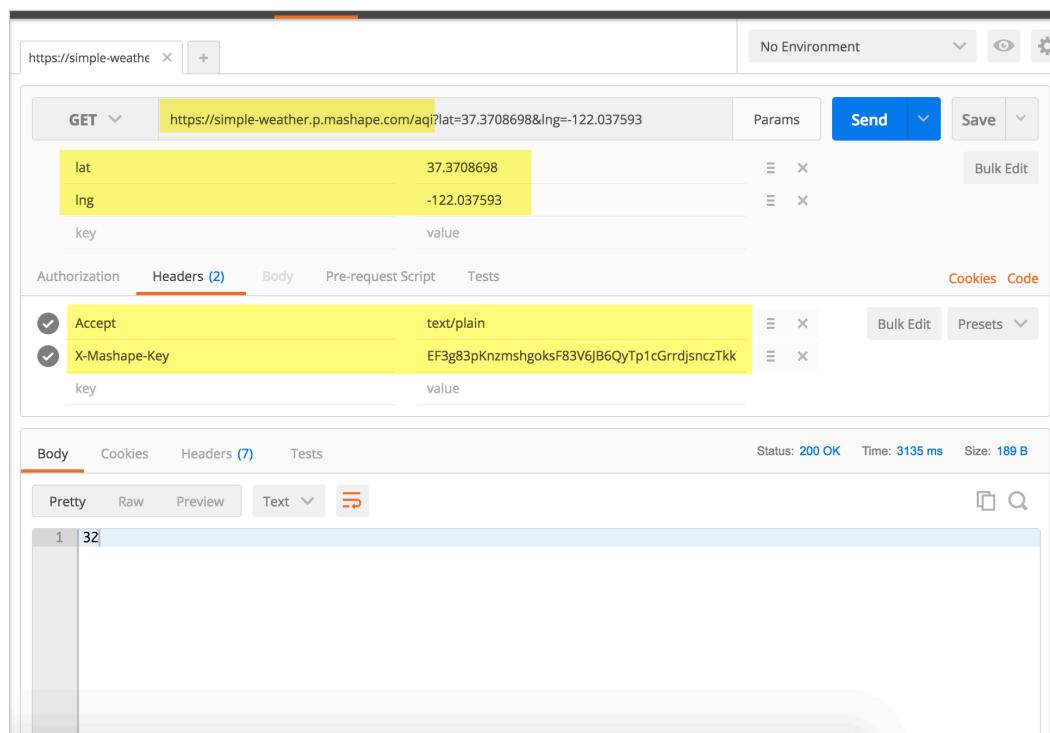


Latitude is listed first, then longitude

When you add these `lat` and `lng` parameters, they will dynamically be added as a query string to the endpoint URI. The query string is the code followed by the `?` in the endpoint URL. For example, your endpoint should now look like this: <https://simple-weather.p.mashape.com/weather?lat=37.3710062&lng=-122.0375935>. Query string parameters appear after the question mark `?` symbol and are separated ampersands `&`.

6. Click the **Headers** tab (below the GET button) and insert the key value pairs: `Accept: text/plain` and `X-Mashape-Key: APIKEY`. (Swap in your own API key in place of `APIKEY`.)

Your inputs should look like this:



- Click **Send**.

The response appears, such as [52](#). In this case, the response is text only. You can switch the format to HTML, JSON, XML, or other formats, but since this response is text only, you won't see any difference. Usually the responses are JSON, which allows you to select a specific part of the response to work with.

If you get a response that says "Unsupported," this means your `lat` and `lng` values aren't supported. Use the `lat` and `lng` values shown here ([?lat=37.3710062&lng=-122.037593](#)).

Save the request

- In Postman, click the **Save** button (next to Send).
- In the Save Request dialog box, create a new collection (for example, weather) by typing the collection name in the "**Or create new collection**" box.
- In the **Request Name** box at the top of the dialog box, type a friendly name for the request, such as "Mashape Weather endpoint."
- Click **Save**.

Saved endpoints appear in the left side pane under Collections.

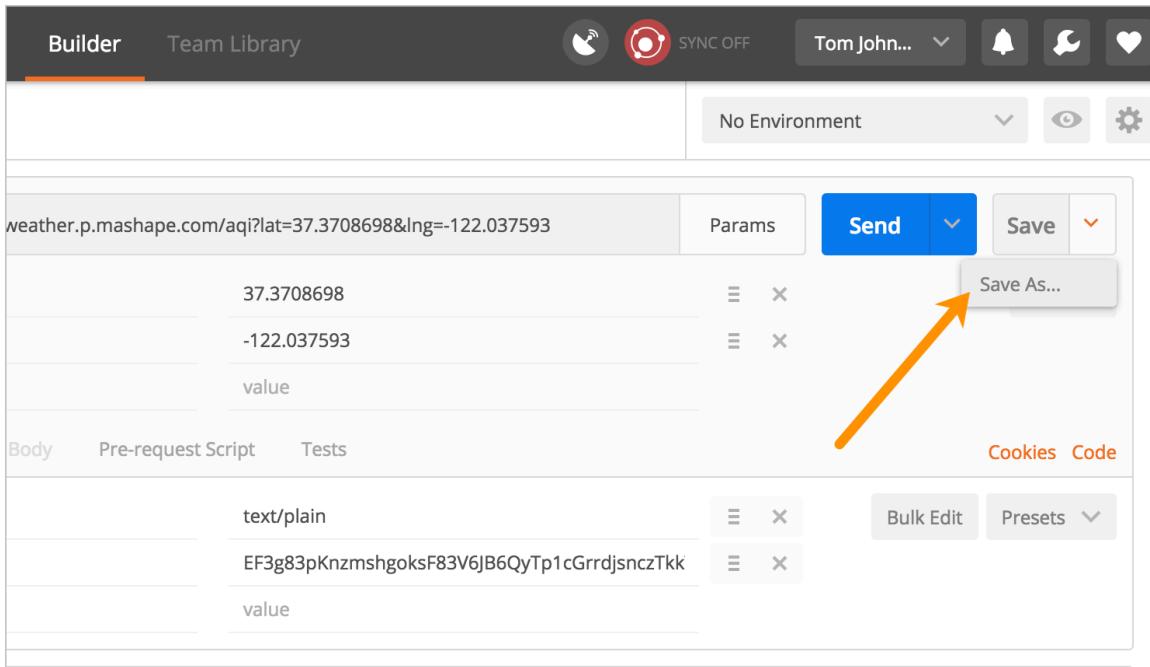
Make requests for the other endpoint

Enter details into Postman for the other endpoint for the Mashape Weather API:

- <https://simple-weather.p.mashape.com/weatherdata>

The Accept header tells the browser what format you will accept the response in. Whereas the first two endpoints (aqi and weather) use `text/plain`, the Accept header for the `weatherdata` endpoint is `application/json`.

When you save these other endpoints, click the arrow next to Save and choose **Save As**. Then choose your collection and request name. (Otherwise you'll overwrite the settings of the existing request.)



(Alternatively, click the + button on the new tab and create each request in separate tabs.)

View the format of the weatherdata response in JSON

While the first two endpoint responses include text only, the weatherdata endpoint response is in JSON.

In Postman, make a request to the weatherdata API. Then toggle the options to **Pretty** and **JSON**.

The screenshot shows the Postman interface with a GET request to <https://simple-weather.p.mashape.com/weatherdata?lat=37.354108&lng=-121.955236>. The response body is displayed in Pretty JSON format, which includes the following data:

```

1 {
2   "query": {
3     "count": 1,
4     "created": "2015-06-12T20:51:32Z",
5     "lang": "en-US",
6     "results": {
7       "channel": {
8         "title": "Yahoo! Weather - Santa Clara, CA",
9         "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA1018_c.html",
10        "description": "Yahoo! Weather for Santa Clara, CA",
11        "language": "en-us",
12        "lastBuildDate": "Fri, 12 Jun 2015 12:53 pm PDT",
13        "ttl": "60",
14        "location": {
15          "city": "Santa Clara",
16          "country": "United States",
17          "region": "CA"
18        },
19        "units": {
20          "distance": "km",
21          "pressure": "mb",
22          "speed": "km/h",
23          "temperature": "C"
}
}
}

```

The Pretty JSON view expands the JSON response into more readable code.

To “prettify” code means to un-minify it and format it with white space that is readable.

For the sake of variety with GUI clients, here's the same call made in Paw:

The screenshot shows the Paw interface with a GET request to <https://simple-weather.p.mashape.com/weatherdata?lat=37.354108&lng=-121.955236>. The response is shown in a tree-view JSON editor. The data structure is identical to the one shown in Postman, representing a weather forecast for Santa Clara, CA.

Like Postman, Paw also allows you to easily see the request headers, response headers, URL parameters, and other data. However, Paw is specific to Mac only.

Enter several requests for the Aeris API into Postman

Now let's switch APIs a bit and see some weather information from the Aeris API. 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 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.

Note that 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`.

Get the weather forecast for your area:

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

In the response, find the wind speed and compare it with the wind from the Mashape API. Are they the same?

Get the weather from a city on the equator — Chimborazo, Ecuador:

```
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:

```
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
```

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.

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.

cURL is a command-line utility that lets you execute HTTP requests with different parameters and methods. In other words, 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 7.51.0 (x86_64-apple-darwin16.0) libcurl/7.51.0 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: AsynchDNS IPv6 Largefile GSS-API Kerberos SPNEGO NTLM NTL
M_WB SSL libz UnixSockets
```

If you don't see this, you need to [download and install cURL](#).

To make a test API call, submit the following:

```
curl --get -k --include "https://simple-weather.p.mashape.com/weather?la
t=1.3319164&lng=103.7231246" -H "X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB60yT
p1cGrrdjsnczTkkYgYrp8p" -H "Accept: text/plain"
```

You should get back a response like this:

```
29 c, Thunderstorms at Singapore, Singapore
```

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](#).

Once 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 7.51.0 (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
```

To make a test API call, submit the following:

```
curl --get -k --include "https://simple-weather.p.mashape.com/weather?la  
t=1.3319164&lng=103.7231246" -H "X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB6QyT  
p1cGrrdjsnczTkkYgYrp8p" -H "Accept: text/plain"
```

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

You should get back a response like this:

```
29 c, Thunderstorms at Singapore, Singapore
```

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. Unfortunately the Weather API on Mashape uses these slashes in their cURL examples.)
- By adding **-k** in the cURL command, you bypass cURL's security certificate, which may or may not be necessary.

Make a cURL call

If you haven't installed cURL, see [cURL intro and installation \(page 44\)](#) first.

In this section, you'll use cURL to make the same weather API requests you made previously with Postman.

Prepare the weather request in cURL format

1. Go back into the [Weather API on Mashape](#).
2. Copy the cURL request example for the second endpoint (`weather`) into your text editor:

```
curl --get --include 'https://simple-weather.p.mashape.com/weather?lat=1.0&lng=1.0' \
-H 'X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p' \
-H 'Accept: text/plain'
```

3. If you're on Windows, do the following:
 - Change the single quotation marks to double quotation marks
 - Remove the backslashes (`\`)
 - Add `-k` after `--get` as well to work around security certificate issues.

The request should now look like this:

```
curl --get -k --include "https://simple-weather.p.mashape.com/weather?lat=1.0&lng=1.0" -H "X-Mashape-Key: APIKEY" -H "Accept: text/plain"
```

4. Swap in your own API key in place of `APIKEY`.

In the instruction in this course, `APIKEY` will always be used instead of an actual API key. You should replace that text with your own API key.

5. Customize the `lat` and `lng` values to the following:
 - `lat=1.3319164`
 - `lng=103.7231246`

Make the request in cURL (Mac)

1. Open a terminal. To open Terminal, press **Cmd + space bar** and type **Terminal**.

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

2. Paste the request you have in your text editor into the command line.

My request for the Mashape Weather API looks like this:

```
curl --get --include 'https://simple-weather.p.mashape.com/weather?lat=1.3319164&lng=103.7231246' -H 'X-Mashape-Key: APIKEY' -H 'Accept: text/plain'
```

For the Aeris Weather observations endpoint, it looks like this:

```
curl --get --include "http://api.aerisapi.com/observations/santa%20
clara,ca?client_id=CLIENTID&client_secret=CLIENTSECRET" "Accept: appl
ication/json"
```

3. Press your **Enter** key.

You should see something like this as a response:

The response should look something like this:

```
29 c, Thunderstorms at Singapore, Singapore
```

Make the request in cURL (Windows 7)

1. Copy the cURL call from your text editor.
2. Go to **Start** and type **cmd** to open up the commandline. (If you're on Windows 8, see [these instructions for accessing the commandline](#).)
3. Right-click and then select **Paste** to insert the call. My call for the Mashape API looks like this:

```
curl --get -k --include "https://simple-weather.p.mashape.com/weath
er?lat=1.3319164&lng=103.7231246" -H "X-Mashape-Key: APIKEY" -H "Acce
pt: text/plain"
```

For the Aeris endpoint, it looks like this:

```
curl --get --include "http://api.aerisapi.com/observations/santa%20cl
ara,ca?client_id=CLIENTID&client_secret=CLIENTSECRET" -H "Accept: app
lication/json"
```

The response should look something like this:

```
29 c, Thunderstorms at Singapore, Singapore
```

Single and Double Quotes with Windows cURL requests

Note that if you're using Windows to submit a lot of cURL requests, you'll eventually 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 run into issues in submitting cURL requests.

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, but you can also reference the complete path to the JSON file.

Make cURL requests for each of the weather endpoints

Make a cURL request for each of the weather endpoints for both the Mashape weather endpoints and the Aeris Weather endpoints, similar to how you made the requests in Postman.

Understand cURL more

Before moving on, let's pause a bit and learn more about 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.

Almost every API shows how to interact with the API using cURL.

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 is the 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:

```
curl --get --include 'https://simple-weather.p.mashape.com/weather?lat=37.354108&lng=-121.955236' \
-H 'X-Mashape-Key: APIKEY' \
-H 'Accept: text/plain'
```

cURL has shorthand names for the various options that you include with your request. The `\` just creates a break for a new line for readability. (Don't use `\` in Windows.)

Here's what the commands mean:

cURL command	Description
<code>--get</code>	The HTTP method to use. (This is actually unnecessary. You can remove this and the request returns the same response, since GET is the method used by default.)
<code>--include</code>	Whether to show the headers in the response. Also represented by <code>-i</code> .
<code>-H</code>	Submits a custom header. Include an additional <code>-H</code> for each header key-value pair you're submitting.

Most cURL commands have a couple of different representations. `--get` can also be written as `-X GET`.

Query strings and parameters

The latitude (`lat`) and longitude (`lng`) 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:

```
?lat=37.354108&lng=-121.955236
```

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>
<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.

Summary

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 from Mashape doesn't allow you to use anything but a GET method, so for this example, we'll use the [petstore API from Swagger](#), but without actually using the Swagger UI (which is something we'll explore later). For now, we just need an API with which we can use to create, update, and delete content.

Swagger Petstore

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.

Find out more about Swagger

<http://swagger.io>
[Contact the developer](#)
[Apache 2.0](#)

pet : Everything about your Pets

Show/Hide | List Operations | Expand Operations

POST	/pet	Add a new pet to the store
PUT	/pet	Update an existing pet
GET	/pet/findByStatus	Finds Pets by status
GET	/pet/findByTags	Finds Pets by tags
DELETE	/pet/{petId}	Deletes a pet
GET	/pet/{petId}	Find pet by ID
POST	/pet/{petId}	Updates a pet in the store with form data
POST	/pet/{petId}/uploadImage	uploads an image

In this example, 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) and the pet name of `fluffy`.

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

Now you want to 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. <cry + tears />

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.

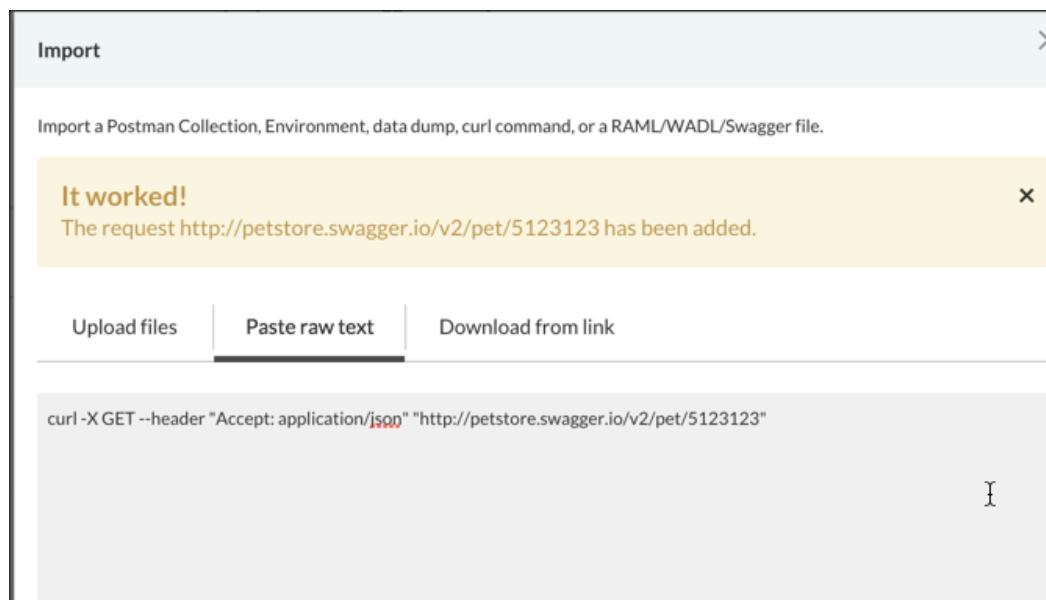
Although Postman is probably easier to use, cURL lends itself to power-level usage. Quality assurance teams often construct advanced test scenarios that iterate through a lot of cURL requests.

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"
```



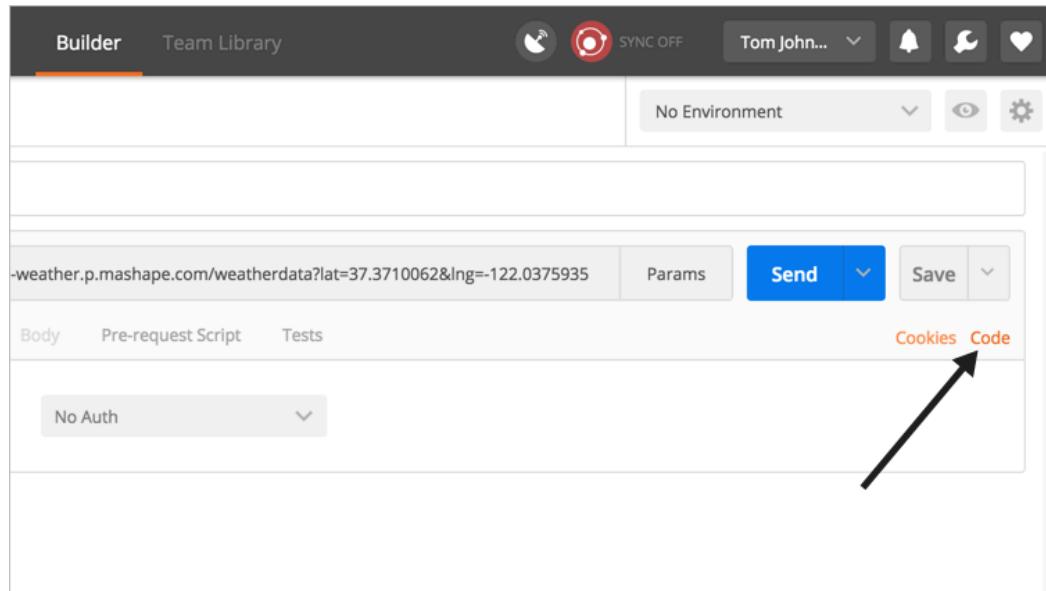
Make sure you don't have any extra spaces at the beginning.

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.



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

Let's look at the JSON response for the Mashape weatherdata endpoint in more depth. The minified response from cURL looks like this:

```
{"query":{"count":1,"created":"2015-06-03T16:24:26Z","lang":"en-US","results":{"channel":{"title":"Yahoo! Weather - Santa Clara, CA","link":"http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA1018_c.html","description":"Yahoo! Weather for Santa Clara, CA","language":"en-us","lastBuildDate":"Wed, 03 Jun 2015 8:52 am PDT","ttl":"60","location":{"city":"Santa Clara","country":"United States","region":"CA"},"units":{"distance":"km","pressure":"mb","speed":"km/h","temperature":"C"}, "wind":{"chill":"16","direction":"0","speed":"0"},"atmosphere":{"humidity":"67","pressure":"1014.8","rising":"0","visibility":"16.09"}, "astronomy":{"sunrise":"5:46 am","sunset":"8:23 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-wea.gif"}, "item":{"title":"Conditions for Santa Clara, CA at 8:52 am PDT","lat":"37.35","long":"-121.95","link":"http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA1018_c.html","pubDate":"Wed, 03 Jun 2015 8:52 am PDT","condition":{"code":"30","date":"Wed, 03 Jun 2015 8:52 am PDT","temp":"16","text":"Partly Cloudy"}, "description": "\n<img src=\"http://l.yimg.com/a/i/us/we/52/30.gif\"/><br />\n<b>Current Conditions:</b>\n<br />\nPartly Cloudy, 16 C<BR />\n<br /><b>Forecast:</b><BR />\nWed - AM Clouds/PM Sun. High: 22 Low: 13<br />\nThu - AM Clouds/PM Sun. High: 22 Low: 13<br />\nFri - AM Clouds/PM Sun. High: 24 Low: 14<br />\nSat - AM Clouds/PM Sun. High: 24 Low: 15<br />\nSun - Partly Cloudy. High: 26 Low: 16<br />\n<br />\n<a href=\"http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA1018_c.html\">Full Forecast at Yahoo! Weather</a><BR /><BR />\n(provided by <a href=\"http://www.weather.com\">The Weather Channel</a>)<br />\n", "forecast":[{"code":"30","date":"3 Jun 2015","day":"Wed","high":"22","low":"13","text":"AM Clouds/PM Sun"}, {"code":"30","date":"4 Jun 2015","day":"Thu","high":"22","low":"13","text":"AM Clouds/PM Sun"}, {"code":"30","date":"5 Jun 2015","day":"Fri","high":"24","low":"14","text":"AM Clouds/PM Sun"}, {"code":"30","date":"6 Jun 2015","day":"Sat","high":"24","low":"15","text":"AM Clouds/PM Sun"}, {"code":"30","date":"7 Jun 2015","day":"Sun","high":"26","low":"16","text":"Partly Cloudy"}], "guid":{"isPermaLink":"false","content":"USCA1018_2015_06_07_7_00_PDT"}}}}}
```

It's not very readable (by humans), so we can use a JSON formatter tool to "prettyify" it:

```
{  
  "query":{  
    "count":1,  
    "created":"2015-06-03T16:24:26Z",  
    "lang":"en-US",  
    "results":{  
      "channel":{  
        "title":"Yahoo! Weather - Santa Clara, CA",  
        "link":"http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA1018_c.html",  
        "description":"Yahoo! Weather for Santa Clara, CA",  
        "language":"en-us",  
        "lastBuildDate":"Wed, 03 Jun 2015 8:52 am PDT",  
        "ttl":60,  
        "location":{  
          "city":"Santa Clara",  
          "country":"United States",  
          "region":CA  
        },  
        "units":{  
          "distance":km,  
          "pressure":mb,  
          "speed":km/h,  
          "temperature":C  
        },  
        "wind":{  
          "chill":16,  
          "direction":0,  
          "speed":0  
        },  
        "atmosphere":{  
          "humidity":67,  
          "pressure":1014.8,  
          "rising":0,  
          "visibility":16.09  
        },  
        "astronomy":{  
          "sunrise":5:46 am,  
          "sunset":8:23 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-we  
a.gif"  
        },  
        "item":{  
          "title":Conditions for Santa Clara, CA at 8:52 am PDT,  
          "lat":37.35,  
          "lon":-122.42  
        }  
      }  
    }  
  }  
}
```

```
        "long":"-121.95",
        "link":"http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA1018_c.html",
        "pubDate":"Wed, 03 Jun 2015 8:52 am PDT",
        "condition":{
            "code":"30",
            "date":"Wed, 03 Jun 2015 8:52 am PDT",
            "temp":"16",
            "text":"Partly Cloudy"
        },
        "description":"\n<img src=\"http://l.yimg.com/a/i/us/we/52/30.gif\"/><br />\n<b>Current Conditions:</b><br />\nPartly Cloudy, 16 C<BR />\n<BR /><b>Forecast:</b><BR />\nWed - AM Clouds/PM Sun. High: 22 Low: 13<br />\nThu - AM Clouds/PM Sun. High: 22 Low: 13<br />\nFri - AM Clouds/PM Sun. High: 24 Low: 15<br />\nSat - AM Clouds/PM Sun. High: 24 Low: 14<br />\nSun - Partly Cloudy. High: 26 Low: 16<br />\nFull Forecast at Yahoo! Weather</a><BR /><BR />\n(provided by <a href=\"http://www.weather.com\">The Weather Channel</a>)<br />\n",
        "forecast": [
            {
                "code":"30",
                "date":"3 Jun 2015",
                "day":"Wed",
                "high":"22",
                "low":"13",
                "text":"AM Clouds/PM Sun"
            },
            {
                "code":"30",
                "date":"4 Jun 2015",
                "day":"Thu",
                "high":"22",
                "low":"13",
                "text":"AM Clouds/PM Sun"
            },
            {
                "code":"30",
                "date":"5 Jun 2015",
                "day":"Fri",
                "high":"24",
                "low":"14",
                "text":"AM Clouds/PM Sun"
            },
            {
                "code":"30",
                "date":"6 Jun 2015",
                "day":"Sat",
                "high":"24",
                "low":"15",
                "text":"AM Clouds/PM Sun"
            }
        ]
    }
```

```
        "text":"AM Clouds/PM Sun"
    },
    {
        "code":"30",
        "date":"7 Jun 2015",
        "day":"Sun",
        "high":"26",
        "low":"16",
        "text":"Partly Cloudy"
    }
],
"guid":{
    "isPermaLink":"false",
    "content":"USCA1018_2015_06_07_7_00_PDT"
}
}
}
}
```

JSON is how most REST APIs structure the response

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.

JSON has two types of basic structures: objects and arrays.

JSON objects are key-value pairs

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), you 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.

Identify the objects and arrays in the weatherdata API response

📝 ACTIVITY

Look at the response from the `weatherdata` endpoint of the 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 JavaScript 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

Mashape [provides some sample code in unirest](#) to parse and display the REST response on a web page. You could use it, but you could also use some auto-generated code from Postman to do pretty much the same thing.

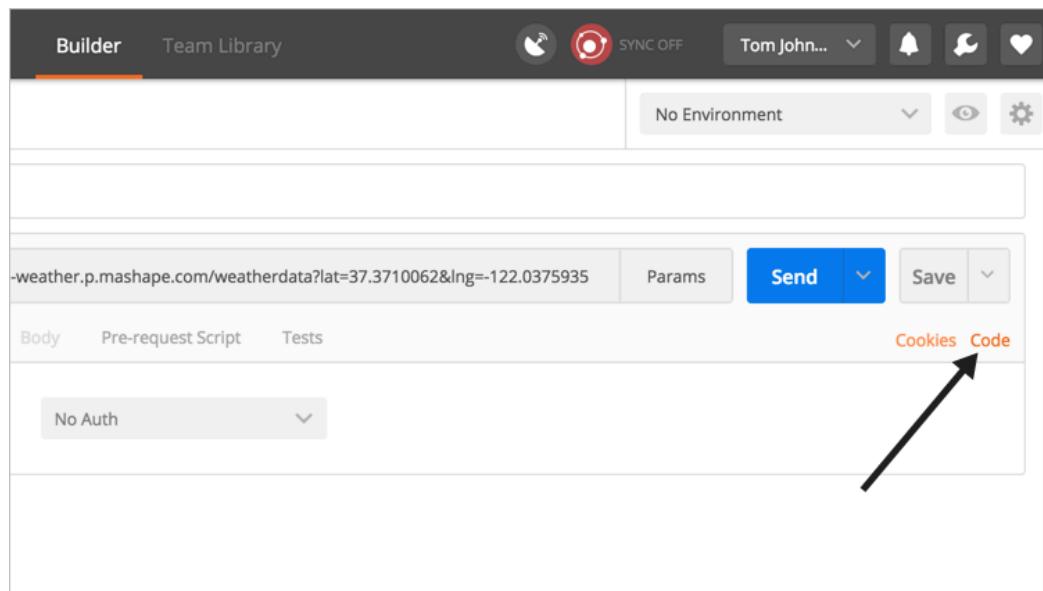
1. Start with a basic HTML template with jQuery referenced, like this:

```
<html>
<head>
<title>Sample Page</title>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
</head>
<body>

</body>
</html>
```

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

1. Open Postman and click the request to the `weatherdata` endpoint that you configured earlier.
2. Click the **Code** button.



3. Select **JavaScript > jQuery AJAX**.

```
Accept: application/json" -H "X-Mashape-Key: mshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p" -H "Cache-Control: no-cache" -H "Authorization: eeaef48db-959b-10dc-956e-6b318fb587f0" "https://simple-weather.p.mashape.com/weatherdata?lat=37.3710062&lng=-122.0375935"
```

4. Copy the code sample.

```
var settings = {
    "async": true,
    "crossDomain": true,
    "url": "https://simple-weather.p.mashape.com/weatherdata?lat=37.371
0062&lng=-122.0375935",
    "method": "GET",
    "headers": {
        "accept": "application/json",
        "x-mashape-key": "EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYr
p8p",
        "cache-control": "no-cache",
        "postman-token": "e41085c0-de85-0002-53ea-b64558309c68"
    }
}

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

5. Insert the Postman code sample between `<script>` tags in the same template you started building in step 1.

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

6. The Postman code sample needs one more parameter: `datatype`. Add `"datatype": "json"` as parameter in `settings`. Otherwise the object returned will be shown in text format rather than JSON.

Your final code should look like this:

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
</head>
<title>Sample Page</title>
<script>
var settings = {
    "async": true,
    "crossDomain": true,
    "url": "https://simple-weather.p.mashape.com/weatherdata?lat=37.3710062&lng=-122.0375935",
    "method": "GET",
    "dataType": "json",
    "headers": {
        "accept": "application/json",
        "x-mashape-key": "EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p",
        "cache-control": "no-cache",
        "postman-token": "e41085c0-de85-0002-53ea-b64558309c68"
    }
}

$.ajax(settings).done(function (response) {
    console.log(response);
});
</script>
<body>
    <h3>Open the JS Console (View > Developer > JavaScript Console) to see the object returned.</h3>
</body>
</html>
```

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

7. Start Chrome and open the JavaScript Console by going to **View > Developer > JavaScript Console**.
8. Open the weatherdata.html file in Chrome (**File > Open File**).

The page body will be blank, but the weatherdata response should be logged to the JavaScript console. You can inspect the payload by expanding the sections.

The screenshot shows the Chrome Developer Tools console tab with the 'Preserve log' option checked. It displays a hierarchical tree view of a JSON object. The object has properties like 'query', 'results', and 'image'. Under 'query', there are 'count', 'created', and 'lang' properties. Under 'results', there are 'channel', 'astronomy', 'atmosphere', and 'item' properties. The 'astronomy' property contains 'sunrise' and 'sunset' values. The 'atmosphere' property contains 'humidity', 'pressure', 'rising', and 'visibility' values. The 'item' property contains a long URL string.

```

Object {query: Object}
  query: Object
    count: 1
    created: "2015-08-30T21:06:14Z"
    lang: "en-US"
  results: Object
    channel: Object
      astronomy: Object
        sunrise: "6:35 am"
        sunset: "7:38 pm"
      __proto__: Object
    atmosphere: Object
      humidity: "47"
      pressure: "1016.5"
      rising: "2"
      visibility: "16.09"
    __proto__: Object
  image: Object
    height: "18"
    link: "http://weather.yahoo.com"
    title: "Yahoo! Weather"
    url: "http://l.yimg.com/a/i/brand/purplelogo//uh/us/news-wea.gif"
    width: "142"
  __proto__: Object
  item: Object
  condition: Object
  description: "<img src='http://l.yimg.com/a/i/us/we/52/30_nif'/><hr /><h3>Current Cond"
  
```

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

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

The AJAX method from jQuery

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.

Inspect the payload

Inspect the payload by expanding each of the sections in the Mashape weather API. Find the section that appears here: **object > query > results > channel > item > description**.

Access and print a specific JSON value

You'll notice that in the main content display of the weatherdata code, the REST response information didn't appear. It only appeared in the JavaScript Console. You need to use dot notation to access the JSON values you want.

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

Let's say you wanted to pull out the `description` part of the JSON response. Here's the dot notation you would use:

```
response.query.results.channel.item.description
```

The dot (`.`) after `response` (the name of the JSON payload) is how you access the values you want from the JSON object. 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.

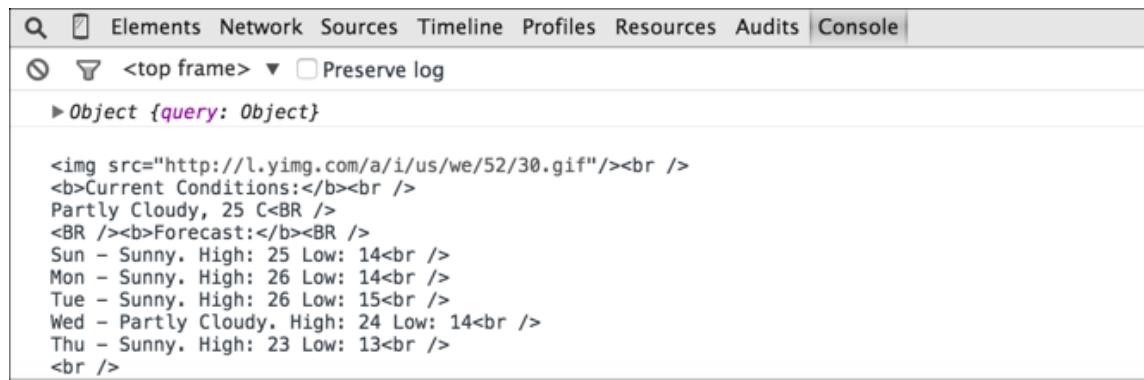
To pull out the description element from the JSON response and display it on the page, add this to your code sample, right below the `console.log(response)` line:

```
console.log(response.query.results.channel.item.description);
```

Your code should look like this:

```
.done(function (response) {
  console.log(response);
  console.log (response.query.results.channel.item.description);
});
```

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



```
<br />
<b>Current Conditions:</b><br />
Partly Cloudy, 25 C<BR />
<BR /><b>Forecast:</b><BR />
Sun - Sunny. High: 25 Low: 14<br />
Mon - Sunny. High: 26 Low: 14<br />
Tue - Sunny. High: 26 Low: 15<br />
Wed - Partly Cloudy. High: 24 Low: 14<br />
Thu - Sunny. High: 23 Low: 13<br />
<br />
```

Printing a JSON value to the page

Let's say you wanted to print part of the JSON (the `description` element) to the page. This involves a little bit of JavaScript or jQuery (to make it easier).

1. Add a named element to the body of your page, like this:

```
<div id="weatherDescription"></div>
```

2. Inside the tags of your `done` method, pull out the value you want into a variable, like this:

```
var content = response.query.results.channel.item.description;
```

3. Below this (same section) use the jQuery `append` method to append the variable to the element on your page:

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

This code says, find the element with the ID `weatherDescription` and append the `content` variable to it.

Your entire code should look as follows:

```
<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,
    "dataType": "json",
    "url": "https://simple-weather.p.mashape.com/weatherdata?lat=37.354108&lng=-121.955236",
    "method": "GET",
    "headers": {
        "accept": "application/json",
        "x-mashape-key": "APIKEY"
    }
}

$.ajax(settings)

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

    var content = response.query.results.channel.item.description;
    $("#weatherDescription").append(content);
});

</script>

<div id="weatherDescription"></div>
</body>
</html>
```

Here's [the result](#).

Dive into dot notation

In the [previous topic \(page 72\)](#), you accessed and printed a specific JSON value to the page. Let's dive into dot notation a little more.

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`.

It's important to 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.

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

Exercise with dot notation

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

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

```
<!DOCTYPE html>
<html>
<head>
<script src="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 right values for each item. For example, the word `green` should appear at the ID tag called `green`.

Answers are listed at the bottom of this page.

Showing wind conditions on the page

At the beginning of the course, I showed an example of embedding the wind speed and other details on a website. Now let's revisit this code example and see how it's put together.

Copy the following code into a basic HTML page, customize the `APIKEY` value, and view the page in the browser:

```
<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>Sample Query to get the wind</title>
<style>
    #wind_direction, #wind_chill, #wind_speed, #temperature, #speed {color: red; font-weight: bold;}
    body {margin:20px;}
</style>
</head>
<body>

<script>

function checkWind() {

    var settings = {
        "async": true,
        "crossDomain": true,
        "dataType": "json",
        "url": "https://simple-weather.p.mashape.com/weatherdata?lat=37.354108&lng=-121.955236",
        "method": "GET",
        "headers": {
            "accept": "application/json",
            "x-mashape-key": "APIKEY"
        }
    }

    $.ajax(settings)

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

        $("#wind_speed").append (response.query.results.channel.wind.speed);
        $("#wind_direction").append (response.query.results.channel.wind.direction);
        $("#wind_direction").append (" degrees");
        $("#wind_chill").append (response.query.results.channel.wind.chill);
        $("#temperature").append (response.query.results.channel.units.temperature);
        $("#speed").append (response.query.results.channel.units.speed);
    });
}

</script>
```

```
<button type="button" onclick="checkWind()" class="btn btn-danger weatherbutton">Check wind conditions</button>

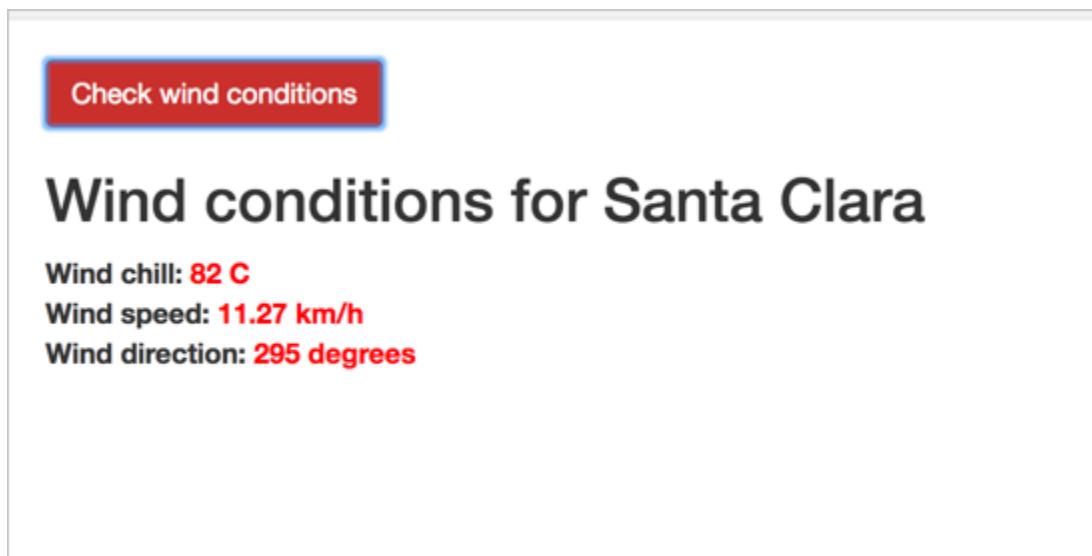
<h2>Wind conditions for Santa Clara</h2>

<b>Wind chill: </b><span id="wind_chill"></span> <span id="temperature"></span><br>
<b>Wind speed: </b><span id="wind_speed"></span> <span id="speed"></span></b>
<b>Wind direction: </b><span id="wind_direction"></span>
</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 `checkWind` runs, the wind chill, speed, and direction values are written to several ID tags on the page. Units for each of these values are also added to the page.
- Some minimal styling is added. Bootstrap is loaded to make the button styling.

When you load the page and click the button, the following should appear:



You can view the file [here](#).

Answers to “Exercise with dot notation activity”

Here's what your dot notation should look like:

```
var sarahjson = john.children[0].child1;  
var greenjson = john.eyes;  
var nikejson = john.shoes.brand;  
var goldenrodjson = john.favcolors[1];  
var jimmyjson = john.children[1].child2;
```

Documenting endpoints

A new endpoint to document	82
Documenting resource descriptions	85
Documenting endpoint definitions and methods	92
Documenting parameters	95
Documenting sample requests	102
Documenting sample responses	108
Documenting status and error codes	121
Documenting code samples	126
Putting it all together	136

A new endpoint to document

Until this point, you've been acting as a developer 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 you're a technical writer working with the Mashape weather API team. The team is asking you to document a new endpoint.

For this exercise, you could equally document a new endpoint for the Aeris Weather API, but since that API is already quite robust, we'll keep it simple and work with the more minimalist Mashape weather API.

You have a new endpoint to document

The project manager calls you over and says they have a new API for you to document for the next release. (By "API," the manager really just means a new endpoint to the existing API. Some APIs like [Alchemy API](#) even refer to each endpoint as an API.)

"Here's the wiki page that contains all the data," the manager says. The information is scattered and random on the wiki page. In reality, you probably wouldn't have all the information available that you need, but to facilitate our scenario (you can't ask the "team" questions about this fictitious new endpoint), the page will help.

ACTIVITY

It's now your task to sort through the information on this page and create documentation from it. Read through the wiki page to get a sense of the information. The upcoming topics will guide you through creating documentation for this new endpoint.

Most technical writers don't start from scratch with documentation projects. Engineers usually dump essential information onto an internal wiki page. However, the information on the wiki page will likely be incomplete, unnecessarily technical in places (like describing the database schema when users won't ever need this info), and have other issues. The info might include internal-only information (e.g., test logins, access protocols). Ultimately, the information will be written for other engineers on the same knowledge level. Your job as a technical writer will be to take this information and turn it into complete, accurate, usable information that meets your audience's goal.

The wiki page: "Surf Report API"

The new endpoint is `/surfreport/{beachId}`. This is for surfers who want to check things like tide and wave conditions to determine whether they should head out to the beach to surf. `{beachId}` is retrieved from a list of beaches on our site.

Optional parameters:

- Number of days: Max is 7. Default is 3. Optional.
- Units: imperial or metric. With imperial, you get feet and knots. With metric, you get centimeters and kilometers per hour. Optional.
- Time: time of the day corresponding to time zone of the beach you're inquiring about. Format is unix time, aka epoch. This is the milliseconds since 1970. Time zone is GMT or UTC. Optional.

If you include the hour, then you only get back the surf condition for the hour you specified. Otherwise you get back 3 days, with conditions listed out by hour for each day.

The response will include the surf height, the wind, temp, the tide, and overall recommendation.

Sample endpoint with parameters:

<https://simple-weather.p.mashape.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 grea
t"
        }
      }
    }
  ]
}
```

// ... the other hours of the day are truncated here.

Negative numbers in the tide represent incoming tide.

The report won't include any details about ripptide conditions.

Note that 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.

Essential sections in REST API documentation

In the next topics, you'll work on sorting this information out into eight common sections in REST API documentation:

- [Resource description \(page 85\)](#)
- [Endpoint definitions and methods \(page 92\)](#)
- [Parameters \(page 95\)](#)
- [Request example \(page 102\)](#)
- [Response example \(page 108\)](#)
- [Status and error codes \(page 121\)](#)
- [Code samples \(page 126\)](#)

Create the basic structure for the endpoint documentation

Open up a new text file and create sections for each of these elements.

Each of your endpoints should follow this same pattern and structure. A common template helps increase consistency and familiarity/predictability with how users consume the information.

Although there are automated ways to publish API docs, we're focusing on content rather than tools in this course. For the sake of simplicity, try just using a text editor (such as [Sublime Text](#)) and [Markdown syntax](#).

Documenting resource descriptions

Exactly what are the “things” that you access using a URL? Here are some of the terms used in different API docs:

- API calls
- Endpoints
- API methods
- Calls
- Resources
- Objects
- Services
- Requests

When it comes to the right terminology to describe these things (which I call “resources”), practices vary. Some docs get around the situation by not calling them anything explicitly.

You could probably choose the terms that you like best. My favorite is to use *resources* (along with *endpoint* for the URL. An API has various “resources” that you access through “endpoints.” The endpoint gives you access to a resource. The endpoint is the URL path (in this example, `/surfreport`). The information the endpoint interacts with, though, is a resource.

Some examples

Take look at [Mailchimp’s API for an example](#).

Quickly review all available resources for MailChimp API 3.0 with this reference overview.

The list of resources includes things like `authorized-apps`, `automations`, `batches`, and more.

With Mailchimp, a sample resource is “Automations.” This endpoint has several methods:

- `/automations` (GET)
- `/automations/{workflow_id}` (GET)
- `/automations/{workflow_id}/actions/pause-all-emails` (POST)
- `/automations/{workflow_id}/actions/start-all-emails` (POST)

Reference:

- Overview
- API Root
- Authorized Apps

Automations

- Emails
- Removed Subscribers

Batch Operations

Batch Webhooks

Campaign Folders

Campaigns

Conversations

E-commerce Stores

File Manager Files

File Manager Folders

Automations

Automation is a paid feature that lets you build a series of triggered emails that are sent to subscribers over a set period of time. Use the Automation API calls to manage Automation workflows, emails, and queues.

Subresources

Emails	Removed Subscribers
------------------------	-------------------------------------

Available methods

Read	Action
GET /automations	Get a list of Automations
GET /automations/{workflow_id}	Get information about a specific Automation workflow

In contrast, look at Twitter's API. In their [API Reference overview](#), they call them endpoint docs:

These are the REST API endpoint reference docs.

A sample endpoint reference doc is [GET statuses/retweets/:id](#). To access it, you use the Resource URL <https://api.twitter.com/1.1/statuses/retweets/:id.json>. They then list out “Resource Information.”

Twitter Developer Documentation

Docs / REST APIs / Reference Documentation / GET statuses/retweets/:id

Products & Services

- [Best practices](#)
- [API overview](#)
- [Websites](#)
- [Cards](#)
- [OAuth](#)
- [REST APIs](#)
- [API Rate Limits](#)
- [Rate Limits: Chart](#)
- [The Search API](#)
- [The Search API: Tweets by Place](#)

GET statuses/retweets/:id

Returns a collection of the 100 most recent retweets of the Tweet specified by the `id` parameter.

Resource URL

<https://api.twitter.com/1.1/statuses/retweets/:id.json>

Resource Information

Response formats	JSON
Requires authentication?	Yes
Rate limited?	Yes
Requests / 15-min window (user auth)	75

Here's the approach by [Instagram](#). Their doc calls resources “endpoints” in the plural – e.g., “Relationship endpoints,” with each endpoint listed on the relationship page.

The screenshot shows the Instagram API documentation. On the left, there's a sidebar with a search bar and links to Overview, Authentication, Login Permissions, Permissions Review, Sandbox Mode, Secure Requests, and Endpoints. Under Endpoints, there are sub-links for Users, Relationships (which is selected), Media, Comments, Likes, Tags, and Locations. The main content area has a title 'Relationship Endpoints' and a table of methods:

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

Below this, a specific endpoint is expanded: **GET /users/self/follows**. It shows the URL `https://api.instagram.com/v1/users/self/follows?access_token=ACCESS-TOKEN`, a 'RESPONSE' button, and detailed requirements: Scope: follower_list. It also specifies the required parameter **ACCESS_TOKEN** as 'A valid access token.'

The EventBrite API shows a list of endpoints, but when you go to an endpoint, the descriptions refer to them as objects. On the object's page you can see the variety of endpoints you can use with the object.

The screenshot shows the Eventbrite APIv3 Documentation for Events. At the top, there's a navigation bar with 'Browse Events', a user profile for 'Tom', 'Help', and a 'CREATE E' button. Below that, the main content area shows the 'Events' section with a title 'Events' and a 'GET /events/search/' endpoint. The description says: '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, there's a 'Parameters' table:

NAME	TYPE	REQUIRED	DESCRIPTION
q	string	No	Return events matching the given keywords.
since_id	string	No	Return events after this Event ID.
popular	boolean	No	Boolean for whether or not you want to only return popular results.
sort_by	string	No	Parameter you want to sort by -

I point out discrepancies with the terminology to reinforce the fact that the terms are somewhat non-standard. Still, you can't go wrong by referring to them as resources. A resource can have many different endpoints and methods you can use with it.

When you're writing documentation, it probably makes sense to group content by resources and then list the available endpoints for each resource on the resource's page, or as subpages under the resource.

This simple example with the Mashape Weather API, however, just has three different endpoints. There's not a huge reason to separate out endpoints by resource.

When it gets confusing to refer to resources by the endpoint

The Mashape Weather API is pretty simple, and just refers to the endpoints available. In this case, referring to the aqi endpoint or the air quality index resource doesn't make a huge difference. But with more complex APIs, using the endpoint path to talk about the resource can get problematic.

At one company I worked at (Badgeville), our endpoints looked like this:

```
// get all users  
api_site.com/{apikey}/users  
  
// get a specific user  
api_site.com/{apikey}/users/{userId}  
  
// get all rewards  
api_site.com/{apikey}/rewards  
  
// get a specific reward  
api_site.com/{apikey}/rewards/{rewardId}  
  
// get all rewards for a specific user  
api_site.com/{apikey}/users/{userId}/rewards  
  
// get a specific reward for a specific user  
api_site.com/{apikey}/users/{userId}/rewards/{rewardId}  
  
// get the rewards for a specific mission related to a specific user  
api_site.com/{apikey}/users/{userId}/rewards/{missionId}  
  
// get the rewards available for a specific mission  
api_site.com/{apikey}/missions/{missionid}/rewards
```

A rewards resource had various endpoints that returned different types of information related to rewards.

To say that you could use the rewards or missions endpoint wasn't always specific enough, because there were multiple rewards and missions endpoints.

It can get awkward referring to the resource by its endpoint path. For example, "When you call `/users/{userId}/rewards/{rewardId}`, you get a specific reward for a user. The `/users/{userId}/rewards/{rewardId}` endpoint takes several parameters..." It's a mouthful.

The same resource can have multiple endpoints

The [Box API](#) has a good example of how the same resource can have multiple endpoints and methods.

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

For the Membership object, as they call it, there are 7 different endpoints or methods you can call. Each of these methods lets you access the Membership object in different ways. Why call it an object? When you GET the Membership resource, the response is a JSON object.

Developers often use the term "call a method" when talking about using a method. If you consider the endpoints as HTTP methods, then you can call an API method.

Wait, I'm confused

You're probably thinking, wait, I'm a bit confused. Exactly what am I supposed to call the things I'm documenting in an API? My recommendation is to call them resources. In your table of contents, you might group all the resources under a larger umbrella called "API Reference."

But my point is that there is no standard practice here. The terminology varies, and this is one of those cases where everyone chooses their favorite term.

When describing the resource, start with a verb

Regardless of the terms you use, the description is usually brief, from 1-3 sentences, and often expressed as a fragment in the active tense.

Review the [surf report wiki page \(page 82\)](#) containing the information about the endpoint, and try to describe the endpoint in the length of one or two tweets (140 characters).

Here are some examples of resource descriptions:

[Delicious API](#)

[/v1/posts/update](#)

Check to see when a user last posted an item. Returns the last updated time for the user, as well as the number of new items in the user's inbox since it was last visited.

Use this before calling posts/all to see if the data has changed since the last fetch.

Foursquare API

Venue Menu

https://api.foursquare.com/v2/venues/VENUE_ID/menu

Returns menu information for a venue.

In some cases, menu information is provided by our partners. When displaying the information from a partner, you must attribute them using the attribution information included in the provider field. Not all menu information available on Foursquare is able to be redistributed through our API.

How I go about it

Here's how I went about creating the endpoint description. If you want to try crafting your own description of the endpoint first, and then compare yours to mine, go for it. However, you can also just follow along here.

I start by making a list of what the resource contains.

Surfreport

- shows surfing conditions about surf height, water temperature, wind, and tide
- must pass in a specific beach ID
- gives overall recommendation about whether to go surfing
- conditions are broken out by hour

Whenever I have a blank page, instead of immediately filling it with full sentences, I usually create a rough outline. Outlines prevent writers block with almost any writing project. After I have an outline, then I craft the sentences.

So now that I have a rough outline here, I craft the sentences:

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.

Critique the Mashape Weather API descriptions

Look over the descriptions of the three endpoints in the weather API. They're pretty short. For example, the aqi endpoint just says "Air Quality Index."

These descriptions are too short. But developers like concision. If shortening the surfreport description, you could also write:

/surfreport/{beachId}

Provides surf condition information.

Compare these descriptions with the endpoint descriptions from the [Aeris Weather API](#).

With Aeris Weather, the description for the forecasts endpoint is as follows:

Endpoint: forecasts

The forecasts endpoint/data set provides the core forecast data for US and international locations. Forecast information is available in daily, day/night intervals, as well as, custom intervals such as 3 hour or 1 hour intervals.

In summary, the description provides a 1-3 sentence summary of the information the resource contains.

Recognize the difference between reference docs versus user guides

One thing to keep in mind is the difference between reference docs and user guides/tutorials:

- **Reference guides:** Concise, bare-bones information that developers can quickly reference.
- **User guides/tutorials:** More elaborate detail about everything, including step-by-step instructions, code samples, concepts, and procedures.

With the description of `surfreport`, you might expand on this with much greater detail in the user guide. But in the reference guide, just provide a short description.

You could link the description to the places in the user guide where you expand on it in more detail. But since developers often write API documentation, they sometimes never write the user guide (as is the case with the Weather API in Mashape).

The description of the endpoint 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.

Documenting endpoints and methods

In the previous section, I noted the variation over terminology related to resources, with some doc sites calling the resources “endpoints.” Although some might call the whole topic “endpoint documentation,” the endpoint usually refers to a specific part in the API. The endpoint literally refers to the resource URL that you call, specifically, the last part of the resource URL (after the base path).

Varied terminology

As you might expect, the terms used for the endpoint vary as well. In addition to “endpoint,” you might see the following:

- Requests
- API methods
- Resource URLs
- URLs
- URL syntax

My preferred term is “endpoint.”

Often there’s no term used at all above the endpoint — you can just list it on the page, styled in a way that makes it obvious what it is.

The endpoint definition usually contains the end path only

When you describe the endpoint, it’s common to list the end path only (hence the nickname “endpoint”).

In our scenario, the endpoint/endpath is just `/surfreport/{beachId}`. You don’t have to list the full URL every time (which would be `https://simple-weather.p.mashape.com/surfreport{beachId}`).

Including the whole path distracts the user from focusing on the path that matters. (In your user guide, you usually explain the full code path in an introductory section.)

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 in another color to set it off:

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

If you set off your code block with `pre` tags (instead of backticks as is common with Github-flavored Markdown syntax), you can use `span` tags to set off specific elements in different colors, since angle brackets get processed as HTML. However, if you do use `pre` tags, you lose out on syntax highlighting, so it’s a tradeoff.

Curly braces are a convention that users will understand. In the above example, almost no URL uses curly braces in the actual path syntax, so the `{campaign_id}` is an obvious placeholder.

Another convention is to represent parameter values with a colon, like this:

```
/campaigns/:campaign_id/actions/send
```

You can see this convention in the [EventBrite API](#) and the [Aeris Weather API](#). But I'm not a fan of it.

In general, if the placeholder name is ambiguous as to whether it's a placeholder or not, clarify it.

You can list the method beside the endpoint

It's common to list the method (GET, POST, PUT, DELETE) next to the endpoint. 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 Content API. On the left sidebar, under the 'Comment Object' section, the 'Create Comment' endpoint is highlighted with a blue box and labeled 'POST'. The main content area shows the 'Create Comment' endpoint with a 'POST' button. Below it, a description states: 'Used to add a comment by the user to a specific file or comment (i.e. as a reply comment.)'. To the right, there are sections for 'Definition' (showing the URL `/comments`), 'Example Request' (with a curl command), and 'Example Response' (showing a 201 Created status and a JSON response object). The JSON response object includes fields like 'type', 'id', and 'message'.

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

Data Formats

Requesting data from the APIs

Unless otherwise specified, all of LinkedIn's APIs will return the information that you request in the XML data format.

GET <https://api.linkedin.com/v1/people/~>

sample response

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

Sometimes the method is referred to as the “verb.” GET, PUT, POST, and DELETE are all verbs or actions.

Multiple endpoints for the same resource

Some resources have multiple endpoints. For example, suppose you had two GET endpoints and one POST endpoint, all of which are highly related to the same resource. Different doc sites organize endpoints in various ways. Some list all the endpoints for the same resource on the same page. Others break them out into separate pages. Others put all the endpoints for all resources on the same page. 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).

In a later section on [design patterns \(page 187\)](#), I explore how [long pages \(page 190\)](#) are common pattern with developer docs, in part because they make content easily findable for developers using Ctrl + F.

Your turn to try: Write the endpoint definition for surfreport

📝 ACTIVITY

List out the endpoint definition and method for the surfreport/{beachId} endpoint.

Here’s my approach:

Endpoint definition

GET [surfreport/{beachId}](#)

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

Documenting parameters

Parameters offer ways to configure the endpoint. The parameters you pass with an endpoint affect the response.

Listing parameters in a table

Many times parameters are listed in a simple table or definition list like this:

Parameter	Required?	Data Type
format	Optional	String

Here's an example from Yelp's documentation:

Yelp Fusion
Search API

Request

Name	Method	Description
/v2/search	GET	Search for local businesses.

API v2

Get started	Note: at this time, the API does not return businesses without any reviews.		
API console			
Documentation			
Introduction	Name	Data Type	Required / Optional
Authentication	term	string	optional
Search API	limit	number	optional
Business API	offset	number	optional
Phone Search API	sort	number	optional
iPhone Apps			
Errors			
Code samples			

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, you should develop styles that make the values easily readable.

Four types of parameters

REST APIs have four types of parameters:

- **Path parameters:** Parameters that appear within the path of the endpoint, before the query string ([?](#))
- **Query string parameters:** Parameters that appear in the query string of the endpoint, after the [?](#).

- **Request body parameters:** Parameters that are included in the request body. Usually submitted as JSON.
- **Header parameters:** Parameters that are included in the request header. Usually header parameters relate to authorization.

The terms for each of these parameter types comes from the OpenAPI spec, which defines a formal specification that includes descriptions of each parameter type. Using industry standard terminology helps you develop a vocabulary to describe different elements of an API.

Data types indicate the format for the values

It's important to list the data type for each parameter — APIs may not process the parameter correctly if it's the wrong data type or wrong format. 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
- **object:** Key-value pairs in JSON format

There are more data types in programming, and if you have more specific data types, be sure to note 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."

Parameters should list allowed values

One of the problems with the Mashape Weather API is that it doesn't tell you which values are allowed for the latitude and longitude. If you type in coordinates for Nepal, for example, `28.3790654` and `81.8856707`, the response is `Not Supported - NA - NA`. Which cities are supported, and where does one look to see a list? This information should be made explicit in the description of parameters.

Parameter order doesn't matter

Often the parameters are added with a query string (`?`) at the end of the endpoint, and then 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, if the parameter is part of the actual endpoint path (not added in the query string), such as with `{beachId}` above, then you usually describe this value in the description of the endpoint itself.

Here's an example from Twilio:

The screenshot shows a REST API documentation page for the Lookups subdomain. The page includes a detailed description of the Lookup service, information about the Lookups Subdomain, and a Resource URI table. The table lists the URL as `lookups.twilio.com/v1/PhoneNumbers/{PhoneNumber}`. A note below the table specifies that `{PhoneNumber}` is the phone number being requested. To the right of the documentation is a code editor window showing a C# example for performing a phone lookup using the Twilio Lookups library.

```

1 // Download the twilio-csharp library from twilio.com/docs/csharp/install
2 using System;
3 using Twilio.Lookups;
4
5 class Example
6 {
7     static void Main(string[] args)
8     {
9         // Find your Account Sid and Auth Token at twilio.com/user/account
10        const string accountSid = "ACXXXXXXXXXXXXXXXXXXXXXX";
11        const string authToken = "your_auth_token";
12        var lookupsClient = new LookupsClient(accountSid, authToken);
13
14        // Look up a phone number in E.164 format
15        var phoneNumber = lookupsClient.GetPhoneNumber("+15108675309", t
16        Console.WriteLine(phoneNumber.Carrier.Type);
17        Console.WriteLine(phoneNumber.Carrier.Name);
18    }
19 }

```

The `{PhoneNumber}` value in `lookups.twilio.com/v1/PhoneNumbers/{PhoneNumber}` is described in the endpoint description rather than in another section that lists the query parameters.

Other important details about parameters are the maximum and minimum values allowed for the parameter, and whether the parameter is optional or required.

When you test an API, try running an endpoint without the required parameters, or with the wrong parameters. See what kind of error response comes back. Include that response in your response codes section. I get deeper with the importance of testing in [Testing your docs \(page 141\)](#).

Color coding parameter values

When you list the parameters in your endpoint, it can help to color code the parameters both in the table and in the endpoint definition. This makes it clear what's a parameter and what's not. Through color you create an immediate connection between the endpoint and the parameter definitions.

For example, suppose your endpoint definition is as follows:

`/service/myendpoint/user/{user}/bicycles/{bicycles}`

Follow through with this same color in your table describing the parameters:

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 URL.

Passing parameters in the JSON body

Frequently with POST requests, you submit a JSON object in the request body. This JSON object may be a lengthy list of key value pairs with multiple levels of nesting.

For example, the endpoint URL may be something simple, such as `/surfreport/{beachId}`. But in the body of the request, you might include a JSON object, like this:

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

This is known as a request body parameter.

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. But if you have a JSON object with multiple objects inside objects, numerous levels of nesting, and lengthy conditional data, it can be trickier. 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.

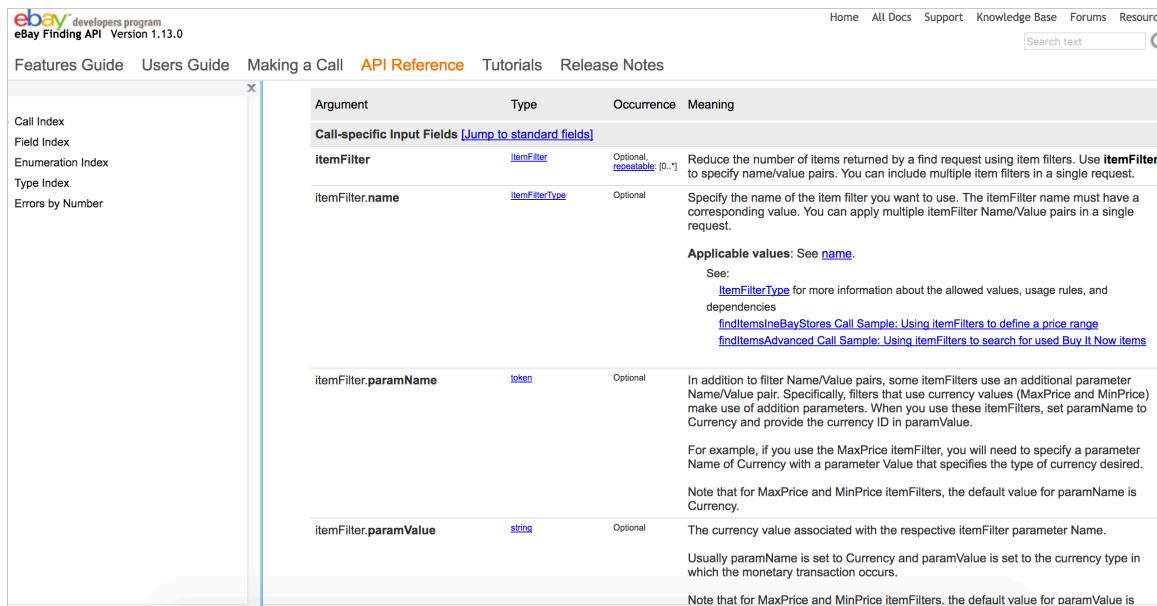
By all means, if the JSON object is relatively small, a table is probably your best option. But there are other approaches that designers have taken as well.

Take a look at eBay's [findItemsByProduct](#) endpoint.

The screenshot shows the eBay Developers Program API Reference page for the Version 1.13.0 of the eBay Finding API. The page has a navigation bar with links to Home, All Docs, Support, Knowledge Base, Forum, and a search bar. Below the navigation, there are links to Features Guide, Users Guide, Making a Call, API Reference (which is highlighted), Tutorials, and Release Notes.

The main content area is titled "Input". It contains a detailed description of the XML schema for the `<findItemsByProductRequest>` element. The schema includes definitions for `itemFilter`, `outputSelector`, `productId`, `affiliate`, and various pagination and sorting parameters. A note at the bottom of the input section says: "The box below lists all fields that could be included in the call request. To learn more about an individual field or its type, click its name in the box (or down to find it in the table below the box). See also [Samples](#)".

There's a table below the sample request that describes each parameter:



The screenshot shows the eBay Finding API Version 1.13.0 API Reference page. On the left, there's a sidebar with links like 'Features Guide', 'Users Guide', 'Making a Call', 'API Reference' (which is highlighted in orange), 'Tutorials', and 'Release Notes'. The main content area has a table titled 'Call-specific Input Fields' with a link 'Jump to standard fields'. The table has columns for 'Argument', 'Type', 'Occurrence', and 'Meaning'. It lists three parameters: 'itemFilter' (ItemFilter type, optional, repeatable: [0..*]), 'itemFilter.name' (ItemFilterType type, optional), and 'itemFilter.paramName' (token type, optional). The 'itemFilter.name' row includes a note about specifying item filter names and values, and links to 'ItemFilterType' and two call samples: 'findItemsInBayStores Call Sample: Using ItemFilters to define a price range' and 'findItemsAdvanced Call Sample: Using ItemFilters to search for used Buy It Now items'. The 'itemFilter.paramName' row includes notes about currency filters and links to 'MaxPrice' and 'MinPrice' item filters.

Argument	Type	Occurrence	Meaning
Call-specific Input Fields Jump to standard fields			
itemFilter	ItemFilter	Optional, repeatable: [0..*]	Reduce the number of items returned by a find request using item filters. Use ItemFilter to specify name/value pairs. You can include multiple item filters in a single request.
itemFilter.name	ItemFilterType	Optional	Specify the name of the item filter you want to use. The itemFilter name must have a corresponding value. You can apply multiple itemFilter Name/Value pairs in a single request.
itemFilter.paramName	token	Optional	In addition to filter Name/Value pairs, some itemFilters use an additional parameter Name/Value pair. Specifically, filters that use currency values (MaxPrice and MinPrice) make use of addition parameters. When you use these itemFilters, set paramName to Currency and provide the currency ID in paramValue. For example, if you use the MaxPrice itemFilter, you will need to specify a parameter Name of Currency with a parameter Value that specifies the type of currency desired. Note that for MaxPrice and MinPrice itemFilters, the default value for paramName is Currency.
itemFilter.paramValue	string	Optional	The currency value associated with the respective itemFilter parameter Name. Usually paramName is set to Currency and paramValue is set to the currency type in which the monetary transaction occurs. Note that for MaxPrice and MinPrice itemFilters, the default value for paramValue is

But the sample request also contains links to each of the parameters. When you click a parameter value in the sample request, you go to a page that provides more details about that parameter value, such as the [ItemFilter](#). 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

Is the display from the [Swagger UI \(page 323\)](#) any better?

The [Swagger UI](#) reads the OpenAPI specification document and displays it in the visual format that you see with examples such as the [Swagger Petstore](#).

The Swagger UI lets you toggle between an “Example Value” and a “Model” view for both responses and request body parameters.

The Example Value shows a sample of the syntax along with examples. When you click the Model (yellow box) in the [/Pet \(POST\) endpoint](#), Swagger inserts the content in the `body` parameter box. Here’s the Pet POST endpoint’s Example Value:

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

Now click **Model** (the grayed out text) and look at the view.

Name	Description
body * required (body)	Pet object that needs to be added to the store Example Value Model <div style="background-color: #e0e0e0; padding: 10px;"> Pet ↴ { id: integer (\$int64) category: Category > {...} name: string * example: doggie photoUrls: [string]* tags: [Tag]* { id: integer (\$int64) name: string }] status: string pet status in the store Enum: [available, pending, sold] } </div>
Responses	Responses

This view describes the various parts of the request, noting the data types and any descriptions in your OpenAPI spec. The model includes expand/collapse toggles with the values. The Petstore spec 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 \(page 259\)](#).

Conclusion

You can see that there's a lot of variety in documenting JSON and XML responses. There's no right way to document the parameters. 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, gnarly parameters, you may have to resort to custom styling and templates to present them clearly.

Construct a table to list the surfreport parameters

ACTIVITY

For our new surfreport endpoint, look through the parameters available and create a table similar to the one above."

List out the endpoint definition and method for the surfreport/{beachId} endpoint.

Here's what my table or definition list looks like:

Parameter	Required	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. With HTML, you can use a `colgroup` property to specify the `col width` for each column.

Documenting sample requests

Although you've already listed the endpoint and parameters, you should also include one or more sample requests that shows the endpoint integrated with parameters in an easy-to-understand way.

Example

In the CityGrid Places API, the basic places endpoint is as follows:

```
https://api.citygridmedia.com/content/places/v2/search/where
```

However, there are 17 possible query string parameters you can use with this endpoint. As a result, the documentation includes several sample requests show the parameters used with the endpoint:

Where Search Usage Examples	
Usage	URL
Find movie theaters in zip code 90045	https://api.citygridmedia.com/content/places/v2/search/where?type=movietheater&where=90045&publisher=test
Find Italian restaurants in Chicago using placement "sec-5"	https://api.citygridmedia.com/content/places/v2/search/where?what=restaurant&where=chicago,IL&tag=11279&placement=sec-5&publisher=test
Find hotels in Boston, viewing results 1-5 in alphabetical order	https://api.citygridmedia.com/content/places/v2/search/where?what=hotels&where=boston,ma&page=1&rpp=5&sort=alpha&publisher=test
Find pharmacies near the L.A. County Music Center, sorted by distance	https://api.citygridmedia.com/content/places/v2/search/where?what=pharmacy&where=135+N+Grand,LosAngeles,ca&sort=dist&publisher=test

These examples show several common combinations of the parameters. Adding multiple requests as samples 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 "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
```

If responses vary a lot, consider including multiple responses with the requests. 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.

Some sample requests in API don't show responses in the endpoint documentation. Instead, you click the request URL, which executes a GET request that doesn't require any authorization. You then see the response dynamically in the browser. The [Open Weather API](#) provides an example.

For example, if you click one of the "Examples of API calls," such as <http://samples.openweathermap.org/data/2.5/weather?q=London>, you see the response dynamically returned in the browser.



```
{  
  "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  
  }  
}
```

This approach is common and works well (for GET requests) when you can pull it off. Unfortunately, this approach makes it difficult to define the responses. (The CityGrid API documentation is detailed and does include information in later sections that describes the responses.)

API explorers provide interactivity with your own data

Many APIs have a feature called an API explorer. For example, here's a typical reference page for Spotify's API docs:

The screenshot shows the Spotify Developer API Console. On the left, there's a sidebar with links like 'Interactive Console', 'Albums' (selected), 'Artists', 'Tracks', 'Search', and 'Playlists'. The main area has a title 'Get an Album' and a table with details: Description 'Get an Album' (with a 'docs' link), Endpoint 'https://api.spotify.com/v1/albums/{id}', HTTP Method 'GET', and OAuth 'Required'. Below the table are input fields for 'Spotify Album ID' (containing '4aawyAB9vmqN3uQ7FjRGTy'), 'Market' (containing 'ES'), and 'OAuth Token' (with a 'GET OAUTH TOKEN' button). At the bottom are 'TRY IT' and 'FILL SAMPLE DATA' buttons, and a cURL command: `curl -X GET "https://api.spotify.com/v1/albums/" -H "Accept: application/json"`.

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

The screenshot shows the Flickr API Explorer for the 'photos.search' endpoint. It has a header with 'flickr', 'Sign Up', 'Explore', 'Create', and a search bar. Below is a title 'The App Garden' with links to 'Create an App', 'API Documentation', 'Feeds', and 'What is the App Garden?'. The main area has a title 'flickr.photos.search' and two sections: 'Arguments' and 'Useful Values'. The 'Arguments' section lists parameters: user_id (optional), tags (optional), tag_mode (optional), text (optional), min_upload_date (optional), max_upload_date (optional), min_taken_date (optional), max_taken_date (optional), and license (optional). The 'Useful Values' section lists 'Recent public photo IDs' and 'Popular public group IDs'.

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.

Here's another example from the New York Times API, which uses Lucybot (powered by Swagger) to handle the interactive API explorer features:

The screenshot shows the Books API documentation using the Swagger 2.0 interface. On the left, a sidebar lists various endpoints: GET_lists-format, GET_lists-best-sellers-history-json, GET_lists-names-format, GET_lists-overview-format, GET_lists-date-list-json, and GET_reviews-format. On the right, four main API endpoints are displayed with their descriptions and 'Try it out' buttons:

- GET /lists.{format}**
Best Seller List
Show details Try it out →
- GET /lists/best-sellers/history.json**
Best Seller History List
Show details Try it out →
- GET /lists/names.{format}**
Best Seller List Names
Show details Try it out →
- GET /lists/overview.{format}**
Best Seller List Overview
Show details Try it out →

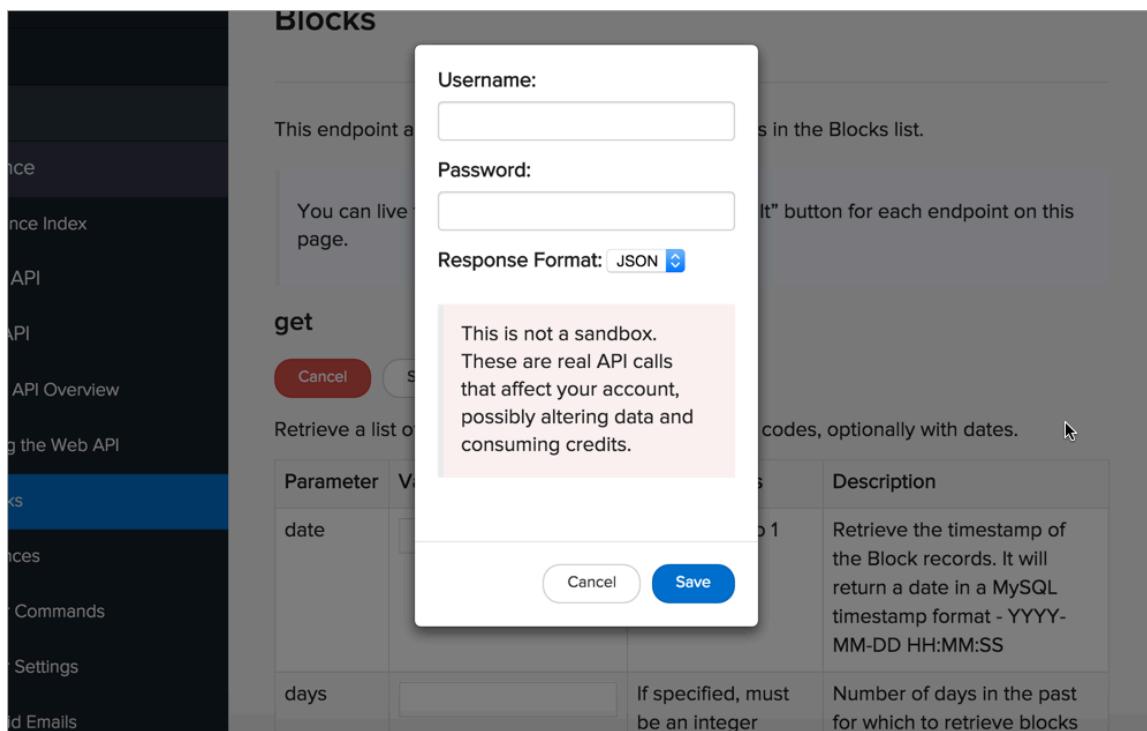
This example compels users to try out the endpoints to get a better understanding of the information they return.

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. With [IBM's Watson APIs](#), which use the Swagger UI, they removed the Try it out button.

In Sendgrid's API, they include a warning message to users before testing out calls with their API Explorer:



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.

```
{
  "meta": {
    "code": 200
  },
  "notifications": [
    {
      "type": "notificationTray",
      "item": {
        "unreadCount": 0
      }
    }
  ]
}
```

The [IBM Watson API](#) use [Swagger UI \(page 323\)](#) but they have suppressed the “Try it out” feature.

As far as integrating other 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) and [Readme.io](#) (which allows you to enter the details manually) can integrate API Explorer functionality directly into your documentation.

Document the sample request with the `surfreport/{beachId}` endpoint

ACTIVITY

Come back to the `surfreport/{beachId}` endpoint example. Create a sample request for it.

Here's mine:

Sample request

```
curl --get --include 'https://simple-weather.p.mashape.com/surfreport/123?units=imperial&days=1&time=1433772000' -H 'X-Mashape-Key: APIKEY' -H 'Accept: application/json'
```

Documenting sample responses

It's important to provide a sample response from the endpoint. This lets developers know if the endpoint contains the information they want, and how that information is labeled.

Example

Here's an example from Flattr's API. In this case, the response actually includes the response header as well as the response body:

The screenshot shows a user interface for documenting API responses. On the left, there is a vertical sidebar with the following menu items:

- Embedded buttons
- Auto-submit URL
- Flattr in feeds and HTML
- Partner site integration
- Tools
- Questions

The main panel is titled "Example response". It displays both the response header and the JSON response body. The header is as follows:

```
HTTP/1.1 200 OK
Content-Type: application/stream+json
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 999
X-RateLimit-Current: 1
X-RateLimit-Reset: 1342521939
```

The JSON response body is:

```
{
  "items": [
    {
      "published": "2012-01-04T10:07:12+01:00",
      "title": "pthulin flattred \"Acoustid\"",
      "actor": {
        "displayName": "pthulin",
        "url": "https://flattr.dev/profile/pthulin",
        "objectType": "person"
      },
      "verb": "like",
      "object": {
        "displayName": "Acoustid",
        "url": "https://flattr.dev/thing/459394/Acoustid",
        "objectType": "bookmark"
      },
      "id": "tag:flattr.com,2012-01-04:pthulin/flattr/459394"
    }
  ]
}
```

If the header information is important, include it. Otherwise, leave it out.

Define what the values mean in the endpoint response

Some APIs describe each item in the response, while others, perhaps because the responses are self-evident, omit the response documentation. In the Flattr example above, the response isn't explained. Neither is the response explained in [Twitter's API](#).

If the labels in the response are abbreviated or non-intuitive, however, you definitely should document the responses. Developers sometimes abbreviate the responses to increase performance by reducing the amount of text sent.

Additionally, if you're documenting some of the response items but not others, the doc will look inconsistent.

One of the problems with the Mashape Weather API is that it doesn't describe the meaning of the responses. If the air quality index is `25`, is that a good or bad value when compared to `65`? What is the scale based on?

Does each city/country define its own index? Does a high number indicate a poor quality of air or a high quality? How does air quality differ from air pollution? These are the types of answers one would hope to learn in a description of the responses.

Strategies for documenting nested objects

Many times the response contains nested objects (objects within objects). Here Dropbox represents the nesting with a slash. For example, `team/name` provides the documentation for the `name` object within the `team` object.

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).

Notice how the response values are in a monospaced font while the descriptions are in a regular font? This helps improve the readability.

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
          }
        ]
      }
    ]
  }
}
```

The indented approach with different levels of bullets can be an eyesore, so I recommend avoiding it.

In [Peter Gruenbaum's API tech writing course on Udemy](#), he also represents the nested objects using tables:

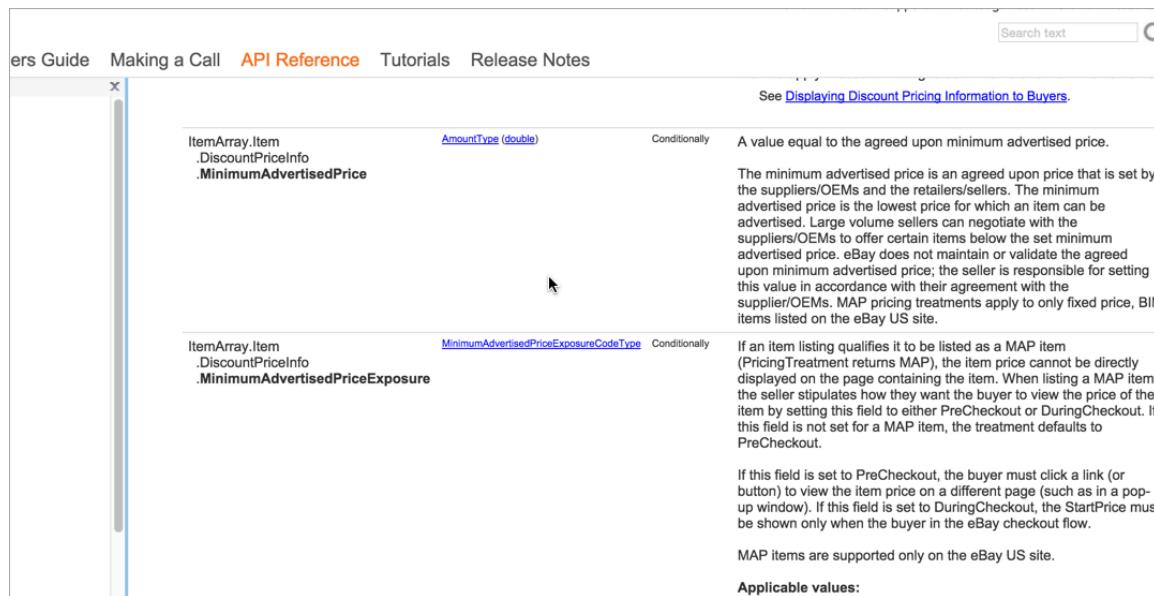
Song JSON Documentation

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.

eBay's approach is a little more unique:



The screenshot shows a detailed API reference page for the eBay API. The top navigation bar includes links for 'Users Guide', 'Making a Call', 'API Reference' (which is highlighted in orange), 'Tutorials', and 'Release Notes'. A search bar is located in the top right corner. The main content area displays a table with two rows, each describing a nested XML field. The first row is for `ItemArray.Item.DiscountPriceInfo.MinimumAdvertisedPrice`. It specifies the type as `AmountType (double)` and notes that it is conditionally applicable. The description explains that this field represents the minimum advertised price set by suppliers/OEMs and retailers/sellers, noting that large volume sellers can negotiate lower prices. The second row is for `ItemArray.Item.DiscountPriceInfo.MinimumAdvertisedPriceExposure`. It also specifies `AmountType (double)` and conditional applicability. The description details how this field affects MAP (Minimum Advertised Price) item listing, mentioning PreCheckout and DuringCheckout options, and notes that MAP items are supported only on the eBay US site. A link 'See Displaying Discount Pricing Information to Buyers.' is also present.

<code>ItemArray.Item .DiscountPriceInfo .MinimumAdvertisedPrice</code>	<code>AmountType (double)</code>	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.
<code>ItemArray.Item .DiscountPriceInfo .MinimumAdvertisedPriceExposure</code>	<code>MinimumAdvertisedPriceExposureCodeType</code>	Conditionally	If this field is set to 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 is in the eBay checkout flow. MAP items are supported only on the eBay US site. Applicable values:

For example, `MinimumAdvertisedPrice` is nested inside `DiscountPriceInfo`, which is nested in `Item`, which is nested in `ItemArray`. (Note also that this response is in XML instead of JSON.)

```

<?xml version="1.0" encoding="utf-8"?>
<FindPopularItemsResponse xmlns="urn:ebay:apis:eBLBaseComponents">
  <!-- Call-specific Output Fields -->
  <ItemArray> SimpleItemArrayType
    <Item> SimpleItemType
      <BidCount> int </BidCount>
      <ConvertedCurrentPrice> AmountType (double) </ConvertedCurrent
      <DiscountPriceInfo> DiscountPriceInfoType
        <MinimumAdvertisedPrice> AmountType (double) </MinimumAdvert
        <MinimumAdvertisedPriceExposure> MinimumAdvertisedPriceExpos
        <OriginalRetailPrice> AmountType (double) </OriginalRetailPri
        <PricingTreatment> PricingTreatmentCodeType </PricingTreatme
        <SoldOffeBay> boolean </SoldOffeBay>
        <SoldOneBay> boolean </SoldOneBay>
      </DiscountPriceInfo>
    <EndTime> dateTime </EndTime>
  
```

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.

A lot of APIs also return responses in XML, especially if the API is an older API. (Initially, XML was more popular than JSON, but now it's the reverse.) Some APIs give you the option of returning responses in either XML or JSON. If you're going to consume the API on a web page, JSON is probably much more popular because you can use JavaScript dot notation to grab the information you want.

Where to include the response

Some APIs collapse the response into a show/hide toggle to save space. Others put the response in a right column so you can see it while also looking at the endpoint description and parameters. Stripe's API made this tri-column design famous:

The screenshot shows the Stripe API documentation interface. On the left is a sidebar with a tree view of API endpoints: Introduction, Authentication, Errors, Pagination, Versioning, Expanding objects, Metadata, Idempotent requests, METHODS, Charges, Create a charge, Retrieve a charge, Update a charge, Capture a charge, List all charges, and Refunds. The main content area has three columns: 1. A left column listing endpoint details like 'Introduction' and 'Authentication'. 2. A middle column with detailed descriptions and parameters for specific endpoints. For example, under 'Charges', it says 'The charge object...' and provides links to 'Create a charge', 'Retrieve a charge', 'Update a charge', 'Capture a charge', and 'List all charges'. 3. A right column showing examples of API responses in various formats: curl, Ruby, Python, PHP, Java, Node, and Go. The 'curl' example shows a JSON object representing a payment source (a card). The JSON includes fields like 'paid', 'status', 'amount', 'currency', 'refunded', 'source' (with a detailed card object), and 'captured'.

A lot of APIs have modeled their design after Stripe's. (For example, see [Slate](#) or [readme.io](#).)

To represent the child objects, Stripe uses an expandable section under the parent (see the "Hide Child Attributes" link in the screenshot above).

I'm not sure that the tri-pane column is so usable. The original idea of the design was to allow you to see the response and description at the same time, but when the description is lengthy (such as is the case with `source`), it creates unevenness in the juxtaposition.

Many times in Stripe's documentation, the descriptions aren't in the same viewing area as the sample response, so what's the point of arranging them side by side? It splits the viewer's focus and causes more up and down scrolling.

Use realistic values in the response

The response should contain realistic values. If developers give you a sample response, make sure each of the possible items that can be included are shown. The values for each should be reasonable (not bogus test data that looks corny).

Format the JSON in a readable way

Use proper JSON formatting for the response. A tool such as [JSON Formatter and Validator](#) can make sure the spacing is correct.

Add syntax highlighting

If you can add syntax highlighting as well, definitely do it. One good Python-based syntax highlighter is [Pygments](#). This highlighter relies on "lexers" to indicate how the code should be highlighted. For example, some common lexers are `java`, `json`, `html`, `xml`, `cpp`, `dotnet`, and `javascript`. A non-python-based equivalent to Pygments is Rouge.

Since your tool and platform dictate the syntax highlighting options available, look for syntax highlighting options within the system that you're using. 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.

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 — it appears to be generated dynamically.

At one company I worked for, we had a test system we used to generate the responses. It was important that the test system had the right data to create good responses. You don't want a bunch of null or missing items in the response.

However, once the test system generated the responses, those responses were imported into the documentation through a script.

Creative approaches in documenting lengthy JSON responses

In addition to using standard tables to document JSON responses, you can also implement some more creative approaches.

The side-by-side approach

In Stripe's API documentation, the writers try to juxtapose the responses in a right side pane with the documentation in the main window.

		curl	Ruby	Python	PHP	Java	Node	Go
<code>livemode</code>	— boolean							
<code>amount</code>	Amount charged in cents positive integer or zero							
<code>captured</code>	If the charge was created without capturing, this boolean represents whether or not it is still uncaptured or has since been captured. boolean							
<code>created</code>	— timestamp							
<code>currency</code>	Three-letter ISO currency code representing the currency in which the charge was made. currency							
<code>paid</code>	<code>true</code> if the charge succeeded, or was successfully authorized for later capture. boolean							

```

"refunded": false,
"source": {
  "id": "card_165Fnw2eZvKYlo2CNypZm0H9",
  "object": "card",
  "last4": "4242",
  "brand": "Visa",
  "funding": "credit",
  "exp_month": 12,
  "exp_year": 2015,
  "country": "US",
  "name": null,
  "address_line1": null,
  "address_line2": null,
  "address_city": null,
  "address_state": null,
  "address_zip": null,
  "address_country": null,
  "cvc_check": "pass",
  "address_line1_check": null,
  "address_zip_check": null,
  "dynamic_last4": null,
  "metadata": {},
  "customer": null
},
"captured": false,
"balance_transaction": "txn_162Z1Z2eZvKYlo2CfodNCnZ",
"failure_message": null,
"failure_code": null,
"amount_refunded": 0,
"statement_descriptor": null
}

```

The idea is that you can see both the description and a sample response at the same time, and just scroll down.

However, the description doesn't always line up with the sample response. (In some places, child attributes are collapsed to save space.) I'm not sure why some items (such as `livemode`) aren't documented.

The no-need-for-descriptions approach

Some sites, like Twitter's API docs, don't seem to describe the items in the JSON response at all. Looking at this [long response for the post status/retweet endpoint](#) in Twitter's API docs, there isn't even an attempt to describe what all the items mean. Maybe they figure most of the items in the response are self-evident?

Example Result

```
{  
    "truncated": false,  
    "retweeted": false,  
    "id_str": "243149503589400576",  
    "coordinates": null,  
    "in_reply_to_screen_name": null,  
    "in_reply_to_status_id_str": null,  
    "geo": null,  
    "in_reply_to_status_id": null,  
    "contributors": null,  
    "source": "\u003Ca href=\"http://jason-costa.blogspot.com\"  
rel=\"nofollow\"\u003EMy Shiny App\u003C/a\u003E",  
    "in_reply_to_user_id_str": null,  
    "created_at": "Wed Sep 05 00:52:13 +0000 2012",  
    "favorited": false,  
    "entities": {  
        "user_mentions": [{
```

Theoretically, each item in the JSON response should be a clearly chosen word that represents what it means in an obvious way. However, to reduce the size and increase the speed of the response, developers often resort to shorter terms or use abbreviations. The shorter the term, the more it needs accompanying documentation.

In one endpoint I documented, the response included about 20 different two-letter abbreviations. I spent days tracking down what each abbreviation meant. Many developers didn't even know what the abbreviations meant.

The RAML API Console approach

When you use [RAML](#) to document endpoints with JSON objects in the request body, the RAML API Console output looks something like this:

BODY

application/json

[Hide Schema](#)

```
{
  "$schema": "http://json-schema.org/draft-03/schema",
  "type": "object",
  "properties": {
    "name": {
      "description": "The new name for the file.",
      "type": "string"
    },
    "description": {
      "description": "The new description for the file.",
      "type": "string"
    },
    "parent": {
      "description": "The parent folder of this file.",
      "type": "object"
    },
    "id": {
      "description": "The ID of the parent folder.",
      "type": "string"
    },
    "shared_link": {
      "description": "An object representing this item's shared link.",
      "type": "object"
    },
    "access": {
      "description": "The level of access required for this shared link.",
      "type": ["open", "company", "collaborators"]
    },
    "unshared_at": {
      "description": "The day that this link should be disabled at.",
      "type": "timestamp"
    },
    "permissions": {
      "description": "The set of permissions that apply to this link.",
      "type": "object"
    },
    "permissions.download": {
      "description": "Whether this link allows downloads.",
      "type": "boolean"
    },
    "permissions.preview": {
      "description": "Whether this link allows previews.",
      "type": "boolean"
    }
  }
}
```

Here each body parameter is a named JSON object that has standard values such as `description` and `type`. While this looks a little cleaner initially, it's also somewhat confusing. The actual request body object won't contain `description` and `type` parameters like this, nor would it contain the `schema`, `type`, or `properties` keys either.

The problem with RAML is that it tries to describe a JSON structure using a JSON structure itself, but the JSON structure of the description doesn't match the JSON structure it describes, so it's confusing.

Further, this approach doesn't provide an example in context, which is what usually clarifies the data for the user.

Custom-styled tables

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.

To the right of the JSON definitions is a code sample with real values. When you select a value, both the element in the table and the element in the code sample highlight at the same time.

Attribute Details		Example json GET response
UID	Guid (36)	
Name	String (30)	
DisplayID	String (6)	
Classification	Description	Classification
Type	Account code format includes separator ie 1-1100	
Number	CountType integer (4)	
Description	String (255)	
ParentAccount		
UID	Guid (36)	
Name	String (6)	
DisplayID	String (6)	
Parent account code format includes separator ie 1-1000		
URI	String	
IsActive	Boolean	
TaxCode		
UID	Guid (36)	

```

"UID" : "eb043b43-1d66-472b-a6ee-ad48def81b96",
"Name" : "Business Bank Account #2",
"DisplayID" : "1-1120",
"Classification" : "Asset",
"Type" : "Bank",
"Number" : 1120,
"Description" : "Bank account clearwtr",
"ParentAccount" : {
    "UID" : "f5cc9506-3472-4227-8c45-7a95c322c38b",
    "Name" : "Bank Accounts",
    "DisplayID" : "1-1100",
    "URI" : "{domain}/{cf_guid}/GeneralLedger/Account/f5cc9506-3472-4227-8c45-7a95c322c38b"
},
"IsActive" : true,
"TaxCode" : {
    "UID" : "94966872-b140-4da2-bc43-5dc74f33a09",
    "Code" : "N-T",
    "URI" : "{domain}/{cf_guid}/GeneralLedger/TaxCode/94966872-b140-4da2-bc43-5dc74f33a09"
},
"Level" : 4,
"OpeningBalance" : 100000.00,
"CurrentBalance" : 5000.00.

```

If you have long JSON objects like this, a custom table with different classes applied to different levels might be the only truly usable solution. It 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, and there isn't any interactivity to try out the endpoints. Still, if you have long JSON objects, it might be worth it.

Create a sample response in your `surfreport/{beachId}` endpoint

For your `surfreport/{beachId}` endpoint, create a section that shows the sample response. Look over the response to make sure it shows what it should.

Here's what mine looks like:

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."  
        }  
      }  
    }  
  ]  
}
```

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.

Response item	Description
{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 (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.
{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 feet 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.

*Because this is a fictitious endpoint, I'm making the descriptions up.

Documenting status and error codes

Status and error codes are essential for understanding responses from requests.

Sample status code in cURL header

Remember when we submitted the cURL call back in [an earlier lesson \(page 46\)](#)? We submitted a cURL call and specified that we wanted to see the response headers (`--include` or `-i`):

```
curl --get -include 'https://simple-weather.p.mashape.com/weather?lat=37.3
54108&lng=-121.955236' \-H 'X-Mashape-Key: APIKEY' \
-H 'Accept: text/plain'
```

The response, including the header, looked like this:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Fri, 16 Jun 2017 21:48:55 GMT
Server: Mashape/5.0.6
Via: 1.1 vegur
X-Powered-By: Express
Content-Length: 41
Connection: keep-alive

30 c, Sunny at Santa Clara, United States
```

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 or not because you get back something in the 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 Mashape Weather 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](#).

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. Twitter's API has a good example of the possible status and error codes you will receive when making requests:

The screenshot shows the Twitter API Documentation Overview page. On the left sidebar, there's a navigation menu with links like 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.

Error Codes & Responses

HTTP Status Codes

The Twitter API attempts to return appropriate [HTTP status codes](#) for every request.

Code	Text	Description
200	OK	Success!
304	Not Modified	There was no new data to return.
400	Bad Request	The request was invalid or cannot be otherwise served. An accompanying error message will explain further. In API v1.1, requests without authentication are considered invalid and will yield this response.

[Authentication credentials](#)

In contrast, with the Flickr API, each “method” (endpoint) lists error codes:

The screenshot shows the Flickr API documentation. At the top, there are buttons for Sign Up, Explore, Create, Upload, and Photos. Below the header, there's a note about photo elements and comments. The main content is titled "Error Codes" and 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.

API Explorer

[API Explorer : flickr.galleries.getPhotos](#)

Either location has merits. Generally, I think status codes should be included with each endpoint. But if you have a lot of status codes, it might make sense to create a single, centralized pages with a table that lists them.

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 codes. In particular, if developers have created special status codes for the API, highlight these in the documentation.

For example, if you exceed the rate limit for a specific call, the API might return a special status code. You would especially need to document this custom code. Listing out all the error codes is a reference section in the "Troubleshooting" topic of your API documentation.

When endpoints have specific status codes

In the Flattr API, sometimes endpoints return particular status codes. For example, when you "Check if a thing exists," the response includes `HTTP/1.1 302 Found` when the object is found. This is a standard HTTP response. If it's not found, you see a 404 status code.

The screenshot shows a section of the Flattr API documentation. It includes examples of responses for a non-existent URL and a request to lookup an autosubmit URL.

Example response when url was not found

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 999
X-RateLimit-Current: 1
X-RateLimit-Reset: 1342521939

{
  "message": "not_found",
  "description": "No thing was found"
}
```

Example request to lookup a autosubmit URL

The lookup resource can now lookup autosubmit URLs, you will need to url encode the data you pass into the `url` parameter. For more information check out the [auto submit documentation](#).

```
GET https://api.flattr.com/rest/v2/things/lookup/?url=http://flattr.co
```

Example response to autosubmit URL

If the status code is specific to a particular endpoint, you can include it with that endpoint's documentation.

Alternatively, you can have a general status and error codes page that lists all possible codes for all the endpoints. For example, with the Dropbox API, the writers list out the error codes related to the API:

OAuth 1.0 /request_token /authorize /access_token	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.																				
Standard API errors																					
OAuth 2.0 /authorize /token /token_from_oauth1	<table border="1"> <thead> <tr> <th>Code</th><th>Description</th></tr> </thead> <tbody> <tr> <td>400</td><td>Bad input parameter. Error message should indicate which one and why.</td></tr> <tr> <td>401</td><td>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.</td></tr> <tr> <td>403</td><td>Bad OAuth request (wrong consumer key, bad nonce, expired timestamp...). Unfortunately, re-authenticating the user won't help here.</td></tr> <tr> <td>404</td><td>File or folder not found at the specified path.</td></tr> <tr> <td>405</td><td>Request method not expected (generally should be GET or POST).</td></tr> <tr> <td>429</td><td>Your app is making too many requests and is being rate limited. 429s can trigger on a per-app or per-user basis.</td></tr> <tr> <td>503</td><td>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.</td></tr> <tr> <td>507</td><td>User is over Dropbox storage quota.</td></tr> <tr> <td>5xx</td><td>Server error. Check DropboxOps.</td></tr> </tbody> </table>	Code	Description	400	Bad input parameter. Error message should indicate which one and why.	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.	403	Bad OAuth request (wrong consumer key, bad nonce, expired timestamp...). Unfortunately, re-authenticating the user won't help here.	404	File or folder not found at the specified path.	405	Request method not expected (generally should be GET or POST).	429	Your app is making too many requests and is being rate limited. 429s can trigger on a per-app or per-user basis.	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.	507	User is over Dropbox storage quota.	5xx	Server error. Check DropboxOps .
Code	Description																				
400	Bad input parameter. Error message should indicate which one and why.																				
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.																				
403	Bad OAuth request (wrong consumer key, bad nonce, expired timestamp...). Unfortunately, re-authenticating the user won't help here.																				
404	File or folder not found at the specified path.																				
405	Request method not expected (generally should be GET or POST).																				
429	Your app is making too many requests and is being rate limited. 429s can trigger on a per-app or per-user basis.																				
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.																				
507	User is over Dropbox storage quota.																				
5xx	Server error. Check DropboxOps .																				
Access tokens /disable_access_token																					
Dropbox accounts /account/info																					
Files and metadata /files (GET) /files_put																					
OAuth 1.0																					

In particular, you should look for codes that return when there is an error, since this information helps developers troubleshoot problems.

You can run some of the cURL calls you made earlier (this time adding `-i`) and looking at the HTTP status code in the response.

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

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 they 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.

Deciding which languages to show code samples in

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 code snippets in a variety of languages. Here's an example from Evernote's API documentation:

	<ul style="list-style-type: none"> ◦ The GUID of the note you'd like to share. ◦ The ID of the Shard that houses the note to be shared. ◦ A valid authentication token or developer token. ◦ Initialized instances of <code>NoteStore.Client</code> and <code>UserStore.Client</code> <p>The Shard ID can be determined at runtime by querying the UserStore:</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">Python</td> <td style="padding: 2px;">Objective-C</td> <td style="padding: 2px;">Ruby</td> <td style="padding: 2px;">PHP</td> <td style="padding: 2px;">Java</td> <td style="padding: 2px;">Node.js</td> </tr> </table> <pre style="font-family: monospace; font-size: 0.9em; margin-top: 10px; padding: 10px;"> 1 def getUserShardId(authToken, userStore): 2 """ 3 Get the User from userStore and return the user's shard ID 4 """ 5 try: 6 user = userStore.getUser(authToken) 7 except (Errors.EDAMUserException, Errors.EDAMSystemException), e: 8 print "Exception while getting user's shardID:" 9 print type(e), e 10 return None 11 12 if hasattr(user, 'shardId'): 13 return user.shardId 14 return None </pre> <p style="margin-top: 10px;"> main.py hosted with ❤ by GitHub view raw </p> </div> <p style="margin-top: 20px;">Assuming all of that is in place, sharing a note is actually quite simple. By calling</p>	Python	Objective-C	Ruby	PHP	Java	Node.js	<p>Error Hand The Sandb</p> <p>Authenticat Developer OAuth Permission Revocation</p> <p>Rate Limit Overview Best Practi</p> <p>Notes Creating N Sharing N Reminders Read-only ENML Note Links Application</p> <p>Notebook Sharing N App Noteb</p>
Python	Objective-C	Ruby	PHP	Java	Node.js			

And another from Twilio:

The screenshot shows the Twilio Docs Examples page. At the top, there's a navigation bar with links to Quickstart, How-Tos, Helper Libraries, API Docs (which is the active tab), and Security. Below the navigation bar, there's a section titled "Examples". Under "Examples", there are two sections: "Example 1" and "Example 2". Each example has a code snippet and a set of language buttons below it.

Example 1

Make a call from 415-867-5309 to 415-555-1212, POSTing to <http://www.myapp.com/myhandler.php>

Code snippet (Node.js):

```
// Download the Node helper Library from twilio.com/docs/node/install
// These vars are your accountSid and authToken from twilio.com/user/account
var accountSid = 'AC3094732a3c49700934481add5ce1659';
var authToken = "{{ auth_token }}";
var client = require('twilio')(accountSid, authToken);

client.calls.create({
  url: "http://demo.twilio.com/docs/voice.xml",
  to: "+14155551212",
  from: "+14158675309"
}, function(err, call) {
  process.stdout.write(call.sid);
});
```

Language buttons for Example 1: JSON, XML, PHP, Python, C#, Java, Ruby, **Node.js**

Example 2

Make a call from 415-867-5309 to a [Twilio Client](#) named `tommy`. Twilio will POST to <http://www.myapp.com/myhandler.php> to fetch TwiML to handle the call.

Language buttons for Example 2: JSON, XML, PHP, Python, C#, Java, Ruby, **Node.js**

However, don't feel so intimidated by this smorgasbord of code samples. Some API doc tools might actually automatically generate these code samples because the patterns for making REST requests in different programming languages follow a common template. This is why many APIs decide to provide one code sample (usually in cURL) and let the developer extrapolate the format in his or her own programming language.

Auto-generating code samples

You can auto-generate code samples from both Postman and Paw, if needed.

Paw has more than a dozen code generator extensions:

The screenshot shows the 'Extensions' section of the Paw application. At the top, there's a header with the Paw logo and a menu icon. Below the header is a rocket ship icon and the word 'Extensions'. A sub-header text reads: 'Supercharge Paw using extensions! Either you want to have generated client code for your favorite language, import 3rd party file formats or compute dynamic values, an extension is probably here for you.' Below this, a note says 'You can also [make your own](#).'. There are four tabs at the top: 'All Extensions' (disabled), 'Code Generators' (selected, indicated by a blue border), 'Dynamic Values', and 'Importers'. A search bar below the tabs contains the placeholder 'Search Extensions...'. The main content area lists two extensions:

- API Blueprint Generator**: Paw extension providing support to export API Blueprint as a code generator.
- Betamax.py Generator**: Generates Betamax.py requests from Paw.

Once you install them, generating a code sample is a one-click operation:

This screenshot shows a dropdown menu titled 'HTTP' containing various code generator options. The 'JavaScript (jQuery)' option is highlighted with a blue selection bar and a checkmark icon. Other listed options include API Blueprint Generator, Betamax Generator, cURL, Go (HTTP), HTTPPie, Java (Apache HttpClient via Fluent API), Objective-C (NSURLConnection), Objective-C (NSURLSession), PHP (cURL), PHP (Guzzle), Python (Requests), Ruby (Net::HTTP), Swift (Alamofire), Swift (NSURLSession), WordPress, Amazon S3 String To Sign, OAuth 1 Signature Base, and Find More Code Generators...

The Postman app has most of these code generators built in by default.

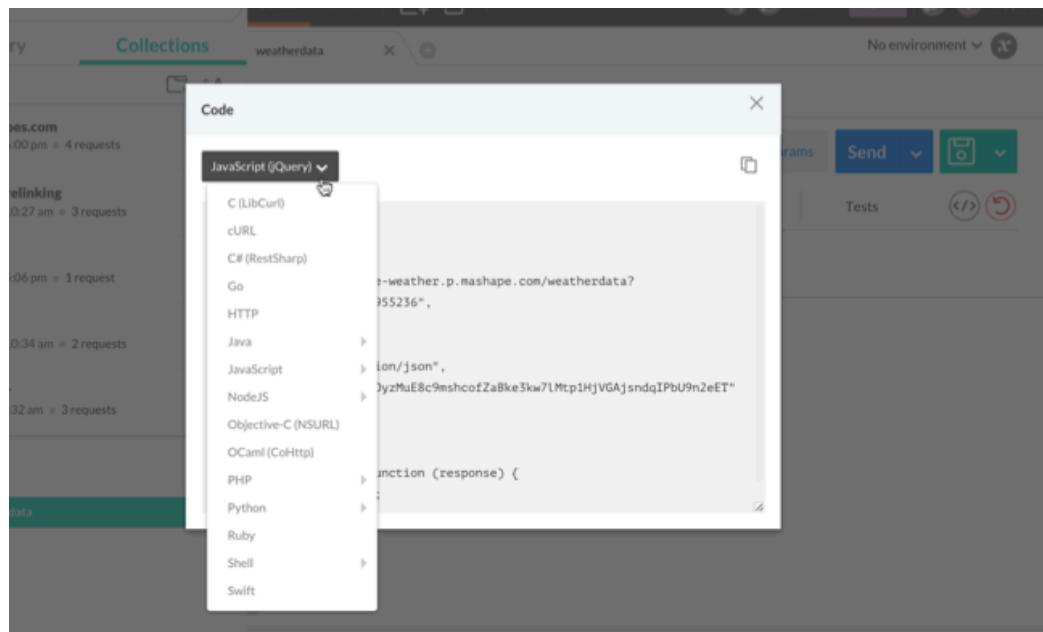
Although these code generators are probably helpful, they may or may not work for your API. Always review code samples with developers. In most cases, developers supply the code samples for the documentation, and technical writers briefly comment on the code samples.

Generate a JavaScript code sample from Postman

We covered some of this material earlier in more depth, so here I just cover it more briefly.

To generate a JavaScript code snippet from Postman:

1. Configure a weatherdata request in Postman (or select one you've saved).
2. Below the Send button, click the **Generate Code Snippets** button.
3. In the dialog box that appears, browse the available code samples using the drop-down menu. Note how your request data is implemented into each of the different code sample templates.
4. Select the **JavaScript > jQuery AJAX** code sample:



5. Copy the content by clicking the **Copy** button.

This is the JavaScript code that you can attach to an event on your page.

Implement the JavaScript code snippet

You usually don't need to show the code sample on a working HTML file, but if you want to show users code they can make work in their own browsers, you can do so.

1. Create a new HTML file with the basic HTML elements:

```
<!DOCTYPE html>
<head>
<title>My sample page</title>
</head>
<body>

</body>
</html>
```

2. Insert the JavaScript code you copied inside some `script` tags inside the `head` :

```
<!DOCTYPE html>
<head>
<script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
<script>
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://simple-weather.p.mashape.com/weatherdata?lat=37.354
108&lng=-121.955236",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "APIKEY"
  }
}

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

</body>
</html>
```

3. The Mashape Weather API requires the `dataType` parameter, which Postman doesn't automatically include. Add `"dataType": "json"`, in the list of `settings` :

```
<!DOCTYPE html>
<head>
<script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
<script>
var settings = {
    "async": true,
    "crossDomain": true,
    "dataType": "json",
    "url": "https://simple-weather.p.mashape.com/weatherdata?lat=37.354
108&lng=-121.955236",
    "method": "GET",
    "headers": {
        "accept": "application/json",
        "x-mashape-key": "APIKEY"
    }
}

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

4. This code uses the `ajax` method from jQuery. The parameters are defined in a variable called `settings` and then passed into the method. The `ajax` method will make the request and assign the response to the `done` method's argument (`response`). The `response` object will be logged to the console.
5. Open the file up in your Chrome browser.
6. Open the JavaScript Developer Console by going to **View > Developer > JavaScript Console**. Refresh the page.

You should see the object logged to the console.

```

Object {
  query: Object {
    count: 1
    created: "2015-08-20T14:36:40Z"
    lang: "en-US"
  }
  results: Object {
    channel: Object {
      astronomy: Object {
        sunrise: "6:27 am"
        sunset: "7:53 pm"
      }
      __proto__: Object
      atmosphere: Object {
        description: "Yahoo! Weather for Santa Clara, CA"
      }
      image: Object
      item: Object
      language: "en-us"
      lastBuildDate: "Thu, 20 Aug 2015 6:52 am PDT"
      link: "http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara,_CA/*http://weather.yahoo.net/weather/us/rd/santaclara.xml"
    }
  }
}

```

Let's say you wanted to pull out the `sunrise` time and append it to a tag on the page. You could do so like this:

```

<!DOCTYPE html>
<head>
<script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
<script>
var settings = {
  "async": true,
  "crossDomain": true,
  "dataType": "json",
  "url": "https://simple-weather.p.mashape.com/weatherdata?lat=37.354108&lon=-121.955236",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "APIKEY"
  }
}

$.ajax(settings).done(function (response) {
  console.log(response);
  $("#sunrise").append(response.query.results.channel.astronomy.sunrise);
});

</script>
</head>
<body>
<h2>Sunrise time</h2>
<div id="sunrise"></div>
</body>
</html>

```

This code uses the `append` method from jQuery to assign a value from the response object to the `sunrise` ID tag on the page.

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, when I worked at Badgeville, we had both a REST API and a JavaScript SDK. Because JavaScript was the target language developers were working in, Badgeville 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. SDKs are usually specific to a particular language platform. Sometimes they are GUI tools. If you have an SDK, you'll want to make more detailed code samples showing how to use the SDK.

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 the Wunderground Weather API Documentation page. At the top, there's a navigation bar with links for Weather, Maps & Radar, Severe Weather, Photos & Video, Community, News, and Climate. Below the navigation is a search bar and a sign-in link. The main content area has a blue header "DOCUMENTATION". Below it, a navigation bar includes links for API Home, Pricing, Featured Applications, Documentation (which is active), and Forums. On the left, there's a sidebar titled "API Table of Contents" with a tree view of the Weather API categories: WunderMap Layers (Radar, Satellite, Radar + Satellite), Data Features (alerts, almanac, astronomy, conditions, currenthurricane, forecast, forecast10day, geolookup, history, hourly, hourly10day, planner, rawtide). The main content area features a large title "Code Samples" and two sections: "PHP" and "Ruby". Each section contains a code snippet in a yellow box. The "PHP" section has the following code:

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

The "Ruby" section has the following code:

```
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
```

On the right side of the main content area, there's a "Page Contents" sidebar with a tree view of the code samples: Code Samples (PHP, Ruby, Python, ColdFusion, JavaScript with jQuery).

Although the Mashape Weather API doesn't provide a code sample in the Weather API page, Mashape as a platform provides a general code sample on their [Consume an API in JS](#) page. The writers explain that you can consume the API with code on an HTML web page like this:

The screenshot shows a section titled "Using JavaScript to consume APIs". It contains a code snippet for an AJAX call:

```

1 $.ajax({
2   url: 'https://SOMEAPI.p.mashape.com/', // The URL to the API. You can get this in the API page of the API you want to consume
3   type: 'GET', // The HTTP Method, can be GET POST PUT DELETE etc
4   data: {}, // Additional parameters here
5   dataType: 'json',
6   success: function(data) { console.dir((data.source)); },
7   error: function(err) { alert(err); },
8   beforeSend: function(xhr) {
9     xhr.setRequestHeader("X-Mashape-Authorization", "YOUR-MASHAPE-KEY"); // Enter here your Mashape key
10   }
11 });

```

The code is hosted on GitHub with a link: [gistfile1.js](#). There is also a "view raw" link.

Below the code, there is a note: "Within an HTML page you can use it this way:" followed by a snippet of HTML code:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <script src="//code.jquery.com/jquery-2.1.1.min.js"></script>
5 <meta charset="utf-8">
6 <title>Mashape Query</title>
7 <script>

```

You already worked with this code earlier, so it shouldn't be new. It's mostly same code as the JavaScript snippet we just used, but here there's an error function defined, and the header is set a bit differently.

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:

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,
  "dataType": "json",
  "url": "https://simple-weather.p.mashape.com/surfreport/25&days=1",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "APIKEY"
  }
}

$.ajax(settings).done(function (response) {
  console.log(response);
  $("#surfheight").append(response.query.results.channel.surfheight);
});

</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 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 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.

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.

Putting it all together

ACTIVITY

Pull together the various parts you've worked on and bring them together to showcase the full example.

I chose to format mine in Markdown syntax in a text editor. Here's my example.

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

GET

Parameters

Parameter	Description	Data Type
days	<i>Optional.</i> The number of days to include in the response. Default is 3.	integer
units	<i>Optional.</i> 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	<i>Optional.</i> 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 --get --include 'https://simple-weather.p.mashape.com/surfreport/123?units=imperial&days=1&time=1433772000' -H 'X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p' -H 'Accept: application/json'
```

Sample response

```
{
  "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.

Response item	Description
{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 feet 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
605	Invalid time parameters. All time parameters must be in Java epoch format.
4112	The beach ID was not found in the lookup.

Code example

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,
  "dataType": "json",
  "url": "https://simple-weather.p.mashape.com/surfreport/25?days=1&units=metric",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "APIKEY"
  }
}

$.ajax(settings).done(function (response) {
  console.log(response);
  $("#surfheight").append(response.query.results.channel.surf.height);
});

</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 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 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.

Structure and templates

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.

Testing your API documentation

Set up a test environment	142
Test all instructions yourself	146
Test your assumptions	151

Set up a test environment

Walking through all the steps in documentation yourself is critical to producing good documentation. 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.

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.

Local builds

Many times, developers work on local instances of the system, meaning they build the system on their local machines and run through test code there. In order to build an application locally, you may need to install some special utilities or frameworks, become familiar with various command line operations to build the code, and more. But if you can get the local builds running on your own machine, do it. It will serve you in the long run and empower you to document content ahead of time, long before the release.

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 them for it. (Also, check the company wiki for instructions on how to set up local builds — engineers do often write internal documentation for other engineers.)

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?

Most of the time, developers respect technical writers tenfold when the technical writers write doc from test systems, trying things out for themselves. Engineers also appreciate any feedback you may have as you run through the system. Technical writers are usually the first users.



Photo from Flickr (<https://flic.kr/p/6Grete>). City water testing laboratory, 1948

If it's really 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.

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. Some developers also like to show off their technical acumen. (One programmer seemed to try to type commands as quickly and loudly as possible on the command line, as if he were on speed, though it could have just been my impression.)

At one company, to gain access to the test system, we had to jump over a series of security hurdles. For example, connections to Amazon Web Services from internal systems required developers to go through an intermediary server. So to connect to the AWS test instance, you had to SSL to the intermediary server, and then connect from the intermediary to AWS. (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. 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. In this case, there are other methods that may be more practical, such as accessing the system from a test server.

Test servers

Instead of working with local builds, you can request that developers or QA deploy the code on a test server that you can access. Interacting with test systems is probably easier than building an application locally because the server will likely have the code and frameworks you need already installed. Depending on the product, you might be able to run all the code from the cloud and execute calls there.

For example, developers often push a test build to a server that QA runs tests against. If this is the case, it's often preferable to test on these alpha or beta web server environments because the code tends to be more stable than local builds.

Further, 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.

Ask your QA team where they keep their test cases — they're not always readily visible. [Testrail](#) is a common tool used to manage test cases — if your QA team uses it, become familiar with it. (The system isn't all that intuitive.)

With the test system, you'll need to get the appropriate URLs, login IDs, roles, etc., from your dev or QA team. Ask them if there's anything you shouldn't alter or delete because sometimes the same testing environment is shared among groups.

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.

Can you see how just getting the test system set up and ready can be challenging? Still, if you want to write good documentation, this 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.

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. Or you may need to download Android Studio and connect it to an actual device. If the app is in PHP, you may need to install PHP.

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 developer 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.

Hardware products

If you're documenting a hardware product, you'll want to secure a device that has the right build (e.g., a development build) installed on it. Big companies often have prototype devices available. At Amazon, there are kiosks where you can "flash" (quickly install) a specific build number on the device.

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 ever got 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 thing, much less swap out a storage disk, run commands in the terminal, replace a RAID, or do some other task (for which I was supposedly writing instructions).

(I learned early on to steer my career towards jobs where I could actually get my hands on and play around with.) 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 or otherwise interface with it.

Next steps

It might take one or more days to get your test environment set up. Be persistent. After you get the test environment set up, it's time to [test your instructions \(page 146\)](#).

Test all instructions yourself

After setting up your [test environment \(page 142\)](#), 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 the system.

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 \(page 46\)](#), [Postman \(page 38\)](#), 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 Amazon, it wasn't until I built my own app and submitted it to the Appstore that I discovered some bugs. At the time, 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.

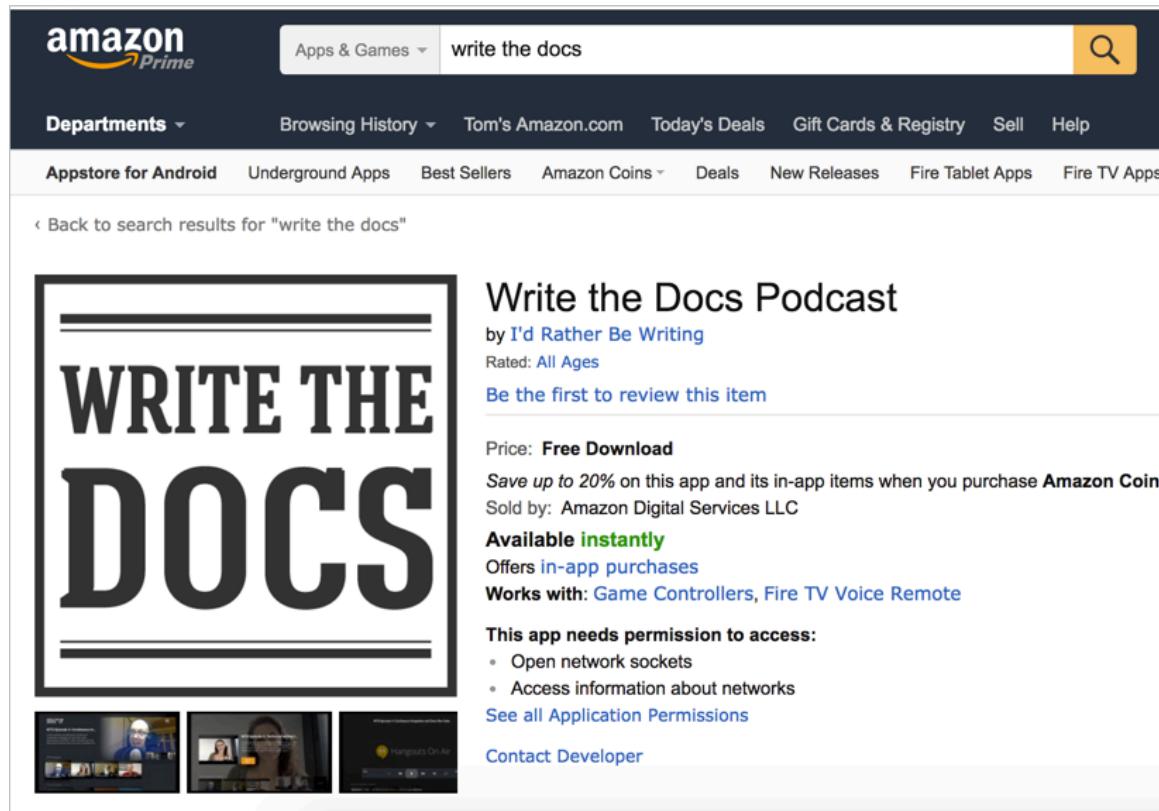
Then I needed to construct the media feed that I would use to integrate with the app template. The app template could read all the media from a feed by targeting it with Jayway Jsonpath or XPath expression syntax.

I used Jekyll to build my feed. You can view my JSON-based feed here: podcast.writethedocs.org/fab.json. I auto-generated the feed through Liquid `for` loops in Jekyll.

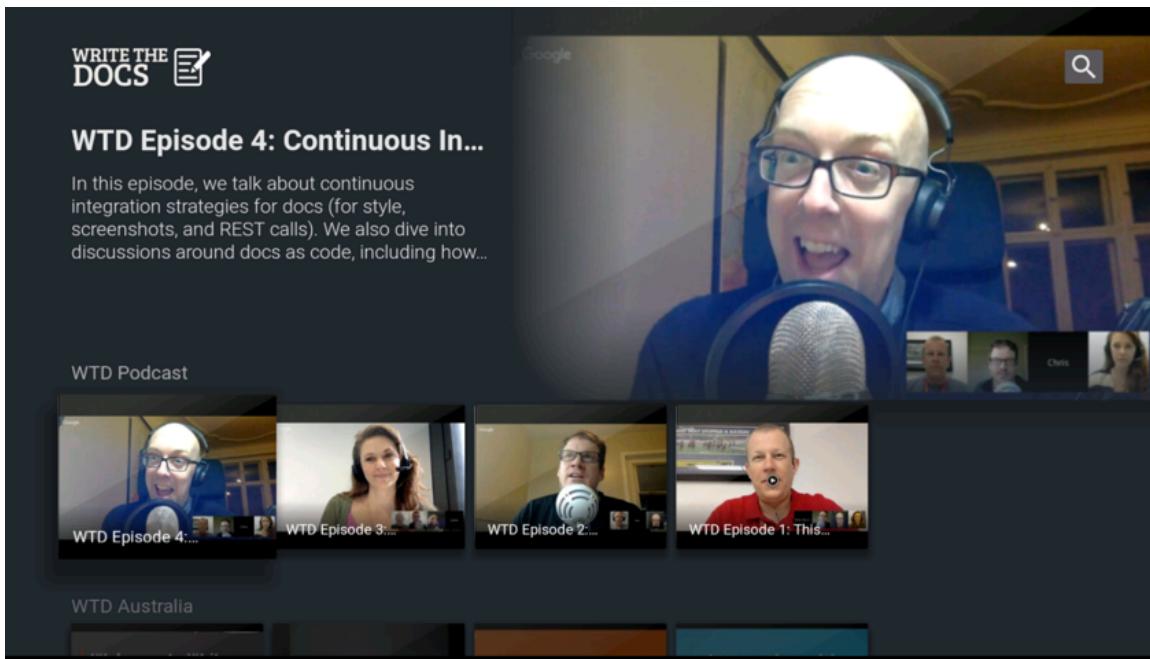
The most difficult part in setting up this feed was configuring the `recommendations` object. Each video has some `tags`. The `recommendations` object needs to show other videos that have the same tag. Getting the JSON valid there was challenging and I relied on some support from the Jekyll forum.

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.

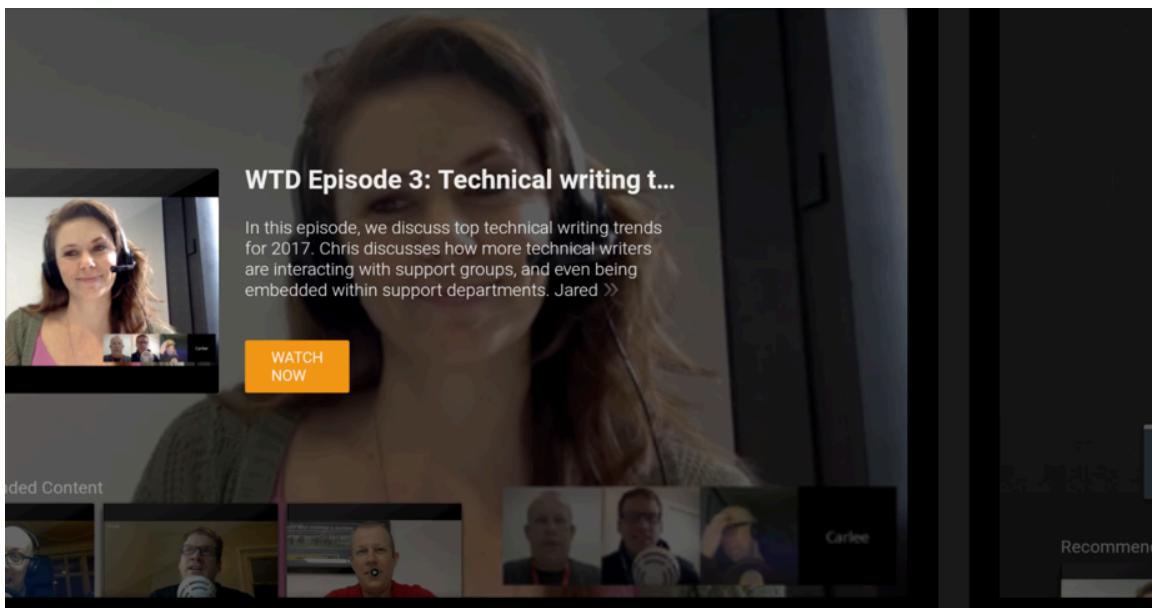
Submitting the app into the Appstore was fun and illuminated parts of the developer's workflow that I hadn't previously understood. You can view the Write the Docs podcast app in the Amazon Appstore website [here](#).



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 automatically 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?

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.

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, using 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.

Enjoyment benefits from 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 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.

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 can capture feedback as your docs get used (feedback from forums, contact email, logs, and other means), you can improve it and see gaps that you likely missed.

Test your assumptions

Almost all documentation builds on assumptions that may or may not be shared with your audience. While [testing your documentation \(page 146\)](#), recognize that what may seem clear to you may be confusing to your users. Learn to identify these assumptions that can interfere with your user's ability to follow the instructions in your docs.

Assumptions about terminology

You may assume that users 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.

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 far 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 whenever this happens I usually get strange feedback, as if a robot were testing my instructions.

QA people test to see whether the instructions are accurate. They don't test whether a user would understand the instructions or whether concepts are clear. And 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, 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.

Documenting non-reference sections

User guide tasks	156
Documenting the API overview	161
Documenting the getting started section	162
Documenting authentication and authorization requirements.....	166
Documenting code samples and tutorials.....	173
How to create the quick reference guide	176
Next phase of course	178

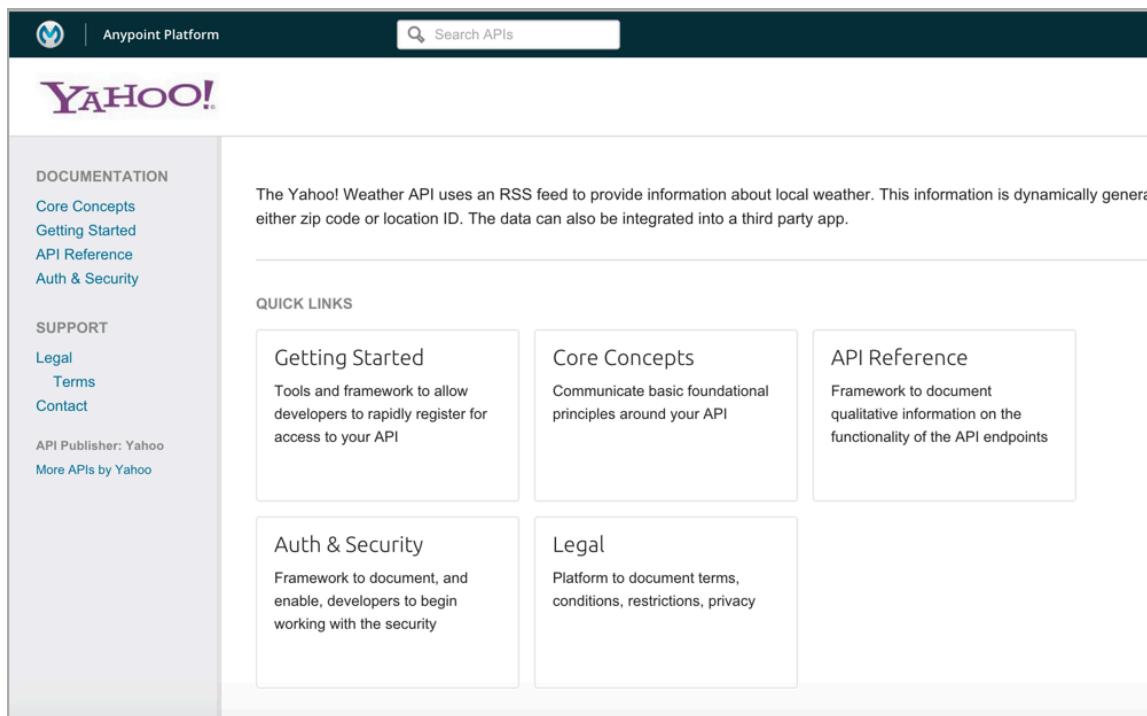
User guide tasks

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. (These other sections are absent from the Mashape Weather API because it's such a simple API.)

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 Anypoint Platform logo and a search bar. The main content area features the Yahoo! logo at the top. On the left, there's a sidebar with sections for DOCUMENTATION (Core Concepts, Getting Started, API Reference, Auth & Security), SUPPORT (Legal, Terms, Contact), and links to API Publisher: Yahoo and More APIs by Yahoo. The main content area contains a large text block about the RSS feed, followed by 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 Yahoo! Weather API documentation page. At the top, there's a navigation bar with the Anypoint Platform logo and a search bar labeled "Search APIs". Below the header, the Yahoo! logo 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, Contact, API Publisher: Yahoo, and More APIs by Yahoo. The main content area contains a brief description of the 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, there's 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).

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 contains a navigation menu with sections like Overview, Authentication, Secure API Requests, Real-time, Mobile Sharing, API Console, and Endpoints. Under Endpoints, there is a subsection for Relationships. 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 and examples:

- 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 examples are three specific requests:

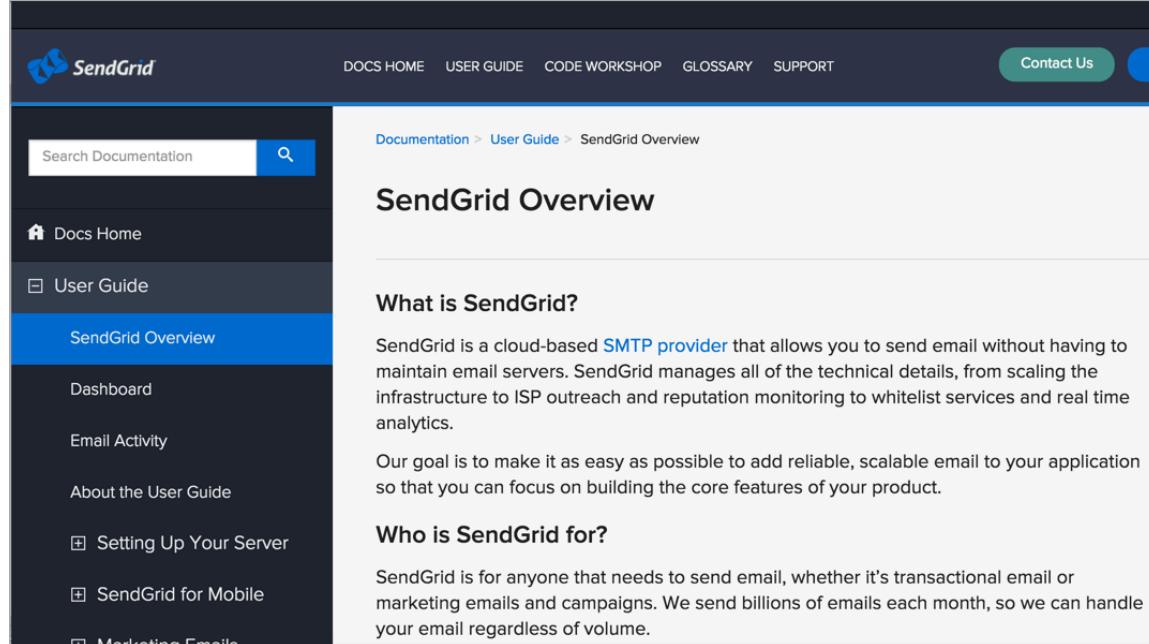
- GET /users/ user-id /follows**:
URL: https://api.instagram.com/v1/users/{user-id}/follows?access_token=ACCESS-TOKEN [RESPONSE]
- GET /users/ user-id /followed-by**:
URL: https://api.instagram.com/v1/users/{user-id}/followed-by?access_token=ACCESS-TOKEN [RESPONSE]
- GET /users/self/requested-by**:
URL: [https://api.instagram.com/v1/users/self/requested-by?access_token=ACCESS-TOKEN](#) [RESPONSE]

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:

A screenshot of the SendGrid User Guide's "SendGrid Overview" page. The page has a dark header with the SendGrid logo, navigation links for Docs Home, User Guide, Code Workshop, Glossary, and Support, and Contact Us and Feedback buttons. A search bar is at the top left. The main content area shows the breadcrumb path: Documentation > User Guide > SendGrid Overview. The title "SendGrid Overview" is centered above two sections: "What is SendGrid?" and "Who is SendGrid for?".

The sidebar on the left contains a navigation menu with the following items:

- Docs Home
- User Guide
 - SendGrid Overview (selected)
 - Dashboard
 - Email Activity
 - About the User Guide
 - Setting Up Your Server
 - SendGrid for Mobile
 - Marketing Emails

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 HTTP URL that users will need to make the request. Listing out the full HTTP URL with each endpoint would be tedious and take up a lot of space.

You generally list the full HTTP 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 (or resource root), 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 left sidebar has sections for Overview, Getting started, Authentication, Webhooks, Title expansion, Rate limiting, and Response codes. Below these are sections for Integrations (Overview, Quick start, In-app dialogs, Third-party libraries), Capabilities API, and Emoticons API. The main content area features a "Getting started" section with a welcome message about building integrations and supported platforms (HipChat.com and HipChat Server). To the right is a "API changelog" box with a link to subscribe, followed by sections for "Build your own integrations" and "Develop an independent add-on".

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

The screenshot shows the Aeris Weather API documentation homepage. At the top, there's a navigation bar with the Aeris logo, followed by links for CONSUME, DEVELOP, VISUALIZE, and MANAGE. Below the navigation, a breadcrumb trail reads HELP CENTER / DOCS / AERIS WEATHER API / GETTING STARTED. On the left, a sidebar for the Aeris Weather API contains links for Getting Started, Reference, and Downloads. The main content area is titled "Getting Started". It includes a brief introduction about needing an active Aeris API subscription and application registration, followed by four steps: 1. Sign up for an Aeris API subscription service. 2. Log in to your account to register your application for an API access key. 3. Find the endpoints and actions that provide you with the data you need. 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 homepage. At the top, there's a logo for SmugMug and a link to PHOTO SHARING. The main navigation menu on the left includes links for SmugMug API (selected), BETA, Guidelines for Beta Users, Release Log, Known Issues, Contact Us, Legal, and Legal (Beta). Under the "Tutorial" section, there are links for Getting an API Key, Your First API Request, Making changes, Getting results in pages, Authorization with OAuth 1.0a, Examples (web app), and Examples (non-web app). Under "API Concepts", there are links for HTTP Methods, Object Identifiers, Creating Objects, and Status Codes. The main content area is titled "Welcome to the SmugMug API!". It features a beta announcement, a list of API key requests, and sections for "What is the SmugMug API?", "Learn the SmugMug API", and a prominent green 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).

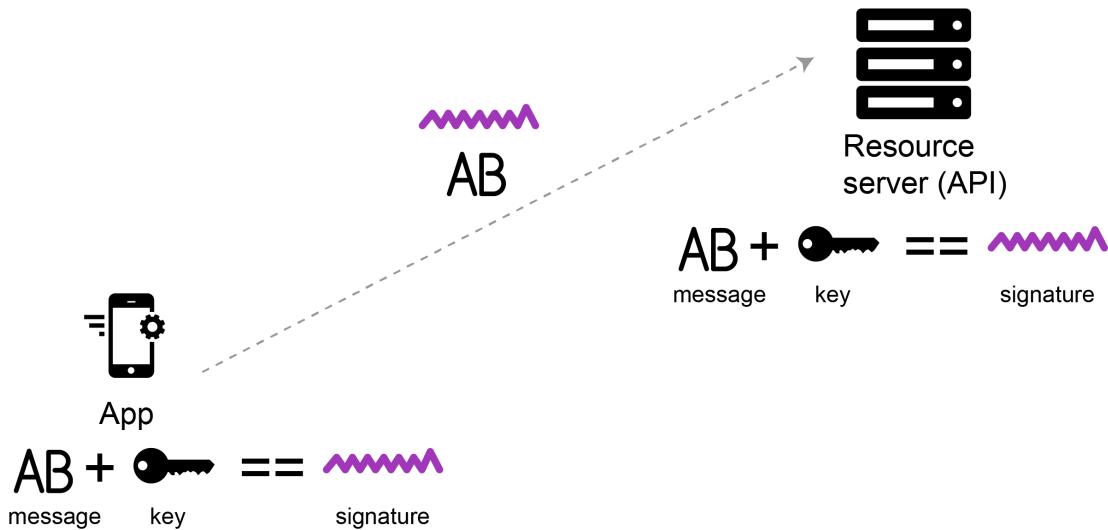
The message is then encoded by the secret key and passed through a secure hashing algorithm (SHA). (A hash is a scramble of a string based on an algorithm.) The resulting value, referred to as a signature, is placed in the request header.

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

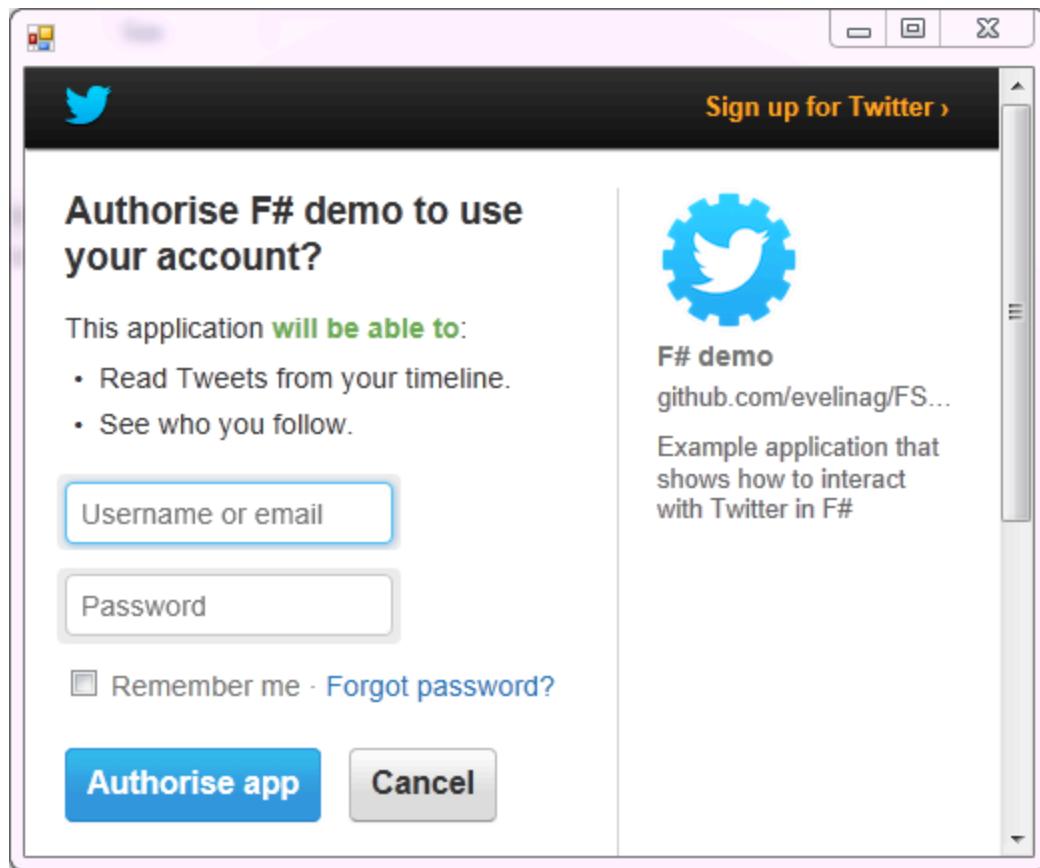


The important point is that the secret key (critical to reconstructing the hash) is known only to the sender and receiver. The secret key is not included in the request.

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.



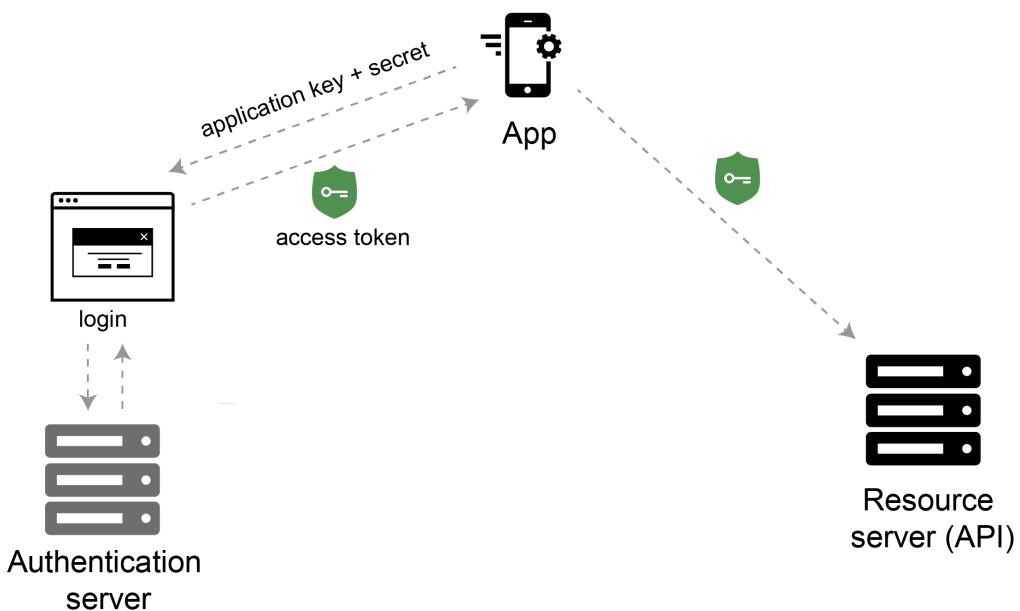
There are a few varieties of OAuth — namely, “one-legged OAuth” and “three-legged OAuth.” One-legged OAuth is used when you don’t have sensitive data to secure. This might be the case if you’re just retrieving general, read-only information (such as news articles).

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 located at the top left. On the left, there is a sidebar menu under the 'User Guide' heading, which includes 'SendGrid Overview', 'Dashboard', 'Email Activity', 'About the User Guide', 'Setting Up Your Server', 'SendGrid for Mobile', and 'Marketing Emails'. The main content area shows the 'API Keys' page. The breadcrumb navigation indicates the path: Documentation > User Guide > Settings > API Keys. The page title is 'API Keys'. Below the title, a paragraph explains that API Keys are used for authentication and are ideal over username and password because they can be revoked without changing credentials. It suggests using API keys for connecting to all services. Definitions for 'Name', 'API Key ID', and 'Action' are provided. A section titled 'Create an API Key' describes how clicking the button opens a window for naming the key. The URL for the page is <https://docs.sendgrid.com/api-reference/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 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.

Here's a sample code sample page from Mashape:

The screenshot shows the Mashape website interface. On the left, there's a sidebar with navigation links: Firewall, IPs and Security; Access Control; Embed Mashape; API Pricing; Private plans; Unirest; Frequently Asked Questions; On-Premises; ADDING AN API (with sub-links: Add your API to Mashape, Define Basic API settings, Documenting an API); and CONSUMING AN API (with sub-link: Consume APIs in JS). The main content area has a title "Using JavaScript to consume APIs". Below the title, a text block explains: "When consuming an API through Mashape, you can run directly in your website or browser console by using this sample code snippet below which is CORS-enabled and uses jQuery: This way you don't even have to worry about cross-domain requests." A code block follows, showing a jQuery AJAX call. At the bottom of the code block, it says "gistfile1.js hosted with ❤ by GitHub" and "view raw". Below the code block, another text block says "Within an HTML page you can use it this way:" followed by a partial HTML code block.

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.

You are not the audience

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 are maintenance heavy 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.

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. It includes a header with the title "Shopify Cheat Sheet" and a "follow on twitter" button. The main content is organized into three columns:

- Liquid** (Left Column):
 - Logic**: Includes tags like {%- comment %}, {%- raw %}, {%- if %}, {%- unless %}, {%- case %}, {%- cycle %}, {%- for %}, {%- tablerow %}, {%- assign %}, {%- capture %}, and {%- include %}.
 - Operators**: Shows operators like ==, !=, <, >, >=, <=, or, and, contains.
 - Images**: Shows the sizes filter.
- 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
- Endpoint definition and method
- 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

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 \(page 382\)](#) section:

- [Overview for exploring other REST APIs \(page 389\)](#)
- [EventBrite example: Get event information \(page 390\)](#)
- [Flickr example: Retrieve a Flickr gallery \(page 397\)](#)
- [Klout example: Retrieve Klout influencers \(page 406\)](#)
- [Aeris Weather Example: Get wind speed \(page 416\)](#)

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. This is the focus of the next part of the course.

Publishing your API documentation

Overview for publishing API docs.....	180
List of 100 API doc sites.....	183
Design patterns with API doc sites.....	187
Tool decisions: Who will write?	195
Docs-as-code tools	197
Github wikis	200
More about Markdown	205
Version control systems	213
Pull request workflows through GitHub.....	221
Jekyll — my favorite static site generator.....	227
Other tool options — a miscellaneous compilation.....	238
What about traditional help authoring tools (HATs)?.....	245
Tools versus Content.....	248
Case study: Switching tools to docs-as-code	249

Overview for publishing API docs

In earlier parts of this course, I used a simple [Weather API from Mashape](#) to demonstrate how to use a REST API. Now I'll explore various tools to publish documentation about this same Mashape Weather API.

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.

My story: Turning from DITA to Jekyll

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, has 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 \(page 197\)](#), committing it to [version control \(page 213\)](#), 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 \(page 0\)](#) and [Doxygen \(page 0\)](#) 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

The reference documentation for APIs is pushed into well-defined templates, which list sections such as endpoint parameters, sample requests, sample responses, and so forth. (I discuss these reference sections in [Documenting endpoints \(page 81\)](#).)

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 \(page 259\)](#) 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 \(page 193\)](#) 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.) The response you see from this explorers is from your own data in the API.

5. With APIs, the doc *is* the 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 [Tripane help and PDF files: past their prime?](#) from Robert Desprez.)

With API documentation, often times the documentation *is* the product — there isn't a separate GUI that clients interact with. That GUI is the documentation, so it has to be sexy and awesome.

Most tripane help doesn't make that cut. If the help looks old and frame-based, it doesn't instill much confidence toward the developers using it.

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 \(page 227\)](#). I've come to really love using Jekyll, working primarily in Markdown, leveraging Liquid for conditional logic, and committing updates to a repository.

I realize that not everyone has the luxury of switching authoring tools, but when I initially made the switch, the company I was working for was a former 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 it's [code-based approach \(page 197\)](#), but there are [many different options and routes \(page 183\)](#) 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 and publishing strategies and into more developer-centric tools.

List of 100 API doc sites

The following are about 100 openly accessible REST APIs that you can browse as a way to look at patterns and examples. Most of these REST API links are available from [programmableweb.com](#). I highly recommend that you [browse APIs by category docs](#) from their site. I initially started gathering a list of the APIs in Programmableweb's [most popular APIs docs](#), but then I just started adding links as I ran across interesting APIs.

Here are 100 API doc sites. Peruse them to get a sense of the variety, but also try to identify common patterns.

1. [Google Places API docs](#)
2. [Twitter API docs](#)
3. [Flickr API docs](#)
4. [Facebook's Graph API docs](#)
5. [Youtube API docs](#)
6. [eBay API docs](#)
7. [Amazon EC2 API docs](#)
8. [Twilio API docs](#)
9. [Last.fm API docs](#)
10. [Bing Maps docs](#)
11. [gpodder.net Web Service docs](#)
12. [Google Cloud API docs](#)
13. [Foursquare Places API docs](#)
14. [Walmart API docs](#)
15. [Dropbox API docs](#)
16. [Splunk API docs](#)
17. [Revit API docs](#)
18. [DocuSign API docs](#)
19. [Geonames docs](#)
20. [Adsense API docs](#)
21. [Box API docs](#)
22. [Amazon API docs](#)
23. [LinkedIn REST API docs](#)
24. [Instagram API docs](#)
25. [Zomato API docs](#)
26. [Yahoo Social API docs](#)
27. [Google Analytics Management API docs](#)
28. [Yelp API docs](#)
29. [Panoramio API docs](#)
30. [Facebook API docs](#)
31. [Eventful API docs](#)
32. [Concur API docs](#)
33. [Paypal API docs](#)
34. [Bitly API docs](#)
35. [Callfire API docs](#)
36. [Reddit API docs](#)
37. [Netvibes API docs](#)
38. [Rhapsody API docs](#)
39. [Donors Choose docs](#)
40. [Sendgrid API docs](#)

41. Photobucket API docs
42. Mailchimp docs
43. Basecamp API docs
44. Smugmug API docs
45. NYTimes API docs
46. USPS API docs
47. NWS API docs
48. Evernote API docs
49. Stripe API docs
50. Parse API docs
51. Opensecrets API docs
52. CNN API docs
53. CTA Train Tracker API
54. Amazon API docs
55. Revit API docs
56. Citygrid API docs
57. Mapbox API docs
58. Groupon API docs
59. AddThis Data API docs
60. Yahoo Weather API docs
61. Glassdoor Jobs API docs
62. Crunchbase API docs
63. Zendesk API docs
64. Validic API docs
65. Ninja Blocks API docs
66. Pushover API docs
67. Pusher Client API docs
68. Pingdom API docs
69. Daily Mile API docs
70. Jive docs
71. IBM Watson docs
72. HipChat API docs
73. Stores API docs
74. Alchemy API docs
75. Indivo API 2.0 docs
76. Socrata API docs
77. Github API docs
78. Mailgun API docs
79. RiotGames API docs
80. Basecamp API docs
81. ESPN API docs
82. Snap API docs
83. SwiftType API docs
84. Snipcart API docs
85. VHX API docs
86. Polldaddy API docs
87. Gumroad API docs
88. Formstack API docs
89. Livefyre API docs
90. Salesforce Chatter REST API docs
91. The Movie Database API docs
92. Free Music Archive API docs

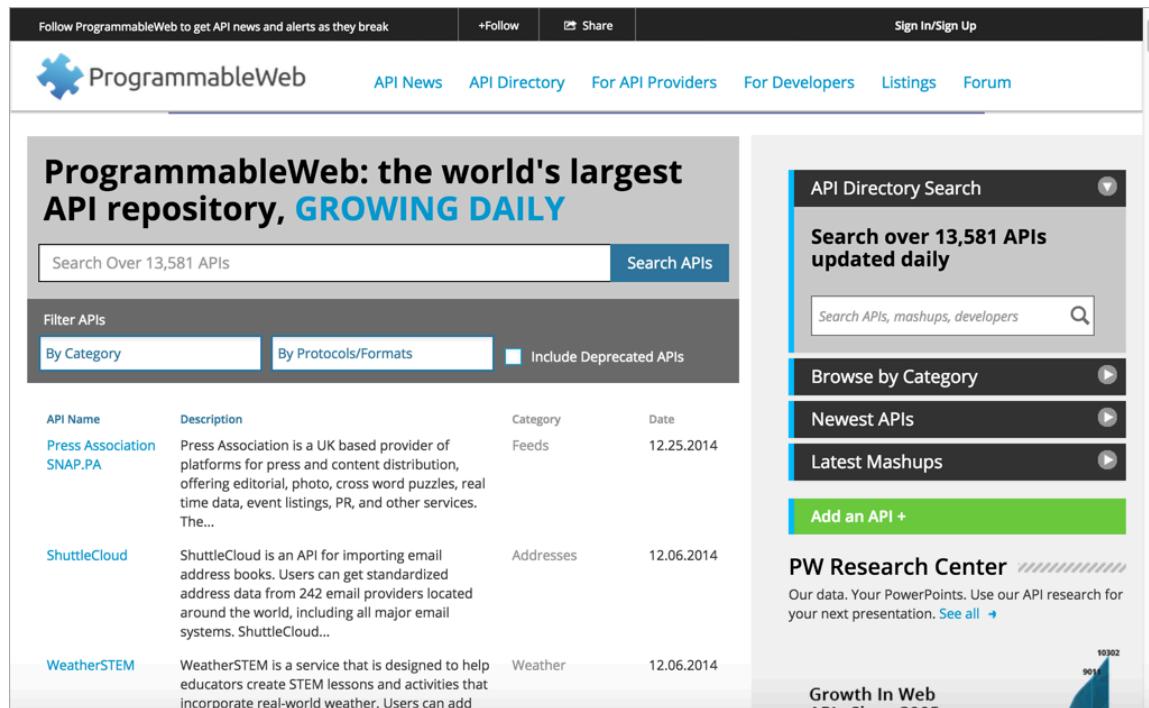
93. [Context.io docs](#)
94. [CouchDB docs](#)
95. [Smart Home API \(Amazon Alexa\) docs](#)
96. [Coinbase docs](#)
97. [Shopify API docs](#)
98. [Authorize.net docs](#)
99. [Trip Advisor docs](#)
100. [Pinterest docs](#)
101. [Uber docs](#)
102. [Spotify API](#)

Tip: I last checked these links in October 2017. 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`.

I include not only impressively designed docs in this list 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 them their own way, with their own site, branding, organization, and typography.

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 more web APIs.



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

The main header reads "ProgrammableWeb: the world's largest API repository, GROWING DAILY". Below it is a search bar with the placeholder "Search Over 13,581 APIs" and a "Search APIs" button. There are also "Filter APIs" buttons for "By Category", "By Protocols/Formats", and "Include Deprecated APIs".

A table lists several APIs with columns for "API Name", "Description", "Category", and "Date". The listed APIs are:

- Press Association (SNAP.PA): 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. Category: Feeds, Date: 12.25.2014
- 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.. Category: Addresses, Date: 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. Category: Weather, Date: 12.06.2014

To the right, there is a sidebar titled "API Directory Search" which says "Search over 13,581 APIs updated daily". It includes a search input, "Browse by Category", "Newest APIs", "Latest Mashups", and a "Add an API +" button. Below that is the "PW Research Center" with a message about data and PowerPoints, and a "Growth In Web APIs Since 2005" chart showing a significant increase from 901 to 10302.

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 \(page 187\)](#).

Note: By analyzing API doc sites first before presenting conclusions, I hope to implement more of an inductive, unbiased approach toward the conclusions I draw about API docs in this course.

Design patterns with API doc sites

In the previous topic, we browsed up to [100 API doc sites \(page 183\)](#) and looked for similar patterns in their design. “Design patterns” are common approaches or techniques in the way something is designed. In looking over the many API doc sites, I tried to identify common approaches in the way the content was published.

The following design patterns are common with API doc sites: structure and templates, website platforms, abundant code examples, long-ish pages, interactive API explorers, and GitHub as a storage platform. 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 \(page 81\)](#). The non-reference topics also share similar topics, which I [touched upon \(page 155\)](#) 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 style, and allow you to more easily change the design without manually reformatting each section. You could just remember to add the exact same sections on each page, but this requires more manual consistency.

With a template, you can insert various values (descriptions, methods, parameters, etc.) into a highly stylized output. Rather than work with all the style tags in your page directly, you can create values that exist as an object on a page. A custom script can loop through the objects and insert the values into your template.

Different authoring tools have different ways of processing templates. In Jekyll, a static site generator, 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>
```

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.

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.

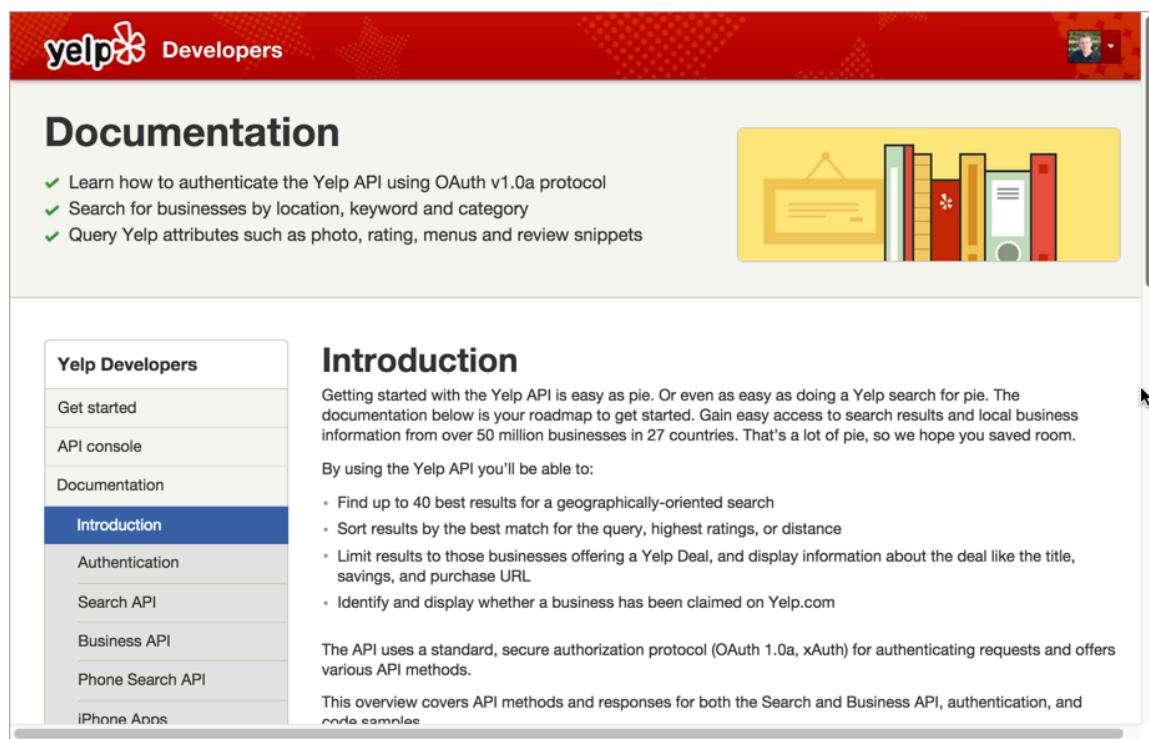
When I worked at Badgeville, a gamification startup, we published using Drupal. We had a design agency construct a highly designed template in Drupal. To publish the API reference documentation, engineers wrote a custom script that generated the content from a database into a JSON script that we imported 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 raw content but also insert it dynamically into the stylized template.

As you look for documentation tools, keep in mind the need to template your API reference documentation.

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 website. The top navigation bar features the Yelp logo and the word "Developers". On the right side of the header is a user profile icon. Below the header, the main content area has a red background with a yellow sidebar on the left. The sidebar contains a list of links: "Documentation", "Get started", "API console", "Introduction" (which is highlighted in blue), "Authentication", "Search API", "Business API", "Phone Search API", and "iPhone Apps". The main content area has a white background. It features a large title "Documentation" and a list of three items under the heading "Learn how to authenticate the Yelp API using OAuth v1.0a protocol": "Search for businesses by location, keyword and category", "Query Yelp attributes such as photo, rating, menus and review snippets". To the right of this list is a decorative graphic of several books of different colors (yellow, green, red, blue) arranged vertically. Below the sidebar, there is a section titled "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." It also lists what can be done with the API and describes the authentication protocol used.

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 product 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 \(page 259\)](#) 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 tri-pane help output), you can leverage all the scripts, 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 \(page 245\)](#); 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 \(page 126\)](#). 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. The top navigation bar includes links for EVERNOTE Developers, Search, Docs, Resources, and a button to 'GET AN API KEY'. On the right side, there are links to API Reference, iOS SDK Reference, and Android SDK Reference. Below these are 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, and Notes (Creating Notes, Sharing Notes, Reminders, Read-only Notes). The main content area features a section titled 'Sharing (and Un-Sharing) Notes' with a sub-section 'Single Note Sharing'. It explains that single notes can be shared via a public note URL or email. It then provides a code snippet for Python:

```

1 def getUserShardId(authToken, userStore):
2     """
3         Get the User from userStore and return the user's shard ID
4     """
5     try:
6         user = userStore.getUser(authToken)
7     except (Errors.EDAMUserException, Errors.EDAMSystemException), e:
8         print "Exception while getting user's shardID:"
9         print type(e), e
10    return None
11

```

The writers at Parse emphasize the importance of code samples in docs:

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 will incorporate one of these highlighting tools (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 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 \(page 126\)](#) in another topic.

Pattern 4: Longish pages

One of the most stark differences between regular end-user documentation and developer documentation is that developer doc pages tend to have longer pages. In a [post on designing great API docs](#), the writers at Parse explain that short pages frustrate developers:

Minimize Clicking

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 1.0.0 documentation page. On the left, there's a sidebar with navigation links: [Backbone.js \(1.0.0\)](#), [» GitHub Repository](#), and [» Annotated Source](#). Below these are sections for [Introduction](#), [Upgrading](#), [Events](#) (with a list of methods: - on, - off, - trigger, - once, - listenTo, - stopListening, - listenToOnce, - Catalog of Built-in Events), [Model](#) (with a list of methods: - extend, - constructor / initialize, - get, - set, - escape, - has, - unset, - clear, - id, - idAttribute, - cid, - attributes), and a [Catalog of Built-in Events](#).

The main content area features the Backbone.js logo (a blue stylized 'B') and the title "BACKBONE.JS". Below the title, there's a paragraph about Backbone.js providing structure to web applications through models, collections, and views, and connecting them via a RESTful JSON interface. It also mentions the project's GitHub repository, annotated source code, test suite, example application, tutorials, and real-world projects.

Further down, there's information on reporting bugs and discussing features on GitHub issues, Freenode IRC, Google Group, and the DocumentCloud wiki. A note states that Backbone is an open-source component of DocumentCloud.

At the bottom, there's a section titled "Downloads & Dependencies" with a note: "(Right-click, and use "Save As")".

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

This is automatically-generated documentation for the reddit API.

The reddit API and code are [open source](#). Found a mistake or interested in helping us improve? Have a gander at [api.py](#) and send us a pull request.

Please take care to respect our [API access rules](#).

overview

listings

Many endpoints on reddit use the same protocol for controlling pagination and filtering. These endpoints are called Listings and share five common parameters: `after` / `before`, `limit`, `count`, and `show`.

Listings do not use page numbers because their content changes so frequently. Instead, they allow you to view slices of the underlying data. Listing JSON responses contain `after` and `before` fields which are equivalent to the "next" and "prev" buttons on the site and in combination with `count` can be used to page through the listing.

The common parameters are as follows:

- `after` / `before` - only one should be specified. these indicate the `fullname` of an item in the listing to use as the anchor point of the slice.
- `limit` - the maximum number of items to return in this slice of the listing.
- `count` - the number of items already seen in this listing. on the html site, the builder uses this to determine when to give values for `before` and `after` in

Why do API doc sites tend to have long-ish 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 only need 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, or rather "everything at your fingertips."
- **Today's navigation controls are sophisticated:** Today there are better navigation controls for moving around on long pages than 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 page. Long pages also present challenges with linking as well. Either way you approach page length, developers probably won't care that much. They will care much more about the content on the page.

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 wiring 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 \(page 259\)](#) to provide the interactivity.

The screenshot shows a web-based API explorer interface. At the top, it says "Response Content Type: application/json". Below that is a table titled "Parameters" with the following columns: Parameter, Value, Description, Parameter Type, and Data Type. The rows are:

Parameter	Value	Description	Parameter Type	Data Type
apikey	<input type="text"/>	Your API key	query	string
html	<input type="text"/> (required)	HTML content (must be URL encoded)	query	string
outputMode	<input type="button" value="json"/>	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 it.

Are API explorers novel, or extremely instructive? If you're going to be making a lot of calls, there's no reason why you couldn't just use [cURL \(page 46\)](#) or [Postman \(page 38\)](#) (particularly the [Postman Run Button \(page 242\)](#)) 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 the 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 \(page 238\)](#).

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 pull this off.

Pattern 6: GitHub as a Storage Platform

Another common pattern with API doc sites is that, given that developers are often heavily involved in the writing, managing, and publishing of the information, the documentation source is often stored in [GitHub](#), which is an online platform for managing Git projects.

Git frequently acts as a storage source that other sites can pull from. For example, with many online platforms, such as [CloudCannon](#) and [Forestry.io](#), you can set your content source as GitHub. These platforms will then pull in your content from GitHub, treating it as the source. This way you can do content management using GitHub but configure the front-end doc experience using one of these platforms.

Many doc sites just use [GitHub Pages](#) directly as their doc site. If your site is a Jekyll site, GitHub Pages will build it automatically when you commit into your repo. Building from the server provides enormous benefits to publishing in a [docs-as-code model \(page 197\)](#), and it's a topic I touch on in a [case study here \(page 249\)](#). GitHub's preference for building with Jekyll is one reason why I focus an [entire topic on Jekyll \(page 227\)](#) later in the course.

Many developer doc sites even promote their online GitHub source with a button that says "Edit on GitHub." See these sites for examples:

- [Smarthings developer documentation](#)
- [Apache MyNewt documentation](#)
- [Quill documentation](#)
- [Jekyll documentation](#)

I won't discuss the challenges of building community and contributions using "Edit on GitHub" links like this. For the most part, my experience with external collaborators has been about the same as with wikis. For more on inviting collaboration from users, see these posts:

- [Crowdsourcing docs with docs-as-code tools – same result as with wikis?](#)
- [My Journey To and From Wikis: Why I Adopted Wikis, Why I Veered Away, and a New Model][<http://idratherbewriting.com/2012/06/11/essay-my-journey-to-and-from-wikis-why-i-adopted-wikis-why-i-veered-away-from-them-and-a-new-model-for-collaboration/>]
- [1% rule](#)

For most part, if you have an online project where the contributors interact and publish online, these "Edit on GitHub" links can facilitate writing and editing within the group. It's not so much that these groups are hoping outsiders will jump in and make a bunch of edits (though that might be welcome). More common is that GitHub is the infrastructure for their open source project, and these links make it easier for existing contributors (already committed to the project) to edit and update pages.

Some non-patterns in API doc sites

Finally, I'd like to mention some non-patterns in API documentation. In the [list of 100 API doc sites \(page 183\)](#), rarely do you see any of the following:

- Video tutorials
- PDFs
- Page commenting features
- Translated sites

By non-patterns, it's not to say these elements aren't a good idea. But generally they aren't emphasized as primary requirements.

Tool decisions: Who will write?

One of the first considerations to make when you think about API doc tooling is who will be doing the writing. If developers will be writing and contributing to the docs, you should integrate the writing tools and process into their toolchain and workflow.

On the other hand, if technical writers will create all the documentation, using the same tools and processes as developers use for code may not be as important.

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 surveys at Google about the workplace, 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 information, 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 tools onto engineers, they fit the documentation into developer tools.

Developers now write documentation in Markdown files in the same repository as their code. Some other engineers wrote a script to display these Markdown files in a browser directly from the code repository.

The method quickly gained traction, with hundreds of developer projects adopting the new method. Now instead of authoring documentation in a separate system (using writers' 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](#).

Pros and cons of having developers write

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 \(page 197\)](#) approach, whereas if only tech writers will edit, you can lean more toward traditional [help authoring tools \(page 245\)](#).

Overall, I'm much in favor of the [docs-as-code tools \(page 197\)](#), which aligns with developer environments. Many times developer documentation is so complex, only developers can really write 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).

Additionally, some developers prefer to just write the doc themselves, communicating from one developer to another. If a developer is the audience, and another developer is the writer, chances are they can cut through some of the guesswork about assumptions, prerequisite knowledge, and accuracy. It's also more efficient than trying to transmit the information to a technical writer.

On the other hand, a developer who creates the API may assume too much of the audiences' technical ability. As a result, the documentation may not be helpful. Steven Pinker explains that the [curse of knowledge](#) is one reason why writing is often poor and confusing. The more you know about a topic, the

more assumptions and background information you have automatically firing away in your brain. You become blind to all of these assumptions, terms, and other details that new learners struggle with. You're so familiar with a topic that you can't see it as a new learner would. You don't know the questions to ask, the things that don't make sense.

Collaborative effort

In most cases, API documentation is a collaborative effort between developers and technical writers. Developers tend to focus more on writing the [reference documentation \(page 81\)](#), while technical writers focus more on the [non-reference documentation \(page 155\)](#).

Developers who are heavily involved in writing documentation might want to store the documentation in the same repository as their code. In fact, if they can auto-generate the documentation from comments in the code, that's usually their preference. This approach reduces documentation drift, meaning it helps keep the code in sync with the documentation.

Either way, both technical writers and developers will often work with each other in a close way.

Docs-as-code tools

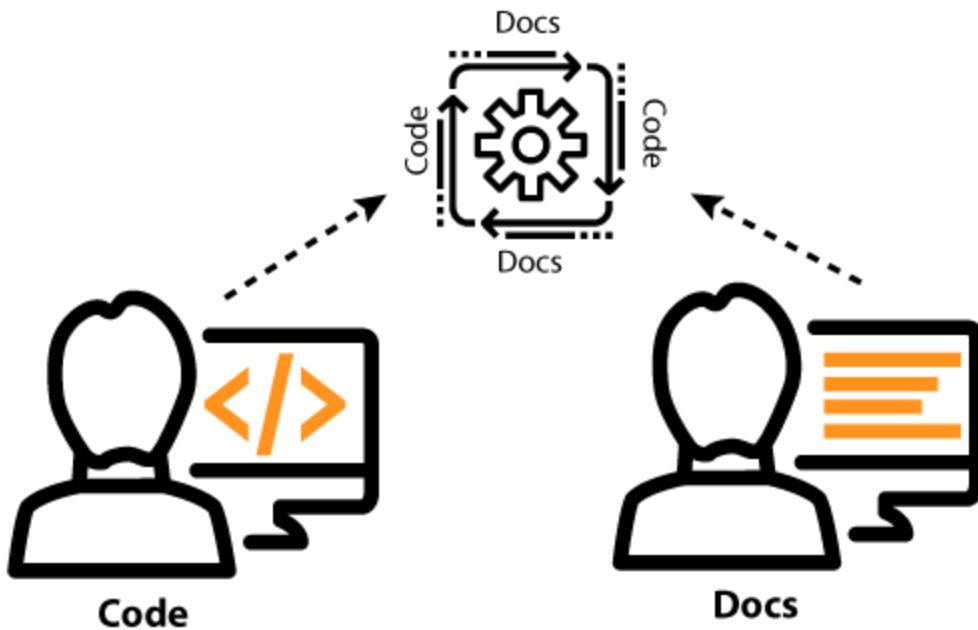
What authoring tool works best for developer documentation, especially when documenting APIs? I've used a number of authoring tools (Flare, Robohelp, Confluence, Word, Drupal, Google Docs, and others), and while different tools fit different scenarios with varying strengths, if you're writing *developer docs*, nothing works quite so well as static site generators coupled with a docs-as-code publishing workflow.

What docs-as-code tools means

Docs-as-code tools 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, Jekyll, or Hugo 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 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. By implementing a docs-as-code approach, you'll enable developers to more easily contribute and participate in the documentation.

Most developers are comfortable with Markdown, enjoy being able to work in their existing IDE to edit content, understand how 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.

Simplified publishing

When you can build from the server by simply pushing content into a Git repository, it greatly simplifies the act of publishing. You can make edits across a number of docs, commit your code into your doc repo, and 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 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 tech writers

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 team mates are working on, the changes they've made, and you can also more easily work on each other's content. By working out of the same repository, you aren't siloed in separate projects that exist in different spaces.

Flexibility

Docs-as-code tools give you incredible flexibility to adapt or 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.

The tools are as flexible and robust as your coding skills allow. At a base level, almost all use HTML, CSS, and JavaScript, so if you are a master at these, there's 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).

To read details about switching to docs as code tools, see [Case study: Switching tools to docs-as-code \(page 249\)](#).

GitHub wikis

One of the easiest toolchains to implement with developer docs is a GitHub wiki. Learning GitHub will also 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.

When you create a repository on GitHub, the repository comes with a wiki that you can add pages to. This wiki can be really convenient if your source code is stored on GitHub.

GitHub example

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 top navigation bar includes 'This repository', 'Search', 'Pull requests', 'Issues', and 'Gist'. The repository name 'basecamp / bcx-api' is displayed, along with 'Watch 50', 'Star 638', 'Fork 176'. The main content area shows 'API documentation and wrappers for the new Basecamp <http://basecamp.com>'. Below this, there are sections for '253 commits', '5 branches', '0 releases', and '23 contributors'. A pull request 'Merge pull request #177 from tjschuck/patch-1' is listed, authored by 'georgeclaghorn' on Apr 9. The 'sections' section contains 'sections' and 'Fix some invalid JSON'. The 'README.md' section contains 'Add documentation for Groups API'. To the right, a sidebar shows 'Code', 'Issues (6)', 'Pull requests (0)', 'Wiki', 'Pulse', and 'Graphs'. It also provides an 'HTTPS clone URL' (https://github.com/basecamp/bcx-api) and links for 'Clone in Desktop' and 'Download ZIP'.

Markdown syntax

You write wiki pages in Markdown syntax. There's a special flavor of Markdown syntax for GitHub wikis. The [GitHub Flavored Markdown](#) allows you to create tables, add classes to code blocks (for proper syntax highlighting), and more.

The wiki repository

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.

Treating doc as code

One of the neat things about using a GitHub repository is that you treat the doc as code, 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 it's in its own repository, technical writers can work in the documentation right alongside project code without getting merge conflicts.

Working locally allows you to leverage other tools

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. This means you can handle all the re-use, conditional filtering, and other logic outside of the GitHub wiki. You can then output your content as Markdown files and then commit them to your GitHub repository.

Limitations with GitHub wikis

There are some limitations with GitHub wikis:

- **Limited branding.** All GitHub wikis look the same.
- **Open access on the web.** If your docs need to be private, GitHub isn't the place to put them.
- **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.

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 it, set up Git on your computer.

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

It's easiest to install Git by [Installing GitHub Desktop](#). Installing GitHub Desktop will include all the Git software as well.

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. You should use that special GitHub Shell rather than the usual command line prompt.

Note that 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. Open the Terminal in the same way — go to **Applications > Utilities > Terminal**.

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

GitHub and Git are not the same thing. Git provides distributed version control. 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.

2. 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 tomjoht 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](#)

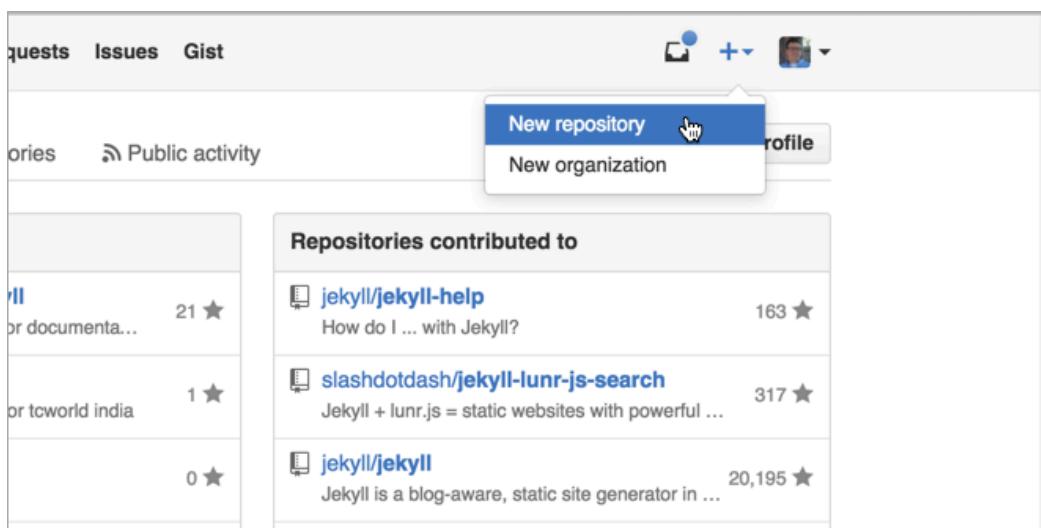
After you make these configurations, close and re-open your terminal.

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.

In this section, you’ll be using Git commands through your terminal or command prompt. In later tutorials, you’ll use the GitHub Desktop and GitHub browser tools. You can interact with GitHub in a variety of ways.

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. Insert your own sample documentation page, preferably using Markdown syntax. Or grab the sample Markdown page of a [fake endpoint called surfreport here](#) and insert it into the page.
6. Click **Save page**.

Notice how GitHub automatically converts the Markdown syntax into HTML with some decent styling.

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 reupload all your edits.

Save the GitHub repository locally

1. While viewing your the 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.)

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.

The “Clone this wiki locally” link allows you to easily insert the URL into a `git clone {url}` command in your terminal.

Note that the wiki is a separate clone URL than the project’s repository. Make sure you’re viewing your wiki and not your project.

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 \(page 201\)](#) above).
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/tomjoht/weatherapi.wiki.git
```

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

5. Navigate to the directory (either using standard ways of browsing for files on your computer or via the terminal) to see the files you downloaded. 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.
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 the file to your staging area:

```
git add --all
```

Git doesn’t 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 `--all`, you’re adding all the files in the folder to Git. You could also type a specific file name here instead of `--all`. Or you can type `git add .` to achieve the same result.

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](#).

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.

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 `vi editor` mode opens up. ("vi" stands for visual, 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 vi commands](#).

7. Push the changes to your repository:

```
git push
```

If you didn't [set up GitHub authentication \(page 201\)](#), you may be prompted for your GitHub user name and password.

8. Now verify that your changes took effect. Browse to your GitHub wiki repository and look to see the changes.

GitHub Workflows

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` to update origin. If two people edit the same content simultaneously between commits, you will likely need to [resolve merge conflicts](#).

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:

* first 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, first 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](#), 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 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 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.

You can work in text-file formats using your favorite code editor

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.
```

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. Or 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. For whatever reason, developers tend to be allergic to XML.

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.

You can treat the Markdown files with the same workflow and routing as code

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.

Markdown is easy to learn

Finally, 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. Markdown allows developers to quickly format content in HTML without investing hardly any time in learning a tool or XML schema or other formatting.

Drawbacks of Markdown

Markdown has a few drawbacks:

- **Limited to HTML tags:** You're pretty much limited to HTML tags. XML advocates like to emphasize how XML offers semantic tagging (and avoids the div soup that HTML can become). However, by and large HTML5 offers many semantic tags, 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.
- **White space sensitivity:** Markdown is white space sensitive, which can be frustrating at times. If you have spaces where there shouldn't be, it 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 has different flavors

Whatever system you adopt, if it uses Markdown, make sure you understand what type of Markdown it supports. There are two components to Markdown. First is the processor that converts the Markdown into HTML. Some processors include Red Carpet, [Kramdown](#), Pandoc, Discount, and more.

Beyond the processor, you need to know which type of Markdown the processor supports. Some examples include basic Markdown, Github-flavored Markdown, Multimarkdown, and others.

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.

Analyzing a Markdown sample

Take a look at the following 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 --get --include 'https://simple-weather.p.mashape.com/surfreport/123?units=imperial&days=1&time=1433772000'
-H 'X-Mashape-Key: W0yzMuE8c9mshcofZaBke3kw7lMtp1HjVGAjsndqIPbU9n2eET'
-H 'Accept: application/json'
```
```

```
## Sample response
```

```
```json
```

```
{
 "surfreport": [
 {
 "beach": "Santa Cruz",
 "monday": {
 "1pm": {
 "tide": 5,
 "wind": 15,
 "watertemp": 80,
 "recommendation": "Go surfing!"
 }
 }
 }
]
}
```

```
 "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.


```

| 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 fe  
et, it is not safe to surf. |  
| **{day}/{time}/recommendation** | An overall recommendation based on a com  
bination of the various factors (wind, watertemp, surfheight). Three respons  
es are possible: (1) "Go surfing!", (2) "Surfing conditions are okay, not gr  
eat", and (3) "Not a good day for surfing." Each of the three factors is sco  
red with a maximum of 33.33 points, depending on the ideal for each elemen  
t. The three elements are combined to form a percentage. 0% to 59% yields re  
sponse 3, 60% – 80% and below yields response 2, and 81% to 100% yields resp  
onse 3. |
```

Error and status codes

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

| 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 t
he surf conditions for a specific beach. In this case, the code is just show
ing the overall recommendation about whether to go surfing.

```
```html  
<!DOCTYPE html>
<html>
<head>
<script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
<meta charset="utf-8">
<title>API Weather Query</title>
<script>

function getSurfReport() {

// use AJAX to avoid CORS restrictions in API calls.
var output = $.ajax({
 url: 'https://simple-weather.p.mashape.com/surfreport/123?units=imperia
l&days=1&time=1433772000',
 type: 'GET',
 data: {},
 dataType: 'json',
 success: function(data) {
 //Here we pull out the recommendation from the JSON object.
 //To see the whole object, you can output it to your browser consol
```

```
e using console.log(data);
 document.getElementById("output").innerHTML = data.surfreport[0].mon
day.2pm.recommendation;
},
error: function(err) { alert(err); },
beforeSend: function(xhr) {
 xhr.setRequestHeader("X-Mashape-Authorization", "W0yzMuE8c9mshcofZaBke3k
w7lMtp1HjVGAjsndqIPbU9n2eET"); // Enter here your Mashape key
}
});

}

</script>
</head>
<body>

<button onclick="getSurfReport()">See the surfing recommendation</button>
<div id="output"></div>

</body>
</html>
```

```

In this example, the `ajax` method from jQuery is used because it allows cross-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.

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.

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 Bootcamp interface. At the top, there is a search bar labeled "Search GitHub" and navigation links for "Pull requests", "Issues", and "Gist". Below the search bar, there is a notification bell icon and a user profile icon. The main content area is titled "GitHub Bootcamp" and contains four numbered steps:

- 1 Set up Git**: A quick guide to help you get started with Git.
- 2 Create repositories**: Repositories are where you'll work and collaborate on projects.
- 3 Fork repositories**: Forking creates a new, unique project from an existing one.
- 4 Work together**: Send pull requests, follow friends. Star and watch projects.

Below the bootcamp, there is a user profile for "tomjohson1492". A comment from "envygeeks" is shown, dated 4 hours ago, commenting on issue #3260. Another comment from "@kenold" is also visible. To the right, there is a sidebar for "Easier feeds for GitHub Pages" and a section for "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.

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.

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.

GUI version control clients

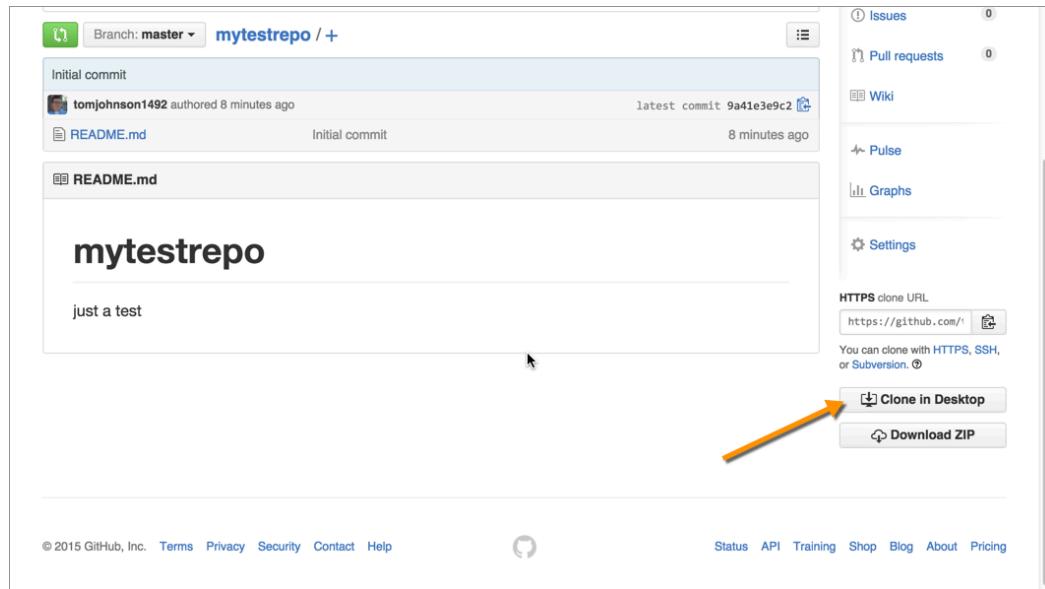
Although most developers use the command line when working with version control systems, there are many GUI clients available that may simplify the whole process. GUI clients might be especially helpful when you're trying to see what has changed in a file, since the GUI can better highlight and indicate the changes taking place.

You can also see changes in a text file format, but the >>>> and <<<< tags aren't always that intuitive.

Follow a typical workflow with a Github project using Github Desktop

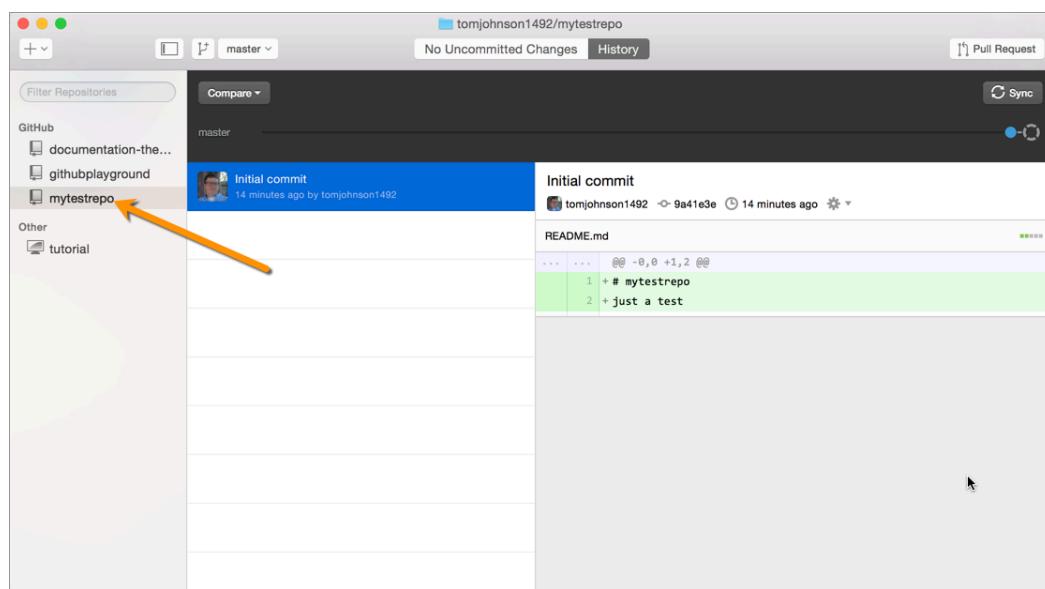
In this tutorial, you'll use Github Desktop to manage the workflow. First download and install [Github Desktop](#). You'll also need a Github account.

1. Go to [Github.com](#) and create a new repository from the the **Repositories** tab.
2. View your repository, and then click the **Clone in Desktop** button.

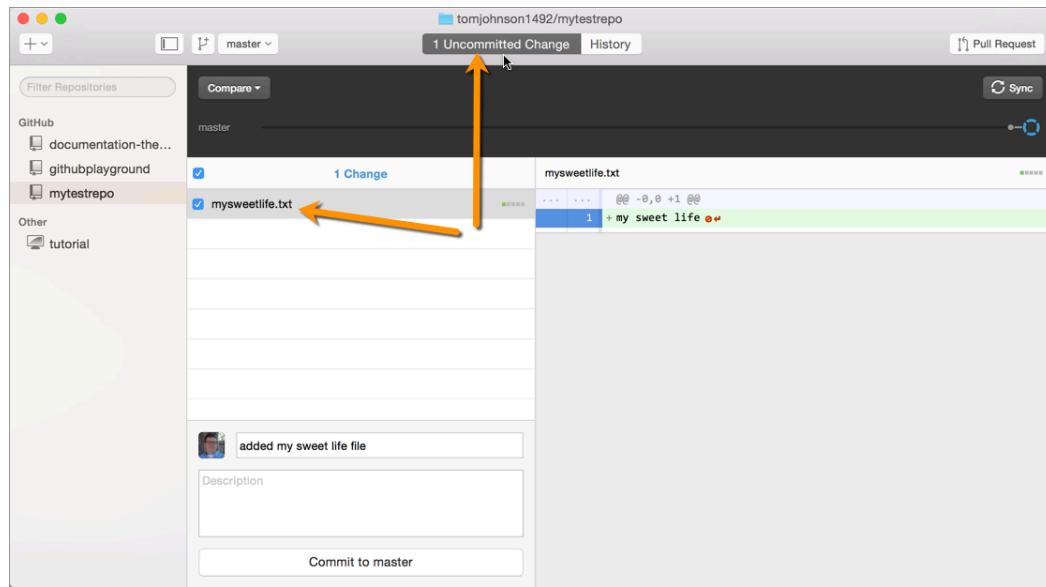


3. Select the folder where you want to clone the repository (such as under your username), and then click **Clone**.

GitHub Desktop should launch (you'll need to allow the application to launch, most likely) and add the newly created repository.



4. Go into the repository (using your Finder or browsing folders normally) and add a simple text file with some content.
5. Go back to GitHub Desktop and click the **Uncommitted Changes** link at the top.



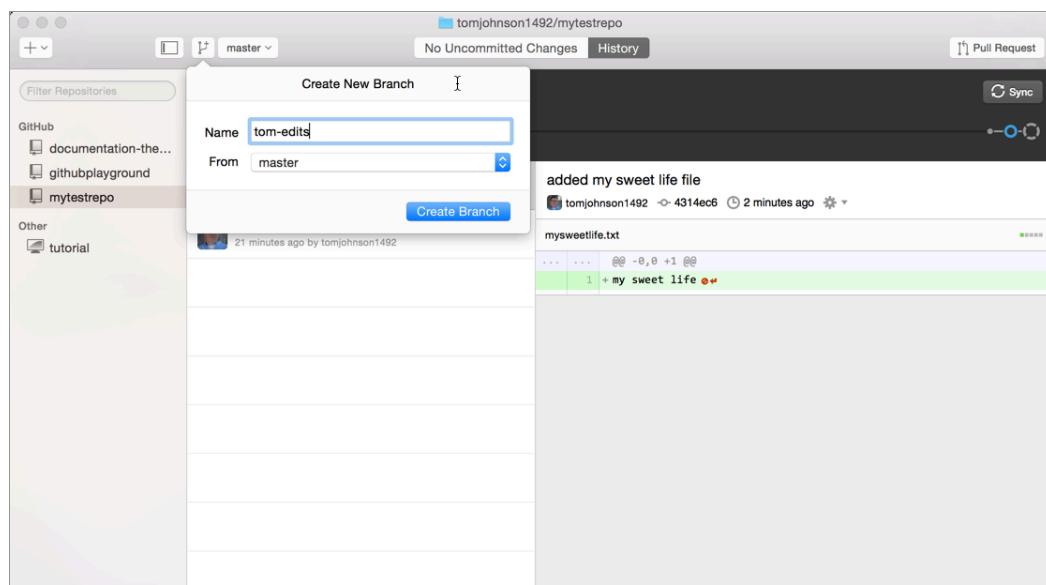
You'll see the new file you added in the list of uncommitted changes.

6. Type a commit message.
7. Click **Commit to Master**.
8. Click the **History** tab at the top. You can see the most recent commit there. If you view your repository online, you'll see that the change you made has been pushed to the master.

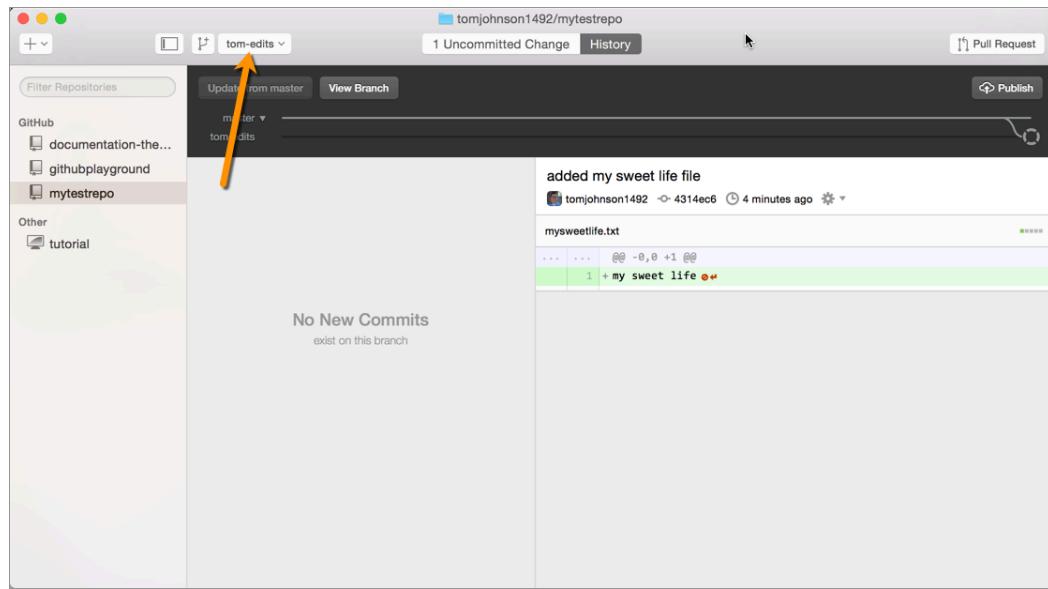
Create a branch

Now let's create a branch, make some changes, and then merge the branch into the master.

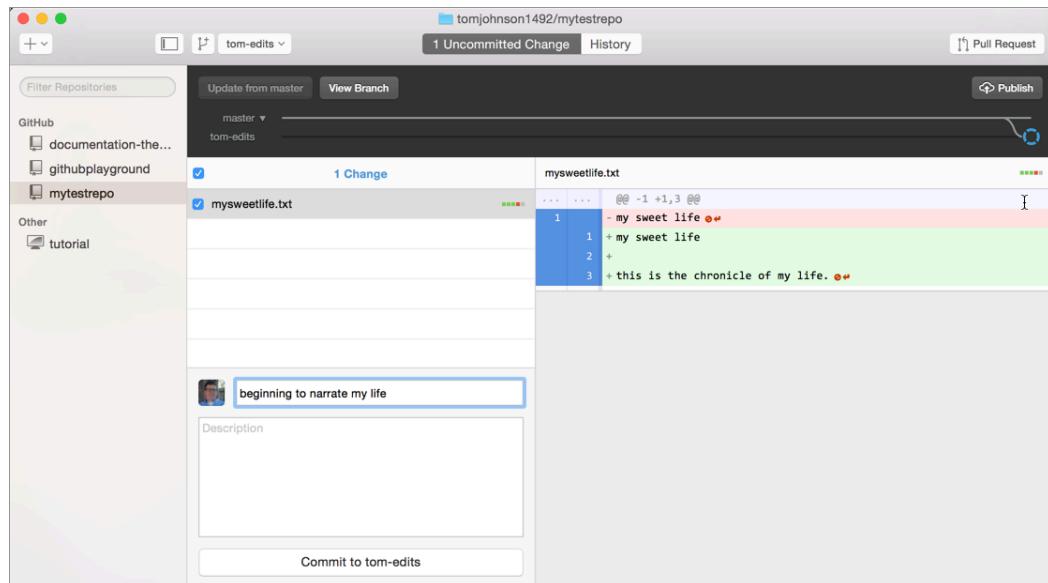
1. Click the **Add a branch** button and create a new branch. Call it something like "tom-edits," but use your own name.



When you create the branch, you'll see the branch drop-down menu indicate that you're working in that branch. A branch is a copy of the master that exists on a separate line. You can see that the visual line in Github Desktop branches off to the side when you create a branch.



2. Browse to the file you created earlier and make a change to it, such as adding a new line with some text.
3. Return to Github Desktop and notice that on the Uncommitted Changes tab, you have new modified files.



The right pane shows the deleted lines in red and new lines in green. This helps you see what changed.

However, if you switch to the master branch, you won't see the modified files. That's because you're working in a branch, and so your changes are associated with that branch. Switching this branch option in GitHub Desktop changes the working directory of your GitHub project to the branch.

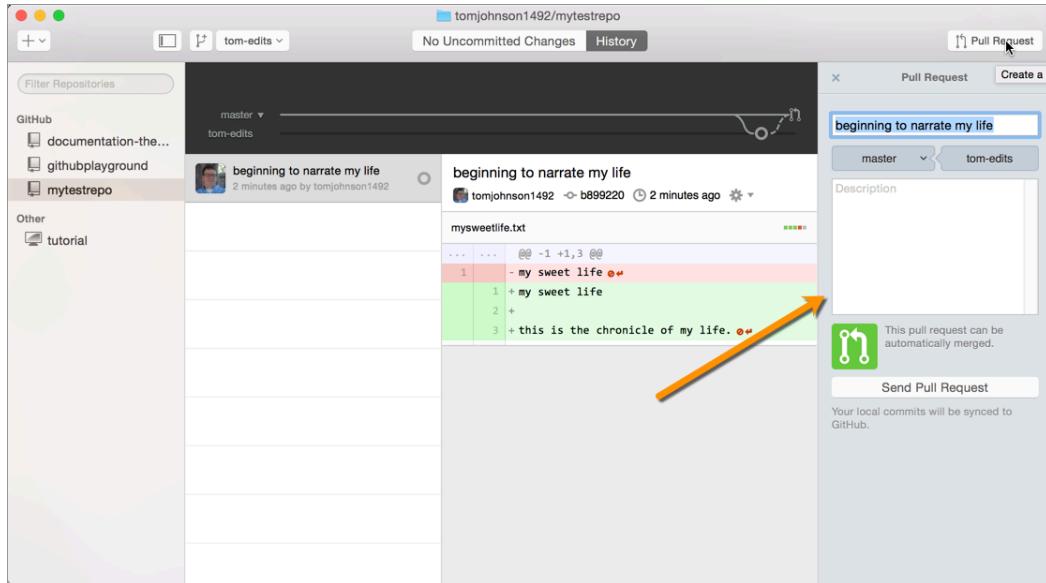
Switch back to your tom-edits branch.

Merge the branch through a pull request

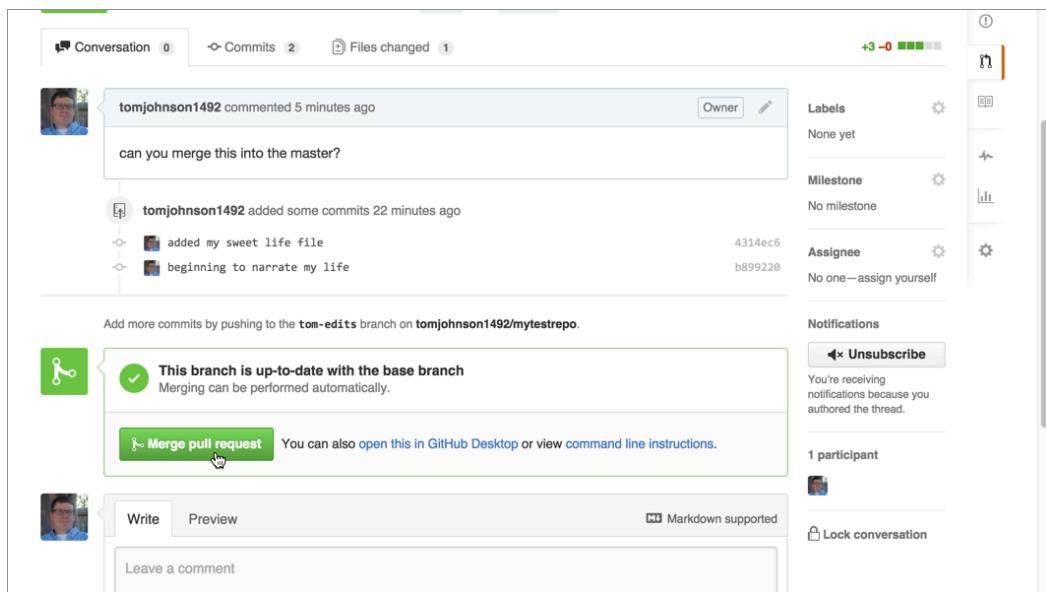
- Now let's merge the tom-edits branch into the master. Click the **Pull Request** button in the upper-right corner.

You're shown that you're merging the tom-edits branch into the master.

- Describe the pull request, and then click **Send Pull Request**.



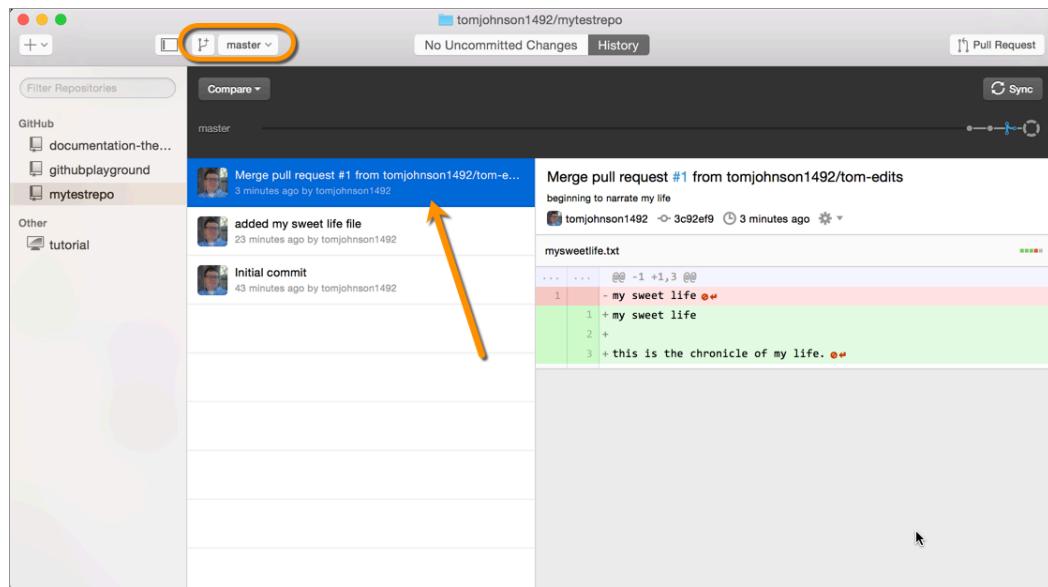
- Go to the link shown to evaluate the pull request online. In the browser interface, you can click the **Files changes** tab to see what files have changed in tom-edit that you are merging into the master.
- Click **Merge Pull Request**.



The branch gets merged into the master. You can delete the tom-edits branch now if you want.

- In your Github Desktop client, select the **master** branch, and then click the **Sync** button.

The Sync button pulls the latest changes from the master and updates your working copy to it. You will see the pull request merged. It shows you the lines that have been added in the files.



Managing conflicts

Suppose you make a change on your local copy of a file in the repository, and someone else changes the same file in conflicting ways and commits it to the repository first. What happens?

When you sync with the repository, you'll see a message prompting you to either discard your changes or to commit them before syncing.

"Syncing would overwrite your uncommitted changes. Please commit or discard your changes and try again."

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."

From the command line, if you run `git status`, it will tell you which files have conflicts. If you open the file with the conflicts, you'll see markers showing you the conflicts. It will look something like this:

```
<<<<< HEAD
I love carrots.
=====
I love bananas.
>>>>> origin/master
```

In this case, HEAD is your local change. Here you changed the line to "I love carrots." Origin/master shows the change someone else made and already committed to the master: "I love bananas."

Fix all the conflicts by adjusting the content between the content markers and then deleting the content markers.

Now you need to re-add the file to git again. To add a specific file:

```
git add home.md
```

To re-add all files:

```
git commit -a
```

Now make a commit and push it to the origin's master branch:

```
git commit -m "fixed conflicts"
```

Your options are the following:

- Run `git pull` to merge the other branch into yours, thereby resolving the conflict.

Pull request workflows through GitHub

In the previous step, you used Github Desktop to manage the workflow of committing files and creating requests. In this tutorial, you'll do a similar thing but using the browser-based interface that Github provides rather than using a terminal or Github Desktop.

When you ask developers to review content, ask the specific developer who created the feature you're documenting. Developer tasks are usually specific. One developer may not understand what another developer is really doing (beyond a superficial level).

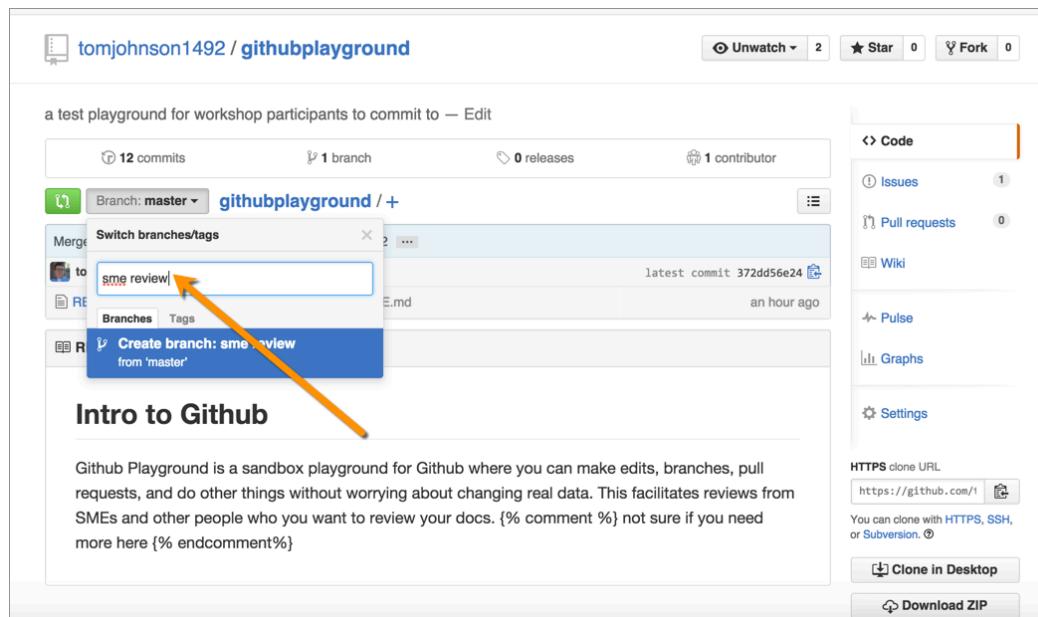
Make edits in a separate branch

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.

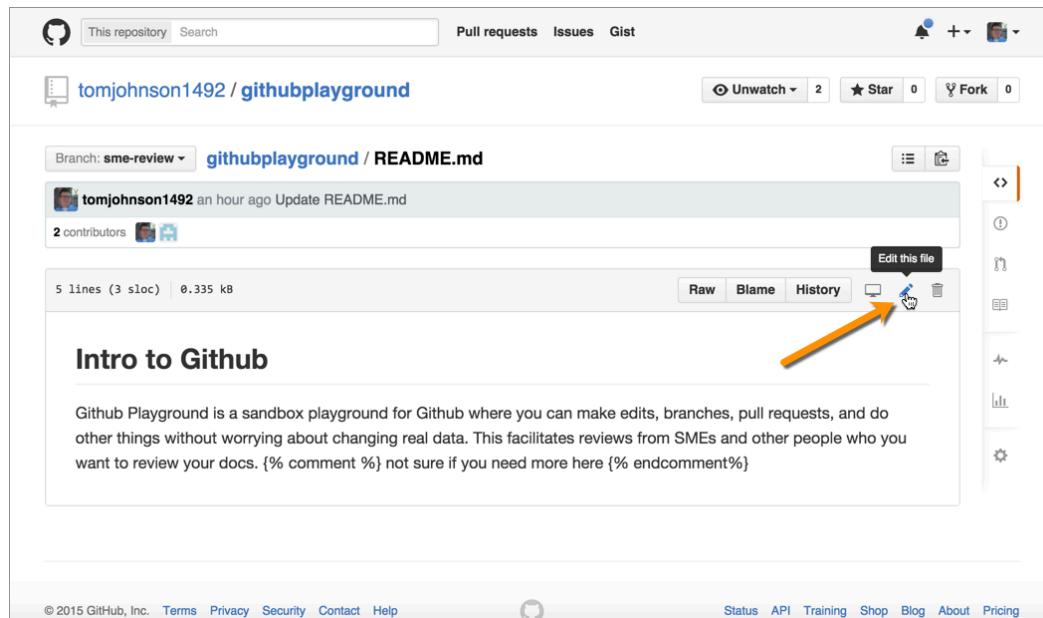
To make edits in a separate branch:

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."



When you create a new branch, the content from the master is copied over into the new branch. The branch is like doing a "Save as" with an existing document.

2. Click the **README.txt** file, and then click the **Edit this file** button (pencil icon) 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.

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.

This screenshot shows the GitHub repository 'tomjohansson1492/githubplayground'. The 'Pull requests' tab is selected. At the top right, there is a green button labeled 'New pull request'. Below it, a message says 'There aren't any open pull requests.' and provides search and filter options.

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

This screenshot shows the 'Comparing changes' screen for the repository. It displays a comparison between 'base: master' and 'compare: sme-review'. The 'sme-review' branch is selected as the head branch. The interface shows 1 commit from Sep 07, 2015, and a file named 'README.md' with 0 additions and 1 deletion. There are 'Unified' and 'Split' viewing modes at the bottom.

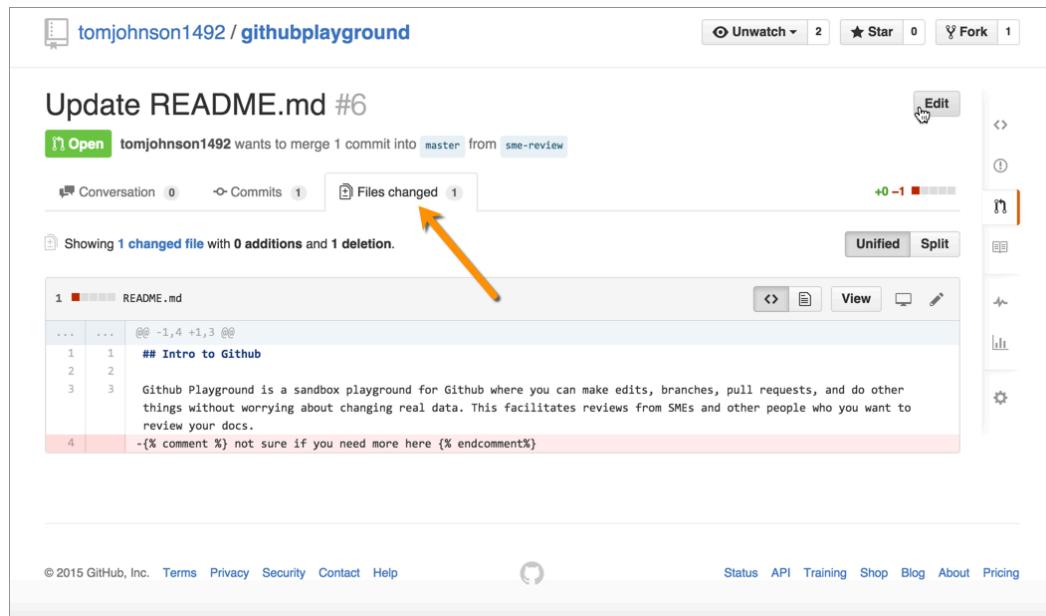
When you compare the branch against the master, you can see a list of all the changes. You can view the changes through two viewing modes: Unified or Split. Unified shows the edits together in the same content area, whereas split shows the two files side by side.

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.

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 a GitHub repository page. At the top, there's a search bar and navigation links for 'Pull requests', 'Issues', and 'Gist'. Below the header, the repository name 'tomjohansson1492 / githubplayground' is displayed, along with 'Unwatch' (2), 'Star' (0), and 'Fork' (1) buttons. The main area shows a list of branches under 'Default branch' and 'Your branches'. The 'sme-review' branch is listed under 'Your branches', showing it was updated an hour ago by tomjohansson1492. It has 1 merge and is labeled '#6 Merged'. A red arrow points to the 'Delete this branch' button next to it. Other branches like 'master' and another 'sme-review' branch are also listed. At the bottom, there are copyright notices for GitHub and 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 need to add collaborators to your Github project so they can commit edits to a branch. If someone isn't a collaborator and they want to make edits, they will receive an error.

If people don't have write access, they can fork the project 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. You can merge a forked repository, 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) on the lower-right.
2. Click the **Collaborators** tab on the left.
3. Type the Github usernames of those you want to have access in the Collaborator area.
4. Click **Add Collaborator**.

This screenshot shows the GitHub repository settings page for the repository `tomjohson1492/githubplayground`. The left sidebar has a 'Collaborators' section selected. In the main area, there is a search bar containing the text `saphira1410`. To the right of the search bar is a blue button labeled `Add collaborator`. An orange arrow points from the bottom-left towards this button. The top navigation bar includes links for `Pull requests`, `Issues`, and `Gist`. The top right corner shows icons for `Unwatch`, `Star`, `Fork`, and a bell icon.

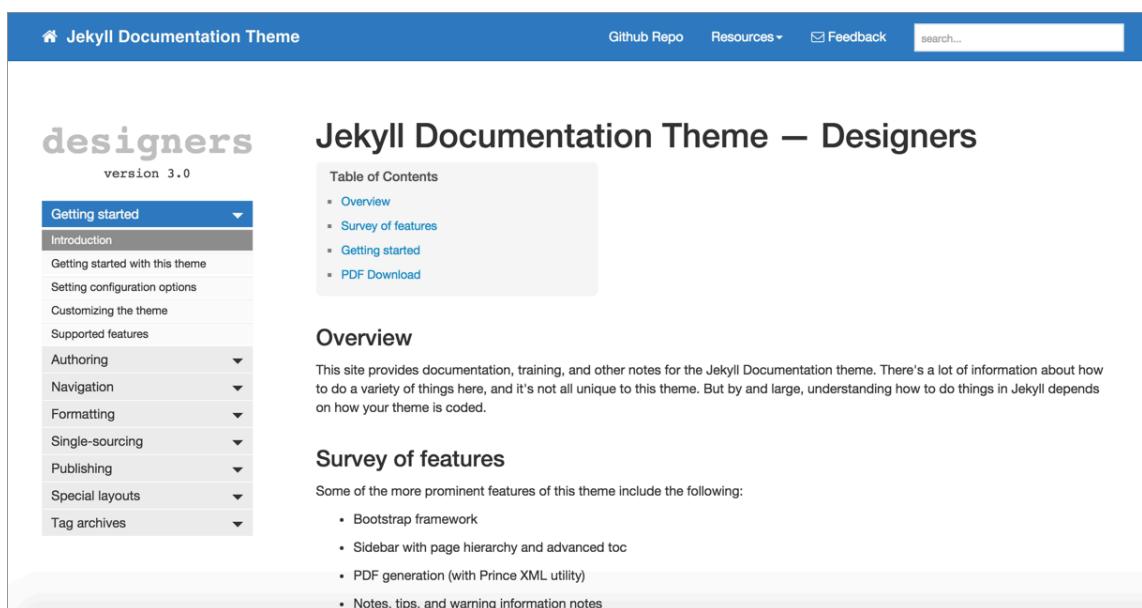
Jekyll — my favorite static site generator

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 website. There are more than 350 different static site generators. You can browse them at staticgen.com. The most popular static site generator (based on number of downloads, usage, and community) is Jekyll, so I'll be focusing on Jekyll here.

Jekyll

Jekyll is one of the most popular static site generators. All of my help content is on 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 the 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 "Jekyll Documentation Theme" and links for "Github Repo", "Resources", "Feedback", and a search bar. The main content area has a sidebar on the left with the title "designers" and "version 3.0". The sidebar contains a dropdown menu for "Getting started" which is currently open, showing options like "Introduction", "Getting started with this theme", "Setting configuration options", "Customizing the theme", and "Supported features". Below this are dropdown menus for "Authoring", "Navigation", "Formatting", "Single-sourcing", "Publishing", "Special layouts", and "Tag archives". The main content area on the right is titled "Jekyll Documentation Theme – Designers" and features a "Table of Contents" sidebar with links to "Overview", "Survey of features", "Getting started", and "PDF Download". The main content below the sidebar includes sections for "Overview" (describing the theme's documentation), "Survey of features" (listing features like Bootstrap framework, sidebar hierarchy, advanced TOC, PDF generation, and notes/tips), and "Getting started" (with a link to the "Getting started" section in the sidebar).

The screenshot shows the "Getting started" dropdown menu expanded, revealing sub-options such as "Introduction", "Getting started with this theme", "Setting configuration options", "Customizing the theme", and "Supported features". The main content area features a "Table of Contents" sidebar with links to "Overview", "Survey of features", "Getting started", and "PDF Download". The "Overview" section provides a brief introduction to the theme's documentation, while the "Survey of features" section lists various features like the Bootstrap framework, sidebar hierarchy, and PDF generation.

There isn't any kind of special API reference endpoint formatting here (yet), 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).

Whereas the Swagger, RAML, and API Blueprint REST specifications mainly just produce an interactive API console, with a static site generator, you have a tool for building a full-fledged website. With the 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:

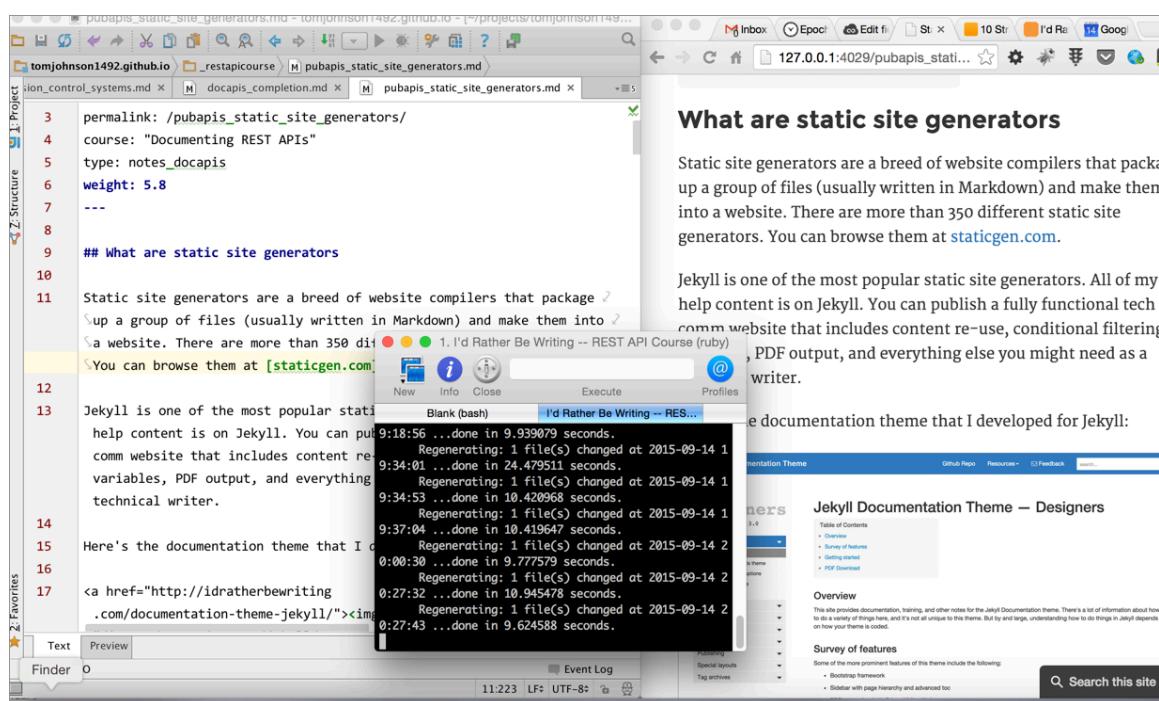
- Write in a text editor

- 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. When you have ordered list items separated by screenshots and result statements, and sometimes the result statements have lists themselves or note formatting, it can be a bit tricky to get the display right.

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 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.

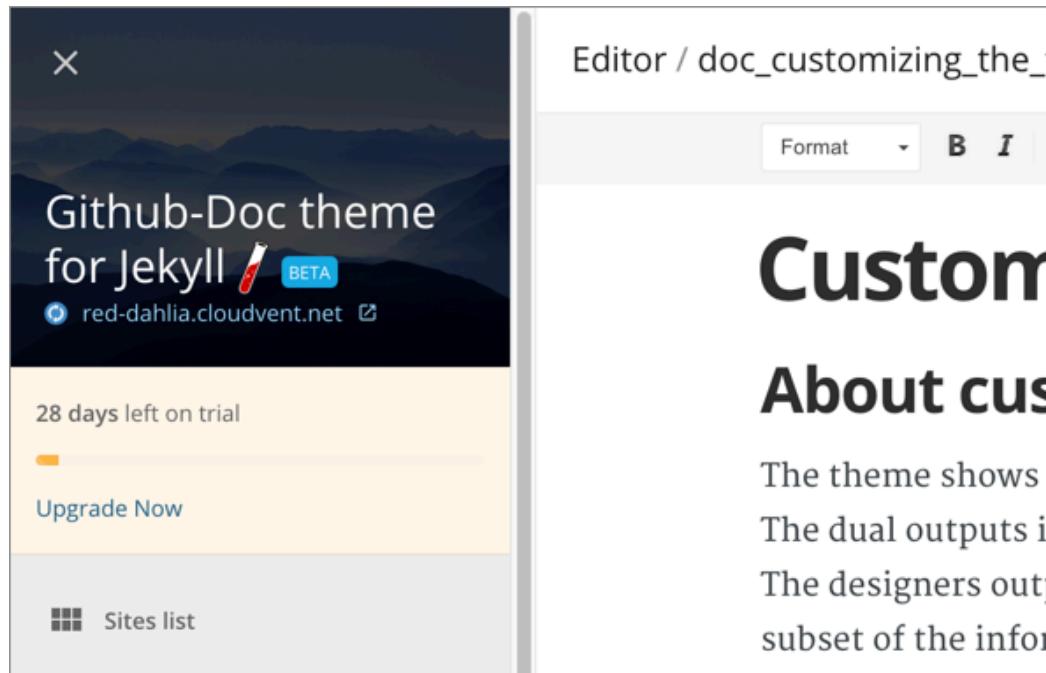
Set up your doc theme on Github

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**.
 4. While viewing your site, in the left sidebar, click **Site Settings**.
 5. On the **Details** tab, clear the **Minify and serve assets from CDN** check box. Then click **Update Site**. Why this step? The theme you'll be connecting to uses relative link paths, which don't play nicely with the CDN caching feature in CloudCannon.
 6. Click **Storage Providers** and then under Github, click **Connect**.
- You'll be taken to Github to authorize CloudCannon's access to your Github repository.
7. When asked which repository to authorize, select the **Documentation theme for Jekyll** repository.
 8. Select the default write direction for changes. The default is for changes made on Github to be pushed to CloudCannon, so the arrow (which represents changes) flow from Github to CloudCannon. That's the direction you want.
 9. Wait about 5 minutes for the files from your Github repository to sync over to CloudCannon. In the left sidebar, click **File Browser**. If you see a bunch of files with a green check mark, it means the files have synced over from the Github repo.
 10. View your CloudCannon site at the preview URL in the upper-left corner.



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

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** button (pencil icon), make an update, and then commit the update.

2. Wait a minute or so, and look for the change at the preview URL to your site on CloudCannon (refresh the page). 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.

What's cool about CloudCannon and Jekyll

Jekyll is a good solution because it provides near infinite flexibility and fits well within the UX web stack.

CloudCannon provides an easy way to allow subject matter experts to author and edit content, since CloudCannon allows you to create editable regions within your Jekyll theme. This would allow a tools team to maintain the site while providing areas for less technical people to author content.

However, CloudCannon wouldn't be a good solution if your docs require authentication in a highly secure environment. Additionally, Jekyll only provides static HTML files. If you want users to log in, and then personalize what they see when logged in, Jekyll won't provide this experience.

Publish the surfreport in the Aviator Jekyll theme using CloudCannon's interface

Let's say you want to use a theme that provides ready-made templates for REST API documentation. In this activity, you'll publish the weatherdata endpoints in a Jekyll theme called Aviator. Additionally, rather than syncing the files from a Github repository, you'll just work with the files directly in CloudCannon.

The [Aviator API documentation theme](#) by Cloud Cannon is designed for REST APIs. You'll use this theme to input a new endpoint. If you're continuing on from the previous sections (Documenting REST APIs), you already have a new endpoint called surfreport.

Documentation

- [Getting Started](#)
- Authentication
- Errors

APIs

- /books GET
- /books POST
- /books/:id GET
- /books/:id PUT
- /books/:id DELETE

Getting Started

An intro to our API

Welcome to our API!

This API document is designed for those interested in developing for our platform.

Warning

Something terrible will happen if you try and do this.

This API is still under development and will evolve.

Authentication

You need to be authenticated for all API requests. You can generate an API key in your developer dashboard.

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

If you're on a Mac (with Rubygems and Jekyll installed), building Jekyll sites is a lot simpler. But even if you're on Windows, it won't matter for this tutorial. You'll be using CloudCannon, a SaaS website builder product, to build the Jekyll files.

a. Download the Jekyll Aviator theme

1. Go to [Aviator API documentation theme](#) and click the **Download ZIP** button.

CloudCannon / Aviator-Jekyll-Theme

A Jekyll theme for API Documentaiton

4 commits 1 branch 0 releases 1 contributor

branch: master

Aviator-Jekyll-Theme / +

File	Commit	Time Ago
_api	Init	5 months ago
_documentation	Added another layout	4 months ago
_includes	Added another layout	4 months ago
_layouts	Added another layout	4 months ago
_more	Init	5 months ago
css	Added another layout	4 months ago
img	Init	5 months ago
js	Added another layout	4 months ago
.gitignore	Init	5 months ago

latest commit c9d4340f92

Issues 0
Pull requests 0
Wiki
Pulse
Graphs

HTTPS clone URL
<https://github.com/>

You can clone with **HTTPS**, **SSH**, or **Subversion**.

Clone in Desktop **Download ZIP**

2. Unzip the files.

b. Add the weatherdata endpoint doc to the theme

1. Browse to the theme's files. In the _api folder, open 1_1_books_list.md in a text editor and look at the format.

In every Jekyll file, 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.

2. Create a new file called 1-6_weatherdata.md and save it in the same _api folder.
3. Get the data from the weatherdata endpoint from this [Weather API on Mashape](#). Put the data from this endpoint into the Aviator theme's template.

The Aviator Jekyll theme has a specific layout that will be applied to all the files inside the _api folder (these files are called a collection). Jekyll will access these values by going to api.title, api.type, and so forth. It will then push this content into the template (which you can see by going to _layouts/multi.md).

Here's what my 1-6_weatherdata.md file looks like. Be sure to put the response within square brackets, indented with one tab (4 spaces). Remove the `raw` and `endraw` tags at the beginning and end of the code sample (which I had to add to keep Jekyll from trying to process it).

```
---
```

```
title: /weatherdata
type: get
description: Get weather forecast by Latitude and Longitude.
parameters:
  title: Weatherdata parameters
  data:
    - lat:
        - string
        - Required. Latitude.
    - lng:
        - string
        - Required. Longitude.
right_code:
return: |
[
{
  "query": {
    "count": 1,
    "created": "2014-05-03T03:57:53Z",
    "lang": "en-US",
    "results": {
      "channel": {
        "title": "Yahoo! Weather - Tebrau, MY",
        "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau__MY/*ht
tp://weather.yahoo.com/forecast/MYXX0004_c.html",
        "description": "Yahoo! Weather for Tebrau, MY",
        "language": "en-us",
        "lastBuildDate": "Sat, 03 May 2014 11:00 am MYT",
        "ttl": "60",
        "location": {
          "city": "Tebrau",
          "country": "Malaysia",
          "region": ""
        },
        "units": {
          "distance": "km",
          "pressure": "mb",
          "speed": "km/h",
          "temperature": "C"
        },
        "wind": {
          "chill": "32",
          "direction": "170",
          "speed": "4.83"
        },
        "atmosphere": {
          "humidity": "66",
          "pressure": "982.05",
          "rising": "0",
          "visibility": "9.99"
        }
      }
    }
  }
}
```

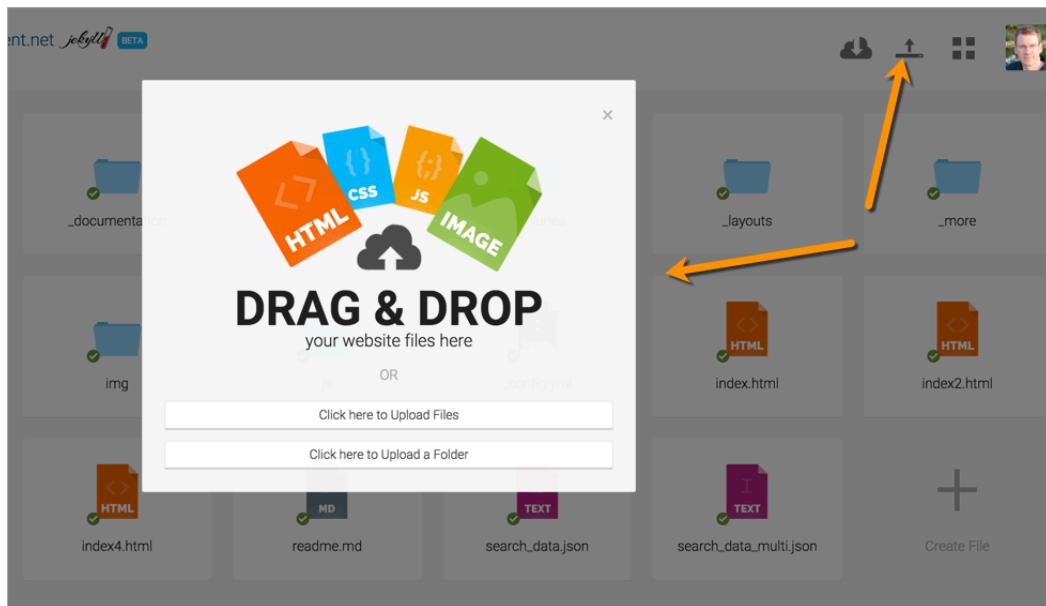
```
        },
      "astronomy": {
        "sunrise": "6:57 am",
        "sunset": "7:06 pm"
      },
      "image": {
        "title": "Yahoo! Weather",
        "width": "142",
        "height": "18",
        "link": "http://weather.yahoo.com",
        "url": "http://l.yimg.com/a/i/brand/purplelogo//uh/us/news-wea.gi
f"
      },
      "item": {
        "title": "Conditions for Tebrau, MY at 11:00 am MYT",
        "lat": "1.58",
        "long": "103.74",
        "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau__M
Y/*http://weather.yahoo.com/forecast/MYXX0004_c.html",
        "pubDate": "Sat, 03 May 2014 11:00 am MYT",
        "condition": {
          "code": "28",
          "date": "Sat, 03 May 2014 11:00 am MYT",
          "temp": "32",
          "text": "Mostly Cloudy"
        },
        "description": "\n<img src=\"http://l.yimg.com/a/i/us/we/52/28.gi
f\"/><br />\n<b>Current Conditions:</b><br />\nMostly Cloudy, 32 C<BR />\n<B
R /><b>Forecast:</b><BR />\nSat - Scattered Thunderstorms. High: 32 Low: 2
6<br />\nSun - Thunderstorms. High: 33 Low: 27<br />\nMon - Scattered Thunde
rstorms. High: 32 Low: 26<br />\nTue - Thunderstorms. High: 32 Low: 26<br
/>\nWed - Scattered Thunderstorms. High: 32 Low: 27<br />\n<br />\n<a href
=\"http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau__MY/*http://weathe
r.yahoo.com/forecast/MYXX0004_c.html\">Full Forecast at Yahoo! Weather</a><B
R/><BR/>\n(provided by <a href=\"http://www.weather.com\">The Weather Chann
el</a>)<br />\n",
        "forecast": [
          {
            "code": "38",
            "date": "3 May 2014",
            "day": "Sat",
            "high": "32",
            "low": "26",
            "text": "Scattered Thunderstorms"
          },
          {
            "code": "4",
            "date": "4 May 2014",
            "day": "Sun",
            "high": "33",
            "low": "27",
            "text": "Scattered Thunderstorms"
          }
        ]
      }
    }
  }
}
```

```
        "text": "Thunderstorms"
    },
    {
        "code": "38",
        "date": "5 May 2014",
        "day": "Mon",
        "high": "32",
        "low": "26",
        "text": "Scattered Thunderstorms"
    },
    {
        "code": "4",
        "date": "6 May 2014",
        "day": "Tue",
        "high": "32",
        "low": "26",
        "text": "Thunderstorms"
    },
    {
        "code": "38",
        "date": "7 May 2014",
        "day": "Wed",
        "high": "32",
        "low": "27",
        "text": "Scattered Thunderstorms"
    }
],
"guid": {
    "isPermaLink": "false",
    "content": "MYXX0004_2014_05_07_7_00_MYT"
}
}
}
}
]

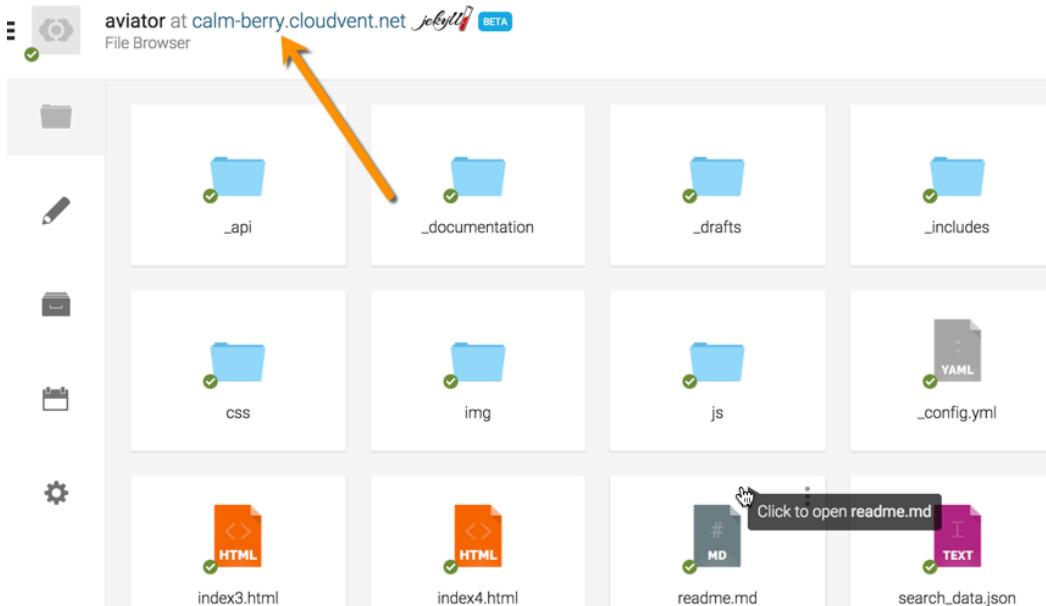
---  
  
<div class="code-viewer">  
  
<pre data-language="cURL">  
curl --get --include 'https://simple-weather.p.mashape.com/weatherdata?la  
t=1.0&lng=1.0' \  
-H 'X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p' \  
-H 'Accept: application/json'  
</pre>  
  
</div>
```

c. Publish your Jekyll project on CloudCannon

1. Go to <http://cloudcannon.com> and, if you don't already have an account, sign up for a free test account by clicking **Sign Up**.
2. After signing up and logging in, click **Create Site**.
3. Log in and click **Create Site**.
4. Type a name for the site (e.g., Aviator Test) and press your **Enter** key.
5. Click the **Upload Files** button in the upper-right corner.



6. Open your Aviator theme files, select them all, and drag them into the upload file dialog box. (Don't just drag the Aviator theme folder into CloudCannon).
7. After the files finish uploading (and little green check marks appear next to the files), click the preview link in the upper-left corner:



8. When prompted for a password for viewing the site to add, add one.

9. Click the preview link to view the site.

The site should appear as follows:

The screenshot shows a Jekyll site's documentation page. On the left, there's a sidebar with links to 'Documentation', 'Getting Started', 'Authentication', 'Errors', and an 'APIs' section. Under 'APIs', there's a single endpoint: '/weatherdata' (GET). Below this, there are sections for 'Surfreport parameters' and 'curl'. The 'curl' section contains a command-line example:

```
curl --get --include 'https://simple-weather.p.mashape.com/weatherdata?lat=1.0&lon=1.0' \
-H 'X-Mashape-Key: EF393pKnmshgokR83V6JB6QyP1cGrrdjsnczTkkYgYrp8p' \
-H 'Accept: application/json'
```

On the right, there's a large code editor window showing a JSON response from the API. The response includes details like 'query', 'count', 'created', 'lang', 'results', 'channel', 'title', 'link', and 'description'.

You can see my site at <http://delightful-nightingale.cloudvent.net/>. The password is `stcsummit`.

If your endpoint doesn't appear, you probably have invalid YML syntax. Make sure the left edge of the response is at least one tab (4 spaces) in.

Each time you save the site, CloudCannon actually rebuilds the Jekyll files into the site that you see.

If you switch between the code editor and visual display, the code sample gets mangled. (The CloudCannon editor will convert the https path into a link.) This is a bug in CloudCannon that will be fixed.

Doc Websites Using Jekyll

Here are some websites using Jekyll:

- <http://dev.iron.io/>
- <http://healthcare.gov>
- <http://getbootstrap.com>
- <http://iamnotarealprogrammer.com/>
- <http://jekyllrb.com/>
- <http://docs.balsamiq.com>

Other tool options — a miscellaneous compilation

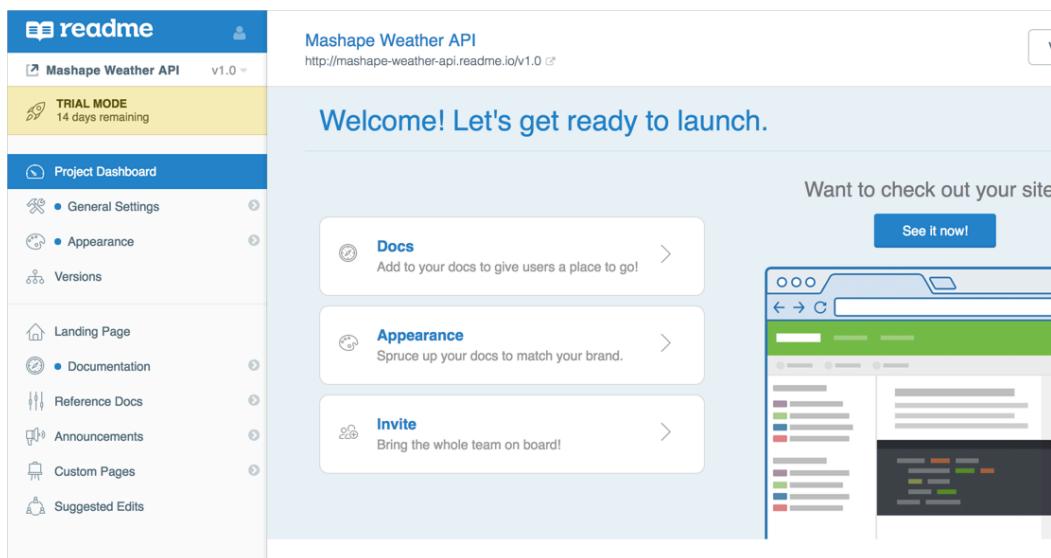
There are many different tools for creating documentation, and there's no clear industry standard in this space. Different tools may better suit different environments, skill sets, and products. On this page, I've listed as many authoring tools as I can find that are related to the developer documentation space.

Readme.io

[readme.io](#) is an online hosted options for docs that offers one of the most robust, full-featured interfaces and options for developer docs available. If you consider how much time it requires to build, maintain, troubleshoot, etc., your own website, it really does make sense to consider an existing third-party platform where someone has already built all of this out for you.

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**.



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

Although you can add your API information manually, you can also import an [OpenAPI \(page 259\)](#) file. You can experiment by choosing one from the [OpenAPI examples](#), such as [this one](#).

Readme.io provides a number of wizard-like screens to move you through the 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

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 Readme output provides an interactive, try-it-out experience with endpoints:

Documentation

Try It Out

lat* 37.3708905 lon* -121.9675525

GET https://simple-weather.p.mashape.com/weatherdata Try It!

200 OK

```
{ "query": { "count": 1, "created": "2015-06-17T06:23:38Z", "lang": "en-US", "results": { "channel": { "title": "Yahoo! Weather - Santa Clara, CA", "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*h ttp://weather.yahoo.com/forecast/USCA1018_c.html", "description": "Yahoo! Weather for Santa Clara, CA", "language": "en-us", "lastBuildDate": "Tue, 16 Jun 2015 8:53 pm PDT" } } }
```

The experience is similar to Swagger in that the response appears directly in the documentation. This API Explorer gives you a sense of the data returned by the API.

Readme.io is a pretty sweet platform, and you don't have to worry about describing your API based on a specification based on either RAML or Swagger. But this also has downsides. It means that your doc is tied to the Readme.io platform. Additionally, you can't auto-generate the output from annotations in your source code.

Additionally, if the cloud location for your docs isn't an option, that may also pose challenges. Finally, there isn't any content re-use functionality, so if you have multiple outputs for your documentation that you're single sourcing, Readme.io may not be for you.

Even so, the output is sharp and the talent behind this site is top-notch. The platform is constantly growing with new features, so maybe all of this functionality will eventually be there.

Here are a few sample API doc sites built with Readme.io:

- [Validic](#)
- [Box API](#)
- [Coinbase API](#)

Miredot

[Miredot](#) is one of the tools you can use to generate reference API documentation from a Java source.

Miredot is a plugin for Maven, which is a build tool that you integrate into your Java IDE. Miredot can generate an offline website that looks like this:

The screenshot shows a web-based API documentation interface for a 'Petstore' application. At the top, it says 'MireDot Petstore-1.6.1' and 'Issues (18)'. On the right, there are 'Resources' and 'Search' buttons. A sidebar on the right lists a 'catalog' with a 'category' section containing endpoints for 'GET', 'POST', 'PUT', and 'DELETE' methods, along with links for 'item', 'product', 'sales', and 'test'. The main content area displays the 'Welcome to the MireDot demo project' page. It includes a 'GET ALL CATEGORIES' section with a green button labeled 'GET http://mydomain.com/catalog/category/'. Below this, there's a 'Returns' section showing a JSON response example:

```

[{"subCategory": "cats", "dogs": "dogs"}]
  
```

At the bottom right of the returns section is a 'Hide descriptions' link.

You can read the [Getting started guide](#) for Miredot for instructions on how to incorporate it into your Java project.

Miredot supports many annotations in the source code to generate the output. The most important annotations they support include those from Jax-rs and Jackson. See [Supported Frameworks](#) for a complete set of supported annotations.

Here's an example of what these annotations look like. Look at the [CatalogService.java](#) file. In it, one of the methods is `updateCategory`.

You can see that above this method is a “doc block” that provides a description, the parameters, method, and other details:

```
/**
 * Update category name and description. Cannot be used to edit products
 * in this category.
 *
 * @param categoryId The id of the category that will be updated
 * @param category The category details
 * @summary Update category name and description
 */
@PUT
@Path("/category/{id}")
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public void updateCategory(@PathParam("id") Long categoryId, Category category);
```

Miredot consumes this information and generates an output.

To explore Miredot’s output:

1. Browse the [Miredot sample code](#).
2. To see how this information gets rendered in the Miredot output, go to the [Petstore example docs](#), expand **Catalog > Category** on the right, and then select **PUT**. Or go directly [here](#).

The screenshot shows the Miredot API documentation interface. At the top, it says "UPDATE CATEGORY NAME AND DESCRIPTION" and "expand / collapse all". Below that is a description: "Update category name and description. Cannot be used to edit products in this category." A blue button labeled "PUT" is followed by the URL "http://mydomain.com/catalog/category/{id}/". Under "Path parameters", there is a parameter "id" described as "The id of the category that will be updated" and "number". Under "Body", there is a "The category details" section. The "Accept" header is set to "application/xml,application/json". The body content is a JSON object with fields: "subCategory" (with values "cats" and "dogs" highlighted in blue), "description" (with a note "//the new description"), "name" (with a note "//The name of the category"), "recommendedProduct" (with a note "//A product that is recommended for this category [...]"), and "type" (with a note ".PromotionalProduct"). A "Hide descriptions" link is located in the top right corner of the body content area.

If you browse the navigation of Miredot’s output, it’s an interesting-looking solution. This kind of documentation might fit a Java-based REST API well.

But the authoring of the docs would really only work for Java developers. It wouldn’t work well for technical writers unless you’re plugged into the source control workflow.

Run in Postman button

The [Run in Postman button](#) provides a **Run in Postman** button that, when clicked, imports your API info into Postman so users can run calls using the Postman client.

To try out Run in Postman, first [import your OpenAPI spec into Postman](#) or enter your API information manually. Then see [Create the Run in Postman button](#).

You can see the many [demos here](#).

For a demo using the sample Mashape weather API, see the [Postman section \(page 242\)](#) in “Tool options for developer docs.”

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 (including sample responses and the model they follow).

Spectacle

[Spectacle](#) is a Github project that builds an output from a Swagger 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 Swagger file.

Here’s a [demo output](#). You can also see an [output that uses the Mashape weather API file](#).

With almost no needed setup or configuration, you can have a world-class output and site for your API docs. As long as the OpenAPI spec that you integrate is fully detailed, the generated Spectacle site will be attractive and full-featured.

You can also build the Spectacle site without the frame so you can embed it into another site. However, in playing with this embed option, I found that I would 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.

I’m also not sure if you can add your own doc pages to the Spectacle site.

Custom UX solutions

If you want to build a beautiful API doc website that rivals sites such as [Parse.com](#) (a custom-built solution that uses Prism.js, Sinatra, and other in-house tools), you’ll most likely need to involve a UX engineer to build it. Fortunately, building an API docs site is a solution that many UX engineers and other web developers are usually excited to tackle.

If you want to integrate your API documentation into your main website, ask the person designing your main website for strategies on integrating the doc site into it. This integration might allow you to leverage authentication (if needed) and other interaction points (such as with forums or support tickets).

Many custom websites are built using a variety of JavaScript, HTML, and CSS tools. Most likely you'll be able to supply a batch of Markdown or HTML files to the web developer to integrate. Your UX developers will often be eager to design a custom solution to make your docs beautiful and seamlessly integrated with the rest of your content.



When I worked at Badgeville, our solution for publishing API documentation was to use custom scripts that pulled some information from source files and pushed them into templates.

The source files were stored on Github, and the writers could edit the descriptions of the parameters, fields, etc. Our developers created scripts that would look into the code of the source files and render content into JSON files in a specific structure.

Since we published all help content on a Drupal site, we hired a Drupal development agency that would take information from a JSON file and push the information into a custom-built template.

After the scripts were integrated into the Drupal site, we would have developers periodically run the build scripts to generate a batch of JSON files.

The upload scripts checked to ensure the JSON files were valid, and then they were pushed into the templates and published. Each upload would overwrite any existing content with the same file names.

If your documentation is published on a web-based CMS, you can probably find a development agency to create a similar script (if you don't have in-house engineers to create them).

A lot of companies have custom solutions for their API documentation. Sometimes this kind of solution just makes sense and allows you to right-size the workflow to fit your specific information.

Tools that can read the OpenAPI specification

Many commercial API tools and platforms can read the OpenAPI specification and generate an interactive documentation website. See the [list of tools on Swagger here](#).

More tools

I'll be adding more tools here in the days to come...

- readthedocs.org/com
- [mkdocs](#)
- [hugo](#)
- [slate](#)
- [dexy](#)

<http://apidocjs.com/>

What about traditional help authoring tools (HATs)?

In contrast to [docs-as-code tools \(page 197\)](#), 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 use these tools to create API documentation. There are pros and cons with HATs.

Example

Here's a sample help output from Flare for the Photobucket API:

The screenshot shows a help documentation interface for the Photobucket API. The left sidebar contains a table of contents (TOC) with sections like 'Using the Help', 'What's New', 'Getting Started', 'Examples', 'Methods', 'Albums', and various sub-sections under 'Albums'. The main content area shows the 'Create New Album' page. At the top, it says 'You are here: Photobucket API Help > Methods > Albums > Create New Album'. Below this is a 'Create New Album' section with the sub-instruction 'Create a new sub-album.' Underneath, there is a 'User Login Required' note and a link to 'End-User Authentication'. The 'HTTP Method' is listed as 'POST'. The 'REST Path' is '/album/[identifier]'. A note says 'See Object Identifiers for an [identifier] explanation.' The 'Parameters' table is shown below:

Parameter	Optional	Description	Variable
identifier	N	Album identifier.	String
name	N	Name of result. Must be between 2 and 50 characters. Valid characters are letters, numbers, underscore (_), hyphen (-), and space.	String

The 'Request Example' section shows a POST request to 'http://api123.photobucket.com/album/**pbapi**' with parameters: 'format=xml&name=thisnewalbum&oauth_consumer_key=0000000000&oauth_nonce=ff84a82e65a6a517e753f5d34846c940&SHA1&oauth_timestamp=1236627874&oauth_token=20.000000_1236627874&oauth_version=1.0'.

The 'Response Example' section is partially visible at the bottom.

Pros of using a help authoring tool

Some advantages of using a HAT include the following:

+ Comfortable authoring environment for writers

If writers are going to 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.

+ Integrates reference information with guides and tutorials.

You won't have a division between an output that is generated from a reference doc generator (such as Swagger) and user guide material. It can be one seamless whole.

+ HATs reinforce the fact that API doc is more than endpoints

HATs reinforce the fact that good API documentation is more than just a set of endpoints and parameters. Good API documentation includes guides and tutorial topics as well. Developers rarely write that kind of information, yet it's just as important as the reference documentation. HATs lend themselves more to these guide and tutorial topics.

Cons of using a help authoring tool

Some disadvantages of using a HAT include the following:

- Most HATs don't run on Macs

Almost no HAT runs on a Mac. But many developers and designers prefer Macs because they have a much better development platform (the command line is much friendlier and functional, for example). To make a cURL call, you just open Terminal and paste in the call. With a Windows machine, installing cURL and libcurl is much more onerous and harder to use.

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.

- Dated UI won't help sell the product

The output from a help authoring tool usually looks dated. The problem with the dated tripane look and feel is that API documentation *is* the interface that users navigate. There isn't a separate GUI interface that the help complements. The help is front and center as the information product that users get.

If you want to promote the idea that your API is modern and awesome, you want a website that looks modern and awesome as well. 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 look and feel.

- Doesn't integrate with other site components

Many of the API doc sites are single-website experiences. The API docs are usually part of the main website, not a link that opens in its own window and frame, separate from the other content.

If you can output a format that another site can consume, great. But if you split and divide the user into separate sites, you're following a less common pattern with API doc sites.

- 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. (Then again, if you don't expect developers to contribute, then this becomes a moot point.)

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 challenges:

- 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.

To handle PDF, 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. Other writers might use [Pandoc](#) to fill simpler PDF requirements.

To handle authentication, we uploaded the HTML output into a Salesforce site.com space and used Salesforce as the authentication layer. The custom uploading was my least favorite part of the solution, but a more integrated authentication solution (potentially using [One Login](#) or a custom authentication solution) would have required more engineering resources.

My point is that you can handle these more challenging factors with non-traditional tools, but it requires more expertise.

Tools versus content

Although this course 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.

Once you get the tooling infrastructure in place, it should mostly take a back seat to the daily tasks of content development.



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. In some ways, the design should be invisible and unobtrusive, not foregrounding the focus on the content. In other words, the user shouldn't be distracted by the tooling.

For the most part, 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, incorporate language switchers to jump from platform to platform – the feedback you'll get is, "This sentence is incorrect." Or, "There's a typo here."

At this point, think about your requirements, your audience, and try to pick the right tools for your situation. Here are a few questions to consider as you think about the right tool for you:

- Will developers be writing or contributing to the content?
- Does your security group restrict you from using third-party platforms to host documentation?
- Do you have a budget to pay a third-party platform for hosting?
- Do you want to manage the web platform details yourself or offload this onto another group/company?
- How many endpoints do you have to document?
- Should you push documentation from the source into your documentation?
- Does the documentation need to be visible on the web, or does it need to be private?
- To what extent do you want customers to have a one-stop-shopping experience – reading docs, logging support tickets, posting to forums, viewing news?
- Do you have UX resources to help build a custom solution?

Case study: Switching tools to docs-as-code

For an overview of the docs as code approach, see [Docs-as-code tools \(page 197\)](#).

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, and 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 a new approach, such as docs as code.

Additionally, you will likely need to make this change outside your normal documentation work, and you will need to develop the new system while still updating and publishing content in the old system.

In this article, I'll describe the challenges I faced into when implementing a docs-as-code approach within Amazon's Appstore and Alexa publishing group. Note that Amazon has a lot of different tech writing groups and tools. At the time of this writing, I worked in a small group of about 10 writers focused on third-party developer documentation for the Amazon Appstore. We published content on developer.amazon.com/documentation. (This is not the same group as AWS or the groups that create end-user documentation for products.)

Previous processes

Previously, the Appstore doc team published content through a content management system called Hippo (now called "Bloomreach".) Hippo was similar to WordPress or Drupal but was Java-based rather than PHP-based (which made it attractive to Java-centric enterprise).

To publish a page of documentation, we tech writers had to create a new page in the Hippo CMS and then paste in our HTML for the page (or try to use the WYSIWYG editor in the CMS, which was disastrous). 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. It was painstaking, 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. The layout included a documentation-specific sidebar (previously absent in Hippo) 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 Hippo CMS. Thus, we were half way with our docs-as-code approach but still had room to go. One of the tenets of docs-as-code is to build your output directly from the server. 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 bandwidth for the project. Once it started, the project was a wild ride of mismatched expectations and understandings. In the end, we succeeded. Most of the lessons learned here are about this process, specifically how we transitioned to building Jekyll directly from the Git repo, the decisions we made and the reasoning behind those decisions, the compromises and other changes of direction, and so on.

Advantages of Hooking into a Larger System

Why did we want to move to docs as code in the first place? At a large company like Amazon, there are plenty of robust, internally developed tools that we wanted to take advantage of. The docs-as-code approach would allow us to integrate into this infrastructure.

Other documentation tools are often independent, standalone tools that offer complete functionality (such as version control, search, and deployment) within their system. But their system is a black box, meaning, you can't really open it up and integrate it into another process or system. With docs-as-code, we had the flexibility to adapt our process to fully integrate in with Amazon's infrastructure. Some of this infrastructure included the following:

- Internal test environments
- Authentication for specific pages based on account profiles
- Search and indexing
- Website templating
- Analytics
- Secure servers with the amazon.com domain
- Media CDN for distributing images
- Git repositories and GUI for managing code
- Build pipelines and build management system

All we really needed to do was generate out the body HTML along with the sidebar and make it available for the existing processes 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.

See the [docs-as-code topic \(page 198\)](#) for more general reasons.

End Solution

In the end, here's what we produced. We stored the Jekyll project in an internal Git repository — the same farm of Git repositories other engineers used for nearly every software project, and which hooked into a build management system. When we pushed our Jekyll doc content to the master branch of our Git repository, a build pipeline would kick off and build the Jekyll project from the server (similar to GitHub Pages).

Our Jekyll layout omitted any header or footer. The built HTML pages were then pulled into an S3 bucket in AWS. This bucket acted somewhat like 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 15 minutes, but tech writers didn't need to do anything during that time. After you typed a few commands in your terminal, the process kicked off and ran all by itself.

From the first release, pretty much everyone loved this new approach. 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 4am to 6am, 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 master branch, merged the master to production. Fifteen 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 all felt like a huge burden had been lifted off our shoulders.

Challenges we faced

Transitioning to the new system was challenging on a number of levels. I'll detail the 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. This is probably my biggest weakness 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. Even so, as project timelines kept getting delayed and pushed out, and we weren't sure if the engineering team's bandwidth would ever lighten up, I wanted to jump ship and just deploy/publish everything myself through the S3_website plugin on AWS S3.

But as I researched domain policies, server requirements, and other environment 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 developer.amazon.com domain, corporate policies required us to host the content on a tier 1 server. It became clear that this would involve a lot more engineering knowledge and time than I had, so we had to wait. We wanted to get this right, because we probably wouldn't get bandwidth from the engineering team again for some time. In the end, waiting turned out to be the right approach.

Understanding each other

When we did finally begin the project, 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 really understand our Jekyll authoring process and needs.

Conversely, we didn't understand the engineer's world very 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. But whereas I had in mind a doghouse, they had in mind a skyscraper. So their processes were probably 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 they were in Seattle and I was in Sunnyvale.

Figuring out repo size

Another challenge was to figure out the correct size for the documentation repos. Across our teams, we probably had 30 different products, each with their doc navigation and pages. 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.

However, storing all content in the same repo simplified single sourcing of content across projects, made link management easier, reduced maintenance efforts, and more. (What I didn't realize was the work required to hook each repo into a build pipeline, because I was unfamiliar with the build management workflows.)

We 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 reported 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, of course. But soon the long build times (1-2 minutes for each build) became painful. We also ran into size warnings in our repo (all images were included in the repos), and sometimes merge conflicts happened.

The long build times were so annoying, we eventually decided to switch back to individual repos. I came up with creative ways (first through git submodules, then through git subtrees) to push the theme files out to small repos in a semi-automated way. However, what constituted a "product" was somewhat unclear, and even a small product that had just a few pages was stored in its own repo.

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. Eventually we settled on two Git repos and two pipelines. We had to reconsolidate all the separate repos back into two repos. 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 in both repos. This was easier than trying to devise an automated method.

Although 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 take the recommendations and advice about various decisions much more seriously. I didn't assume they misunderstood our authoring needs and requirements so much, and instead followed their direction more respectfully.

Limitations in the build management system

Another challenge was in dealing with limitations in the build management system. Our build management system was an engineering tool used to build outputs or other artifacts from source code stored in git repositories. My understanding of the build management system was limited, since this was an engineering-heavy tool used for managing software and other code. It was not a documentation management tool.

Engineers had trouble running Bundler in the build process on the server. Bundler is a tool that automatically gets the right RubyGems for your Jekyll project based on the Jekyll version you are using. Without Bundler, each writer just installed the jekyll gem locally and built their Jekyll project that way.

Ideally, you want everyone on the team using the same version of Jekyll to build their projects (specifically, the `jekyll` gem used from RubyGems). This ensures that local builds match the builds from the server. However, the latest supported version of Jekyll in the build management system was an older version of Jekyll (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. (Bundler would have helped us stay in sync with Jekyll versions.) However, 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.

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. In theory, getting everyone to sync both their local build processes with the way the server builds is key to avoiding surprises when the server builds your output. You don't want to later discover the some lists didn't render correctly or some code samples didn't highlight correctly because of a mismatch of gems.

Figuring out translation workflows

Figuring out the right process for translation was also difficult. (I believe this process still has room to evolve and mature as well.) 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 which vendors could even be used. Eventually, we decided to revert to sending only HTML to vendors.

However, if you send only the HTML output from Jekyll to vendors, it makes it difficult to apply updates. With Jekyll (and most static site generators), your sidebar and layout is packaged into each of your individual doc pages. If you add a new page to your sidebar, or update any aspect of your layout, you would need to edit each individual file instance to make those updates across the documentation.

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. A bit of manual labor, but acceptable.

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.

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.

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. Satisfying the requirements from the engineering team required us to create workarounds and hacks in our Jekyll theme.

Other challenges

There were a handful of other challenges worth mentioning (but not worth full development in their own sections). I'll 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 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 handled these redirects at the server level. I created a script that iterated through our sidebars and generated out a list of old and new URLs in a JSON format.
- **Implementing new processes while still supporting the old.** While the 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 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 your content into a new system while continuing to publish in the older system, without making updates in both places, is a testament to the flexibility of Jekyll.
- **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. 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 too much in flux and didn't like to hear about updates or changes 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.
- **Styling the tech docs within a larger site.** 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). 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 dismay among the tech writers. The engineers weren't happy that we had the ability to break the display of content with our layout code. At the same time, we wanted the ability to push out content that relied on jQuery or other scripts.

- **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 on the command line and version control workflows. Some writers already had this background, 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 variety you can choose from). 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.
- **Striking a balance between simplicity and robustness in doc tooling.** We had to support a nearly impossible requirement: 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, 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, you would have to understand Git, Markdown, and Jekyll. For non-technical users, we directed them to a GUI they could interact with (similar to GitHub's GUI) to make edits in the repository. Then we would merge in and deploy their changes.
- **Building a system that scales.** Although we were using open source tools for this solution, the system had to be able to scale in an enterprise way. I think our system will scale nicely. Because the content used Markdown as the format, anyone can 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 are required to scale out the authoring. A writer just uses Atom editor (or another IDE). The writer opens up the project and works with the text, treating docs like code.

Conclusion

Almost everyone on the team was happy about the way our doc solution turned out. Of course, there were always areas for improvement. But the existing solution was head and shoulders above the previous processes.

I outline the challenges here to reinforce that implementing docs-as-code is no small undertaking. It doesn't have to be an endeavor that takes months, but at a large company, if you're integrating with engineering infrastructure and building out a process that will scale and grow, it can require a decent amount of engineering expertise and effort.

If you're implementing docs-as-code at a small company, you can simplify processes and use a system that meets your needs. For example, you could use the [S3_website plugin](#) to publish on AWS S3, or better yet, host your docs on GitHub and publish through GitHub Pages. 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.

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

For more reading about docs as code, see Anne Gentle's book [Docs Like Code](#).

OpenAPI specification and Swagger

Overview of REST API specification formats.....	258
Introduction to the OpenAPI spec and Swagger.....	259
OpenAPI tutorial overview	270
OpenAPI Tutorial Step 1: openapi object.....	274
OpenAPI Tutorial Step 2: info object	277
OpenAPI Tutorial Step 3: servers object	279
OpenAPI Tutorial Step 4: paths object.....	281
OpenAPI Tutorial Step 5: components object.....	292
OpenAPI Tutorial Step 6: security object	314
OpenAPI Tutorial Step 7: tags object.....	317
OpenAPI Tutorial Step 8: externaldocs object	319
OpenAPI specification activity: Create your own specification document.....	322
Swagger UI tutorial	323
Swagger UI activity: Create your own Swagger UI docs	331
Swagger UI Demo	333
Integrating Swagger UI with the rest of your docs	334
SwaggerHub introduction and tutorial	340
More about YAML	348
RAML tutorial	352
API Blueprint tutorial.....	363

Overview of REST API specification formats

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

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 \(page 0\)](#), 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 \(page 155\)](#) 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 \(page 334\)](#).

Introduction to the OpenAPI specification and Swagger

For step-by-step tutorial on creating an OpenAPI specification document, see the [OpenAPI tutorial here \(page 270\)](#).

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.

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

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"
```

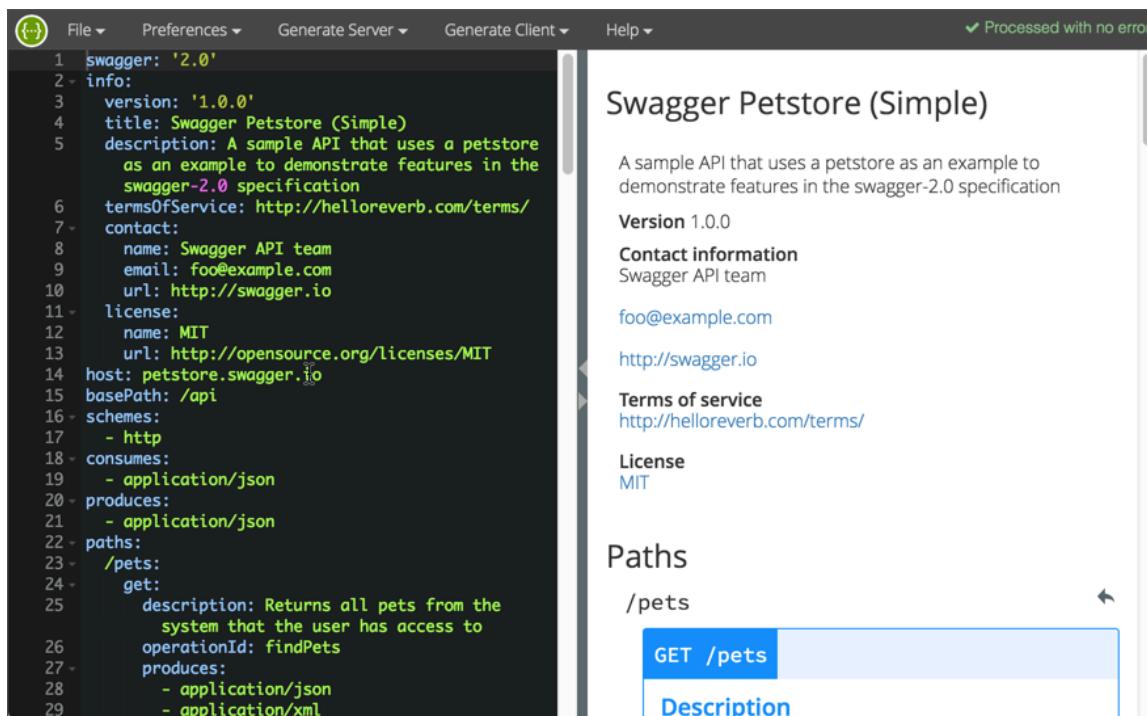
This [YAML code](#) comes from the [Swagger Petstore demo](#).

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, a code editor displays a YAML-based OpenAPI specification for a "Swagger Petstore (Simple)" API. The specification includes details like the API version (2.0), contact information (Swagger API team, email foo@example.com, URL http://swagger.io), and a license (MIT). It also defines a single endpoint at /pets with a GET method that returns all pets. On the right, the UI shows the title "Swagger Petstore (Simple)" and a brief description: "A sample API that uses a petstore as an example to demonstrate features in the swagger-2.0 specification". Below this, it lists the "Version 1.0.0", "Contact information", "License", and the "Paths" section. Under "Paths", the "/pets" endpoint is shown with its "GET /pets" method highlighted. A tooltip for this method provides a "Description" of "Returns all pets from the system that the user has access to".

```

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. See [Comparison of Automatic API Code Generation Tools For Swagger](#) by API Evangelist. He mentions [Swagger Codegen](#), [REST United](#), [Restlet Studio](#), and [APIMATIC](#).

These libraries, specific to your programming language, will parse through your code's annotations and generate a 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 spec. 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.

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 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 \(page 340\)](#) (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 help 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 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).

Rendering Your API Docs 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 API documentation. 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 base URL: '[Base URL: petstore.swagger.io/v2] [http://petstore.swagger.io/v2/swagger.json]'. A message below states: 'This is a sample server Petstore server. You can find out more about Swagger at <http://swagger.io> or on [irc.freenode.net, #swagger](#). For this sample, you can use the api key **special-key** to test the authorization filters.' There are links for 'Terms of service', 'Contact the developer', 'Apache 2.0', and 'Find out more about Swagger'. On the left, a 'Schemes' dropdown is set to 'HTTP'. On the right, there's an 'Authorize' button with a lock icon. The main content area shows two sections: 'pet' (Everything about your Pets) and 'store' (Access to Petstore orders). The 'store' section contains four API endpoints: 'GET /store/inventory' (Returns pet inventories by status), 'POST /store/order' (Place an order for a pet), 'GET /store/order/{orderId}' (Find purchase order by ID), and 'DELETE /store/order/{orderId}' (Delete purchase order by ID). The 'DELETE' endpoint is highlighted with a red background.

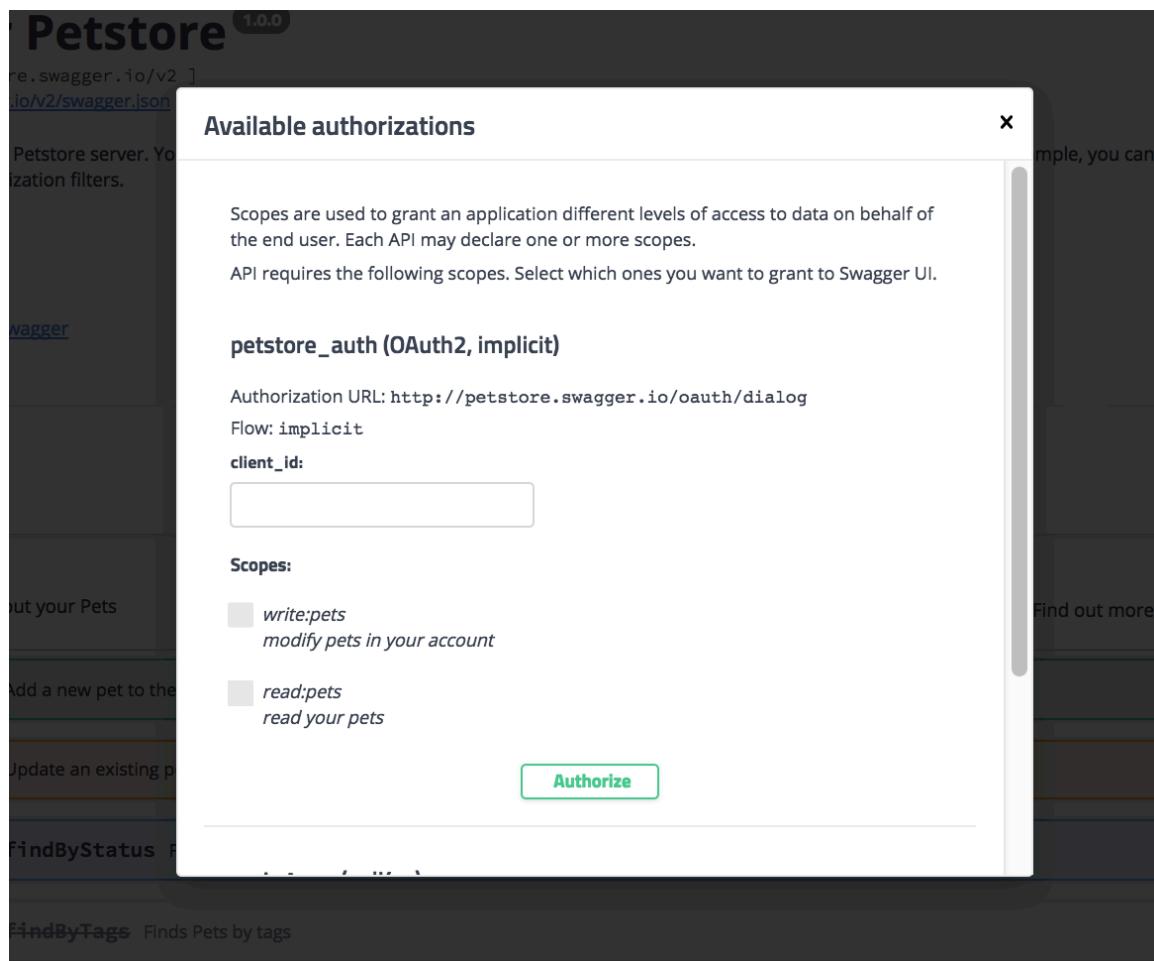
Also check out the [sample Swagger UI integration with a simple weather API](#) used as a course example.

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**.

pet Everything about your Pets

POST /pet Add a new pet to the store

Find out more: <http://swagger.io>

Parameters

Name **Description**

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

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. 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

Cancel

Parameters

Name **Description**

body * 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.)

The screenshot shows the Swagger UI interface. At the top, there's a 'Responses' section and a dropdown for 'Response content type' set to 'application/json'. Below that is a 'Curl' section containing a command to post a pet to the /pet endpoint. The 'Request URL' is listed as <http://petstore.swagger.io/v2/pet>. Under 'Server response', there's a table with two rows. The first row has 'Code' 200 and 'Details' 'Undocumented'. The second row shows the 'Response body' which is a JSON object:

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

The Petstore is a functioning API, and you have actually created a pet. For fun, expand the GET `/pet/{petId}` endpoint, click Try it out, enter the pet ID you used in the previous operation, and then execute the request. You should see your pet's name returned.

Other tools for reading OpenAPI spec

There are other tools besides Swagger UI that can parse your OpenAPI specification document. Some of these tools include [Restlet Studio](#), [Apiary](#), [Apigee](#), [Lucybot](#), [Gelato/Mashape](#), [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 \(page 323\)](#)
- [OpenAPI specification tutorial \(page 270\)](#)
- [Swagger/OpenAPI specification](#)
- [Swagger2postman](#)
- [Swagger-ui Responsive theme](#)
- [Swagger-ui](#)
- [Undisturbed REST: A Guide to Designing the Perfect API](#), by Michael Stowe

OpenAPI 3.0 tutorial overview

In the [Swagger tutorial \(page 323\)](#), 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 [Mashape Weather 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 \(page 340\)](#). 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 \(page 383\)](#).)

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 \(page 274\)](#)
- [Step 2: info object \(page 277\)](#)
- [Step 3: servers object \(page 279\)](#)

- [Step 4: paths object \(page 281\)](#)
- [Step 5: components object \(page 292\)](#)
- [Step 6: security object \(page 314\)](#)
- [Step 7: tags object \(page 317\)](#)
- [Step 8: externalDocs object \(page 319\)](#)

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. 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:

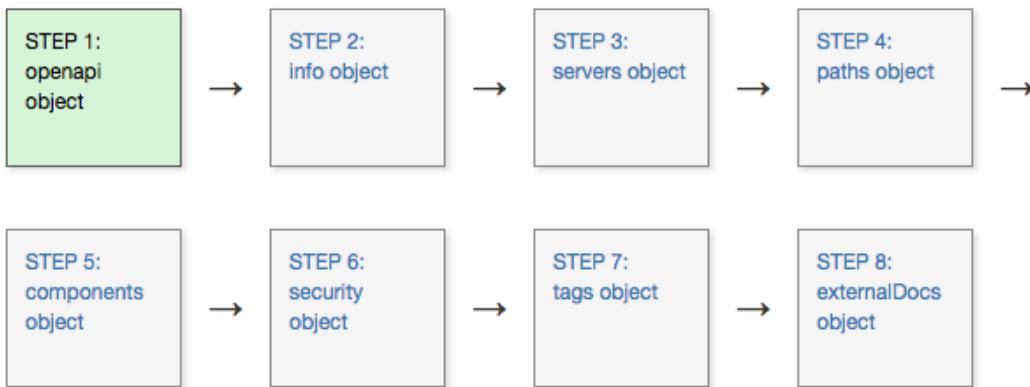
- [Swagger UI with Mashape Weather API](#)
- [Embedded Swagger with Mashape Weather API \(page 333\)](#)

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](#).

OpenAPI tutorial step 1: The openapi object



OpenAPI tutorial overview

Before diving into the first step of the OpenAPI tutorial here, read the [OpenAPI tutorial overview \(page 270\)](#) 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 \(page 30\)](#) 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 Mashape Weather 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 \(page 292\)](#) 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 [3.0.0](#).

```
openapi: "3.0.0"
```

3.0 was released in July 2017, so much of the information and examples online, as well as supporting tools, often relate to 2.0.

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 Weather API. At the top, there's a green header bar with the Swagger logo, the URL `/learnapidoc/docs/rest_api_specifications/openapi_weather.yml`, and a 'Explore' button. Below the header, the title 'Weather API from Mashape' is displayed with a '1.0' version indicator and an 'OAS3' badge. A black arrow points to the 'OAS3' badge. Below the title, there's a brief description of the API and its purpose. Underneath the description, there's a note about API keys and examples of latitude and longitude values. At the bottom of the page, there are links for 'Terms of service', 'fyhao - Website', 'Send email to fyhao', and 'Limited license'.

Validator errors

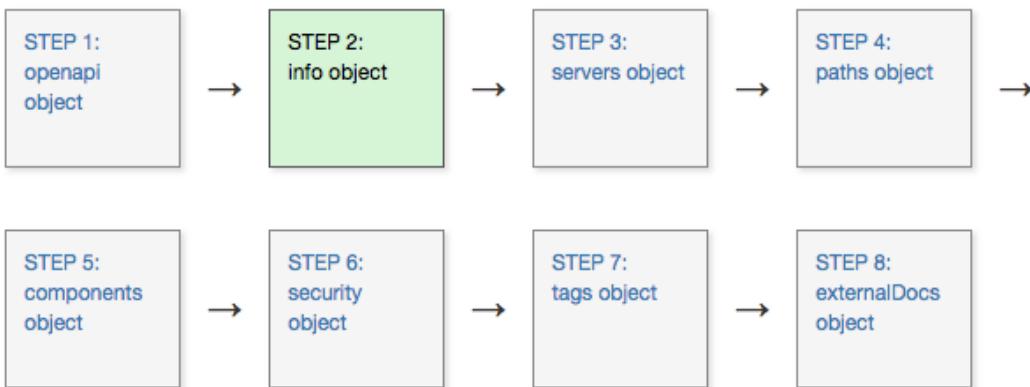
If your spec doesn't validate, the Swagger UI display often won't load the content or will show an error. For example, if you have an incorrect indentation in your YAML syntax, an error message might appear that indicates a `bad indentation of a mapping entry`. You can click the **Error** button in the lower right to see more information.

The screenshot shows the Swagger UI after a validation error has occurred. The main content area is redacted. In the top right corner, there's a red box containing the error message: 'Parser error on line 7' followed by 'bad indentation of a mapping entry'. To the right of this message is a 'Hide' button. At the bottom right of the UI, there's a red button labeled 'ERROR' with a red icon.

Clicking this error button takes you to 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.

OpenAPI tutorial step 2: The info object



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: Weather API from Mashape
  description: "This is a sample spec that describes a Mashape Weather API as an example to demonstrate features in the Swagger-2.0 specification. This output is part of the <a href=\"http://idratherbewriting.com/learnapidoc\">Documenting REST API course</a> on my site. The Weather API displays forecast data by latitude and longitude. It's a simple weather API, but the data comes from Yahoo Weather Service. The weatherdata endpoint delivers the most robust package of information of the endpoints here.\n\nTo explore the API, you'll need an API key. You can sign up for an API through Mashape, or you can just use this one: `EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p`. For the latitude and longitude parameters, you can get this information from the URL of a location on Google Maps. For example, for Santa Clara, California, use the following:\n* **lat**: `37.3708698`\n* **lng**: `-122.037593` \n"
  version: "1.0"
  termsOfService: https://konghq.com/terms/
  contact:
    name: fyhao
    url: https://market.mashape.com/fyhao
    email: some_email@gmail.com
  license:
    name: Limited license
    url: https://konghq.com/terms/
```

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.

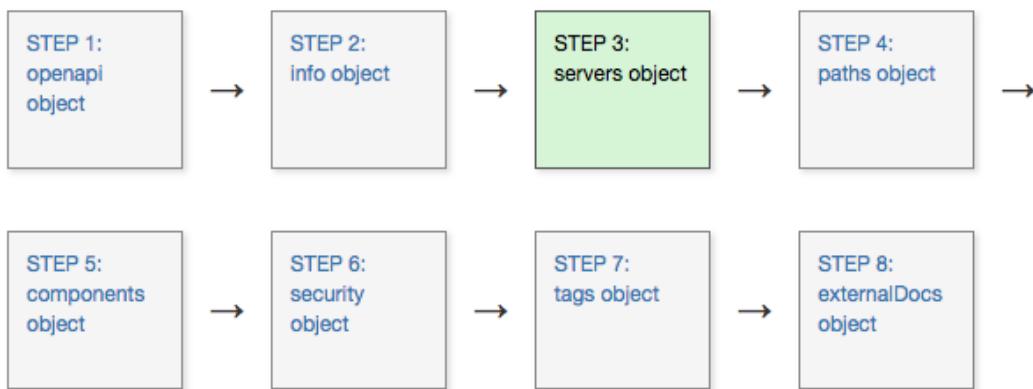
Appearance in Swagger UI

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

The screenshot shows the Swagger UI interface for the Weather API from Mashape. At the top, there is a green header bar with the 'swagger' logo, the URL '/learnapidoc/docs/rest_api_specifications/openapi_weather.yml', and a 'Explore' button. Below the header, the title 'Weather API from Mashape' is displayed, along with '1.0' and 'OAS3' badges. A link to the YAML specification is provided: '/learnapidoc/docs/rest_api_specifications/openapi_weather.yml'. The main content area contains a brief description of the API, mentioning it is a sample spec for the Mashape Weather API, part of the 'Documenting REST API course', and that it displays forecast data by latitude and longitude. It notes the data comes from Yahoo Weather Service. Below the description, there is a note about API keys and examples for location parameters (lat: 37.3708698, lon: -122.037593). At the bottom of the main content area, there are links for 'Terms of service', 'fyhao - Website', 'Send email to fyhao', and 'Limited license'.

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.

OpenAPI tutorial step 3: The servers object



In the `servers` object, you specify the basepath used in your API requests. The basepath is the part of the URL that appears before the endpoint.

Sample servers object

The following is a sample `servers` object:

```
servers:
- url: https://simple-weather.p.mashape.com
  description: Production server
```

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 `/weatherdata`, when Swagger UI submits the request, it will submit it to `{server URL}{path}` or `https://simple-weather.p.mashape.com/weatherdata`.

Options with the server URL

You have some flexibility and configuration options for your server URL. You can specify multiple server URLs that might relate to different environments (test, beta, production). If you have multiple server URLs, users can select the environment from a servers drop-down box. For example, you can specify multiple server URLs like this:

```
servers:
- url: https://simple-weather.p.mashape.com
  description: Production server
- url: https://beta.simple-weather.p.mashape.com
  description: Beta server
- url: https://some-other.simple-weather.p.mashape.com
  description: Some other server
```

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

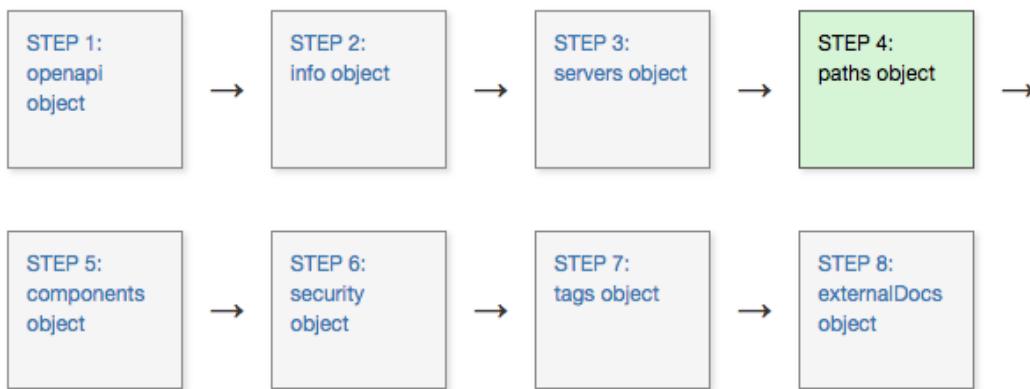
The screenshot shows the Swagger UI interface for an API. At the top, there is a sidebar with links: 'Terms of service', 'fyhao - Website', 'Send email to fyhao', 'Limited license', and 'Full documentation for the API'. On the right side, there is a green 'Authorize' button with a lock icon. Below these, there is a section labeled 'Servers' with a dropdown menu. The dropdown menu contains three options: 'https://simple-weather.p.mashape.com' (which is checked), 'https://beta.simple-weather.p.mashape.com', and 'https://some-other.simple-weather.p.mashape.com'. 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 the 'https://simple-weather.p.mashape.com' entry in the dropdown. The main content area shows an 'Air Quality' section with a 'GET /aqi getAqi' button and a lock icon.

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 281) 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.

OpenAPI tutorial step 4: The paths object



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 Mashape Weather API, there are just 3 paths, each with the `get` operation:

```
paths:  
  /aqi:  
    get:  
  
  /weather:  
    get:  
  
  /weatherdata:  
    get:
```

Operation Objects

Each path item object contains an `operation object`. Operations are the GET, POST, PUT, and DELETE methods we [explored in endpoints definitions and methods \(page 92\)](#). 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 \(page 292\)](#)).
- **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 \(page 292\)](#)).
- **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 \(page 314\)](#).
- **servers** (object): A servers object that might differ for this path than the [global servers object \(page 279\)](#).

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:  
  /aqi:  
    get:  
      tags:  
      summary:  
      description:  
      operationId:  
      externalDocs:  
      parameters:  
      responses:  
      deprecated:  
      security:  
      servers:  
      requestBody:  
      callbacks:  
  
  /weather:  
    get:  
      tags:  
      summary:  
      description:  
      operationId:  
      externalDocs:  
      parameters:  
      responses:  
      deprecated:  
      security:  
      servers:  
      requestBody:  
      callbacks:  
  
  /weatherdata:  
    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 Mashape Weather API paths contain request body parameters.
- There's no need to include the `servers` object because the paths just use the same global

- The `servers` URL that we defined globally (page 279) 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 314)).
- 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:
  /aqi:
    get:
      tags:
        summary:
        description:
        operationId:
        externalDocs:
        parameters:
        responses:

  /weather:
    get:
      tags:
        summary:
        description:
        operationId:
        externalDocs:
        parameters:
        responses:

  /weatherdata:
    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.

Rather than defining the schema here, it's common to place a `$ref` (reference) pointer to a fuller definition in the `components` object. This approach also allows you to single source the schema, because now you can have many references pointing to the same defined schema in `components`. You can use the same technique for `responses` , `parameters` , and other properties. I explain more about using `$ref` below in [Re-using definitions across objects \(page 288\)](#) and in [step 5 \(page 292\)](#).

Here's the `operation` object defined for the Mashape Weather API:

```
paths:  
  /aqi:  
    get:  
      tags:  
        - Air Quality  
      summary: getAqi  
      description: Gets the air quality index  
      operationId: GetAqi  
      externalDocs:  
        description: More details  
        url: "https://market.mashape.com/fyhao/weather-13#aqi"  
      parameters:  
  
        - name: lat  
          in: query  
          description: "Latitude coordinates."  
          required: true  
          style: form  
          explode: false  
          schema:  
            type: string  
            example: "37.3708698"  
  
        - name: lng  
          in: query  
          description: "Longitude coordinates."  
          required: true  
          style: form  
          explode: false  
          schema:  
            type: string  
            example: "-122.037593"  
  
      responses:  
        200:  
          description: AQI response  
          content:  
            text/plain:  
              schema:  
                type: string  
                description: AQI response  
                example: 52  
  /weather:  
    get:  
      servers:  
        - url: https://simple-weather.p.mashape.com  
      tags:  
        - Weather Forecast  
      summary: getWeather  
      description: Gets the weather forecast in abbreviated form  
      operationId: GetWeather
```

```
externalDocs:  
  description: More details  
  url: "https://market.mashape.com/fyhao/weather-13#weather"  
parameters:  
  
- name: lat  
  in: query  
  description: "Latitude coordinates."  
  required: true  
  style: form  
  explode: false  
  schema:  
    type: string  
  example: "37.3708698"  
  
- name: lng  
  in: query  
  description: "Longitude coordinates."  
  required: true  
  style: form  
  explode: false  
  schema:  
    type: string  
  example: "-122.037593"  
  
responses:  
  200:  
    description: weather response  
    content:  
      text/plain:  
        schema:  
          type: string  
          description: weather response  
          example: 26 c, Mostly Clear at Singapore, Singapore  
  
/weatherdata:  
  get:  
    tags:  
    - Weather Forecast  
    summary: getWeatherData  
    description: Get weather forecast with lots of details  
    operationId: GetWeatherData  
    externalDocs:  
      description: More details  
      url: "https://market.mashape.com/fyhao/weather-13#weatherdata"  
    parameters:  
  
- name: lat  
  in: query  
  description: "Latitude coordinates."  
  required: true
```

```
style: form
explode: false
schema:
  type: string
example: "37.3708698"

- name: lng
  in: query
  description: "Longitude coordinates."
  required: true
  style: form
  explode: false
  schema:
    type: string
example: "-122.037593"

responses:
  200:
    description: Successful operation
    content:
      application/json:
        schema:
          description: Successful operation
          $ref: '#/components/schemas/WeatherdataResponse'
```

Re-using definitions across objects

In this API, the `lat` and `lng` parameters are duplicated in each path. Copying and pasting this 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.

I'll dive into more details in the [components step \(page 292\)](#), but for now, note that we can use a `$ref` property to refer to more details in the `components` object. See how `parameters` simply contains a `reference object`:

```
paths:  
  /aqi:  
    get:  
      tags:  
        - Air Quality  
      summary: getAqi  
      description: Gets the air quality index  
      operationId: GetAqi  
      externalDocs:  
        description: More details  
        url: "https://market.mashape.com/fyhao/weather-13#aqi"  
      parameters:  
        - $ref: '#/components/parameters/latParam'  
        - $ref: '#/components/parameters/lngParam'  
      responses:  
        200:  
          description: AQI response  
          content:  
            text/plain:  
              schema:  
                type: string  
                description: AQI response  
                example: 52  
  /weather:  
    get:  
      servers:  
        - url: https://simple-weather.p.mashape.com  
      tags:  
        - Weather Forecast  
      summary: getWeather  
      description: Gets the weather forecast in abbreviated form  
      operationId: GetWeather  
      externalDocs:  
        description: More details  
        url: "https://market.mashape.com/fyhao/weather-13#weather"  
      parameters:  
        - $ref: '#/components/parameters/latParam'  
        - $ref: '#/components/parameters/lngParam'  
      responses:  
        200:  
          description: weather response  
          content:  
            text/plain:  
              schema:  
                type: string  
                description: weather response  
                example: 26 c, Mostly Clear at Singapore, Singapore  
  /weatherdata:  
    get:  
      tags:
```

```

  - Weather Forecast
  summary: getWeatherData
  description: Get weather forecast with lots of details
  operationId: GetWeatherData
  externalDocs:
    description: More details
    url: "https://market.mashape.com/fyhao/weather-13#weatherdata"
  parameters:
    - $ref: '#/components/parameters/latParam'
    - $ref: '#/components/parameters/lngParam'
  responses:
    200:
      description: Successful operation
      content:
        application/json:
          schema:
            description: Successful operation
            $ref: '#/components/schemas/WeatherdataResponse'

```

Now we're not repeating the parameter information multiple times. Instead, in `components`, we define these parameters:

```

components:
  parameters:
    - name: lat
      in: query
      description: "Latitude coordinates."
      required: true
      style: form
      explode: false
      schema:
        type: string
      example: "37.3708698"

    - name: lng
      in: query
      description: "Longitude coordinates."
      required: true
      style: form
      explode: false
      schema:
        type: string
      example: "-122.037593"

```

See [Storing re-used parameters in components \(page 293\)](#) for more details. Also see [Describing Parameters](#) in Swagger's OpenAPI documentation.

Appearance of paths in Swagger UI

Swagger UI displays the `paths` object like this:

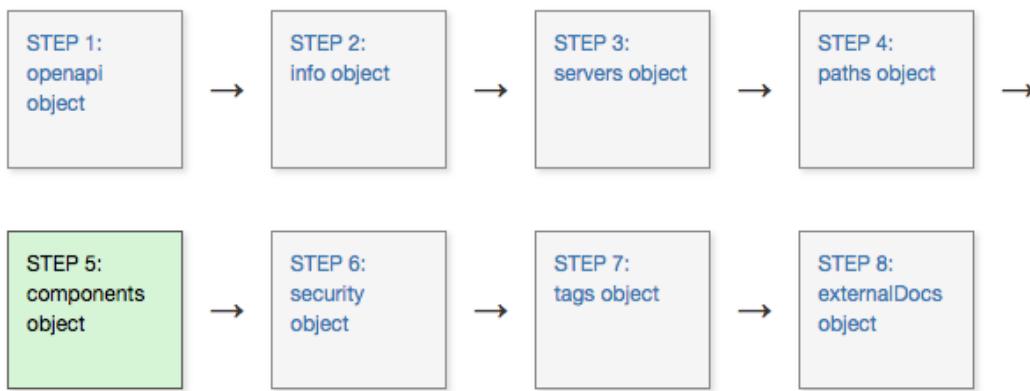
The screenshot shows the Swagger UI interface for a 'getWeather' endpoint. At the top, there is a 'GET' button, the path '/weather', and a 'getWeather' operation name. A lock icon is also present. Below the path, a description states 'Gets the weather forecast in abbreviated form'. A 'Find more details' link is available, along with a link to 'https://market.mashape.com/fyhao/weather-13#weather'. The 'Parameters' section is expanded, showing two required parameters: 'lat' and 'lon'. The 'lat' parameter is described as 'Latitude coordinates.' with a value of '37.3708698'. The 'lon' parameter is described as 'Longitude coordinates.' with a value of '-122.037593'. A 'Cancel' button is located in the top right corner of the parameters panel.

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. Currently, there isn't a parameter to hide the Models section or to disable the Try It Out section, but you can hide these functions through `display: none` with CSS, targeting the elements you want to hide. Additional Swagger UI parameters may be added in the future.

OpenAPI tutorial step 5: The components object



In the [step 4 \(page 281\)](#), when we described the `requestBody` and `responses` object objects, 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. While our Mashape weather API is simple, the response from the `weatherdata` endpoint is complex. 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 through `$ref`, that refers to a specific definition in the `components` object.

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 parameters in components

In the Mashape Weather API, the `lat` and `lng` parameters are the same for each of the paths. Rather than duplicate this same description in each path, we can store the object in the `components`. To do this, in the `path` object, we simply include a `$ref` pointer that refers to the location in `components` that contains these details:

```
paths:  
  /aqi:  
    get:  
      ...  
      parameters:  
        - $ref: '#/components/parameters/latParam'  
        - $ref: '#/components/parameters/lngParam'  
      ...
```

(Ellipses (`...`) indicate truncated code.)

The `latParam` and `lngParam` names are arbitrary. These names just act like variables that are defined in the `components` section.

Notice that the `parameters` are stored under `components/parameters`. Now let's define the parameters one time in `components`:

```
components:  
  parameters:  
    latParam:  
      name: lat  
      in: query  
      description: "Latitude coordinates."  
      required: true  
      style: form  
      explode: false  
      schema:  
        type: string  
        example: "37.3708698"  
    lngParam:  
      name: lng  
      in: query  
      description: "Longitude coordinates."  
      required: true  
      style: form  
      explode: false  
      schema:  
        type: string  
        example: "-122.037593"
```

See [Using \\$ref](#) for more details on this standard JSON reference property.

Re-using response objects

Response objects are another common object to re-use across paths. In this API, suppose we had a [404](#) response for each path indicating a resource is not found. Here's how we can reference that in the response object:

```
paths:  
  /aqi:  
    get:  
      ...  
    responses:  
      200:  
        description: AQI response  
        content:  
          text/plain:  
            schema:  
              type: string  
              description: AQI response  
              example: 52  
      404:  
        $ref: '#/components/responses/404'
```

Then in `components/responses`, we define it:

```
components:  
  responses:  
    404:  
      description: Not found  
      content:  
        text/plain:  
          schema:  
            type: string  
            description: Not found  
            example: Not found
```

The weatherdata response

Although we don't need to reuse the `weatherdata` response, it's so lengthy and complex, it'll be better to organize it under `components`. If you recall in the previous step ([OpenAPI tutorial step 4: The paths object \(page 281\)](#)), the `responses` object for the `weatherdata` endpoint looked like this:

```
responses:  
  200:  
    description: Successful operation  
    content:  
      application/json:  
        schema:  
          description: Successful operation  
          $ref: '#/components/schemas/WeatherdataResponse'
```

The `$ref` points to a definition stored in the `components` object. Before we describe the response in the `components` object, it might be helpful to review what the `weatherdata` response looks like. The response contains multiple nested objects at various levels. (Note that this Mashape Weather API builds off of a [Yahoo weather service API](#), so the data returned in the `weather` and `weatherdata` endpoints is highly similar to the data returned by the Yahoo weather service API.)

There are a couple of ways to go about describing this response. You could create one long description like this:

```
components:
  schemas:
    WeatherdataResponse:
      title: weatherdata response
      type: object
      properties:
        query:
          type: object
          properties:
            count:
              type: integer
              description: The number of items (rows) returned -- specifically, the number of sub-elements in the results property
              format: int32
              example: 1
            created:
              type: string
              description: The date and time the response was created
              example: 6/14/2017 2:30:14 PM
            lang:
              type: string
              description: The locale for the response
              example: en-US
            results:
              type: object
              properties:
                channel:
                  type: object
                  properties:
                    units:
                      description: Units for various aspects of the forecast. Note that the default is to use metric formats for the units (Celsius, kilometers, millibars, kilometers per hour).
                    Units:
                      description: Units for various aspects of the forecast. Note that the default is to use metric formats for the units (Celsius, kilometers, millibars, kilometers per hour).
                      type: object
                      properties:
                        distance:
                          type: string
                          description: Units for distance, mi for miles or km for kilometers
                          example: km
                        pressure:
                          type: string
                          description: Units of barometric pressure, "in" for pounds per square inch or "mb" for millibars.
                          example: mb
                        speed:
                          type: string
```

```
        description: Units of speed, "mph" for miles per hour or "kph" for kilometers per hour.
        example: km/h
      temperature:
        type: string
        description: Degree units, "f" for Fahrenheit or "c" for Celsius.
        example: C

      title:
        type: string
        description: The title of the feed, which includes the location city
        example: Yahoo! Weather - Singapore, South West, SG

      link:
        type: string
        description: The URL for the Weather page of the forecast for this location
        example: http://us.rd.yahoo.com/dailynews/rss/weather/Country__Country/*https://weather.yahoo.com/country/state/city-91792352/

      description:
        type: string
        description: The overall description of the feed including the location
        example: Yahoo! Weather for Singapore, South West, SG

      language:
        type: string
        description: The language of the weather forecast
        example: en-us

      lastBuildDate:
        type: string
        description: The last time the feed was updated. The format is in the date format defined by [RFC822 Section 5](http://www.rfc-editor.org/rfc/rfc822.txt).
        example: Wed, 14 Jun 2017 10:30 PM SGT

      ttl:
        type: string
        description: Time to Live -- how long in minutes this feed should be cached.
        example: 60

      location:
        description: The location of this forecast
        type: object
        properties:
          city:
```

```
        type: string
        description: City name
        example: Singapore
    country:
        type: string
        description: Two-character country code
        example: Singapore
    region:
        type: string
        description: State, territory, or region, if given
        example: South West

    wind:
        description: Forecast information about wind
        type: object
        properties:
            chill:
                type: integer
                description: Wind chill in degrees
                format: int32
                example: 81
            direction:
                type: integer
                description: Wind direction, in degrees
                format: int32
                example: 158
            speed:
                type: integer
                description: Wind speed, in the units specified i
n the speed attribute of the units element (mph or kph)
                format: int32

    atmosphere:
        description: Forecast information about current atmosp
heric pressure, humidity, and visibility
        type: object
        properties:
            humidity:
                type: integer
                description: humidity, in percent
                format: int32
                example: 87
            pressure:
                type: integer
                description: Barometric pressure, in the units spe
cified by the pressure attribute of the weather:units element ("in" or "m
b").
                format: int32
            rising:
                type: integer
                description: 'State of the barometric pressure: st
```

```
eady (0), rising (1), or falling (2).'
    format: int32
    example: 0
  visibility:
    type: integer
    description: Visibility, in the units specified by the distance attribute of the units element ("mi" or "km"). Note that the visibility is specified as the actual value * 100. For example, a visibility of 16.5 miles will be specified as 1650. A visibility of 14 kilometers will appear as 1400.
    format: int32

  astronomy:
    description: Forecast information about current astronomical conditions
    type: object
    properties:
      sunrise:
        type: string
        description: Today's sunrise time. The time is a string in a local time format of "h:mm am/pm"
        example: 6:59 am
      sunset:
        type: string
        description: Today's sunset time. The time is a string in a local time format of "h:mm am/pm"
        example: 7:11 pm

  image:
    description: The image used to identify this feed
    type: object
    properties:
      title:
        type: string
        description: The title of the image.
        example: Yahoo! Weather
      width:
        type: string
        description: The width of the image, in pixels
        example: 142
      height:
        type: string
        description: The height of the image, in pixels
        example: 18
      link:
        type: string
        description: Description of link
        example: http://weather.yahoo.com
      url:
        type: string
        description: The URL of Yahoo! Weather
```

```
example: http://l.yimg.com/a/i/brand/purplelogo//uh/us/news-wea.gif

item:
  description: The item element contains current conditions and forecast for the given location. There are multiple weather forecast elements for today and tomorrow.
  type: object
  properties:
    title:
      type: string
      description: The forecast title and time.
      example: Conditions for Singapore, South West, SG
at 09:00 PM SGT
    lat:
      type: string
      description: The latitude of the location.
      example: 1.33464
    long:
      type: string
      description: The longitude of the location.
      example: 103.726471
    link:
      type: string
      description: The Yahoo! Weather URL for this forecast.
      example: http://us.rd.yahoo.com/dailynews/rss/weather/Country__Country/*https://weather.yahoo.com/country/state/city-91792352/
    pubDate:
      type: string
      description: The date and time this forecast was posted, in the date format defined by [RFC822 Section 5](http://www.rfc-editor.org/rfc/rfc822.txt).
      example: Wed, 14 Jun 2017 09:00 PM SGT
    condition:
      description: The current weather conditions
      type: object
      properties:
        code:
          type: integer
          description: The condition code for this forecast. You could use this code to choose a text description or image for the forecast. The possible values for this element are described in the [Condition Codes](https://developer.yahoo.com/weather/).documentation.html.
          format: int32
          example: 33
        date:
          type: string
          description: The current date and time for which this forecast applies. The date is in RFC822 Section 5 format.
          example: Wed, 14 Jun 2017 09:00 PM SGT
```

```
        temp:  
          type: integer  
          description: The current temperature, in the units specified by the units element.  
          format: int32  
          example: 26  
        text:  
          type: string  
          description: A textual description of conditions  
          example: Mostly Clear  
  
        forecast:  
          description: The item element contains current conditions and forecast for the given location. There are multiple weather forecast elements for today and tomorrow.  
          type: array  
          items:  
            type: object  
            properties:  
              code:  
                type: integer  
                description: The condition code for this forecast. You could use this code to choose a text description or image for the forecast. The possible values for this element are described in the [Condition Codes](https://developer.yahoo.com/weather/documentation.html#codes).  
                format: int32  
                example: 4  
              date:  
                type: string  
                description: The date to which this forecast applies. The date is in 'dd Mmm yyyy' format, for example '30 Nov 2005'  
                example: 14 Jun 2017  
              day:  
                type: string  
                description: Day of the week to which this forecast applies. Possible values are Mon Tue Wed Thu Fri Sat Sun.  
                example: Wed  
              high:  
                type: integer  
                description: The forecasted high temperature for this day, in the units specified by the units element.  
                format: int32  
                example: 28  
              low:  
                type: integer  
                description: The forecasted low temperature for this day, in the units specified by the units element.  
                format: int32  
                example: 25
```

```

    text:
      type: string
      description: A textual description of conditions
      example: Thunderstorms
      description: The weather forecast for a specific day. The item element contains multiple forecast elements for today and tomorrow.

    Guid:
      title: guid
      type: object
      properties:
        isPermaLink:
          type: string
          description: The attribute isPermaLink is false.
          example: false
          description: Unique identifier for the forecast, made up of the location ID, the date, and the time.
        description:
          type: string
          description: A simple summary of the current conditions and tomorrow's forecast, in HTML format, including a link to Yahoo! Weather for the full forecast.
          example: "<![CDATA[<img src=\"http://l.yimg.com/a/us/we/52/4.gif\"/>
          <br /><b>Current Conditions:</b><br />
          Thunderstorms<br /><br /><b>Forecast:</b><br />
          Fri - Thunderstorms. High: 30 Low: 25<br />
          Sat - Thunderstorms. High: 28 Low: 25<br />
          Sun - Thunderstorms. High: 28 Low: 25<br />
          Mon - Thunderstorms. High: 28 Low: 25<br />
          Tue - Thunderstorms. High: 28 Low: 25<br />
          <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>
          <br /><br />
          (provided by <a href=\"http://www.weather.com\">The Weather Channel</a>)<br />
          <br />]]>"
```

guid:

type: string

description: Unique identifier for the forecast, made up of the location ID, the date, and the time.

format: uuid

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.

```
components:
  schemas:
    WeatherdataResponse:
      title: weatherdata response
      type: object
      properties:
        query:
          $ref: '#/components/schemas/Query'
    Query:
      title: query
      type: object
      properties:
        count:
          type: integer
          description: The number of items (rows) returned -- specifically, the number of sub-elements in the results property
          format: int32
          example: 1
        created:
          type: string
          description: The date and time the response was created
          example: 6/14/2017 2:30:14 PM
        lang:
          type: string
          description: The locale for the response
          example: en-US
        results:
          $ref: '#/components/schemas/Results'
    Results:
      title: results
      type: object
      properties:
        channel:
          $ref: '#/components/schemas/Channel'
    Channel:
      title: channel
      type: object
      properties:
        units:
          description: Units for various aspects of the forecast. Note that the default is to use metric formats for the units (Celsius, kilometers, milibars, kilometers per hour).
          $ref: '#/components/schemas/Units'
        title:
          type: string
          description: The title of the feed, which includes the location city
          example: Yahoo! Weather - Singapore, South West, SG
        link:
          type: string
          description: The URL for the Weather page of the forecast for thi
```

```
s location
    example: http://us.rd.yahoo.com/dailynews/rss/weather/Country_Cou
ntry/*https://weather.yahoo.com/country/state/city-91792352/
    description:
        type: string
        description: The overall description of the feed including the loc
ation
        example: Yahoo! Weather for Singapore, South West, SG
    language:
        type: string
        description: The language of the weather forecast
        example: en-us
    lastBuildDate:
        type: string
        description: The last time the feed was updated. The format is in
the date format defined by [RFC822 Section 5](http://www.rfc-editor.org/rfc/
rfc822.txt).
        example: Wed, 14 Jun 2017 10:30 PM SGT
    ttl:
        type: string
        description: Time to Live -- how long in minutes this feed should
be cached.
        example: 60
    location:
        description: The location of this forecast
        $ref: '#/components/schemas/Location'
    wind:
        description: Forecast information about wind
        $ref: '#/components/schemas/Wind'
    atmosphere:
        description: Forecast information about current atmospheric pressu
re, humidity, and visibility
        $ref: '#/components/schemas/Atmosphere'
    astronomy:
        description: Forecast information about current astronomical condi
tions
        $ref: '#/components/schemas/Astronomy'
    image:
        description: The image used to identify this feed
        $ref: '#/components/schemas/Image'
    item:
        description: The item element contains current conditions and fore
cast for the given location. There are multiple weather forecast elements fo
r today and tomorrow.
        $ref: '#/components/schemas/Item'
    Units:
        title: units
        type: object
        properties:
            distance:
                type: string
```

```
        description: Units for distance, mi for miles or km for kilometers
        example: km
    pressure:
        type: string
        description: Units of barometric pressure, "in" for pounds per square inch or "mb" for millibars.
        example: mb
    speed:
        type: string
        description: Units of speed, "mph" for miles per hour or "kph" for kilometers per hour.
        example: km/h
    temperature:
        type: string
        description: Degree units, "f" for Fahrenheit or "c" for Celsius.
        example: C
    description: Units for various aspects of the forecast. Note that the default is to use metric formats for the units (Celsius, kilometers, millibars, kilometers per hour).
    Location:
        title: location
        type: object
        properties:
            city:
                type: string
                description: City name
                example: Singapore
            country:
                type: string
                description: Two-character country code
                example: Singapore
            region:
                type: string
                description: State, territory, or region, if given
                example: South West
            description: The location of this forecast
    Wind:
        title: wind
        type: object
        properties:
            chill:
                type: integer
                description: Wind chill in degrees
                format: int32
                example: 81
            direction:
                type: integer
                description: Wind direction, in degrees
                format: int32
                example: 158
            speed:
```

```
        type: integer
        description: Wind speed, in the units specified in the speed attribute of the units element (mph or kph)
        format: int32
    description: Forecast information about wind
    Atmosphere:
        title: atmosphere
        type: object
        properties:
            humidity:
                type: integer
                description: humidity, in percent
                format: int32
                example: 87
            pressure:
                type: integer
                description: Barometric pressure, in the units specified by the pressure attribute of the weather:units element ("in" or "mb").
                format: int32
            rising:
                type: integer
                description: 'State of the barometric pressure: steady (0), rising (1), or falling (2).'
                format: int32
                example: 0
            visibility:
                type: integer
                description: Visibility, in the units specified by the distance attribute of the units element ("mi" or "km"). Note that the visibility is specified as the actual value * 100. For example, a visibility of 16.5 miles will be specified as 1650. A visibility of 14 kilometers will appear as 1400.
                format: int32
    description: Forecast information about current atmospheric pressure, humidity, and visibility
    Astronomy:
        title: astronomy
        type: object
        properties:
            sunrise:
                type: string
                description: Today's sunrise time. The time is a string in a local time format of "h:mm am/pm"
                example: 6:59 am
            sunset:
                type: string
                description: Today's sunset time. The time is a string in a local time format of "h:mm am/pm"
                example: 7:11 pm
    description: Forecast information about current astronomical conditions
Image:
```

```
title: image
type: object
properties:
  title:
    type: string
    description: The title of the image.
    example: Yahoo! Weather
  width:
    type: string
    description: The width of the image, in pixels
    example: 142
  height:
    type: string
    description: The height of the image, in pixels
    example: 18
  link:
    type: string
    description: Description of link
    example: http://weather.yahoo.com
  url:
    type: string
    description: The URL of Yahoo! Weather
    example: http://l.yimg.com/a/i/brand/purplelogo//uh/us/news-wea.gi
f
  description: The image used to identify this feed
Item:
  title: item
  type: object
  properties:
    title:
      type: string
      description: The forecast title and time.
      example: Conditions for Singapore, South West, SG at 09:00 PM SGT
    lat:
      type: string
      description: The latitude of the location.
      example: 1.33464
    long:
      type: string
      description: The longitude of the location.
      example: 103.726471
    link:
      type: string
      description: The Yahoo! Weather URL for this forecast.
      example: http://us.rd.yahoo.com/dailynews/rss/weather/Country_Cou
ntry/*https://weather.yahoo.com/country/state/city-91792352/
  pubDate:
    type: string
    description: The date and time this forecast was posted, in the da
te format defined by [RFC822 Section 5](http://www.rfc-editor.org/rfc/rfc82
2.txt).
```

```
example: Wed, 14 Jun 2017 09:00 PM SGT
condition:
  description: The current weather conditions
  $ref: '#/components/schemas/Condition'
forecast:
  type: array
  items:
    $ref: '#/components/schemas/ForecastArray'
    description: ''
description:
  type: string
  description: A simple summary of the current conditions and tomorrow's forecast, in HTML format, including a link to Yahoo! Weather for the full forecast.
  example: "<![CDATA[<img src=\"http://l.yimg.com/a/i/us/we/52/4.gif\"\n/>\n<BR /\>\n<b>Current Conditions:</b>\n<BR /\>Thunderstorm\ns\n<BR /\>\n<BR /\>\n<b>Forecast:</b>\n<BR /\> Fri – Thunderstorms. High: 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/*http://weather.yahoo.com/country/state/city-91792352/\">Full Forecast at\nYahoo! Weather</a>\n<BR /\>\n<BR /\>\n(n(provided by <a href=\"http://www.weather.com\">The Weather Channel</a>)\n<BR /\>\n)]>"
guid:
  type: string
  description: Unique identifier for the forecast, made up of the location ID, the date, and the time.
  format: uuid
  description: The item element contains current conditions and forecast for the given location. There are multiple weather forecast elements for today and tomorrow.
Condition:
  title: condition
  type: object
  properties:
    code:
      type: integer
      description: The condition code for this forecast. You could use this code to choose a text description or image for the forecast. The possible values for this element are described in the [Condition Codes](https://developer.yahoo.com/weather/).documentation.html.
      format: int32
      example: 33
    date:
      type: string
      description: The current date and time for which this forecast applies. The date is in RFC822 Section 5 format.
      example: Wed, 14 Jun 2017 09:00 PM SGT
    temp:
      type: integer
```

```
        description: The current temperature, in the units specified by th
e units element.
        format: int32
        example: 26
      text:
        type: string
        description: A textual description of conditions
        example: Mostly Clear
      description: The current weather conditions
    ForecastArray:
      title: forecast array
      type: object
      properties:
        code:
          type: integer
          description: The condition code for this forecast. You could use t
his code to choose a text description or image for the forecast. The possibl
e values for this element are described in the [Condition Codes](https://dev
eloper.yahoo.com/weather/documentation.html#codes).
          format: int32
          example: 4
        date:
          type: string
          description: The date to which this forecast applies. The date is
in 'dd Mmm yyyy' format, for example '30 Nov 2005'
          example: 14 Jun 2017
        day:
          type: string
          description: Day of the week to which this forecast applies. Possi
ble values are Mon Tue Wed Thu Fri Sat Sun.
          example: Wed
        high:
          type: integer
          description: The forecasted high temperature for this day, in the
units specified by the units element.
          format: int32
          example: 28
        low:
          type: integer
          description: The forecasted low temperature for this day, in the u
nits specified by the units element.
          format: int32
          example: 25
      text:
        type: string
        description: A textual description of conditions
        example: Thunderstorms
      description: The weather forecast for a specific day. The item elemen
t contains multiple forecast elements for today and tomorrow.
    Guid:
      title: guid
```

```
type: object
properties:
  isPermaLink:
    type: string
    description: The attribute isPermaLink is false.
    example: false
    description: Unique identifier for the forecast, made up of the location ID, the date, and the time.
```

Not only can you use `$ref` properties in other parts of your spec, you can use it within `components` too.

Appearance of components in Swagger UI

Swagger UI displays each object in `components` in a section called `Models` at the end of your Swagger UI display. If you decided to consolidate all schemas into a single object, without using the `$ref` property to point to new objects, then you will see just one object in Models. If you split out the objects, then you see each object listed separately, including the object that contains all the references.

Because I want to re-use objects, I'm going define each object in `components` separately. As a result, the Models section looks like this:

The screenshot shows the 'Models' section of the Swagger UI. It lists several objects, each represented by a button-like element with a label and a placeholder '...' to its right. The objects are: 'weatherdata response > {...}', 'query > {...}', 'results > {...}', 'channel > {...}', 'units > {...}', 'location > {...}', and 'wind > {...}'. A small downward arrow icon is located in the top right corner of the section.

Reason for models in the first place

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>Successful operation</p> <p>application/json</p> <p>Controls Accept header.</p> <p>Example Value Model ←</p> <pre> weatherdata response ↴ { query query ↴ { count integer (\$int32) example: 1 The number of items (rows) returned -- specifically, the number of sub-elements in the results property created string example: 6/14/2017 2:30:14 PM The date and time the response was created lang string example: en-US The locale for the response results results ↴ { channel channel ↴ { units units ↴ { description: Units for various aspects of the weather } } } } </pre>	No links

Hiding the Models section

You might confuse users by including the Models section. Currently, there isn't a [Swagger UI](#) parameter to hide the Models section. To hide Models, simply add this style to the Swagger UI page:

```

<style>
section.models {
    display: none;
}
</style>

```

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.

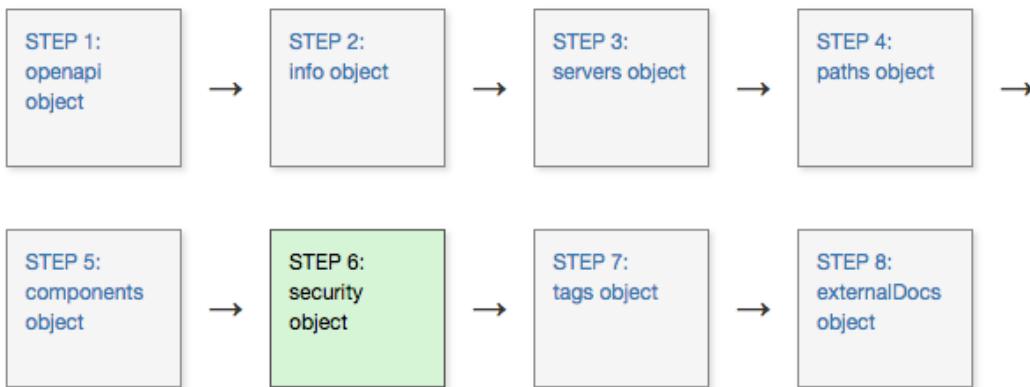
Additionally, look at some example schemas. You can view [3.0 examples here](#). I usually find a spec that resembles what I'm trying to represent and mimic the same properties and structure.

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 \(page 314\)](#).

OpenAPI tutorial step 6: The security object



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 \(page 166\)](#). 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 \(page 168\)](#) 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 Mashape weather API we're using in this course uses an API key passed in the request header (`X-Mashape-Key: 123456789`). If you submit a request without this header (and 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:
  - Mashape-Key: []
```

`Mashape-Key` is the arbitrary name we gave to this security scheme in our `securitySchemes` object. We could have named it anything. We'll define `Mashape-Key` in `components`.

All paths will use the `Mashape-Key` security method by default unless it's overridden by a value at the [path object level \(page 281\)](#). For example, at the path level we could overwrite the global security method as follows:

```
/weatherdata:
  get:
    ...
    security:
      - Some-Other-Key: []
```

Then the `weatherdata` path would use the `Some-Other-Key` security method, while all other paths would use the globally declared security, `Mashape-Key`.

Referencing the security scheme in components

In the [components object \(page 292\)](#), we add a `securitySchemes` object that defines details about the security scheme we're using:

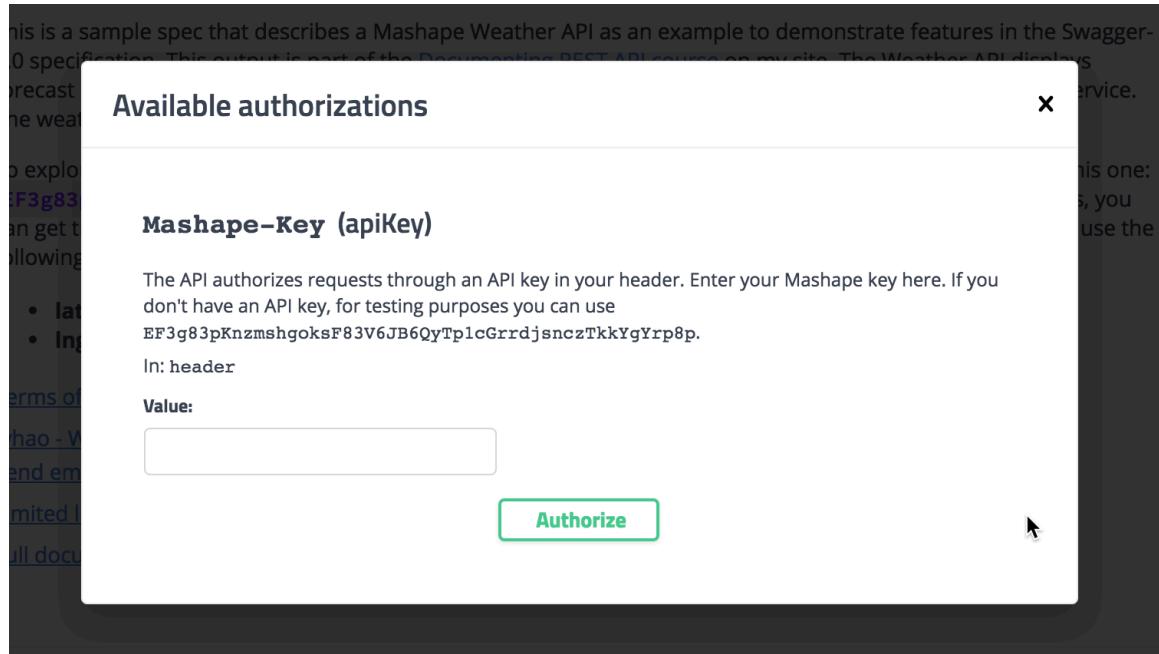
```
components:
  ...
  securitySchemes:
    Mashape-Key:
      type: apiKey
      description: "The API authorizes requests through an API key in your header. Enter your Mashape key here. If you don't have an API key, for testing purposes you can use `EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p`."
      name: X-Mashape-Key
      in: header
```

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 click **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 "https://simple-weather.p.mashape.com/weather?lat=37.3708698&ln=g=-122.037593" -H "accept: text/plain" -H "X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p"
```

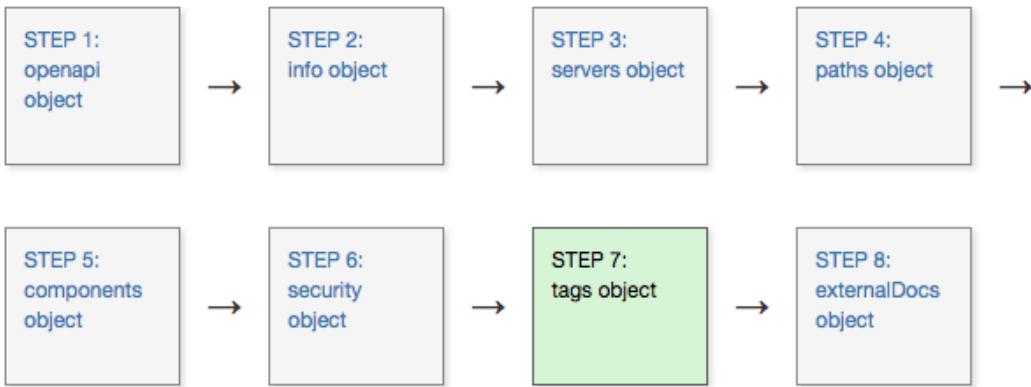
The `-H "X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p"` indicates that a header is being sent with the API key. (For more on cURL, see [Make a cURL call \(page 46\)](#).)

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.

OpenAPI tutorial step 7: The tags object



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 \(page 281\)](#)).

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

```
tags:
  - name: Air Quality
    description: The pollution quality of the air.
  - name: Weather Forecast
    description: A full list of details about the current weather.
```

In this simple weather API, there are two tags. You can list both the `name` and a `description` for each tag.

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.

For example, in the operations object for the `/weather` path, we used the tag `Weather Forecast`:

```

paths:
  ...
  /weather:
    get:
      servers:
        - url: https://simple-weather.p.mashape.com
      tags:
        - Weather Forecast
      ...
  ...

```

We used the same tag with the `/weatherdata` path:

```

paths:
  ...
  /weatherdata:
    get:
      tags:
        - Weather Forecast
      summary: getWeatherData
      description: Get weather forecast with lots of details
      operationId: GetWeatherData
  ...

```

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 Forecast` tag will be grouped together under the title `Weather Forecast`. Each group title is a collapsible/expandable toggle. The `/aqi` path has the `Air Quality` tag.

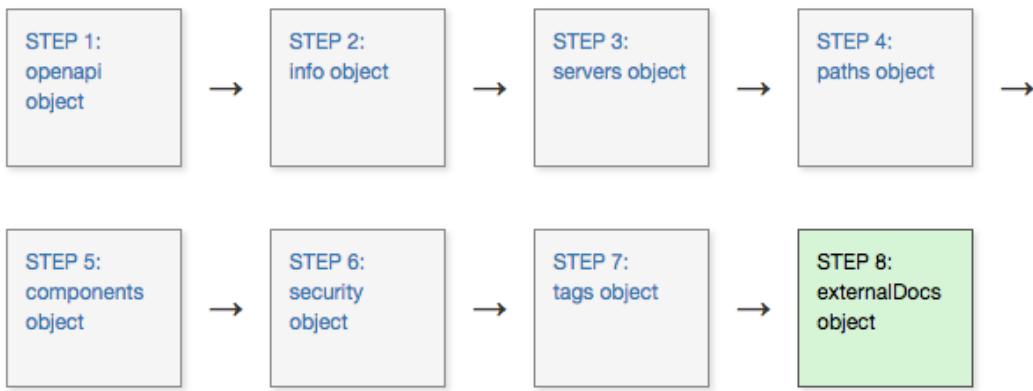
The screenshot shows the Swagger UI interface with the following structure:

- Air Quality**: A group title for the `/aqi` endpoint. It includes a description: "The pollution quality of the air." and a collapse/expand arrow.
- Weather Forecast**: A group title for the `/weather` and `/weatherdata` endpoints. It includes descriptions: "A full list of details about the current weather." for `/weather` and "Get weather forecast with lots of details" for `/weatherdata`, along with a collapse/expand arrow.
- Models**: A section labeled "Models" with a collapse/expand arrow.

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 this simple weather API, tags don't seem all that necessary. 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.

OpenAPI tutorial step 8: The externalDocs object



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: Full documentation for the API  
  url: https://market.mashape.com/fyhao/weather-13
```

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 Weather API. At the top, there's a green header bar with the 'swagger' logo, a URL field containing '/learnapidoc/docs/rest_api_specifications/openapi_weather.yml', and a 'Explore' button. Below the header, the title 'Weather API from Mashape' is displayed with a '1.0 OAS3' badge. A link to the YAML file is provided below the title. A descriptive text block explains that this is a sample spec for the Mashape Weather API, mentioning it's part of a REST API course and uses Yahoo Weather Service data. It also provides instructions for exploring the API using an API key or by using a specific key (EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjjsnczTkkYgYrp8p). It notes that latitude and longitude can be obtained from Google Maps URLs. Below this, there are several links: 'Terms of service', 'fyhao - Website', 'Send email to fyhao', 'Limited license', and 'Full documentation for the API'. An arrow points from the text 'For example, for Santa Clara, California, use the following:' towards the 'Full documentation for the API' link.

See the related topic, [Integrating Swagger UI with the rest of your docs \(page 334\)](#) 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 Weather API from Mashape. At the top, there's a green header bar with the 'swagger' logo, the URL '/learnapidoc/docs/rest_api_specifications/openapi_weather.yml', and an 'Explore' button. Below the header, the title 'Weather API from Mashape' is displayed, along with a '1.0 OAS3' badge and a link to the YAML file. A descriptive text block explains that this is a sample spec for the Mashape Weather API, noting it uses Yahoo Weather Service data. It also provides instructions for exploring the API using an API key or specific coordinates (lat: 37.3708698, lon: -122.037593). Below this, there are links for 'Terms of service', 'the developer - Website', 'Send email to the developer', 'Limited license', and 'Full documentation for the API'. On the right side of the main content area, there's a 'Authorize' button with a lock icon. The main content area shows the API endpoints. Under 'Air Quality', there's a single endpoint: GET /aqi getAqi. Under 'Weather Forecast', there are two endpoints: GET /weather getWeather and GET /weatherdata getWeatherData. Each endpoint row has a lock icon on the right.

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.

OpenAPI specification activity: Create your own specification document

ACTIVITY

The [OpenAPI tutorial \(page 270\)](#) 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

First, find an API that's relatively simple. If you're already documenting an API for your work, by all means use that API. But if you're just taking this course to learn general API documentation, try creating an OpenAPI specification document for this simple [Sunrise and sunset times API](#). This 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 \(page 274\)](#)
- [Step 2: info object \(page 277\)](#)
- [Step 3: servers object \(page 279\)](#)
- [Step 4: paths object \(page 281\)](#)
- [Step 5: components object \(page 292\)](#)
- [Step 6: security object \(page 314\)](#)
- [Step 7: tags object \(page 317\)](#)
- [Step 8: externalDocs object \(page 319\)](#)

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](#).

Note that the Sunrise and sunset times API doesn't require authorization, so you can skip [Step 6: security object \(page 314\)](#).

You can use this OpenAPI specification document when working through the [Swagger UI activity \(page 331\)](#).

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 \(page 259\)](#).

For step-by-step tutorial on creating an OpenAPI specification document, see the [OpenAPI tutorial \(page 270\)](#).

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 \(page 383\)](#).

This tutorial focuses on Swagger UI. For a deep dive into the OpenAPI spec, see my [OpenAPI tutorial here \(page 270\)](#).

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 the title "Swagger Petstore 1.0.0" with a note about the base URL. A message at the top says it's a sample server and provides links for terms of service, developer contact, Apache 2.0 license, and more about Swagger. On the left, there's a dropdown for "Schemes" set to "HTTP" and an "Authorize" button with a lock icon. The main content area is titled "pet" with a description "Everything about your Pets". It lists four endpoints:

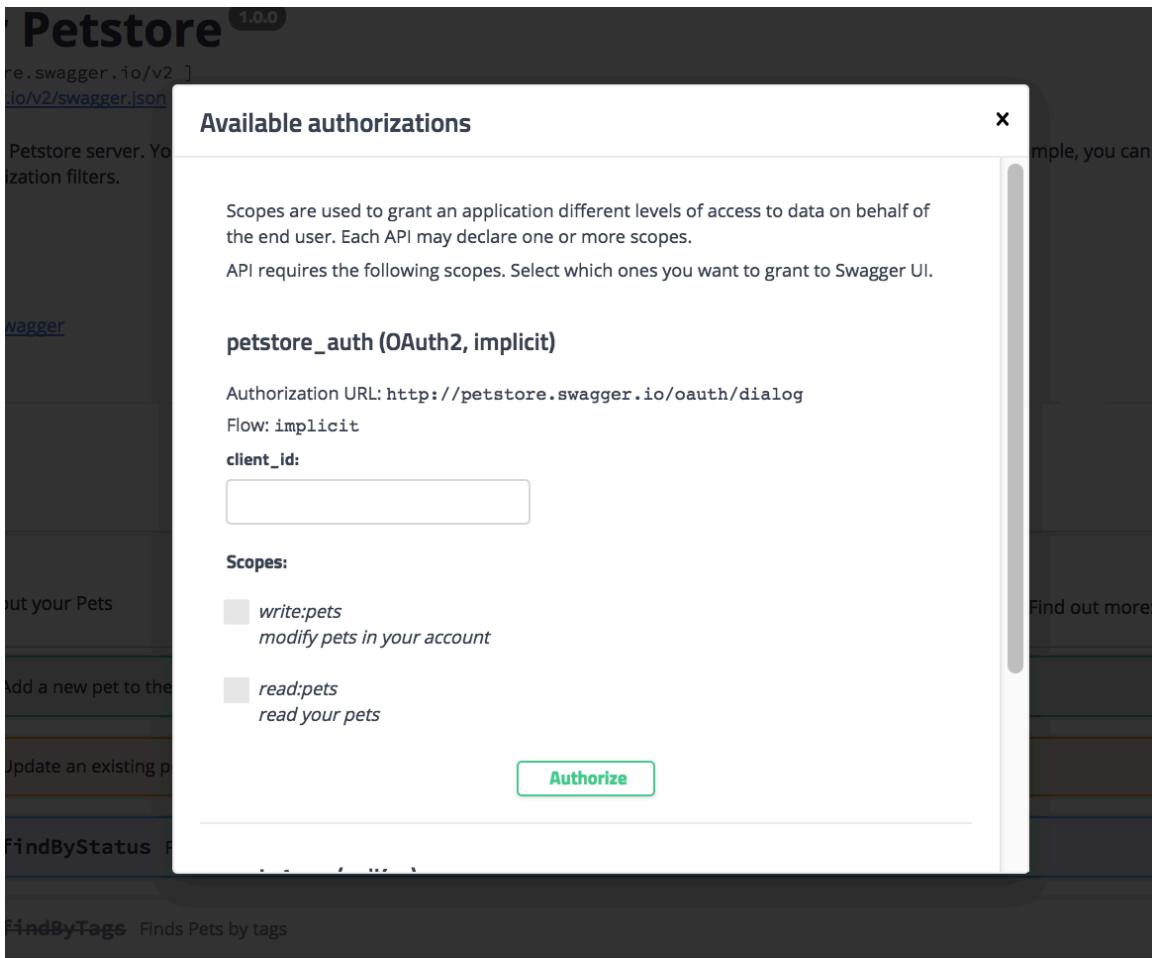
- POST /pet**: Add a new pet to the store.
- PUT /pet**: Update an existing pet.
- GET /pet/findByStatus**: Finds Pets by status.
- GET /pet/findByTags**: Finds Pets by tags.

The endpoints are grouped 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

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

Example Value Model

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

Parameter content type application/json

Responses

Response content type application/xml

After you click Try it out, the example value in the Request Body field becomes editable.

3. In the Example Value field, change the first `id` value to a random integer, such as `193844`. Change the second `name` value to something you'd recognize (your pet's name).
4. Click **Execute**.

POST /pet Add a new pet to the store

Parameters

Name Description

body * required 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 \(page 46\)](#). The Responses section shows the [response \(page 108\)](#). (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 Petstore API. The 'Responses' tab is active, showing a curl command to post a pet and its JSON response body.

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

Important: The Petstore is a functioning API, and you have actually created a pet. You now need to take responsibility for your pet and begin feeding and caring for it! All joking aside, most users don't realize they're playing with real data when they execute responses in an API (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:

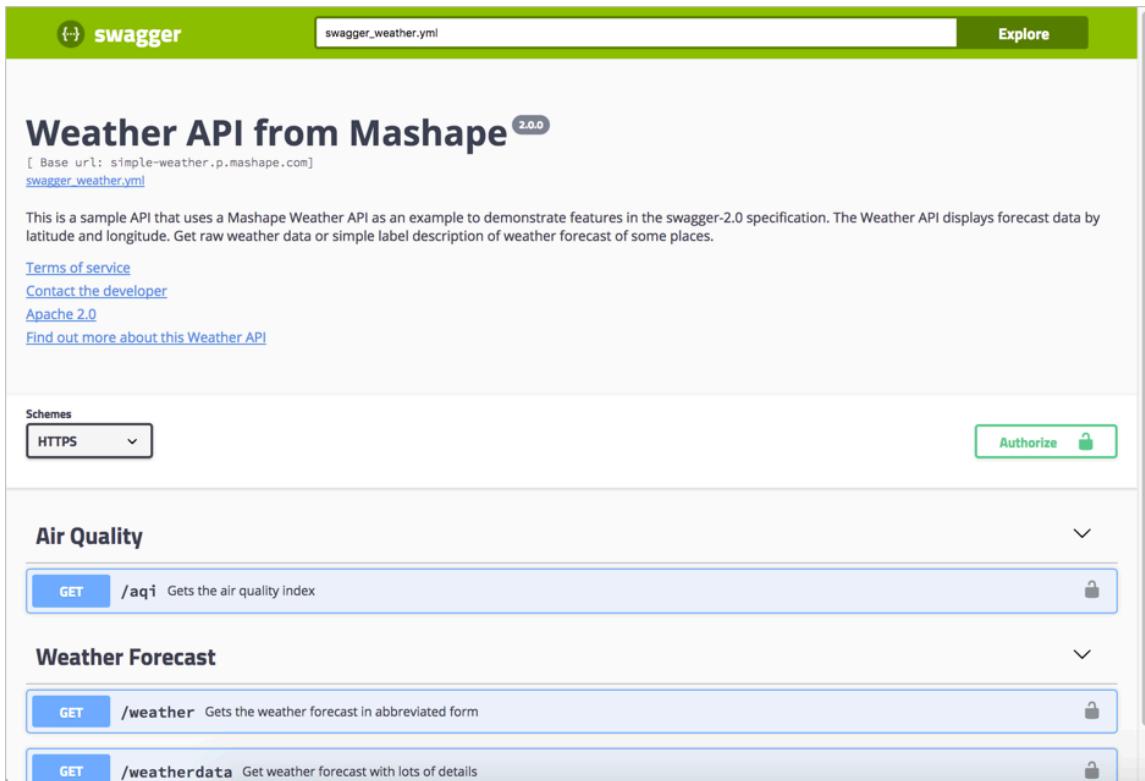
- [Reverb](#)
- [VocaDB](#)
- [Watson Developer Cloud](#)
- [The Movie Database API](#)
- [Zomato API](#)

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 weatherdata endpoint in this [Mashape Weather 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, the file name 'swagger_weather.yml', and an 'Explore' button. Below the header, the title 'Weather API from Mashape' is displayed with a version '2.0.0' badge. A note says '[Base url: simple-weather.p.mashape.com]' and provides a link to 'swagger_weather.yml'. Below this, a descriptive text states: 'This is a sample API that uses a Mashape Weather API as an example to demonstrate features in the swagger-2.0 specification. The Weather API displays forecast data by latitude and longitude. Get raw weather data or simple label description of weather forecast of some places.' There are links for 'Terms of service', 'Contact the developer', 'Apache 2.0', and 'Find out more about this Weather API'. The main content area shows the API structure. Under 'Schemes', 'HTTPS' is selected. An 'Authorize' button with a lock icon is visible. 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'. Each endpoint has a lock icon next to it.

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 \(page 270\)](#) 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 **swagger_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 \(page 281\)](#) and the [info object \(page 277\)](#).

- 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 \(page 334\)](#) for more details on workarounds. You can [customize Swagger UI](#) with your own branding, but it will take 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 \(page 270\)](#) 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](<https://swagger.io/tools/>).

Swagger UI activity: Create your own Swagger UI docs

ACTIVITY

In the [Swagger tutorial \(page 323\)](#), you explored the [Swagger Petstore site](#), learned how the [Swagger Editor](#) works, and how to download and integrate your [OpenAPI specification document \(page 270\)](#) 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 \(page 322\)](#), you should have a functional specification document from an API. If you don't, you can use the [openapi_sunrise_sunset.yml](#). This 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 \(page 331\)](#) 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". At the top, there's a green header bar with the "swagger" logo, a "openapi_sunrise_sunset.yml" link, and an "Explore" button. Below the header, the title "Sunset and sunrise times API" is displayed, along with a "1.0 OAS3" badge. A link to "openapi_sunrise_sunset.yml" is also present. The main content area describes the API as offering sunset and sunrise times for a given latitude and longitude, with links to the "API team - Website" and "Documentation and main site". A "Servers" dropdown menu is set to "https://api.sunrise-sunset.org". Below this, under the "default" section, a "GET /json Get sunrise/sunset times" endpoint is listed. A "Models" dropdown menu is also visible.

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](#), 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 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 \(page 156\)](#) 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 `weatherdata` endpoint in the [sample Mashape Weather API \(page 323\)](#) that we've been using in this course, there's a whole table about condition codes that is essential to interpreting the `item` property in the response. Here's a sample:

```
"condition": {  
    "code": "33",  
    "date": "Wed, 14 Jun 2017 09:00 PM SGT",  
    "temp": "26",  
    "text": "Mostly Clear"  
}
```

What does the code `33` mean? If you go to the [Yahoo Weather API docs](#) (which is where the data for this Mashape weather API originates), you'll see a Condition Codes table that tells you that `33` means "fair (night)". That long table (which includes nearly 50 separate condition codes) will be difficult to include in the parameter details in the OpenAPI spec.

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. 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 \(embedded\) \(page 333\)](#). 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 \(page 333\)](#) 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 \(page 270\)](#), 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. 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:
  description: |
    This is a sample spec that describes a Mashape Weather API as an example to demonstrate features in the Swagger-2.0 specification. This output is part of the <a href="http://idratherbewriting.com/learnapidoc">Documenting REST API course</a> on my site. The Weather API displays forecast data by latitude and longitude. It's a simple weather API, but the data comes from Yahoo Weather Service. The weatherdata endpoint delivers the most robust package of information of the endpoints here.

    To explore the API, you'll need an API key. You can sign up for an API through Mashape, or you can just use this one\: `EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p`. For the latitude and longitude parameters, you can get this information from the URL of a location on Google Maps. For example, for Santa Clara, California, use the following\:
      * **lat**: `37.3708698`
      * **lng**: `-122.037593`
```

With one Swagger API project I worked on, I referenced Bootstrap CSS and JS in the header of the index.html of the Swagger UI project, and then incorporated Bootstrap alerts and expand/collapse buttons in this `description` element. Here's an example:

```
info:
  description: >
    ACME offers a lot of configuration options...
    <div class="alert alert-success" role="alert"><i class="fa fa-info-circle"></i> <b>Tip:</b> See the resources available in the portal for more detail.</div>
    <div class="alert alert-warning" role="alert"><i class="fa fa-info-circle"></i> <b>Note:</b> The network includes a firewall that protects your access to the resources...</div>

    <div class="container">
      <div class="apiConfigDetails">
        <button type="button" class="btn btn-warning" data-toggle="collapse" data-target="#demo">
          <span class="glyphicon glyphicon-collapse-down"></span> See API Configuration Details
        </button>
        <div id="demo" class="collapse">

          <h2>Identifiers Allowed</h2>

          <p>Based on this configuration, ACME will accept any of the following identifiers in requests.</p>

          <table class="table">
            <thead>
              <tr>
                <th>Request Codes</th>
                <th>Data Type</th>
                <th>Comparison Method</th>
              </tr>
            </thead>
            <tbody>
              <tr>
                ...
              </tr>
            </tbody>
          </table>
        </div>
      </div>
    </div>
```

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](#), [DynamicAPIs](#), [Run in Postman](#), [SwaggerHub \(page 340\)](#), 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 \(page 238\)](#). 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, Doxygen, 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 \(page 323\)](#) as a way to render your [OpenAPI specification document \(page 270\)](#). **SwaggerHub** is the commercial version of Swagger UI. You can see a comparison of features [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 interface. At the top, there's a header with the SwaggerHub logo and a user profile for 'tomjohnson1492'. Below the header is a search bar and filter options for sorting by 'Recently updated', 'Type', 'Visibility', 'State', and 'Owner'. A sidebar on the left includes links for 'MY hub', 'Search', and 'Organizations'. The main content area displays a single API entry: 'IdRatherBeWritin...' (API type, Public/Unpublished) titled 'MashapeWeatherAPI'. The description states it's a weather API by fyhao that appears on Mashape and is part of a course on REST API documentation. At the bottom of the page, there's a footer with the SMARTBEAR logo and links to various sections like @swaggerhub, Community, Contact, FAQ, Help, Blog, Terms, Pricing, and Changelog.

The dashboard shows a list of the APIs you've created. In this example, you see the [Weather API](#) (page 30) 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)

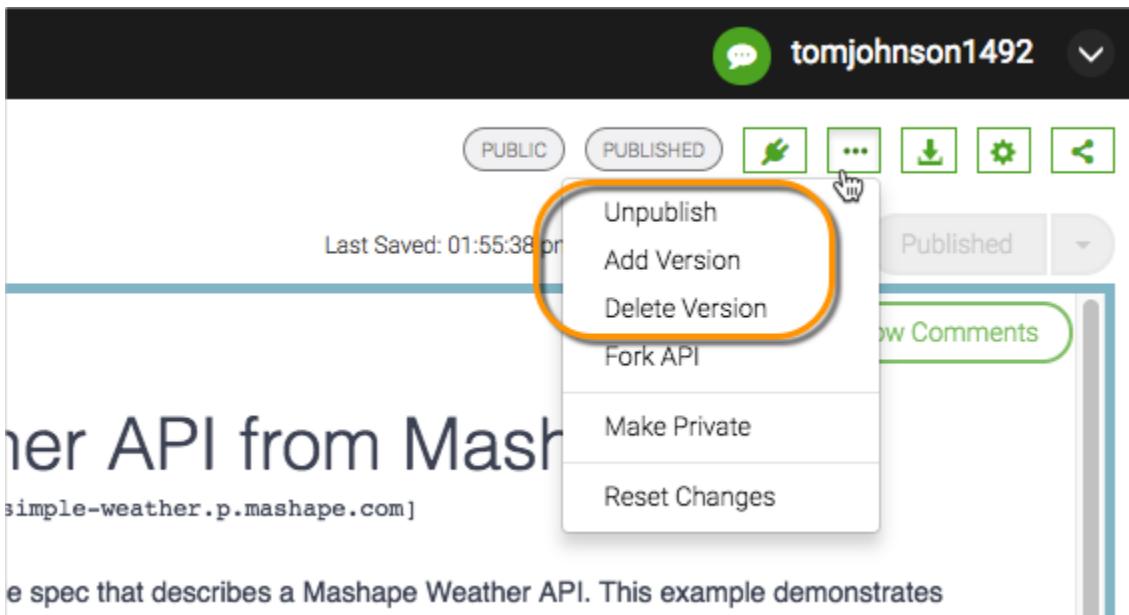
The screenshot shows the SwaggerHub Editor in 'Split' mode. On the left, there's a sidebar with navigation icons for back, forward, and home. The main area has a URL bar showing '/apis / IdRatherBe.../ MashapeWea... / 2.2 -'. Below the URL bar are three tabs: 'Editor' (selected), 'Split', and 'UI'. The 'Editor' tab displays the raw Swagger JSON spec, which includes the 'swagger' key set to '2.0', an 'info' object with a 'description' key containing a sample API description. The 'Split' tab shows the same content as the 'Editor' tab. The 'UI' tab shows a preview of the API's user interface, which is currently empty.

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 Weather API published on SwaggerHub that uses version 2.0 of the spec: <https://app.swaggerhub.com/apis/IdRatherBeWriting/MashapeWeatherAPI/2.2>.

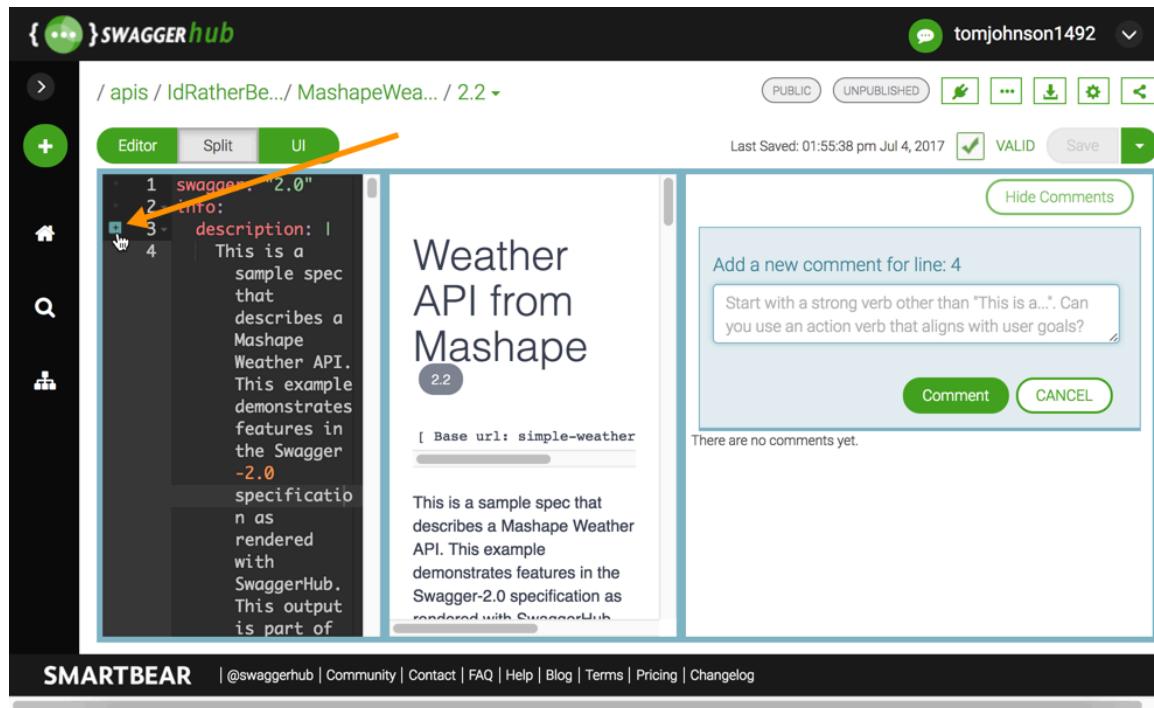
You can also send users to the default version when they go they don't include the version at the end. For example, if you go to <https://app.swaggerhub.com/apis/IdRatherBeWriting/MashapeWeatherAPI>, you get forwarded to the latest published version of the spec automatically.

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.



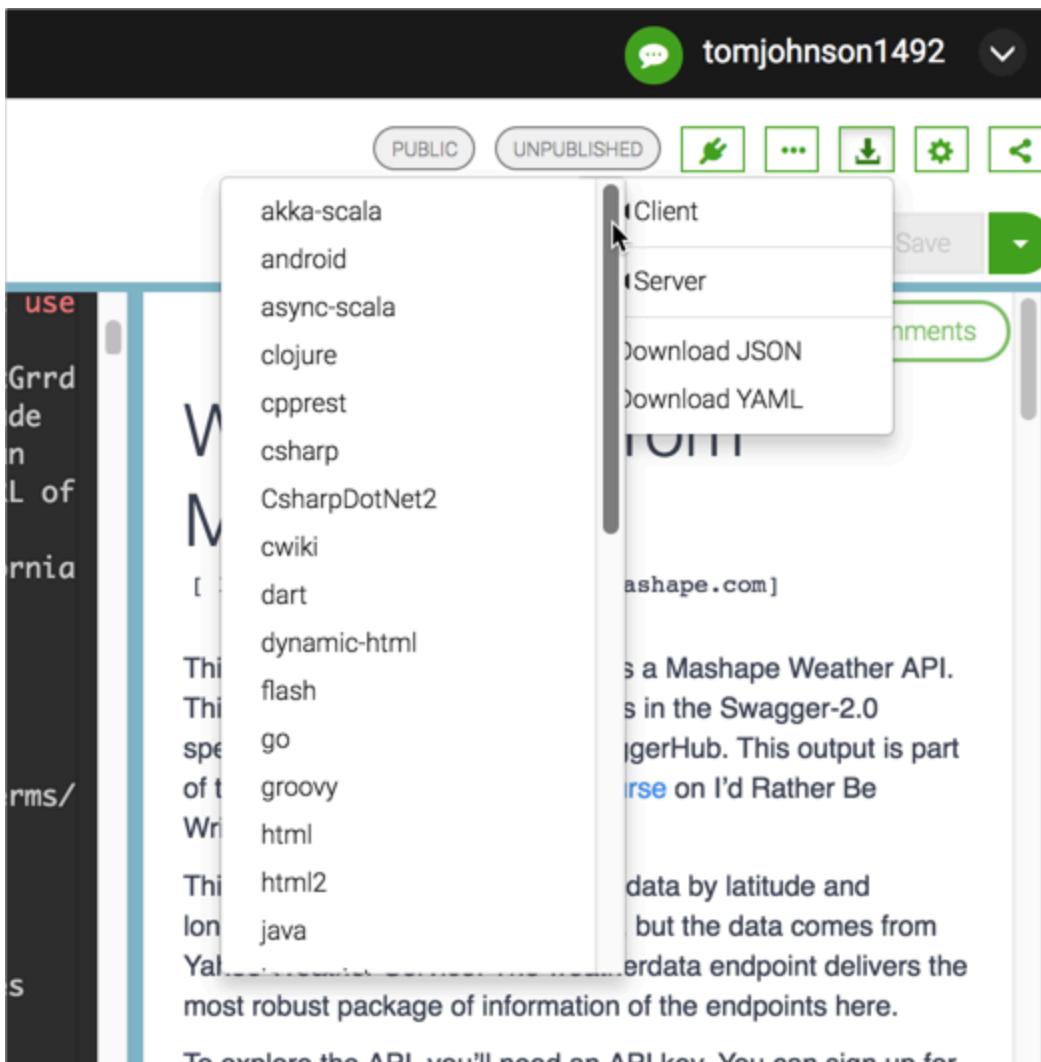
When you click the plus sign, a comment pane appears on the right where you can elaborate on comments, and where others can reply. Users can edit, delete, or resolve the comments. This commenting feature helps facilitate the review process in a way that tightly integrates with your content. You can also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

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 \(page 334\)](#).

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 interface for the `/weatherdata` endpoint. At the top, there is a green button labeled "GET". Below it, the endpoint path `/weatherdata` is shown. A "Usage and SDK Samples" section follows, featuring tabs for "Curl", "Java", "Android", "Obj-C", "JavaScript" (which is selected), "C#", and "PHP". The JavaScript tab displays the following code snippet:

```
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 juking 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.

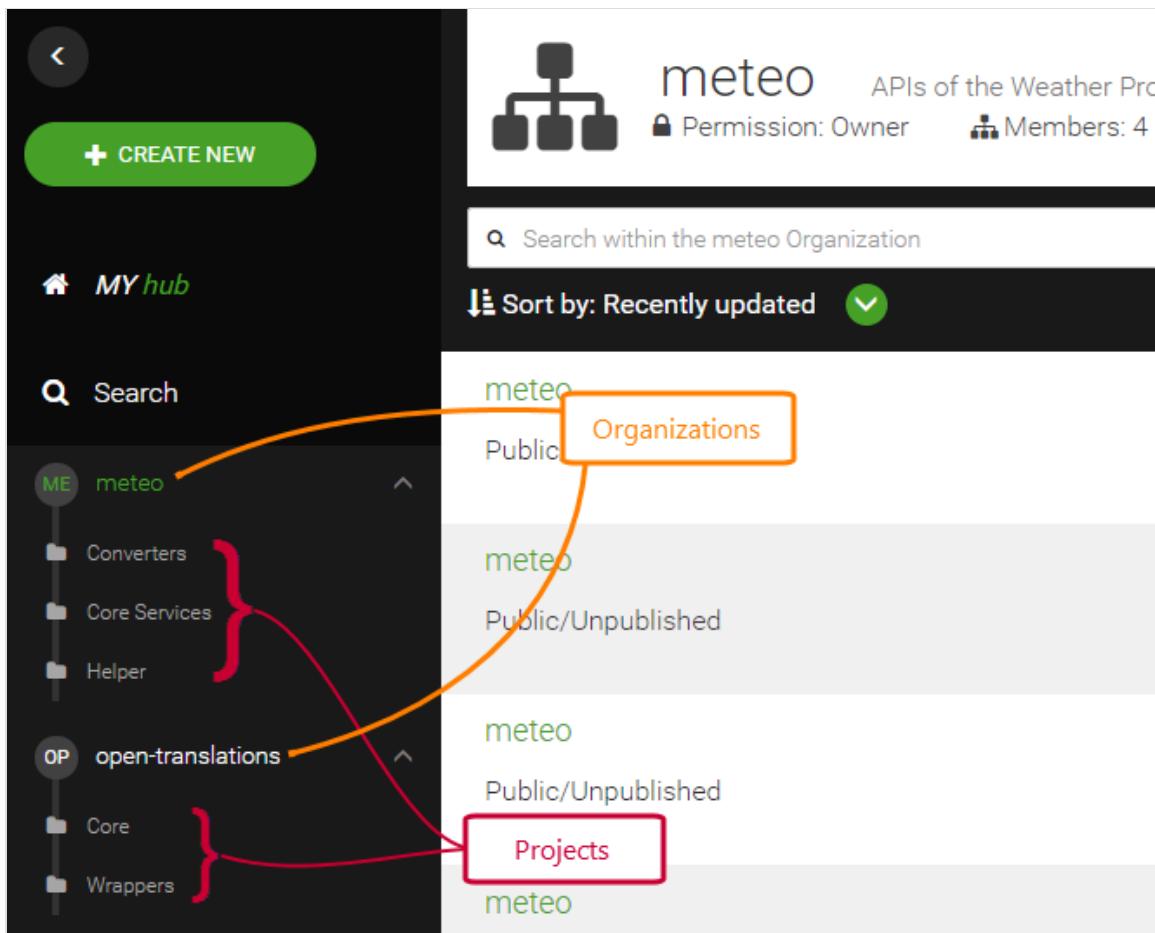
When you create definitions for your requests and responses, you may find yourself re-using the same code over and over. Rather than duplicating this code, you can save it as a domain. When you want to re-use the code, you select this domain.

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.

More about YAML

When you created the [OpenAPI specification \(page 270\)](#), you usually use a syntax called YAML. The file extension can be either “.yaml” or “.yml.” (You can also use [JSON \(page 60\)](#), but the prevailing trend with the OpenAPI document format is YAML.) YAML stands for “YAML Ain’t Markup Language.” This means that the YAML syntax doesn’t have markup tags such as `<` or `>`. Instead, it uses colons to denote an object’s properties and hyphens to denote an array.

Working with YAML

YAML is easier to work with because it removes the brackets, curly braces, and commas that get in the way of reading content.

```
*YAML 1.2
---
YAML: YAML Ain't Markup Language

What It Is: YAML is a human friendly data serialization
standard for all programming languages.

YAML Resources:
  YAML 1.2 (3rd Edition): http://yaml.org/spec/1.2/spec.html
  YAML 1.1 (2nd Edition): http://yaml.org/spec/1.1/
  YAML 1.0 (1st Edition): http://yaml.org/spec/1.0/
  YAML Issues Page: https://github.com/yaml/yaml/issues
  YAML Mailing List: yaml-core@lists.sourceforge.net
  YAML IRC Channel: "#yaml on irc.freenode.net"
  YAML Cookbook (Ruby): http://yaml4r.sourceforge.net/cookbook/ (local)
  YAML Reference Parser: http://yaml.org/ypaste/

Projects:
  C/C++ Libraries:
    - libyaml      # "C" Fast YAML 1.1
    - Syck        # (dated) "C" YAML 1.0
    - yaml-cpp    # C++ YAML 1.2 implementation
  Ruby:
    - psych       # libyaml wrapper (in Ruby core for 1.9.2)
    - RbYaml       # YAML 1.1 (PyYaml Port)
    - yaml4r       # YAML 1.0, standard library syck binding
  Python:
    - PyYaml       # YAML 1.1, pure python and libyaml binding
    - PySyck       # YAML 1.0, syck binding
  Java:
    - JvYaml       # Java port of RbYaml
    - SnakeYAML    # Java 5 / YAML 1.1
    - YamlBeans    # To/from JavaBeans
    - TVaml         # Original Java Implementation
```

The YAML site itself is written using YAML, which you can immediately see is not intended for coding web pages.

YAML is an attempt to create a more human readable data exchange format. It’s similar to JSON (JSON is actually a subset of YAML) but uses spaces, colons, and hyphens to indicate the structure.

Many computers ingest data in a YAML or JSON format. It’s a syntax commonly used in configuration files and an increasing number of platforms (like Jekyll), so it’s a good idea to become familiar with it.

YAML is a superset of JSON

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](#).

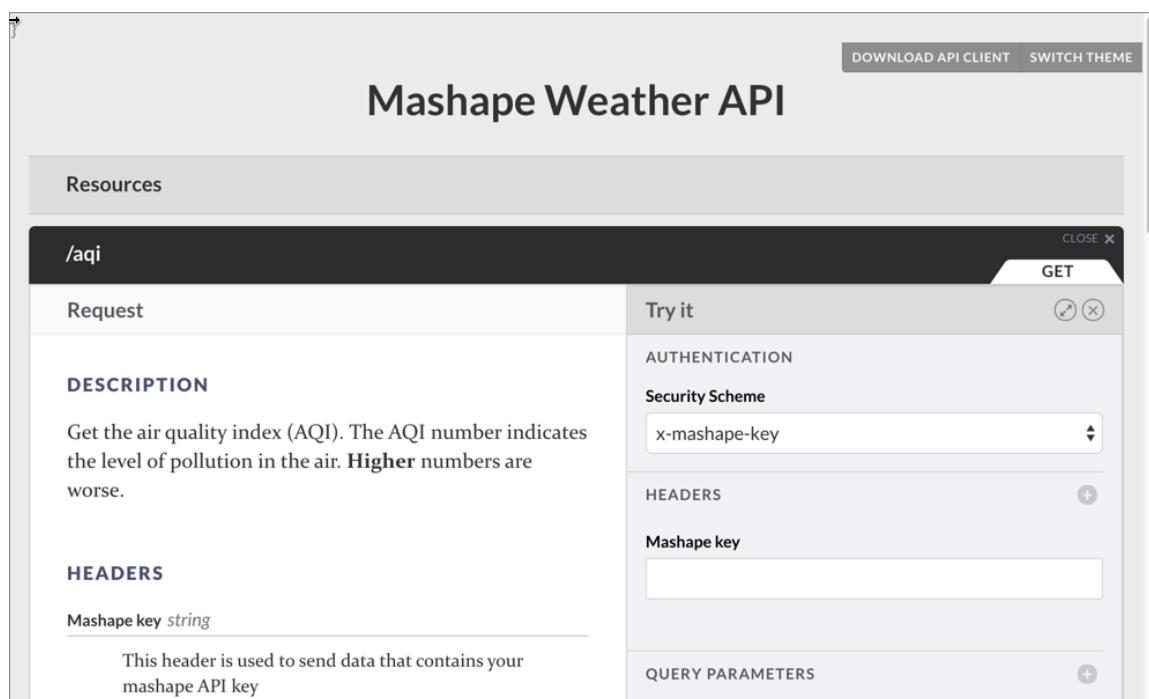
RAML tutorial

RAML stands for REST API Modeling Language and is similar to the [OpenAPI specification \(page 270\)](#). RAML is backed by [Mulesoft](#), a commercial 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 \(page 270\)](#) instead.

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.



The screenshot shows the Mashape Weather API RAML interface. At the top, there's a navigation bar with 'DOWNLOAD API CLIENT' and 'SWITCH THEME' buttons. The main title is 'Mashape Weather API'. Below the title, there's a 'Resources' section with a search bar containing '/aqi'. A modal window titled 'GET /aqi' is open, showing the request details. The 'Request' tab contains a 'DESCRIPTION' section with the text: 'Get the air quality index (AQI). The AQI number indicates the level of pollution in the air. **Higher** numbers are worse.' It also has a 'HEADERS' section with a 'Mashape key string' entry and a note: 'This header is used to send data that contains your mashape API key'. The 'Try it' tab contains sections for 'AUTHENTICATION' (Security Scheme: 'x-mashape-key'), 'HEADERS' (Mashape key), and 'QUERY PARAMETERS'.

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 Mashape Weather 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 Mashape Weather API we've been using in this course formatted in the RAML spec:

```
#%RAML 0.8
---
title: Mashape Weather API
baseUri: https://simple-weather.p.mashape.com
version: v1

/aqi:
  get:
    description: Get the air quality index (AQI). The AQI number indicates the level of pollution in the air. **Higher** numbers are worse.
    headers:
      x-mashape-key:
        displayName: Mashape key
        description: This header is used to send data that contains your mashape API key
        type: string
    queryParameters:
      lat:
        displayName: Latitude
        description: The latitude coordinate
        type: number
        required: true
        example: 37.354108
      lng:
        type: number
        description: The longitude coordinate
        required: true
        example: -121.955236
    responses:
      200:
        body:
          application/text:
            example: |
              65

/weather:
  get:
    headers:
      x-mashape-key:
        displayName: Mashape key
        description: This header is used to send data that contains your mashape API key
        type: string
    description: Gets the weather forecast for the current day
    queryParameters:
      lat:
        displayName: Latitude
        description: The latitude coordinate
        type: number
        required: true
```

```
    example: 37.354108
  lng:
    type: number
    description: The longitude coordinate
    required: true
    example: -121.955236
  responses:
    200:
      body:
        application/text:
          example: |
            28 c, Partly Cloudy at Santa Clara, United States
/weatherdata:
  get:
    headers:
      x-mashape-key:
        displayName: Mashape key
        description: This header is used to send data that contains your mashape API key
        type: string
        description: Gets a detailed weather object containing a lot of different weather information in a JSON object.
    queryParameters:
      lat:
        displayName: Latitude
        description: The latitude coordinate
        type: number
        required: true
        example: 37.354108
      lng:
        type: number
        description: The longitude coordinate
        required: true
        example: -121.955236
    responses:
      200:
        body:
          application/json:
            example: |
              {
                "query": {
                  "count": 1,
                  "created": "2014-05-03T03:57:53Z",
                  "lang": "en-US",
                  "results": {
                    "channel": {
                      "title": "Yahoo! Weather - Tebrau, MY",
                      "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau_MY/*http://weather.yahoo.c/forecast/MYXX0004_c.html",
                      "description": "Yahoo! Weather for Tebrau, MY",
                      "language": "en-us",
                    }
                  }
                }
              }
```

```
"lastBuildDate": "Sat, 03 May 2014 11:00 am MYT",
"ttl": "60",
"location": {
    "city": "Tebrau",
    "country": "Malaysia",
    "region": ""
},
"units": {
    "distance": "km",
    "pressure": "mb",
    "speed": "km/h",
    "temperature": "C"
},
"wind": {
    "chill": "32",
    "direction": "170",
    "speed": "4.83"
},
"atmosphere": {
    "humidity": "66",
    "pressure": "982.05",
    "rising": "0",
    "visibility": "9.99"
},
"astronomy": {
    "sunrise": "6:57 am",
    "sunset": "7:06 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-wea.gif"
},
"item": {
    "title": "Conditions for Tebrau, MY at 11:00 am MYT",
    "lat": "1.58",
    "long": "103.74",
    "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau__MY/*http://weather.yahoo.com/forecast/MYXX004_c.html",
    "pubDate": "Sat, 03 May 2014 11:00 am MYT",
    "condition": {
        "code": "28",
        "date": "Sat, 03 May 2014 11:00 am MYT",
        "temp": "32",
        "text": "Mostly Cloudy"
    },
    "description": "\n<img src=\"http://l.yimg.com/a/i/us/we/52/28.gif\"/><br />\n<Current Conditions:</b><br />\nMostly Cloudy, 32 C<BR
```

```
>>\n<BR /><b>Forecast:</b><BR />\nSat - Scattered Thunderstorms. High: 32 Lo  
w: 26<br />\nSun - Thunderstorms. High: 33 Low: 27<br />\nMon - Scattered Th  
understorms. High: 32 Low: 26<br />\nTue - Thunderstorms. High: 32 Low: 26<br  
/>\nWed - Scattered Thunderstorms. High: 32 Low: 27<br />\n<br />\nFull Forecast at Yahoo! Weather</a><B  
R/><BR/>\n\(provided by <a href=\"http://www.weather.com\">The Weather Chann  
el</a>\)<br/>\n",  
    "forecast": \[  
        {  
            "code": "38",  
            "date": "3 May 2014",  
            "day": "Sat",  
            "high": "32",  
            "low": "26",  
            "text": "Scattered Thunderstorms"  
        },  
        {  
            "code": "4",  
            "date": "4 May 2014",  
            "day": "Sun",  
            "high": "33",  
            "low": "27",  
            "text": "Thunderstorms"  
        },  
        {  
            "code": "38",  
            "date": "5 May 2014",  
            "day": "Mon",  
            "high": "32",  
            "low": "26",  
            "text": "Scattered Thunderstorms"  
        },  
        {  
            "code": "4",  
            "date": "6 May 2014",  
            "day": "Tue",  
            "high": "32",  
            "low": "26",  
            "text": "Thunderstorms"  
        },  
        {  
            "code": "38",  
            "date": "7 May 2014",  
            "day": "Wed",  
            "high": "32",  
            "low": "27",  
            "text": "Scattered Thunderstorms"  
        }  
    "guid": {
```

Outputs

You can generate outputs using the RAML spec from a variety of platforms. Here are three ways:

- [Developer Portal on Anypoint platform](#). Choose this option if you are developing and delivering your API on Mulesoft’s Anypoint platform.
 - [API Console output](#). 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](#). Choose this option if you want a simple HTML output of your API documentation. No interactive console is included.

Deliver doc through the Anypoint Platform Developer Portal

1. Log into the [Anypoint platform](#).
 2. Click **APIs** on the top navigation.
 3. Click **Add new API** and complete the details of the dialog box.
 4. Click the API you just added.
 5. In the API Definition box, click **Edit in API Designer**.
 6. Input your RAML spec here (copy it from the above section), and then click **Save**.
 7. Click **APIs** on the top navigation, and then click your API.
 8. In the API Portal section, click **Create new portal**.
 9. In the left pane, select **API Reference**.

Note that you can add additional pages to your documentation here.

The screenshot shows the Anypoint Platform interface with the following details:

- Header:** CloudHub, API, Exchange, Support, Settings, tomjohnson1492
- Breadcrumbs:** API administration > weather - 1.0 > Portal > API reference
- Title:** weather
- Actions:** Private (with lock icon), Themes (with globe icon), Live portal (with monitor icon)
- Left sidebar (dropdown menu):**
 - Add
 - Page (selected, indicated by a red arrow)
 - API Notebook
 - API Reference
 - Header
 - External link
- Right main area:**

API reference

API is behind a firewall (?)

Resources	HTTP Method
/aqi	GET
/weather	GET
/weatherdata	GET

RAML URL: /apiplatform/repository/v2/organizations/bdd21472-36b7-43d2-a8a1-20d51081ba5/public/apis/34130/versions/35503/files/root

(Kudos to the Mulesoft team for recognizing that API documentation is more than just a set of reference endpoints.)

One of the options here is an API Notebook. This is a unique tool designed by Mulesoft that allows you to provide interactive code examples that leverage your RAML spec. You can read more about [API Notebooks here](#).

10. Click the **Set to visible** icon (looks like an eye).
11. Click **Live Portal**.

The screenshot shows the MuleSoft API Developer portal interface. At the top, there's a header with a user icon and the text 'tomjohnson1492'. Below the header, a navigation bar has 'Developer portal' and 'weather - 1.0' selected. On the left, a sidebar has 'Home' and 'API reference' tabs, with 'API reference' being the active one. The main content area is titled 'API reference' and shows a 'Resources' section. Three API endpoints are listed:

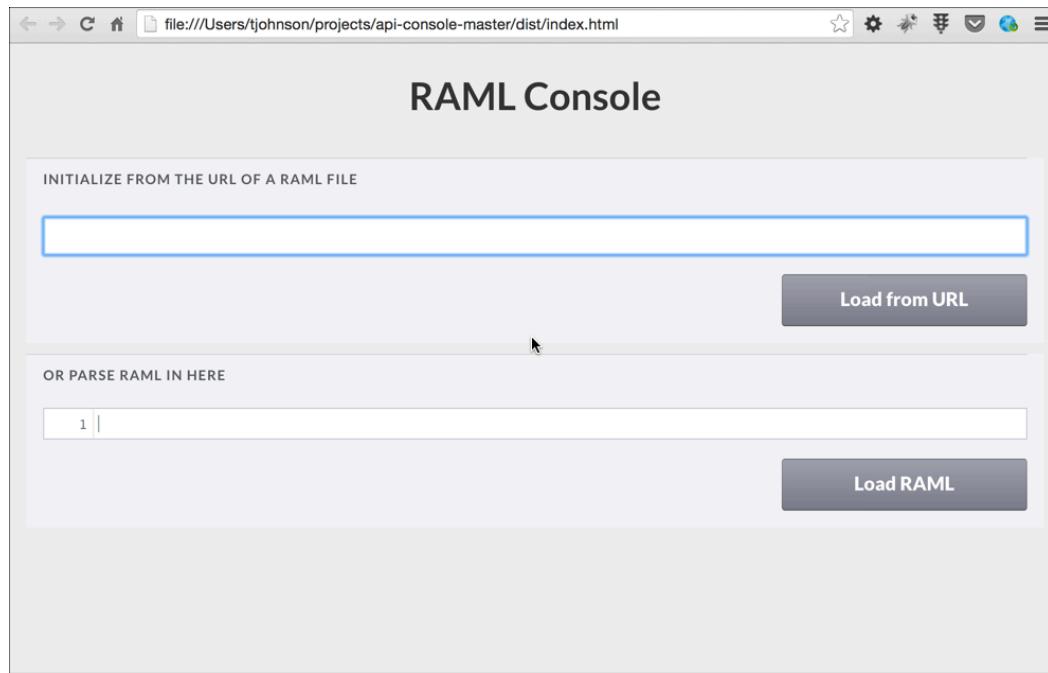
- /aqi (with a 'GET' button)
- /weather (with a 'GET' button)
- /weatherdata (with a 'GET' button)

A note at the bottom says 'API is behind a firewall (?)' with a checkbox. At the very bottom, it shows the RAML URL: [/apiplatform/repository/v2/organizations/bdd21472-36b7-43d2-a8a1-20d51081bba5/public/apis/34130/versions/35503/files/root](https://apiplatform/repository/v2/organizations/bdd21472-36b7-43d2-a8a1-20d51081bba5/public/apis/34130/versions/35503/files/root).

Deliver doc through the API Console Project

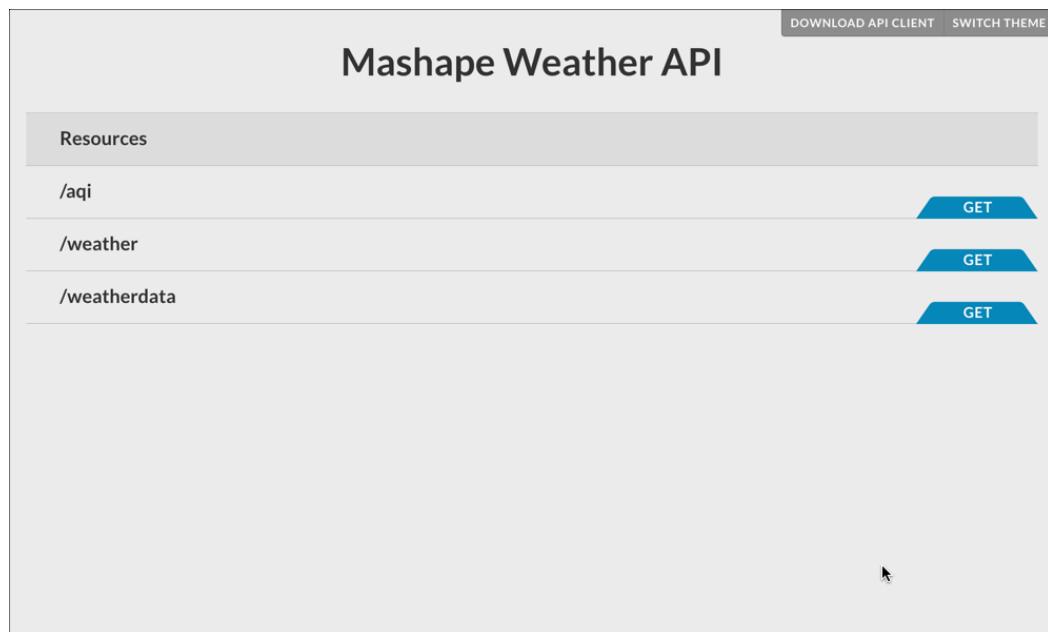
You can also download the same code that generates the output on the Anypoint Platform and create your own API Console.

1. Download the [API Console code from Github](#).
2. Save your RAML file to some place locally on your computer (such as weather.raml on Desktop).
3. In the code you downloaded from Github, go to dist/index.html in your browser.



4. Copy the RAML code you created.
5. Insert your copied code into the **Or parse RAML here** text box. Then click **Load RAML**.

The API Console loads your RAML content:



6. To auto-load a specific RAML file, add this to the body of the index.html file:

```
<div style="overflow:auto; position:relative">
  <raml-console src="examples/weather.raml"></raml-console>
</div>
```

In this example, the RAML file is located in examples/weather.raml.

7. Remove the following line:

```
<raml-initializer></raml-initializer>
```

View the file in your web browser. Note that if the file doesn't load in Chrome, open it in Firefox. Chrome tends to block local JavaScript for security reasons.

Here's a [sample RAML API Console output](#) that integrates the weather.raml file. Here's a [generic RAML API Console](#) that allows you to insert your own RAML spec code.

Deliver doc through the RAML2HTML Utility

Here's [an example](#) of what the RAML2HTML output looks like. It's a static HTML output without any interactivity.

To generate this kind of output:

1. Install RAML2HTML through either of these methods:
 - [Through Node](#)
 - [Through wget](#)
2. In Terminal, enter this command:

```
raml2html generate -i input_file.raml -o output_file.html
```

For example, if you're already in the directory where your RAML file is (named weather.raml), and you want the output file to be named index.html, then enter this:

```
raml2html generate -i weather.raml -o index.html
```

Here's the result:

The screenshot shows the Mashape API documentation interface. On the left, there's a sidebar with endpoints: /aqi, /weather, and /weatherdata. The /aqi endpoint is currently selected. A modal window is open for the /aqi endpoint, showing the following details:

- Method:** GET /aqi
- Description:** Get the air quality index (AQI). The AQI number indicates the level of pollution in the air. **Higher numbers are worse.**
- Request:** Headers
 - Mashape key: (string) This header is used to send data that contains your mashape API key
- Query Parameters:**
 - Latitude: required (number) The latitude coordinate
 - Example: 37.354108
 - Long: required (number) The longitude coordinate
 - Example: -121.955236

To see this example in your browser, go to </learnapidoc/assets/files/examples/index.html>.

Other platforms that consume RAML and Swagger

[Restlet Studio](#) is another platform to check out. Restlet Studio can process either Swagger or RAML specs.

One advantage to using Restlet Studio is that it provides a form where you can assemble the needed spec by populating different form values, which theoretically should make building the RAML or Swagger file easier.

Exploring more platforms in depth is beyond the scope of this tutorial, but the concept is more or less the same. Large platforms that process and display your API documentation can only do so if your documentation aligns with a spec their tools can parse.

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 \(page 270\)](#) 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. On the left, there is a code editor window containing the API Blueprint syntax for the Polls API. The code defines a resource named 'Polls' with a root endpoint. It includes a detailed description of the endpoint and a response example. On the right, the generated API documentation is displayed. It features a header titled 'Reference > Polls API Root' and a main section titled 'Polls'. Below the title, there is an 'INTRODUCTION' section with a brief description of the API's purpose. At the bottom of the documentation page, there is a 'REFERENCE' section.

```
1 FORMAT: 1A
2 HOST: http://polls.apiblueprint.org/
3
4 # Polls
5
6 Polls is a simple API allowing consumers to view polls and
7
8 # Polls API Root [/]
9
10 This resource does not have any attributes. Instead it offers
11 API affordances in the form of the links in the JSON body.
12
13 It is recommended to follow the "url" link values,
14 [Link](https://tools.ietf.org/html/rfc5988) or Location header
15 applicable to retrieve resources. Instead of constructing URLs
16 to keep your client decoupled from implementation details.
17
18 ## Retrieve the Entry Point [GET]
19
20 + Response 200 (application/json)
21
22     {
23         "questions_url": "/questions"
24     }
25
26 ## Group Question
```

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 Mashape weather API information into the blueprint format.

4. Copy the following code, which aligns with the API Blueprint spec, and paste it into the Apiary blueprint editor.

```
FORMAT: 1A
HOST: https://simple-weather.p.mashape.com

# Weather API

Display Weather forecast data by latitude and longitude. Get raw weather data OR simple label description of weather forecast of some places.

# Weather API Root [/]

# Group Weather

Resources related to weather in the API.

## Weather data [/weatherdata{?lat,lng}]

#### Get the weather data [GET]

Get the weather data in your area.

+ Parameters
  + lat: 55.749792 (required, number) - Latitude
  + lng: 37.632495 (required, number) - Longitude

+ Request JSON Message

  + Headers
    X-Mashape-Authorization: APIKEY
    Accept: text/plain

+ Response 200 (application/json)

  + Body
    [
      {
        "query": {
          "count": 1,
          "created": "2014-05-03T03:57:53Z",
          "lang": "en-US",
          "results": {
            "channel": {
              "title": "Yahoo! Weather - Tebrau, MY",
              "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau_MY/*http://weather.yahoo.com/forecast/MYXX0004_c.html",
              "description": "Yahoo! Weather for Tebrau, MY",
              "language": "en-us",
              "lastBuildDate": "Sat, 03 May 2014 11:00 am MYT",
              "ttl": "60",
            }
          }
        }
      }
    ]
```

```
        "location": {
            "city": "Tebrau",
            "country": "Malaysia",
            "region": ""
        },
        "units": {
            "distance": "km",
            "pressure": "mb",
            "speed": "km/h",
            "temperature": "C"
        },
        "wind": {
            "chill": "32",
            "direction": "170",
            "speed": "4.83"
        },
        "atmosphere": {
            "humidity": "66",
            "pressure": "982.05",
            "rising": "0",
            "visibility": "9.99"
        },
        "astronomy": {
            "sunrise": "6:57 am",
            "sunset": "7:06 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-wea.gif"
        },
        "item": {
            "title": "Conditions for Tebrau, MY at 11:00 am MYT",
            "lat": "1.58",
            "long": "103.74",
            "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Tebrau__MY/*http://weather.yahoo.com/forecast/MYXX0004_c.html",
            "pubDate": "Sat, 03 May 2014 11:00 am MYT",
            "condition": {
                "code": "28",
                "date": "Sat, 03 May 2014 11:00 am MYT",
                "temp": "32",
                "text": "Mostly Cloudy"
            },
            "description": "\n<img src=\"http://l.yimg.com/a/i/us/we/52/28.gif\"/><br />\n<b>Current Conditions:</b><br />\nMostly Cloudy, 32 C<BR />\n<b>Forecast:</b><BR />\nSat - Scattered T
```

```
hunderstorms. High: 32 Low: 26<br />\nSun - Thunderstorms. High: 33 L
ow: 27<br />\nMon - Scattered Thunderstorms. High: 32 Low: 26<br />\n
Tue - Thunderstorms. High: 32 Low: 26<br />\nWed - Scattered Thunders
torms. High: 32 Low: 27<br />\n<br />\nFull Forecast at Yahoo! Weather</a><BR/><BR/>\n\(p
rovided by <a href="http://www.weather.com/">The Weather Channe
l</a><br/>\n",
    "forecast": \[
        {
            "code": "38",
            "date": "3 May 2014",
            "day": "Sat",
            "high": "32",
            "low": "26",
            "text": "Scattered Thunderstorms"
        },
        {
            "code": "4",
            "date": "4 May 2014",
            "day": "Sun",
            "high": "33",
            "low": "27",
            "text": "Thunderstorms"
        },
        {
            "code": "38",
            "date": "5 May 2014",
            "day": "Mon",
            "high": "32",
            "low": "26",
            "text": "Scattered Thunderstorms"
        },
        {
            "code": "4",
            "date": "6 May 2014",
            "day": "Tue",
            "high": "32",
            "low": "26",
            "text": "Thunderstorms"
        },
        {
            "code": "38",
            "date": "7 May 2014",
            "day": "Wed",
            "high": "32",
            "low": "27",
            "text": "Scattered Thunderstorms"
        }
    \],
    "guid": {
```

```
        "isPermaLink": "false",
        "content": "MYXX0004_2014_05_07_7_00_MYT"
    }
}
}
]
}
```

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.

Getting a job in API documentation

The job market for API technical writers.....	374
How much code do you need to know?	379

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.

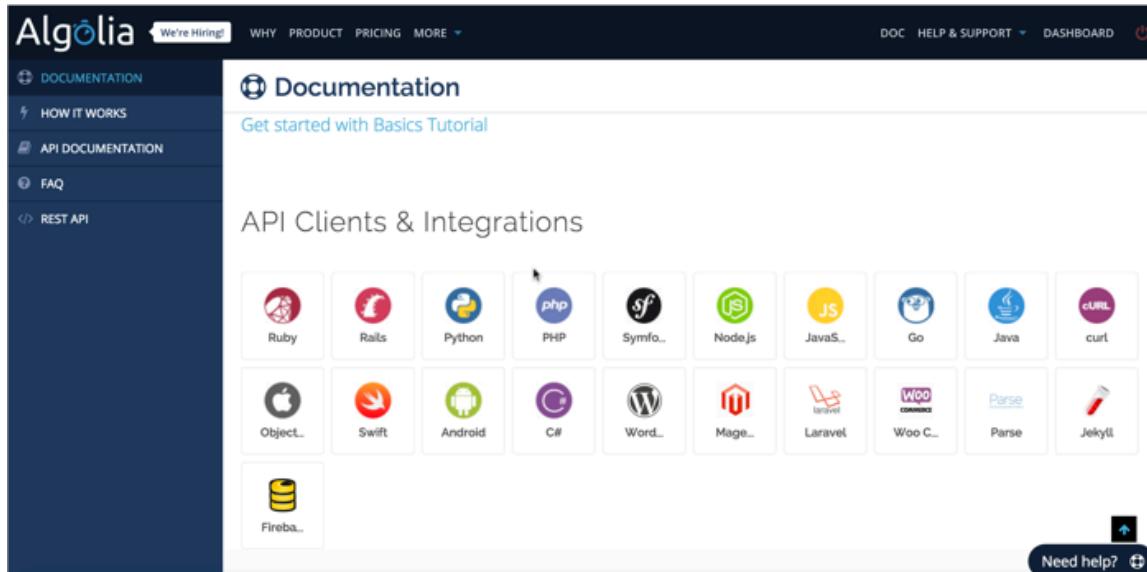
Ability to read programming languages

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#, Word, Mage, Laravel, Woo C, 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.....	383
Overview for exploring other REST APIs	389
EventBrite example: Get event information.....	390
Flickr example: Retrieve a Flickr gallery	397
Klout example: Retrieve Klout influencers.....	406
Aeris Weather Example: Get wind speed	416

API glossary

API

Application Programming Interface. Enables different systems to interact with each other programmatically. Two types of APIs are web services and library-based APIs. See [What is a REST API? \(page 22\)](#).

API Console

Renders an interactive display for the RAML spec. Similar to Swagger UI, but for [RAML \(page 353\)](#). See github.com/mulesoft/api-console.

APIMATIC

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

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.

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 \(page 363\)](#) in this course, or go to [API Blueprint's homepage](#) to learn more.

Apigee

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

Asciidoc

A lightweight text format that provides more semantic features than Markdown. Used in some static site generators, such as [Asciidoctor](#) or [Nanoc](#). See <http://asciidoc.org/>.

branch

In Git, a branch is a copy of the repository that is often used for developing new features. Usually you

work in branches and then merge the branch into the master branch when you're ready to publish. If you're editing documentation in a code repository, developers will probably have you work in a branch to make your edits. The developers will then either merge 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 \(page 46\)](#) and [Understand cURL more \(page 49\)](#).

endpoint

The end part of the request URL (after the base path). Also sometimes used to refer to the entire API reference topic. See [Describe the endpoints and methods \(page 92\)](#).

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 \(page 200\)](#) in this course as well as on pages.github.com/.

git repo

A tool for consolidating and managing many smaller repos with one system. See [git-repo](#).

HAT

Help Authoring Tool. Refers to the traditional help authoring tools (RoboHelp, Flare, Author-it, etc.) used by technical writers for documentation. Tooling for API docs tend to use [Docs as code tools \(page 197\)](#) more than [HATs \(page 245\)](#).

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](#).

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 \(page 60\)](#) 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/>.

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 \(page 352\)](#). See <https://www.mulesoft.com/>.

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

A value usually passed into an endpoint that affects the response in some way. REST has four possible parameter types: head, path, query, and body parameters. See [Documenting parameters \(page 95\)](#) for more.

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>.

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 \(page 352\)](#) 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).

REST API

Stands for Representational State Transfer. Uses web protocols (HTTP) to make requests and provide responses in a language agnostic way, meaning that users can choose whatever programming language they want to make the calls. See [What is a REST API? \(page 22\)](#).

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 \(page 227\)](#) 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 \(page 340\)](#).

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 \(page 348\)](#).

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 \(page 390\)](#)
- [Flickr example: Retrieve a Flickr gallery \(page 397\)](#)
- [Klout example: Retrieve Klout influencers \(page 406\)](#)
- [Aeris Weather Example: Get wind speed \(page 416\)](#)

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. 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
<code>q</code>	<code>string</code>	No	Return events matching the given keywords.
<code>since_id</code>	<code>string</code>	No	Return events after this Event ID.
<code>popular</code>	<code>boolean</code>	No	Boolean for whether or not you want to only return popular results.
<code>sort_by</code>	<code>string</code>	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 \(page 270\)](#) 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 an event titled "An Aggressive Approach to Concise Writing, with Joe Welinske". The event is marked as "COMPLETED". It was a Webinar (online through gotomeeting) on Thursday, September 24, 2015, from 12:00 PM to 1:00 PM (PDT). The dashboard includes sections for "EDIT", "DESIGN", and "MANAGE". On the left, there's a sidebar with links for "Order Options", "Invite & Promote", "Analyze", "Manage Attendees", and "Extensions". The main area displays a summary card with "Completed" status, "23 Tickets Sold / 24" (represented by a green progress bar), and a circular progress bar for "Tickets sold All time" which is mostly green. To the right, there are cards for "Page views" (12) and "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><SPAN STYLE=\"font-size: medium;\"><STRONG>Webinar Description</STRONG></SPAN></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.<BR> <BR>Even outside of the mobile space, your ability to streamline your content improves the likelihood it will be effectively consumed by your target audience.<BR> <BR>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><SPAN STYLE=\"font-size: medium;\"><STRONG>About Joe Welinske</STRONG></SPAN><BR>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](#):

The screenshot shows a web browser window with the following details:

- Title Bar:** file:///Users/tjohnson/Desktop/eventbrite.html
- Content Area:**
 - Section Header:** An Aggressive Approach to Concise Writing, with Joe Welinske
 - Section Header:** Webinar Description
 - Text:** 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.
 - 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.
 - Text:** 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.
- Right Sidebar:** Contains a section titled "About Joe Welinske" with a bio describing his expertise in user experience and software development.

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.

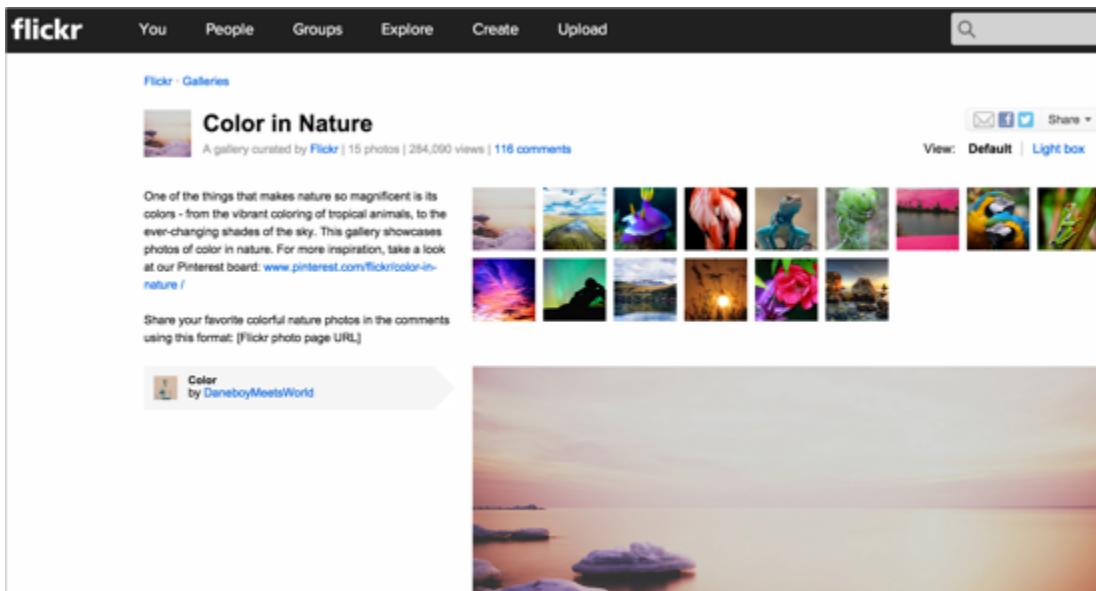
Flickr example: Retrieve a Flickr gallery

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 page. At the top, there is a navigation bar with links for 'Sign Up', 'API Documentation', 'Explore', 'Create', 'Feedback', and 'What is the App Garden?'. Below the navigation bar, the title 'flickr.galleries.getPhotos' is displayed in large bold letters. A description box below the title states 'Return the list of photos for a gallery'. The main content area is titled 'Authentication' in pink, followed by a note that authentication is not required. It then lists several arguments with descriptions:

- 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, 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)

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 links for 'Explore' and 'Create'. 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'. Under this, there's a section for 'Arguments' with a table:

Name	Required	Send	Value
url	required	<input type="checkbox"/>	<input type="text"/>

Below the table, there's a dropdown menu set to 'XML (REST)'. Underneath it, there are two radio buttons: one selected ('Sign call with no user token?') and one unselected ('Do not sign call?'). At the bottom of the form is a 'Call Method...' button.

At the very bottom of the form, there's a link 'Back to the flickr.urls.lookupGallery documentation'.

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				Useful Values
Name	Required	Send	Value	
gallery_id	required	<input checked="" type="checkbox"/>	66911286-7215764727	Your user ID: 86824645@N00
extras	optional	<input type="checkbox"/>		Your recent photo IDs: 19271714336 - 19111657009 - 19110196618 -
per_page	optional	<input type="checkbox"/>		Your recent photaset IDs: 72157651486110150 - Auto U 72157649733910233 - hover 72157649565389074 - Discover
page	optional	<input type="checkbox"/>		Your recent group IDs:

Output: **JSON**

Sign call as tomhenryjohnson with full permissions?

Sign call with no user token?

Do not sign call?

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": "b0e55d455f",
        "server": "8450",
        "farm": 1
      },
      {
        "id": "2209143676",
        "owner": "14478436@N02",
        "secret": "ae987333b5",
        "server": "2072",
        "farm": 1
      },
      {
        "id": "399296912",
        "owner": "58329132@N00",
        "secret": "6adcc29651",
        "server": "161",
        "farm": 1
      },
      {
        "id": "5812344633",
        "owner": "47690289@N02",
        "secret": "af53e53b1",
        "server": "3277",
        "farm": 1
      },
      {
        "id": "4960822520",
        "owner": "48600090482@N01",
        "secret": "d30948b0d5",
        "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

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:



The screenshot shows a browser window displaying a JSON response from the Flickr API. The URL in the address bar is `https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api_key=acada0ff2bb2747f40e0b22767&gallery_id=66911286-72157647277042064&format=json&nojsoncallback=1`. The JSON response is shown in a collapsible tree view. The root object contains a "photos" key, which has properties: "page": 1, "pages": 1, "perpage": 500, and "total": 15. The "photo" key is an array containing two objects, each representing a photo. The first photo has an ID of 8432423659, owned by user 37107167@N07, with a secret of dd1b834ec5, server 8187, farm 9, title "Color", ispublic 1, isfriend 0, isfamily 0, is_primary 1, and has_comment 0. The second photo has an ID of 8047948330, owned by user 70121902@N00, with a secret of b0e55d455f, server 8450, farm 9, title "Owens River and Sea Grass", ispublic 1, and isfriend 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}_{mstzb}.jpg  
or  
https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{o-secret}_o.{jpg|gif|png}
```

Here's what an item in the JSON response looks like:

```
"photos": {  
    "page": 1,  
    "pages": 1,  
    "perpage": 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

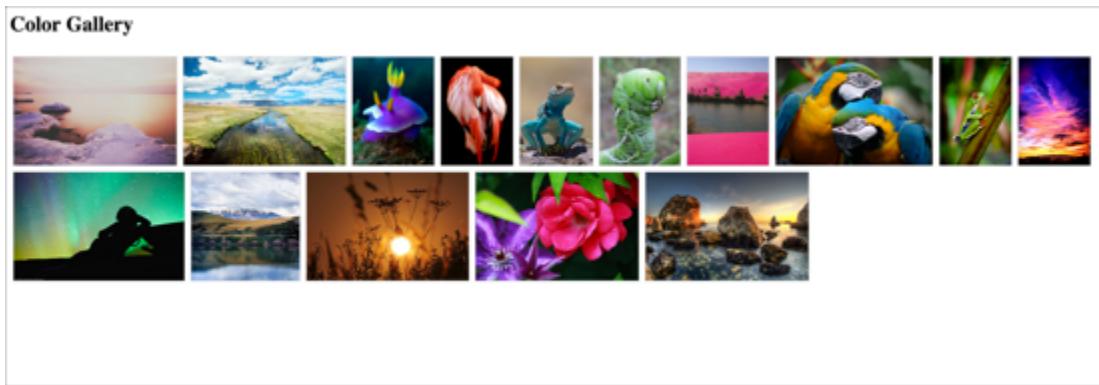
    $("#flickr").append('');
  });
});

</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. On the left, there's a sidebar with links for 'INTERACTIVE CONSOLE', 'TERMS OF SERVICE', 'CODE LIBRARY', 'DOCS' (with a 'VERSION 2' link), and 'REGISTER AN APP'. The main 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 it, an 'App/Key:' field contains the placeholder 'I'd rather be writing: urgey4a79n5x6df6xx4p64dr'. There are two buttons: 'Manually provide key information' (in orange) and 'Toggle All Endpoints' (in grey). Another 'Toggle All Methods' button is also present. The central part is titled 'Identity Methods' with 'List Methods' and 'Expand Methods' links. It lists four GET endpoints: 'Identity(twitter_id)', 'Identity(Google+)', 'Identity(Instagram)', and 'Identity(twitter_screen_name)'. The 'Identity(twitter_screen_name)' endpoint is expanded, showing a table with a single row for 'screenName' with value 'tomjohnson' and type 'string'. A note below the table says: 'This method allows you to retrieve a KloutID for a twitter screen_name'.

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.

GET `/identity.json/twitter`

This method allows you to retrieve a KloutID for a twitter screen_name

Parameter	Value	Type	Description
screenName	tomjohnson	string	A twitter screen name (e.g. jtimmerlake)

Try it! **Clear Results**

Request URI

```
http://api.klout.com/v2/identity.json/twitter?
screenName=tomjohnson&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: 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
```

Response Body **Select content**

```
{
  "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('<li><a href="http://twitter.com/' + inf.entity.payload.nick + '">' + inf.entity.payload.nick + '</a></li>');
    });
    $.each( data.myInfluencers, function( i, inf ) {
        $("#kloutinfluencers").append('<li><a href="http://twitter.com/' + inf.entity.payload.nick + '">' + inf.entity.payload.nick + '</a></li>');
    });
});
</script>

<h2>My influencees (people I influence)</h2>
<ul id="kloutinfluencees"></ul>

<h2>My influencers (people who influence me)</h2>
<ul id="kloutinfluencers"></ul>

</body>
</html>
```

Remember to swap in your own API key in place of `APIKEY`. The result looks like this:

The screenshot shows a web browser window with the URL `file:///Users/tjohnson/Desktop/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 Mashape, it's probably a good idea to compare this simple weather API with a more robust 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" and links to "Getting Started", "Reference", and "Downloads". The main content area has a title "Endpoint: observations". It describes the `observations` data set, mentioning METARS and PWS sources. Below this, there's a URL link <https://api.aerisapi.com/observations/>, a "Data Coverage" section showing "Global", and a "Included With" section listing "API Developer", "API Basic", and "API Premium". At the bottom, there are tabs for "ACTIONS", "PARAMETERS", "FILTERS", "QUERIES", "SORTING", "EXAMPLES", "RESPONSE", and "PROPERTIES".

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](#):

