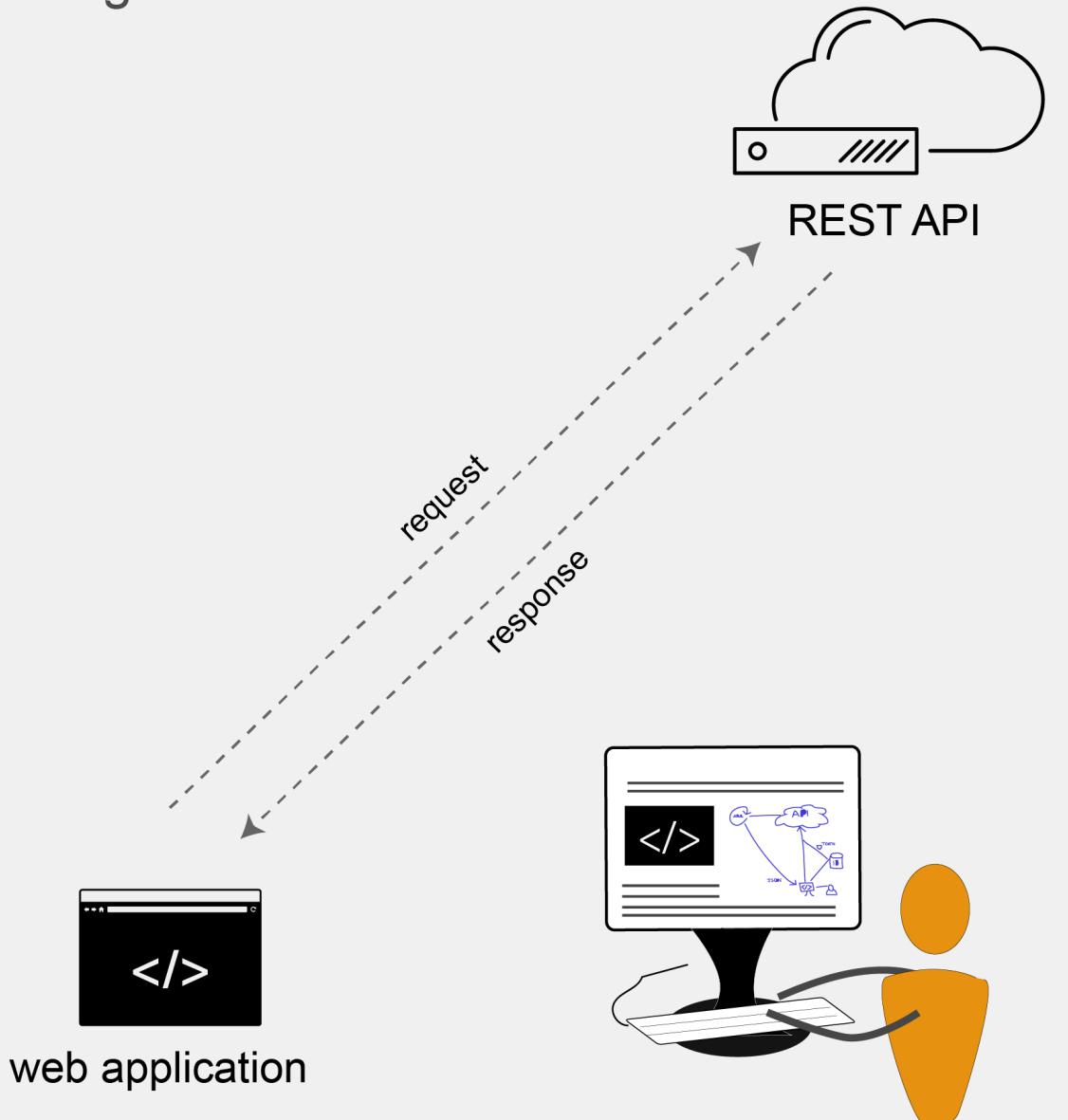




Documenting REST APIs

A guide for technical writers



by Tom Johnson

I'd Rather Be **Writing**

Last generated: April 14, 2017

© 2017 by Tom Johnson.

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	7
Course overview	8
Foreword	11
The market for REST API documentation.....	13
What is a REST API?.....	23
Using a REST API as a developer	31
Scenario for using a weather API.....	32
Getting authorization keys	37
Submit calls through Postman.....	41
Installing cURL.....	48
Making a cURL call.....	50
Understand cURL more	54
Using methods with cURL	59
Analyze the JSON response	65
Log JSON to the console.....	71
Access and print a specific JSON value.....	76
Diving into dot notation.....	80
Documenting a new API endpoint.....	86
New API endpoint to document.....	87
Documenting resource descriptions.....	91
Documenting endpoint definitions and methods.....	99
Documenting parameters	103
Documenting sample requests	111
Documenting sample responses	115
Documenting code samples	129
Putting it all together.....	140
Testing your documentation.....	145
Set up a test environment.....	146
Test all instructions yourself	150
Test your assumptions.....	155
Documenting non-reference sections.....	160
Creating user guide tasks	161
Writing the API overview	163
Writing the Getting Started section	165
Documenting authentication and authorization.....	169
Documenting response and error codes	176
Documenting code samples and tutorials	181
Creating the quick reference guide.....	185
Exploring other REST APIs	187

Exploring more REST APIs	188
EventBrite example.....	189
Flickr example.....	196
Klout example	205
Aeris Weather example	216
Next phase of course.....	223
Publishing API docs.....	224
Overview	225
List of about 100 APIs.....	228
Breaking down API doc	232
Tool decisions	236
Github wikis	242
More about Markdown	249
Using a version control system.....	257
Pull request workflows through Github in browser.....	266
REST API specification formats	272
Implementing Swagger with your API docs.....	273
Swagger tutorial	285
More about YAML.....	296
RAML	301
API Blueprint	313
Static site generators (Jekyll)	325
Readme.io.....	338
Miredot.....	342
Custom UX solutions	345
Help authoring tools.....	347
Tools versus Content	350
Design patterns.....	351
Design patterns.....	352
Structure and templates	354
Web platforms.....	356
Abundant code samples	358
Long-ish pages	360
API interactivity	363
Adding an Edit on Github button	365
Challenging factors	377
Documenting native library APIs.....	378
Overview of native library APIs	379
Getting the Java source.....	382
Java crash course.....	386
Generate a Javadoc.....	394
Javadoc tags.....	399
Exploring the Javadoc output.....	409

Making edits to Javadoc tags.....	412
Doxygen, another document generator.....	413
Documenting non-reference API docs	416
Course summary.....	418
Getting a Job in API Documentation	420
The job market for API technical writers.....	421
How much code do you need to know to create developer documentation?	427
Resources	431
Copyright	432

Introduction to REST APIs overview

In this section:

- [The market for REST API documentation \(page 13\)](#)
- [What is a REST API? \(page 23\)](#)
- [Foreward \(page 11\)](#)
- [Copyright \(page 432\)](#)

Documenting APIs: A guide for technical writers

In this book 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 more meaningful.

REST APIs involve requests and responses over HTTP protocol

After you use the 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
- Endpoint definitions and methods
- Parameters
- Sample requests
- Sample responses
- Error codes
- Code samples

Diving into these sections will give you a solid understanding of how to document REST APIs.

Finally, you'll dive into different ways to publish REST API documentation, exploring tools and specifications such as API Blueprint, Swagger, RAML, readme.io, Jekyll, and more.

You'll learn how to leverage templates, build interactive API consoles so users can try out requests and see responses, and learn different ways to host and publish your documentation.

Book organization

Organizationally, this book is divided into the following sections:

- **Introduction to REST APIs**

- **Using a REST API like a developer**
- **Documenting endpoints**
- **Testing your documentation**
- **Documenting non-reference content**
- **Exploring other REST APIs**
- **Publishing API documentation**
- **Design patterns**
- **Documenting native library APIs**

You don't have to read the chapters in order — skip around as you prefer. But some of the earlier sections on using a REST API like a developer and documenting endpoints follow a somewhat sequential order with the same weather API scenario.

Because the purpose of the book 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*.

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 book assumes you're a beginner, though.

Note that 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).

What you'll need

Here are a few things you'll need in this course:

- **Text editor.** ([Sublime Text](http://www.sublimetext.com/) (<http://www.sublimetext.com/>) is a good option (it works on both Mac and Windows), but any text editor will do. On Windows, [Notepad++](https://notepad-plus-plus.org/) (<https://notepad-plus-plus.org/>) and [Komodo Edit](http://komodoide.com/komodo-edit/) (<http://komodoide.com/komodo-edit/>) are also good.)
- **Chrome browser** (<http://www.google.com/chrome/>). (Other browsers are fine too, but we'll be using Chrome's Developer Console.)
- **Postman - REST Client (Chrome or Mac app)** (<http://www.getpostman.com/>).

Postman is an app that allows you to make requests and see responses through a GUI client. You can either use the Chrome or Mac app.

- **cURL (<http://curl.haxx.se/>)**. 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](http://www.confusedbycode.com/curl/#downloads) (<http://www.confusedbycode.com/curl/#downloads>).
- **Git (<https://git-scm.com/>)**. Git is a version control tool developers often use to collaborate on code. See [Set Up Git](https://help.github.com/articles/set-up-git/) (<https://help.github.com/articles/set-up-git/>) for more details.

Stay updated

If you're taking 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/tomjohnson1492> (<https://tinyletter.com/tomjohnson1492>).

Foreward

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 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 writer said \(<http://idratherbewriting.com/learnapidoc/index.html#comment-3124829110>\):](#)

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 said \(<https://disqus.com/by/helengriffith/>\):](#)

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.

Another said (<http://idratherbewriting.com/learnapidoc/index.html#comment-2570927595>):

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!)

Other readers noted problems in obtaining API keys or getting the correct responses. As much as possible, I helped clarify and strengthen the content on those pages.

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 book 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.

I wrote this book in kramdown Markdown, using Jekyll. I single sourced the content into several outputs: Web, Kindle (Mobi), EPUB, and PDF. When you buy the book, you can access any version at any time. However, to provide for discoverability online, I decided to leave the web version free.

I welcome feedback. Feel free to drop me a note at tom@idratherbewriting.com. And follow my blog at <http://idratherbewriting.com>.

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](https://ffeathers.wordpress.com/2014/02/16/api-types/) (<https://ffeathers.wordpress.com/2014/02/16/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	OS functions and routines
SOAP	Access to file system
XML-RPC and JSON-RPC	Access to user interface
REST	
WebSocket APIs	Object remoting APIs
	CORBA
	.NET Remoting
Library-based APIs	Hardware APIs
JavaScript	Video acceleration
TWAIN	Hard disk drives
Class-based APIs (object orientation)	PCI buses
Java API	
Android API	

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.

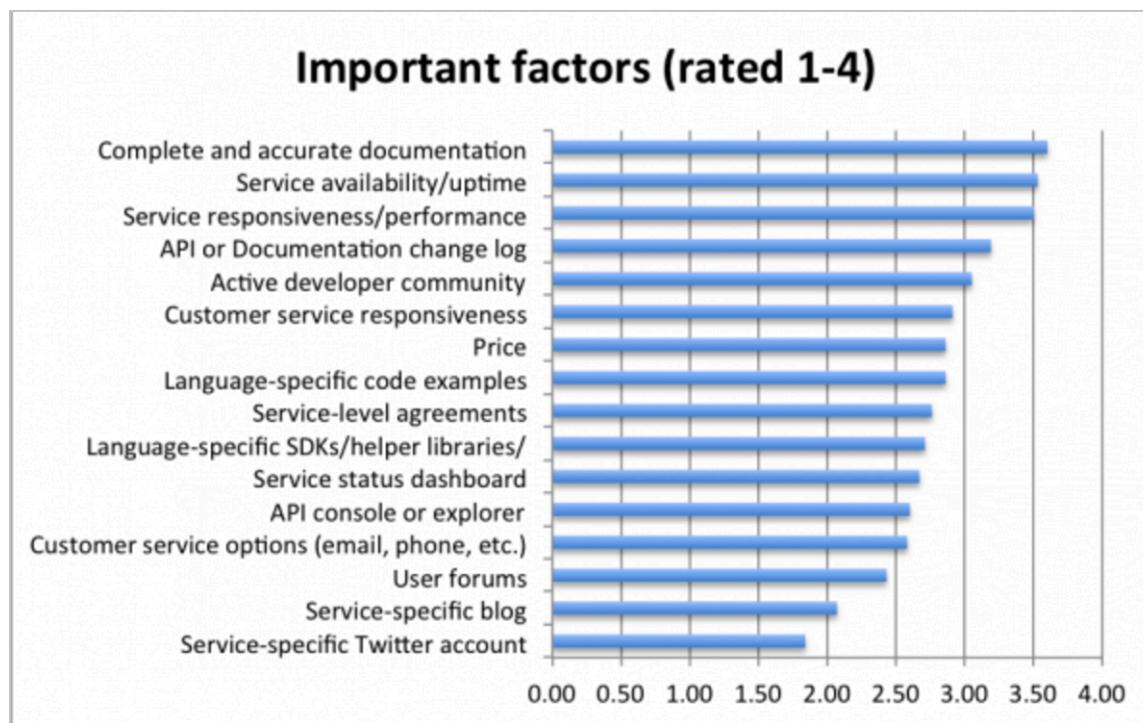
REST APIs follow the same protocol as the web. When you open a browser and type a website URL (such as <http://idratherbewriting.com>), you're actually making a GET request for a resource on a server. The server responds with the content and the browser makes the content visible.

This course focuses mostly on REST APIs because they're 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](http://www.programmableweb.com/news/api-consumers-want-reliability-documentation-and-community/2013/01/07) (<http://www.programmableweb.com/news/api-consumers-want-reliability-documentation-and-community/2013/01/07>) (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.



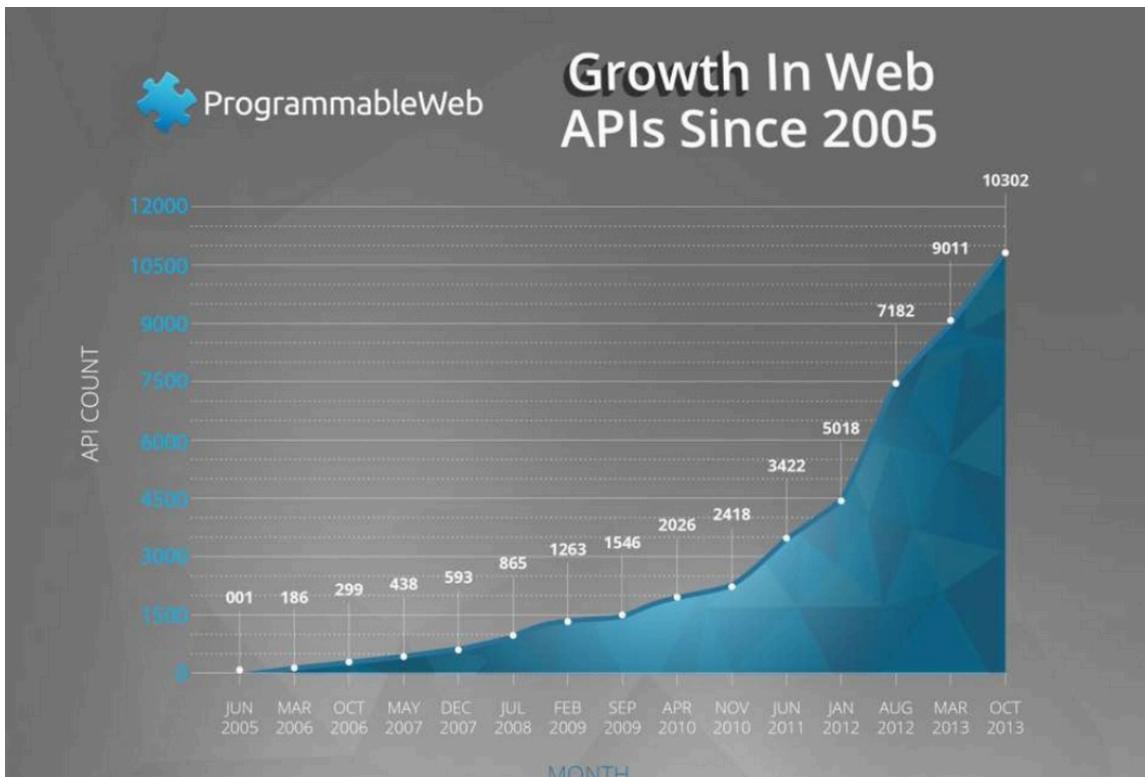
John Musser, one of the founders of Programmableweb.com, emphasizes the importance of documentation in his presentations. In “10 reasons why developers hate your API,” he says the number one reason developers hate your API is because “Your documentation sucks.”

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](https://swiftype.com/developers) (<https://swiftype.com/developers>). Rather than building your own payment gateway, you might integrate [Stripe and its API](https://stripe.com/docs/api) (<https://stripe.com/docs/api>). Rather than building your own login system, you might use [UserApp and its API](https://app.userapp.io/#/docs) (<https://app.userapp.io/#/docs>). Rather than building your own e-commerce system, you might use [Snipcart and its API](http://docs.snipcart.com/api-reference/introduction) (<http://docs.snipcart.com/api-reference/introduction>). 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](http://cloudcannon.com/tips/2014/12/12/the-ultimate-list-of-services-for-static-websites.html) (<http://cloudcannon.com/tips/2014/12/12/the-ultimate-list-of-services-for-static-websites.html>) that you can integrate into your static site.

The screenshot shows a dark-themed interface for managing services. At the top, the word "Services" is displayed in large white letters. Below it, a sub-section titled "Newsletters" is shown with the subtext "Capture email addresses and send periodic newsletters." Five service icons are listed: AWeber (blue square with white sound waves), Campaign Monitor (white square with blue envelope icon), MailChimp (white square with brown monkey head icon), MailerLite (white square with green "lite" text), and Sendicate (orange square with white stylized "S" and wavy lines). At the bottom, there is another section titled "Analytics".

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

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

Job market is hot for API technical writers

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

Find Jobs	Find Resumes	Employers / Post Job
 what: <input type="text"/> job title, keywords or company where: <input type="text"/> city, state, or zip		
Contract API Tech Writer, Palo Alto Synergistech - Palo Alto, CA Principals only, please This stealth-mode software startup needs a Contract Technical Writer with strong software development skills to create conceptual and reference content - including working code samples - for their persistent cloud storage system. You'll need enough software industry and engineering experience to help define and improve the products, and the ability to write modern copy-paste-tweak-and-run code examples to support APIs in Objective C, Java, REST, and C. The client wants to find someone who'll emulate Dropbox's developer documentation (for example, https://www.dropbox.com/developers/sync/start/android) or similar. If you've participated actively in API development cycles, providing feedback on the APIs themselves, and can show samples of developer tutorials and, ideally, dynamic websites, this company wants to meet you. In this role, you'll need to work onsite in Palo Alto at least a couple days/week throughout the project. You can work corp-to-corp, as a 1099-based independent contractor, or as a W2 temporary employee for as long as mutually agreed. The project has no fixed term, and is renewable in three (3) month increments. Required : Strong code reading and sample-code writing skills in one or more of these languages (Objective C, Java, C) or the REST protocol Experience providing feedback on APIs during development cycles Showable portfolio samples that include cut-and-pasteable code samples		

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

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

Here's what the Dropbox API looks like:

The screenshot shows the Dropbox API v2 documentation website. At the top right, there's a user profile for 'Tom Johnson' with a dropdown arrow. To the left of the profile is a blue bell icon. The main title 'Build your app on the Dropbox platform' is centered above a subtext 'A powerful API for apps that work with files.' On the left side, there's a sidebar with links: 'API v2', 'My apps', 'API Explorer', 'Documentation', 'Community SDKs', 'References', 'Authentication types', 'Branding guide', 'Content hash', and 'Data ingress guide'. The 'Documentation' section is expanded, showing links for various programming languages: 'HTTP', '.NET', 'Java', 'JavaScript', 'Python', 'Swift', 'Objective-C', and 'Community SDKs'. Below the sidebar, there's a central area with three boxes: 'Read our docs' (with subtext 'Docs are organized by language, from .NET to Swift.'), 'Create your app' (with subtext 'Getting started is simple and quick from the App Console.'), and 'Test your ideas' (with subtext 'It's easy to prototype and test examples with our API Explorer.'). Above these boxes is a cartoon illustration of four people working on computers, with gears and clouds around them.

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> (<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 book 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.

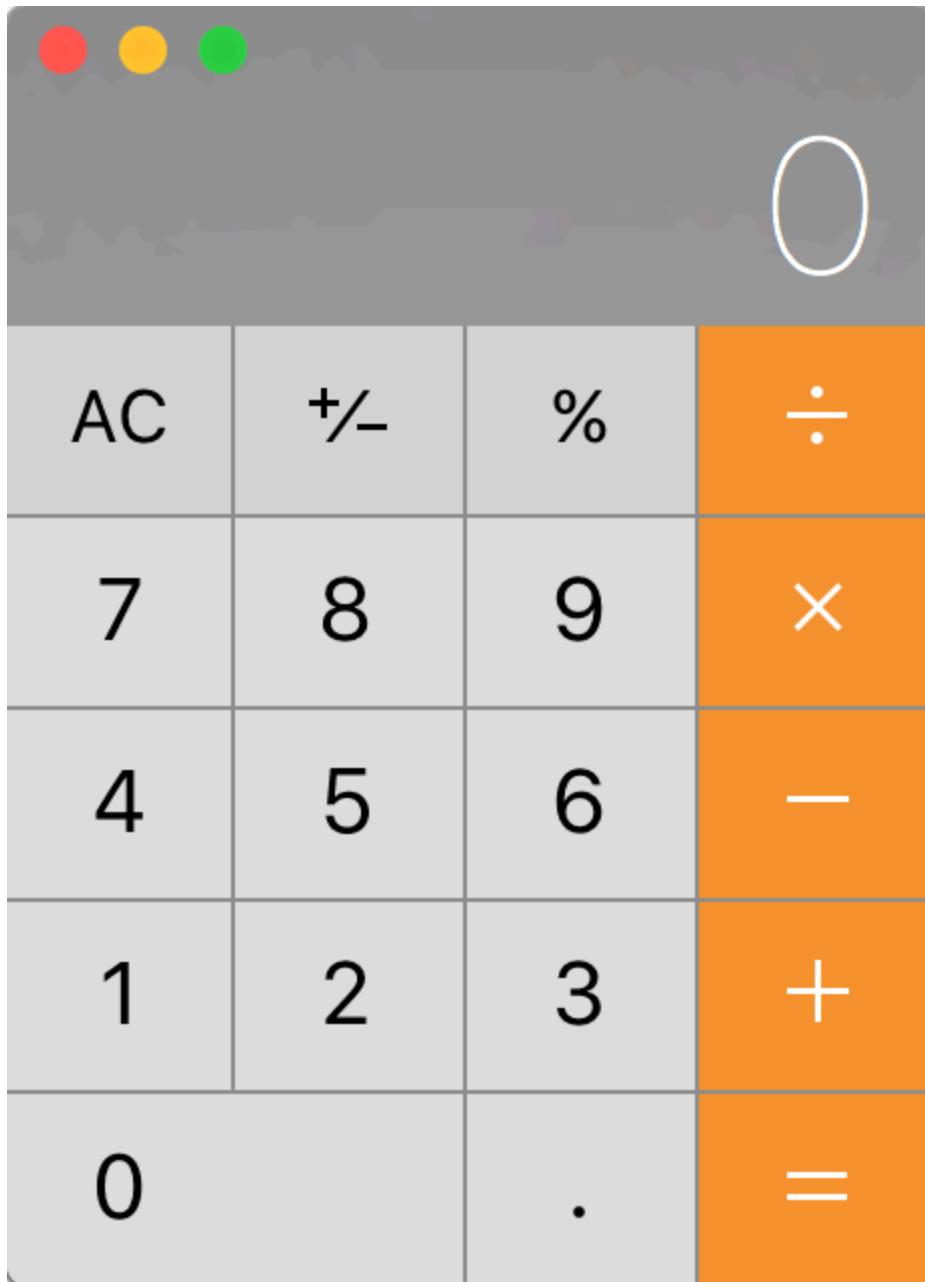
What is an API?

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 \(<http://www.soapui.org/testing-dojo/world-of-api-testing/soap-vs--rest-challenges.html>\)](http://www.soapui.org/testing-dojo/world-of-api-testing/soap-vs--rest-challenges.html).

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> (<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](#) (https://en.wikipedia.org/wiki/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 chapters 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.

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](http://www.restapitutorial.com/lessons/whatisrest.html) (<http://www.restapitutorial.com/lessons/whatisrest.html>). 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 \(page 285\)](#) or [RAML \(page 301\)](#) 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.

Using an API like a developer

In this section:

- Scenario for using a weather API (page 32)
- Getting authorization keys (page 37)
- Submit requests through Postman (page 41)
- cURL intro and installation (page 48)
- Make a cURL call (page 50)
- Understand cURL more (page 54)
- Using methods with cURL (Petstore example) (page 59)
- Analyze the JSON response (page 65)
- Using the JSON from the response payload (page 71)
- Access and print a specific JSON value (page 76)
- Diving into dot notation (page 80)

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.

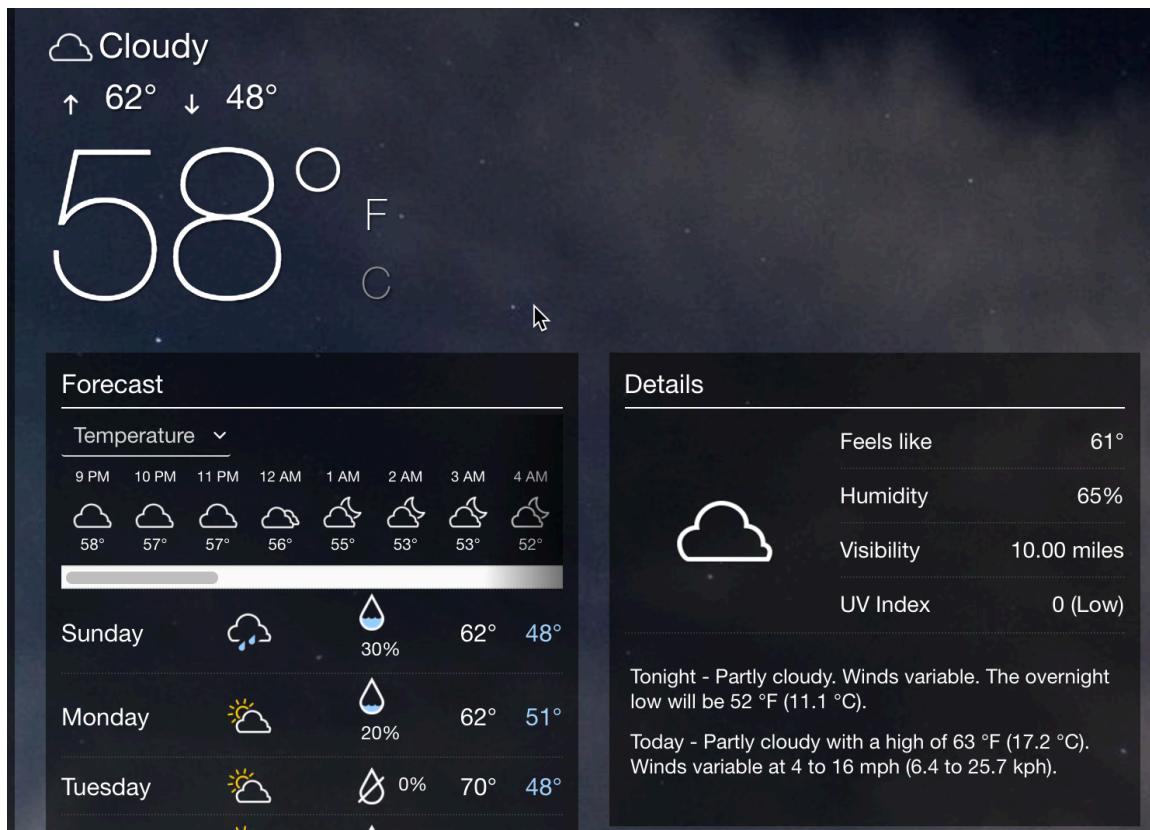
Get an idea of the end goal

To give you an idea of the end goal, here's a sample. It's not necessarily styled the same as the mockup, but it answers the question, "How windy is it?"

Go to this URL: [\(http://idratherbewriting.com/files/restapicourse/wind-mashape.html\)](http://idratherbewriting.com/files/restapicourse/wind-mashape.html).

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.

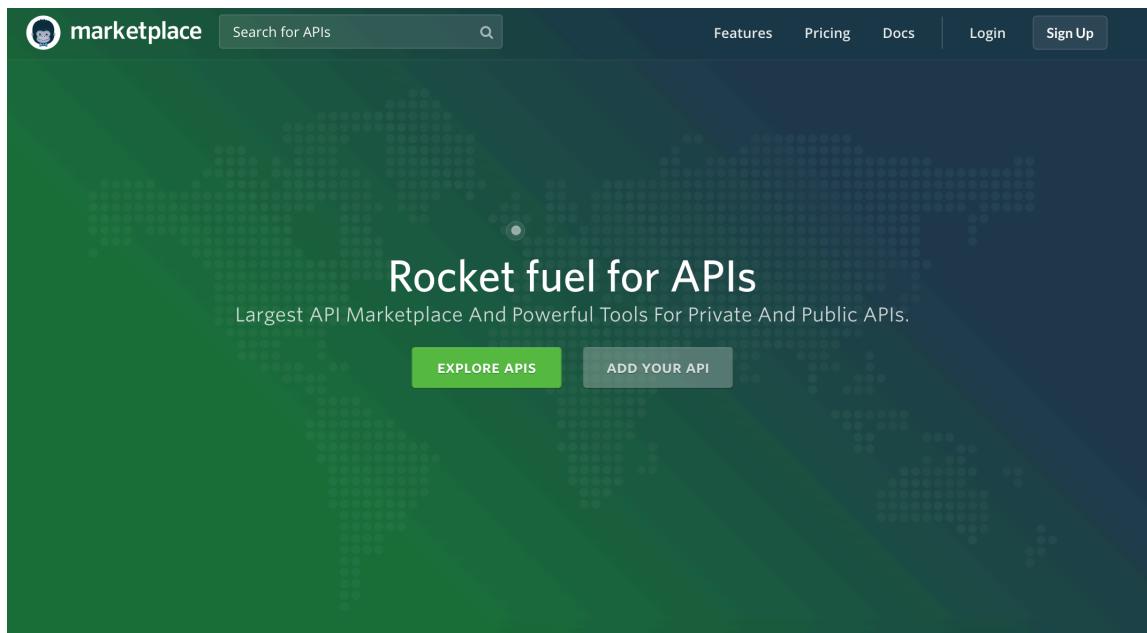
The above example is extremely simple. You could also build an attractive interface like this:



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?

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

1. Go to [Mashape Marketplace](https://market.mashape.com/) (<https://market.mashape.com/>) 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> (<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.

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 \(<http://www.aerisweather.com/>\)](http://www.aerisweather.com/) 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.

1. Go to [www.aerisweather.com \(<http://www.aerisweather.com>\).](http://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 website. At the top, there is a navigation bar with links for APPS, DEVELOPER, INDUSTRIES, SUPPORT, and CASE STUDIES. Below the navigation bar, the URL path is shown as HELP CENTER / DOCS / AERIS WEATHER API / REFERENCE / ENDPOINTS. The main content area has a sidebar on the left with sections for Aeris Weather API, Getting Started, Reference, and Downloads. The main content area is titled 'API Endpoints'. It contains a brief description of what endpoints are and lists supported endpoints. A callout box points to a note about 'Batch Requests'. Below this, there is a table with columns for Endpoint and Description, showing details for the 'advisories' endpoint, including data coverage and included APIs. There is also a note about the 'advisories/summary' endpoint.

Endpoint	Description
advisories	<p>The advisories data set provides access to all currently active US advisories as issued National Weather Service (NWS). The NWS issues a variety of severe weather warning advisories and statements that may be issued for a single forecast zone or county, or region.</p> <p>Data Coverage: Continental US, Alaska, Hawaii Included With: API Developer, API Basic, API Premium</p>
advisories/summary	<p>The advisories/summary endpoint provides an overall summary of the currently active advisories within the USA based on search/filter criteria. Example uses include obtaining the total count of active advisories, or filtering by state or county.</p>

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: <http://idratherbewriting.com/files/restapicourse/wind-aeris.html> (<http://idratherbewriting.com/files/restapicourse/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

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.

Getting 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 \(page 37\)](#)
- [Get the Aeris Weather API secret and ID \(page 38\)](#)
- [Text editor tips \(page 39\)](#)

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 \(https://market.mashape.com/\)](https://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 Kong API Gateway dashboard. At the top, there's a navigation bar with 'Search APIs', 'Explore APIs', 'Docs', 'Features', 'Applications' (with a count of 3), 'My APIs' (with a count of 0), and a user profile for 'tomjohnson1492'. Below the navigation is a banner with links to 'Kong', 'Galileo', and 'Gelato'. The main area displays four API cards: 'Simple Weather' (by 'Ser Programadores', mxrck), 'Weather' (by 'fyhao', icon showing 73), 'Indeed' (by 'indeed', icon showing one search result), and 'FOAAS' (by 'community', icon showing a bar chart). Each card has a 'No Data Available' message below it. In the top right corner of the main area, there's a blue button labeled 'GET THE KEYS'.

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.

- 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.)

The screenshot shows a modal dialog titled 'ENVIRONMENT KEYS'. Inside, there's a message: '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.' Below the message are two buttons: a dropdown menu set to 'Testing' and a 'COPY' button in blue, which is highlighted with a red box. To the right of the 'COPY' button is a 'REGENERATE' button in red.

- 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.

- Go to <http://www.aerisweather.com> (<http://www.aerisweather.com>) and click **Sign Up** in the upper-right corner.
- Under **Developer**, click **TRY FOR FREE**. (Note that the free version limits the number of requests per day and per minute you can make.)
- 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.

The screenshot shows the Aeris Weather API registration interface. At the top, there's a navigation bar with links for ACCOUNT, LOG OUT, and SIGN UP. Below that is a secondary navigation bar with links for DASHBOARD, USAGE, APPS, MAP BUILDER, LEGEND GENERATOR, ADD SUBSCRIPTION, DOCS, SUPPORT, and BILLING. The APPS link is underlined, indicating it's the active section. A blue header bar contains the Aeris Weather logo and a search bar. The main content area is titled "Registered Apps". It displays a table with one row for an application named "API testing". The table columns include ID, SECRET, and a link to the application's details. To the right of the table is a "NEW APPLICATION" button, which is highlighted with a red arrow pointing to it from the left.

ID:	SECRET:	Actions
ByruDorHEne2JB64BhP1k	uBDNO535gYHULH8mqx3skZmU13EV4nlf4GvB6jhY	

<https://www.aerisweather.com/account/apps#>

5. Click **Apps** (on the second navigation row, to the right of “Usage”), and then click **New Application**.
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 (<http://www.sublimetext.com/>) (Mac or PC)
- TextWrangler (<http://www.barebones.com/products/textwrangler/>) or BBedit (<http://www.barebones.com/products/bbedit/>) (Mac)
- WebStorm (<https://www.jetbrains.com/webstorm/>) (Mac or PC)
- Notepad++ (<https://notepad-plus-plus.org/>) (PC)
- Atom (<https://atom.io/>) (Mac or Windows)
- Komodo Edit (<http://komodoide.com/komodo-edit/>) (Mac or PC)
- Coda (<https://panic.com/coda/>) (Mac)

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

Submit requests through Postman

When you're testing endpoints with different parameters, you can use one of the many GUI REST clients available. 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](http://www.getpostman.com/) (<http://www.getpostman.com/>)
- Advanced REST Client (<https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbjeloo>) (Chrome browser extension)
- REST Console (<https://chrome.google.com/webstore/detail/rest-console/cokgbifommojglbmbpenphppikmnon>)
- Paw (<https://luckymarmot.com/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.

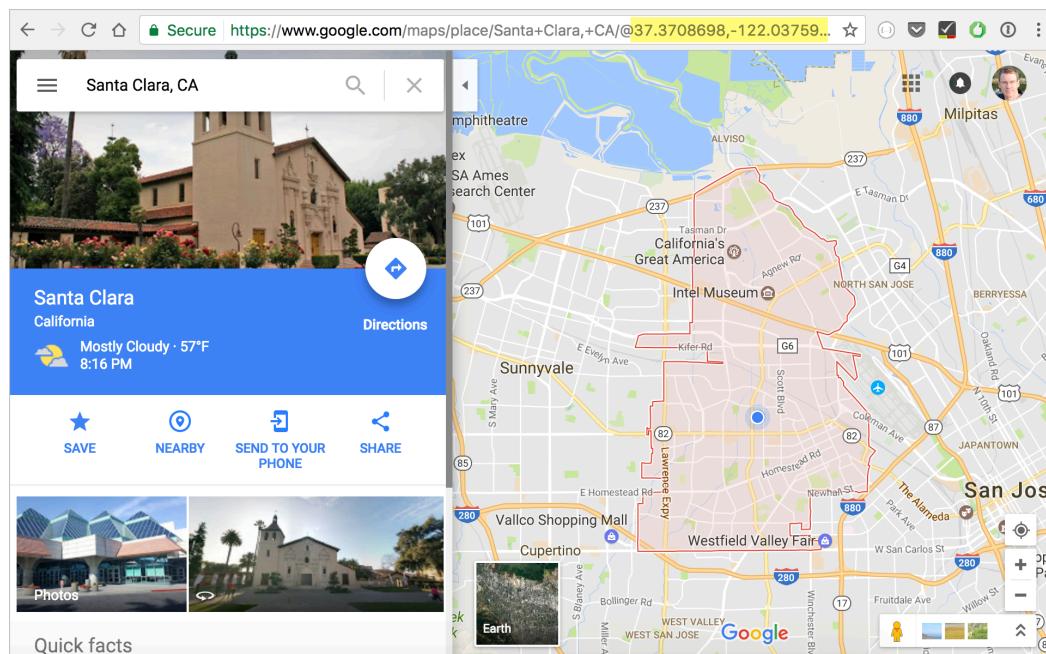
A lot of times abstract concepts don't make sense until you can contextualize them with some kind of action. In this course, I'm following more of an act-first-then-understand type of methodology. After you do an activity, we'll explore the concepts in more depth. So if it seems like I'm glossing over 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> (<https://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 first endpoint ([aqi](#)) in the Mashape Weather API. Select **GET** for the method.
4. Insert the endpoint into the main box (next to the method, which is GET by default):
<https://simple-weather.p.mashape.com/aqi>

- 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 the `aqi` call — 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 AQI for Santa Clara, California is `lat: 37.3710062` and `lng: -122.0375935`. For Singapore, it's `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.

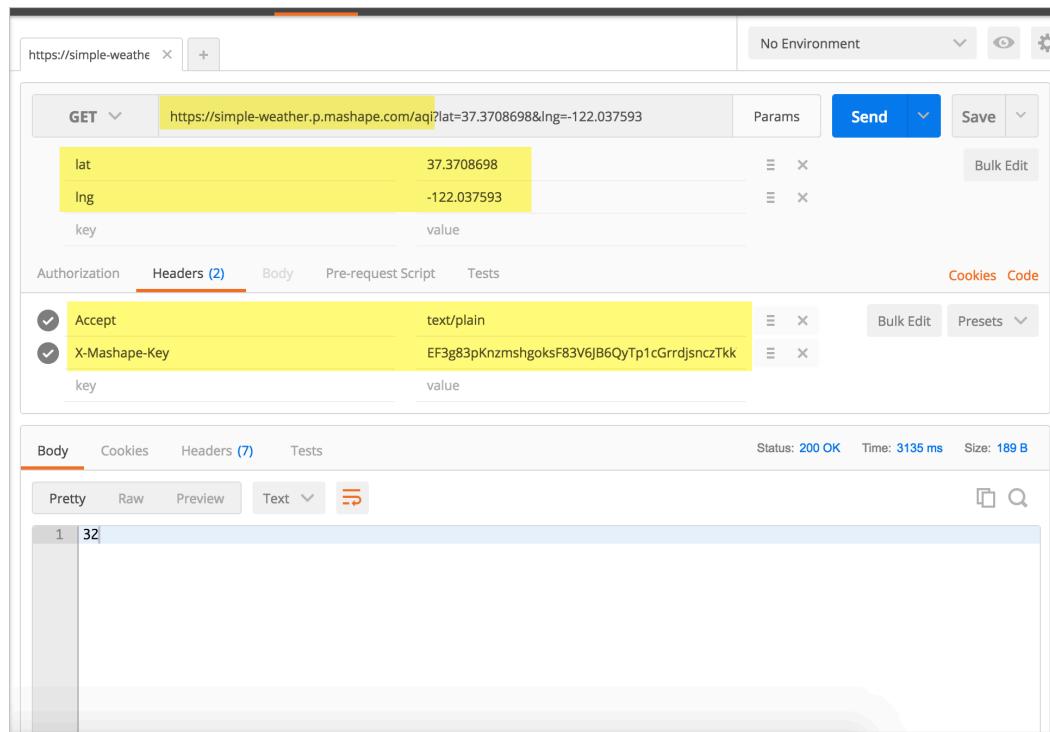


[Latitude is listed first, then longitude](#)

</figure> 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/aqi?lat=37.3710062&lng=-122.0375935`. Query string parameters appear after the question mark `?` symbol and are separated ampersands `&`.

- 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:



7. 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

1. In Postman, click the **Save** button (next to Send).
2. 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.
3. In the **Request Name** box at the top of the dialog box, type a friendly name for the request, such as "Mashape AQI endpoint."
4. Click **Save**.

Saved endpoints appear in the left side pane under Collections.

Make requests for the other endpoints

Enter details into Postman for the other two endpoints for the Mashape Weather API:

- weather

- 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.)

The screenshot shows the Postman Builder interface. At the top, there are tabs for 'Builder' (which is selected), 'Team Library', and various status indicators like 'SYNC OFF' and user info ('Tom John...'). Below the tabs, there's a dropdown for 'Environment' set to 'No Environment'. The main area shows a request URL: 'weather.p.mashape.com/aqi?lat=37.3708698&lng=-122.037593'. To the right of the URL are 'Params', 'Send', and 'Save' buttons. A large orange arrow points from the text above to the 'Save' button, specifically to the dropdown menu labeled 'Save As...'. Below the URL, there are input fields for parameters: '37.3708698' and '-122.037593', each with a delete icon. Underneath these are 'value' labels. At the bottom of the request section, there are tabs for 'Body', 'Pre-request Script', 'Tests', 'Cookies', and 'Code'. The 'Body' tab is selected. In the body section, there are two rows: 'text/plain' and 'EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkk', each with a delete icon. A 'value' label is also present. To the right of the body section are 'Bulk Edit' and 'Presets' buttons.

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

```

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:

Headers		URL Params	Body	Options	Info		Request	Response	Headers	JSON	Raw
Header Name	Header Value				Key or Index	Type	Value				
<input checked="" type="checkbox"/> X-Mashape-Key	EF3gB3pKnzmshgoksF83V6JB6QyTp1...				Root		1 item				
<input checked="" type="checkbox"/> Accept	text/plain				query		4 items				
		Add Header Name	Add Header Value		count		1				
					created		A 2015-06-03T16:24:26Z				
					lang		A en-US				
					results		1 item				
					channel		13 items				
					title		A Yahoo! Weather - Santa Clara, CA				
					link		A http://us.rd.yahoo.com/dailynews/rss...				
					description		A Yahoo! Weather for Santa Clara, CA				
					language		A en-us				
					lastBuildDate		A Wed, 03 Jun 2015 8:52 am PDT				
					ttl		A 60				
					location		3 items				
					units		4 items				
					wind		3 items				
					atmosphere		4 items				
					astronomy		2 items				
					image		5 items				
					item		9 items				

```

1 GET /weatherdata?lat=37.354108&lng=-121.955236 HTTP/1.1
2 X-Mashape-Key: EF3gB3pKnzmshgoksF83V6JB6QyTp1...
3 Accept: text/plain
4 Host: simple-weather.p.mashape.com
5 Connection: close
6 User-Agent: Paw/2.2.1 (Macintosh; OS X/10.10.3) GCDHTTPRequest
7
8

```

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 SecureTrans
port zlib/1.2.8
Protocols: dict file ftp ftps gopher http https imap imaps ldap
ldaps pop3 pop3s rtsp smb smbs smtp smtps telnet tftp
Features: AsynchDNS IPv6 Largefile GSS-API Kerberos SPNEGO NTLM
NTLM_WB SSL libz UnixSockets
```

If you don't see this, you need to [download and install cURL](http://curl.haxx.se/) (<http://curl.haxx.se/>).

To make a test API call, submit the following:

```
curl --get -k --include "https://simple-weather.p.mashape.com/aqi?lat=1.3319164&lon=103.7231246" -H "X-Mashape-Key: EF3g83pKnzmshgoksF83V6
JB6QyTp1cGrrdjsnczTkkYgYrp8p" -H "Accept: text/plain"
```

You should get back a two-digit number in the response, such as 63. (This is the "air quality index" for the weather.)

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](http://www.confusedbycode.com/curl/#downloads) (<http://www.confusedbycode.com/curl/#downloads>).

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 z
lib/1.2.5
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps
pop3 pop3s 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/aqi?la
t=1.3319164&lng=103.7231246" -H "X-Mashape-Key: EF3g83pKnzmshgoksF83V6
JB6QyTp1cGrrdjsnczTkkYgYrp8p" -H "Accept: text/plain"
```

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

You should get back a two-digit number in the response. (This is the “air quality index” for the weather.)

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](http://www.faqforge.com/windows/fix-the-program-can-t-start-because-msvcr100-dll-is-missing-from-your-computer-error-on-windows/) (<http://www.faqforge.com/windows/fix-the-program-can-t-start-because-msvcr100-dll-is-missing-from-your-computer-error-on-windows/>) 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 \(page 48\)](#) 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 (<https://www.mashape.com/fyhao/weather-13>).
2. Copy the cURL request example for the first endpoint (aqi) into your text editor:

```
curl --get --include 'https://simple-weather.p.mashape.com/aq
i?lat=1.0&lng=1.0' \
-H 'X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkk
YgYrp8p' \
-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/aq
i?lat=1.0&lng=1.0" -H "X-Mashape-Key: APIKEY" -H "Accept: text/p
lain"
```

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 part with your own API key.

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](https://www.iterm2.com/) (<https://www.iterm2.com/>) 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:

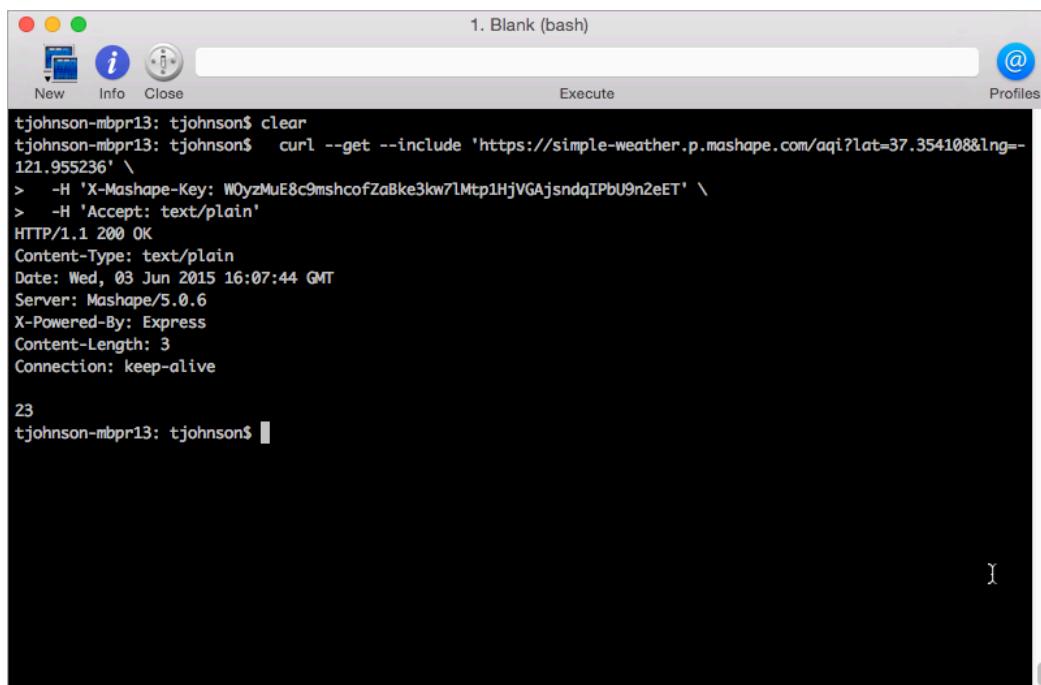
```
curl --get --include 'https://simple-weather.p.mashape.com/aqi?lat=1.3319164&lng=103.7231246' -H 'X-Mashape-Key: APIKEY' -H 'Accept: text/plain'
```

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

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

3. Press your **Enter** key.

You should see something like this as a response:



The screenshot shows a terminal window titled '1. Blank (bash)'. The window has standard OS X window controls (red, yellow, green buttons) and a toolbar with 'New', 'Info', 'Close', 'Execute', and a profile icon. The terminal output is as follows:

```
tjohnson-mbpr13: tjohnson$ clear
tjohnson-mbpr13: tjohnson$ curl --get --include 'https://simple-weather.p.mashape.com/aqi?lat=37.354108&lng=-121.955236' \
> -H 'X-Mashape-Key: W0yzMuE8c9mshcofZaBke3kw7lMtp1HjVGAjsndqIPbU9n2eET' \
> -H 'Accept: text/plain'
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Wed, 03 Jun 2015 16:07:44 GMT
Server: Mashape/5.0.6
X-Powered-By: Express
Content-Length: 3
Connection: keep-alive

23
tjohnson-mbpr13: tjohnson$
```

The response is just a single number: the air quality index for the location specified. (This response is just text, but most of the time responses from REST APIs are in JSON.)

Make the request in cURL (Windows 7)

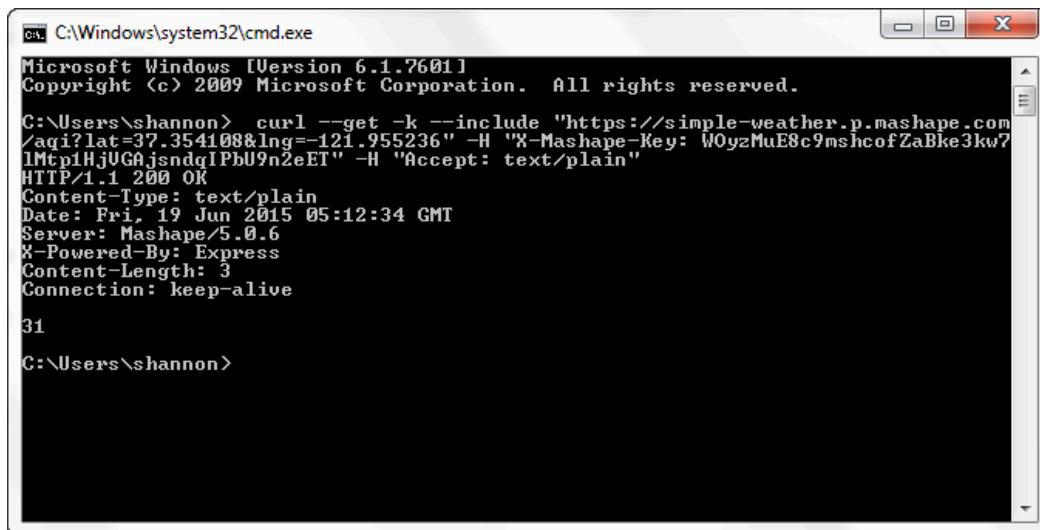
1. Copy the cURL call from your text editor.
2. Go to **Start** and type **cmd** to open up the commandline. (If you're on Windows 8, see [these instructions for accessing the commandline](http://pcsupport.about.com/od/windows-8/a/command-prompt-windows-8.htm) (<http://pcsupport.about.com/od/windows-8/a/command-prompt-windows-8.htm>)).
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/
aqi?lat=1.3319164&lng=103.7231246" -H "X-Mashape-Key: APIKEY"
-H "Accept: text/plain"
```

For the Aeris endpoint, it looks like this:

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

The response from Mashape looks like this:



A screenshot of a Windows cmd.exe window. The title bar says 'C:\Windows\system32\cmd.exe'. The window displays the following command and its output:

```
C:\> C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\> curl --get -k --include "https://simple-weather.p.mashape.com/
aqi?lat=37.354108&lng=-121.955236" -H "X-Mashape-Key: W0yzMuE8c9mshcofZaBke3kw7
1MtpiHjUGAjsndqIPbU9nZeET" -H "Accept: text/plain"
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Fri, 19 Jun 2015 05:12:34 GMT
Server: Mashape/5.0.6
X-Powered-By: Express
Content-Length: 3
Connection: keep-alive

31
C:\>
```

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 \(http://example.com\)](http://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.

3. 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/aqi?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
--------------	-------------

- | | |
|---|------------------------|
| 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>--get</code> |
| Whether to show the headers in the response. Also represented by <code>-i</code> . | <code>--include</code> |
| Submits a custom header. Include an additional <code>-H</code> for each header key-value pair you're submitting. | <code>-H</code> |

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</code> <code>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 (http://www.w3schools.com/tags/ref_urlencode.asp). Data can also be passed in the request body.	<code>curl -d "data-to-post"</code> <code>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"</code> <code>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"</code> <code>http://www.example.com</code>
<code>@filename</code>	Load content from a file	<code>curl -X POST -d @mypet.json</code> <code>http://www.example.com</code>

See the [cURL documentation](http://curl.haxx.se/docs/manpage.html) (<http://curl.haxx.se/docs/manpage.html>) 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 instructs the server to process the post body as JSON.

Summary

As a summary, keep the following in mind:

- `-i` means to include the response header
- `-H` means to pass a header into the request
- `-X POST` means to use the `POST` method in the request
- `-d` means to include data in the request

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](http://blogs.plexibus.com/2009/01/15/rest-esting-with-curl/) (<http://blogs.plexibus.com/2009/01/15/rest-esting-with-curl/>).

Using methods with cURL (Petstore example)

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](http://petstore.swagger.io/) (<http://petstore.swagger.io/>), 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

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

[\(http://petstore.swagger.io/\)](http://petstore.swagger.io/)

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. D

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 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](http://www.restapitutorial.com/lessons/idempotency.html) (<http://www.restapitutorial.com/lessons/idempotency.html>). 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](http://jsonprettyprint.com/) (<http://jsonprettyprint.com/>):

```
{  
    "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.

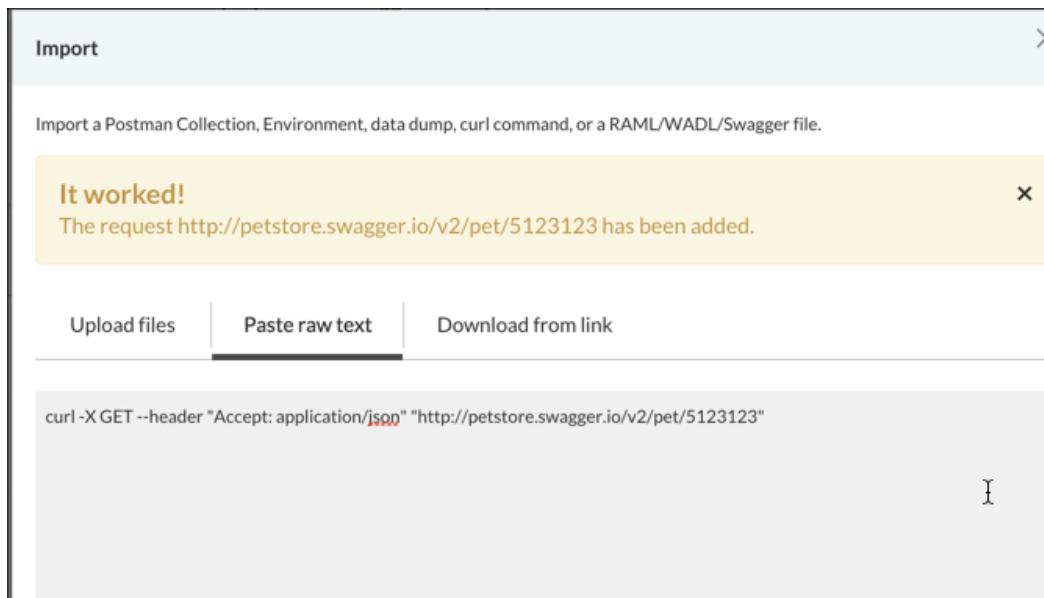
Although Postman is probably easier to use, cURL lends itself to power-level usage. Quality assurance teams often construct advanced test scenarios that iterate through a lot of cURL requests.

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"
```



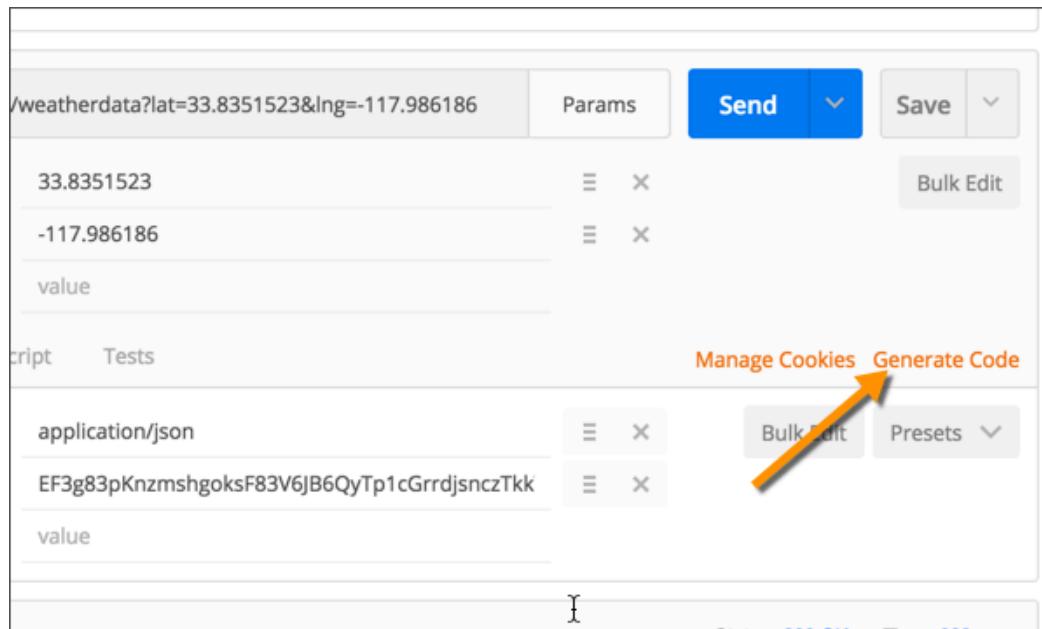
Make sure you don't have any extra spaces at the beginning.

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\"/>\n<br />\n<b>Current Conditions:</b>\n<br />\nPartly Cloudy, 16 C\n<B R />\n<BR />\n<b>Forecast:</b>\n<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>\n<BR />\n<BR />\n(Provided by <a href=\"http://www.weather.com\">The Weather Channel</a>)\n<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](#) (<http://jsonformatter.curiousconcept.com/>) 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/Sant  
a_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/ne  
ws-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><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<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"
                }
            ]
        }
    }
```

```
        "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"
}
}
}
```

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 set off with curly braces `{ }` and contain key-value pairs.

Sometimes those values are arrays. Arrays are lists and are set off 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

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](https://www.json.com/) (<https://www.json.com/>).

Using 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](http://docs.mashape.com/javascript) (<http://docs.mashape.com/javascript>) to parse and display the REST response on a web page using JavaScript. You could use it, but you could also use some auto-generated code from Postman to do pretty much the same thing.

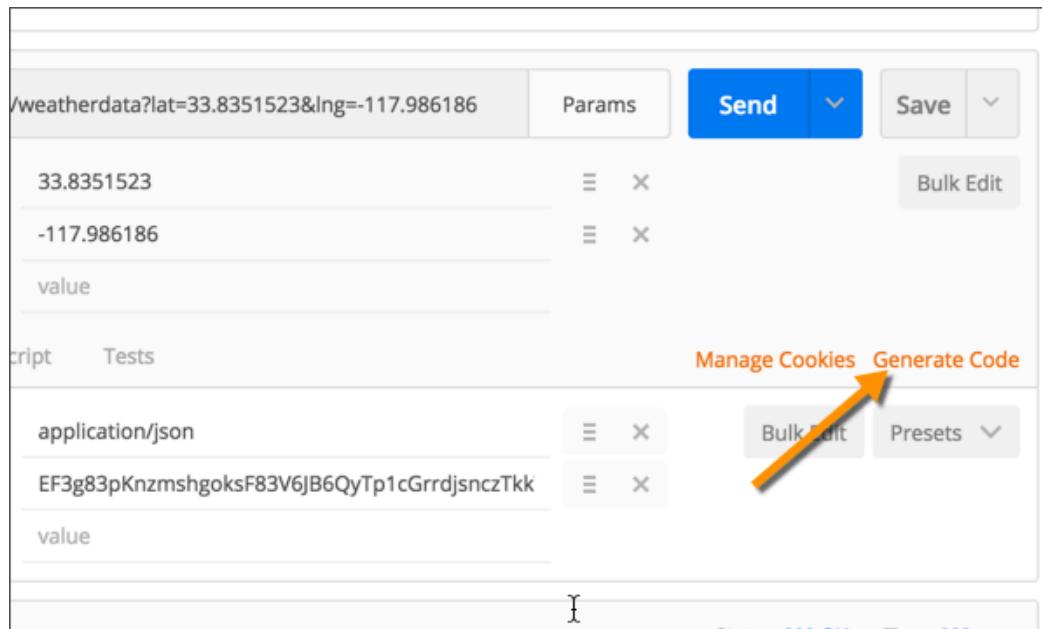
1. Start with a basic HTML template with jQuery referenced, like this:

```
<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 **Generate Code Snippet** button.



3. Select **JavaScript > jQuery AJAX**.
4. Copy the code sample.
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`.

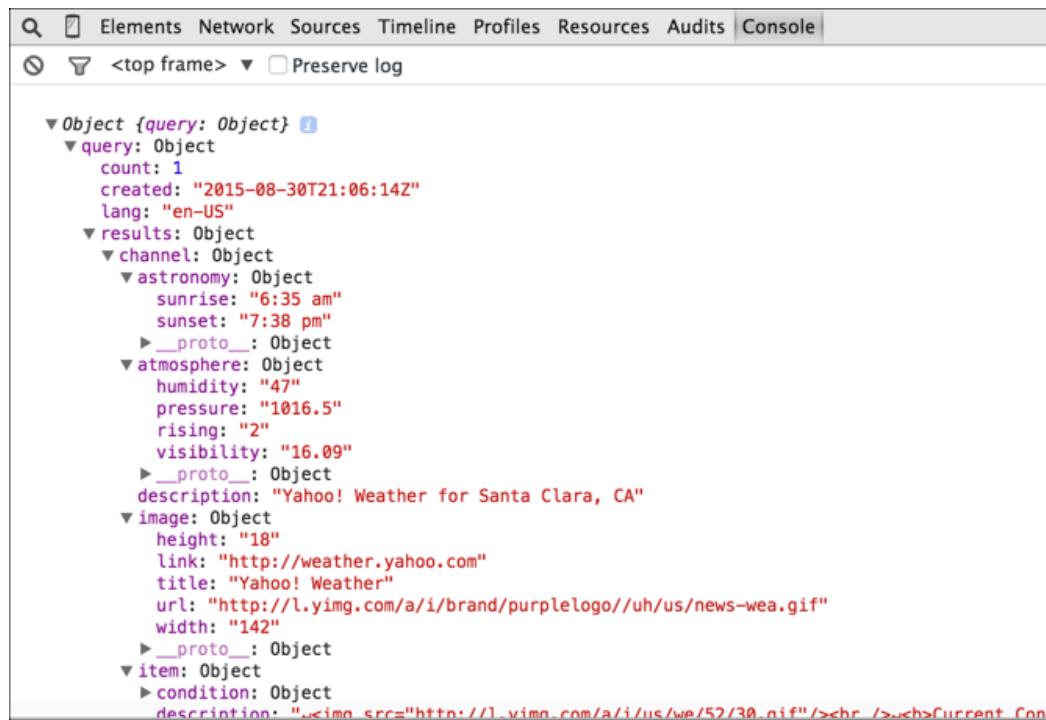
Make sure you capitalize the `T` in `dataType`.

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,
    "dataType": "json",
    "url": "https://simple-weather.p.mashape.com/weatherdata?lat=3
7.354108&lng=-121.955236",
    "method": "GET",
    "headers": {
        "accept": "application/json",
        "x-mashape-key": "APIKEY"
    }
}
$.ajax(settings).done(function (response) {
    console.log(response);
});
</script>
<body>
</body>
</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 DevTools console with the "Console" tab selected. It displays a hierarchical JSON object structure. The root object is an object with a single child named "query". The "query" object has several properties: "count" (1), "created" ("2015-08-30T21:06:14Z"), and "lang" ("en-US"). It also contains a nested object "results" which itself has a nested object "channel". The "channel" object contains an "astronomy" object with "sunrise" ("6:35 am") and "sunset" ("7:38 pm"). There are also "atmosphere" and "image" objects, and a "item" object which further contains a "condition" object.

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](http://api.jquery.com/jquery.ajax/#jQuery-ajax-settings) (<http://api.jquery.com/jquery.ajax/#jQuery-ajax-settings>).

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:

```
data.query.results.channel.item.description
```

The dot (`.`) after `data` (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.

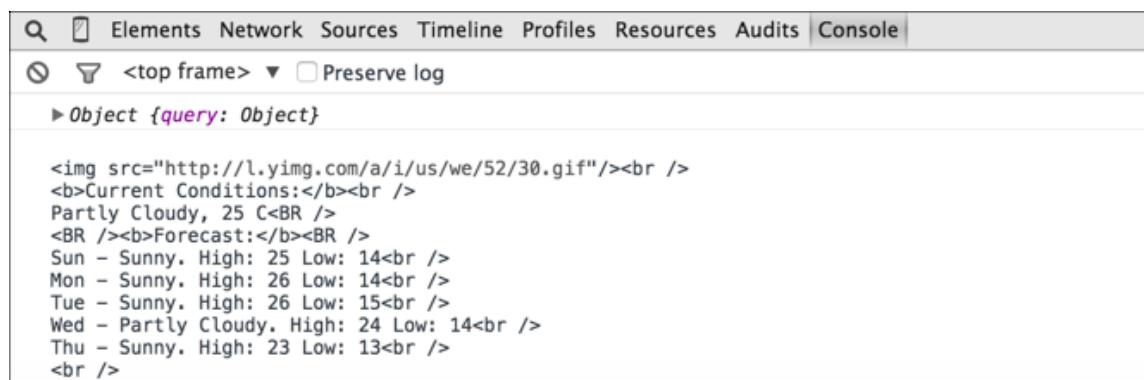
To pull out the `description` element from the JSON response and display it on the page, add this to your code sample, right below the `console.log(response)` line:

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

Your code should look like this:

```
.done(function (data) {
  console.log(data);
  console.log (data.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 = "data.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/1.1
1.1/jquery.min.js"></script>

<script>
var settings = {
    "async": true,
    "crossDomain": true,
    "dataType": "json",
    "url": "https://simple-weather.p.mashape.com/weatherdata?lat=3
7.354108&lng=-121.955236",
    "method": "GET",
    "headers": {
        "accept": "application/json",
        "x-mashape-key": "APIKEY"
    }
}

$.ajax(settings)

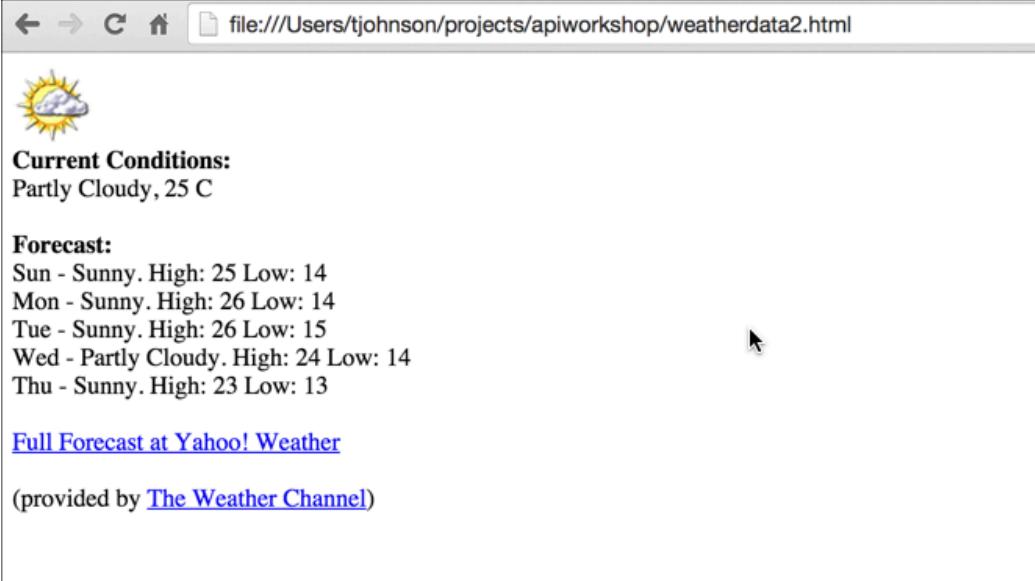
.done(function (response) {
    console.log(response);

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

</script>

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

Here's the result:



The screenshot shows a web browser window with the URL `file:///Users/tjohnson/projects/apiworkshop/weatherdata2.html`. The page displays weather information. At the top left is a sun and cloud icon. Below it, the text "Current Conditions:" is followed by "Partly Cloudy, 25 C". Under "Forecast:", there is a list of daily weather predictions: Sun - Sunny. High: 25 Low: 14; Mon - Sunny. High: 26 Low: 14; Tue - Sunny. High: 26 Low: 15; Wed - Partly Cloudy. High: 24 Low: 14; Thu - Sunny. High: 23 Low: 13. A blue link "Full Forecast at Yahoo! Weather" is present. Below the forecast, the text "(provided by The Weather Channel)" is shown. The browser interface includes standard navigation buttons (back, forward, search) and a status bar.

Now change the display to access the wind speed instead.

Diving into dot notation

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_chill").append (response.query.results.channel.wind.chill);
            $("#temperature").append (response.query.results.channel.units.temperature);
            $("#speed").append (response.query.results.channel.units.speed);
        })
    }
}

checkWind();
</body>

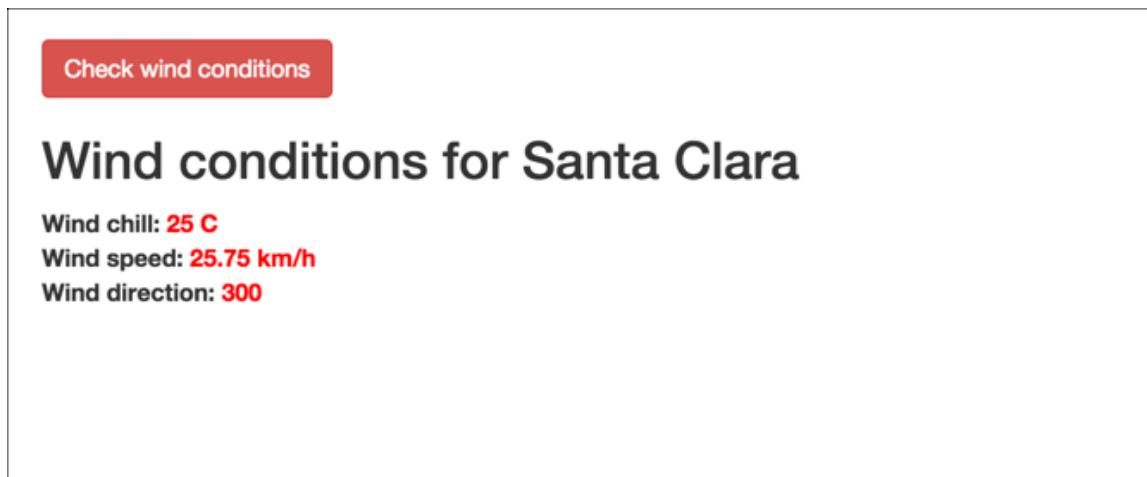
```

```
});  
}  
</script>  
  
<button type="button" onclick="checkWind()" class="btn btn-danger weat  
herbutton">Check wind conditions</button>  
  
<h2>Wind conditions for Santa Clara</h2>  
  
<b>Wind chill: </b><span id="wind_chill"></span> <span id="temperatur  
e"></span><br>  
<b>Wind speed: </b><span id="wind_speed"></span> <span id="speed"></sp  
an><br>  
<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:



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

In this section:

- [New API endpoint to document \(page 87\)](#)
- [Documenting resource descriptions \(page 91\)](#)
- [Documenting the endpoints and methods \(page 99\)](#)
- [Documenting parameters \(page 103\)](#)
- [Documenting sample requests \(page 111\)](#)
- [Documenting sample responses \(page 115\)](#)
- [Documenting code samples \(page 129\)](#)
- [Putting it all together \(page 140\)](#)

New API 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](http://www.alchemyapi.com/api/) (<http://www.alchemyapi.com/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 the course scenario (you can't ask the "team" questions about this fictitious new endpoint), the page will help.

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 (time: feet)
- wind (time: kts)
- tide (time: feet)
- water temperature (time: F degrees)
- recommendation - string ("Go surfing!", "Surfing conditions okay, not great", "Not today -- try some other activity.")

The recommendation is based on an algorithm that takes optimal surfing conditions, scores them in a rubric, and includes one of three responses.

Sample format:

```
{  
    "surfreport": [  
        {  
            "beach": "Santa Cruz",  
            "monday": {  
                "1pm": {  
                    "tide": 5,  
                    "wind": 15,  
                    "watertemp": 60,  
                    "surfheight": 5,  
                    "recommendation": "Go surfing!"  
                },  
                "2pm": {  
                    "tide": -1,  
                    "wind": 1,  
                    "watertemp": 50,  
                    "surfheight": 3,  
                    "recommendation": "Surfing conditions are okay, not great"  
                }  
            }  
            // ... 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 rip tide 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 (http://www.surfline.com/surf-report/south-beach-ca-northern-california_5088/) 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
- Endpoint
- Methods
- Parameters
- Request submission example
- Request response example
- Status and error codes
- Code samples

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](https://www.sublimetext.com/) (<https://www.sublimetext.com/>)) and [Markdown syntax](https://help.github.com/articles/github-flavored-markdown) (<https://help.github.com/articles/github-flavored-markdown>).

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 \(<http://developer.mailchimp.com/documentation/mailchimp/reference/overview/>\).](http://developer.mailchimp.com/documentation/mailchimp/reference/overview/)

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

Subresources

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

Available methods

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

In contrast, look at Twitter's API. In their [API Reference overview \(<https://dev.twitter.com/rest/reference>\)](https://dev.twitter.com/rest/reference), they call them endpoint docs:

These are the REST API endpoint reference docs.

A sample endpoint reference doc is [GET statuses/retweets/:id](https://dev.twitter.com/rest/reference/get/statuses/retweets/:id) (<https://dev.twitter.com/rest/reference/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 \(https://instagram.com/developer/endpoints/relationships/\)](https://instagram.com/developer/endpoints/relationships/). 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 interface. The left sidebar has a search bar and links to Overview, Authentication, Login Permissions, Permissions Review, Sandbox Mode, Secure Requests, and Endpoints. Under Endpoints, there are links to Users, Relationships (which is selected), Media, Comments, Likes, Tags, and Locations. The main content area is titled "Relationship Endpoints" and lists several API endpoints:

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

https://api.instagram.com/v1/users/self/follows?access_token=ACCESS-TOKEN

Get the list of users this user follows.

REQUIREMENTS

Scope: follower_list

PARAMETERS

ACCESS_TOKEN A valid access token.

The [EventBrite API \(https://www.eventbrite.com/developer/v3/endpoints/events/\)](https://www.eventbrite.com/developer/v3/endpoints/events/) 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 API v3 Documentation. The top navigation bar includes "Browse Events", a user profile for "Tom", "Help", and a "CREATE E" button. The main content area shows the "Events" endpoint:

Events ↗

GET /events/search/ ↗

Allows you to retrieve a paginated response of public `event` objects from across Eventbrite's directory, regardless of which user owns the event.

Parameters ↗

NAME	TYPE	REQUIRED	DESCRIPTION
q	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
```

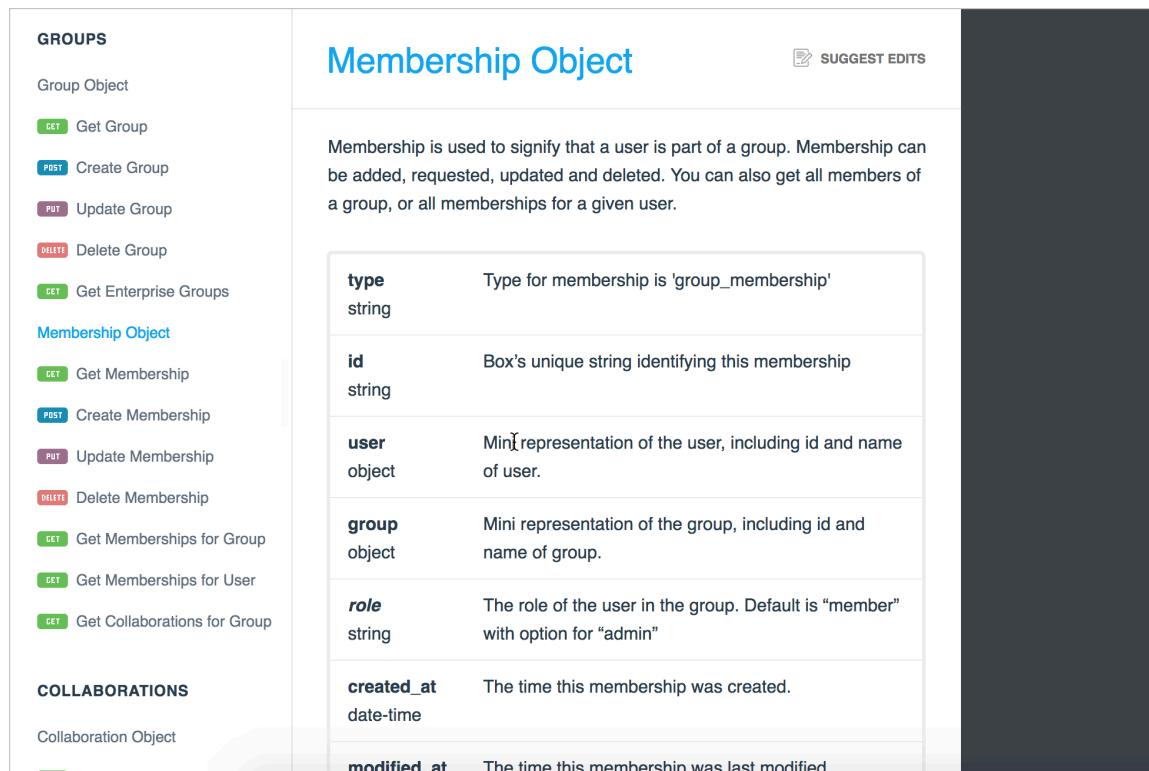
A rewards resource had various endpoints that returned different types of information related to rewards.

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](https://docs.box.com/reference#membership-object) (<https://docs.box.com/reference#membership-object>) has a good example of how the same resource can have multiple endpoints and methods.



The screenshot shows a detailed API documentation page for the **Membership Object**. On the left sidebar, under the **GROUPS** section, there are several endpoints listed:

- Group Object
 - GET** Get Group
 - POST** Create Group
 - PUT** Update Group
 - DELETE** Delete Group
 - GET** Get Enterprise Groups
- Membership Object**
 - GET** Get Membership
 - POST** Create Membership
 - PUT** Update Membership
 - DELETE** Delete Membership
 - GET** Get Memberships for Group
 - GET** Get Memberships for User
 - GET** Get Collaborations for Group
- COLLABORATIONS**
 - Collaboration Object

The main content area is titled **Membership Object** and contains a detailed description: "Membership is used to signify that a user is part of a group. Membership can be added, requested, updated and deleted. You can also get all members of a group, or all memberships for a given user." Below this description is a table of properties:

type string	Type for membership is 'group_membership'
id string	Box's unique string identifying this membership
user object	Mini representation of the user, including id and name of user.
group object	Mini 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 \(page 87\)](#) 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 (<https://github.com/SciDevs/delicious-api/blob/master/api/posts.md#v1postsupdate>)

/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 (<https://developer.foursquare.com/docs/venues/menu>)

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](http://www.aerisweather.com/support/docs/api/reference/endpoints/) (<http://www.aerisweather.com/support/docs/api/reference/endpoints/>).

With Aeris Weather, the description for the forecasts endpoint (<http://www.aerisweather.com/support/docs/api/reference/endpoints/forecasts/>) 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 the 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 it to represent parameter values with a colon, like this:

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

You can see this convention in the [EventBrite API](https://www.eventbrite.com/developer/v3/) (<https://www.eventbrite.com/developer/v3/>) and the [Aeris Weather API](http://www.aerisweather.com/support/docs/api/) (<http://www.aerisweather.com/support/docs/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 button labeled 'POST'. The main content area shows the 'Create Comment' endpoint with a 'POST' method. It describes the action as adding a comment to a specific file or comment. Below this is a 'Parameters' table:

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

On the right side, there is a 'Definition' section showing the URL `/comments`, an 'Example Request' with a curl command, and an 'Example Response' showing a 201 Created status with a JSON object.

(<https://box-content.readme.io/#comment-object>)

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

The screenshot shows the LinkedIn API documentation. The main title is 'Data Formats'. Below it is a section titled 'Requesting data from the APIs'. It states that unless otherwise specified, all LinkedIn APIs return XML data format. A 'GET' request example is shown for the endpoint `https://api.linkedin.com/v1/people/~`. The 'sample response' is displayed as XML:

```

<?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 chapter on design patterns, I explore how [long pages \(page 360\)](#) 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

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.

List parameters in a table

Many times parameters are listed in a simple table 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
API v2		/v2/search	GET	Search for local businesses.
Get started		Note: at this time, the API does not return businesses without any reviews.		
API console		General Search Parameters		
Documentation		Name	Data Type	Required / Optional
Introduction		term	string	optional
Authentication		Search term (e.g. "food", "restaurants"). If term isn't included we search everything. The term keyword also accepts business names such as "Starbucks".		
Search API		limit	number	optional
Business API		offset	number	optional
Phone Search API		sort	number	optional
iPhone Apps		Number of business results to return		
Errors		Offset the list of returned business results by this amount		
Code samples		Sort mode: 0=Best matched (default), 1=Distance, 2=Highest Rated. If the mode is 1 or 2 a search may retrieve an additional 20 businesses past the initial limit of the first 20 results. This is done by specifying an offset and limit of 20. Sort by distance is only supported for a location or geographic search. The rating sort is not strictly sorted by the rating value, but by an adjusted rating value that takes into account the number of ratings, similar to a bayesian		

You can format the values in a variety of ways (aside from a table). If you're using a definition list or other non-table format, 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 Swagger 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 the Twilio REST API documentation for the Lookups subdomain. On the left, there's a sidebar with 'Overview' and a main content area titled 'REST API: LOOKUP'. The content area includes a description of the Lookup subdomain, examples for 'Lookups Subdomain' and 'Resource URI', and a note about providing a number in local/national format. On the right, there's a code editor for C# showing a sample program to look up a phone number using the Twilio Lookups API. The code uses the twilio-csharp library and includes constants for Account SID and Auth Token, and a method to get a phone number by its E.164 formatted number.

```

1 // Download the twilio-csharp library from twilio.com/docs/csharp/api
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 }

```

(<https://www.twilio.com/docs/api/rest/lookups>)

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 or 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 \(page 145\)](#).

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 ParameterDescription

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](http://developer.ebay.com/DevZone/finding/CallRef/findItemsByProduct.html) (<http://developer.ebay.com/DevZone/finding/CallRef/findItemsByProduct.html>) endpoint.

```

<?xml version="1.0" encoding="utf-8"?>
<findItemsByProductRequest xmlns="http://www.ebay.com/marketplace/search/v1/services">
  <!-- Call-specific Input Fields -->
  <itemFilter> ItemFilter
    <name> ItemFilterType </name>
    <paramName> token </paramName>
    <paramValue> string </paramValue>
    <value> string </value>
    <!-- ... more value values allowed here ... -->
  </itemFilter>
  <!-- ... more itemFilter nodes allowed here ... -->
  <outputSelector> OutputSelectorType </outputSelector>
  <!-- ... more outputSelector values allowed here ... -->
  <productId type="string"> ProductId (string) </productId>
  <!-- Standard Input Fields -->
  <affiliate> Affiliate
    <customId> string </customId>
    <geoTargeting> boolean </geoTargeting>
    <networkId> string </networkId>
    <trackingId> string </trackingId>
  </affiliate>
  <buyerPostalCode> string </buyerPostalCode>
  <paginationInput> PaginationInput
    <entriesPerPage> int </entriesPerPage>
    <pageNumber> int </pageNumber>
  </paginationInput>
  <sortOrder> SortOrderType </sortOrder>
</findItemsByProductRequest>

```

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

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

But the sample request also contains links to each of the parameters. When you click a parameter value in the sample request, you go to a page that provides more details about that parameter value, such as the [ItemFilter](http://developer.ebay.com/DevZone/finding/CallRef/types/ItemFilter.html) (<http://developer.ebay.com/DevZone/finding/CallRef/types/ItemFilter.html>). 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](http://idratherbewriting.com/pubapis_swagger/) (http://idratherbewriting.com/pubapis_swagger/) any better?

The [Swagger UI](https://github.com/swagger-api/swagger-ui) (<https://github.com/swagger-api/swagger-ui>) reads the Swagger spec file and displays it in the visual format that you see with examples such as the [Swagger Petstore](http://petstore.swagger.io/) (<http://petstore.swagger.io/>).

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](http://petstore.swagger.io/#/pet/addPet) (<http://petstore.swagger.io/#/pet/addPet>), 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
	<pre> 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] } </pre>
Responses	Responses

This view describes the various parts of the request, noting the data types and any descriptions in your Swagger 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 chapter, I dive into Swagger. If you want to skip there now, go to [Introduction to Swagger \(page 273\)](#).

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

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

Here's what my table looks like:

Parameter	Required	Description	Type
days	Optional	The number of days to include in the response. Default is 3. 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.	Integer
units	Optional	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 integer. Unix current hour will be returned in the format (ms since response. 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	
The following table provides some example uses and their corresponding URL with query parameters. Click on the links to try them out.	
Usage	URL
Find movie theaters in zip code 90045	https://api.citygridmedia.com/content/places/v2/search/where?type=movietheater&where=90045&publisher=test
Find Italian restaurants in Chicago using placement "sec-5"	https://api.citygridmedia.com/content/places/v2/search/where?what=restaurant&where=chicago,IL&tag=11279&placement=sec-5&publisher=test
Find hotels in Boston, viewing results 1-5 in alphabetical order	https://api.citygridmedia.com/content/places/v2/search/where?what=hotels&where=boston,ma&page=1&rpp=5&sort=alpha&publisher=test
Find pharmacies near the L.A. County Music Center, sorted by distance	https://api.citygridmedia.com/content/places/v2/search/where?what=pharmacy&where=135+N+Grand,LosAngeles,ca&sort=dist&publisher=test

(<http://docs.citygridmedia.com/display/citygridv2/Places+API>)

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
```

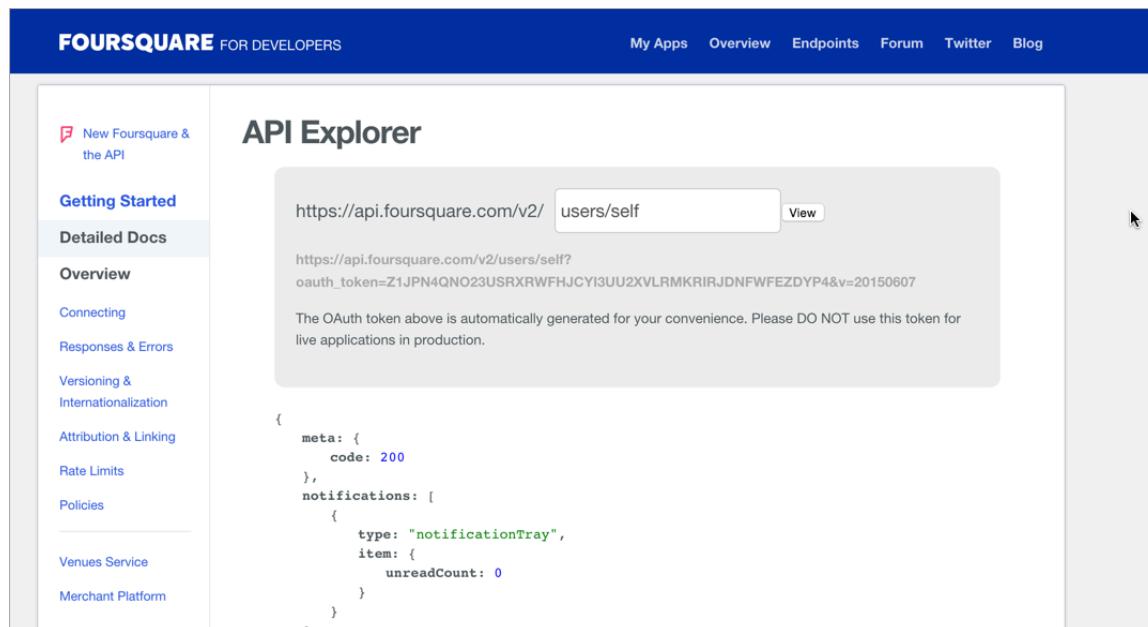
If responses vary a lot, consider including multiple responses with the requests. How many different requests and responses should you show? There's probably no easy answer, but probably no more than a few. You decide what makes sense for your API.

In the CityGrid Places API, notice how the examples don't include the sample responses on the same page but rather link to live examples. When you click the URL link, you execute the request in your browser and can see the response. (Here's [an example](http://api.citygridmedia.com/content/places/v2/search/where?type=movietheater&where=90045&publisher=test) (<http://api.citygridmedia.com/content/places/v2/search/where?type=movietheater&where=90045&publisher=test>)).

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, you can see Foursquare's API explorer here:



The screenshot shows the Foursquare API Explorer. The left sidebar has a navigation menu with links like 'New Foursquare & the API', 'Getting Started' (which is selected), 'Detailed Docs', 'Overview', 'Connecting', 'Responses & Errors', 'Versioning & Internationalization', 'Attribution & Linking', 'Rate Limits', 'Policies', 'Venues Service', and 'Merchant Platform'. The main area is titled 'API Explorer' and contains a code editor with a URL input field containing 'https://api.foursquare.com/v2/users/self'. Below the URL is a generated OAuth token: 'oauth_token=Z1JPN4QNO23USRXRFHJCYI3UU2XVLRMKRIRJDNFWFEDYP4&v=20150607'. A note says, 'The OAuth token above is automatically generated for your convenience. Please DO NOT use this token for live applications in production.' At the bottom of the code editor, there is a JSON response preview:

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

(<https://developer.foursquare.com/docs/explore>)

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 several endpoint names: 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 specific endpoints are detailed with their methods, 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 →**

(http://developer.nytimes.com/books_api.json)

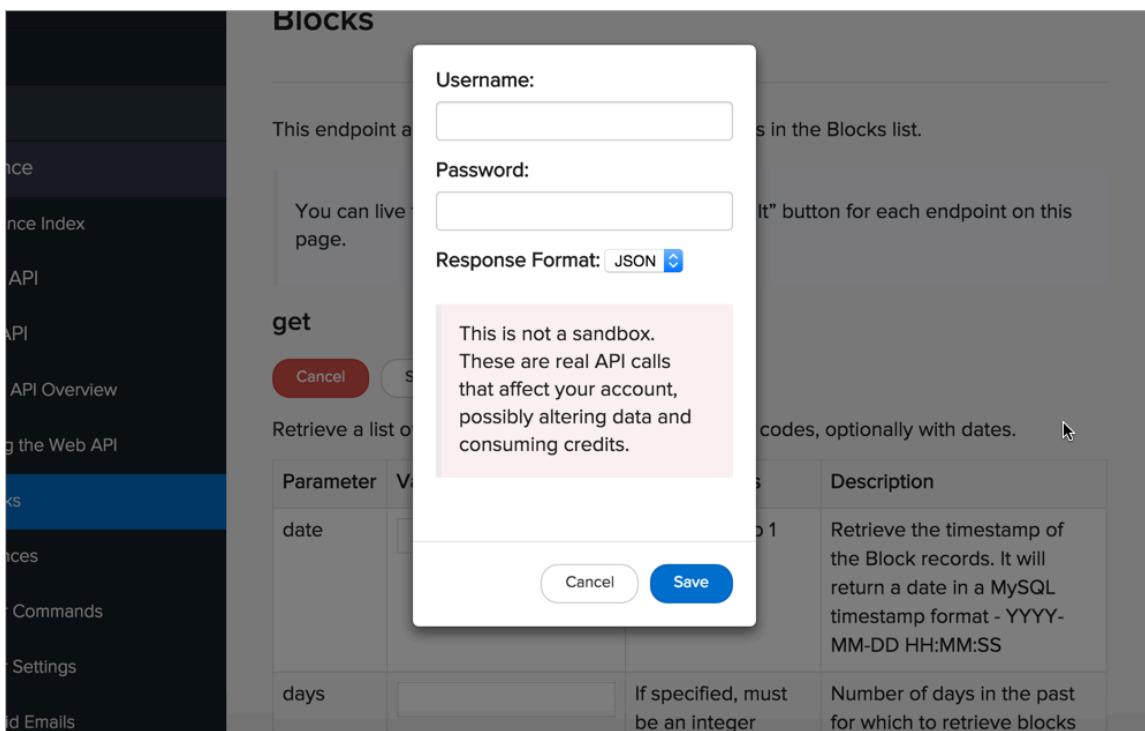
This example compels users to try out the endpoints to get a better understanding of the information they return.

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](http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/apis/) (<http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/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:



(https://sendgrid.com/docs/API_Reference/Web_API/blocks.html)

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](http://swagger.io/swagger-ui/) (<http://swagger.io/swagger-ui/>) (which parses a Swagger spec file) and [Readme.io](http://readme.io) (<http://readme.io>) (which allows you to enter the details manually) can integrate an API Explorer functionality directly into your documentation.

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

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'
```

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:

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

```
{  
  "items": [  
    {  
      "published": "2012-01-04T10:07:12+01:00",  
      "title": "pthulin flattened \"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"  
    }  
  ]  
}
```

(<http://developers.flattr.net/api/resources/activities/>)

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](https://dev.twitter.com/rest/public) (<https://dev.twitter.com/rest/public>).

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
<code>uid</code>	The user's unique Dropbox ID.
<code>display_name</code>	The user's display name.
<code>name_details/given_name</code>	The user's given name.
<code>name_details/surname</code>	The user's surname.
<code>name_details/familiar_name</code>	The locale-dependent familiar name for the user.
<code>referral_link</code>	The user's referral link .
<code>country</code>	The user's two-letter country code, if available.
<code>locale</code>	Locale preference set by the user (e.g. en-us).
<code>is_paired</code>	If true, there is a paired account associated with this user.
<code>team</code>	If the user belongs to a team, contains team information. Otherwise, null.
<code>team/name</code>	The name of the team the user belongs to.
<code>team/team_id</code>	The ID of the team the user belongs to.
<code>quota_info/normal</code>	The user's used quota outside of shared folders (bytes).
<code>quota_info/shared</code>	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).
<code>quota_info/quota</code>	The user's total quota allocation (bytes). If the user belongs to a team, the team's total quota allocation (bytes).

(<https://www.dropbox.com/developers/core/docs#disable-token>)

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

(http://dev.bitly.com/user_info.html)

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 (<https://www.udemy.com/api-documentation-1-json-and-xml/>), he also represents the nested objects using tables:

Song JSON Documentation

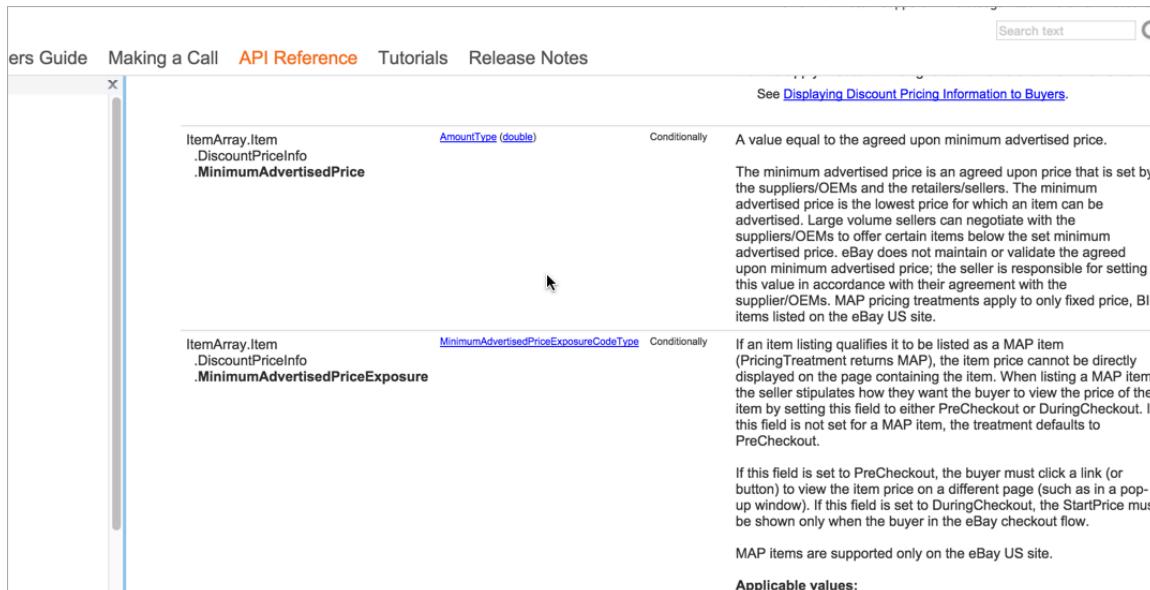
Represents a song.

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	

(<http://idratherbewriting.com/2015/05/22/api-technical-writing-course-on-udemy/>)

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 eBay. At the top, there are navigation links: 'Users Guide', 'Making a Call', 'API Reference' (which is highlighted in orange), 'Tutorials', and 'Release Notes'. A search bar is located at the top right. Below the header, there is a section titled 'See Displaying Discount Pricing Information to Buyers.' with a link. The main content area contains two table rows. The first row is for 'ItemArray.Item.DiscountPriceInfo.MinimumAdvertisedPrice'. It shows the type as 'AmountType (double)', a condition as 'Conditionally', and a description as 'A value equal to the agreed upon minimum advertised price.' The second row is for 'ItemArray.Item.DiscountPriceInfo.MinimumAdvertisedPriceExposure'. It shows the type as 'MinimumAdvertisedPriceExposureCodeType', a condition as 'Conditionally', and a description as 'If an item listing qualifies it to be listed as a MAP item (PricingTreatment returns MAP), the item price cannot be directly displayed on the page containing the item. When listing a MAP item, the seller stipulates how they want the buyer to view the price of the item by setting this field to either PreCheckout or DuringCheckout. If this field is not set for a MAP item, the treatment default to PreCheckout.' Below these descriptions, there is a note: 'MAP items are supported only on the eBay US site.' and a section for 'Applicable values:'.

(<http://developer.ebay.com/Devzone/shopping/docs/CallRef/FindPopularItems.html>)

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>
  
```

(<http://developer.ebay.com/Devzone/shopping/docs/CallRef/FindPopularItems.html>)

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 for the charge object. On the left is a sidebar with links like 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. The first column contains endpoint descriptions and parameters. The second column contains detailed descriptions of parameters. The third column shows examples of how to use the API, including curl commands and code snippets in Ruby, Python, PHP, Java, Node.js, and Go. In the third column, there is a 'Hide child attributes' button, which is highlighted with a mouse cursor.

stripe api		
Introduction	currency	3-letter ISO code for currency. REQUIRED
Authentication	customer	optional, either customer or source is required The ID of an existing customer that will be charged in this request.
Errors	source	optional, either source or customer is required A payment source to be charged, such as a credit card. If you also pass a customer ID, the source must be the ID of a source belonging to the customer. Otherwise, if you do not pass a customer ID, the source you provide must either be a token, like the ones returned by Stripe.js, or a dictionary containing a user's credit card details, with the options described below. Although not all information is required, the extra info helps prevent fraud.
Pagination		
Versioning		
Expanding objects		
Metadata		
Idempotent requests		
METHODS		
Charges		
The charge object		
Create a charge		
Retrieve a charge		
Update a charge		
Capture a charge		
List all charges		
Refunds		

[Hide child attributes](#)

	curl	Ruby	Python	PHP	Java	Node	Go
"paid": true,							
"status": "succeeded",							
"amount": 45000,							
"currency": "usd",							
"refunded": false,							
"source": {							
"id": "card_16B0TbZeZvKYlo2CMJn4vwjg",							
"object": "card",							
"last4": "4242",							
"brand": "Visa",							
"funding": "credit",							
"exp_month": 11,							
"exp_year": 2018,							
"country": "US",							
"name": "admin@wechatgogo.com",							
"address_line1": null,							
"address_line2": null,							
"address_city": null,							
"address_state": null,							
"address_zip": null,							
"address_country": null,							
"cvc_check": "pass",							
"address_line1_check": null,							
"address_zip_check": null,							
"dynamic_last4": null,							
"metadata": {							
},							
"customer": null							
},							
"captured": true,							

(https://stripe.com/docs/api#create_charge)

A lot of APIs have modeled their design after Stripe's. (For example, see [Slate](https://github.com/tripit/slate) (<https://github.com/tripit/slate>) or readme.io (<http://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 tripane 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](http://jsonformatter.curiousconcept.com/) (<http://jsonformatter.curiousconcept.com/>) 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](http://pygments.org/) (<http://pygments.org/>). 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 (<http://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](https://developer.rhapsody.com/api) (<https://developer.rhapsody.com/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 scrolling-to-definitions approach

In my [documentation theme for Jekyll](http://idratherbewriting.com/documentation-theme-jekyll) (<http://idratherbewriting.com/documentation-theme-jekyll>), I tried an approach to documenting JSON that uses a jQuery plugin called ScrollTo. You can [see it here](http://idratherbewriting.com/documentation-theme-jekyll/mydoc_scroll.html) (http://idratherbewriting.com/documentation-theme-jekyll/mydoc_scroll.html):

The screenshot shows a 'Scroll demo' page. On the left, there is a large block of JSON code:

```
{  
  "apples": "red fruit at the store",  
  "bananas": "yellow bananas in a bunch",  
  "carrots": "orange vegetables that grow in the ground",  
  "dingbats": "a type of character symbol on a computer",  
  "eggs": "chickens lay them, and people eat them",  
  "falafel": "a Mediterranean sandwich consisting of lots of different stuff i don't know much about",  
  "giraffe": "tall animal, has purple tongue",  
  "hippo": "surprisingly dangerous amphibian",  
  "igloo": "an ice shelter made by eskimos",  
  "jeep": "the only car that starts with a j",  
  "kilt": "something worn by scottish people, not a dress",  
  "lamp": "you use it to read by your bedside at night",  
  "manifold": "an intake mechanism on a car, like a valve, i thin k",  
  "octopus": "eight tentacles, shoots ink, lives in dark caves, ve ry mysterious",  
  "paranoid": "the constant feeling that others are out to get yo
```

To the right of the JSON, there are two columns of definitions. The first column contains the words 'apples' and 'bananas'. The second column contains their definitions:

apples
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer magna massa, euismod sed rutrum at, ullamcorper quis tellus. Vestibulum erat purus, aliquet sit amet pellentesque eget, tempus at ante. Nulla justo nisi, elementum nec nisi eget, consectetur varius tortor.

bananas
Curabitur quis nibh sed eros viverra tempus et quis lorem. Nulla convallis sit amet risus vitae rutrum. Nulla at faucibus lectus.

(http://idratherbewriting.com/documentation-theme-jekyll/mydoc_scroll.html)

When you click on an item in the JSON object, the right-pane scrolls to the item's description. I like this approach, though I've not really seen it done in other API documentation sites.

One problem is that you end up with three scroll bars on one page, which isn't the best design. Additionally, the descriptions in this demo are just paragraphs. Usually you structure the information with more detail (for example, data type, description, notes, etc.).

Also, this approach doesn't allow for easy scanning. However, this scrolling view might be an alternative view to a more scannable table. That is, you could store the definitions in another file and then include the definitions in both this scrolling view and a master table list, allowing the user to choose the view he or she wants.

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.

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

curl Ruby Python PHP Java Node Go

```

"re珑emo": false,
"source": {
  "id": "card_165Fnw2eZvKYlo2CnypZm9H9",
  "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_162Ziz2eZvKYlo2CfodNCnZ",
"failure_message": null,
"failure_code": null,
"amount_refunded": 0,
"statement_descriptor": null
}

```

(https://stripe.com/docs/api#charge_object)

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](#) (<https://dev.twitter.com/rest/reference/post/statuses/retweet/%3Aid>) 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\"\u003CMy 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": []  
    }  
}
```

(<https://dev.twitter.com/rest/reference/post/statuses/retweet/%3Aid>)

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](http://idratherbewriting.com/pubapis_raml/) (http://idratherbewriting.com/pubapis_raml/) to document endpoints with JSON objects in the request body, the RAML API Console output looks something like this:

The screenshot shows a RAML API documentation interface. At the top, it says "BODY" and "application/json". Below that is a "Hide Schema" button. The main area contains a JSON schema for a file object:

```
{
  "$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.

The screenshot shows a JSON schema table on the left and a code sample on the right. The table has columns for 'Attribute Details' and 'Example json GET response'. A tooltip for the 'Type' column in the table is expanded, showing a description about account code format. The code sample shows a JSON object with various fields like UID, Name, Classification, Type, Number, Description, ParentAccount, and TaxCode, each with its corresponding value.

```

{
  "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": "f5cc95d6-3472-4227-8c45-7a95c322e38b",
    "Name": "Bank Accounts",
    "DisplayID": "1-1100",
    "URI": "(domain)/{cf_guid}/GeneralLedger/Account/f5cc95d6-3472-4227-8c45-7a95c322e38b"
  },
  "IsActive": true,
  "TaxCode": {
    "UID": "94966872-b140-4da2-bc43-5dd74f33a09",
    "Code": "N-T",
    "URI": "(domain)/{cf_guid}/GeneralLedger/TaxCode/94966872-b140-4da2-bc43-5dd74f33a09"
  },
  "Level": 4,
  "OpeningBalance": 100000.00,
  "CurrentBalance": 5000.00
}

```

(<http://developer.myob.com/api/accountright/v2/generalledger/account/#GET>)

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. 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}/wind	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}/watertemp	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}/surfheight	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
{day}/{time}/recommendationsurfing."	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 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#, Ruby, 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:

The screenshot shows a section of the Evernote API documentation. On the left, there's a sidebar with navigation links for Error Handling, The Sandbox, Authentication, Developer, OAuth, Permissions, Revocation, Rate Limits, Overview, Best Practices, Notes, Creating Notes, Sharing Notes, Reminders, Read-only, ENML, Note Links, Applications, Notebooks, Sharing Notebooks, and App Notebooks. The main content area has a heading "Sharing Notes" and a sub-section "Sharing a Note". It contains a list of requirements:

- The GUID of the note you'd like to share.
- The ID of the Shard that houses the note to be shared.
- A valid authentication token or developer token.
- Initialized instances of `NoteStore.Client` and `UserStore.Client`

The Shard ID can be determined at runtime by querying the UserStore:

Python	Objective-C	Ruby	PHP	Java	Node.js
--------	-------------	------	-----	------	---------

```

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

```

[main.py hosted with ❤ by GitHub](#) [view raw](#)

Assuming all of that is in place, sharing a note is actually quite simple. By calling

[\(https://dev.evernote.com/doc/articles/note-sharing.php\)](https://dev.evernote.com/doc/articles/note-sharing.php)

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".

Example 1: This section contains a code snippet for making a call using the Node.js helper library. The code uses the Twilio Client to make a POST request to a specified URL with the account SID and auth token. The code is as follows:

```
// 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);
});
```

Example 2: This section contains a code snippet for making a call from a Twilio Client named "tommy". The Twilio Client will POST to a specified URL to fetch TwiML to handle the call. The code is as follows:

```
(https://www.twilio.com/docs/api/rest/making-calls)
```

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 app. At the top, there's a header with the Paw logo and a rocket icon. Below the header, the word 'Extensions' is centered. 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, highlighted in blue), '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.

(<https://luckymarmot.com/paw/extensions/>)

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

A dropdown menu titled 'HTTP' is shown, listing various code generators. The 'JavaScript (jQuery)' option is selected and highlighted with a blue background and a checkmark icon. Other options include:

- API Blueprint Generator
- Betamax Generator
- cURL
- Go (HTTP)
- HTTPie
- Java (Apache HttpClient via Fluent API)
- JavaScript (jQuery) **✓**
- Objective-C (NSURLConnection)
- Objective-C (NSURLSession)
- PHP (cURL)
- PHP (Guzzle)
- Python (Requests)
- Ruby (Net::HTTP)
- Swift (Alamofire)
- Swift (NSURLSession)
- WordPress

Below the main list, there are two additional items:

- Amazon S3 String To Sign
- OAuth 1 Signature Base

At the bottom of the menu, there's a link: 'Find More Code Generators...'.

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

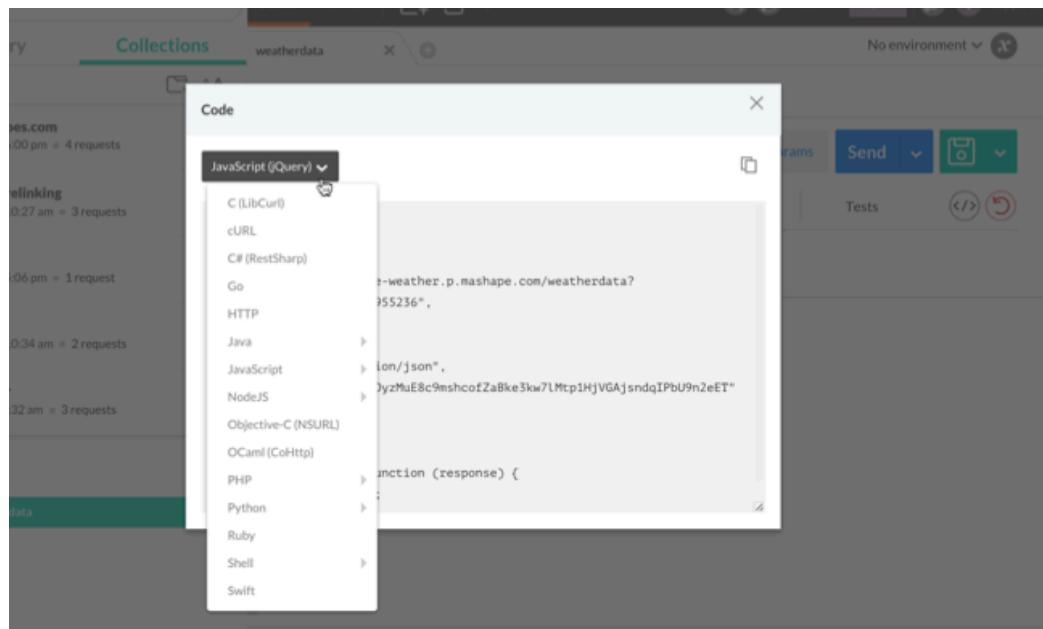
Although these code generators are probably helpful, they may or may not work for your API. Always review code samples with developers. In most cases, developers supply the code samples for the documentation, and technical writers briefly comment on the code samples.

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=37.354108&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>
```

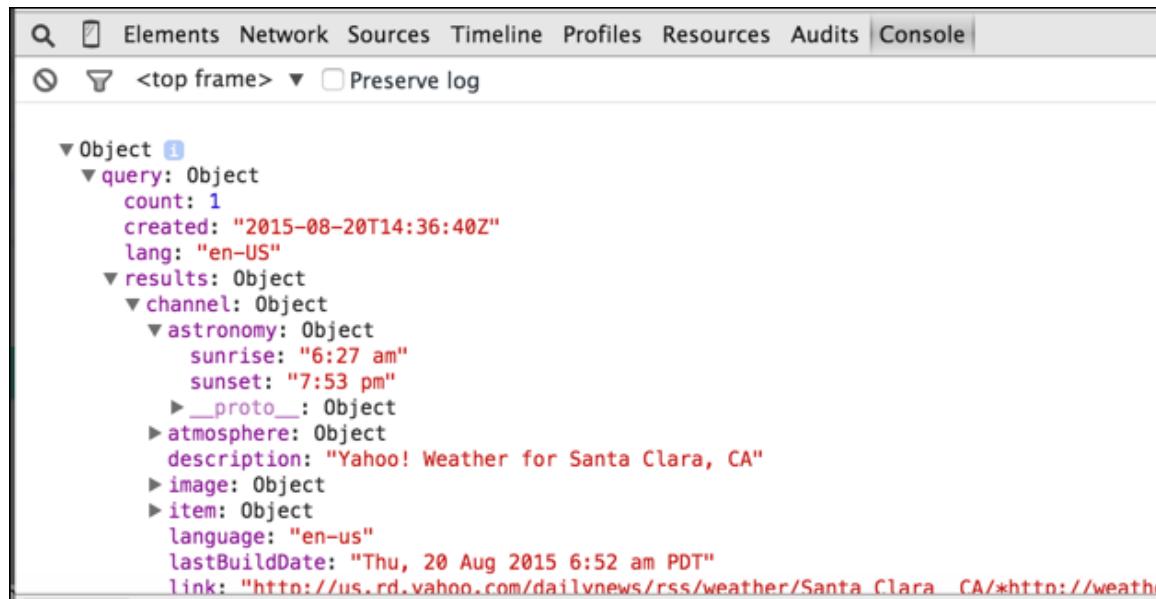
3. The Mashape Weather API requires the `dataType` parameter, which Postman doesn't automatically include. Add `"dataType": "json"`, in the list of `settings`:

```
<!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=37.354108&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.



The screenshot shows the Chrome DevTools Elements tab with the "Console" tab selected. A dropdown menu is open, showing the path <top frame>. The tree view displays a nested object structure:

- Object 1
 - 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/CA/Santa_Clara.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&lng=-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, Climate, and a Sign In button. Below the navigation is a blue header bar with the word "DOCUMENTATION". Underneath, there's a menu bar with links for API Home, Pricing, Featured Applications, Documentation, and Forums. On the left, there's a sidebar titled "API Table of Contents" which lists categories like Weather API, WunderMap Layers, Data Features, and others. The main content area has a title "Code Samples" and a section for "PHP" containing sample 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";
?>
```

Below the PHP section is a section for "Ruby" containing sample 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 sidebar titled "Page Contents" with links for Code Samples, PHP, Ruby, Python, ColdFusion, and JavaScript with jQuery.

(<http://www.wunderground.com/weather/api/d/docs?d=resources/code-samples&MR=1>)

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](#) (<http://docs.mashape.com/javascript>) 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 conditions for a specific beach. In this case, the code shows the overall recommendation about whether to go surfing.

```
<!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",
    "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.

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

In this example, let's pull together the various parts you've worked on 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'
```

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.
{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 per hour or kilometers per hour depending on the units you specify. Wind affects the surf height and general wave conditions.
{day}/{time}/watertemp	Wind speeds of more than 15 knots per hour make surf conditions undesirable, since the wind creates white caps and choppy waters.
{day}/{time}/surfheight	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}/recommendationsurfing	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. 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 codeMeaning

609	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 docs

In this section:

- Set up a test environment (page 146)
- Test all instructions yourself (page 150)
- Test your assumptions (page 155)

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 are necessary to test the API docs as closely as possible to what actual users will do or use.

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. It can be challenging to build out local versions of code on your own box. You may need to install some special utilities or frameworks, get 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 want a sample app to actually test out the calls or other code. You might need to ask them for it. Many times the company wiki has instructions on how to set up local builds.

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.



(<https://flic.kr/p/6Grete>)

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. Developers also like to show off their technical acumen.

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. The first time I asked a dev 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.

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, since 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 web server environments because the code tends to be more stable. Further, if you can hook into a set of scripts QA uses to run tests, you can often get a jump start on the tasks you're documenting. Good test cases usually list out the steps required to produce a result, and the scripts can inform the documentation you write.

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

You may also need to construct certain YML files necessary to configure a server with the settings you want to test. Understanding exactly how to create the YML 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 may be 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. You often include a sample app (or multiple apps in various programming languages) that 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 it 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.

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.

Hardware products

If you're documenting a hardware product, you'll want to secure a device that has the 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 my jobs where I could actually get my hands on and play around with.)

Next steps

It might take one 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 \(page 150\)](#).

Test all instructions yourself

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

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 you can confront an engineer and tell him or her that something isn't working correctly, or you can start to make suggestions for improving things. You can't do this if you're just taking notes about what engineers say, or if you're just copying information from specs or engineer-written pages.

When things don't work, you can identify and log bugs. 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 probably submit the test calls using cURL or Postman. This will allow you to save the calls so you can test 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 (a different one from the engineer who said the database would need configuration) 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 (like a starter kit) called [Fire App Builder](https://developer.amazon.com/public/solutions/devices/fire-tv/docs/fire-app-builder-overview) (<https://developer.amazon.com/public/solutions/devices/fire-tv/docs/fire-app-builder-overview>). This app template was 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](http://podcast.writethedocs.org/) podcast (<http://podcast.writethedocs.org/>) and various WTD meetups and use that media for the app.

Since Fire App Builder doesn't support YouTube as a web host, I downloaded the MP4s from YouTube and uploaded them directly to my web host, Bluehost.

Then I needed to construct the media feed that I would use to integrate with Fire App Builder. Fire App Builder basically reads all the media from a feed whose syntax you target with Jayway Jsonpath or XPath expression queries.

I used Jekyll to build my feed. You can view my JSON-based feed here: podcast.writethedocs.org/fab.json (<http://podcast.writethedocs.org/fab.json>). The feed is automatically generated 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 Talk forum](http://talk.jekyllrb.com/t/how-to-exclude-comma-in-last-item-in-for-loop-that-is-prefaced-by-if-condition-and-output-valid-json/380/4) (<http://talk.jekyllrb.com/t/how-to-exclude-comma-in-last-item-in-for-loop-that-is-prefaced-by-if-condition-and-output-valid-json/380/4>).

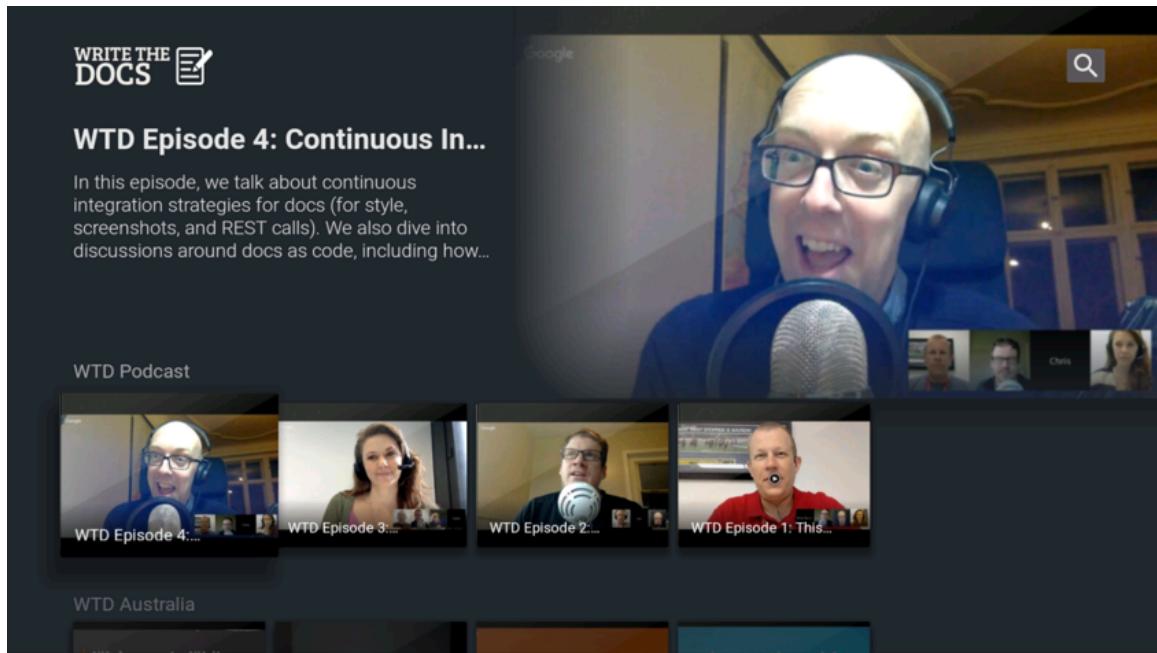
After I had the media and the feed, integrating it into Fire App Builder was easy because, after all, I wrote the documentation on how to do that.

Submitting the app into the Appstore was fun. You can view the Write the Docs podcast app in the Amazon Appstore website [here](https://www.amazon.com/dp/B06Y23TNC4/ref=sr_1_1?ie=UTF8&qid=1491708630&sr=1-1&keywords=write+the+docs) (https://www.amazon.com/dp/B06Y23TNC4/ref=sr_1_1?ie=UTF8&qid=1491708630&sr=1-1&keywords=write+the+docs).

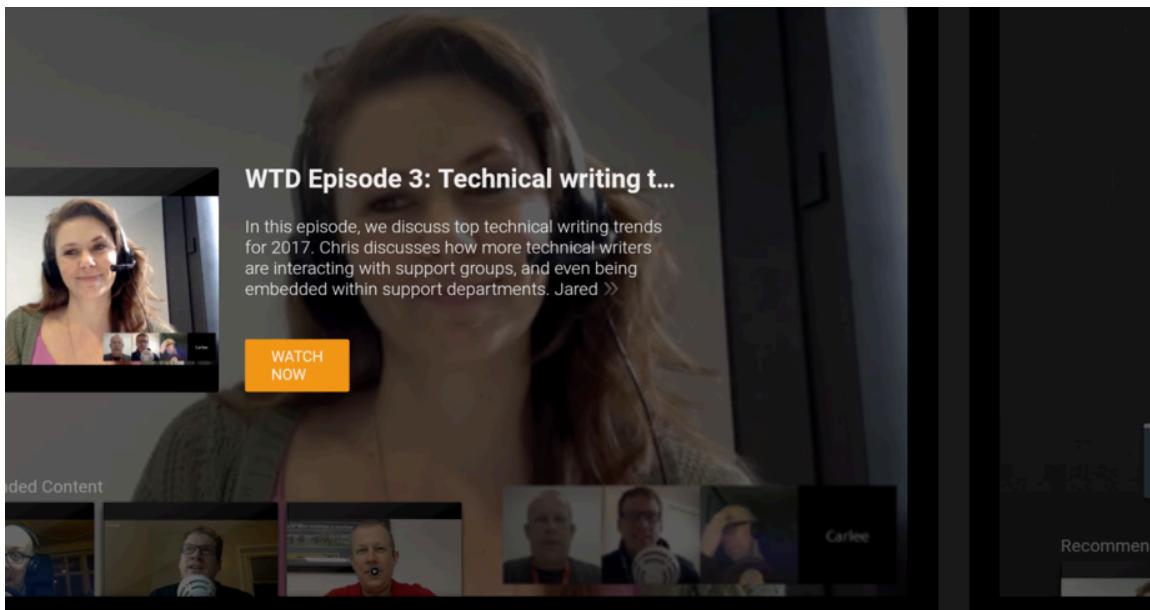
The screenshot shows the Amazon Appstore for Android interface. At the top, the search bar contains "write the docs". Below the search bar, there are navigation links for "Departments", "Browsing History", "Tom's Amazon.com", "Today's Deals", "Gift Cards & Registry", "Sell", and "Help". A horizontal menu bar includes "Appstore for Android", "Underground Apps", "Best Sellers", "Amazon Coins", "Deals", "New Releases", "Fire Tablet Apps", and "Fire TV Apps". Below this, a link says "Back to search results for 'write the docs'". The main content area features a large graphic with the text "WRITE THE DOCS" in bold, uppercase letters. To the right of the graphic, the title "Write the Docs Podcast" is displayed, followed by the developer "I'd Rather Be Writing" and the rating "All Ages". A button "Be the first to review this item" is shown. Below this, the price is listed as "Free Download" with a note about saving up to 20% on Amazon Coins. It also mentions being sold by "Amazon Digital Services LLC" and available "instantly" with in-app purchases. The app works with "Game Controllers, Fire TV Voice Remote". A section titled "This app needs permission to access:" lists "Open network sockets" and "Access information about networks", with a link to "See all Application Permissions" and "Contact Developer".

(https://www.amazon.com/I-Rather-Be-Writing-Podcast/dp/B06Y23TNC4/ref=sr_1_1?s=mobile-apps&ie=UTF8&qid=1491708630&sr=1-1&keywords=write+the+docs)

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 out 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 Fire App Builder template, though.

I also discovered an issue with Fire App Builder that developers had been unaware of. Although developers had tested the app template for many months, they hadn't tested pushing apps into the Appstore with Fire App Builder. It turns out that Fire App Builder's in-app purchases component (not active or configured by default) automatically triggers 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 third-party devs 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 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 the fact that it is absolutely vital 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?

Empowered to test additional features

It is often difficult to set up scenarios to test developer tools, but it's necessary, and once you set up a system, it empowers you to test and try out many other features. For example, after I had the Android app built, I could then test out the recommendations feature, which was the big feature in the release. I could also test out device targeting, audio focus handling, and other 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.

For example, for general Fire TV users who aren't using the Fire App Builder 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 this information about recommendations 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. When you actually send recommendations to Fire TV, Fire TV only uses *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.

Putting together an app from scratch that leverages all the recommendation API calls requires more effort, 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.

Testing all 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.

Test your assumptions

While [testing your documentation \(page 150\)](#), recognize that what may seem clear to you may be confusing to your users. All documentation builds on assumptions that may or may not be shared with your audience.

Assumptions in developer docs

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.

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.

For example, does a user know how to *clear their cache*, or update *Flash*, or ensure the *JRE* is installed, or clone a git repository? Do the users know how to open a *terminal*, *deploy* a web app, import a *package*, *cd* on the command line, 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.

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, they might seem perfectly clear to you. 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*.

Step 3: Test the instructions against an audience

Almost no developer can push out their code without running it through QA, but for some reason technical writers are usually exempt from QA processes. There are some cases where tech docs are "tested" by QA, but whenever this happens I usually get the 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, a "quality assurance" process in the most literal sense of the word.

Strangely, few IT shops actually have a consistent structure for this doc-quality-assurance role. You wouldn't dream of setting up an IT shop without a quality assurance group for developers, but few technical writers have access to a dedicated editor or to a usability group to ensure quality.

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 grammar — blogs and YouTube are proof of that. But users can't overlook instructions that don't work, that don't speak to the real steps and challenges they face.

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. But the editor there 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 problems.

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 least, you can ask a colleague to try out the instructions.

Ask a colleague to try out your instructions

Other technical writers are usually both curious and generous when you ask them to try out 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 (those 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 lacking. *I got confused right here because I couldn't find the X button, and I didn't know what Y meant,* they'll say. They know what you need to hear.

In general, it's always good to have a non-expert test something rather than an expert, because experts can often compensate for shortcomings in documentation with their own expertise. 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 is more than highlighting shortcomings about poor instruction. You may bring up matters of style, or you might foster great team discussions through 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 that feature is in a real context.

Should you watch when users test?

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 stupid when they're following what should be relatively simple 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 it when users actually share their thoughts in a running monologue. They'll 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 a gotomeeting screenshare. This can allow you to give the user some privacy while also watching them directly.

In my documentation projects, I'm sorry to admit that I've veered far away from usability testing. It's been years since I've actually tested my documentation this way, despite the eye-opening benefits I get when I do it. (Writing about it now, I'm making serious plans to mend my ways.)

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 consider that feedback and potential bugs to log against the documentation. (*And if you don't hear anything, then the doc must be great, right?*)

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 any kind of quality assurance process is to ensure users get a quality product.

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. I particularly hate it when the interface has typos or misspellings that I have to follow in documentation commands just to keep the two in sync. (For example, "Click the **Multi tenancy** button.")

Enjoyment benefits from testing

One of the main benefits to testing is that it makes writing documentation much more enjoyable. 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, that's when things become interesting. Actually, the more you learn about the knowledge domain itself, the work of technical writing increases dramatically.

In contrast, if you just stick to technical editing, formatting, publishing, and curating — these activities are all worthwhile, but they are not fulfilling as a career. 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 start to become engaging.

Accounting for the necessary time

Note well that it takes time to try out the instructions yourself and with users. It probably doubles or triples the documentation time. You don't always have this time before release. I can't say that I've tested out all the different parts of my documentation, because I have four different programming languages across various platforms, but for the doc that I do test, it makes a world of difference.

One way to shorten the testing period is by leveraging the test scripts used by your QA team. These test cases often give you a clear picture about the functionality provided by the system, along with sample calls to see if each piece works. QA scripts are usually much more thorough than you need, but they're helpful in pointing you in the right direction.

Documenting non-reference sections overview

In this section:

- [Creating the user guide \(page 161\)](#)
- [Writing the overview section \(page 163\)](#)
- [Writing the Getting Started section \(page 165\)](#)
- [Documenting authentication and authorization \(page 169\)](#)
- [Documenting response and error codes \(page 176\)](#)
- [Documenting code samples and tutorials \(page 181\)](#)
- [Creating the quick reference guide \(page 185\)](#)

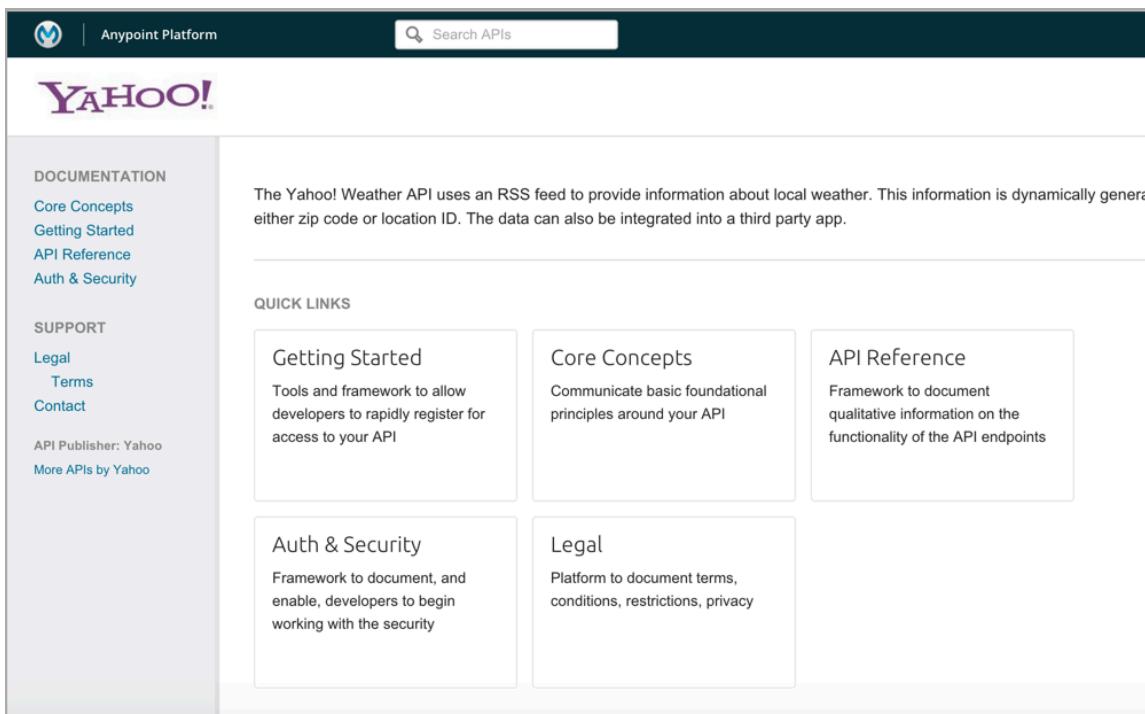
Creating the user guide

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 has a light background. On the left, there's a sidebar with '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 has a heading 'YAHOO!' and a paragraph about the API using an RSS feed. Below that is a section titled 'QUICK LINKS' with four boxes: 'Getting Started' (Tools and framework to allow developers to rapidly register for access to your API), 'Core Concepts' (Communicate basic foundational principles around your API), 'Auth & Security' (Framework to document, and enable, developers to begin working with the security), and 'Legal' (Platform to document terms, conditions, restrictions, privacy).

(<http://api-portal.anypoint.mulesoft.com/yahoo/api/yahoo-weather-api>)

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](https://sendgrid.com/docs) (<https://sendgrid.com/docs>) 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

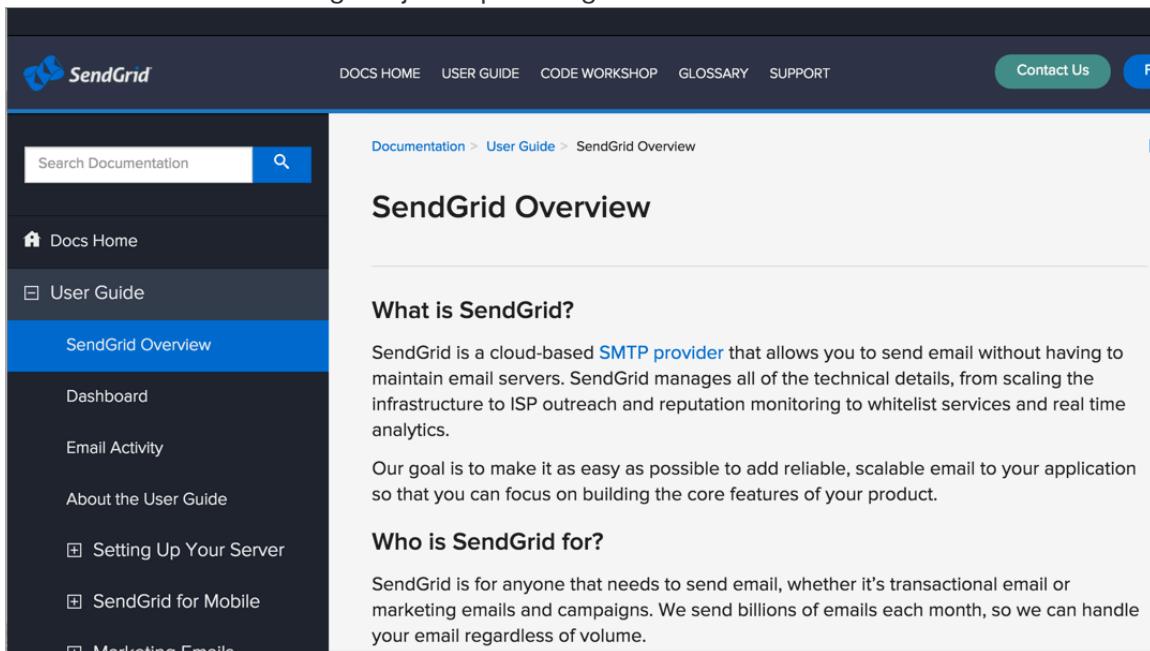
Provide how-to's for these tasks just like you would with any user guide. Seeing the tasks users can do with an API may be a little less familiar because you don't have a GUI to click through. But the basic concept is the same — ask what will users want to do with this product, what can they do, and how do they do it.

Writing the overview section

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

Sample overview

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



The screenshot shows the SendGrid User Guide Overview page. The left sidebar has a dark background with white text. It includes a search bar, a navigation menu with 'Docs Home' and 'User Guide' sections, and a detailed 'SendGrid Overview' section which is currently selected and highlighted in blue. The main content area has a light background. At the top, it shows the breadcrumb path: Documentation > User Guide > SendGrid Overview. Below this, the title 'SendGrid Overview' is displayed. Under the title, there are two sections: 'What is SendGrid?' and 'Who is SendGrid for?'. The 'What is SendGrid?' section describes SendGrid as a cloud-based SMTP provider that manages email infrastructure. The 'Who is SendGrid for?' section states that SendGrid is for anyone needing reliable, scalable email.

(https://sendgrid.com/docs/User_Guide/index.html)

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.

Writing the 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](#)

(<http://www.alchemyapi.com/developers/getting-started-guide>)

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

The screenshot shows the HipChat API Documentation homepage. The left sidebar contains a navigation menu with sections like Overview, Getting started, Authentication, Webhooks, Title expansion, Rate limiting, Response codes, Integrations, and various API endpoints (CAPABILITIES API, EMOTICONS API). The main content area features several sections: 'Getting started' (with a welcome message), 'Supported Platforms' (listing HipChat.com and HipChat Server), 'Build your own integrations' (with instructions and a link to 'Send a message to a HipChat room'), and 'Develop an independent add-on' (with a note about building with others and a link to 'Download a development environment'). A sidebar on the right displays the 'API changelog' with a button to 'Follow' and links to 'Pages Tracked: 76' and 'Latest Changes: 33'.

(<https://www.hipchat.com/docs/apiv2>)

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, 'WEATHER' text, and links for 'CONSUME', 'DEVELOP', 'VISUALIZE', and 'MANAGE'. Below the header, a breadcrumb trail reads 'HELP CENTER / DOCS / AERIS WEATHER API / GETTING STARTED'. To the left is a sidebar with a blue header '① Aeris Weather API' containing 'Getting Started', 'Reference', and 'Downloads' links. The main content area has a large title 'Getting Started'. It includes a paragraph about needing an active 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.

(<http://www.aerisweather.com/support/docs/api/getting-started/>)

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 navigation bar with the SmugMug logo and 'PHOTO SHARING' text. The main content area features a large title 'Welcome to the SmugMug API!'. It includes a paragraph about the beta program and a list of steps: Request a new API key, Get beta program access for your existing API keys, Documentation for the stable API v1.3.0, and API v2 tutorial. Below this, there are sections for 'What is the SmugMug API?', 'Learn the SmugMug API', and a green button labeled 'First step: Getting an API Key'. On the left side, there's a sidebar with sections for 'SmugMug API' (Beta), 'Tutorial' (with links to API Key, First API Request, Making changes, Results in pages, Authorization with OAuth 1.0a, Examples), 'API Concepts' (with links to HTTP Methods, Object Identifiers, Creating Objects, Status Codes), and 'Advanced Topics'.

(<https://api.smugmug.com/api/v2/doc>)

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

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 bG9sOnNlY3VyZQ==
```

APIs that use Basic Auth will also use HTTPS, which means the message content will be encrypted within the HTTP transport protocol. (Without HTTPS, it would be easy for people to decode the username and password.)

When the API server receives the message, it decrypts the message and examines the header. After decoding the string and analyzing the username and password, it then decides whether to accept or reject the request.

In Postman, you can configure Basic Authorization 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.

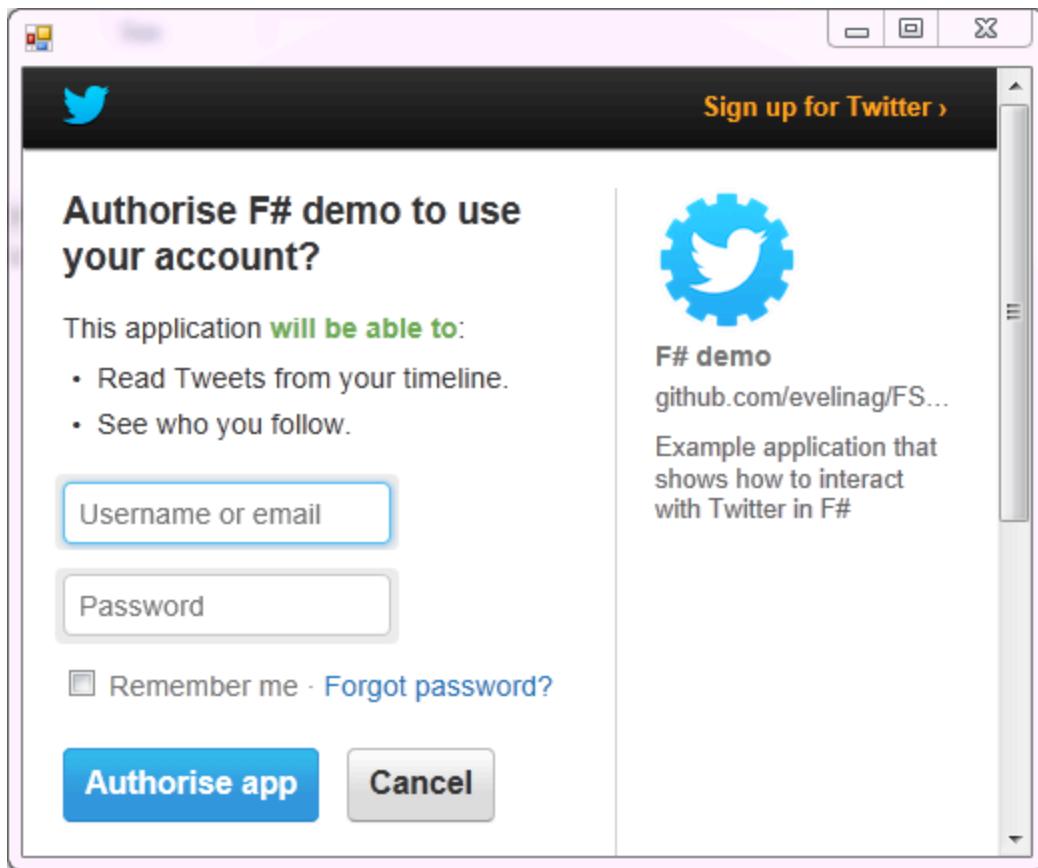
Here's a diagram depicting this workflow:

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:

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](https://www.udemy.com/learn-api-technical-writing-2-rest-for-writers/) (<https://www.udemy.com/learn-api-technical-writing-2-rest-for-writers/>)
- OAuth simplified (<https://aaronparecki.com/articles/2012/07/29/1/oauth2-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, and SUPPORT, along with a 'Contact Us' button. The left sidebar has a search bar and a navigation menu under 'User Guide' 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' section, which explains what API Keys are used for, defines 'Name', 'API Key ID', and 'Action', and provides instructions for creating an API key. A note at the bottom states that the name of the API key follows you through the SendGrid customer portal.

Documentation > User Guide > Settings > API Keys

API Keys

API Keys are used by your application, mail client, or website to authenticate to SendGrid. They are ideal over a username and password, because you can revoke an API key at any time without changing your username and password. We suggest that you use API keys for connecting to all of SendGrid's services.

Name - The name you defined for your API Key.

API Key ID - The way you would reference your API Key for management through the API (e.g. editing or deleting a key)

Action - Actions you can perform on your API Keys.

Create an API Key

When you click the "Create API Key" button, a window will pop out of the side of the page which will allow you to name your API key. This name will follow your API key around through the SendGrid customer portal, so it is important that you name it in such a way

(https://sendgrid.com/docs/User_Guide/Settings/api_keys.html)

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.
[\(https://developer.github.com/v3/#rate-limiting\)](https://developer.github.com/v3/#rate-limiting)

Documenting response and error codes

Response and error codes are essential for understanding errors in submitting or processing requests.

Sample status code in cURL header

Remember when we submitted the cURL call back in [an earlier lesson \(page 50\)](#)? 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/aqi?lat=37.354108&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: Mon, 08 Jun 2015 14:09:34 GMT
Server: Mashape/5.0.6
X-Powered-By: Express
Content-Length: 3
Connection: keep-alive
```

16

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.

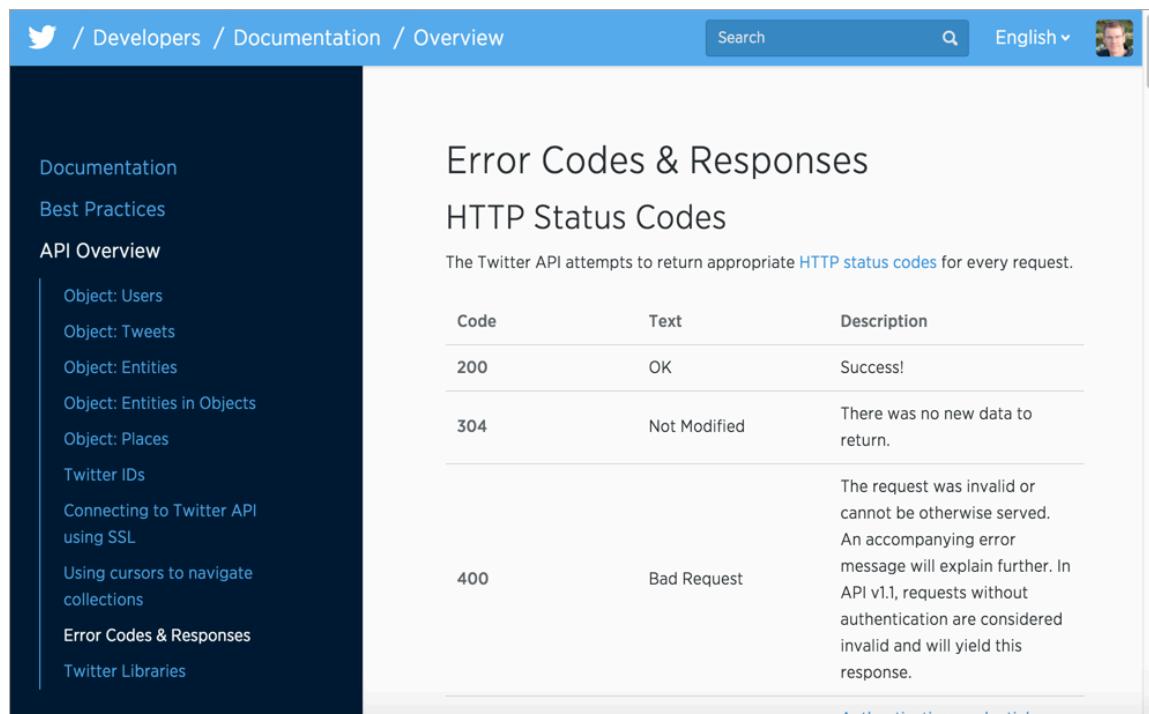
Common status codes follow standard protocols

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 (<http://www.restapitutorial.com/httpstatuscodes.html>) and a general list of HTTP status codes here (http://en.wikipedia.org/wiki/List_of_HTTP_status_codes).

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 Developers Documentation Overview page. On the left, there is a sidebar with links: Documentation, Best Practices, API Overview, Object: Users, Object: Tweets, Object: Entities, Object: Entities in Objects, Object: Places, Twitter IDs, Connecting to Twitter API using SSL, Using cursors to navigate collections, Error Codes & Responses, and Twitter Libraries. The main content area is titled "Error Codes & Responses" and "HTTP Status Codes". It states: "The Twitter API attempts to return appropriate [HTTP status codes](#) for every request." Below this is a table:

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

At the bottom right of the main content area, there is a link: "Authentication credentials".

(<https://dev.twitter.com/overview/api/response-codes>)

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

The screenshot shows a portion of the Flickr API documentation. At the top, there's a navigation bar with links for 'Sign Up', 'Explore', 'Create', 'Upload', and a search bar labeled 'Photos'. Below the navigation, there's a note about standard photo responses. The main content area is titled 'Error Codes' in pink. It 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.

Below the error codes, there's a link to 'API Explorer' in pink, followed by a blue link 'API Explorer : flickr.galleries.getPhotos'.

(<https://www.flickr.com/services/api/flickr.galleries.getPhotos.html>)

Either location has merits, but my preference is a single centralized page for the entire API because listing them out on each endpoint page would add a lot of extra repeated words on each page.

Where to get error codes

Error code 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.

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

(<http://developers.flattr.net/api/resources/things/#update-a-thing>)

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:

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

OAuth 1.0

(<https://www.dropbox.com/developers/core/docs>)

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, somewhat like this:

Status code Meaning

200	Successful request and response.
400	Malformed parameters or other bad request

Status codes aren't readily visible

Status codes are pretty subtle, but when a developer is working with an API, these codes may be the only “interface” the developer has. If you can control the messages the developer sees, it can be a huge win. All too often, status codes are uninformative, poorly written, and communicate little or no helpful information to the user to overcome the error.

Status/error codes can assist in troubleshooting

Status and error codes can be particularly helpful when it comes to troubleshooting. Therefore, you can think of these error codes as complementary to a section on troubleshooting.

Almost every set of documentation could benefit from a section on troubleshooting. Document what happens when users get off the happy path and start stumbling around in the dark forest.

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

- The wrong API keys are used
- Invalid API keys are used
- The parameters don't fit the data types
- The API throws an exception
- There's no data for the resource to return
- The rate limits have been exceeded
- The parameters are outside the max and min boundaries of what's acceptable
- A required parameter is absent from the endpoint

Where possible, document the exact text of the error in the documentation so that it easily surfaces in searches.

Documenting code samples and tutorials

As you write documentation for developers, you'll start to include more and more code samples. You might not include these more detailed code samples with the endpoints you document, but as you create tasks and more sophisticated workflows about how to use the API to accomplish a variety of tasks, you'll end up leveraging different endpoints and showing how to address a variety of scenarios.

Here's a sample code sample page from Mashape:

The screenshot shows a navigation sidebar on the left with links like Firewall, IPs and Security, Access Control, Embed Mashape, API Pricing, Private plans, Unirest, Frequently Asked Questions, and On-Premises. Below these are sections for ADDING AN API (Add your API to Mashape, Define Basic API settings, Documenting an API) and CONSUMING AN API (Consume APIs in JS). The main content area has a title 'Using JavaScript to consume APIs'. It includes a snippet of JavaScript code for making an AJAX request to an API endpoint, followed by a link to the raw GitHub file. Below this, there's a section titled 'Within an HTML page you can use it this way:' with a snippet of HTML code.

Using JavaScript to consume APIs

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.

```
1 $.ajax({  
2   url: 'https://SOMEAPI.p.mashape.com/', // The URL to the API. You can get this in th  
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 y  
10    }  
11  });
```

gistfile1.js hosted with ❤ by GitHub [view raw](#)

Within an HTML page you can use it this way:

```
1 <!DOCTYPE html>  
2 <html>  
3 <head>
```

(<http://docs.mashape.com/javascript>)

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 skill set 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 this [Mashape code sample](#) (<http://docs.mashape.com/javascript>), `dataType` 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.

Creating 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

(<http://api.eventful.com/docs>)

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 is divided into three main vertical sections:

- Liquid** (Left Column):
 - Logic**: Includes sections for {% 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, append, prepend, size, join, downcase, upcase, strip_html, strip_newlines, truncate, truncatewords, date, first, last, newline_to_br, replace, replace_first, and remove.
- 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.

(<http://cheat.markdunkley.com/>)

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).

Exploring other APIs

In this section:

- Exploring more REST APIs (page 188)
- EventBrite example: Get Event information and display it on a page (page 189)
- Flickr example: Retrieve a Flickr gallery and display it on a web page (page 196)
- Klout example: Retrieve Klout influencers and influencees (page 205)
- Aeris Weather Example: Get wind speed and use as conditional value (page 216)
- Next phase of course (page 223)

Exploring more REST APIs

Now it's time to 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.)

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](http://idratherbewriting.com/files/restapicourse/apikeys.txt) (<http://idratherbewriting.com/files/restapicourse/apikeys.txt>).

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 and display it on a page

Use the EventBrite API to get the event title and description of [this event](https://www.eventbrite.com/myevent?eid=17920884849) (<https://www.eventbrite.com/myevent?eid=17920884849>).

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](https://www.eventbrite.com/developer/v3/reference/authentication/) (<https://www.eventbrite.com/developer/v3/reference/authentication/>). Although it's best to pass an Oauth token in the header, for simplicity purposes you can just get a token to make direct calls.

If you want to sign up for your own token, register your app [here](https://www.eventbrite.com/myaccount/apps/) (<https://www.eventbrite.com/myaccount/apps/>). Then copy the "Anonymous access OAuth token."

2. Determine the resource and endpoint you need

The EventBrite API documentation is here: developer.eventbrite.com (<https://www.eventbrite.com/developer/v3/>). Looking through the endpoints available (listed under Endpoints in the sidebar). Which endpoint should we use?

To get event information, we'll use the [event](https://www.eventbrite.com/developer/v3/endpoints/events/) (<https://www.eventbrite.com/developer/v3/endpoints/events/>) object.

[Eventbrite APIv3 Documentation](#) > Events

Events

GET /events/search/ 

Allows you to retrieve a paginated response of public **event** objects from across Eventbrite's directory, regardless of which user owns the event.

Parameters

NAME	TYPE	REQUIRED	DESCRIPTION
q	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 - options are "id", "date", "name", "city", "distance" and "best". Prefix with a hyphen to reverse the order, e.g. "-date"

(<https://www.eventbrite.com/developer/v3/endpoints/events/>)

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](https://www.eventbrite.com/developer/v3/endpoints/events/#ebapi-get-events-id) (<https://www.eventbrite.com/developer/v3/endpoints/events/#ebapi-get-events-id>) 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.

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 "LIVE". The date and time listed are Thursday, September 24, 2015 from 12:00 PM to 1:00 PM (PDT). A navigation bar at the top includes links for "EDIT", "DESIGN", and "MANAGE". On the left, a sidebar menu lists "Event Dashboard", "Order Options" (which is expanded to show "Order Form", "Order Confirmation", "Event Type & Language", and "Waitlist"), and "Live!". The main content area displays the "Event Dashboard" title, a "Live!" status indicator, and a message stating "Your event is up and running and tickets are on sale." There is also a "Public" icon indicating the event is searchable.

(<https://www.eventbrite.com/myevent?eid=17920884849>)

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}>

4. Make a request and analyze the response

Now that you have an endpoint and API token, make the request.

The response from the endpoint is as follows:

```
{  
    "name": {  
        "text": "An Aggressive Approach to Concise Writing, with Joe W  
elinske",  
        "html": "An Aggressive Approach to Concise Writing, with Joe W  
elinske"  
    },  
    "description": {  
        "text": "Webinar Description \nWriting concisely is one of th  
e fundamental skills central to any mobile user assistance. The minima  
l screen real estate can\u2019t support large amounts of text and grap  
hics without extensive gesturing by the users. Using small font sizes  
just makes the information unreadable unless the user pinches and stre  
tches the text. Even outside of the mobile space, your ability to st  
reamline your content improves the likelihood it will be effectively c  
onsumed by your target audience. This session offers a number of exa  
mples and techniques for reducing the footprint of your prose while ma  
intaining a quality message. The examples used are in the context of m  
obile UA but can be applied to any technical writing situation. \nAbou  
t Joe Welinske Joe Welinske specializes in helping your software devel  
opment effort through crafted communication. The best user experience  
features quality words and images in the user interface. The UX of a r  
obust product is also enhanced through comprehensive user assistance.  
This includes Help, wizards, FAQs, videos and much more. For over twen  
ty-five years, Joe has been providing training, contracting, and consu  
lting services for the software industry. Joe recently published the b  
ook, Developing User Assistance for Mobile Apps. He also teaches cours  
es for Bellevue College, the University of California, and the Univers  
ity of Washington. Joe is an Associate Fellow of STC.",  
        "html": "<P><SPAN STYLE=\"font-size: medium;\"><STRONG>Webina  
r Description</STRONG></SPAN></P>\r\n<P>Writing concisely is one o  
f the fundamental skills central to any mobile user assistance. The mi  
nimal screen real estate can\u2019t support large amounts of text and grap  
hics without extensive gesturing by the users. Using small font si  
zes just makes the information unreadable unless the user pinches and stre  
tches the text.<BR> <BR>Even outside of the mobile space, your abi  
lity to streamline your content improves the likelihood it will be eff  
ectively consumed by your target audience.<BR> <BR>This session offer  
s a number of examples and techniques for reducing the footprint of yo  
ur prose while maintaining a quality message. The examples used are i  
n the context of mobile UA but can be applied to any technical writin  
g situation.</P>\r\n<P><SPAN STYLE=\"font-size: medium;\"><STRONG>Abo  
ut Joe Welinske</STRONG></SPAN><BR>Joe Welinske specializes in helpi  
ng your software development effort through crafted communication. Th  
e best user experience features quality words and images in the user i  
nterface. The UX of a robust product is also enhanced through comprehe  
nsive user assistance. This includes Help, wizards, FAQs, videos and m
```

uch 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": "http://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": "2015-07-27T16:19:40Z",
"capacity": 24,
"status": "live",
"currency": "USD",
"shareable": true,
"online_event": false,
"tx_time_limit": 480,
"logo_id": null,
"organizer_id": "7774592843",
"venue_id": "11047889",
"category_id": "102",
"subcategory_id": "2004",
"format_id": "2",
"resource_uri": "https://www.eventbriteapi.com/v3/events/17920884849/",
"logo": null
}

5. Pull out the information you need

The information has a lot more than we need. We just want to display the event's title and description on our site. To do this, we use some simple jQuery code to pull out the information and append it to a tag on our web page:

```
<html>
<body>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

<script>
var settings = {
    "async": true,
    "crossDomain": true,
    "url": "https://www.eventbriteapi.com/v3/events/17920884849/?token=APIKEY",
    "method": "GET",
    "headers": {}
}

$.ajax(settings).done(function (data) {
    console.log(data);
    var content = "<h2>" + data.name.text + "</h2>" + data.description.html;
    $("#eventbrite").append(content);
});
</script>

<div id="eventbrite"></div>

</body>
</html>
```

We covered this approach earlier in the course, so I won't go into much detail here.

My API key is hidden from the above code sample to protect it from unauthorized access.

Here's the result:

An Aggressive Approach to Concise Writing, with Joe Welinske

Webinar Description

Writing concisely is one of the fundamental skills central to any mobile user assistance. The minimal screen real estate can't support large amounts of text and graphics without extensive gesturing by the users. Using small font sizes just makes the information unreadable unless the user pinches and stretches the text.

Even outside of the mobile space, your ability to streamline your content improves the likelihood it will be effectively consumed by your target audience.

This session offers a number of examples and techniques for reducing the footprint of your prose while maintaining a quality message. The examples used are in the context of mobile UA but can be applied to any technical writing situation.

About Joe Welinske

Joe Welinske specializes in helping your software development effort through crafted communication. The best user experience features quality words and images in the user interface. The UX of a robust product is also enhanced through comprehensive user assistance. This includes Help, wizards, FAQs, videos and much more. For over twenty-five years, Joe has been providing training, contracting, and consulting services for the software industry. Joe recently published the book, *Developing User Assistance for Mobile Apps*. He also teaches courses for Bellevue College, the University of California, and the University of Washington. Joe is an Associate Fellow of STC.

Code explanation

The result is as plain-jane 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.

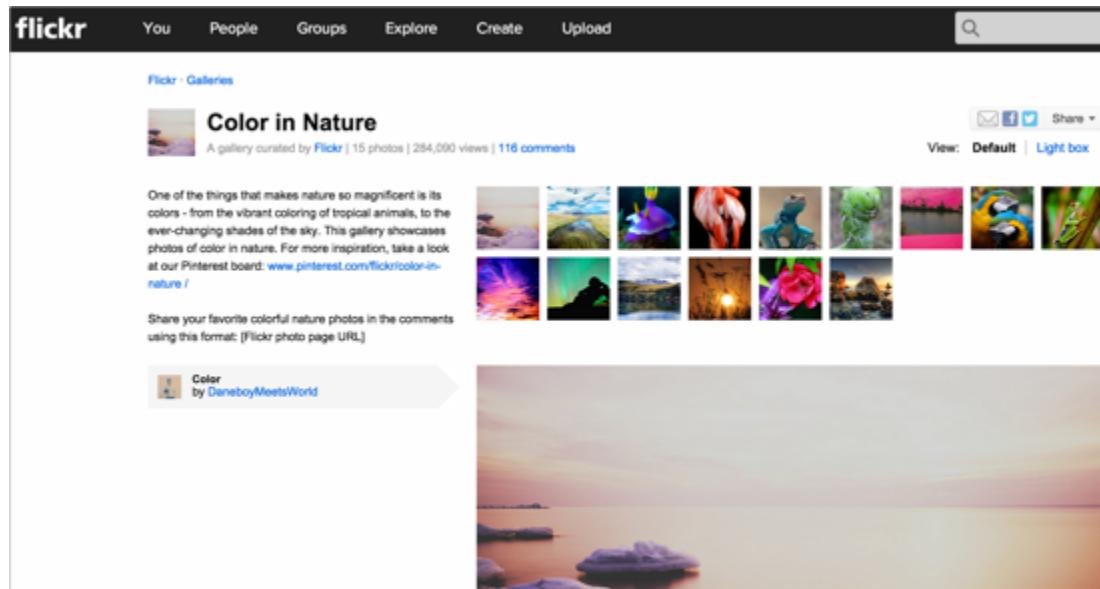
Flickr example: Retrieve a Flickr gallery and display it on a web page

Use the Flickr API to get photo images from this Flickr gallery (<https://www.flickr.com/photos/flickr/galleries/72157647277042064/>).

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 (<https://www.flickr.com/photos/flickr/galleries/72157647277042064/>) and display them on a web page. Here's the gallery we want:



(<https://www.flickr.com/photos/flickr/galleries/72157647277042064/>)

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](https://www.flickr.com/services/apps/create/) (<https://www.flickr.com/services/apps/create/>). 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 (<https://www.flickr.com/services/api/>), the [flickr.galleries.getPhotos](https://www.flickr.com/services/api/flickr.galleries.getPhotos.html) (<https://www.flickr.com/services/api/flickr.galleries.getPhotos.html>) endpoint, which is listed under the galleries resource, is the one that will get photos from a gallery.

The screenshot shows the 'The App Garden' API Documentation. Under the 'galleries' resource, the 'flickr.galleries.getPhotos' endpoint is detailed. It describes the method as returning the list of photos for a gallery, noting that authentication is not required. The arguments section includes 'api_key' (Required), 'gallery_id' (Required), 'extras' (Optional), 'per_page' (Optional), and 'page' (Optional). The URL for this endpoint is provided as [\(https://www.flickr.com/services/api/flickr.galleries.getPhotos.html\)](https://www.flickr.com/services/api/flickr.galleries.getPhotos.html).

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](https://www.flickr.com/services/api/explore/flickr.urls.lookupGallery) (<https://www.flickr.com/services/api/explore/flickr.urls.lookupGallery>) endpoint listed in the URLs resource section to get the gallery ID from a gallery URL:

The App Garden

Create an App API Documentation Feeds What is the App Garden?

flickr.urls.lookupGallery

Arguments

Name	Required	Send	Value
url	required	<input checked="" type="checkbox"/>	<input type="text" value="https://www.flickr.com/p1"/>

Output: **JSON**

Sign call as tomhenryjohnson with full permissions?

Sign call with no user token?

Do not sign call?

Useful Values

Your user ID:
86824645@N00

Your recent photo IDs:
15755702302 -
15754163565 -
15568292559 -

Your recent photoset IDs:
72157649210564562 - Crystal Spring
72157648581395238 - San Francisco
72157648460412980 - San Francisco ride

Your recent group IDs:

Your contact IDs:
30744708@N00 - katiew
92673622@N00 - dulcelife

```
{ "gallery": { "id": "66911286-72157647277042064", "url": "https://www.flickr.com/photos/colorinnature/66911286-72157647277042064", "title": { "_content": "Color in Nature" }, "description": { "_content": "One of the things that makes nature so magnificent is its..." }}
```

(<https://www.flickr.com/services/api/explore/flickr.urls.lookupGallery>)

The gallery ID is **66911286-72157647277042064**. We now have the arguments we need for the [flickr.galleries.getPhotos](#) (<https://www.flickr.com/services/api/flickr.galleries.getPhotos.html>) 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](#) (<https://www.flickr.com/services/api/explore/flickr.galleries.getPhotos>), 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 photoset 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?

[Back to the flickr.galleries.getPhotos documentation](#)

(<https://www.flickr.com/services/api/explore/flickr.galleries.getPhotos>)

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": "af53e53bf1",
        "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": "8948667145",
        "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": "5a469ddaa2",
        "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:

Page 199

```
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 with the URL https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api_key=acada0ff2bb2747f40e0b22767. The page displays a JSON object representing a gallery of photos. The structure includes a top-level object with properties like 'photos' (containing 'page', 'pages', 'perpage', 'total') and 'photo' (an array containing individual photo objects). Each photo object has properties such as 'id', 'owner', 'secret', 'server', 'farm', 'title', 'ispublic', 'isfriend', 'isfamily', 'is_primary', and 'has_comment'. The JSON is presented in a readable, indented format with syntax highlighting for different data types.

```
{  
  "photos": {  
    "page": 1,  
    "pages": 1,  
    "perpage": 500,  
    "total": 15,  
    "photo": [  
      {  
        "id": "8432423659",  
        "owner": "37107167@N07",  
        "secret": "dd1b834ec5",  
        "server": "8187",  
        "farm": 9,  
        "title": "Color",  
        "ispublic": 1,  
        "isfriend": 0,  
        "isfamily": 0,  
        "is_primary": 1,  
        "has_comment": 0  
      },  
      {  
        "id": "8047948330",  
        "owner": "70121902@N00",  
        "secret": "b0e55d455f",  
        "server": "8450",  
        "farm": 9,  
        "title": "Owens River and Sea Grass",  
        "ispublic": 1,  
        "isfriend": 0,  
        "isfamily": 0  
      }  
    ]  
  }  
}
```

I'm using the [JSON Formatting extension for Chrome \(<https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa?hl=en>\)](https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa?hl=en) 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 \(<https://www.flickr.com/services/api/misc.urls.html>\)](https://www.flickr.com/services/api/misc.urls.html) 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

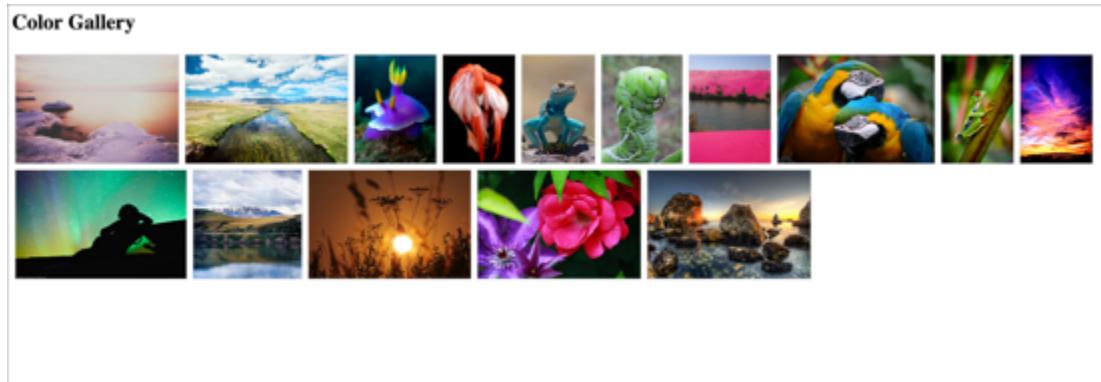
    $("#flickr").append('');
  });
});

</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](http://api.jquery.com/jquery.ajax/) (<http://api.jquery.com/jquery.ajax/>) 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 the photo object. The `title` is accessed through this dot notation: `data.photos.photo[0].title`.
- To get each item in the object, jQuery's [each method](http://api.jquery.com/jquery.each/) (<http://api.jquery.com/jquery.each/>) 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 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`.

- 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: `html // 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.

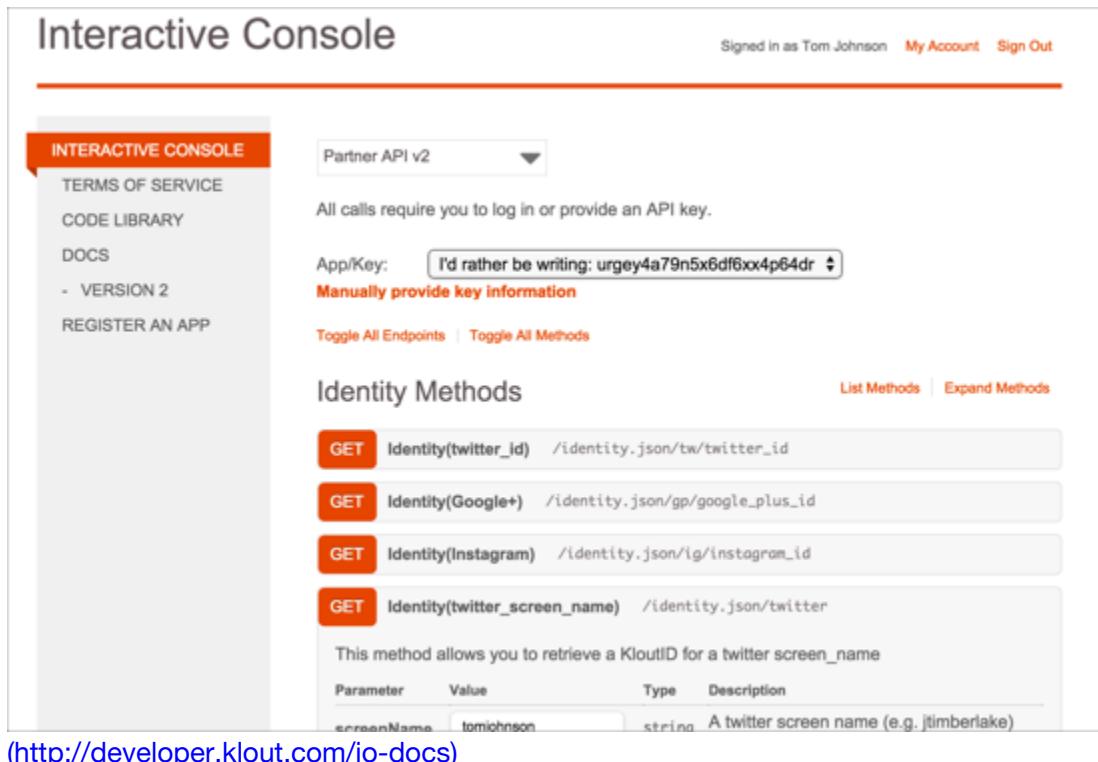
Klout example: Retrieve Klout influencers and influencees

Use the Klout API to get your Klout score and a list of your influencers and influencees.

About Klout

Klout (<http://klout.com>) is a service that gauges your online influence (your klout) by measuring tweets, retweets, likes, etc. from a variety of social networks using a sophisticated algorithm. In this tutorial, you'll use the Klout API to retrieve a Klout score for a particular Twitter handle, and then a list of your influencers.

Klout has an “interactive console” driven by Mashery I/O docs that allows you to insert parameters and go to an endpoint. The interactive console also contains brief descriptions of what each endpoint does.



The screenshot shows the Klout Interactive Console interface. At the top, it says "Interactive Console" and "Signed in as Tom Johnson". Below that, there's a sidebar with links for "INTERACTIVE CONSOLE", "TERMS OF SERVICE", "CODE LIBRARY", "DOCS", and "REGISTER AN APP". The main area shows "Partner API v2" selected in a dropdown. A note says "All calls require you to log in or provide an API key." There's a field for "App/Key" containing "I'd rather be writing: urgey4a79n5x6df6xx4p64dr" with a link to "Manually provide key information". Below that are links for "Toggle All Endpoints" and "Toggle All Methods". Under "Identity Methods", there are four GET requests listed:

Method	Endpoint
GET	/identity.json/tw/twitter_id
GET	/identity.json/gp/google_plus_id
GET	/identity.json/ig/instagram_id
GET	/identity.json/twitter

A note below the first method says "This method allows you to retrieve a KloutID for a twitter screen_name". A table shows a parameter "screenName" with value "tomjohnson" and type "string", with a description "A twitter screen name (e.g. jtimberlake)".

(<http://developer.klout.com/io-docs>)

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](http://developer.klout.com/apps/mykeys) (<http://developer.klout.com/apps/mykeys>) page to register your app and get the keys.

2. Make requests for the resources you need

The API is relatively simple and easy to browse.

To get your Klout score, you need to use the `score` endpoint. This endpoint requires you to pass your Klout ID.

Since you most likely don't know your Klout ID, use the `identity(twitter_screen_name)` endpoint first.

The screenshot shows a request made via the Klout API console. The method is `GET /identity.json/twitter`. The parameter `screenName` is set to `tomjohnson`. The response status is `200 OK`, and the response body contains the Klout ID `1134760`.

Parameter	Value	Type	Description
screenName	tomjohnson	string	A twitter screen name (e.g. jtimmerlake)

Request URI: `http://api.klout.com/v2/identity.json/twitter?screenName=tomjohnson&key=u4r7nd3r7bj9ksxfx3cuy6hw`

Request Headers: `X-Originating-Ip: 24.23.183.203`

Response Status: `200 OK`

Response Headers:

```
Cf-Ray: 21edbfff82cc1ec5-SJC
Content-Type: application/json; charset=utf-8
Date: Tue, 01 Sep 2015 03:04:49 GMT
Server: cloudflare-nginx
X-Apikey-Qps-Allotted: 10
X-Apikey-Qps-Current: 1
X-Apikey-Quota-Allotted: 20000
X-Apikey-Quota-Current: 1
X-Mashery-Responder: prod-j-worker-us-west-1b-26.mashery.com
X-Quota-Reset: Wednesday, September 2, 2015 12:00:00 AM GMT
Transfer-Encoding: chunked
Connection: keep-alive
```

Response Body:

```
{ "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,  
                    "monthChange": 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": "Mrtnlrssn",
            "score": {
                "score": 22.45014953758632,
                "bucket": "20-29"
            },
            "scoreDeltas": {
                "dayChange": -0.3481056157609004,
                "weekChange": -2.132213372307284,
                "monthChange": -2.315034722843535
            }
        }
    }
}
```

```
        "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": "fab_i_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 (`#kloutinfluencees`).

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/influence?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 \(<https://en.wikipedia.org/wiki/JSONP>\)](#).

Aeris Weather Example: Get wind speed and use as conditional value

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](http://www.aerisweather.com/support/docs/api/) (<http://www.aerisweather.com/support/docs/api/>). This is one of the most interesting, well-documented and powerful 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](http://www.aerisweather.com/support/docs/api/getting-started/) (<http://www.aerisweather.com/support/docs/api/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](http://www.aerisweather.com/support/docs/api/reference/endpoints/) (<http://www.aerisweather.com/support/docs/api/reference/endpoints/>) and look for one that would give you the wind speed. The [forecasts](http://www.aerisweather.com/support/docs/api/reference/endpoints/forecasts/) (<http://www.aerisweather.com/support/docs/api/reference/endpoints/forecasts/>) resource provides information about wind speed.

The screenshot shows the Aeris Weather API documentation page for the 'forecasts' endpoint. The top navigation bar includes links for CONSUME, DEVELOP, VISUALIZE, MANAGE, SUPPORT, and DOCUMENTATION. The left sidebar has a 'Getting Started' section with links to Authentication, Responses, Advanced Queries, Sorting, Reducing Output, and Batch Requests. The main content area title is 'Endpoint: forecasts'. It describes the endpoint as providing core forecast data for US and international locations at various intervals. Below this is a redacted URL: <http://api.aerisapi.com/forecasts/>. A horizontal navigation bar below the URL includes links for ACTIONS, PARAMETERS, FILTERS, EXAMPLES, RESPONSE, and PROPERTIES. The 'Supported Actions' section is highlighted with a blue underline. It lists supported actions: [\(http://www.aerisweather.com/support/docs/api/reference/endpoints/forecasts/\)](http://www.aerisweather.com/support/docs/api/reference/endpoints/forecasts/).

To get the forecast details for Santa Clara, California, add it after `/forecasts`, 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 T
01830139",
    "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 then 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%20clar
a,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:



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.

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 APIs overview

In this section:

- [Publishing API docs \(page 225\)](#)
- [List of about 100 APIs \(page 228\)](#)
- [Breaking down API doc \(page 232\)](#)
- [Tool decisions \(page 236\)](#)
- [Github wikis \(page 242\)](#)
- [More about Markdown \(page 249\)](#)
- [Version control systems \(page 257\)](#)
- [Pull request workflows through Github in the browser \(page 266\)](#)
- [REST API specification formats \(page 272\)](#)
- [Implementing Swagger \(OpenAPI specification\) with your REST API documentation \(page 273\)](#)
- [Swagger tutorial \(page 285\)](#)
- [More about YAML \(page 296\)](#)
- [RAML tutorial \(page 301\)](#)
- [API Blueprint tutorial \(page 313\)](#)
- [Static site generators \(page 325\)](#)
- [Readme.io \(page 338\)](#)
- [Miredot \(page 342\)](#)
- [Custom UX solutions \(page 345\)](#)
- [Help authoring tools \(page 347\)](#)
- [Tools versus content \(page 350\)](#)

Publishing API docs

In earlier parts of this workshop, we used a simple [Weather API from Mashape](https://www.mashape.com/fyhao/weather-13) (<https://www.mashape.com/fyhao/weather-13>) to demonstrate how to use a REST API.

Now we'll explore various tools to publish information from the 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 other kinds of documentation that it merits its own section? How and why does the approach here 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 into developer and 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](http://www.programmableweb.com/apis/directory) (<http://www.programmableweb.com/apis/directory>), 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 ProgrammableWeb 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 outright rejected tech comm tools (and their 50 years of evolution)?

Granted, there is the occasional HAT, as with [Photobucket's API](https://pic.photobucket.com/dev_help/WebHelpPublic/Content/PB%20API%20Introduction.htm) (https://pic.photobucket.com/dev_help/WebHelpPublic/Content/PB%20API%20Introduction.htm), but they're rare. And I've not yet found an API doc site that structures all content in DITA.

I asked a recruiter (who specializes in API documentation jobs) 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 this space.

My recruiter friend knows the market — especially the Silicon Valley market — extremely well. He urged me to look at the static site generator route. He said that many small companies, especially startups, are looking for writers who can publish documentation that looks beautiful, like the many modern web outputs on Programmableweb.

Five reasons why developer doc doesn't use HATs

I think there are at least five reasons why developers reject tech comm authoring tools:

1. The HAT tooling doesn't match developer workflows and environments

If devs are going to contribute or write docs, the tools need to fit their own development tools and workflows. Their tooling is to treat doc as code, committing it to source control, building outputs from the server, etc. They want to package the doc in with their other code, check it into their repos, and include it in the builds.

Why are engineers writing 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, 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 simply generate the doc from those annotations. They've been doing that with Java and C++ doc for the past 20 years. There are quite a few tools in the developer doc space that will auto-generate documentation from source code annotations, but it's not something that HATs or GUI doc tools do.

3. API doc follows a specific structure and pattern not modeled in any HAT

The reference documentation is pushed into well-defined templates, which list sections such as endpoint parameters, sample requests, sample responses, and so forth. Sometimes this template can be driven from the source code itself.

If you have a lot of endpoints, you need a system for pushing the content into these templates. There are many templating frameworks that handle these scenarios nicely. Other times you need custom scripts. Either way, not many HATs handle this kind of template-driven publishing scenario.

4. Many APIs have interactive API consoles, allowing you to try out the calls.

You won't find an interactive API console in a HAT. By interactive API console, I mean you enter your own API key and values, and then run the call directly in the documentation. The response you see is from your own data in the API.

5. With APIs, the doc *is* the interface, so it has to be sexy enough to sell the product.

Most output from HATs look dated and old. They look like a relic of the pre-2000 Internet era.

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 in the developers using it.

A new direction

Based on all of these factors, I decided to put DITA authoring on pause and try a new tool with my documentation: Jekyll. 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 since my company is somewhat small, and I'm one of three writers, I wasn't burned by a ton of legacy content or heavy processes, so I could innovate.

Jekyll is just one documentation publishing option in this space. I personally enjoy working with a more code-based approach, but there are many different options and routes to explore.

Now 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.

List of about 100 APIs

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 (<http://programmableweb.com>). I initially started gathering a list of the APIs in Programmableweb's "Most Popular" category, but then I just started adding links as I ran across interesting APIs.

1. Google Places API (<https://developers.google.com/places/webservice/intro>)
2. Twitter API (<https://dev.twitter.com/rest/public>)
3. Flickr API (<https://www.flickr.com/services/api/>)
4. Facebook API (<https://developers.facebook.com/docs/atlas-api/reference/gettingstarted>)
5. Youtube API (<https://developers.google.com/youtube/v3/>)
6. eBay API (<https://go.developer.ebay.com/api-documentation>)
7. Amazon API (<https://developer.amazon.com/appsandservices/apis>)
8. Twilio API (<https://www.twilio.com/docs/api>)
9. Last.fm API (<http://www.last.fm/api>)
10. Bing API (<http://www.bing.com/dev/>)
11. Delicious API (<https://delicious.com/developers>)
12. Google Cloud API (<https://cloud.google.com/appengine/docs>)
13. Foursquare API (<https://developer.foursquare.com/>)
14. Google Data API (<https://developers.google.com/gdata/>)
15. Dropbox API (<https://www.dropbox.com/developers/core/docs>)
16. Splunk API (<http://dev.splunk.com/restapi>)
17. Flattr API (<http://developers.flattr.net/api>)
18. DocuSign API (<https://www.docusign.com/developer-center/documentation>)
19. Geonames (<http://www.geonames.org/export/web-services.html>)
20. Adsense API (<https://developers.google.com/adsense/management/>)
21. Box API (<https://developers.box.com/>)
22. Amazon API (<http://docs.aws.amazon.com/AWSEC2/latest/APIReference>Welcome.html>)
23. Linkedin API (<https://developer.linkedin.com/>)
24. Instagram API (<https://instagram.com/developer/>)
25. Yahoo BOSS API (<https://developer.yahoo.com/boss/search/>)
26. Yahoo Social API (https://developer.yahoo.com/social/rest_api_guide/index.html)
27. Google Analytics API (<https://developers.google.com/analytics/devguides/config/>)
28. Yelp API (<https://www.yelp.com/developers/documentation>)
29. Panoramio API (<http://www.panoramio.com/api/widget/api.html>)
30. Facebook API (<https://developers.facebook.com/docs/graph-api>)
31. Eventful API (<http://api.eventful.com/docs>)
32. Concur API (<https://developer.concur.com/docs-and-resources/documentation>)
33. Paypal API (<https://developer.paypal.com/docs/api/>)

34. Bitly API (<http://dev.bitly.com/>)
35. Hostip API (<http://www.hostip.info/use.html>)
36. Reddit API (<http://www.reddit.com/dev/api>)
37. Netvibes API (<https://uwa.netvibes.com/docs/Uwa/html/index.html>)
38. Rhapsody API (<https://developer.rhapsody.com/>)
39. Donors Choose (<http://data.donorschoose.org/docs/overview/>)
40. Sendgrid API (https://sendgrid.com/docs/API_Reference/index.html)
41. Photobucket API (<http://bit.ly/1rMDb5b>)
42. Mailchimp (<http://kb.mailchimp.com/>
[api/?utm_source=apidocs&utm_medium=internal_ad&utm_campaign=api_v3](#))
43. Basecamp API (<https://github.com/basecamp/bcx-api>)
44. Smugmug API (<https://smugmug.atlassian.net/wiki/display/API/Home>)
45. NYTimes API (http://developer.nytimes.com/docs/read/article_search_api_v2)
46. USPS API (<https://www.usps.com/business/web-tools-apis/track-and-confirmed-api.htm>)
47. NWS API (http://www.nws.noaa.gov/mdl/survey/pgb_survey/dev/rest.php)
48. Evernote API (<https://dev.evernote.com/doc/>)
49. Stripe API (<https://stripe.com/docs/api>)
50. Parse API (<https://parse.com/docs/rest/guide>)
51. Opensecrets API (<https://www.opensecrets.org/resources/create/apis.php>)
52. Compete API (<https://developer.compete.com/>)
53. CNET API (<http://api.cnet.com/dashboard.html>)
54. Amazon API (<http://docs.aws.amazon.com/AlexaWebInfoService/latest/>)
55. Hoiio API (http://developer.hoiio.com/docs/voice_call.html)
56. Citygrid API (<http://docs.citygridmedia.com/display/citygridv2/CityGrid+APIs>)
57. Mapbox API (<https://www.mapbox.com/developers/api/>)
58. Groupon API (<https://www.groupon.com/pages/api>)
59. AddThis Menu API (<http://support.addthis.com/customer/portal/articles/381262-addthis-api-and-sdks>)
60. Yahoo Weather API (<https://developer.yahoo.com/weather/>)
61. SimplyHired API (<http://www.simplyhired.com/a/publishers/overview>)
62. Crunchbase API (<http://data.crunchbase.com/>)
63. Zendesk API (https://developer.zendesk.com/rest_api/docs/core/introduction)
64. nDango API (<http://apidocs.ondango.com/rest/sales/get.php>)
65. Ninja Blocks API (<http://docs.ninja.is/rest/user.html>)
66. Pushover API (<https://pushover.net/api>)
67. Pusher API (<https://pusher.com/docs>)
68. Pingdom API (<https://www.pingdom.com/resources/api>)
69. Daily Mile API (<http://www.dailymile.com/api/documentation#streams>)
70. Jive (<https://developers.jivesoftware.com/api/v3/cloud/rest/>)
71. IBM Watson (uses Swagger) (<http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/apis/>)
72. HipChat API (<https://www.hipchat.com/docs/apiv2>)
73. Stores API (<https://developer.bigcommerce.com/api/>)

74. Alchemy API (<http://www.alchemyapi.com/api/>)
75. Indivo API 1.0 (<http://docs.indivohealth.org/en/latest/api.html>) and Indivo API 2.0 (<http://docs.indivohealth.org/en/2.0/api.html>) on readthedocs platform
76. Socrata API (<http://dev.socrata.com/>)
77. Github API (<https://developer.github.com/>)
78. Mailgun API (https://documentation.mailgun.com/api_reference.html)
79. RiotGames API (<https://developer.riotgames.com/api/methods>) example of Swagger
80. Basecamp API (<https://github.com/basecamp/bcx-api>) example of Github
81. UserApp API (<https://app.userapp.io/#/docs/>)
82. Kimono Labs API (<https://www.kimonolabs.com/apidocs>)
83. SwiftType API (<https://swiftype.com/documentation/>)
84. Snipcart API (<http://docs.snipcart.com/api-reference/introduction>)
85. VHX API (<http://dev.vhx.tv/docs/api/>)
86. Polldaddy API (<http://support.polldaddy.com/api/>)
87. Gumroad API (<https://gumroad.com/api>)
88. Formstack API (<http://developers.formstack.com/>)
89. Livefyre API (<http://answers.livefyre.com/developers/api-reference>)
90. Salesforce Chatter REST API (<https://developer.salesforce.com/docs/atlas.en-us.chatterapi.meta/chatterapi/>)
91. Rotten Tomatoes API (<http://developer.rottentomatoes.com/docs>)
92. Github (<https://developer.github.com/v3/issues/comments/#create-a-comment>)
93. Context.io (https://context.io/docs/lite/users/email_accounts)

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](http://www.programmableweb.com/apis/directory) (<http://www.programmableweb.com/apis/directory>). You can browse more than 13,000 web APIs.

The screenshot shows the ProgrammableWeb website's API directory search interface. At the top, there is a navigation bar with links for Follow, +Follow, Share, Sign In/Sign Up, API News, API Directory, For API Providers, For Developers, Listings, and Forum. Below the navigation bar, the main title "ProgrammableWeb: the world's largest API repository, GROWING DAILY" is displayed. A search bar contains the placeholder "Search Over 13,581 APIs" and a "Search APIs" button. Below the search bar, there is a "Filter APIs" section with three buttons: "By Category", "By Protocols/Formats", and "Include Deprecated APIs". The main content area displays a table of APIs with columns for API Name, Description, Category, and Date. Three entries are listed:

API Name	Description	Category	Date
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. The...	Feeds	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...	Addresses	12.06.2014
WeatherSTEM	WeatherSTEM is a service that is designed to help educators create STEM lessons and activities that incorporate real-world weather. Users can add	Weather	12.06.2014

On the right side of the page, there is a sidebar titled "API Directory Search" which says "Search over 13,581 APIs updated daily". It includes a search bar, a "Browse by Category" link, a "Newest APIs" link, a "Latest Mashups" link, and a green "Add an API +" button. Below the sidebar, there is a section titled "PW Research Center" with a sub-section "Growth In Web APIs Since 2005" featuring a small chart.

(<http://www.programmableweb.com/apis/directory>)

Note that Programmableweb only lists web APIs, meaning APIs that you can access on the web. They don't list the countless internal, firewalled-off APIs that many companies provide at a cost to paying customers. There are many more thousands of firewalled, private APIs out there that most of us will never know about.

Look at 5 different APIs

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

Breaking down API doc

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.

No common tooling

Just as websites have a diversity of engines, platforms, and approaches, so too does API documentation. There is no common tooling like there is among GUI documentation. You can't usually determine what platform is driving the outputs, and often the branding fits in seamlessly with the other company content.

Similar patterns and structures

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.

Three kinds of API doc content

In a [blog post by the writers at Parse](http://blog.parse.com/learn/engineering/designing-great-api-docs/) (<http://blog.parse.com/learn/engineering/designing-great-api-docs/>), 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.</blockquote>

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

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

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

(<http://api-portal.anypoint.mulesoft.com/yahoo/api/yahoo-weather-api?ref=apihub>)

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:



(<https://www.parse.com/tutorials>)

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 for Relationship Endpoints. The left sidebar has a search bar and links to Overview, Authentication, Secure API Requests, Real-time, Mobile Sharing, API Console, and Endpoints. Under Endpoints, there are links for Users, Relationships (which is expanded), Media, Comments, Likes, Tags, Locations, and Geographies. The main content area is titled "Relationship Endpoints" and explains that relationships are expressed using terms like outgoing_status and incoming_status. It lists several API endpoints with their descriptions:

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

Below these, three specific examples are shown in boxes:

- GET /users/ user-id /follows**: Description: Get the list of users this user follows. URL: https://api.instagram.com/v1/users/{user-id}/follows?access_token=ACCESS-TOKEN. Response button.
- GET /users/ user-id /followed-by**: Description: Get the list of users this user is followed by. URL: https://api.instagram.com/v1/users/{user-id}/followed-by?access_token=ACCESS-TOKEN. Response button.
- GET /users/self/requested-by**: Description: Get the list of users who have requested to follow.

(<https://instagram.com/developer/endpoints/relationships/>)

Tool decisions

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, generating doc content from the source code may only prove to be a complicated hassle with little benefit.

Integrating into engineering tools and workflows

Riona Macnamara is a technical writer at Google. Riona 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.

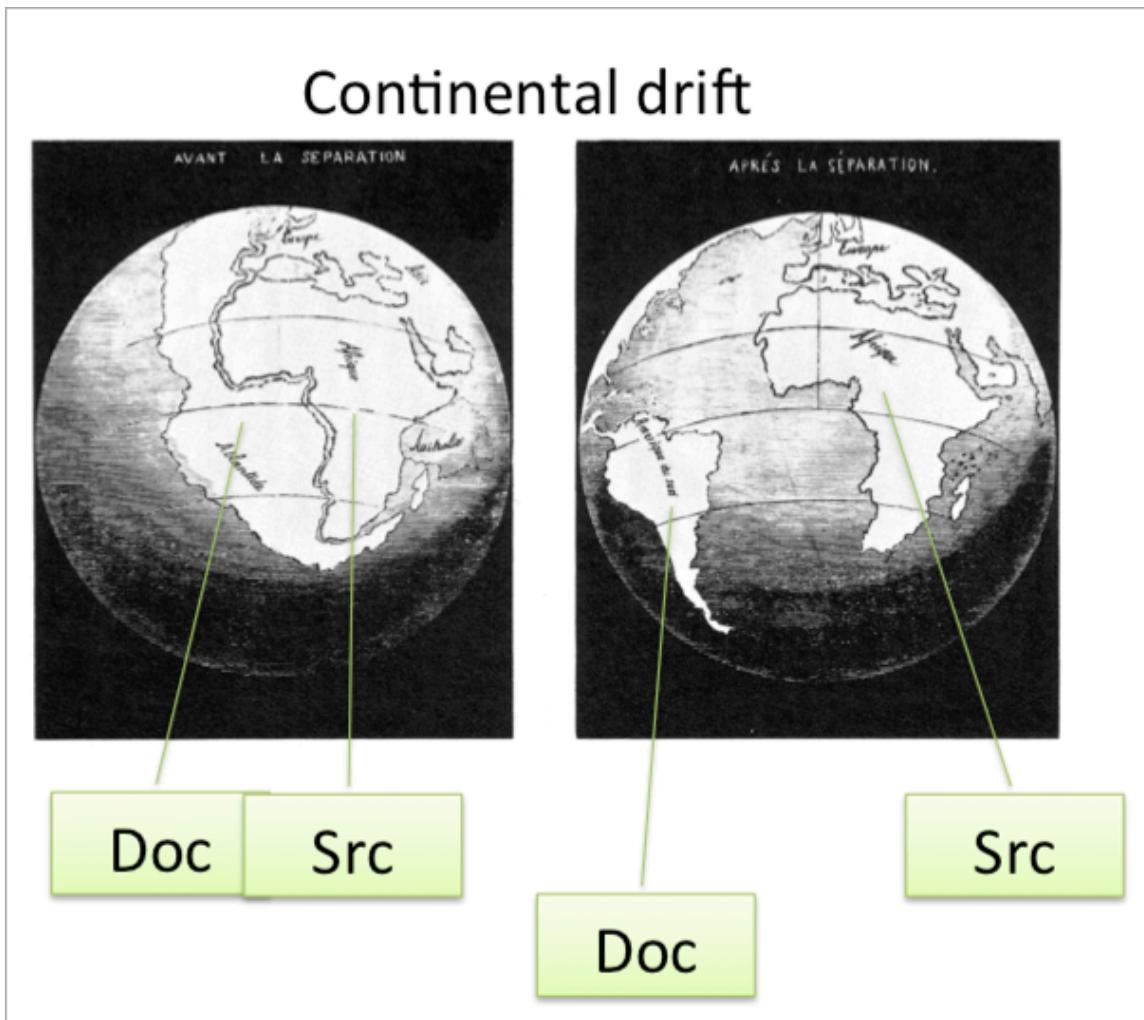
If you plan to have developers write, definitely check out Riona Macnamara's Write the Docs 2015 presentation: [Documentation, Disrupted: How two technical writers changed Google engineering culture](https://www.youtube.com/embed/EnB8GtPuauw) (<https://www.youtube.com/embed/EnB8GtPuauw>).

Pros of having developers write

Having developers write or contribute to documentation provides several advantages.

Avoids documentation drift

By keeping documentation tightly coupled with code, you can avoid documentation drift. The idea is that documentation that exists separate from the documentation has a tendency to get out of sync with the actual code. As developers add new parameters, functions, and other details, the technical writers may not be aware of all these details. But many in-source document generators will actually drive the output directly from the parameters and classes in the code.



(http://en.wikipedia.org/wiki/Continental_drift)

Allows the person who creates the code (and so best understands it) to also document it

Let's face it – sometimes 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.

Sometimes 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.

Cons of having developers write

Here are a few cons in having developers write documentation.

Problem 1: The curse of knowledge

A developer who creates the API may assume too much of the audiences' technical ability. As a result, the descriptions may not be helpful. Steven Pinker explains that the [curse of knowledge](http://idratherbewriting.com/2007/01/24/the-curse-of-knowledge-the-more-you-know-the-worse-communicator-you-become/) (<http://idratherbewriting.com/2007/01/24/the-curse-of-knowledge-the-more-you-know-the-worse-communicator-you-become/>) is one reason why writing is often bad.

The screenshot shows a news article from The Wall Street Journal. At the top, the masthead 'THE WALL STREET JOURNAL.' is visible, followed by a navigation bar with links to Home, World, U.S., Politics, Economy, Business, Tech, Markets, Opinion, and Advertising. Below the navigation, the word 'ESSAY' is written in blue. The main title of the article is 'The Source of Bad Writing' in large, bold, black letters. A subtitle follows: 'The 'curse of knowledge' leads writers to assume their readers know more than they do.' To the left of the text is a portrait photograph of Steven Pinker, a man with curly grey hair, wearing a dark suit and tie. On the right side of the article, there is a vertical sidebar with several small, partially visible images and text snippets, such as 'Stay Mea' and 'Cash v the Rig'.

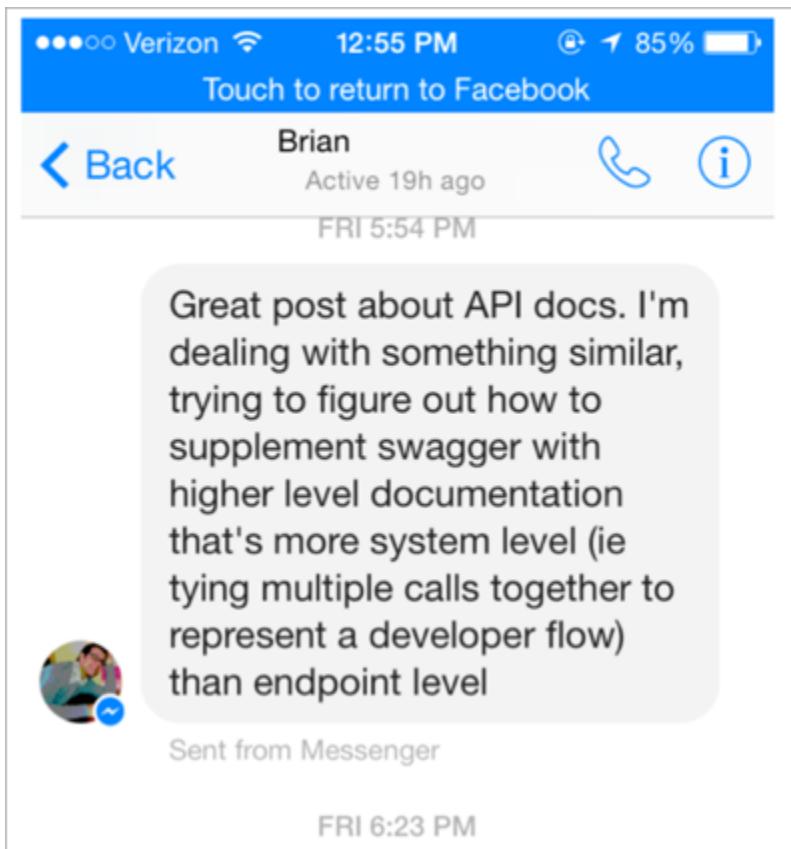
(<http://online.wsj.com/articles/the-cause-of-bad-writing-1411660188>)

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.

Problem 2: Not task-focused

Documentation generated from source files is feature-based. It's the equivalent of writing documentation tab by tab in a GUI interface. In contrast, task-based doc includes multiple calls and workflows in support of goals. Task-based documentation might make use of several different objects and methods across the reference doc.

If developers write the documentation in the source, most likely the result will be somewhat useless feature-based documentation. Here's a text one of my colleagues, a project manager, sent me about the challenges he's facing with documentation:



Capturing and describing the interdependencies, goals, workflows, and other tasks that cut across endpoints and setups is more of a task suited to a technical writer, not a developer who is simply defining a parameter in the source file of a class he or she created.

Problem 3: Output doesn't integrate with user guide doc

Documentation generated from the source doesn't integrate directly into a website except as a link from your other web pages. Like a HAT-produced webhelp file, the auto-doc is its own little website. Here's an example from Netty's documentation that shows how the auto-generated doc is separate from the rest of the site.

The screenshot shows a web browser window for the 'Netty: Home' page at netty.io. The main content area features a large image of a person working on a laptop, with text overlaying it: 'Netty is an asynchronous event-driven framework for rapid development of maintainable network applications such as protocol servers & clients.' To the right of this text is a sidebar with a navigation menu:

- User guide
- Javadoc - 5.0
- Javadoc - 4.1
- Javadoc - 4.0
- Javadoc - 3.9
- All Documents
- Related Articles

Below this menu is a detailed description of Netty's capabilities:

Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server.

'Quick and easy' doesn't mean that a resulting application will suffer from a maintainability or a performance issue. Netty has been designed carefully with the experiences earned from the implementation of a lot of protocols such as FTP, SMTP, HTTP, and various binary and text-based legacy protocols. As a result, Netty has succeeded to find a way to achieve ease of development, performance, stability, and flexibility without a compromise.

Features

Design

- Unified API for various transport types - blocking and non-blocking socket
- Based on a flexible and extensible event model which allows close separation of...

Transport Services

Socket & Datagram	HTTP & WebSockets
HTTP Tunnel	zlib/gzip Compression
In-VM Pipe	Large Message Support

Core

Extensible	Universal Compatibility
Zero-Copy Capabilities	

Having separate outputs creates a somewhat fragmented or disjointed documentation experience. Branding the outputs to create one seamlessly branded site may require a lot of cobbling together and overwriting of stylesheets.

Problem 4: Gives illusion of having complete doc

Finally, when documentation is generated from the source, written by developers, it can give the illusion of documentation. This is something [Jacob Kaplan Moss](http://jacobian.org/writing/what-to-write/) (<http://jacobian.org/writing/what-to-write/>) writes about. He says,

... auto-generated documentation is worse than useless: it lets maintainers fool themselves into thinking they have documentation, thus putting off actually writing good reference by hand. If you don't have documentation just admit to it. Maybe a volunteer will offer to write some! But don't lie and give me that auto-documentation crap".

Auto-generated just means the documentation is generated from code annotations in the source files. If you have an output like this, it may give the idea that you've got all the documentation you need. In reality, the reference documentation is just one part of API documentation. The user guides and tutorials – elements that can't be auto-generated – are just as important as the reference documentation.

Github wikis

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 the GitHub repository page for `basecamp/bcx-api`. The repository has 253 commits, 5 branches, 0 releases, and 23 contributors. The master branch is selected. A merge pull request #177 from `tjchuck/patch-1` is shown, authored by `georgeclaghorn` on Apr 9, fixing some invalid JSON. The README.md file adds documentation for Groups API. The repository contains a single file, `README.md`. The main content area displays the "The new Basecamp API" page, which explains the transition to a new REST-style API using JSON and OAuth 2. The sidebar on the right shows links for Code, Issues (6), Pull requests (0), Wiki, Pulse, and Graphs. It also provides an HTTPS clone URL (`https://github.com/basecamp/bcx-api`) and options to Clone in Desktop or 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](https://help.github.com/articles/github-flavored-markdown/) (<https://help.github.com/articles/github-flavored-markdown/>) allows you to create tables, add classes to code blocks (for proper syntax highlighting), and more.

The wiki repository

The wiki you create is its own repository that you can clone 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:

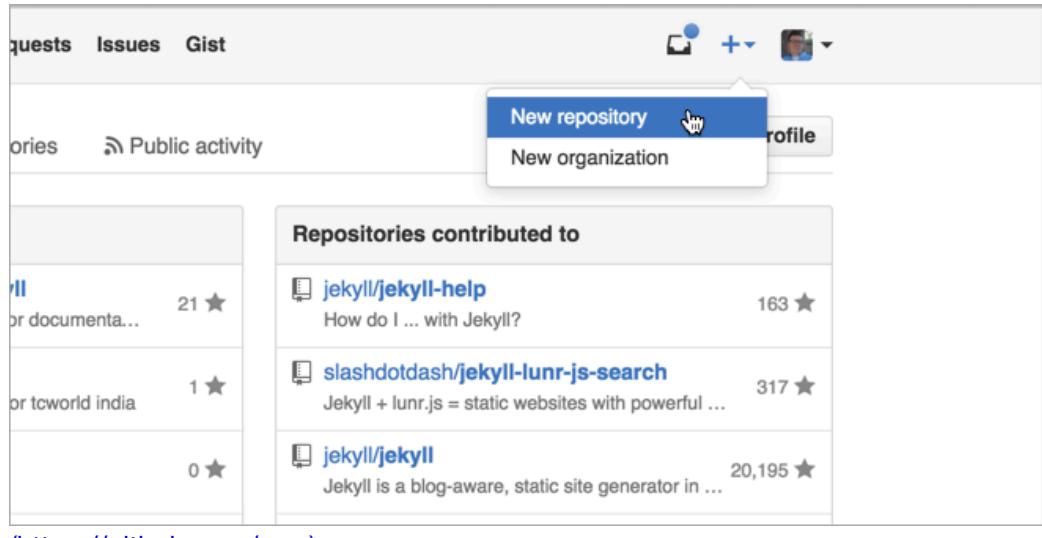
- **Limited branding.** All Github wikis look the same.
- **Open access on the web.** If your docs need to be private, Github isn't the place to put them.
- **No structure.** The Github wiki pages give you a blank page and basically allow you to add sections. You won't be able to do any advanced styling or sexy-looking interactive API doc.

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.

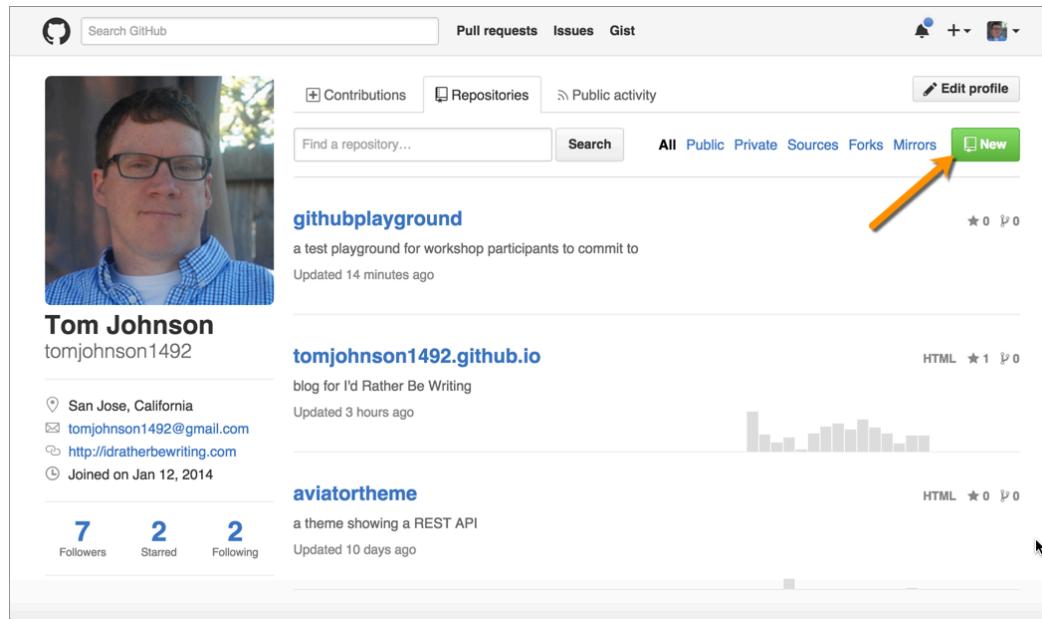
In this section, you'll be using Git commands through your terminal or command prompt. In later tutorials, you'll use the Github Desktop and Github browser tools. Basically, you can interact with Github in a variety of ways.

1. Go to [Github.com \(<http://github.com>\)](http://github.com) and either sign in or create an account.
2. After you're signed in, click the + button in the upper-right corner and select **New repository**.



(<https://github.com/new>)

3. Give the repository a name, description, select **Public**, select to **Initialize the repo with a README**, and then click **Create repository**.



4. Click the **Wiki** link at the top of the repository.
5. Click **Create first page**.
6. Insert your own sample documentation page, preferably using Markdown syntax. Or grab the sample Markdown page of a [fake endpoint called surfreport here](http://idratherbewriting.com/files/restapicourse/surreportendpointdoc.md) (<http://idratherbewriting.com/files/restapicourse/surreportendpointdoc.md>) and insert it into the page.
7. 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 for clone repo link next to the HTTPS button. Copy the link by clicking the **Copy to clipboard** button.

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.

More than just copying the files, though, when you clone a repo, you initialize Git in the cloned folder. Git starts tracking your edits to the files, providing version control. You can run “pull” commands to get updates of the online repository pulled down to your local copy. You can also commit your changes and then push your changes back up to the 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.)
3. In your terminal, either use the default directory or browse 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. The command should look like this:

```
git clone https://github.com/tomjohnson1492/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, you will also see a git folder.

Set up Git and Github authentication

1. Set up Git on your computer.

It's easiest to install Git by [installing Github Desktop](https://desktop.github.com) (<https://desktop.github.com>). 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).

On a Mac, however, you don't need a special Git Shell. Open the Terminal in the same way — go to **Applications > Utilities > Terminal**.

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

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](https://help.github.com/articles/set-up-git/) (<https://help.github.com/articles/set-up-git/>) (Note that when you configure your username, use your Github username, which will be something like `tomjohnson1492` instead of "Tom Johnson".)
- [Generating a new SSH key](https://help.github.com/articles/generating-a-new-ssh-key/) (<https://help.github.com/articles/generating-a-new-ssh-key/>)
- [Adding a new SSH key to the ssh-agent](https://help.github.com/articles/adding-a-new-ssh-key-to-the-ssh-agent/) (<https://help.github.com/articles/adding-a-new-ssh-key-to-the-ssh-agent/>)
- [Adding a new SSH key to your GitHub account](https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/) (<https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/>)

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

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`.

Just use Git to track text 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.

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 command above is a shortcut for committing and typing a commit message in the same command. 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 (<http://www.cs.rit.edu/~cslab/vi.html>).

7. Push the changes to your repository:

```
git push
```

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

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 Grubber, a blogger, first created Markdown (see his [Markdown documentation here](http://daringfireball.net/projects/markdown/) (<http://daringfireball.net/projects/markdown/>)). 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](https://help.github.com/articles/github-flavored-markdown/) (<https://help.github.com/articles/github-flavored-markdown/>), [Multimarkdown](http://fletcherpenney.net/multimarkdown/) (<http://fletcherpenney.net/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 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](http://kramdown.gettalong.org/) (<http://kramdown.gettalong.org/>), 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](https://docs.shopify.com/themes/liquid-documentation/basics) (<https://docs.shopify.com/themes/liquid-documentation/basics>) or [Coffeescript](http://coffeescript.org/) (<http://coffeescript.org/>). 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

`{beachId}` refers to the ID for the beach you want to look up. All Beach ID codes are a

## 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. I | string |
| time | *Optional*. If you include the time, then only the current hour will be returned. | boolean |

## Sample request

```
curl --get --include 'https://simple-weather.p.mashape.com/surfreport/123?units=imperial'
 -H 'X-Mashape-Key: W0yzMuE8c9mshcofZaBke3kw7lMtp1HjVGAjsndqIPbU9n2eET'
 -H 'Accept: application/json'
```

## Sample response

```json
{
 "surfreport": [
 {
 "beach": "Santa Cruz",
 "monday": {
 "1pm": {
 "tide": 5,
 "wind": 15,
 "watertemp": 80,
 "surfheight": 5,
 "recommendation": "Go surfing!"
 },
 "2pm": {
 "tide": -1,
 }
 }
 }
]
}
```

```

 "wind": 1,
 "watertemp": 50,
 "surfheight": 3,
 "recommendation": "Surfing conditions are okay, not great."
 },
 "3pm": {
 "tide": -1,
 "wind": 10,
 "watertemp": 65,
 "surfheight": 1,
 "recommendation": "Not a good day for surfing."
 }
}
]
}
```

```

The following table describes each item in the response.

| Response item | Description |
|--|--|
| **beach** | The beach you selected based on the beach ID in the request. The beach name is returned in the response. |
| **{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 specific time in the request. |
| **{day}/{time}/tide** | The level of tide at the beach for a specific day and time. Tide levels are measured in feet or centimeters. |
| **{day}/{time}/wind** | The wind speed at the beach, measured in knots per hour or kilometers per hour. |
| **{day}/{time}/watertemp** | The temperature of the water, returned in Fahrenheit or Celsius. |
| **{day}/{time}/surfheight** | The height of the waves, returned in either feet or centimeters. |
| **{day}/{time}/recommendation** | An overall recommendation based on a combination of the other items. |

Error and status codes

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

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

Code example

The following code samples shows how to use the surfreport endpoint to get the surf conditions for a specific beach.

```

```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=imperial&days=1&time',
 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 console using console.log(data);
 document.getElementById("output").innerHTML = data.surfreport[0].monday.2pm.recommendation;
 },
 error: function(err) { alert(err); },
 beforeSend: function(xhr) {
 xhr.setRequestHeader("X-Mashape-Authorization", "W0yzMuE8c9mshcofZaBke3kw7lMtp1HjVGA");
 }
 });
 }

</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 requests. For simple demo purposes, the response is assigned to the `data` argument of the success callback.
```

Write some Markdown on a page

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

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

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 interface. At the top, there's a search bar with 'Search GitHub' and navigation links for 'Pull requests', 'Issues', and 'Gist'. Below the search bar is a blue header bar with the text 'GitHub Bootcamp' and a close button ('x'). The main content area is divided into four yellow boxes numbered 1 through 4, each featuring a cartoon GitHub cat character and a brief description:

- 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 tutorial is a user profile for 'tomjohnson1492'. Underneath the profile, there are two recent comments:

- 4 hours ago: envygeeks commented on issue [jekyll/jekyll#3260](#). The comment says: '@kenold please ask your question at <https://talk.jekyllrb.com> since this is not related to a bug.'
- 4 hours ago: kenold commented on issue [jekyll/jekyll#3260](#). The comment says: '@MCMic Thanks for the fix. It works OK with strings. How would I go about getting the last

To the right of the comments is a blue box with the text 'Easier feeds for GitHub Pages' and a link to 'View 56 new broadcasts'. Below the comments is a section titled 'Repositories you contribute to' with two entries:

- [jekyll/jekyll-help](#) 162 ★
- [slashdotteddash/jekyll-lunr-js-search](#) 347 ★

(<http://github.com>)

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

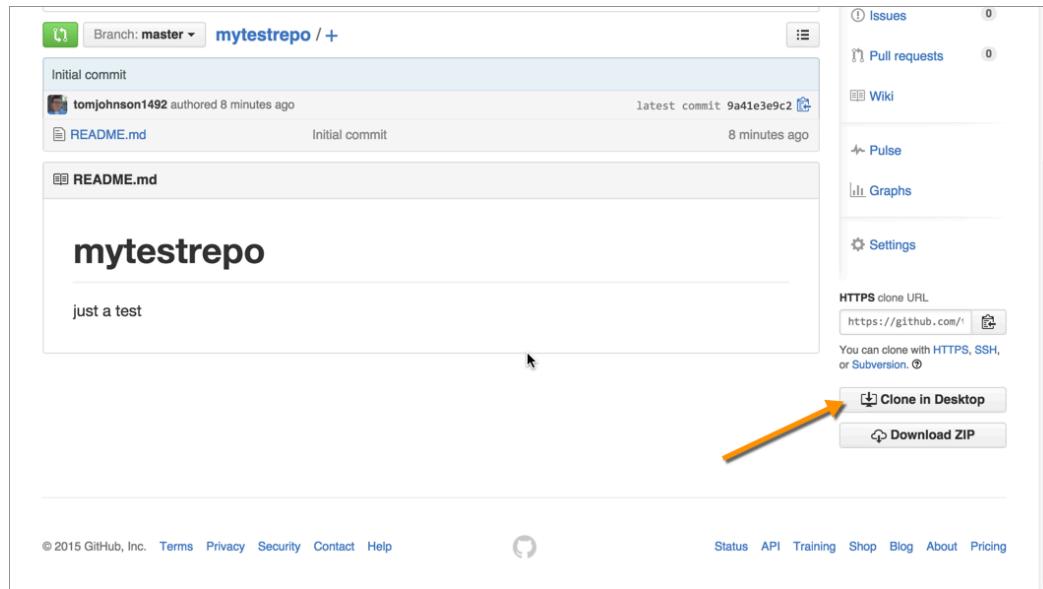
Although most developers use the command line when working with version control systems, there are many GUI clients available that may simplify the whole process. GUI clients might be especially helpful when you’re trying to see what has changed in a file, since the GUI can better highlight and indicate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

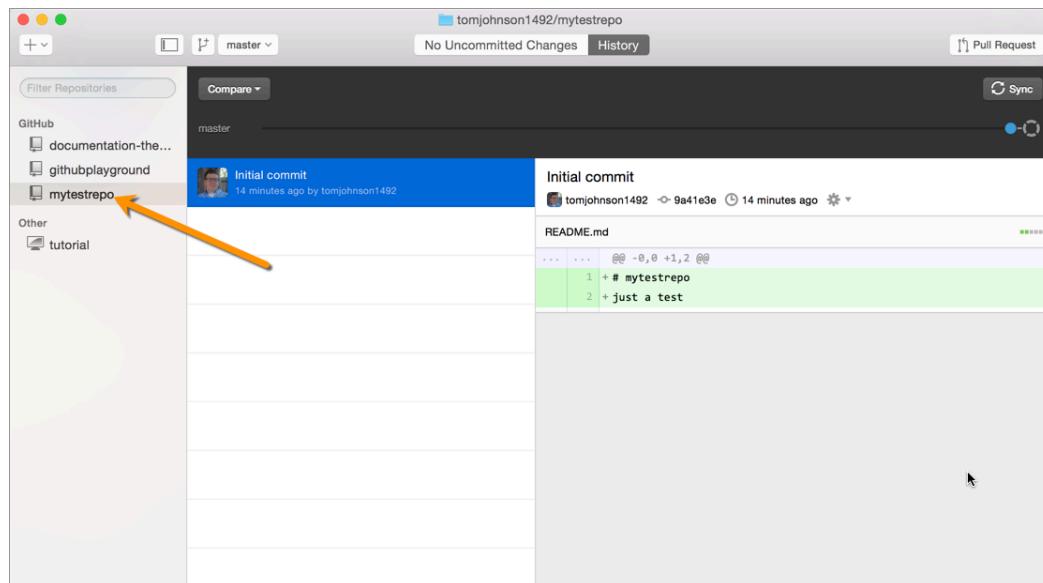
In this tutorial, you’ll use Github Desktop to manage the workflow. First download and install [Github Desktop \(https://desktop.github.com/\)](https://desktop.github.com/). You’ll also need a Github account.

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

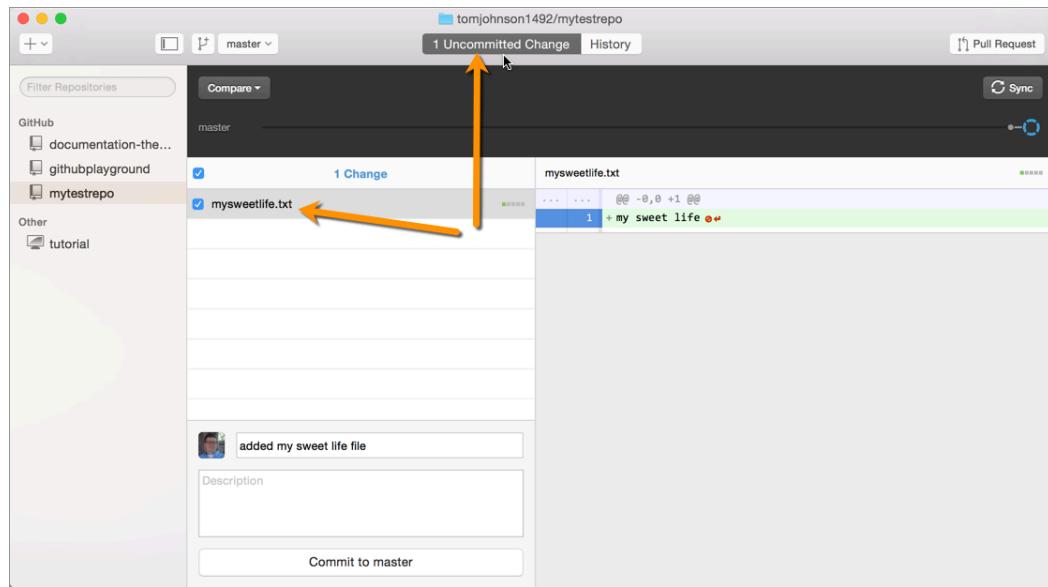


3. Select the folder where you want to clone the repository (such as under your username), and then click **Clone**.

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.



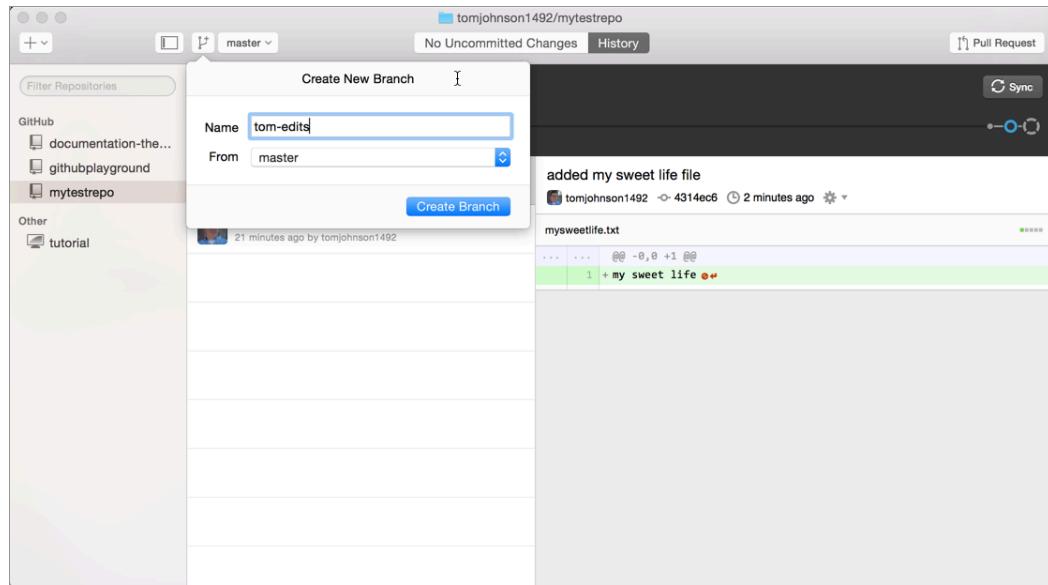
You'll see the new file you added in the list of uncommitted changes.

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.

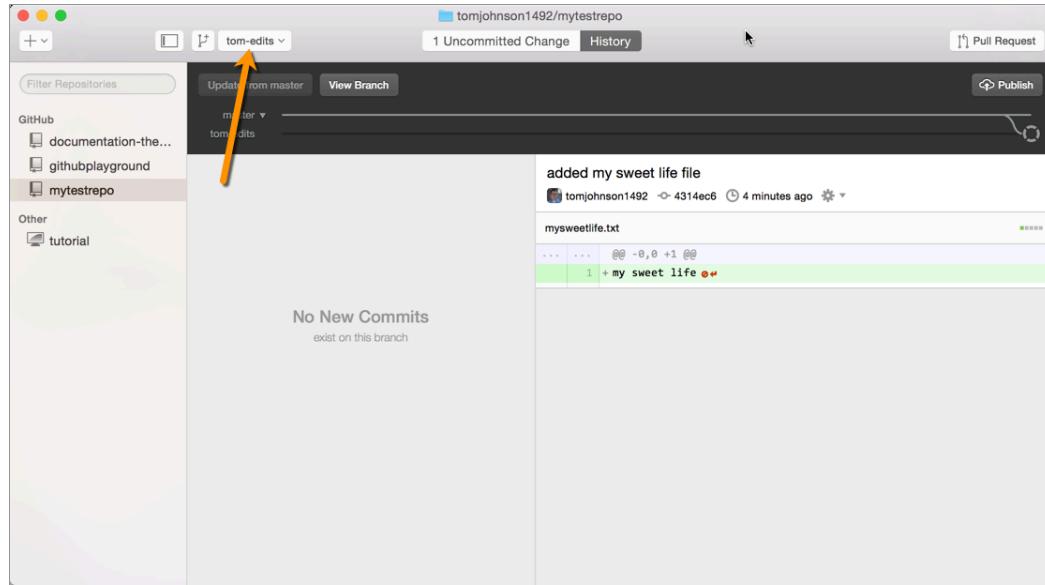
Create a branch

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

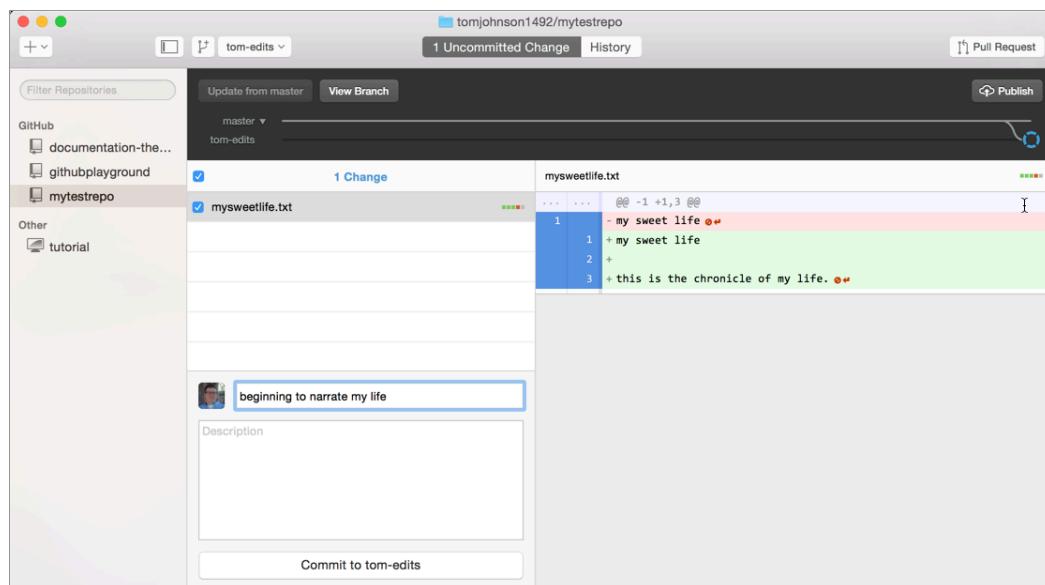
1. Click the **Add a branch** button and create a new branch. Call it something like "tom-edits," but use your own name.



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.

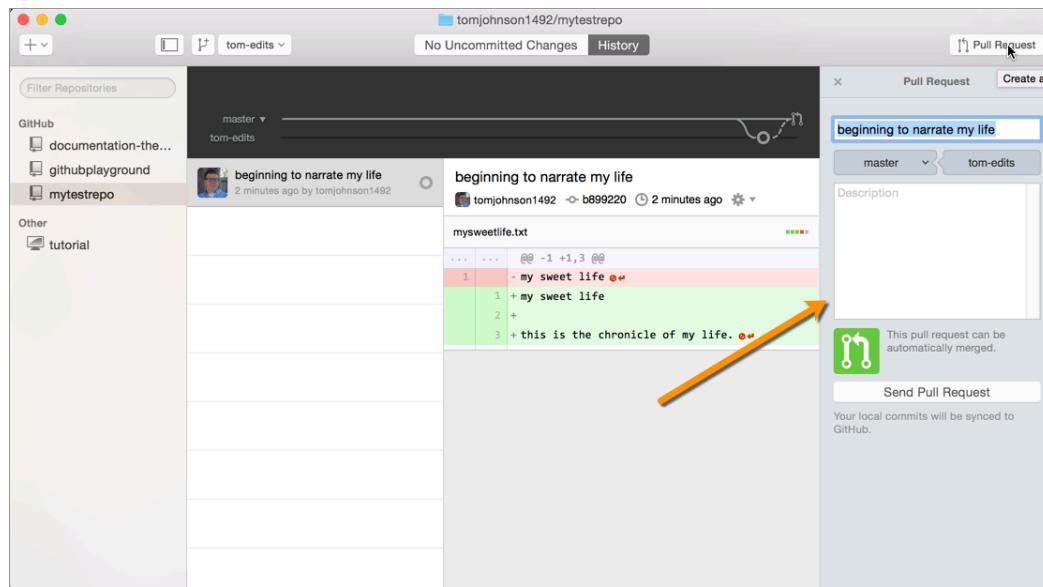
Switch back to your tom-edits branch.

Merge the branch through a pull request

1. 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.

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



3. 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.
4. Click **Merge Pull Request**.

The screenshot shows a GitHub pull request page for a repository named `tomjohnson1492/mytestrepo`. The branch `tom-edits` has been merged into the `master` branch. The pull request summary indicates that the branch is up-to-date with the base branch and merging can be performed automatically. A green button labeled `Merge pull request` is visible. The right sidebar shows basic repository metadata: Labels (None yet), Milestone (No milestone), and Assignee (No one—assign yourself). Notifications show one participant and an option to unsubscribe. The bottom of the page includes a comment input field.

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

5. 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.

The screenshot shows the GitHub Desktop application interface. The left sidebar lists repositories: `documentation-the...`, `githubplayground`, and `mytestrepo` (selected). The top bar shows the repository name `tomjohnson1492/mytestrepo` and a dropdown set to `master`. The main pane displays a pull request titled `Merge pull request #1 from tomjohnson1492/tom-edits`. An orange arrow points from the `master` dropdown in the top bar to the pull request title in the main pane. The right side of the screen shows the diff for the file `mysweetlife.txt`, which contains the commit message `beginning to narrate my life` and the content of the file.

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 browser

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).

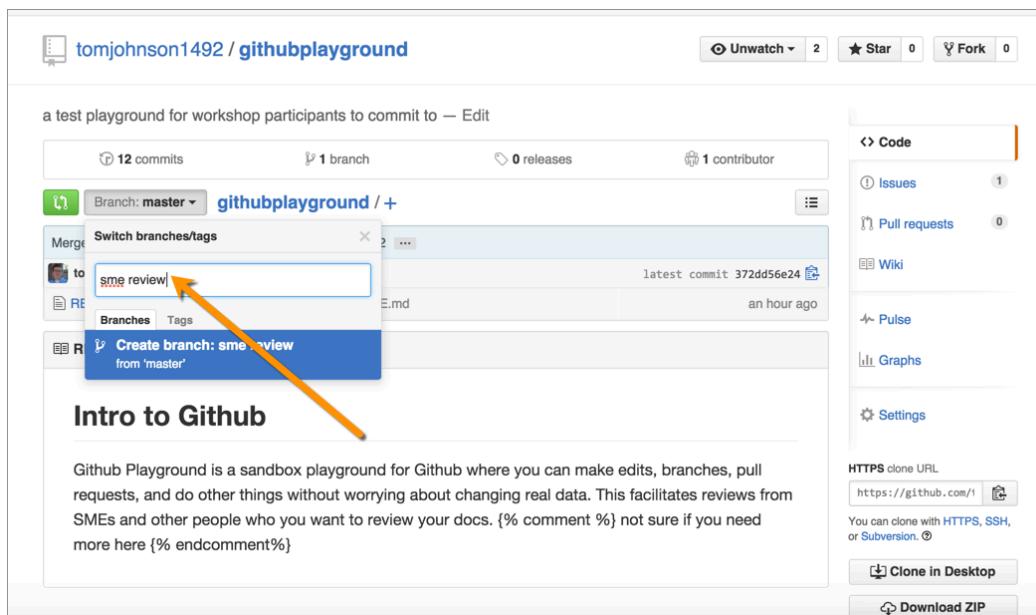
Make edits in a separate branch

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.

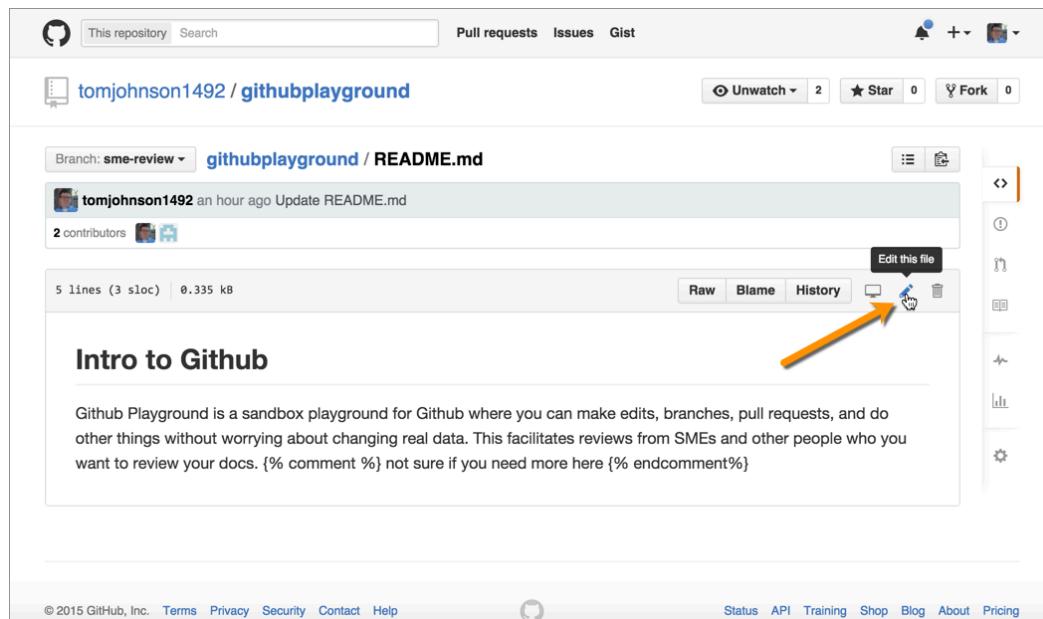
To make edits in a separate branch:

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, and then click **Commit Changes**.

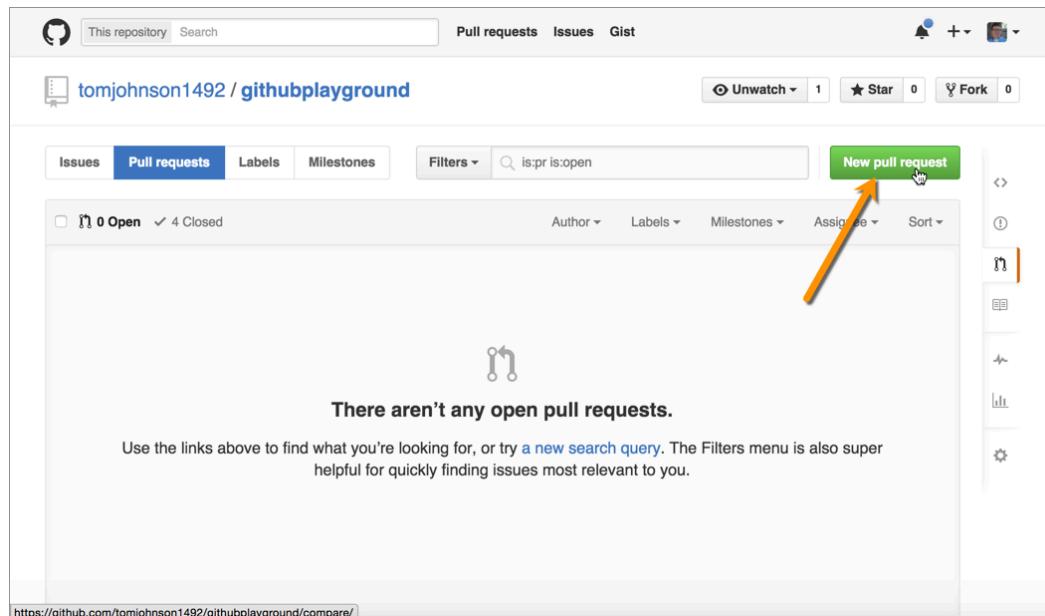
Reviewers could continue making edits this way until they have finished reviewing all of the documentation. All of the changes are made on a branch, not the master.

Create a pull request

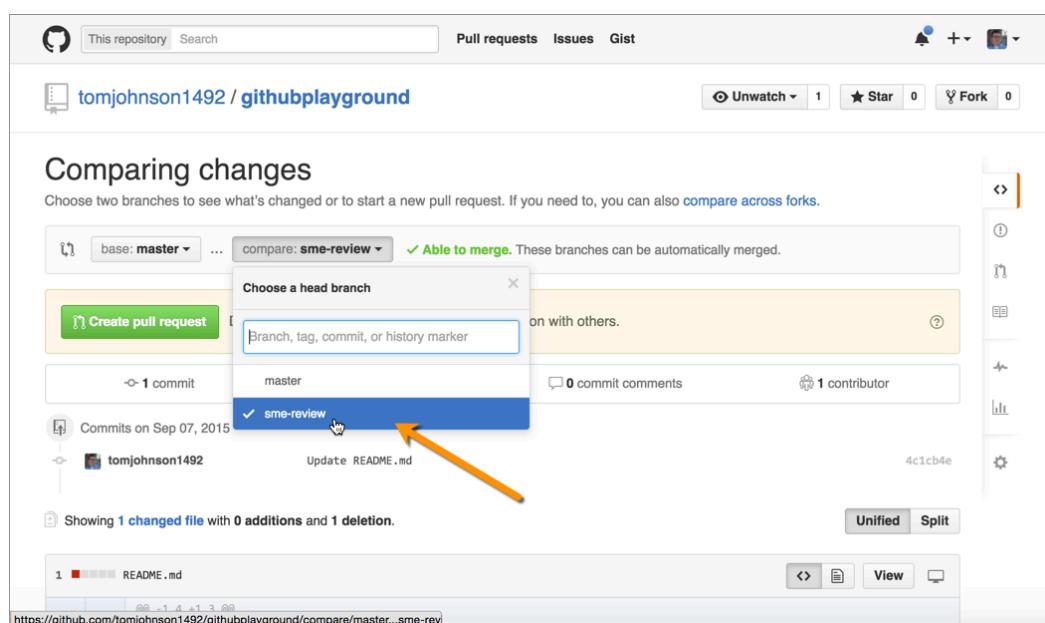
Now that the review process is complete, it's time to merge the branch into the master. You merge the branch into the master through a pull request. Any “collaborator” on the team with write access can initiate and complete the pull request. You can add collaborators through Settings.

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.



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



When you compare the branch against the master, you can see a list of all the changes. You can view the changes through two viewing modes: Unified or Split. 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.

The screenshot shows a GitHub pull request page for a repository named 'tomjohnson1492 / githubplayground'. The pull request is titled 'Update README.md #6'. The 'Files changed' tab is selected, indicated by a blue border and a small number '1'. An orange arrow points to this tab. Below the tabs, it says 'Showing 1 changed file with 0 additions and 1 deletion.' The diff view shows a single change in README.md:

```

@@ -1,4 +1,3 @@
 1 ## Intro to Github
 2
 3 Github Playground is a sandbox playground for Github where you can make edits, branches, pull requests, and do other
     things without worrying about changing real data. This facilitates reviews from SMEs and other people who you want to
     review your docs.
 4 -(% comment %) not sure if you need more here {%

```

At the bottom of the page, there are links for 'Status API Training Shop Blog About Pricing'.

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.

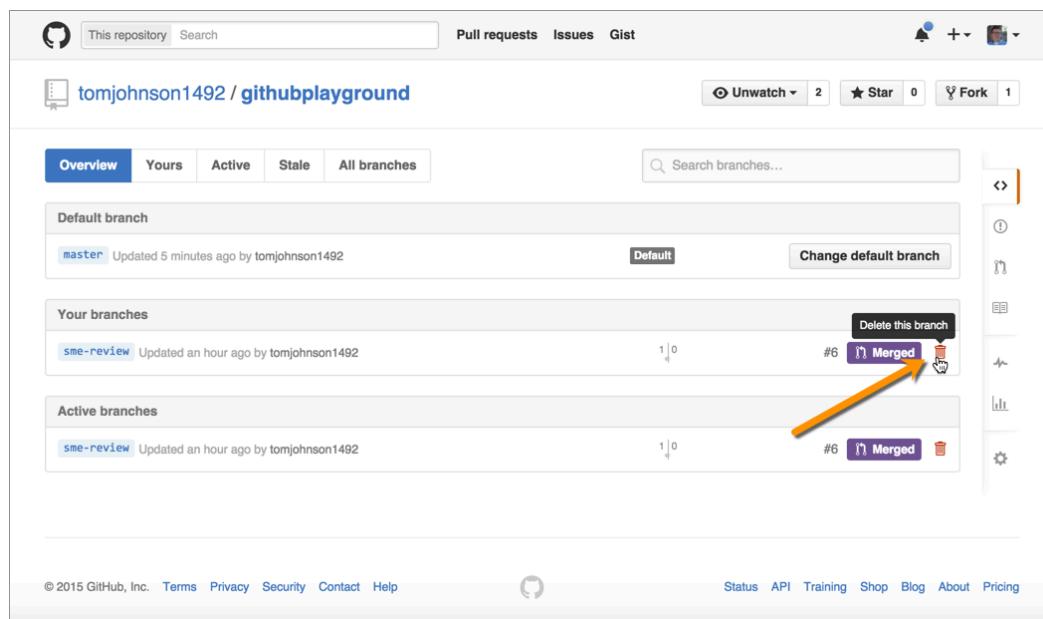
Note also that if the pull request is made against an older version of the master, such that the master's original content no longer exists or has moved elsewhere, the merges will be more difficult to make.

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.



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 to this button, indicating the action to be taken.

REST API specification formats

In an earlier lesson, I mentioned that REST APIs follow an architectural style, not a specific standard. However, there are several REST specifications that have been formulated to try to provide better documentation, tooling, and structure with REST APIs. The three most popular REST API specifications are as follows:

- [Swagger \(http://swagger.io/\)](http://swagger.io/)
- [RAML \(http://raml.org/\)](http://raml.org/)
- [API Blueprint \(https://apiblueprint.org/\)](https://apiblueprint.org/)

Should you use an automated solution?

In a [survey on API documentation \(http://idratherbewriting.com/2015/01/06/api-doc-survey-automating-rest-api-documentation/\)](http://idratherbewriting.com/2015/01/06/api-doc-survey-automating-rest-api-documentation/), I asked people if they were automating their REST API documentation through one of these standards. Only about 30% of the people said yes.

Keep in mind that these specifications just describe the reference endpoints in an API, for the most part. While the reference topics are important, in my documentation projects, the bulk of the documentation is actually not the reference topics. There is a tremendous amount of documentation about how to configure the services that use the endpoint, how to deploy the services, what the various resources and rules are, and so forth.

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 a boatload of documentation about how to actually use your API.

For an excellent overview and comparison of these three REST specification formats, see [Top Specification Formats for REST APIs \(http://nordicapis.com/top-specification-formats-for-rest-apis/\)](http://nordicapis.com/top-specification-formats-for-rest-apis/) by Kristopher Sandoval on the Nordic APIs blog.

Implementing Swagger (OpenAPI specification) with your REST API documentation

I recently gave a presentation that covers the same concepts in this article. You can watch it on YouTube here: <https://goo.gl/n4Hvtq> (<https:// goo.gl/n4Hvtq>).

Experiences that prompted me toward Swagger

On a recent project, after I created documentation for a new API (Application Programming Interface), 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, along with their parameters, 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 \(https://github.com/OAI/OpenAPI-Specification\)](https://github.com/OAI/OpenAPI-Specification) (which is now called the OpenAPI specification but still commonly

referred to as Swagger). I knew that Swagger was a way to make my API documentation interactive. Looking at the [Swagger demo \(`http://petstore.swagger.io`\)](http://petstore.swagger.io), I knew I had to figure it out.

About Swagger

Swagger is a specification for describing REST APIs. This means Swagger 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 Swagger 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 Swagger 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 Swagger specification is available in a [Github repository \(`https://github.com/OAI/OpenAPISpecification`\)](https://github.com/OAI/OpenAPISpecification). Some of these elements are `path` , `parameters` , `responses` , and `security` . Each of these elements is actually an “object” (instead of an XML element) that holds a number of fields and arrays.

In the Swagger specification, your endpoints are `paths` . If you had an endpoint called “pet”, your Swagger specification for this endpoint might look as follows:

```

paths:
  /pets:
    get:
      description: Returns all pets from the system that the user has
      access to
      operationId: findPets
      produces:
        - application/json
        - application/xml
        - text/xml
        - text/html
      parameters:
        - name: tags
          in: query
          description: tags to filter by
          required: false
          type: array
          items:
            type: string
          collectionFormat: csv
        - name: limit
          in: query
          description: maximum number of results to return
          required: false
          type: integer
          format: int32
      responses:
        '200':
          description: pet response
          schema:
            type: array
            items:
              $ref: '#/definitions/pet'

```

This [YAML code \(<https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v2.0/yaml/petstore.yaml>\)](https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v2.0/yaml/petstore.yaml) actually comes from the [Swagger Petstore demo \(<http://petstore.swagger.io/>\).](http://petstore.swagger.io/)

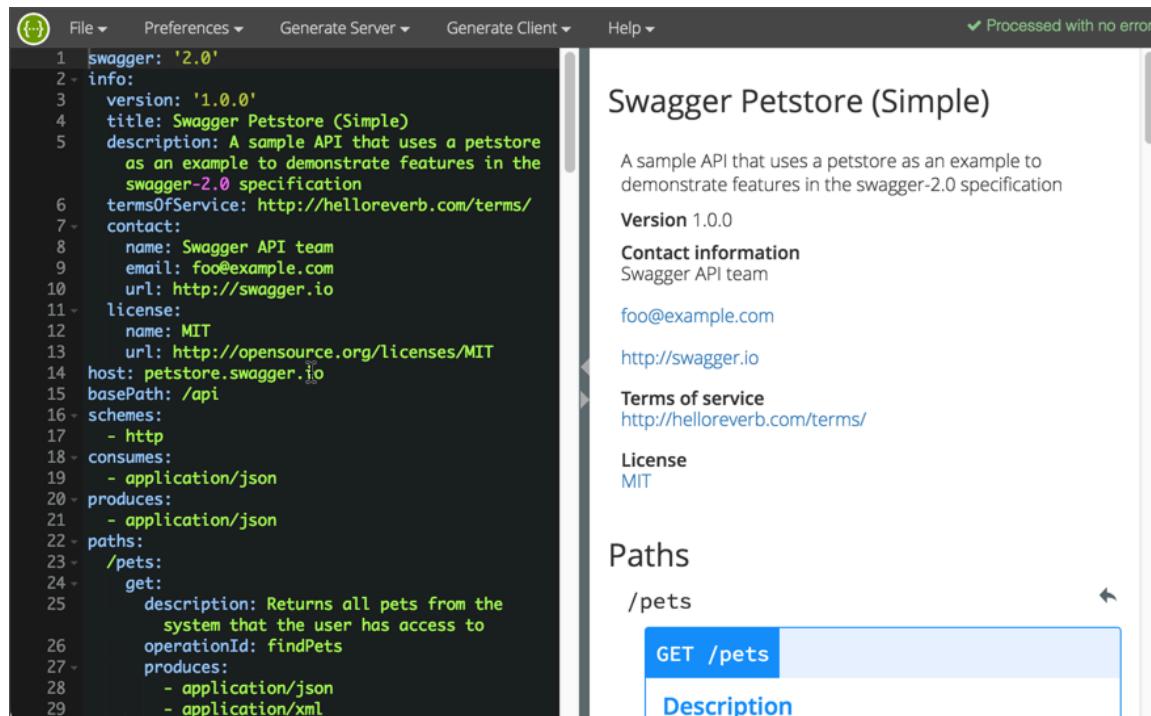
Here's what these objects mean:

- `/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.
- `$ref` is actually a reference to another part of your implementation where the response is defined. (Swagger has a lot of `$ref` references like this to keep your

code clean and to facilitate re-use.)

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

When you're implementing the specification, instead of working in a text editor, you can write your code in the [Swagger editor \(`http://editor.swagger.io/`\)](http://editor.swagger.io/). The Swagger Editor dynamically validates whether the specification file you're creating is valid.



The screenshot shows the Swagger Editor interface. On the left, a code editor displays a YAML-based Swagger specification for a Petstore 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 the host (petstore.swagger.io), basePath (/api), and schemes (HTTP). The paths section contains a single endpoint for getting a list of pets, which returns all pets from the system and has an operation ID of findPets. This endpoint produces JSON or XML. On the right, the generated documentation is shown under the title "Swagger Petstore (Simple)". It includes a brief description of the API as a sample that uses a petstore to demonstrate features in the swagger-2.0 specification. Below this, it lists the version (1.0.0), contact information, and terms of service. Under the "Paths" section, the "/pets" endpoint is detailed, showing the GET method and its description.

```

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://heloreverb.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 file's format, you have the choice of working in either JSON or YAML. The previous code sample is in [YAML \(`http://yaml.org/`\)](http://yaml.org/). 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 trickier to get the spacing right.

Manual or automated?

So far I've been talking about creating the Swagger specification file 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 developers can also automate the specification file through annotations in the programming source code.

Swagger offers a variety of libraries that you can add to your programming code. These libraries will parse through your code's annotations and generate a specification file. Of course, someone has to know exactly what annotations to add and how to add them. Then someone has to write content for each of the annotation's values (describing the endpoint, the parameters, and so on).

Still, many developers get excited about this approach because it offers a way to "automatically" generate documentation from code annotations, which is what developers have been doing for years with other programming languages such as Java (using [Javadoc](http://www.oracle.com/technetwork/articles/java/index-137868.html) (<http://www.oracle.com/technetwork/articles/java/index-137868.html>)) or C++ (using [Doxygen](http://www.stack.nl/~dimitri/doxygen/) (<http://www.stack.nl/~dimitri/doxygen/>)). They usually feel that generating documentation from the code results in less documentation drift.

Although you can generate your specification file from code annotations, not everyone agrees that this is the best approach. In [Undisturbed REST: A Guide to Designing the Perfect API](https://www.mulesoft.com/lp/ebook/api/restbook) (<https://www.mulesoft.com/lp/ebook/api/restbook>), Michael Stowe (<https://twitter.com/mikegstowe>) recommends that teams implement the specification by hand and then treat the specification file as a contract that developers use when doing the actual coding.

In other words, developers consult the specification file 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 file.

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.

From the Swagger specification file, some tools can generate a mock API that you can put before users to have them try out the requests.

The mock API 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 project, our developers weren't that familiar with Swagger, so I simply created the specification file 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.

Parsing the Swagger specification

Once you have a valid Swagger specification file 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 Swagger specification is [Swagger UI](https://github.com/swagger-api/swagger-ui) (<https://github.com/swagger-api/swagger-ui>). After you download Swagger UI, you basically just open up the **index.html** file inside the **dist** folder (which contains the Swagger UI project build) and reference your own Swagger specification file in place of the default one.

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

The screenshot shows the Swagger Petstore interface. At the top, there's a green header bar with the Swagger logo, the URL <http://petstore.swagger.io/v2/swagger.json>, and buttons for Authorize and Explore. Below the header, the title "Swagger Petstore" is displayed, followed by a brief description about the sample server and how to use the API key `special-key`. There are links to find out more about Swagger, including <http://swagger.io>, [Contact the developer](#), and [Apache 2.0](#). The main content area is titled "pet : Everything about your Pets". It lists several API operations under this category:

| Method | Path | Description |
|--------|--------------------------|---|
| 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 |

Below this, there's another section titled "store : Access to Petstore orders".

Some designers criticise the Swagger UI's expandable/collapsible output as being dated. I somewhat agree: the collapsed design makes it difficult to scan the information and easily see the details. However, at the same time, developers find the one-page model attractive and like the ability to zoom out or in for details.

As with most Swagger-based outputs, Swagger UI provides a “Try it out” button. First you populate the endpoint parameters with values. In the following image, users click the Example Value (yellow field) to populate the body parameter with the required JSON. In query parameters, there’s a simple form where you enter the values.

pet : Everything about your Pets

POST /pet Add a new pet to the store

Parameters

| Parameter | Value | Description | Parameter Type | Data Type |
|-----------|--|--|----------------|---|
| body | {
"id": 19,
"category": {
"id": 0,
"name": "string"
},
"name": "mysamplepet19",
}

Parameter content type: application/json | Pet object that needs to be added to the store | body | Model Example Value
{
"id": 0,
"category": {
"id": 0,
"name": "string"
},
"name": "doggie",
"photoUrls": [
"string"
],
"tags": [
{id": 0,
"name": "string"
]
},
"status": "available"
}' http://petstore.swagger.io/v2/pet |

Response Messages

| HTTP Status Code | Reason | Response Model | Headers |
|------------------|---------------|----------------|---------|
| 405 | Invalid input | | |

Try it out!

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

After customizing the parameters, you click **Try it out!** Swagger UI shows you the cURL format of the request followed by the request URL and response. The response is usually returned in JSON format.

Try it out! Hide Response

Curl

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{  
  "id": 19,  
  "category": {  
    "id": 0,  
    "name": "string"  
  },  
  "name": "mysamplepet19",  
  "photoUrls": [  
    "string"  
  ],  
  "tags": [  
    {id": 0,  
    "name": "string"  
  ]  
},  
  "status": "available"  
' http://petstore.swagger.io/v2/pet'
```

Request URL

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

Response Body

```
{  
  "id": 19,  
  "category": {  
    "id": 0,  
    "name": "string"  
  },  
  "name": "mysamplepet19",  
  "photoUrls": [  
    "string"  
  ],  
  "tags": [  
    {id": 0,  
    "name": "string"  
  ]  
},  
  "status": "available"  
' http://petstore.swagger.io/v2/pet'
```

There are other tools besides Swagger UI that can parse your Swagger specification file. Some of these tools include [Restlet Studio](https://restlet.com/products/restlet-studio/) (<https://restlet.com/products/restlet-studio/>), [Apiary](https://apiary.io/) (<https://apiary.io/>), [Apigee](http://apigee.com/about/) (<http://apigee.com/about/>), [Lucybot](https://lucybot.com/) (<https://lucybot.com/>), [Gelato](https://gelato.io/) (<https://gelato.io/>)/[Mashape](https://www.mashape.com/) (<https://www.mashape.com/>), [Readme.io](https://readme.io/)

(<http://readme.io/>), [swagger2postman](https://github.com/josephconley/swagger2postman) (<https://github.com/josephconley/swagger2postman>), [swagger-ui responsive theme](https://github.com/jensoleg/swagger-ui) (<https://github.com/jensoleg/swagger-ui>), [Postman Run Buttons](https://www.getpostman.com/docs/run_button) (https://www.getpostman.com/docs/run_button) and more.

Some web designers have created integrations of Swagger with static site generators such as Jekyll (see [Carte](https://github.com/Wiredcraft/carte) (<https://github.com/Wiredcraft/carte>)). More tools roll out regularly for parsing and displaying content from a Swagger specification file.

In fact, once you have a valid Swagger specification, using a tool called [API Transformer](https://apitransformer.com) (<https://apitransformer.com>), you can even transform it into other API specifications, such as [RAML](http://raml.org/) (<http://raml.org/>) or [API Blueprint](https://apiblueprint.org/) (<https://apiblueprint.org/>). In this way you can 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.)

Responses to Swagger documentation

With my project, I used the Swagger UI to parse my Swagger 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 managers, they loved it. They 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 had a clue about 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, I began to realize there were some downsides to Swagger.

Swagger's output is still just a reference document. It 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 thousand other details that go into a user guide.

So, 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, 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 Swagger 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, or you have to link between the two.

Peter Gruenbaum (<https://www.udemy.com/user/petergruenbaum/>), 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 will 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.

Additionally, some users may not realise 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.)

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. With this sort of request body parameter, Swagger UI's display fell hopelessly short of being usable. The team reverted to much more primitive approaches (such as tables and spreadsheets) for listing all of the parameters and their descriptions.

Some consolations

Despite the shortcomings of Swagger, I still highly recommend it for describing your API.

Swagger is quickly becoming a way for more and more tools (from Postman Run buttons to nearly every API platform) to quickly ingest the information about your API and make it discoverable and interactive with robust, instructive tooling. Through your Swagger specification, you can port your API onto many platforms and systems, as well as automatically set up unit testing and prototyping.

Swagger does provide 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 Swagger specification and describing my API helped inform my own API vocabulary. By poring through the specification, I realised that there were four 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.

Swagger 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, Swagger 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 Swagger, 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.

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.

cURL

A command line utility often used to interact with REST API endpoints. Used in documentation for request code samples.

Endpoint

The end part of the request URL (after the base path). Also sometimes used to refer to the entire API reference topic.

JSON

JavaScript Object Notation. A lightweight syntax containing objects and arrays, usually

used (instead of XML) to return information from a REST API.

OpenAPI

The official name for Swagger. Now under the Open API Initiative with the Linux Foundation (instead of SmartBear, the original development group), the OpenAPI specification aims to be vendor neutral.

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.

Swagger

An official specification for REST APIs. Provides objects used to describe your endpoints, parameters, responses, and security. Now called OpenAPI specification.

Swagger Editor

Swagger specification validator. An online editor that dynamically checks whether your Swagger specification file is valid.

Swagger UI

A display framework. The most common way to parse a Swagger specification file and produce the interactive documentation as shown in the Petstore demo.

YAML

Recursive acronym for “YAML Ain’t No Markup Language.” A human- readable, space-sensitive syntax used in the Swagger specification file.

Resources and further reading

See the following resources for more information on Swagger:

- [API Transformer \(<https://apitransformer.com>\)](https://apitransformer.com)
- [APIMATIC \(<http://www.apimatic.io>\)](http://www.apimatic.io)
- [Carte \(<https://github.com/Wiredcraft/carte>\)](https://github.com/Wiredcraft/carte)
- [Swagger editor \(<http://editor.swagger.io>\)](http://editor.swagger.io)
- [Swagger Hub \(<https://swaggerhub.com>\)](https://swaggerhub.com)
- [Swagger Petstore demo \(<http://petstore.swagger.io>\)](http://petstore.swagger.io)
- [Swagger Tools \(<http://swagger.io/tools>\)](http://swagger.io/tools)
- [Swagger tutorial \(long\) \(<http://apihandyman.io/writing-openapi-swaggerspecification-tutorial-part-1-introduction>\)](http://apihandyman.io/writing-openapi-swaggerspecification-tutorial-part-1-introduction)
- [Swagger tutorial \(short\) \(\[http://idratherbewriting.com/pubapis_swagger\]\(http://idratherbewriting.com/pubapis_swagger\)\)](http://idratherbewriting.com/pubapis_swagger)
- [Swagger/OpenAPI specification \(<https://github.com/OAI/OpenAPISpecification>\)](https://github.com/OAI/OpenAPISpecification)
- [Swagger2postman \(<https://github.com/josephpconley/swagger2postman>\)](https://github.com/josephpconley/swagger2postman)
- [Swagger-ui Responsive theme \(<https://github.com/jensoleg/swagger-ui>\)](https://github.com/jensoleg/swagger-ui)
- [Swagger-ui \(<https://github.com/swagger-api/swagger-ui>\)](https://github.com/swagger-api/swagger-ui)

- Undisturbed REST: A Guide to Designing the Perfect API
(<http://www.mulesoft.com/lp/ebook/api/restbook>), by Michael Stowe

Swagger tutorial

Swagger is one of the most popular specifications for REST APIs for a number of reasons:

- Swagger generates an interactive API console for people to quickly learn about and try the API.
- Swagger generates the client SDK code needed for implementations on various platforms.
- The Swagger file can be auto-generated from code annotations on a lot of different platforms.
- Swagger has a strong community with helpful contributors.

The Swagger spec provides a way to describe your API using a specific JSON or YAML schema that outlines the names, order, and other details of the API.

You can code this Swagger file by hand in a text editor, or you can auto-generate it from annotations in your source code. Different tools can consume the Swagger file to generate the interactive API documentation.

The interactive API documentation generated by the Swagger file is minimal. It shows the resources, parameters, requests, and responses. However, it's not going to provide any other detail about how your API works.

The Swagger Petstore example

In order to get a better understanding of Swagger, let's explore the Petstore example. Note that this UI is Swagger UI. Swagger can be rendered into different visual displays based on the visual framework you decide to use to parse the Swagger spec.

The screenshot shows the Swagger Petstore UI. At the top, there is a navigation bar with a logo, a search bar containing 'http://petstore.swagger.io/v2/swagger.json', an 'api_key' input field, and a 'Explore' button. Below the navigation bar, the title 'Swagger Petstore' is displayed. A brief description 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.' Below this, there is a link to 'Find out more about Swagger' with links to <http://swagger.io>, [Contact the developer](#), and [Apache 2.0](#). The main content area is titled 'pet : Everything about your Pets'. It lists four operations: 'POST /pet' (Add a new pet to the store), 'PUT /pet' (Update an existing pet), 'GET /pet/findByStatus' (Finds Pets by status), and 'GET /pet/findByTags' (Finds Pets by tags). The 'POST /pet' and 'PUT /pet' buttons are highlighted in orange, while the 'GET' buttons are blue. At the bottom of the page, there is a footer with the URL '(<http://petstore.swagger.io/>)'.

There are three resources: pet, store, and user.

Create a pet

1. In the **Pet** resource, expand the **Post** method.
2. Click the yellow JSON in the Model Schema section:

pet : Everything about your Pets

POST /pet Add a new pet to the store

Parameters

| Parameter | Value | Description | Parameter Type | Data Type |
|-------------|---|--|----------------|-----------|
| body | <pre>{ "id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }] }</pre> <p>Parameter content type: application/json</p> | Pet object that needs to be added to the store | body | Model |

Model **Model Schema**

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

Click to set as parameter value

Response Messages

| HTTP Status Code | Reason | Response Model | Headers |
|------------------|---------------|----------------|---------|
| 405 | Invalid input | | |

This populates the body value with the JSON. This is the JSON you must submit in order to create a pet.

3. Change the value for the first `id` tag. (Make it really unique so that others don't use the same `id`, and don't start with `0`.)
4. Change `name` value to something unique. Here's an example:

```
{
  "id": 37987,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "Mr. Fluffernutter",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

5. Click the **Try it out!** button.

Look and see the response.



The screenshot shows the "Response Body" section of the Swagger UI. It displays a JSON object representing a pet. The JSON structure is as follows:

```
{  
  "id": 37987,  
  "category": {  
    "id": 0,  
    "name": "string"  
  },  
  "name": "Mr. Fluffernutter",  
  "photoUrls": [  
    "string"  
  ],  
  "tags": [  
    {  
      "id": 0,  
      "name": "string"  
    }  
  ],  
  "status": "available"  
}
```

Below the response body, there is a "Response Code" section which is currently empty.

Find your pet by the ID

1. Expand the GET `pet/{petId}` method.
2. Insert your pet's ID in the `petId` value box.
3. Click **Try it out!**

The pet you created is returned in the response.

By default, the response will be in XML. Change the **Response Content Type** selector to **application/json** and click **Try it out!** again.

The pet response is returned in JSON format.

The emphasis in Swagger documentation is to learn by doing. You make real requests and see responses to better understand how the Petstore API really works. However, note also that now you have created a new pet in your actual Petstore database. 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.

Sorting out the Swagger components

Swagger has a number of different pieces:

Swagger spec (<https://github.com/swagger-api/swagger-spec>): The Swagger spec is the official schema about name and element nesting, order, and so on. If you plan on hand-coding the Swagger files, you'll need to be extremely familiar with the Swagger spec.

Swagger editor (<http://editor.swagger.io/#/>): The Swagger Editor is an online editor that validates your YML-formatted content against the rules of the Swagger spec. YML is a syntax that depends on spaces and nesting. You'll need to be familiar with YML syntax and

the rules of the Swagger spec to be successful here. The Swagger editor will flag errors and give you formatting tips. (Note that the Swagger spec file can be in either JSON or YAML format.)

The screenshot shows the Swagger Editor interface. On the left, a code editor displays a Swagger specification file (YAML) for a Petstore API. The file includes details like the API title, version, contact information, and various API endpoints for managing pets. On the right, the generated UI for the Petstore API is shown. It features a header with the title "Swagger Petstore" and a note that it's a sample server. Below this are sections for "Contact information" (with email apiteam@wordnik.com), "Terms of service" (link to http://helloreverb.com/terms/), and "License" (Apache 2.0). A "Security" section contains two authentication methods: "api_key (API Key)" and "petstore_auth (OAuth 2.0)". Each method has an "Authenticate" button.

```

swagger: "2.0"
info:
  description: |
    This is a sample server Petstore server.
  [Learn about Swagger](http://swagger.wordnik.com) or join the IRC channel '#swagger' on irc.freenode.net.
  For this sample, you can use the api key 'special-key' to test the authorization filters
  version: "1.0.0"
  title: Swagger Petstore
  termsOfService: http://helloreverb.com/terms/
  contact:
    name: apiteam@wordnik.com
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  host: petstore.swagger.wordnik.com
  basePath: /v2
  schemes:
    - http
  paths:
    /pets:
      post:
        tags:
          - pet
        summary: Add a new pet to the store
        description: ""
        operationId: addPet
        consumes:
          - application/json
          - application/xml
        produces:
          - application/json
          - application/xml
        parameters:
          - in: body
            name: body
            description: Pet object that needs to be added to the store

```

(<http://editor.swagger.io/>)

Swagger-UI (<https://github.com/swagger-api/swagger-ui>): The Swagger UI is an HTML/CSS/JS framework that parses a JSON or YAML file that follows the Swagger spec and generates a navigable UI of the documentation. This is the tool that transforms your spec into the Swagger Petstore-like UI output.

Swagger-codegen (<https://github.com/swagger-api/swagger-codegen>): This utility generates client SDK code for a lot of different platforms (such as Java, JavaScript, Scala, Python, PHP, Ruby, Scala, and more). This client 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. An SDK is supportive tooling that helps developers use the REST API.

Some sample Swagger implementations

Before we get into this tutorial, check out a few Swagger implementations:

- Reverb (<https://reverb.com/swagger#!/accounts>)
- VocaDB (<http://vocadb.net/swagger/ui/index>)
- Watson Developer Cloud (<http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/apis/>)

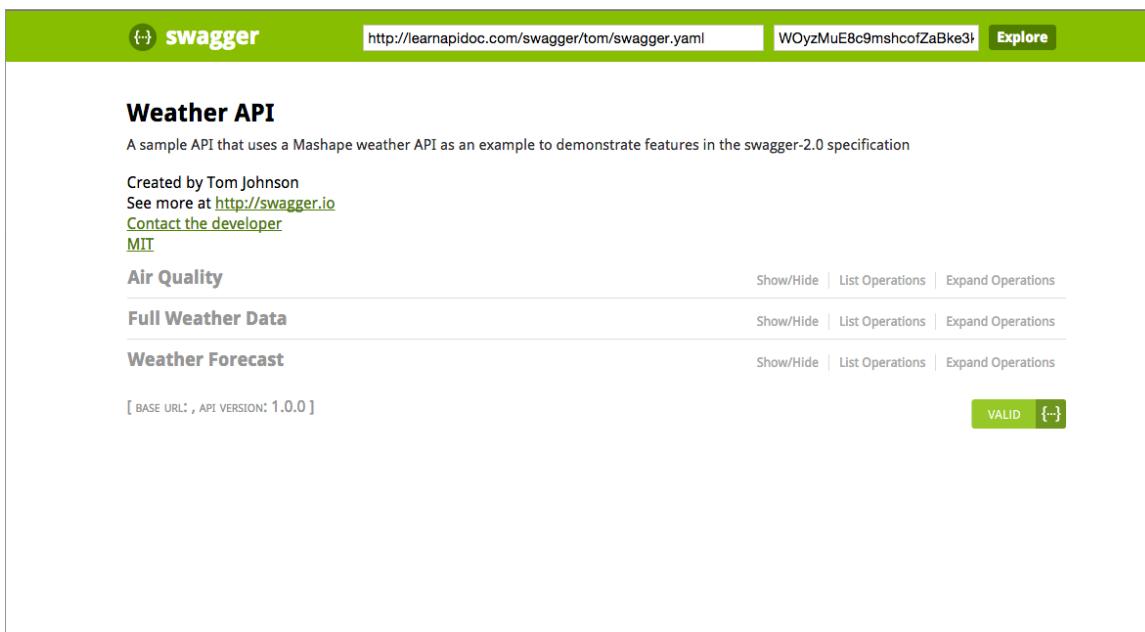
Most of them look pretty much the same, with minimal branding. You'll notice the documentation is short and sweet in a Swagger implementation. This is because the Swagger display is meant to be an interactive experience where you can try out calls and see responses — using your own API key to see your own data. It's the learn-by-doing-and-seeing-it approach.

Note a few limitations with the Swagger approach:

- There's not much room to describe in detail the workings of the endpoint.
- The Swagger UI looks mostly the same for each output.
- The Swagger UI will be a separate site from your other documentation.

Create a Swagger UI display

In this activity, you'll create a Swagger UI display for the weatherdata endpoint in this [Mashape Weather API](https://www.mashape.com/fyhao/weather-13#weatherdata) (<https://www.mashape.com/fyhao/weather-13#weatherdata>). (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](http://idratherbewriting.com/files/restapicourse/swagger) (<http://idratherbewriting.com/files/restapicourse/swagger>):

A screenshot of the Swagger UI interface. At the top, there is a green header bar with the word "swagger" and a logo. Below the header, the title "Weather API" is displayed, followed by a subtitle: "A sample API that uses a Mashape weather API as an example to demonstrate features in the swagger-2.0 specification". Underneath the title, there is developer information: "Created by Tom Johnson", "See more at <http://swagger.io>", "Contact the developer", and "MIT". Below this, there are three main sections: "Air Quality", "Full Weather Data", and "Weather Forecast", each with "Show/Hide", "List Operations", and "Expand Operations" buttons. At the bottom left, it says "[BASE URL: , API VERSION: 1.0.0]". On the right side, there is a green "VALID" button and a copy icon. The entire interface is contained within a white frame.

(<http://idratherbewriting.com/files/restapicourse/swagger>)

a. Create a Swagger spec file

To create a Swagger spec file:

1. Go to the [Swagger online editor](http://editor.swagger.io/#/) (<http://editor.swagger.io/#/>).
2. Select **File > Open Example** and choose **PetStore Simple**. Click **Open**.

You could just customize this sample YML file with the weatherdata endpoint documentation. However, if you're new to Swagger it will take you some time to learn the spec. For the sake of convenience, just go to the following file, and then copy and paste its code into the Swagger editor: [swagger.yaml](http://idratherbewriting.com/files/restapicourse/swagger/swagger.yaml) (<http://idratherbewriting.com/files/restapicourse/swagger/swagger.yaml>).

Here's what the Swagger YAML file looks like:

```
swagger: "2.0"
info:
  version: "1.0.0"
  title: "Weather API"
  description: "A sample API that uses a Mashape weather API as a
n example to demonstrate features in the swagger-2.0 specificati
on"
  termsOfService: "http://helloreverb.com/terms/"
  contact:
    name: "Tom Johnson"
    email: "tomjohnson1492@gmail.com"
    url: "http://swagger.io"
  license:
    name: "MIT"
    url: "http://opensource.org/licenses/MIT"
host: "simple-weather.p.mashape.com"
schemes:
- "https"
consumes:
- "application/json"
produces:
- "application/text"
paths:
/aqi:
  get:
    tags:
      - "Air Quality"
    description: "gets air quality index"
    operationId: "getAqi"
    produces:
      - "text"
    parameters:
      -
        name: "lat"
        in: "query"
        description: "latitude"
        required: false
        type: "string"
      -
        name: "lng"
        in: "query"
        description: "longitude"
        required: false
        type: "string"
    responses:
      200:
        description: "aqi response"
```

```
default:
  description: "unexpected error"

/weather:
  get:
    tags:
      - "Weather Forecast"
    description: "gets weather forecast in short label"
    operationId: "getWeather"
    produces:
      - "text"
    parameters:
      -
        name: "lat"
        in: "query"
        description: "latitude"
        required: false
        type: "string"
      -
        name: "lng"
        in: "query"
        description: "longitude"
        required: false
        type: "string"
    responses:
      200:
        description: "weather response"
        default:
          description: "unexpected error"
/weatherdata:
  get:
    tags:
      - "Full Weather Data"
    description: "Get weather forecast by Latitude and Longitude"
    operationId: "getWeatherData"
    produces:
      - "application/json"
    parameters:
      -
        name: "lat"
        in: "query"
        description: "latitude"
        required: false
        type: "string"
      -
        name: "lng"
```

```
in: "query"
description: "longitude"
required: false
type: "string"
responses:
  200:
    description: "weatherdata response"
    default:
      description: "unexpected error"

securityDefinitions:
  internalApiKey:
    type: apiKey
    in: header
    name: X-Mashape-Key
```

Notice that this is YML instead of JSON. YML syntax is a more human-readable form of JSON. With YML, spacing matters. New levels are set with two indented spaces. The colon indicates an object. Hyphens represent a sequence or list (like an array). If you [download this file \(<http://idratherbewriting.com/files/restapicourse/swagger/swagger.yaml>\)](http://idratherbewriting.com/files/restapicourse/swagger/swagger.yaml) instead of copy-and-pasting it above, you're less likely to run into spacing errors.

The Swagger editor shows you how the file will look in the output. You'll also be able to see if there are any validity errors. Without this online editor, you would only know that the YML syntax is valid when you run the code (and see errors indicating that the YAML file couldn't be parsed).

3. Make sure the YAML file is valid in the Swagger editor. If there are any errors, fix them.
4. Go to **File > Download YAML** and save the file as “swagger.yaml” on your computer. (You could also just copy the code and insert it into a blank file and call it swagger.yaml.)

You can also choose JSON as the format, but YAML is more readable and works just as well.

b. Set Up the Swagger UI

1. Go to the [Swagger UI \(<https://github.com/swagger-api/swagger-ui>\)](https://github.com/swagger-api/swagger-ui) Github project. Click the **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. Everything else is used only if you're regenerating the files, which is beyond the scope of this tutorial.

2. Drag the **dist** folder out of the swagger-ui-master folder so that it stands alone.

- Then delete the swagger-ui-master folder.
3. Inside your **dist** folder, open **index.html** in a text editor.
 4. Look for the following code:

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

5. Change the `url` value from `http://petstore.swagger.io/v2/swagger.json` to the following: `"swagger.yaml";`.
6. Drag the **swagger.yaml** file that you created earlier into the same directory as the `index.html` file you just edited.
7. Save the file.
8. To view the file, open it in Firefox. (Chrome may block the local Javascript.)

c. Upload the Files to a Web Host

In addition to viewing the Swagger file locally in Firefox, you can also run a web server locally on your computer through XAMPP:

1. Download and install [XAMPP](https://www.apachefriends.org/) (<https://www.apachefriends.org/>).
2. After installation, in your Applications folder, open the XAMPP folder and start the “manager-osx” console.
3. Click the **Manage Servers** tab in the console manager.
4. Select **Apache Web Server** and click **Start**.
5. Open the **htdocs** folder where XAMPP was installed. On a Mac, the location is usually in `/Applications/XAMPP/xamppfiles/htdocs`.
6. Drag the dist folder into this space.
7. In your browser, go to localhost/dist.

The Swagger UI display should appear.

Interact with the Swagger UI

1. Go to the URL where you uploaded your Swagger files. Alternatively, if you’re using XAMPP locally, go to localhost/dist.
2. In the upper-right corner, click **Authorize** and enter your Mashape API key. If you don’t have a Mashape API key, you can use **EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p**.
3. Go to Google Maps and search for an address.
4. Get the latitude and longitude from the URL, and plug it into your Swagger UI. (For example, **1.3319164** for lat, **103.7231246** for lng.)
5. Click **Try it out**.

If successful, you should see something in the response body like this:

```
9 c, Mostly Cloudy at South West, Singapore
```

If you see a response that says “Not supported,” try the lat and lng coordinates used here.

Try working with each of your endpoints and see the data that gets returned.

Note that if you refresh the page, you’ll need to re-enter your API key.

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](#) (<http://www.infoq.com/articles/swagger-scalatra>). The annotation methods for Swagger doc blocks vary based on the programming language.

For other tools and libraries, see [Swagger services and tools](#) (<http://swagger.io/open-source-integrations/>).

More about YAML

When you created the Swagger file, you used a syntax called YML. YML stands for “YAML Ain’t Markup Language.” This means that the YAML syntax doesn’t have markup tags such as `<` or `>`.

Working with YAML

YML is easier to work with because it generally 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:
- JyYaml       # Java port of RbYaml
- SnakeYAML   # Java 5 / YAML 1.1
- YamlBeans    # To/from JavaBeans
- Yaml         # Original Java Implementation
```

(<http://yaml.org/>)

The YAML site itself is written using YAML, which you can immediately see is not intended for coding web pages.

YML 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 to indicate the structure.

Many computers ingest data in a YML 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 swagger.json or swagger.yaml files equivalently. Pretty much any parser that reads JSON will also read YAML. However, some YAML parsers might not read JSON, because there are a few features YAML has that JSON lacks (more on that later).

YAML syntax

With a YML file, spacing is significant. Each two-space indent represents a new level:

```
level1:  
  level2:  
    level3:
```

With YAML, you 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"
```

YAML files begin with `---`. 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, then you'll want to enclose it in quotation marks. And if there's a double quotation mark, then 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 \(`http://www.unserialize.me/`\)](http://www.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) 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`.

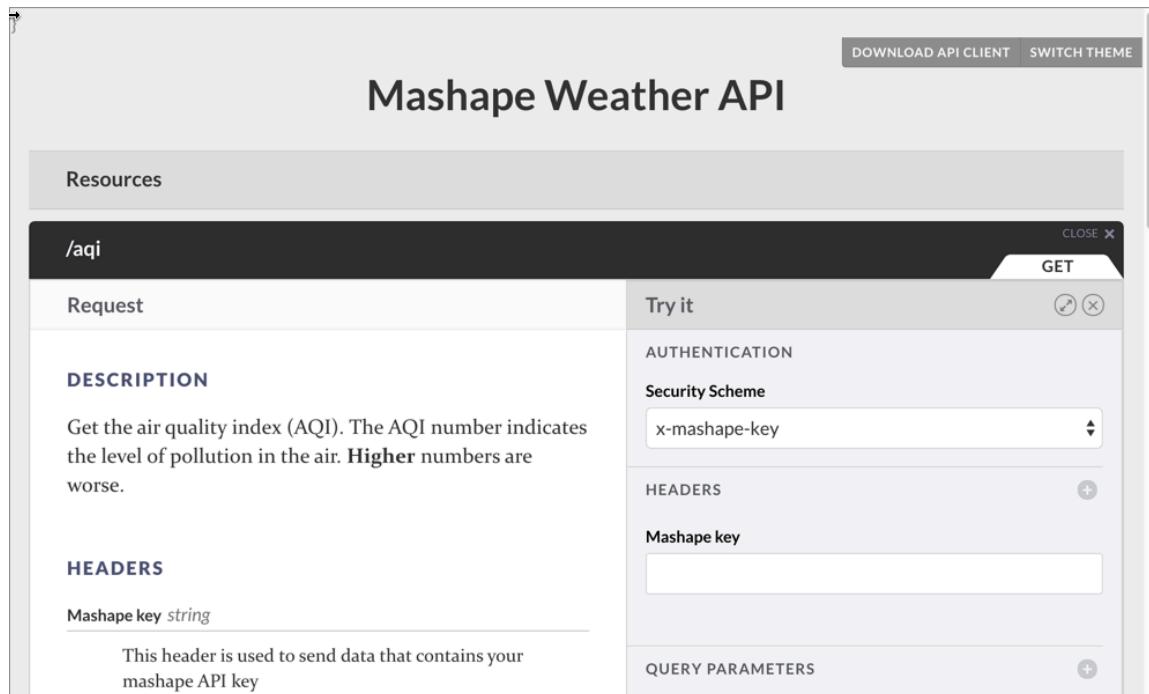
For details on other differences, see [Learn YAML in Minutes](http://learnxinyminutes.com/docs/yaml/) (<http://learnxinyminutes.com/docs/yaml/>). To learn more about YML, see this [YML tutorial](http://rhnh.net/2011/01/31/yaml-tutorial) (<http://rhnh.net/2011/01/31/yaml-tutorial>).

RAML tutorial

RAML stands for REST API Modeling Language and is similar to Swagger and other API specifications. RAML is backed by Mulesoft (<https://www.mulesoft.com/>), a commercial API company, and uses a more YAML-based syntax in the specification.

RAML overview

Similar to Swagger, once 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, tries to be 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. Under 'Resources', there's a detailed view for the '/aqi' endpoint. The endpoint is listed as a 'GET' request. The 'DESCRIPTION' section contains the text: 'Get the air quality index (AQI). The AQI number indicates the level of pollution in the air. Higher numbers are worse.' The 'HEADERS' section includes a 'Mashape key string' header, described as 'This header is used to send data that contains your mashape API key'. On the right side of the interface, there are sections for 'AUTHENTICATION' (with 'Security Scheme' set to 'x-mashape-key'), 'HEADERS' (with 'Mashape key' input field), 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 specs (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 push for using a spec 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](http://raml.org/spec.html) (<http://raml.org/spec.html>) and look at some examples. See also [this RAML tutorial](http://raml.org/docs.html#step-introduction) (<http://raml.org/docs.html#step-introduction>) and [this video tutorial](https://www.youtube.com/embed/5o_nExedezw?autoplay=1) (https://www.youtube.com/embed/5o_nExedezw?autoplay=1).

Even so, the documentation for the RAML spec isn't always so clear. For example, when I was trying to get the right syntax for the security scheme, the information was lacking on how to create security schemes that were based on a custom key in the header.

Here's the Mashape Weather API 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/MYXX004_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/weathe
r/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/u
s/we/52/28.gif\"/><br />\n<Current Conditions:><br />\nMostly Cloud
y, 32 C<BR />\n<BR /><b>Forecast:</b><BR />\nSat - Scattered Thunderst
orms. High: 32 Low: 26<br />\nSun - Thunderstorms. High: 33 Low: 27<b
r />\nMon - Scattered Thunderstorms. High: 32 Low: 26<br />\nTue - Thu
nderstorms. High: 32 Low: 26<br />\nWed - Scattered Thunderstorms. Hig
h: 32 Low: 27<br />\n<br />\n<a href=\"http://us.rd.yahoo.com/dailynew
s/rss/weather/Tebrau_MY/*http://weather.yahoo.com/forecast/MYXX00
4_c.html\">Full Forecast at Yahoo! Weather</a><BR/><BR/>\n(provided b
y <a href=\"http://www.weather.com\">The Weather Channel</a>)<b
r />\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"  
}  
}  
}  
}  
}  
}  
}
```

Outputs

You can generate outputs using the RAML spec from a variety of platforms. Here are three ways:

- [Developer Portal on Anypoint platform](https://anypoint.mulesoft.com/apiplatform) (<https://anypoint.mulesoft.com/apiplatform>). Choose this option if you are developing and delivering your API on Mulesoft's Anypoint platform.
- [API Console output](https://github.com/mulesoft/api-console) (<https://github.com/mulesoft/api-console>). 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](http://raml2html.leanlabs.io/) (<http://raml2html.leanlabs.io/>). 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](https://anypoint.mulesoft.com/apiplatform) (<https://anypoint.mulesoft.com/apiplatform>).
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 for managing APIs. On the left, there's a sidebar with a dropdown menu open, showing options like 'Page', 'API Notebook', 'API Reference' (which is highlighted with a red arrow), 'Header', and 'External link'. The main content area is titled 'API reference' and lists three resources: '/aqi' (GET), '/weather' (GET), and '/weatherdata' (GET). At the bottom of the API reference section, there's a link to 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).

(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](https://api-notebook.anypoint.mulesoft.com/#examples) (<https://api-notebook.anypoint.mulesoft.com/#examples>).

10. Click the **Set to visible** icon (looks like an eye).
11. Click **Live Portal**.

The screenshot shows the MuleSoft API Platform Developer portal. The top navigation bar includes the Mule logo, the text "weather 1.0", and a user dropdown "tomjohnson1492". Below the header, there are tabs for "Developer portal", "weather - 1.0", and "API reference". On the left sidebar, there are links for "Home" and "API reference". The main content area is titled "API reference" and contains a section for "Resources". It lists three endpoints: "/aqi" (GET), "/weather" (GET), and "/weatherdata" (GET). A note "API is behind a firewall (?)" with a checkbox follows. At the bottom, the RAML URL is displayed: [/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](https://github.com/mulesoft/api-console) (<https://github.com/mulesoft/api-console>).
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.

The screenshot shows the RAML Console application running in a web browser. The title bar indicates the file path: "file:///Users/tjohnson/projects/api-console-master/dist/index.html". The main interface has two main sections: "INITIALIZE FROM THE URL OF A RAML FILE" and "OR PARSE RAML IN HERE". The first section contains a text input field and a "Load from URL" button. The second section contains a text input field containing the number "1" and a "Load RAML" button.

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:

The screenshot shows the Mashape Weather API RAML console. At the top right are links for "DOWNLOAD API CLIENT" and "SWITCH THEME". The main title is "Mashape Weather API". Below it is a "Resources" section. Three API endpoints are listed: "/aqi" (GET), "/weather" (GET), and "/weatherdata" (GET). The interface is clean and modern, typical of developer tooling.

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 (<http://idratherbewriting.com.com/files/restapicourse/raml/specific.html>) that integrates the weather.raml file. Here's a generic RAML API Console (<http://idratherbewriting.com.com/files/restapicourse/raml/examples/weather.raml>) that allows you to insert your own RAML spec code.

Deliver doc through the RAML2HTML Utility

Here's [an example \(http://raml2html.leanlabs.io/github\)](http://raml2html.leanlabs.io/github) of what the RAML2HTML output looks like. It's a static HTML output without any interactivity.

To generate this kind of output:

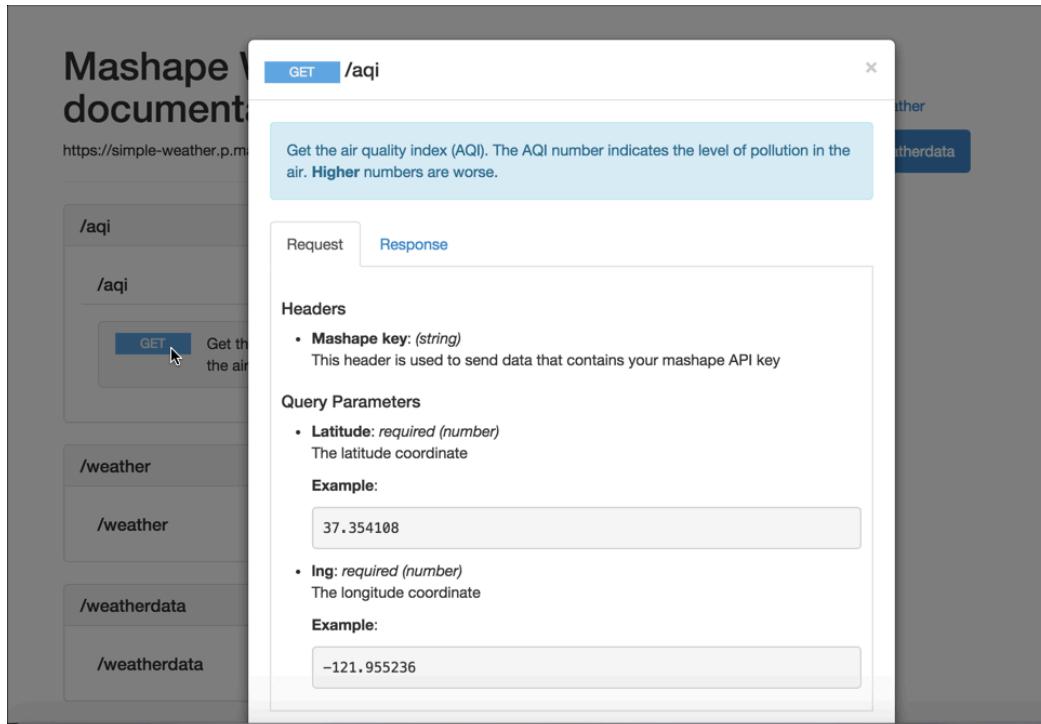
1. Install RAML2HTML through either of these methods:
 - [Through Node \(https://www.npmjs.com/package/raml2html\)](https://www.npmjs.com/package/raml2html)
 - [Through wget \(http://raml2html.leanlabs.io/documentation\)](http://raml2html.leanlabs.io/documentation)
2. In Terminal, enter this command:

```
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:



To see this example in your browser, go to idratherbewriting.com/files/restapicourseraml/examples/index.html (<http://idratherbewriting.com/files/restapicourse/raml/examples/index.html>).

Other platforms that consume RAML and Swagger

Restlet Studio (<http://studio.restlet.com>) 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 spec (which you can [read here \(`https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md`\)](https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md)). If you describe your API with this blueprint, then different tools can read and display the information.

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 recommended to follow the "url" link values, [Link](https://tools.ietf.org/html/rfc5988) or Location headers where applicable to retrieve resources. Instead of constructing your own URLs, to keep your client decoupled from implementation details.

## Retrieve the Entry Point [GET]

+ Response 200 (application/json)

{
    "questions_url": "/questions"
}

## Group Question

Resources related to questions in the API.

## Question [/questions/{question_id}]

A Question object has the following attributes:

+ question
+ published_at - An ISO8601 date when the question was published.
+ url
+ choices - An array of Choice objects.

+ Parameters
    + question_id: 1 (required, number) - ID of the Question in form of an integer

### 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](https://apiary.io/blueprint) (<https://apiary.io/blueprint>) or this [tutorial on Github](https://github.com/apiaryio/api-blueprint/blob/master/Tutorial.md) (<https://github.com/apiaryio/api-blueprint/blob/master/Tutorial.md>).

You can find [examples of different blueprints here](https://github.com/apiaryio/api-blueprint/tree/master/examples) (<https://github.com/apiaryio/api-blueprint/tree/master/examples>). The examples can often clarify different aspects of the spec.

Parsing the blueprint

There are many tools that can parse an API blueprint. [Drafter](https://github.com/apiaryio/drafter) (<https://github.com/apiaryio/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](https://github.com/danielgtaylor/aglio) (<https://github.com/danielgtaylor/aglio>) can parse a blueprint and generate static HTML files.

For a more comprehensive list of tools, see the [Tooling](https://apiblueprint.org/#tooling) (<https://apiblueprint.org/#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 machineapio.)

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 (<https://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 titled "API Blueprint Syntax Tutorial" containing the following API Blueprint code:

```
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     {
24       "questions_url": "/questions"
25     }
26   ]
27
28 ## Group Question
```

On the right, the "Reference" pane is open, showing the "Polls API Root" section. The title "Polls" is displayed, followed by a "INTRODUCTION" section which states: "Polls is a simple API allowing consumers to view polls and vote in them." Below this is a "REFERENCE" section.

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](https://raw.githubusercontent.com/apiaryio/api-blueprint/master/examples/Polls%20API.md) (<https://raw.githubusercontent.com/apiaryio/api-blueprint/master/examples/Polls%20API.md>).

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](https://apiary.io/blueprint) (<https://apiary.io/blueprint>) 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 info formatted in the blueprint format.

4. Copy the following code, which aligns with the API Blueprint spec, and paste it into the Apiary blueprint editor.

```
FORMAT: 1A
HOST: https://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/MYXX000
```

```
4_c.html",
  "description": "Yahoo! Weather for Tebrau,
MY",
  "language": "en-us",
  "lastBuildDate": "Sat, 03 May 2014 11:00 a
m 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 1
1:00 am MYT",
    "lat": "1.58",
    "long": "103.74",
    "link": "http://us.rd.yahoo.com/dailynew
```

```
s/rss/weather/Tebrau_MY/*http://weather.yahoo.com/forecast/MYXX
0004_c.html",
    "pubDate": "Sat, 03 May 2014 11:00 am MY
T",
    "condition": {
        "code": "28",
        "date": "Sat, 03 May 2014 11:00 am MY
T",
        "temp": "32",
        "text": "Mostly Cloudy"
    },
    "description": "\n<img src=\"http://l.yim
g.com/a/i/us/we/52/28.gif\"/><br />\n<b>Current Condition
s:</b><br />\nMostly Cloudy, 32 C<BR />\n<BR /><b>Forecas
t:</b><BR />\nSat - Scattered Thunderstorms. High: 32 Low: 26<b
r />\nSun - Thunderstorms. High: 33 Low: 27<br />\nMon - Scatter
ed Thunderstorms. High: 32 Low: 26<br />\nTue - Thunderstorms. H
igh: 32 Low: 26<br />\nWed - Scattered Thunderstorms. High: 32 L
ow: 27<br />\n<br />\n<a href=\"http://us.rd.yahoo.com/dailynew
s/rss/weather/Tebrau_MY/*http://weather.yahoo.com/forecast/MYXX
0004_c.html\">Full Forecast at Yahoo! Weather</a><BR /><BR />\n(pr
ovided 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_MY
T"
      }
    }
  }
]
```

If the code isn't easy to copy and paste, you can [view and download the file here](http://idratherbewriting.com/files/publishingapidocs/apiblueprintweatherdata.md) (<http://idratherbewriting.com/files/publishingapidocs/apiblueprintweatherdata.md>).

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.

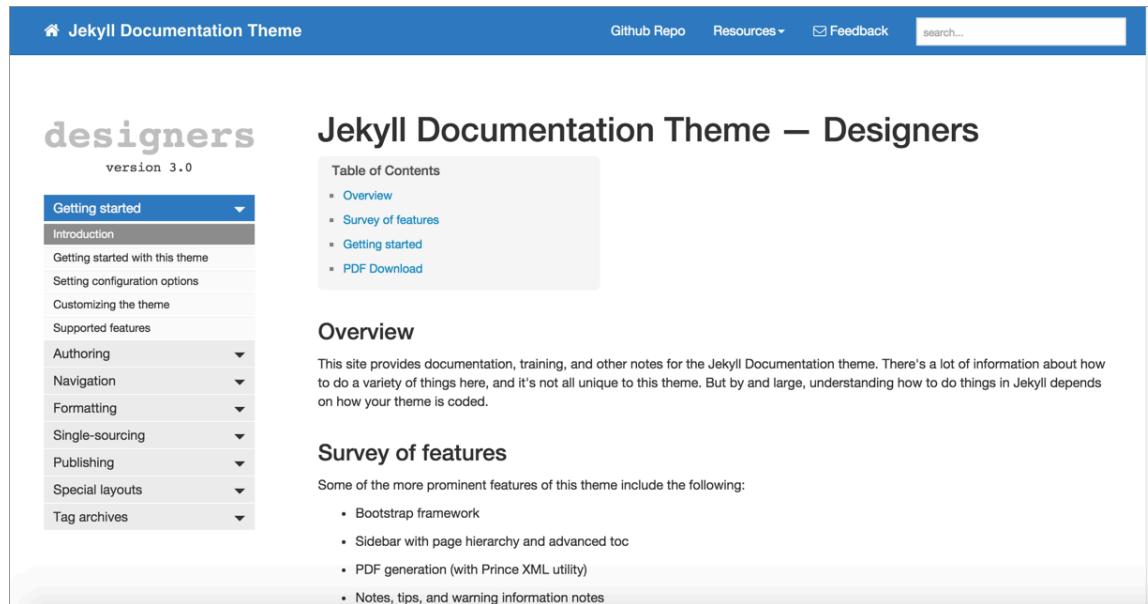
Static site generators

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](http://www.staticgen.com/) (<http://www.staticgen.com/>).

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:



The screenshot shows a web browser displaying the "Jekyll Documentation Theme — Designers" website. The header is blue with the title "Jekyll Documentation Theme". Below the header, there is a navigation bar with links to "Github Repo", "Resources", "Feedback", and a search bar. The main content area has a sidebar on the left titled "designers" with "version 3.0". The sidebar contains a "Getting started" dropdown menu with options like "Introduction", "Getting started with this theme", "Setting configuration options", "Customizing the theme", and "Supported features". Under "Supported features", there 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". It 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" and "Survey of features". The "Overview" section contains a paragraph about the theme's documentation and training, and a list of its features: Bootstrap framework, Sidebar with page hierarchy and advanced toc, PDF generation (with Prince XML utility), and Notes, tips, and warning information notes. The "Survey of features" section lists some prominent features of the theme.

(<http://idratherbewriting.com/documentation-theme-jekyll/>)

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.

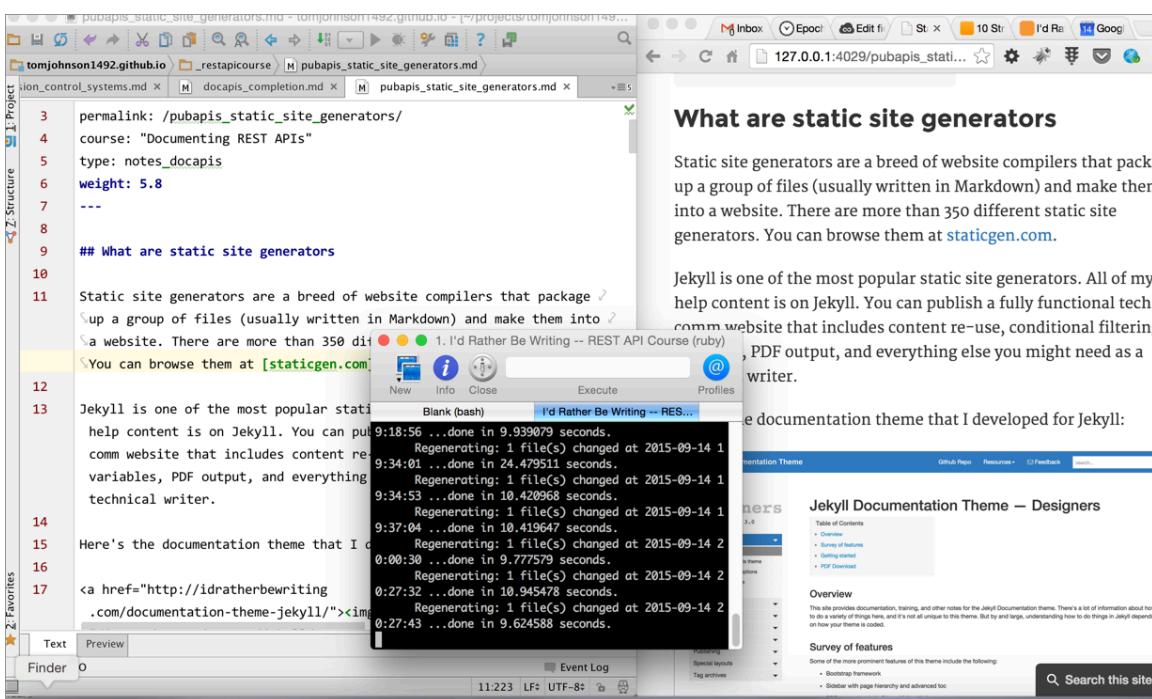
With static site generators, you can do the following:

- 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](https://github.com/tomjohnson1492/documentation-theme-jekyll) (<https://github.com/tomjohnson1492/documentation-theme-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

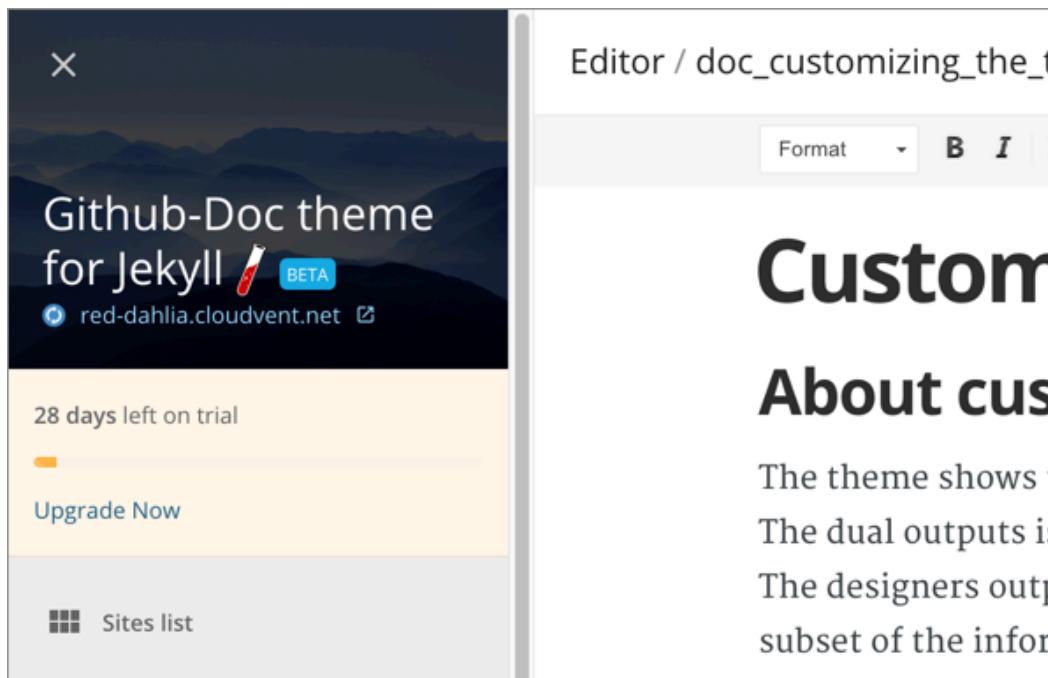
<https://github.com/{your github username}/documentation-theme-jekyll>.

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](https://help.github.com/articles/fork-a-repo/) (<https://help.github.com/articles/fork-a-repo/>).

2. Sign up for a free account at [CloudCannon](http://cloudcannon.com) (<http://cloudcannon.com>).
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](http://idratherbewriting.com/documentation-theme-jekyll) (<http://idratherbewriting.com/documentation-theme-jekyll>).

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 provide 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 \(<https://github.com/CloudCannon/Aviator-Jekyll-Theme>\)](https://github.com/CloudCannon/Aviator-Jekyll-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 course (Documenting REST APIs), you already have a new endpoint called surfreport.

The screenshot shows a Jekyll theme for API documentation. On the left sidebar, there are sections for 'Documentation' (with 'Getting Started' highlighted), 'Authentication', and 'Errors'. Under 'APIs', there are links for '/books' (GET, POST, PUT, DELETE). The main content area has two sections: 'Getting Started' (intro to the API, warning about development status) and 'Authentication' (instructions for API key generation). A code snippet for jQuery cURL is shown on the right.

(<https://github.com/CloudCannon/Aviator-Jekyll-Theme>)

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 \(<https://github.com/CloudCannon/Aviator-Jekyll-Theme>\)](https://github.com/CloudCannon/Aviator-Jekyll-Theme) and click the **Download ZIP** button.

The screenshot shows the GitHub repository page for 'CloudCannon / Aviator-Jekyll-Theme'. It displays basic repository statistics (4 commits, 1 branch, 0 releases, 1 contributor) and a list of recent commits. An orange arrow points to the 'Download ZIP' button located at the bottom right of the commit list.

(<https://github.com/CloudCannon/Aviator-Jekyll-Theme>)

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](https://www.mashape.com/fyhao/weather-13#weatherdata) (<https://www.mashape.com/fyhao/weather-13#weatherdata>). 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). You can also [download the file here](http://idratherbewriting.com/files/publishingapidocs/1-6_weatherdata.md) (http://idratherbewriting.com/files/publishingapidocs/1-6_weatherdata.md). 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/*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-w
ea.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/Tebra
u_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<BR /><b>Forecast:</b><BR />\nSat - Scattered Thunderstorms.
High: 32 Low: 26<br />\nSun - Thunderstorms. High: 33 Low: 27<br />\nM
on - Scattered Thunderstorms. High: 32 Low: 26<br />\nTue - Thundersto
rms. High: 32 Low: 26<br />\nWed - Scattered Thunderstorms. High: 32 L
ow: 27<br />\n<br />\n<a href=\"http://us.rd.yahoo.com/dailynews/rss/w
eather/Tebrau_MY/*http://weather.yahoo.com/forecast/MYXX0004_c.htm
l\">Full Forecast at Yahoo! Weather</a><BR /><BR />\n(provided by <a hre
f=\"http://www.weather.com\">The Weather Channel</a>)<br />\n",
        "forecast": [
            {
                "code": "38",
                "date": "3 May 2014",
                "day": "Sat",
                "high": "32",
                "low": "26",
                "precip": "0",
                "wind": "0"
            }
        ]
    }
}
```

```
        "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"
}
}
}
}
]
---
```

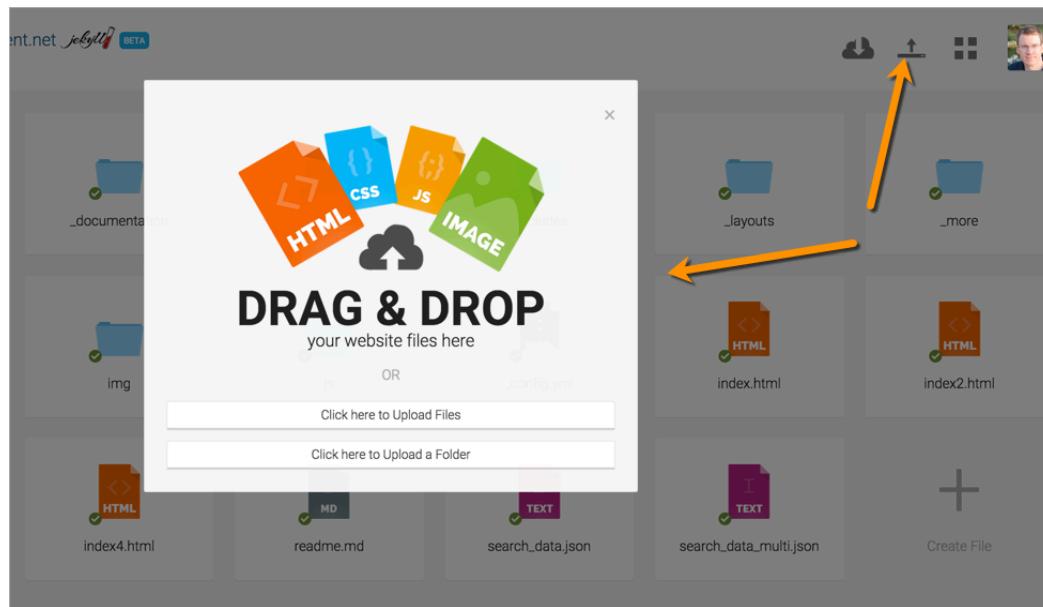
```
<div class="code-viewer">

<pre data-language="cURL">
curl --get --include 'https://simple-weather.p.mashape.com/weatherdata?lat=1.0&lng=1.0' \
-H 'X-Mashape-Key: EF3g83pKnzmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p' \
-H 'Accept: application/json'
</pre>

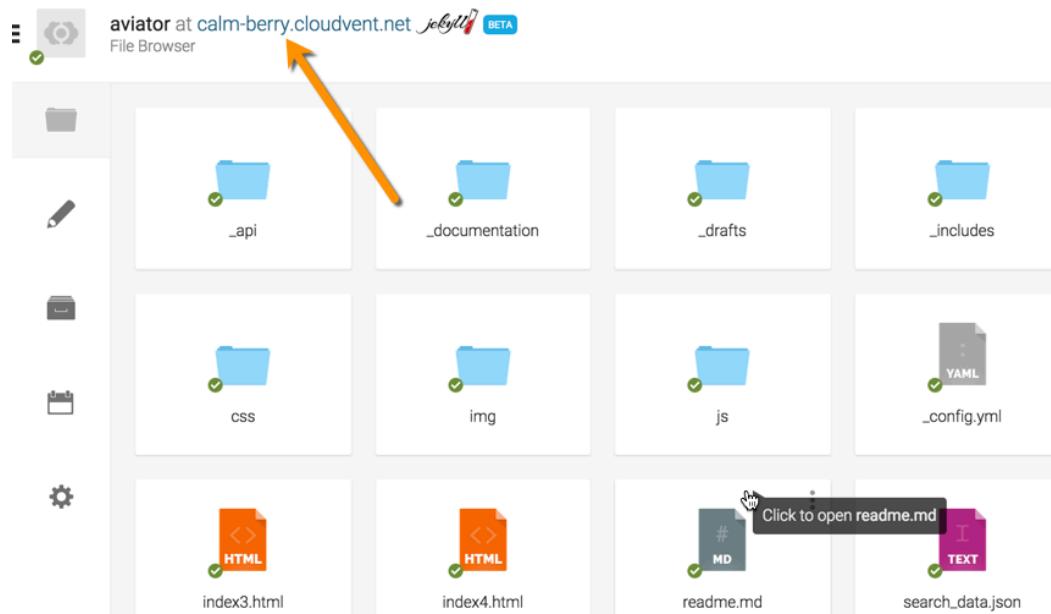
</div>
```

c. Publish your Jekyll project on CloudCannon

1. Go to <http://cloudcannon.com> (<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:

```

Documentation
Getting Started
Authentication
Errors

APIs
/weatherdata GET
/books GET
/books POST
/books/:id GET
/books/:id PUT
/books/:id DELETE

/weatherdata
  lat string
    Required. Latitude.
  lng string
    Required. Longitude.

curl
curl --get --include 'https://simple-weather.p.mashape.com/weatherdata?
lat=1.0&lng=1.0' \
-H 'X-Mashape-Key: 
EP3g83pKnmshgokaf83V6JB6QyTpIcGrrdjsnczTkkYgYrp8p' \
-H 'Accept: application/json'

{
  "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/weather/MIXX0004_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",
          ...
        }
      }
    ]
  }
}

```

You can see my site at <http://delightful-nightingale.cloudvent.net/> (<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://dev.iron.io/>)
- <http://healthcare.gov> (<http://healthcare.gov>)
- <http://getbootstrap.com> (<http://getbootstrap.com>)
- <http://iamnotarealprogrammer.com/> (<http://iamnotarealprogrammer.com/>)
- <http://jekyllrb.com/> (<http://jekyllrb.com/>)
- <http://docs.balsamiq.com> (<http://docs.balsamiq.com>)

Readme.io

You can publish documentation on hosted platforms specifically built for API and developer documentation. Two promising platforms are the following:

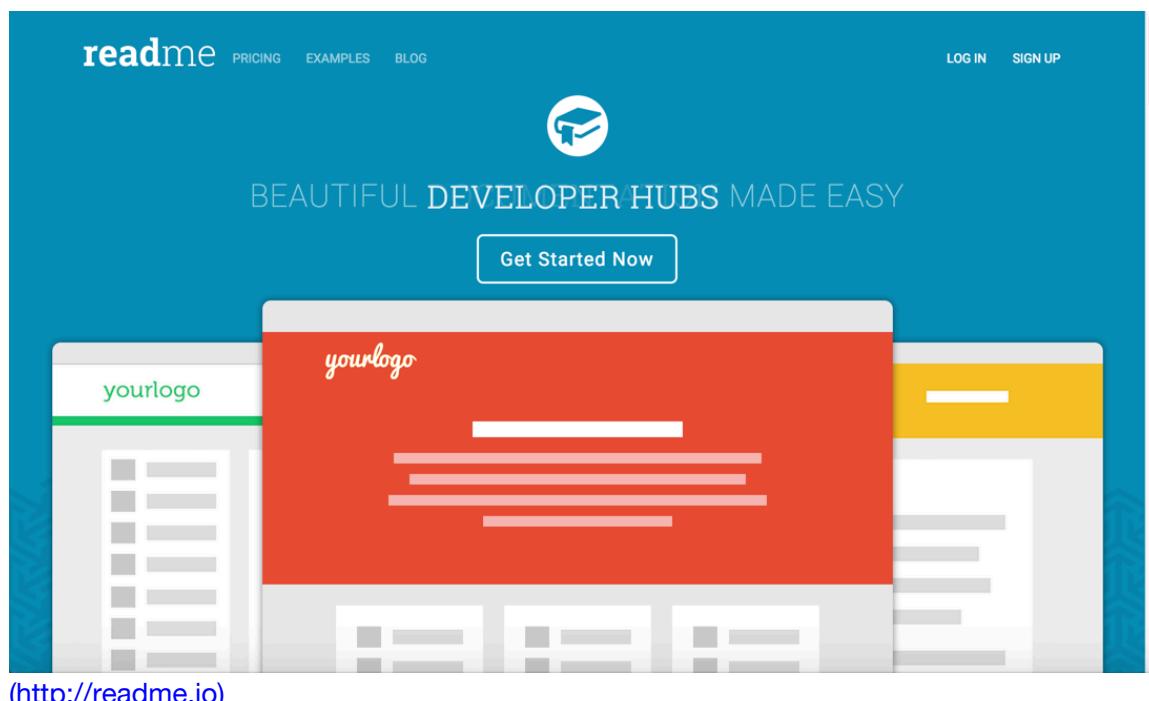
- [readme.io \(<http://readme.io>\)](http://readme.io)
- [readthedocs.com \(<http://readthedocs.com>\)](http://readthedocs.com)

No need to spend time developing your own site

If you consider how much time it requires to build, maintain, troubleshoot, etc. your own website, then it really does make sense to consider an existing third-party platform where someone has already built all of this out for you.

Publish endpoint documentation on [readme.io](#)

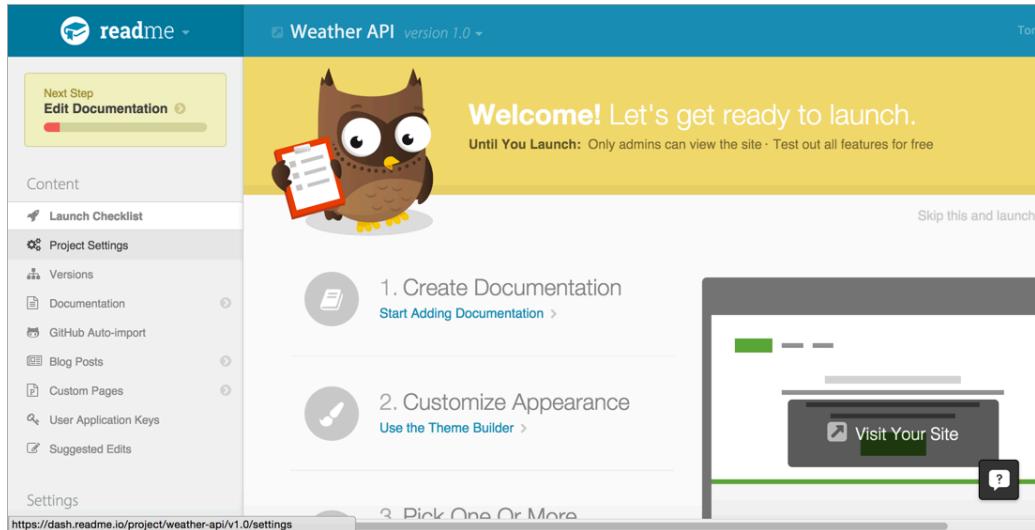
In this tutorial we'll explore how to publish content on [readme.io](#) in more depth.



In this workshop activity, you'll publish this [weatherdata endpoint documentation](https://www.mashape.com/fyhao/weather-13#weatherdata) (<https://www.mashape.com/fyhao/weather-13#weatherdata>) on [readme.io](#).

a. Set up a readme.io project

1. Click the **Sign Up** button in the upper-right corner and sign up for an account.
2. Add a Project Name (e.g., Weather API), Subdomain (e.g., weather-api), and Project Logo. Then click **Create Project**.



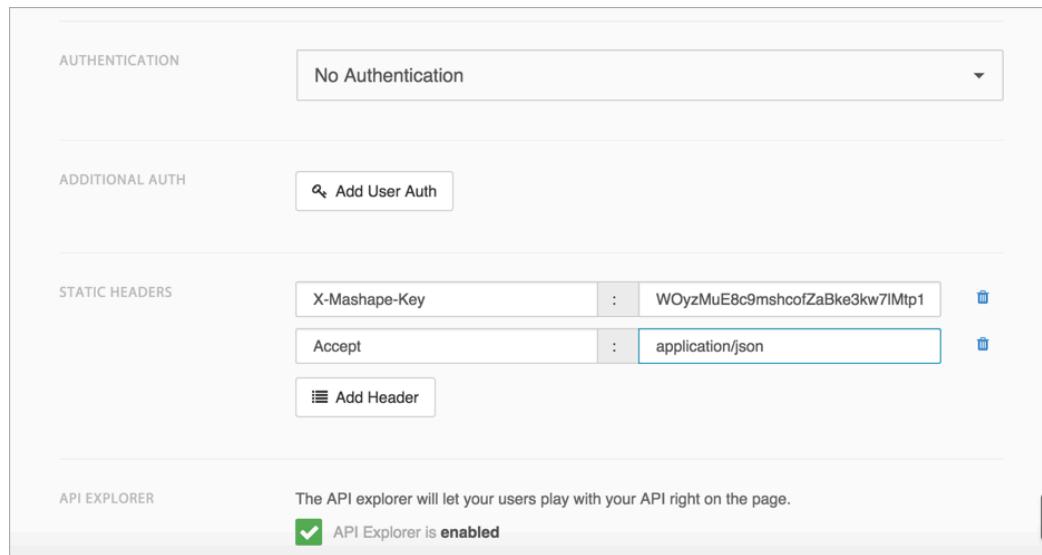
b. Configure API settings

1. In the left sidebar, under Settings, click **API Settings**.

This is where you add the authentication information necessary for making calls to the API.

2. For the API Base URL, enter <https://simple-weather.p.mashape.com>.
3. Leave the other settings not mentioned here at the defaults.
4. In the Static Headers section, click **Add Header** add these two headers as key-value pairs in the appropriate fields:

```
X-Mashape-Key APIKEY  
Accept application/json
```



5. Select the **API Explorer** check box (if it's not already selected).
6. Click **Save**.

c. Add endpoint documentation

1. In the left sidebar, click **Documentation**.
2. Click **+** to add a new page, and choose the **RESTful API** template.
3. Select the **GET** method next to title.
4. Add in the documentation from the [weatherdata endpoint documentation](https://www.mashape.com/fyhao/weather-13#weatherdata) (<https://www.mashape.com/fyhao/weather-13#weatherdata>). For example, add the description, parameters, CURL call, and response.

The screenshot shows the 'weatherdata' endpoint documentation page. The title is 'weatherdata' with a 'GET' method selected. The 'Save' button is at the top right. The main area is titled 'API Setup'.

- ENDPOINT:** https://simple-weather.p.mashape.com/weatherdata. A note says: 'Use :param_name to include params in URL. Use :version to dynamically include the version id. Set the API base URL on the settings page.'
- PARAMS:** A table with two rows:

lat	*	String	37.3708905	latitude
lng	*	String	-121.9675525	longitude

 An 'Add Param' button is below the table.
- EXAMPLE:** A code block containing a CURL command:


```
curl --get --include 'https://simple-weather.p.mashape.com/weatherdata?lat=1.0&lng=1.0' \
-H 'X-Mashape-Key: EF3g83pKnmshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p' \
-H 'Accept: application/json'
```

5. Click **Save**.

- At the top of the screen, click the project name to view the site.

d. Interact with the documentation

- Click **Documentation** in the header to go to your site.
- Click the **Weatherdata** endpoint in the sidebar.
- In the Try It Out section, insert some values into the lat and lng fields, and then click **Try It**.

The screenshot shows the 'Documentation' page for a project. In the sidebar, 'Weatherdata' is selected. The main area is titled 'Try It Out'. It has two input fields: 'lat*' containing '37.3708905' and 'lng*' containing '-121.9675525'. Below these is a URL field with 'https://simple-weather.p.mashape.com/weatherdata' and a 'Try It!' button. A modal window is open, showing a 200 OK response with the following JSON data:

```
{ "query": { "count": 1, "created": "2015-06-17T06:23:38Z", "lang": "en-US", "results": { "channel": { "title": "Yahoo! Weather - Santa Clara, CA", "link": "http://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA1018_c.html", "description": "Yahoo! Weather for Santa Clara, CA", "language": "en-us", "lastBuildDate": "Tue, 16 Jun 2015 8:53 pm PDT" } } }
```

The experience is similar to Swagger in that the response appears directly in the documentation. This API Explorer gives you a sense of the data returned by the API.

Limitations with Readme.io

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.

Miredot

One of the tools you can use to generate API documentation from source – as long as your source is Java-based – is Miredot.

Miredot overview

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 the MireDot Petstore-1.6.1 application. The top navigation bar includes links for 'Issues (18)', 'miredot', 'Resources', and 'Search'. The main content area features a 'Welcome to the MireDot demo project' message. Below it, a section titled 'GET ALL CATEGORIES' provides details about the endpoint, including the URL (`http://mydomain.com/catalog/category/`) and a sample response body showing JSON data for categories like 'cats' and 'dogs'. The sidebar on the right contains a navigation tree with nodes for 'catalog' (expanded), 'category' (expanded), 'item' (collapsed), 'product' (collapsed), 'sales' (collapsed), and 'test' (collapsed). The 'category' node has child nodes for '/GET', 'POST', '(id)', 'GET', 'PUT', and 'DELETE'.

(<http://miredot.com/exampledocs/>)

You can read the [Getting started guide](http://miredot.com/docs/getting-started/) (<http://miredot.com/docs/getting-started/>) 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](http://www.miredot.com/docs/supportedframeworks/) (<http://www.miredot.com/docs/supportedframeworks/>) for a complete set of supported annotations.

Example annotations

Here's an example of what these annotations look like. Look at the [CatalogService.java](https://github.com/Qmino/miredot-petstore/blob/master/src/main/java/com/qmino/miredot/petstore/service/CatalogService.java) (<https://github.com/Qmino/miredot-petstore/blob/master/src/main/java/com/qmino/miredot/petstore/service/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.

Activity: Explore Miredot's output

1. First browse the [Miredot sample code \(<https://github.com/Qmino/miredot-petstore/blob/master/src/main/java/com/qmino/miredot/petstore/service/CatalogService.java>\)](https://github.com/Qmino/miredot-petstore/blob/master/src/main/java/com/qmino/miredot/petstore/service/CatalogService.java).
2. To see how this information gets rendered in the Miredot output, go to the [Petstore example docs \(<http://miredot.com/exampledocs/>\)](http://miredot.com/exampledocs/), expand **Catalog > Category** on the right, and then select **PUT**. Or go directly [here \(<http://www.miredot.com/exampledocs/#268738548>\)](http://www.miredot.com/exampledocs/#268738548).

The screenshot shows the Miredot API documentation interface. At the top, it says "UPDATE CATEGORY NAME AND DESCRIPTION" and has a "expand / collapse all" button. Below that, a note says "Update category name and description. Cannot be used to edit products in this category." A blue "PUT" button is followed by the URL "http://mydomain.com/catalog/category/{id}/". The "Path parameters" section shows "id" as a "number" type with the description "The id of the category that will be updated". The "Body" section shows "The category details" with an "Accept" field set to "application/xml,application/json". Below that is a JSON schema:

```
{  
  subCategory:  
    "cats"  
    "dogs"  
  description: string //the new description  
  name: string //The name of the category  
  recommendedProduct: //A product that is recommended for this category [...]  
  {  
    type:  
      ".PromotionalProduct"  
  }  
}
```

There are "Hide descriptions" and "..." buttons next to the schema. On the far right, there is a vertical scroll bar.

(<http://www.miredot.com/exampledocs/#268738548>)

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.

Custom UX solutions

If you want to build a beautiful API doc website that rivals sites such as [Parse.com](http://parse.com) (<http://parse.com>) and others, you'll most likely need to involve a UX engineer to build it. Fortunately, this is a solution that many UX engineers and other web developers are usually excited to tackle.

When it makes sense to partner with UX

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.



Solution at Badgeville

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.

Developing custom solutions

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.

Help authoring tools

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. Sure, you can use these tools to create API documentation.

Example

Here's a sample help output from Flare for the Photobucket API:

The screenshot shows a detailed API documentation page for the Photobucket API. The left sidebar contains a table of contents with sections like 'Using the Help', 'What's New', 'Getting Started', 'Examples', 'Methods', and 'Albums'. The main content area is titled 'Create New Album' and describes how to create a new sub-album. It includes sections for 'User Login Required', 'HTTP Method' (POST), 'REST Path' (/album/[identifier]), 'Parameters' (with a table showing identifier as String and name as String), and 'Request Example' (a POST request to http://api123.photobucket.com/album/pbapi). A 'Response Example' section is also present. The bottom of the page has links to 'TOC', 'Index', 'Search', and 'Glossary'.

(https://pic.photobucket.com/dev_help/WebHelpPublic/PhotobucketPublicHelp_Left.htm#CSHID=FAQ/FAQOverview.htm|StartTopic=Content/FAQ/FAQOverview.htm|SkinName=WebHelp)

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.

+ 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.

+ 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.

+ 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

Using a HAT also presents some disadvantages. First, almost no HAT runs on a Mac. But many developers and designers prefer Macs because they have a much better development platform. 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.

- 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, 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.

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.



(<https://flic.kr/p/QMVMw>)

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."

Design Patterns overview

In this section:

- Design pattern: Structure and templates (page 354)
- Website platform (page 356)
- Abundant code samples (page 358)
- Long-ish pages (page 360)
- API Interactivity (page 363)
- Design patterns (page 352)
- Adding an Edit on Github button (page 365)
- Challenging factors (page 377)

Design patterns

Design patterns are common themes in the way something is designed. In looking over the many API doc sites, I tried to find some common design patterns in the way the content was published. I already mentioned the division between guides, tutorials, and reference documentation. Here I want to explore more design-specific elements in API doc sites.



(<https://flic.kr/p/ssQqiL>)

Several design patterns with API docs

Here are several design patterns with API doc sites:

- Structure and templates
- Website platform
- Abundant code examples
- Longish pages
- Interactive API explorers

I'll explore each of these elements in depth in upcoming pages.

Some non-patterns

Here are some non-patterns. By this, I mean these are elements that aren't as common in API doc sites:

- PDF output
- Mobile display

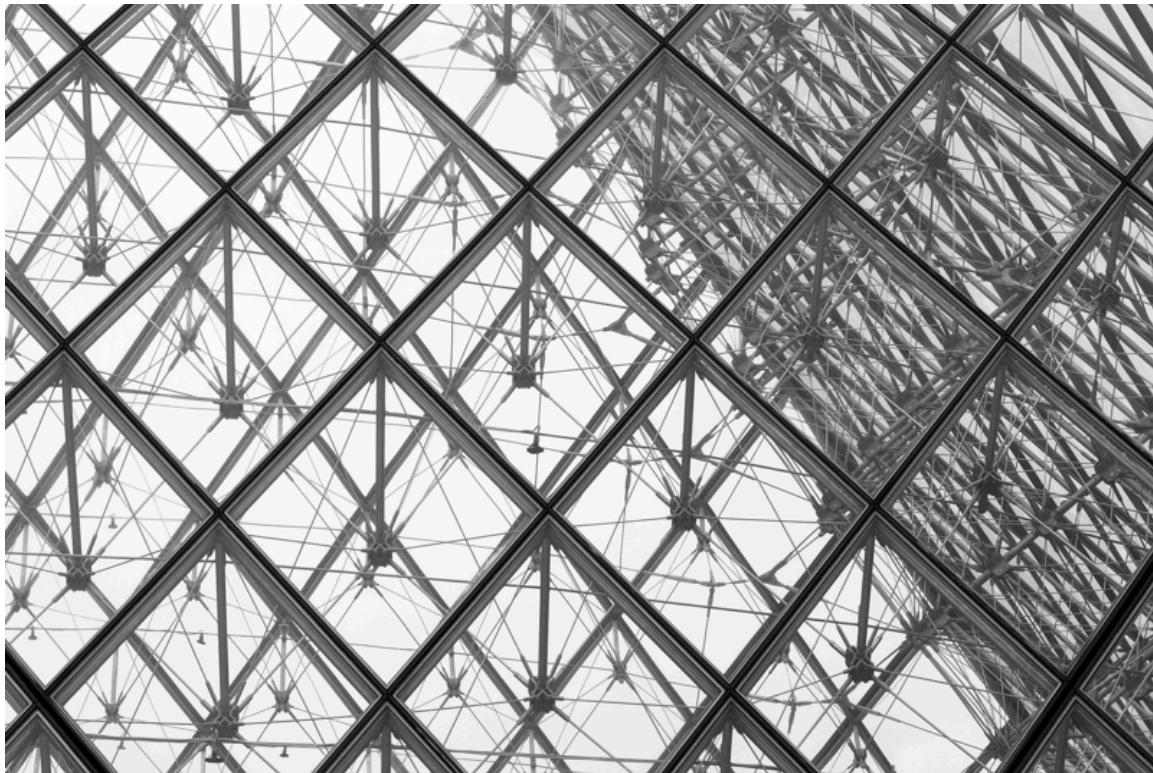
- Comments on pages
- Video tutorials

By non-patterns, it's not to say these elements aren't a good idea. But generally they aren't emphasized in many of the API doc sites.

Design pattern: Structure and templates

If you have a lot of endpoints, you can construct a template that forces specific values in the same template. This is important because you want to establish a consistency with each endpoint. You're basically filling in the blanks.

You could just remember to add the exact same sections on each page, but this requires more manual consistency.



(<https://flic.kr/p/oFD6MM>)

Pushing values into more stylized outputs

You might want to 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.

Templates in Jekyll

Different authoring tools have different ways of processing templates. In Jekyll, a static site generator, this is how you do it.

In the frontmatter of a page, you list out the key value pairs:

```
resource_name: surfreport
resource_description: Gets the surf conditions for a specific beach.
endpoint: /surfreport
```

And so on.

You 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>
```

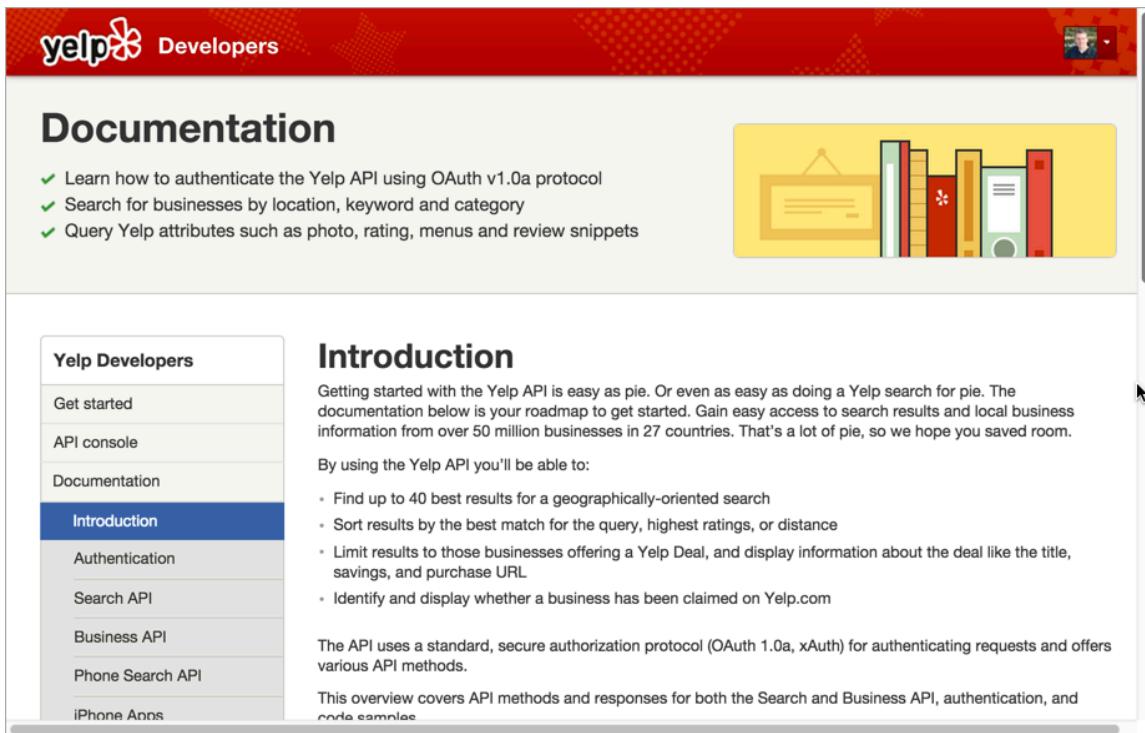
Templates make it easy to change display globally

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 something, 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 really automate the template process.

Website platform

Many API doc sites provide one integrated website to find all of the information. You usually aren't opening help in a new window, separate from the other content. The website is branded with the same look and feel as the product. Here's an example from Yelp:



The screenshot shows the Yelp Developers Documentation page. At the top, there's a red header bar with the Yelp logo and the word "Developers". Below the header, the main title is "Documentation". To the right of the title is a yellow box containing icons of a clipboard, books, and a gear. On the left, there's a sidebar with a navigation menu. The "Introduction" option is highlighted with a blue background. The main content area has a section titled "Introduction" with text about getting started with the Yelp API and its features. There's also a note about the API using OAuth 1.0a and xAuth.

[\(https://www.yelp.com/developers/documentation\)](https://www.yelp.com/developers/documentation)

Documentation as product interface

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.

Integrating information across the entire site

One of the challenges in using documentation generated from Swagger, Miredot, 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 like Drupal or some other web framework?

Abundant code samples

More than anything else, developers love code examples. Usually the more code you can add to your documentation, the better.

Here's an example from Evernote's API:

The screenshot shows a web browser displaying the Evernote Developers API documentation. The URL is [\(https://dev.evernote.com/doc/articles/note-sharing.php\)](https://dev.evernote.com/doc/articles/note-sharing.php). The page title is "Sharing (and Un-Sharing) Notes". Below the title, there is a brief description: "How to start and stop sharing a single note, as well as how to retrieve a list all of the shared notes in an Evernote account." A section titled "Single Note Sharing" contains text about sharing single notes and a list of requirements. Below this is a code block with tabs for Python, Objective-C, Ruby, PHP, Java, and Node.js. The Python tab is selected, showing the following code:

```

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

```

The right sidebar contains links to various API reference sections: API Reference, iOS SDK Reference, Android SDK Reference, Quick-start Guides (Android, JavaScript, Python, Ruby, iOS), Core Concepts (Overview, Data Structure, Error Handling, The Sandbox), Authentication (Developer Tokens, OAuth, Permissions, Revocation and Expiration), Rate Limits (Overview, Best Practices), and Notes (Creating Notes, Sharing Notes, Reminders, Read-only Notes).

(<https://dev.evernote.com/doc/articles/note-sharing.php>)

The writers at Parse emphasize the importance of code samples (<http://blog.parse.com/learn/engineering/designing-great-api-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.

Syntax highlighting

For code samples, you want to incorporate syntax highlighting. There are numerous syntax highlighters that you can usually incorporate into your platform. For example, Jekyll uses either Pygments or Rouge. These highlighters have stylesheets prepared to highlight languages based on specific syntax.

When you include a code sample, you usually instruct the syntax highlighter what language to use. If you don't have access to a syntax highlighter for your platform, you can always [manually add the highlighting using syntax highlighter library \(<http://code.tutsplus.com/tutorials/quick-tip-how-to-add-syntax-highlighting-to-any-project--net-21099>\)](http://code.tutsplus.com/tutorials/quick-tip-how-to-add-syntax-highlighting-to-any-project--net-21099).

Code formatting

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.

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 \(<http://google.github.io/styleguide/javascriptguide.xml>\)](http://google.github.io/styleguide/javascriptguide.xml).

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.

Long-ish pages

One of the most stark differences between regular GUI documentation and developer documentation is that developer doc pages tend to be longer. In a [post on designing great API docs](http://blog.parse.com/learn/engineering/designing-great-api-docs/) (<http://blog.parse.com/learn/engineering/designing-great-api-docs/>), the writers at Parse explain

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.

Examples of long pages

Here's the Backbone.js documentation:

Backbone.js (1.0.0)

- » GitHub Repository
- » Annotated Source

Introduction

Upgrading

Events

- on
- off
- trigger
- once
- listenTo
- stopListening
- listenToOnce
- Catalog of Built-in Events

Model

- extend
- constructor / initialize
- get
- set
- escape
- has
- unset
- clear
- id
- idAttribute
- cid
- attributes

BACKBONE.JS

Backbone.js gives structure to web applications by providing **models** with key-value binding and custom events, **collections** with a rich API of enumerable functions, **views** with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

The project is hosted on [GitHub](#), and the [annotated source code](#) is available, as well as an online [test suite](#), an [example application](#), a [list of tutorials](#) and a [long list of real-world projects](#) that use Backbone. Backbone is available for use under the [MIT software license](#).

You can report bugs and discuss features on the [GitHub issues page](#), on Freenode IRC in the `#documentcloud` channel, post questions to the [Google Group](#), add pages to the [wiki](#) or send tweets to [@documentcloud](#).

Backbone is an open-source component of DocumentCloud.

Downloads & Dependencies (Right-click, and use "Save As")

(<http://documentcloud.github.io/backbone/>)

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

MY SUBREDDITS ▾ FRONT - ALL - RANDOM | FUNNY - PICS - AWW - TODAYILEARNED - GAMING - VIDEOS - GIFS - NEWS - ASKREDDIT - WORLDNEWS - SHOWERTHOUGHTS - MILFIDYINTEREST EDIT »

reddit API DOCUMENTATION freefromlimitations (1) | preferences | logout

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

(<https://www.reddit.com/dev/api>)

Why long pages?

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 search altogether. This is partly because Google does such a better job at search, the in-site search feature of any website is going to be 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, 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 \(<http://getbootstrap.com/javascript/#scrollspy>\)](http://getbootstrap.com/javascript/#scrollspy) 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.

Is it a best practice to make long pages?

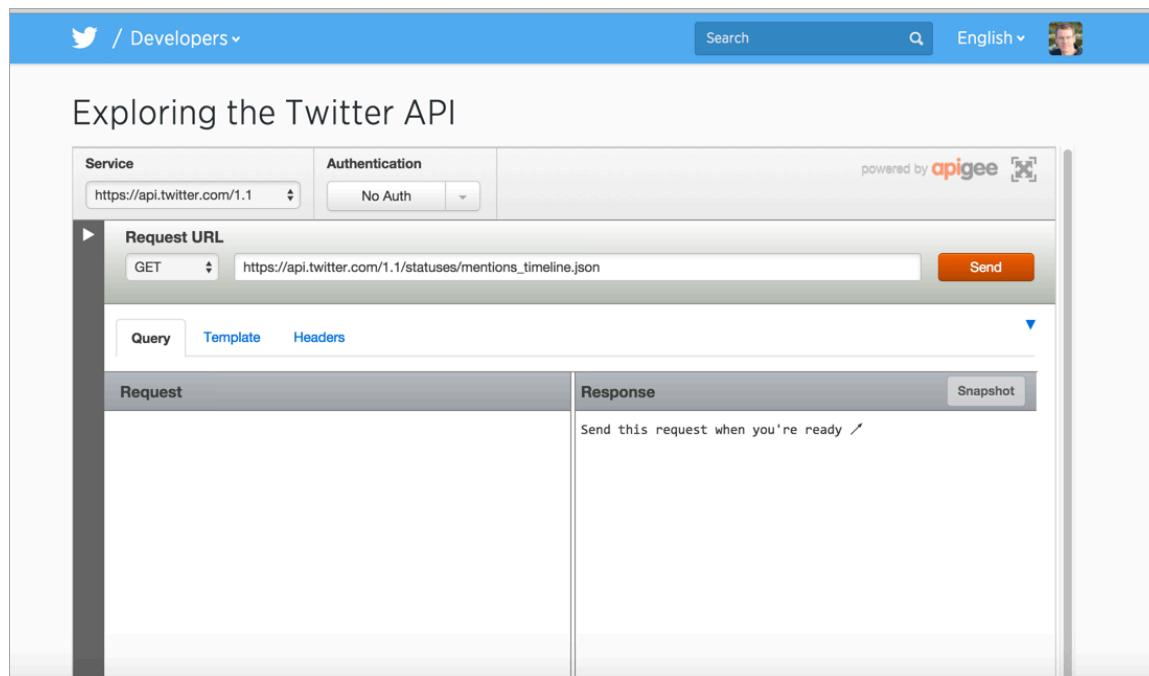
Usually the long pages on a site are the reference pages. Personally, I'm not a fan of listing every endpoint on the same page. Either way you approach it, developers probably won't care that much. They will care much more about the content on the page rather than the page length.

API Interactivity

A recurring feature in many API doc publishing sites is interactivity. Swagger, readme.io, Apiary, and many other platforms allow you to try out calls and see responses.

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 a feasible task for a UI developer.

Here's a sample API explorer from Twitter:



The screenshot shows the Twitter API Explorer interface. At the top, there's a blue header bar with the Twitter logo, 'Developers', a search bar, and language selection ('English'). Below the header, the title 'Exploring the Twitter API' is displayed. The main area has several sections: 'Service' (set to https://api.twitter.com/1.1), 'Authentication' (set to 'No Auth'), and a central workspace. In the workspace, under 'Request URL', a GET request is made to 'https://api.twitter.com/1.1/statuses/mentions_timeline.json'. Below this, tabs for 'Query', 'Template', and 'Headers' are visible. The 'Request' section is empty, and the 'Response' section contains the placeholder text 'Send this request when you're ready.' A 'Snapshot' button is located in the top right corner of the response area. The entire interface is branded with 'powered by apigee'.

(<https://dev.twitter.com/rest/tools/console>)

Novel or actually instructive?

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 to quickly make the request and see the response. The API Explorer provides more of a GUI, however, that makes the endpoints accessible to more people. You don't have to worry about entering exactly the right syntax in your cURL 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.

Dynamically populated code samples with API keys

If your users log in, you can store their API keys and dynamically populate the calls with API keys. Not doing so seems a bit lazy with the user experience. The API key can most likely be a variable that stores the user's API key.

However, if you store customer API keys on your site, this might create authentication and login requirements that make your site more complicated. 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. [readme.io \(`http://readme.io`\)](http://readme.io) is one of the platforms that allows you to store API keys for users and dynamically populate your code samples with them.

Adding an Edit on Github button

One common design pattern in API docs is a button or link that says “Edit on Github.” This button takes users to the source file on Github, where they can edit the content. I’d seen this button on many API doc sites, so I decided to give it a try with my own docs.

Here’s what my docs look like with the Edit on Github button:

The screenshot shows a section of a documentation page. At the top, the title "Getting Started Developing Apps and Games for Amazon Fire TV" is displayed in large orange text. Below the title, there is a button labeled "Edit on GitHub". Underneath the button, a paragraph of text reads: "To get started building apps for Fire TV, first decide whether you want to build an Android app or web app:". Following this paragraph is a bulleted list with two items:

- **Android App:** If you're an Android Java developer, you can use existing tools (like Android Studio) and frameworks (including Unity) to develop apps and games for the 10-foot experience. Sample code, documentation, and guidelines are available to help you make the most of your apps. If you're building a streaming media app, you can use [Fire App Builder](#) — a Java-based Android starter kit — to get up and running quickly.
- **Web App:** If you're an HTML5 web developer, you can leverage the Amazon WebView to develop apps and games. You have the option to build [HTML5 web apps](#), [Cordova apps](#) using the Fire OS port, or [hybrid apps](#). If you're building a streaming media app, you can use the [Web App Starter Kit for Fire TV](#) to get up and running quickly.

(<https://developer.amazon.com/public/solutions/devices/fire-tv/docs/getting-started-developing-apps-and-games-for-amazon-fire-tv>)

When I announced the new Github integration to my project team, the project manager responded enthusiastically about the potential for “crowdsourcing docs.”

The term “crowdsource” hadn’t been in my mind when putting my docs on GitHub, but I nevertheless welcomed the idea.

It seemed I was well on my way to setting up a massive collaboration engine for documentation. I expected developers to immediately start logging issues, submitting pull requests, and more. But even after a few weeks, almost no one made an edit, logged an issue, or contributed any pull requests. (Well, except one mysterious person, who created a small PR but then closed it.)

In a short time period one can’t expect much, especially without promoting and educating users about it. Still, I hoped for a bit more.

I decided to ask the gurus on WTD Slack how they got developers to contribute edits.

Eric Holscher said I need to build up a culture that cares about documentation, and then encourage developers to help.

Anne Gentle said that developers are more likely to contribute to docs when the docs co-exist in the same repos as the code.

I understand that getting developers to contribute to docs is not a change that happens overnight. Culture is always hard to transition.

Putting docs into the same repo as the code might be helpful, but not all docs have corresponding code repos. Additionally, for docs that do have corresponding repos, devs may want to release quarterly (after some QA and verification). In contrast, I want to make doc updates as needed, on the fly.

As I was browsing around on some tech comm blogs, I ran across this [slide deck](#) (<http://www.scriptorium.com/2017/03/tcworld-india-2017-focus-future/>) from Sarah O'Keefe about her keynote at tcworld India.

The slide has a blue header bar with the text "A fad may be part of a trend...". Below the header is a large blue arrow pointing from left to right. To the left of the arrow are four items: "Hand-coding HTML", "Custom XML flavors", "Markdown", and "Virtual reality goggles". To the right of the arrow are four items: "Web publishing", "Structured content", "Mixed authoring environments", and "Augmented reality".

 Scriptorium #tcworldIndia17

(<http://www.scriptorium.com/2017/03/tcworld-india-2017-focus-future/>)

Slide 29

Although I don't have more details about this part of her presentation, the word "fad" got me thinking. Is docs as code a fad? How exactly is the docs-as-code trend different from the wiki trend, which peaked about eight years ago and then floundered?

Rewind – what exactly happened to the wiki trend?

Let's turn back the clock a bit. About 8 years ago, many people in tech comm were excited about wikis. Why? Wikis allowed anyone to contribute to a body of information, without knowledge of anything more than a simple wiki syntax.

Wikis decentralized docs by enabling push-button collaboration and publishing. Sites like Wikipedia showed us the power of collaboration on a massive scale, and wiki platforms took off.

In [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/) (<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/>), I wrote about why I embraced wikis. I embraced wikis because I alone didn't have the needed perspective to address all the information needs that users would have. I wanted to tap into the users' insights and knowledge to complete the gaps in my docs. I wrote:

When the community owns the content, community members can also keep content up to date when the original author flounders. Many times the original author isn't aware of all the places that content is out of date. As community members use the documentation, they often find places that need updating. Because the content is on a wiki, they can quickly and easily make these updates.



Additionally, I wanted to iterate continuously on the docs. Wikis removed the need to build, publish, and deploy content. It all happened magically in the browser when you hit save.

I also noticed that every interesting site online had a massive community driving it:

The potential reward in collaborating with the masses is an idea that shouldn't be underestimated. Almost everything interesting happening on the web comes about through community. Think of any well-known site. Its innovation isn't the technical features of the platform but rather the community.

During the height of wikis, I had my docs on Mediawiki and also wrangled the efforts of a large volunteer community. I worked for a non-profit in Utah at the time, and many people wanted to help out. Although docs were on a wiki and editable by anyone, we primarily tried to crowdsource our blog. We had dozens of volunteers who indicated they wanted to write, and I started assigning articles on various topics to them.

To manage 60+ different non-specialized volunteer writers, I used a lot of index cards pinned up into various quadrants on my cube walls. I invited volunteers to tackle specific subjects they had some knowledge about, and then helped connect them with the right info to write it.

I found the [90-9-1 \(\[https://en.wikipedia.org/wiki/1%25_rule_\\(Internet_culture\\)\]\(https://en.wikipedia.org/wiki/1%25_rule_\(Internet_culture\)\)\)](https://en.wikipedia.org/wiki/1%25_rule_(Internet_culture)) rule about volunteers to be true. 90% of the people on any community project remain quiet lurkers; 9% contribute moderately, and 1% contribute actively. But even with just 9% + 1% of people contributing, the volunteer writing effort showed some fruits. We generated about 18 articles in total (over a period of about 6 months). It seemed superficially successful.

Then I started measuring times. I began to realize that the overhead of getting volunteers to write outweighed the time I spent managing the volunteers. If it took 2-3 hours to manage a volunteer through the writing cycle, or an equivalent 2-3 hours for me to write the article myself, was I really gaining a lot?

One barrier insiders frequently ran up against was the inability to access knowledge behind the firewall. Volunteers didn't have access to the insider resources they needed. They didn't have any of the following:

- JIRA sites
- SharePoint sites
- Test environments
- Internal contacts
- Product roadmaps
- Team standups and other meetings

In short, they lacked the information, resources, and contacts to actually get the information they needed to write the articles. But if I had to get all this info and somehow transfer it to the volunteer writer, I felt I'd done most of the work already.



Besides contributing to the blog, some people also made edits to docs on the wiki. But the same problems existed. All contributions happened post-release, so their edits were applied to existing docs. The people who made edits to existing docs usually did so in awkward ways that required fixing or other edits.

To make an addition to docs, you can't just add a paragraph wherever you want. You have to understand the whole so you know what exists and where. You also have to understand doc style and conventions and voice so that your content can fit seamlessly in. It's not so easy, especially for someone who might not want to dedicate much time to this effort.

Ultimately, I don't remember more than 10 small edits to tech docs overall.

Eventually, the efforts to crowdsource the blog and wiki died. I explained why the effort failed:

I tried the volunteer writing model for more than a year. I kept thinking that if I just figured out the right approach, it could take off into endless productivity. With 100+ writers creating content on a regular basis, we could publish new blog posts daily. We could translate documentation overnight. We could sketch out a skeleton of tasks to write about, and then have volunteers fill in the details. It didn't happen.

The reasons, as I have stated, can be summarized in three bullet points:

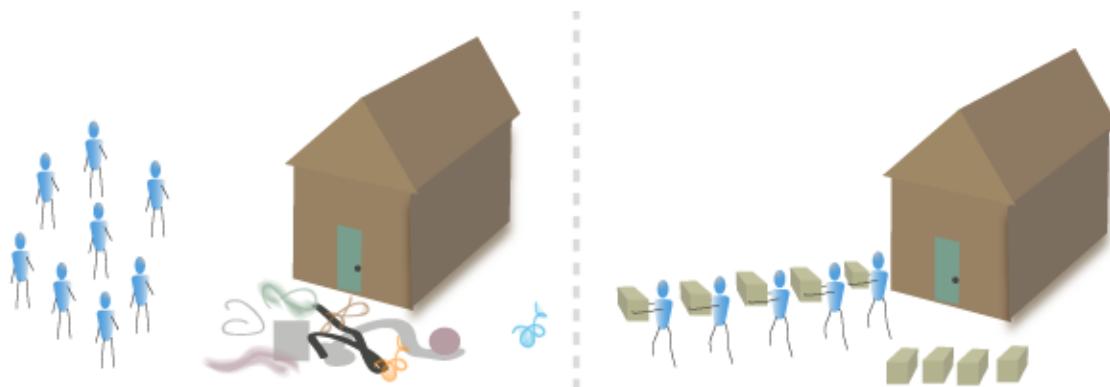
- Writing requires insider knowledge that volunteers lack.
- Professional writing standards are usually beyond volunteer capabilities.
- The management overhead in coordinating volunteer writing barely outperforms the return.

This marked a turning point for me in using wikis as a collaborative platform.

Because collaborative writing failed, I didn't see any need to continue using wikis as a platform, since help authoring tools could provide more robust outputs in different formats.

To make crowdsourcing work, you have to chunk tasks into a little, almost effortless pieces of work. When you multiply that tiny effort by thousands of contributors, you suddenly have something significant.

I compared the crowdsourcing strategy to moving projects. If you put everything in boxes, each volunteer can easily grab and carry a box. But if everything isn't already packed up in easy-to-transport boxes, the volunteers flounder.



But writing doesn't granularize into little effortless chunks/boxes like that, so these efforts fail. For example, you can't ask 100 people to each write 1 sentence in an article.

I also found that you don't need a wiki to enable collaboration. In fact, volunteers preferred to make edits to Word documents shared via Dropbox. I wrote:

Many volunteers who write articles are uncomfortable publishing them directly on the wiki. They prefer to write in Microsoft Word and upload the articles to Dropbox or JIRA.....When I started to realize that wikis weren't a necessary platform for interacting with volunteers, I started to rethink the wiki as a platform. You can have a lot of success with community and collaboration without using a wiki at all. In fact, because the content is not being published live, for the whole world to see and immediately interact with, volunteers actually feel more comfortable contributing.

This is a significant observation that is worth expanding on. In my experience, most engineers don't want their content to make it directly into production, without an editorial workflow. This is their worst fear, actually. They want to do quick brain dumps on internal wiki pages (or in Word docs) and send them to tech writers to clean up, verify, and polish before publishing.

Now back to Docs as Code

With that context, let's look at docs as code again. Suppose you're implementing docs as code for purposes of crowdsourcing docs among engineers. If writing docs isn't the engineers' main job, the result will likely be the same as crowdsourcing efforts with wikis – *dismal*.

Sure, there will be some shimmering lights here and there. Just like some wikis were (or are) successful. But in general, the majority of tech writers who try to crowdsource their docs will find a lack of significant contributions. The contributions they do get will demand editorial and management overhead.

(Note that from a tech comm career perspective, this is good. This means companies can't crowdsource you out of a job. No new tech platform or workflow is going to make that magic river of free documentation start flowing.)

Where wikis took off and skyrocketed

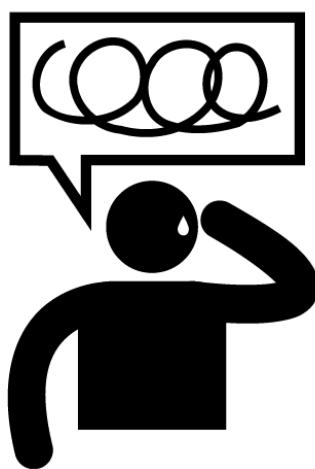
Wikis may have died out with docs as a way to crowdsource info, but they didn't die out as a corporate platform. Wikis, in fact, have become the **standard corporate platform** for employees to share and publish information. Wikis such as Confluence have largely replaced SharePoint.

In my last 3 jobs, wikis were used across the company for employees to write and publish internal information. In my current job at Amazon, there are at least a million internal wiki pages (across multiple wiki platforms – Mediawiki, Confluence, and XWiki). Why? Because wikis give people *easy tools to do their jobs*. (Or at least it simplifies the part of their job that involves sharing information.)

Docs-as-code tooling can have similar gains. It can empower engineers to do their jobs as far as documentation is concerned. If it's the engineer's **job** or **role** to document something, the engineer will be more likely to do it using the coding tools and workflows familiar to them.

Even for non-engineers, giving people a simple tool and publishing workflow can help people author and distribute content. The simple tool might be Markdown syntax and a repository that automatically builds and deploys the content.

Again, forget about crowdsourcing. Docs as code is giving people simple, free, open-source tools to do the writing/publishing tasks they need to do. For that purpose alone, docs as code has the power to move beyond just being a fad. There is real value in empowering common people to author and publish content (without expensive help authoring tools, without knowledge of XML syntax, without years of professional tech writing know-how, etc.).



Proprietary, professional-grade
help authoring tools can be
confusing



Docs-as-code tools can
be simple and easy

Simplification of authoring and publishing tools is at the heart of the docs-as-code movement. No more black boxes that handle your content. No more expensive, proprietary systems to submit to. No more impossible-to-adjust-outputs-unless-you-know XSLT/XSFO-XS-whatever to style your output. You can integrate it all simply, easily, and inexpensively. It works with other web tools and other systems too. You can integrate with the latest web technologies and tools. You can leverage help from modern UX devs.

Here are some of the revolutionary benefits that docs-as-code tools provide:

- Read in plain text with Markdown (or other similar lightweight syntax). You don't need a special editor to provide a WYSIWYG display because of all the angle bracket tags.
- Manage files as plain text (rather than binary files only machines can read). This allows version control systems like Git to do diffs on the content.
- Manage files in version control repos. You can branch content to manage multiple versions without creating various copies of files. You can collaborate in a distributed fashion with multiple team members.
- Automate builds and deployment of your content from repos using scripts. Because you're working with simple text files, you can integrate this content into deployment scripts.

In my GitHub repo, there are active contributions from a writer who is our localization manager. She's been editing the Japanese translations. When we started, she was only marginally familiar with basic HTML. But with docs-as-code tools, she regularly updates Markdown files, commits to the repo, pulls updates, and runs build scripts.

Simplicity has its tradeoffs

Simplicity has its own tradeoffs, though. In exchange for simpler tools, you give up more robust functionality. In a recent comment on a previous post, a reader who was accustomed to using DITA and DocBook for over a decade lamented about the loss of switching to Markdown. [Kate writes \(<http://idratherbewriting.com/2016/10/28/markdown-or-restructuredtext-or-dita-choosing-format-tech-docs/#comment-3193454817>\):](http://idratherbewriting.com/2016/10/28/markdown-or-restructuredtext-or-dita-choosing-format-tech-docs/#comment-3193454817)

I am desperately trying to embrace Markdown after 10 years of using DocBook and then DITA. But I am increasingly finding it difficult.

I'm not certain that building/generating docs from Markdown is that much simpler. I think that a lot of customization is still required to get a nice output. And if people are increasingly adding HTML in order to be able use a table or to create an ID for a section, are we really making things that much simpler? There seems to be a lot of customizations and plugins, etc to do things like check links—these are great, but they can easily become out of date.

I miss having the concept of a map file that contains pointers to my topics.

I miss using oXygen. I miss it's SEARCH. I miss using it's link checker. I miss seeing reused content appear inline in my topic. I miss being able to filter out conditioned content. I miss wysiwig table editors and I miss seeing my images inline. I miss seeing tag errors inline. I am a fairly technical person—I often used the text editor more than the Wysiwyg editors...

While some aspects of authoring are simplified, others – such as developing your own custom output, or engineering search functionality – become more complex. In the end, you have to weigh whether the simplification of some aspects is worth the complication of others.

For example, is writing in a simple Markdown syntax worth it if it makes it more problematic to push your content through translation workflows? Is a system that allows for occasional edits from contributing engineers worth it if you have to develop a number of custom, complicated programming scripts to validate, build, and deploy your content?

Conclusion

If you can cultivate a community where devs contribute their time and attention to docs, great. But if the crowdsourcing efforts fail, it doesn't mean docs as code or even wikis failed. You've still given people simple tools to write and publish documentation.

Although I mentioned crickets as a response for crowdsourcing docs in my current role, it's not entirely fair. We do have several groups outside of our own that learned the doc-as-code tools and published their docs. They didn't edit my docs – they created their own.

Regardless of the participation, I'm planning to leave my docs on GitHub for a lot longer before evaluating the efforts. I suspect it's too early to evaluate much of anything. Most engineers probably don't realize they *can* edit docs, so I have some education efforts to make both internally and externally. Only after I get the collaboration engine moving along can I begin to evaluate whether the tradeoffs for simplicity were worth it.

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. If you have to deal with some of these challenges, you may have to resort to more traditional tech comm tooling.

- Translation
- Content re-use
- Versioning
- Authentication
- PDF

You can handle all of this through a custom platform such as Jekyll, but it's not going to be a push-button experience. It will require a higher degree of technical skill and maneuvering.

With my Jekyll doc theme, I'm single sourcing one of my projects into about 9 different outputs (for different product lines and programming languages). Jekyll provides a templating language called Liquid that allows you to do conditional filtering, content re-use, variables, and more.

To handle PDF, I'm using a tool called Prince that converts a list of HTML pages into a PDF document, complete with running headers and footers, page numbering, and other print styling.

To handle authentication, I upload the content into a Salesforce site.com and use Salesforce as the authentication layer. It's my least favorite part of the solution, but a more integrated authentication will probably involve some engineering resources to help out.

Documenting native library APIs overview

In this section:

- [Overview to native library APIs \(page 379\)](#)
- [Getting the Java source \(page 382\)](#)
- [Java in a nutshell \(page 386\)](#)
- [Create a Javadoc \(page 394\)](#)
- [Javadoc tags \(page 399\)](#)
- [Exploring the Javadoc output \(page 409\)](#)
- [Making edits to Javadoc tags \(page 412\)](#)
- [Doxygen, another document generator \(page 413\)](#)
- [Creating non-ref docs with native library APIs \(page 416\)](#)
- [Course summary \(page 418\)](#)

Overview to native library APIs

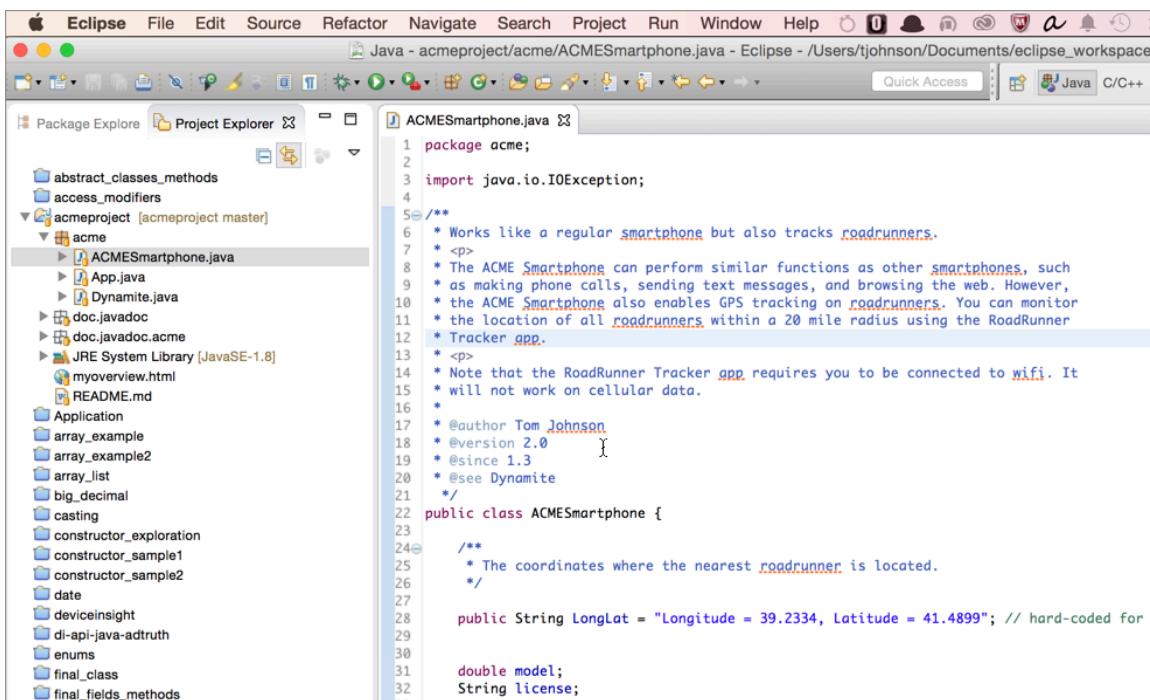
In previous parts of the course, we focused exclusively on REST APIs. Now let's explore native library APIs, which are more common when building native apps.

Characteristics of native library APIs

Native library APIs (also called class-based APIs or just APIs) are notably different in the following ways:

- **Installed locally.** Native library APIs are installed locally, compiled into the programmer's code as an additional library. The programmer can then use the classes, methods, or other functions available in the library. (The API part refers to the *public* classes the users use to access the functions in the library. There are probably lots of helper and utility classes in the Java library that aren't public. The *public* functions that the developer audience uses form the API, since this is how people make use of the library.)
- **No requests and responses.** The classes in the native library API don't use HTTP protocol, nor are there request and responses sent across the web. The native library API is simply a collection of functions that enhance the existing code with more capabilities. It's entirely on-premises.
- **Language specific.** Native library APIs are language specific. There are as many different types of APIs as there are programming languages, more or less. You can have a Java API, C++ API, C# or .NET API, JavaScript API, and so on.
- **Requires some programming knowledge to document.** To understand how the API works, you need to have a general understanding of the programming language the API is written for. You don't need to be a programmer, but you should be familiar with the nuts and bolts of the programming language, the correct terms, how the different parts fit together, and how developers will use the API.

We will focus this section on Java APIs, since they're probably one of the most common. However, many of the concepts and code conventions mentioned here will apply to the other languages, with minor differences.



```
1 package acme;
2
3 import java.io.IOException;
4
5 /**
6 * Works like a regular smartphone but also tracks roadrunners.
7 * <p>
8 * The ACME Smartphone can perform similar functions as other smartphones, such
9 * as making phone calls, sending text messages, and browsing the web. However,
10 * the ACME Smartphone also enables GPS tracking on roadrunners. You can monitor
11 * the location of all roadrunners within a 20 mile radius using the RoadRunner
12 * Tracker app.
13 * <p>
14 * Note that the RoadRunner Tracker app requires you to be connected to wifi. It
15 * will not work on cellular data.
16 *
17 * @author Tom Johnson
18 * @version 2.0
19 * @since 1.3
20 * @see Dynamite
21 */
22 public class ACMESmartphone {
23
24 /**
25 * The coordinates where the nearest roadrunner is located.
26 */
27
28 public String LongLat = "Longitude = 39.2334, Latitude = 41.4899"; // hard-coded for
29
30
31 double model;
32 String license;
```

Do you have to be a programmer to document native library APIs?

Because native library APIs are so dependent on a specific programming language, the documentation is usually written or driven by engineers rather than generalist technical writers. This is one area where it helps to be a former software engineer when doing documentation.

Even so, you don't need to be a programmer. You just need a minimal understanding of the language. Technical writers can contribute a lot here in terms of style, consistency, clarity, tagging, and overall professionalism.

You know what happens when engineers write — the content is cryptic and often incomplete. Usually the developer assumes everyone is as knowledgeable as he or she is, and any kind of extra explanatory detail, examples, cross-references, glossaries, or other helpful information is omitted.

My approach to teaching native library API doc

There are many books and online resources you can consult to learn a specific programming language. This section of the course will not try to teach you Java. However, to understand a bit about Java API documentation (which uses a document generator called Javadoc), you will need some understanding of Java.

To keep the focus on API documentation, we'll take a documentation-centric approach to understanding Java. You'll learn the various parts of Java by looking at a specific Javadoc file and sorting through the main components.

What you need to install

For this part of the course, you need to install the following:

- **Java Development Kit (JDK).** You can [download the JDK here](http://www.oracle.com/technetwork/java/javase/downloads/index.html) (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>). Click the Java button on the left (not Netbeans) and then select the appropriate download for your machine.
- **Eclipse IDE for Java Developers.** Use the [Eclipse Installer to download Eclipse](https://eclipse.org/downloads/) (<https://eclipse.org/downloads/>).

To make sure you have Java installed, you can do the following:

- On Mac: Open Terminal and type `java -version`.
- On Windows: Open a Command Prompt and type `where java`.

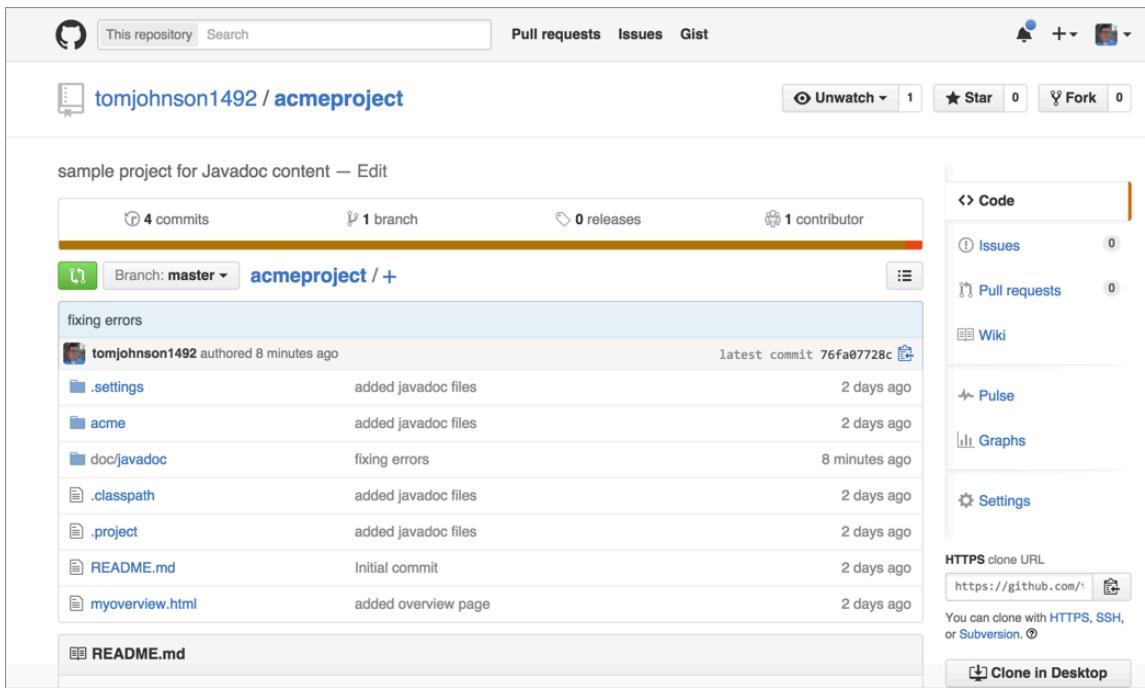
Also, start Eclipse and make sure it doesn't complain that you don't have the JDK.

(Since we'll just be using Java within the context of Eclipse, Windows users don't need to add Java to their class path. But if you want to be able to compile Java from the command line, you can do this.)

Getting the Java source

In order to understand documentation for Java APIs, it helps to have a context of some sort. As such, I created a simple little Java application to demonstrate how the various tags get rendered into the Javadoc.

The sample Java project is a little application about different tools that a coyote will use to capture a roadrunner. There are two classes (ACMESmartphone and Dynamite) and another class file called App that references the classes.



A screenshot of a GitHub repository page. The repository is named "tomjohansson1492 / acmeproject". The page shows a commit history for the "master" branch. The commits are:

- fixing errors (by tomjohansson1492, 8 minutes ago)
- .settings (added javadoc files, 2 days ago)
- acme (added javadoc files, 2 days ago)
- doc/javadoc (fixing errors, 8 minutes ago)
- .classpath (added javadoc files, 2 days ago)
- .project (added javadoc files, 2 days ago)
- README.md (Initial commit, 2 days ago)
- myoverview.html (added overview page, 2 days ago)

The right sidebar shows links for Code, Issues (0), Pull requests (0), Wiki, Pulse, Graphs, and Settings. It also provides an HTTPS clone URL: <https://github.com/tomjohansson1492/acmeproject>.

(<https://github.com/tomjohansson1492/acmeproject>)

This program doesn't really do anything except print little messages to the console, but it's hopefully simple enough to be instructive in its purpose — to demonstrate different doc tags, their placement, and how they get rendered in the Javadoc.

Clone the source on Github

One of your immediate challenges to editing Javadoc will be to get the source code into your IDE. The acmeproject is [here on Github](https://github.com/tomjohansson1492/acmeproject) (<https://github.com/tomjohansson1492/acmeproject>).

First clone the source using version control. We covered some version control basics [earlier in the course \(page 257\)](#).

You can clone the source in a couple of ways:

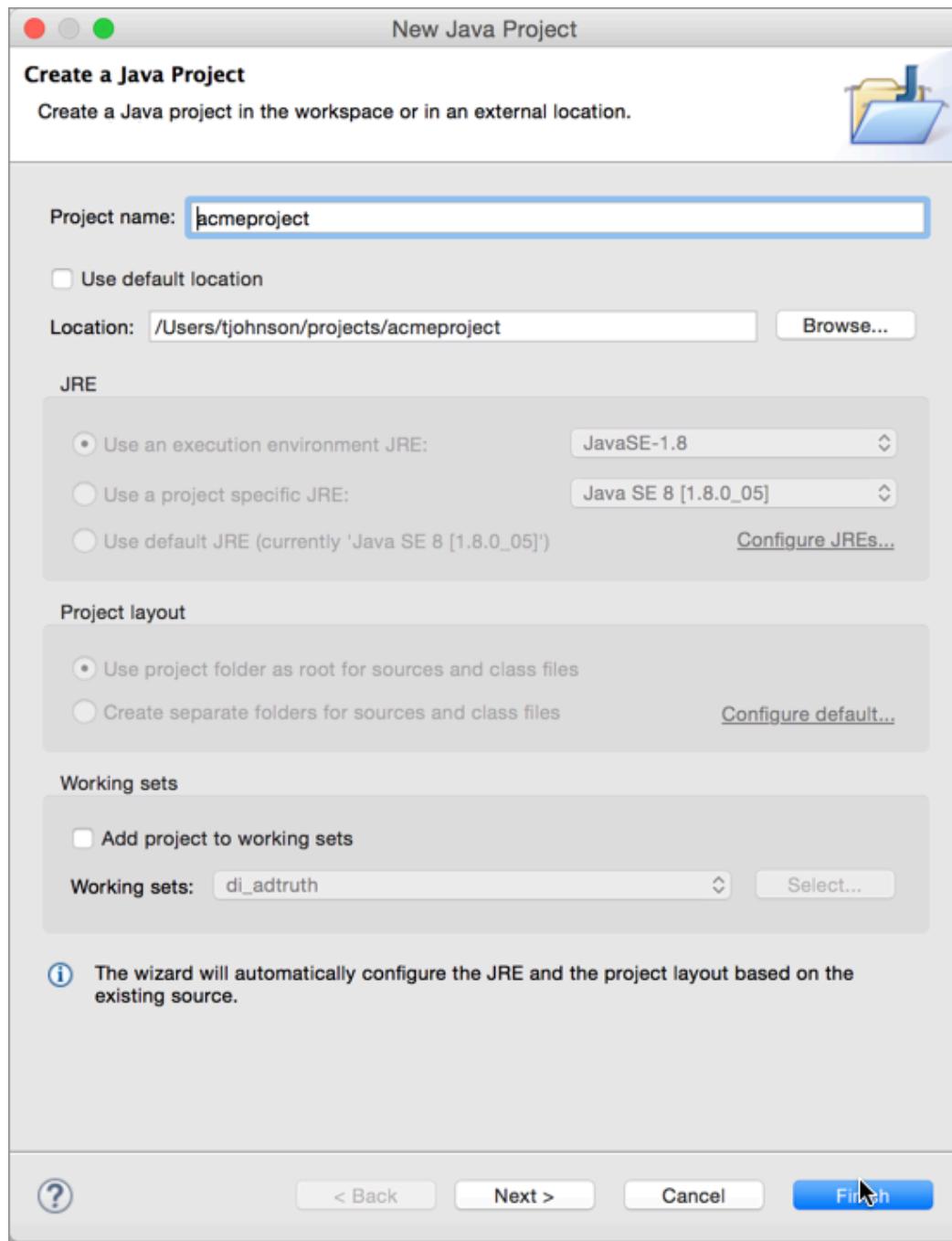
```
git clone https://github.com/tomjohansson1492/acmeproject
```

Or click **Clone in Desktop** and navigate to the right path in Github Desktop.

(If you don't want to clone the source, you could click **Download ZIP** and download the content manually.)

Open the right location in Eclipse

1. After you've cloned or downloaded the Java project, open Eclipse. Go to **File > New Java Project**.
2. Clear the **Use default location** check box, and then browse to where you cloned the Github project.



3. Click **Finish**.

The Java files should be visible within your Eclipse IDE.

Maven projects

Java projects often have a lot of dependencies on packages that are third-party libraries or at least non-standard Java utilities. Rather than requiring users to download these additional packages and add them to their class manually, developers frequently use Maven to manage the packages.

Maven projects use a pom.xml file that defines the dependencies. Eclipse ships with Maven already installed, so when you import a Maven project and install it, the Eclipse Maven plugin will retrieve all of the project dependencies and add them to your project.

The sample project doesn't use Maven, but I wanted to add a note about Maven here anyway because chances are if you're getting a Java project from developers, you won't import it in the way previously described. Instead, you'll import it as an existing Maven project.

To import a Maven project into Eclipse:

1. In Eclipse, go to **File > Import > Maven > Existing Maven Projects** and click **Next**.
2. In the Root Directory field, click **Browse** and browse to the Java project folder (which contains the Maven pom.xml file) and then click **Open**. Then click **Finish** in the dialog box.
3. In the Project Explorer pane in Eclipse, right-click the Java folder and select **Run as Maven Install**.

Maven retrieves the necessary packages and builds the project. If the build is successful, you will see a "BUILD SUCCESS" message in the console. You then use the source code in the built project.

Java in a nutshell

To understand the different components of a Javadoc, you have to first understand a bit about Java. Just being familiar with the names of the different components of Java will allow you to enter conversations and understand code at a high level. When you describe different aspects of sample code, knowing when to call something a class, method, parameter, or enum can be critical to your documentation's credibility.

I'll run you through a brief crash course in the basics. For more detail about learning Java, I recommend consulting lynda.com (<http://lynda.com>) and [safaribooksonline](http://safaribooksonline.com) (<http://safaribooksonline.com>). Below I'll focus on some basic concepts in Java that will be important in understanding the Javadoc tags and elements.

About Java

Java is one of the most common languages used because of its flexibility. Java isn't tied to a specific language platform because Java code compiles into byte code. The platform you deploy your code on contains a Java Virtual Machine (JVM) that interprets the byte code. Hence through JVMs, different platforms can interpret and run Java code. This gives Java more flexibility with different platforms.

Classes

Classes are templates or blueprints that drive pretty much everything in Java. It's easiest to understand classes through an example. Think of a class like a general blueprint of a "bicycle." There are many different types of bicycles (Trek bikes, Specialized bikes, Giant bikes, Raleigh bikes, etc.). But they're all just different instances of the general class of a bicycle.

In Java, you start out by defining classes. Each class is its own file, and begins with a capital letter. The file name matches the class name, which means you have just one class per file.

Each class can contain a number of fields (variables for the class) and methods (subroutines the class can do).

Before the class name, an access modifier indicates how the class can be accessed.

Several options for access modifiers are:

- `public` : Anyone can access
- `private` : Only other packages can access
- `static` : No one can change the class
- `abstract` : The class can't be instantiated, only sub-classed.

Here's an example of a class:

```
public class Bicycle{  
    //code...  
}
```

You mostly need to focus on `public` classes, since these are the classes that will be used by your audience. The `public` classes are the API of the library.

Methods

Methods are subroutines or actions that the class can do. For example, with a bicycle you can pedal, brake, and turn. A class can have as many methods as it needs.

Methods can take arguments, so there are parentheses `()` after the method name. The arguments are variables that are used within the code for that method. For example:

```
add(a, b) {  
    sum = a + b;  
}
```

Methods can return values. When a method finishes, the value can be returned to the caller of the method.

Before the method name, the method indicates what type of data it returns. If the method doesn't return anything, `void` is listed. Other options are `String` or `int`.

Here's an example of some methods for our Bicycle class:

```
class Bicycle {  
  
    void turn() {  
        // code ...  
    }  
    void pedal(int rotations) {  
        System.out.println("Your speed is " + rotations + " per minute");  
    }  
  
    int brake(int force, int weight) {  
        torque == force * weight;  
        return torque;  
    }  
}
```

See how the `brake` method accepts two arguments — `force` and `weight`? These arguments are integers, so Java expects whole numbers here. (You must put the data type before the parameters in the method.) The arguments passed into this method get used to calculate the `torque`. The `torque` is then returned to the caller.

In Javadoc outputs, you'll see methods divided into two groups:

- **Instance methods:** Means that you can use the method in the instance of an object. If the method isn't an instance method, it's considered a static method. Static methods can be used directly from the class without instantiating an object first. Static methods can't be changed by any object or subclass.
- **Concrete methods:** These are methods that can be used when you instantiate an object. If a method isn't concrete, it's called an "abstract method." The only way you use an abstract method is by creating a subclass for the method.

Somewhere in your Java application, users will have something called a `main` method that looks like this:

```
public static void main(String[] args) {  
}
```

Inside the main method is where you add your code to make your program run. This is where the Java Virtual Machine will look to execute the code.

Fields

Fields are variables available within the class. A variable is a placeholder that is populated with a different value depending on what the user wants.

Fields indicate their data types, because all data in Java is "statically typed" (meaning, its format/length is defined) so that the data doesn't take up more space than it needs. Some data types include `byte`, `short`, `long`, `int`, `float`, or `double`. Basically these are numbers or decimals of different sizes. You can also specify a `char`, `string`, or `boolean`.

Here's an example of some fields in class:

```
class Bicycle {  
  
    String brand;  
    int size;  
}
```

Many times fields are “encapsulated” with getter-setter methods, which means their values are set in a protected way. Users call one method to set the field’s value, and another method to get the field’s value. This way you can avoid having users set improper values or incorrect data types for the fields.

Fields that are constant throughout the Java project are called ENUMS. Alternatively, the fields are given `public static final` modifiers.

Objects

Objects are instances of classes. They are the Treks, Raleighs, Specialized, etc., of the Bicycle class.

If I wanted to use the `Bicycle` class, I would create an instance of the class. The instance of the class is called an object. Here’s what it looks like when you “instantiate” the class:

```
Bicycle myBicycle = new Bicycle();
```

You write the class name followed by the object name for the class. Then assign the object to be a new instance of the class. Now you’ve got `myBicycle` to work with.

The object inherits all of the fields and methods available to the class.

You can access fields and methods for the object using dot notation, like this:

```
myBicycle Bicycle = new Bicycle();  
  
myBicycle.brand = "Trek";  
myBicycle.pedal();
```

You probably won’t see many objects in the native library. Instead, the developers who implement the API will create objects. However, if you have a reference implementation or sample code on how to implement the API, you will see a lot of objects.

Constructors

Constructors are methods used to create new instances of the class. The default constructor for the class looks like the one above, with `new Bicycle()`.

The constructor uses the same name as the class and is followed by parentheses (because constructors are methods).

Often classes have constructors that initialize the object with specific values passed in to the constructor.

For example, suppose we had a constructor that initialized the object with the brand and size:

```
public class Bicycle{  
  
    public Bicycle(String brand, int size) {  
        this.brand = model;  
        this.size = size;  
    }  
  
}
```

Now I use this constructor when creating a new Bicycle object:

```
Bicycle myBicycle = new Bicycle ("Trek", 22);
```

It's a best practice to include a constructor even if it's just the default.

Packages

Classes are organized into different packages. Packages are like folders or directories where the classes are stored. Putting classes into packages helps avoid naming conflicts.

When you create your class, if it's in a package called `vehicles`, you list this package at the top of the class:

```
package vehicles  
  
public class Bicycle{  
  
}
```

Classes also set boundaries on access based on the package. If the access modifier did not say `public`, the class would only be accessible to members of the same package. If the access modifier were `protected`, the class would only be accessible to the class, package, and subclasses.

When you want to instantiate the class (and your file is outside the package), you need to import the package into your class, like this:

```
import vehicles  
  
public static void main(String[] args) {  
  
}
```

When packages are contained inside other packages, you access the inner packages with a dot, like this:

```
import transportation.motorless.vehicles.
```

Here I would have a transportation package containing a package called motorless containing a package called vehicles. Package naming conventions are like URLs in reverse (com > yoursite > subdomain).

Maven handles package management for Java projects. Maven will automatically go out and get all the package dependencies for a project when you install a Maven project.

Exceptions

In order to avoid broken code, developers anticipate potential problems with exception handling. Exceptions basically say, if there's an issue here, flag the error with this exception and then continue on through the code.

Different types of errors throw different exceptions. By identifying the type of exception thrown, you can more easily troubleshoot problems when code breaks because you know the specific error that's happening.

You can identify a specific exception the class throws in the class name after the keyword `throws`:

```
public class Bicycle throws IOException {  
}
```

When you indicate the exception here, you list the type of exception using a specific Javadoc tag (explained later).

Inheritance

Some classes can extend other classes. This means a class inherits the properties of another class. When one class extends another class, you'll see a note like this:

```
public class Bicycle extends Vehicle {  
}
```

This means that `Bicycle` inherits all of the properties of `Vehicle` and then can add to them.

Interfaces

An interface is a class that has methods with no code inside the method. Interfaces are intended to be implemented by another class that will insert their own values for the methods. Interfaces are a way of formalizing a class that will have a lot of subclasses, when you want all the subclasses to standardize on common strings and methods.

JAR files and WAR files

The file extension for a class is .java, but when compiled by the Java Development Kit into the Java program, the file becomes .class. The .class file is binary code, which means only computers (in this case, the Java Virtual Machine, or JVM) can read it.

Developers often package up java files into a JAR file, which is like a zip file for Java projects. When you distribute your Java files, you'll likely provide a JAR file that the developer audience will add to their Java projects.

Developers will add their JAR to their class path to make the classes available to their project. To do this, you right-click your project and select **Properties**. In the dialog box, select **Java Build Path**, then click the **Libraries** tab. Then click **Add JARs** and browse to the JAR.

When you deliver a JAR file, developers can use the classes and methods available in the JAR. However, the JAR will not show them the source code, that is, the raw Java files. For this, users will consult the Javadoc.

If you're distributing a reference implementation that consists of a collection of Java source files (so that developers can see how to integrate your product in Java), you'll probably just send them a zip file containing the project.

A WAR file is a web application archive. A WAR is a compiled application that developers deploy on a server to run an application. Whereas the JAR is integrated into a Java project while the developers are actively building the application, the WAR is the deployed program that you run from your server.

That's probably enough Java to understand the different components of a Javadoc.

Summary

Here's a quick summary of the concepts we talked about:

- **Class:** blueprints for something
- **Object:** an instance of a class
- **Methods:** what the object/class can do
- **Fields:** variables in the object/class
- **Constructor:** a method to create an object for a class
- **Package:** a folder that groups classes

- **Access modifier** (e.g, public): the scope at which a thing can be accessed
- **Interface**: a skeleton class with empty methods (used for standardizing)
- **Enum**: a data type offering predefined constants
- **Subclass**: a class that inherits the fields + methods of another class
- **JAR file**: a zip-like file containing Java classes
- **WAR file**: a compiled Java web application to be deployed on a server

Create a Javadoc

Javadoc is the standard output for Java APIs, and it's really easy to build a Javadoc. The Javadoc is generated through something called a “doclet.” Different doclets can parse the Java annotations in different ways and produce different outputs. But by and large, almost every Java documentation uses Javadoc. It's standard and familiar to Java developers.

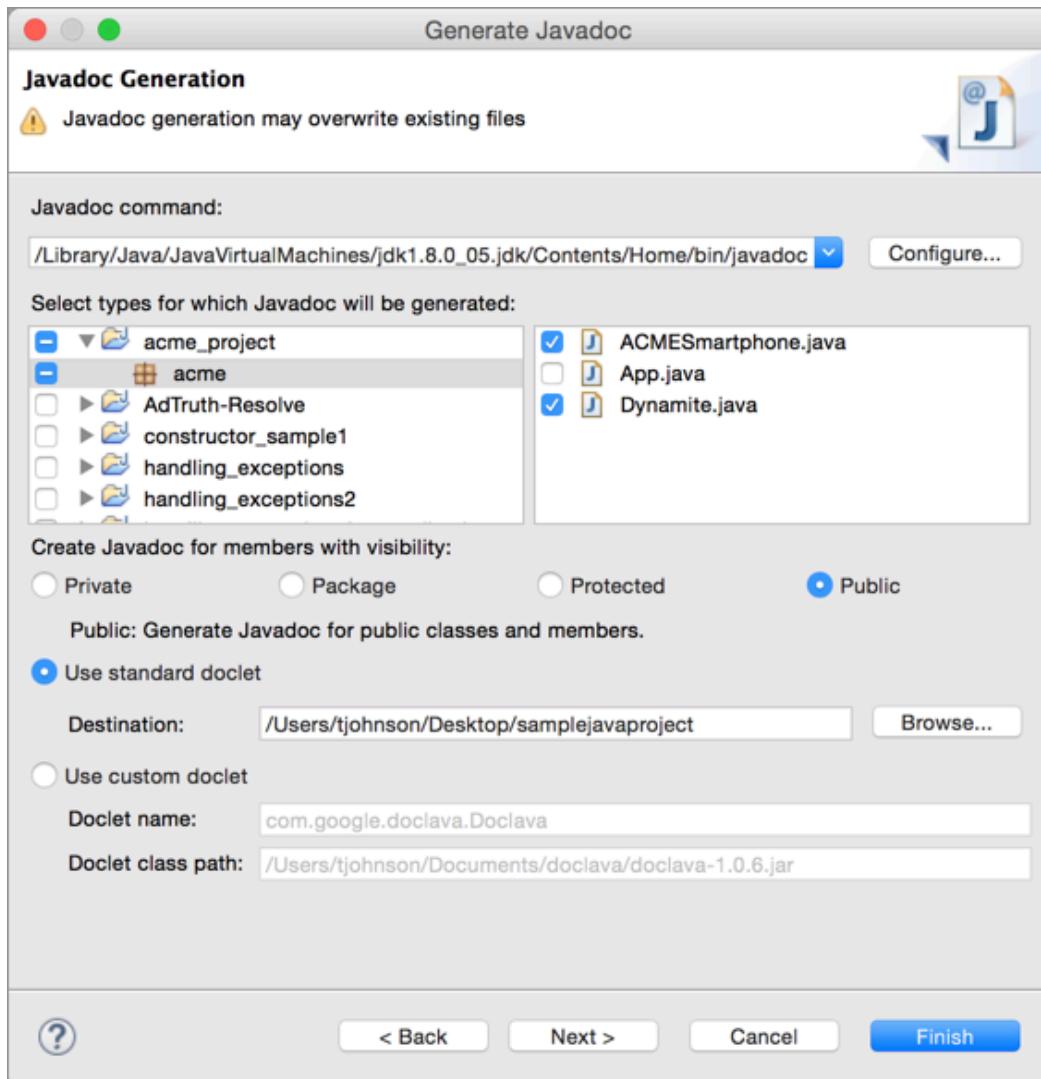
Characteristics of Javadoc

Here are some other characteristics of Javadoc:

- Javadoc is supported by Oracle
- Javadoc's output integrates directly into IDEs that developers use.
- The Javadoc output is skinnable, but you can't add non-ref files to it.
- The Javadoc comment style is highly similar to most other document generators.

Generate a Javadoc

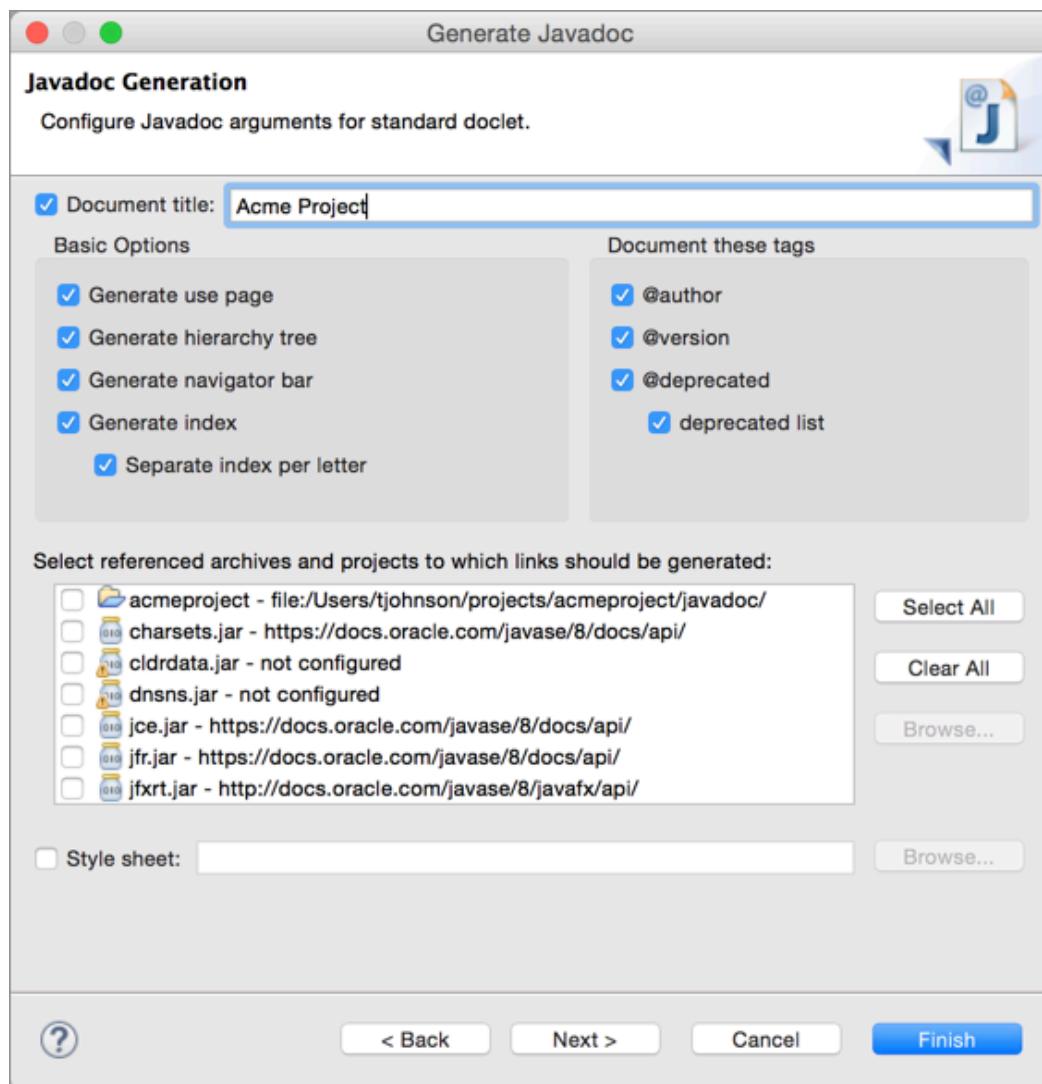
1. Go to **File > Export**.
2. Expand **Java** and select **Javadoc**. Then click **Next**.
3. Select your project and package. Then in the right pane, select the classes you want included in the Javadoc. Don't select the class that contains your main method.



4. Select which visibility option you want: Private, Package, Protected, or Public. Generally you select **Public**.

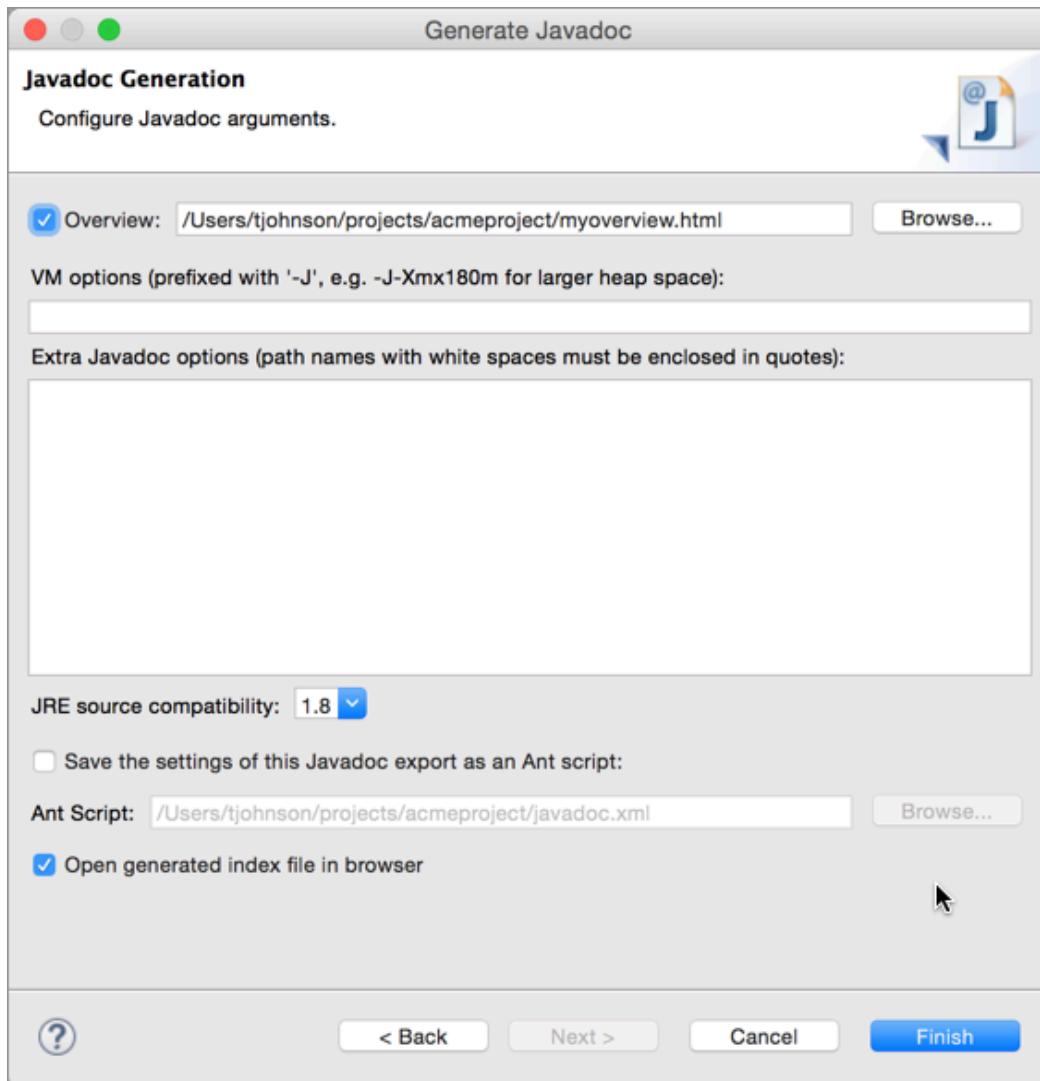
Your API probably has a lot of helper or utility classes used on the backend, but only a select number of classes will actually be used by your developer audience. These classes are made public. It's the public classes that your developer audience will use that form the API aspect of the class library.

5. Make sure the **Use standard doclet** radio button is selected.
6. Click the **Browse** button and select the output location where you want the Javadoc generated.
7. Click **Next**.



Here you can select if you want to omit some tags, such as @author and @deprecated. Generally you don't include the @author tag, since it may only be important internally, not externally. You can also select different options in the Javadoc frame. If you have a custom stylesheet, you can select it here. Most likely you would only make superficial style changes such as with colors.

8. Click **Next**.



Here you can select an HTML page that you want to be your overview page in the Javadoc. You can select any HTML page and it will be included in the index.

9. Click **Finish**.

Javadoc and error checking

Javadoc also checks your tags against the actual code. If you have parameters, exceptions, or returns that don't match up with the parameters, exceptions, or returns in your actual code, then Javadoc will show some warnings.

Try removing a parameter from a method and generate the Javadoc again. Make sure the console window is open.

The screenshot shows the Eclipse IDE interface with the Javadoc generation process in progress. The top menu bar includes 'OVERVIEW', 'PACKAGE', 'CLASS', 'USE', 'TREE', 'DEPRECATED', 'INDEX', and 'HELP'. Below the menu is a toolbar with 'PREV' and 'NEXT' buttons, and 'FRAMES' and 'NO FRAMES' options. The main content area displays the 'Acme Project' overview, stating 'This is my overview page.' and providing a link to 'Description'. A 'Packages' section lists a single package named 'acme'. At the bottom of the screen, the 'Console' view shows the output of the Javadoc generation command:

```
<terminated> Javadoc Generation
Loading source files for package acme...
Constructing Javadoc information...
Standard Doclet version 1.8.0_05
Building tree for all the packages and classes...
Generating /Users/tjohnson/projects/acmeproject/doc/javadoc/acme/ACMESmartphone.html...
/Users/tjohnson/projects/acmeproject/acme/ACMESmartphone.java:41: error: reference not found
    * @exception Throws NullPointerException if model or license is null
      ^
/Users/tjohnson/projects/acmeproject/acme/ACMESmartphone.java:43: warning: no @param for model
    public ACMESmartphone(double model, String license) {
      ^
Generating /Users/tjohnson/projects/acmeproject/doc/javadoc/acme/App.html...
/Users/tjohnson/projects/acmeproject/acme/App.java:12: warning: no description for @param
    * @param args
      ^
/Users/tjohnson/projects/acmeproject/acme/App.java:13: warning: no description for @throws
```

Javadoc tags

Javadoc is a document generator that looks through your Java source files for specific annotations. It parses out the annotations into the Javadoc output. Knowing the annotations is essential, since this is how the Javadoc gets created.

Common Javadoc tags

The following are the most common tags used in Javadoc. Each tag has a word that follows it. For example, `@param latitude` means the parameter is “latitude”.

Tip: To see a lengthy Javadoc tag, see this [example from Oracle](http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#examples) (<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#examples>).

The following are some common Javadoc tags:

- `@author` A person who made significant contribution to the code. Applied only at the class, package, or overview level. Not included in Javadoc output. It's not recommended to include this tag since authorship changes often.
- `@param` A parameter that the method or constructor accepts. Write the description like this: “`@param count` Sets the number of widgets you want included.”
- `@deprecated` Lets users know the class or method is no longer used. This tag will be positioned in a prominent way in the Javadoc. Accompany it with a `@see` or `{@link}` tag as well.
- `@return` What the method returns.
- `@see` Creates a see also list. Use `{@link}` for the content to be linked.
- `{@link}` Used to create links to other classes or methods. Example: `{@link Foo#bar}` links to the method `bar` that belongs to the class `Foo`. To link to the method in the same class, just include `#bar`.
- `@since 2.0` The version since the feature was added.
- `@throws` The kind of exception the method throws. Note that your code must indicate an exception thrown in order for this tag to validate. Otherwise Javadoc will produce an error. `@exception` is an alternative tag.
- `@override` Used with interfaces and abstract classes. Performs a check to see if the method is an override.

Comments versus Javadoc tags

A general comment in Java code is signaled like this:

```
// sample comment...  
  
/*  
sample comment  
*/
```

Javadoc does nothing with these comments.

To include content in Javadoc, you add *two asterisks* at the start, before the class or method:

```
/**  
*  
*  
*  
*  
*/
```

(In Eclipse, if you type `/**` and hit return, it autofills the rest of the syntax automatically.)

The format for adding the various elements is like this:

```
/**  
* [short description]  
* <p>  
* [long description]  
*  
* [author, version, params, returns, throws, see, other tags]  
* [see also]  
*/
```

Here's a real example of Javadoc comments for a method.

```
/**  
 * Zaps the roadrunner with the amount of volts you specify.  
 * <p>  
 * Do not exceed more than 30 volts or the zap function will backfire.  
 * For another way to kill a roadrunner, see the {@link Dynamite#blowDynamite()}{method}.  
 *  
 * @exception IOException if you don't enter an data type amount for the voltage  
 * @param voltage the number of volts you want to send into the roadrunner's body  
 * @see #findRoadRunner  
 * @see Dynamite#blowDynamite  
 */  
public void zapRoadRunner(int voltage) throws IOException {  
    if (voltage < 31) {  
        System.out.println("Zapping roadrunner with " + voltage + " volts!!!!");  
    }  
    else {  
        System.out.println("Backfire!!! zapping coyote with 1,000,000 volts!!!!");  
    }  
}
```

Where the Javadoc tag goes

You put the Javadoc description and tags *before* the class or method (no need for any space between the description and class or method).

What elements you add Javadoc tags to

You add Javadoc tags to classes, methods, and fields.

- For the @author and @version tags, add them only to classes and interfaces.
- The @param tags get added only to methods and constructors.
- The @return tag gets added only to methods.
- The @throws tag can be added to classes or methods.

Public versus private modifiers and Javadoc

Javadoc only includes classes, methods, etc. marked as public. Private elements are not included. If you omit `public`, the default is that the class or method is available to the package only. In this case, it is not included in Javadoc.

The description

There's a short and long description. Here's an example showing how the description part is formatted:

```
/**  
 * Short one line description.  
 * <p>  
 * Longer description. If there were any, it would be  
 * here.  
 * <p>  
 * And even more explanations to follow in consecutive  
 * paragraphs separated by HTML paragraph breaks.  
 *  
 * @param variable Description text text text.  
 * @return Description text text text.  
 */  
public int methodName (...) {  
// method body with a return statement  
}
```

(This example comes from [Wikipedia entry \(http://en.wikipedia.org/wiki/Javadoc\).](http://en.wikipedia.org/wiki/Javadoc))

The short description is the first sentence, and gets shortened as a summary for the class or method in the Javadoc. After a period, the parser moves the rest of the description into a long description. Use `<p>` to signal the start of a new paragraph. You don't need to surround the paragraphs with opening and closing `<p>` tags – the Javadoc compiler automatically adds them.

Also, you can use HTML in your descriptions, such as an unordered list, code tags, bold tags, or others.

After the descriptions, enter a blank line (for readability), and then start the tags. You can't add any more description content below the tags. Note that only methods and classes can have tags, not fields. Fields (variables) just have descriptions.

Note that the first sentence is much like the `shortdesc` element in DITA. This is supposed to be a summary of the entire class or method. If one of your words has a period in it (like `Dr. Jones`), then you must remove the space following the period by adding `Dr. Jones` to connect it.

Avoid using links in that first sentence. After the period, the next sentence shifts to the long paragraph, so you really have to load up that first sentence to be descriptive.

The verb tense should be present tense, such as *gets, puts, displays, calculates...*

What if the method is so obvious (for example, `printPage`) that your description (“prints a page”) becomes obvious and looks stupid? Oracle says in these cases, you can omit saying “prints a page” and instead try to offer some other insight:

Add description beyond the API name. The best API names are “self-documenting”, meaning they tell you basically what the API does. If the doc comment merely repeats the API name in sentence form, it is not providing more information. For example, if method description uses only the words that appear in the method name, then it is adding nothing at all to what you could infer. The ideal comment goes beyond those words and should always reward you with some bit of information that was not immediately obvious from the API name. – [How to write javadoc comments](#) (<http://www.oracle.com/technetwork/articles/java/index-137868.html>)

Avoid @author

Commenting on Javadoc best practices, one person says to avoid `@author` because it easily slips out of date and the source control provides better indication of the last author. ([Javadoc coding standards](#) (<http://blog.joda.org/2012/11/javadoc-coding-standards.html>))

Order of tags

Oracle says the order of the tags should be as follows:

```
@author (classes and interfaces)
@version (classes and interfaces)
@param (methods and constructors)
@return (methods)
@throws (@exception is an older synonym)
@see
@since
@serial
@deprecated
```

@param tags

@param tags only apply to methods and constructors, both of which take parameters.

After the @param tag, add the parameter name, and then a description of the parameter, in lowercase, with no period, like this:

```
@param url the web address of the site
```

The parameter description is a phrase, not a full sentence.

The order of multiple @param tags should mirror their order in the method or constructor.

Stephen Colebourne recommends (<http://blog.joda.org/2012/11/javadoc-coding-standards.html>) adding an extra space after the parameter name to increase readability (and I agree).

As far as including the data type in the parameter description, Oracle says:

By convention, the first noun in the description is the data type of the parameter. (Articles like “a”, “an”, and “the” can precede the noun.) An exception is made for the primitive int, where the data type is usually omitted. – [How to write doc comments using Javadoc](http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag) (<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag>)

The example they give is as follows:

```
@param ch the character to be tested
```

However, the data type is visible from the parameters in the method. So even if you don't include the data types, it will be easy for users to see what they are.

Note that you can have multiple spaces after the parameter name so that your parameter definitions all line up.

@param tags must be provided for every parameter in a method or constructor. Failure to do so will create an error and warning when you render Javadoc.

Note that usually classes don't have parameters. There is one exception: Generics. Generic classes are classes that work with different type of objects. The object is specified as a parameter in the class in diamond brackets: `<>`. Although the Javadoc guidance from Oracle doesn't mention them, you can add a @param tag for a generic class to note the parameters for the generic class. See this [StackOverflow post](http://stackoverflow.com/questions/2015972/is-there-a-javadoc-tag-for-documentating-generic-type-parameters) (<http://stackoverflow.com/questions/2015972/is-there-a-javadoc-tag-for-documentating-generic-type-parameters>) for details. Here's an example from that page:

```
/**  
 * @param <T> This describes my type parameter  
 */  
class MyClass<T>{  
  
}
```

@return tag

Only methods return values, so only methods would receive a @return tag. If a method has `void` as a modifier, then it doesn't return anything. If it doesn't say `void`, then you must include a @return tag to avoid an error when you compile Javadoc.

@throws tag

You add @throws tags to methods or classes only if the method or class throws a particular kind of error.

Here's an example:

```
@throws IOException if your input format is invalid
```

Stephen Colebourne recommends starting the description of the throws tag with an "if" clause for readability.

The @throws feature should normally be followed by "if" and the rest of the phrase describing the condition. For example, "@throws if the file could not be found". This aids readability in source code and when generated.

If you have multiple throws tag, arrange them alphabetically.

Doc comments for constructors

It's a best practice to include a constructor in a class. However, if the constructor is omitted, Javadoc automatically creates a constructor in the Javadoc but omits any description of the constructor.

Constructors have @param tags but not @return tags. Everything else is similar to methods.

Doc comments for fields

Fields have descriptions only. You would only add doc comments to a field if the field were something a user would use.

Cases where you don't need to add doc comments

Oracle says there are 3 scenarios where the doc comments get inherited, so you don't need to type them:

When a method in a class overrides a method in a superclass
When a method in an interface overrides a method in a superinterface
When a method in a class implements a method in an interface – [How to write Javadoc comments](http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag)
(<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag>)

@see tags

The @see tag provides a see also reference. There are various ways to denote what you're linking to in order to create the link. If you're linking to a field, constructor, or method within the same field, use #.

If you're linking to another class, put that class name first followed by the # and the constructor, method, or field name.

If you're linking to a class in another package, put the package name first, then the class, and so on. See this sample from Oracle:

```
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id)
@see Class#method(Type, Type,...)
@see Class#method(Type id, Type id,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id)
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type id, Type, id)
```

- [How to write Javadoc comments \(<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag>\)](http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag)

Links

You can create links to other classes and methods using the `{@link}` tag.

Here's an example from [Javadoc coding standards \(<http://blog.joda.org/2012/11/javadoc-coding-standards.html>\)](http://blog.joda.org/2012/11/javadoc-coding-standards.html) on making links:

```
/***
 * First paragraph.
 * <p>
 * Link to a class named 'Foo': {@link Foo}.
 * Link to a method 'bar' on a class named 'Foo': {@link Foo#bar}.
 * Link to a method 'baz' on this class: {@link #baz}.
 * Link specifying text of the hyperlink after a space: {@link Foo the
 * Foo class}.
 * Link to a method handling method overload {@link Foo#bar(String,in
 * t)}.
 */
public ...
```

To link to another method within the same class, use this format: `{@link #baz}`. To link to a method in another class, use this format: `{@link Foo#baz}`. However, don't over hyperlink. When referring to other classes, you can use `<code>` tags.

To change the linked text, put a word after `#baz` like this: `@see #baz Baz method`.

Previewing Javadoc comments

In Eclipse, you can use the Javadoc tab at the bottom of the screen to preview the Javadoc information included for the class you're viewing.

The screenshot shows the Eclipse IDE interface with the Javadoc tab selected. At the top, there is a code editor window displaying Java code. Below it is a toolbar with tabs: Problems, Javadoc (which is active), Declaration, Search, Console, Results, and Markdown Help. The main content area displays the Javadoc documentation for the `ACME_Smartphone.zapRoadRunner` method. The documentation includes a brief description, parameters, throws information, and see also links.

```
60     if (voltage < 31) {
61         System.out.println("Zapping roadrunner with " + voltage + " "
62     } else {
63         System.out.println("Backfire!!! zapping coyote with 1,000,000 "
64     }
65 }
66
67
68 }
```

Problems Javadoc Declaration Search Console Results Markdown Help

void javadoc_tags.ACME_Smartphone.zapRoadRunner(int voltage) throws IOException

Zaps the roadrunner with the amount of volts you specify.
Do not exceed more than 30 volts or the zap function will backfire. For another way to kill a roadrunner, see [Dynamite.blowDynamite](#).

Parameters:
voltage the number of volts you want to send into the roadrunner's body

Throws:
[IOException](#) – if you don't enter an data type amount for the voltage

See Also:
[findRoadRunner](#)
[Dynamite.blowDynamite](#)

More information

- Oracle's explanation of Javadoc tags (<http://www.oracle.com/technetwork/articles/javase/index-137868.html>)
- Javadoc (<http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html>)

Exploring the Javadoc output

The Javadoc output hasn't changed much in the past 20 years, so in some sense it's predictable and familiar. On the other hand, the output is dated and lacks some critical features, like search, or the ability to add more pages. Anyway, it is what it is.

Class summary

The class summary page shows a short version of each of the classes. The description you write for each class (up to the period) appears here. It's kind of like a quick reference guide for the API.

The screenshot shows a Javadoc interface. On the left, a sidebar lists "All Classes" with entries for "ACMEsmartphone" and "Dynamite". The main content area has a header bar with links: PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. Below this is another set of links: PREV PACKAGE, NEXT PACKAGE, FRAMES, and NO FRAMES. The main content displays a table titled "Package javadoc_tags". The table has two columns: "Class" and "Description". The "Class" column contains "ACMEsmartphone" and "Dynamite". The "Description" column contains the descriptions: "Works like a regular smartphone but also tracks roadrunners." for ACMEsmartphone and "Provides ways to explode dynamite to blow up roadrunners." for Dynamite. At the bottom of the content area is a footer bar with the same set of links as the header: PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP, followed by PREV PACKAGE, NEXT PACKAGE, FRAMES, and NO FRAMES.

You click a class name to dive into the details.

Class details

When you view a class page, you're presented with a brief summary of the fields, constructors, and methods for the class. Again this is just an overview. When you scroll down, you can see the full details about each of them.

All Classes

ACMESmartphone
Dynamite

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

javadoc_tags

Class ACMESmartphone

java.lang.Object
javadoc_tags.ACMESmartphone

public class **ACMESmartphone**
extends java.lang.Object

Works like a regular smartphone but also tracks roadrunners.

The ACME Smartphone can perform similar functions as other smartphones, such as making phone calls, sending text messages, and browsing the web. However, the ACME Smartphone also enables GPS tracking on roadrunners. You can monitor the location of all roadrunners within a 20 mile radius using the RoadRunner Tracker app.

Note that the RoadRunner Tracker app requires you to be connected to wifi. It will not work on cellular data.

Since:
1.3

Version:
2.0

Author:
Tom Johnson

See Also:

<http://docs.oracle.com/javase/7/docs/api/>

Other navigation

If you click **Package** at the top, you can also browse the classes by package. Or you can go to the classes by clicking the class name in the left column. You can also browse everything by clicking the **Index** link.

All Classes

ACMESmartphone
Dynamite

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

javadoc_tags

Class Dynamite

java.lang.Object
javadoc_tags.Dynamite

public class **Dynamite**
extends java.lang.Object

Provides ways to explode dynamite to blow up roadrunners.

Try using the `ACMESmartphone.zapRoadRunner(int)` before resorting to `blowDynamiteLoudBang(int)`. However, if you do have a lot of roadrunners, the `zapRoadRunner` method may not be efficient enough. You'll probably just want to resort to `blowDynamiteLoudBang()` to annihilate them all at once.

Constructor Summary

Constructors
Constructor and Description
Dynamite()

file:///Users/tjohnson/Desktop/samplejavaproject/javadoc_tags/Dynamite.h

For more information about how the Javadoc is organized, click the **Help** button.

Making edits to Javadoc tags

It's pretty common for developers to add Javadoc tags and brief comments as they're creating Java code. In fact, if they don't add it, the IDE will usually produce a warning error.

However, the comments that developers add are usually poor, incomplete, or incomprehensible. A tech writer's job with Javadoc is often to edit the content that's already there, providing more clarity, structure, inserting the right tags, and more.

When you make edits to Javadoc content, look for the following:

- **Missing doc.** Lots of Javadoc is incomplete. Look for missing documentation.
- **Consistent style.** See if the existing tags follow Java's style conventions.
- **Clarity.** Some descriptions are unintelligible due to the curse of knowledge, but it's hard to judge without a stronger grasp of Java.

In this exercise, you'll make some edits to the Javadoc tags and see how they get rendered in the output.

Make some edits

Make some edits to a class and method. Then regenerate the Javadoc and find your changes.

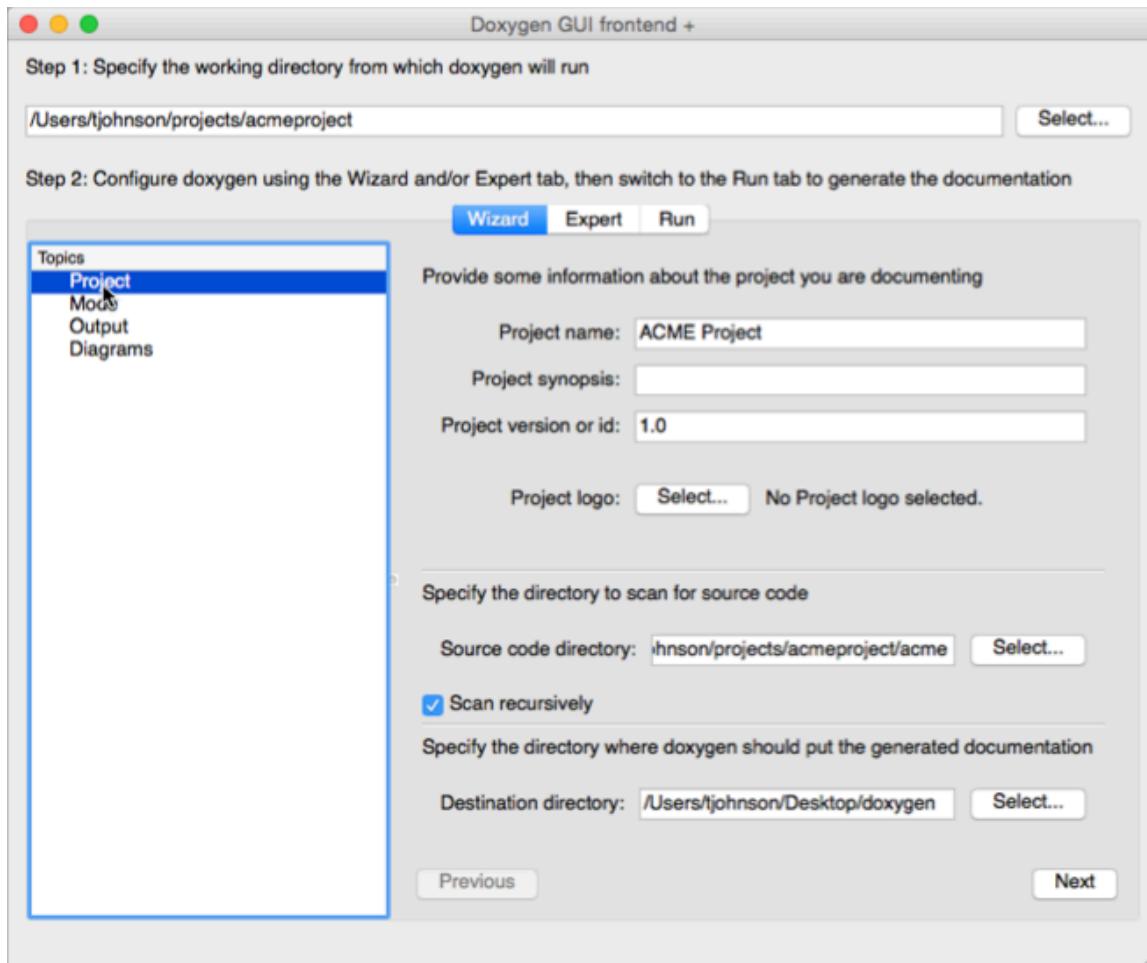
Doxygen, another document generator

An alternative to Javadoc is Doxygen. Doxygen works highly similarly to Javadoc, except that you can process more languages (Java, C++, C#, and more) with it. Doxygen is most commonly used with C++. Additionally, there's a GUI tool (called Doxywizard) that makes it really easy to generate the file.

Download Doxywizard

You can download the Doxywizard tool when you install Doxygen. See the [Doxygen](http://www.stack.nl/~dimitri/doxygen/download.html) (<http://www.stack.nl/~dimitri/doxygen/download.html>) download page for more information.

Here's Doxygen's front-end GUI generator (Doxywizard):



Here's the Doxygen output:

By the way, you don't need to use the wizard. You can also just generate Doxygen through a configuration file. This is how developers typically run Doxygen builds from a server.

In contrast to Javadoc, Doxygen also allows you to incorporate external files written in Markdown. And Doxygen provides a search feature. These are two features that Javadoc lacks.

Doxygen is maintained by a single developer and, like Javadoc, hasn't changed much over the years. In my opinion, the interface is highly dated and kind of confusing.

Integrating builds automatically

In a lot of developer shops, document generators are integrated into the software build process automatically. Doxygen allows you to create a configuration file that can be run from the command line (rather than using the frontend GUI). This means when developers build the software, the reference documentation is automatically built and included in the output.

Other document generators

You don't need to limit yourself to either Javadoc or Doxygen. There are dozens of different document generators for a variety of languages. Just search for “document generator + {programming language}” and you'll find plenty. However, don't get very excited about this genre of tools. Document generators are somewhat old, produce static front-ends that look dated, are often written by engineers for other engineers, and not very flexible.

Perhaps the biggest frustration of document generators is that you can't really integrate the rest of your documentation with them. You're mostly stuck with the reference doc output. You'll need to also generate your how-to guides and other tutorials, and then link to the reference doc output. As such, you won't end up with a single integrated experience of documentation. Additionally, it will be hard to create links between the two outputs.

Creating non-ref docs with native library APIs

Although much attention tends to be given to the reference documentation with APIs, actually the bulk of what technical writers usually do with native library API docs is provide non-reference documentation. This is the stuff that engineers rarely write.

Engineers will throw a quick description of a class in a file and generate a Javadoc, and they'll give that Javadoc to the user as if it represents a complete set of documentation, but reference docs don't tell even half the story.

Reference docs can be an illusion for real doc

Jacob Kaplan Moss says (<http://jacobian.org/writing/what-to-write/>) that reference docs can be an illusion:

... auto-generated documentation is worse than useless: it lets maintainers fool themselves into thinking they have documentation, thus putting off actually writing good reference by hand. If you don't have documentation just admit to it. Maybe a volunteer will offer to write some! But don't lie and give me that auto-documentation crap. – Jacob Kaplan Moss

Other people seem to have [similar opinions](https://communities.cisco.com/community/developer/blog/2014/09/03/introducing-devnet-slate) (<https://communities.cisco.com/community/developer/blog/2014/09/03/introducing-devnet-slate>):

Auto-generated documentation that documents each API end-point directly from source code have their place (e.g., its great for team that built the API and its great for a reference document) but hand-crafted quality documentation that walks you through a use case for the API is invaluable. It should tell you about the key end-points that are needed for solving a particular problem and it should provide you with code samples.”

In general, document generators don't tell you a whole lot more than you would discover by browsing the source code itself. Some people even refer to auto-generated docs as a glorified source-code browser.

Reference docs are feature-based, not task-based

One of the main problems with reference documentation is that it's feature based rather than task based. It's the equivalent of going tab-by-tab through an interface and describing what's on each tab, what's in each menu, and so on. We know that's a really poor way to approach documentation, since users organize their mental model by the tasks they want to perform.

When you write API documentation, consider the tasks that users will want to do, and then organize your information that way. Reference the endpoints as you explain how to accomplish the tasks. Users will refer to the reference docs as they look for the right parameters, data types, and other class details. But the reference docs won't guide them through tasks alone.

Course summary

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.

Tooling shouldn't be your main focus

Once you get the tooling infrastructure in place, it should mostly take a back seat to the daily tasks of content development.



(<https://flic.kr/p/QMVMw>)

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."

Summary

During this course, we explored the following:

- Using a REST API as a developer
- Documenting a new API endpoint
- Documenting non-reference sections
- Exploring other REST APIs
- Publishing API docs
- Design patterns

You learned new tools and technologies, ways to publish and create interactive experiences for your audience. Now as you go forward into API documentation, hopefully you have a solid foundation to start.

The remaining sections contain miscellaneous topics related to API documentation.

Getting a Job in API Documentation

In this section:

- [Exploring more REST APIs \(page 188\)](#)
- [EventBrite example: Get Event information and display it on a page \(page 189\)](#)
- [Flickr example: Retrieve a Flickr gallery and display it on a web page \(page 196\)](#)
- [Klout example: Retrieve Klout influencers and influencees \(page 205\)](#)
- [Aeris Weather Example: Get wind speed and use as conditional value \(page 216\)](#)
- [Next phase of course \(page 223\)](#)

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](https://www.algolia.com/doc/rest) (<https://www.algolia.com/doc/rest>). 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. On the left, there's a sidebar with links to Documentation, How It Works, API Documentation, FAQ, and REST API. The main content area has a title 'Documentation' and a sub-section 'Get started with Basics Tutorial'. Below this, there's a heading 'API Clients & Integrations' followed by a grid of icons representing various programming languages and frameworks: Ruby, Rails, Python, PHP, Symfony, Node.js, JavaScript, Go, Java, curl, Objective-C, Swift, Android, C#, WordPress, Magento, Laravel, WooCommerce, Parse, Parse.com, and Jekyll. At the bottom right of the main content area is a button labeled 'Need help?'. The URL <https://www.algolia.com/doc> is visible at the bottom of the page.

(<https://www.algolia.com/doc>)

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 provided 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](http://www.safaribooksonline.com/) (<http://www.safaribooksonline.com/>)). 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 to create developer documentation?

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](http://hackwrite.com/posts/enough-to-be-dangerous/) (<http://hackwrite.com/posts/enough-to-be-dangerous/>), 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](http://hackwrite.com/posts/enough-to-be-dangerous/) (<http://hackwrite.com/posts/enough-to-be-dangerous/>))

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](http://idratherbewriting.com/2016/12/20/changing-roles-of-technical-writers/) (<http://idratherbewriting.com/2016/12/20/changing-roles-of-technical-writers/>), 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](https://withintheordinary.wordpress.com/2016/12/21/adding-value-as-a-technical-writer/) (<https://withintheordinary.wordpress.com/2016/12/21/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](http://drjamesneiman.com/) (<http://drjamesneiman.com/>), 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](http://www.synergistech.com/) (<http://www.synergistech.com/>) recently gave a presentation titled [Finding the right API Technical Writer](https://www.youtube.com/embed/lmNHBg20ql0) (<https://www.youtube.com/embed/lmNHBg20ql0>) at a API conference in London last October.

See [this video recording on YouTube](https://www.youtube.com/embed/lmNHBg20ql0?start=22m33s&end=24m17s) (<https://www.youtube.com/embed/lmNHBg20ql0?start=22m33s&end=24m17s>) (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

In this section:

Copyright

Copyright © 2017 by Tom Johnson

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.