Documenting REST APIs

Tom Johnson

Last generated: August 31, 2015

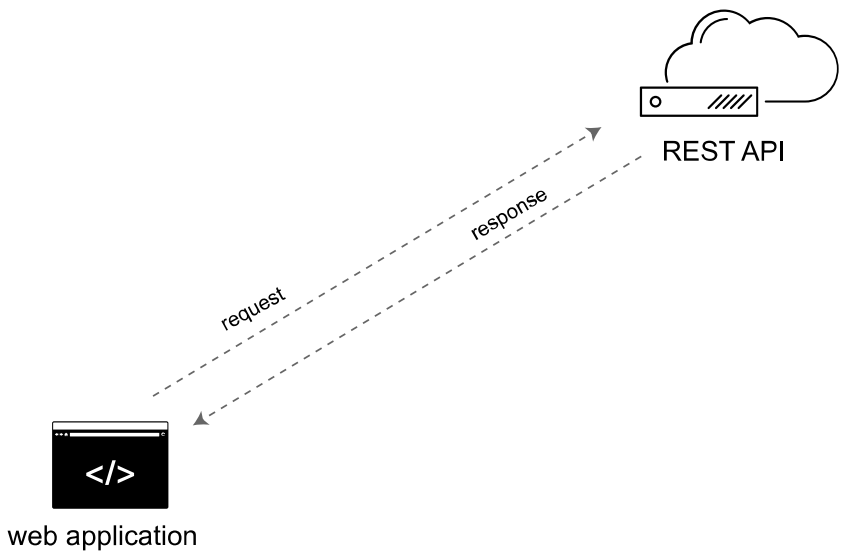


Table of Contents

Exploring other REST APIs

Documenting REST APIs

Focus of the course is REST APIs

This course provides a tutorial on documenting REST APIs (not platform APIs such as Java or C++). This course is intended to be sequential, walking you through a series of concepts and activities that build on each other.

Because the purpose of the course is to help you learn, there are numerous activities that require hands-on coding, exploring, and other exercises. Between the learning activities, there are more conceptual deep dives, but the focus is always on learning by doing.

Time to completion

This course is expected to take about 3 hours.

Learn with a real example and context

Rather than present abstract concepts, this course contextualizes REST APIs with a more direct, hands-on approach. You learn about API documentation in the context of using a simple weather API to put a weather forecast on your site.

As you use the API, you learn about cURL, JSON, endpoints, parameters, data types, authentication, and other details associated with REST APIs. The point is that, rather than learning about these concepts independent of any context, you learn them by immersing yourself in a real context.

After you use the API as a developer might, you then shift perspectives and become a technical writer tasked with documenting a new endpoint that has been added to the API.

> ℹ **Note:** This courses focuses entirely on creating REST API documentation (also referred to as web APIs). Other courses will focus on publishing, and others on platform APIs.

No programming skills required

You don't need any programming background or other prerequisites, but if you do have some familiarity with programming concepts, you can speed through the sections and jump ahead to the topics you want to learn more about. This course assumes you're a beginner, though. If something is obvious, feel free to skip to where you feel you're actually learning something.

What you'll need

- **Text editor**. (Sublime Text (http://www.sublimetext.com/) is a good option (both Mac and PC). On Windows, Notepad++ (https://notepad-plus-plus.org/) and 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 app) (https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm?hl=en)**

- **cURL (http://curl.haxx.se/)**. cURL is essential for making requests to endpoints from the command line. See the following section for more details on cURL.

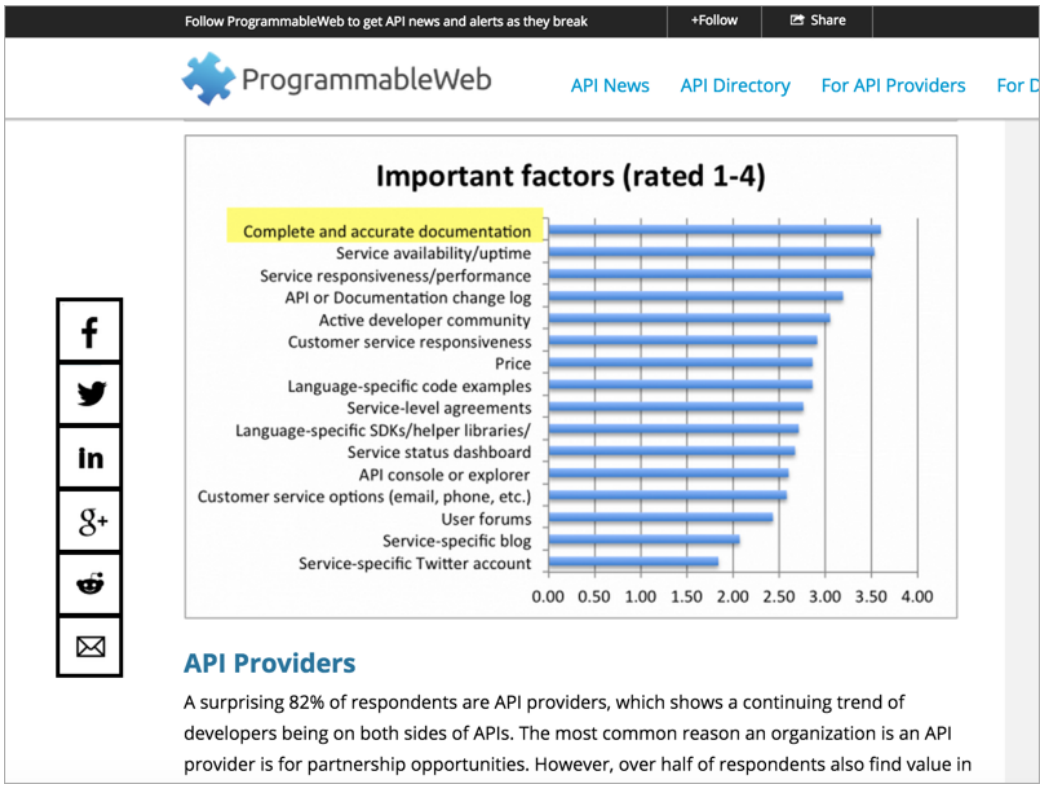
- **Network connection**. Your computer needs to be able to connect to a wifi network.

The market for REST API documentation

Complete and accurate docs are most important factor in APIs

Before we get into the nuts and bolts of documenting REST APIs, let me provide some context about the popularity of 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), about 250 developers were asked to rank the most important factors in an API. "Complete and accurate documentation" ranked as #1.



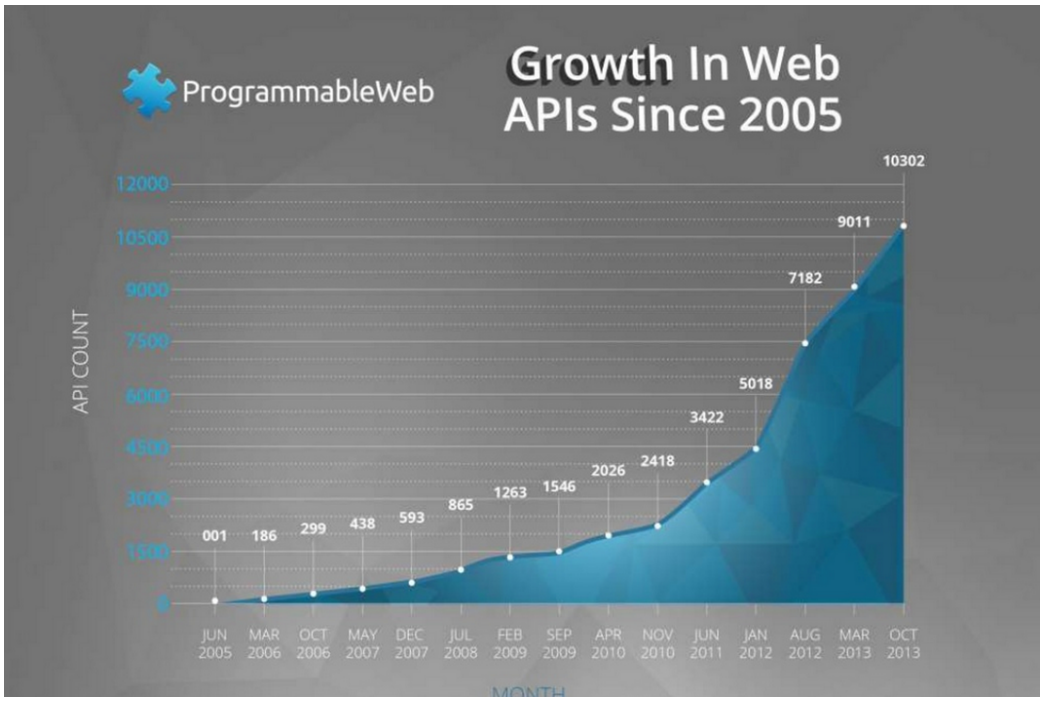
(http://www.programmableweb.com/news/api-consumers-want-reliability-documentation-and-community/2013/01/07)

John Musser, one of the founders of Programmableweb.com, has followed up on the importance of good API documentation in some of 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."

(http://www.slideshare.net/jmusser/ten-reasons-developershateyourapi)

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 prevalance of web APIs.



(http://www.slideshare.net/programmableweb/web-api-growthsince2005)

eBay's API in 2005 was one of the first web APIs. Since then, the tremendous growth in web APIs — coupled with the importance of documentation with APIs — presents a perfect opportunity for technical writers. Technical writers can apply their communication skills to fill a gap in a market that is exploding.

Why is documentation so important for REST APIs?

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 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. In order to understand the request and response format for the REST API, you don't consult the SOAP message specification or look at the WSDL file. Instead, you have to consult the REST API 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 with REST APIs. As long as there is variety, there will be a strong need for technical writers.

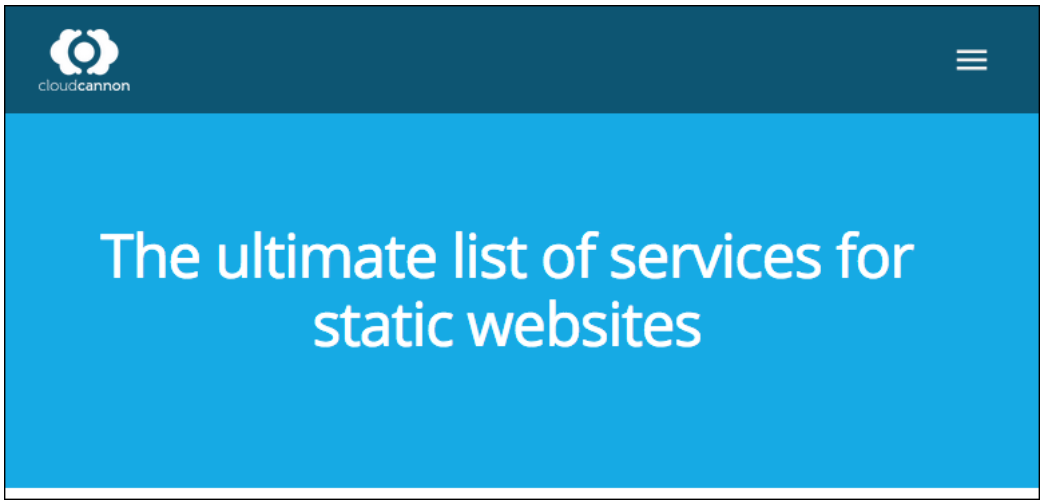
The web is a mashup of APIs

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 Switftype instead and leverage their service through their Swifttype API (https://swiftype.com/developers).

Rather than building your own payment gateway, you integrate Stripe and its 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/). And so on. Rather than building your own e-commerce system, you might use Snipcart and its API (http://docs.snipcart.com/api-reference/introduction).

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. For example, there's no newsletter integration, analytics, search, commenting systems, forms, chat ecommerce, surveys, or other systems. Instead, you leverage the services you need into your static site.

CloudCannon has a long list of services (http://cloudcannon.com/tips/2014/12/12/the-ultimate-list-of-services-for-static-websites.html) that you can interate into your static site.



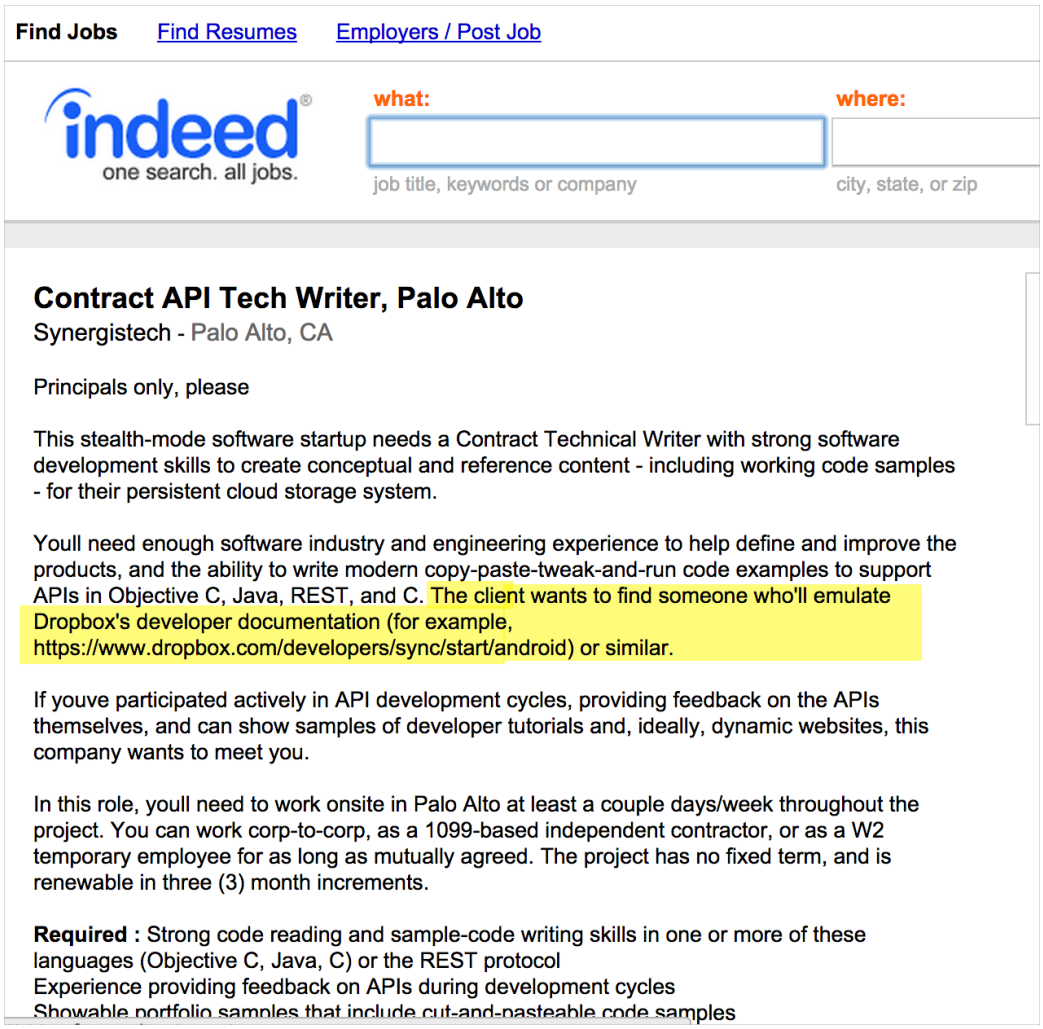
(http://cloudcannon.com/tips/2014/12/12/the-ultimate-list-of-services-for-static-websites.html)

This cafeteria style model is replacing the massive, swiss-army-site model that tries to do anything and everything. It's better to let specialists create a very powerful, robust tool (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. The web is becoming an interwoven mashup of lots of different services from different APIs interacting with each other.

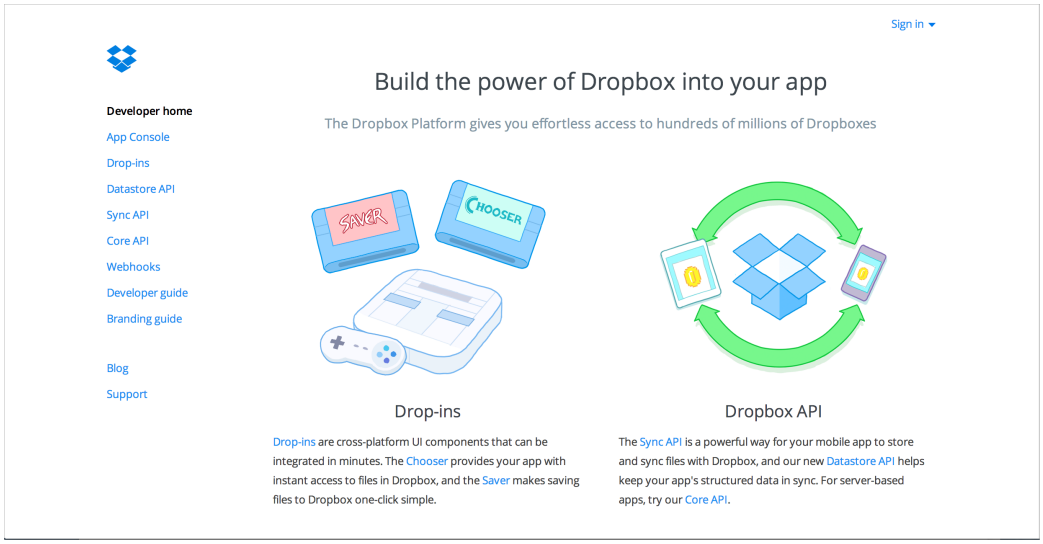
Job market is hot for API technical writers

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



(https://www.dropbox.com/developers/sync/start/android)

It's not a sophisticated design. But its simplicity and brevity is one of its strengths.

API doc is a new world for most tech writers

API documentation is mostly a new world to technical writers. Many of the components may seem foreign:

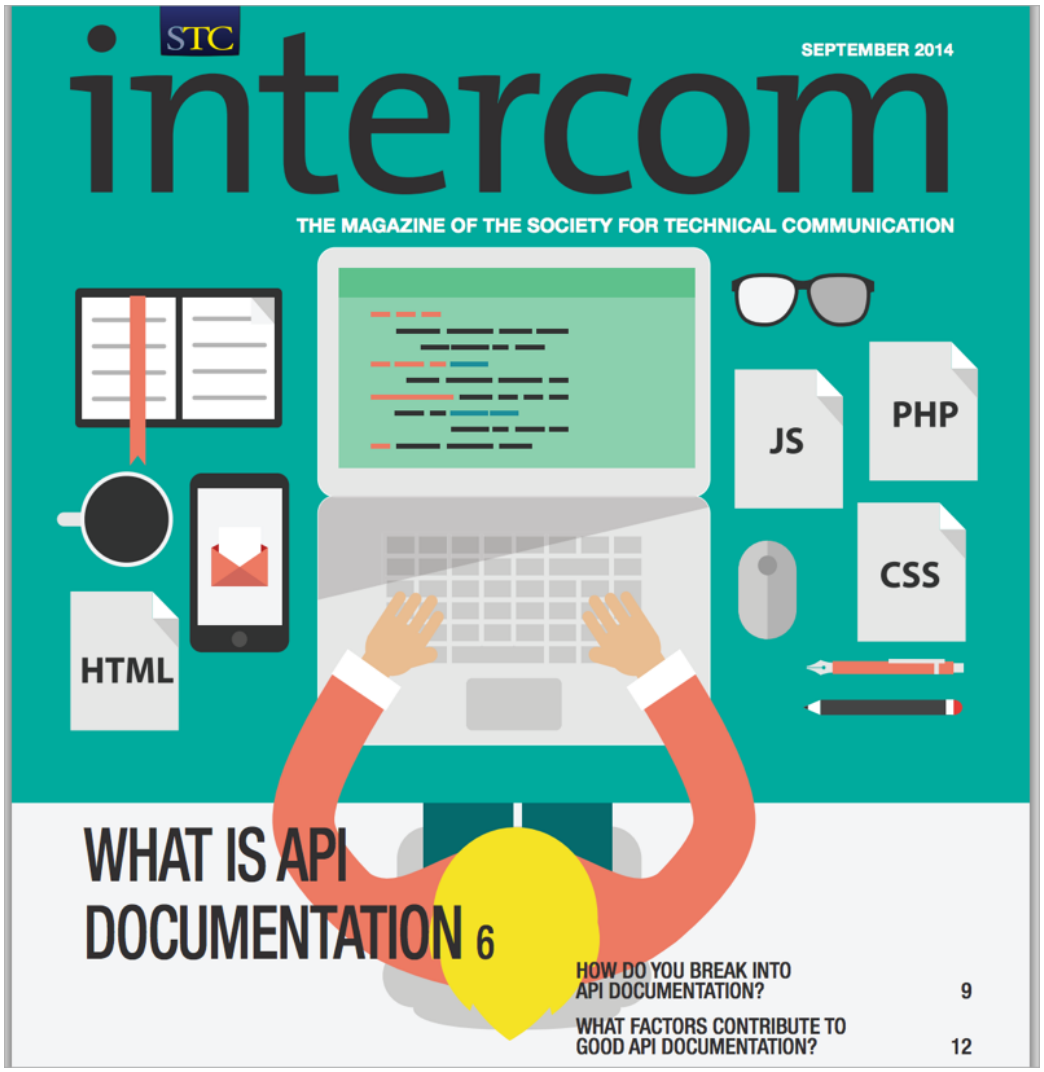
- API doc authoring tools
- Developer audience
- Programming languages
- Endpoint reference topics
- Ways the product is used

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

(http://bit.ly/ZFYIoT)

Increasing learning materials about API documentation

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



(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).

Last year, the STC Summit in Columbus held its first ever API documentation track.

Technical writers are hungry to learn more about APIs. To help address this ongoing need, I've created this course to teach technical writers about API documentation.

What is a REST API?

This course is all about learning by doing, but while *doing* various activities, I'm going to periodically pause and dive into some more abstract concepts to fill in more detail. This is one of those moments.

About web services

In general, 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 (Application Programming Interfaces), including both REST and SOAP APIs. Web services are basically request and response interactions between clients and servers (one computer makes the request, and the API provides the response).

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

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 platform agnostic and therefore interoperable across different languages and platforms.

SOAP APIs: The predecessor to REST

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.

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).

Problems with SOAP and XML

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 more slow.

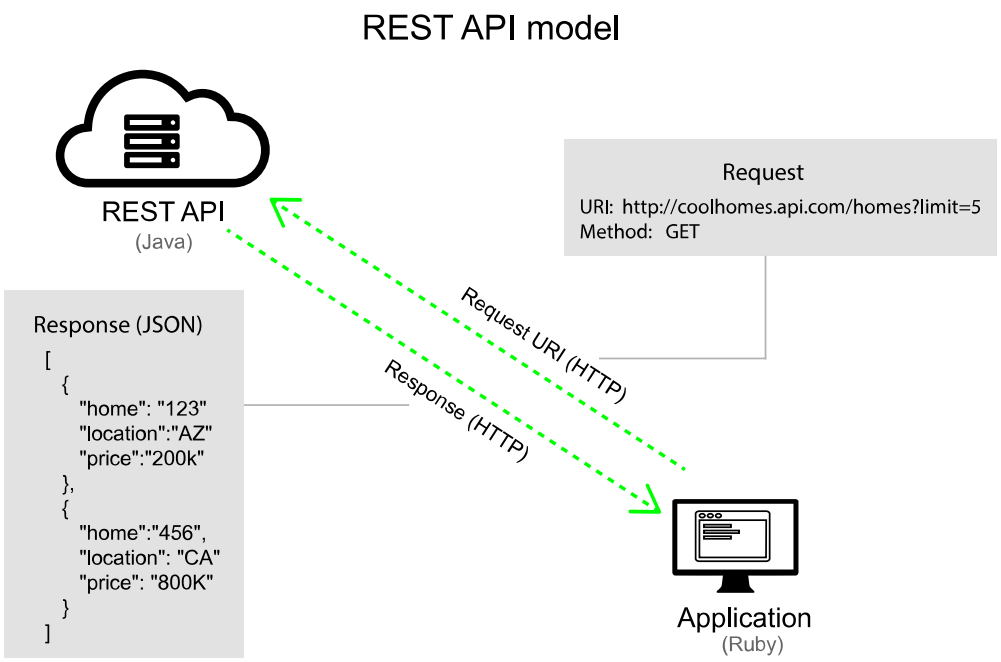
SOAP is still used in enterprise application scenarios with server-to-server communication, but in the past 5 years, SOAP and XML have largely been replaced by REST and JSON, especially for APIs on the open web. You can browse some SOAP APIs at http://xmethods.com/ve2/index.po (http://xmethods.com/ve2/index.po).

REST

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.

> **Note:** Sometimes REST APIs are called _RESTful_ APIs, because REST is an architectural style (not a defined standard) that the API follows.

Here's the general model of a REST API:



Any message format can be used

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.

JSON most common

Despite the variety of message format options, most REST APIs use JSON (JavaScript Object Notation) as the default message format. This 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.

Focus on resources through URIs

REST APIs focus on *resources* (that is, *things*, rather than actions, as SOAP does), 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 also may include 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 (rather than whatever the default might be).

☑ **Tip:** 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 a REST API.

Sample URIs/endpoints

Here's what a sample REST URI might look like:

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

This URI would get the homes resource and limit the result to 5. It would return the response in JSON format.

You can have multiple URIs (also called "endpoints") that refer to the same resource. Here's one variation:

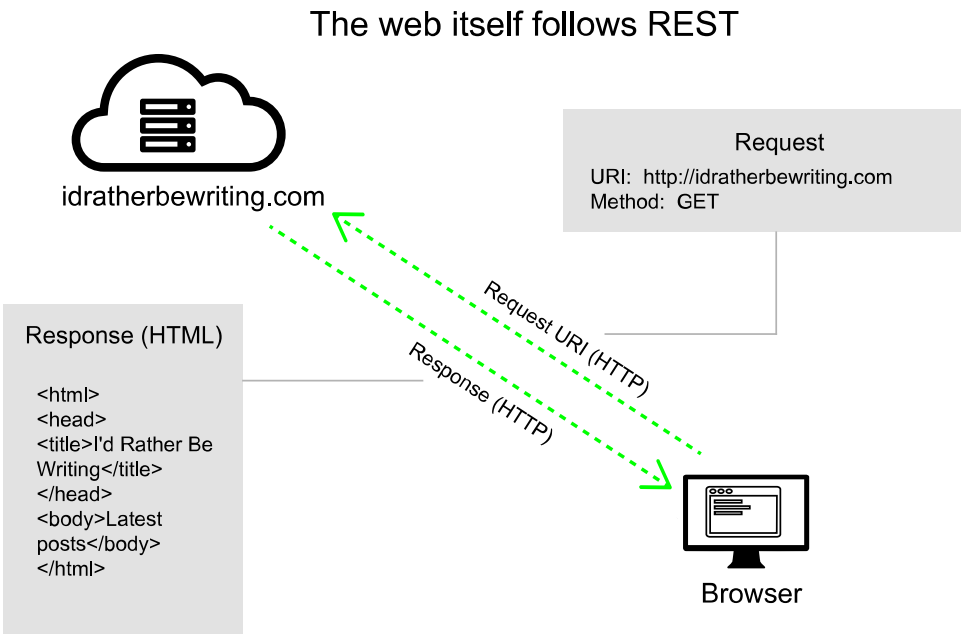
```
http://apiserver.com/homes/1234
```

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 a certain criteria, homes in a specific format, and so on), but here we want to see home 1234.

The web follows REST

The terminology of URIs and GET requests and message responses transported over HTTP protocol might seem unfamiliar, but really this is just the official REST terminology to describe what's happening. If you've used the web, you're already familiar with how REST APIs work, because the web itself more or less follows a RESTful style.

If you open a browser and go to http://idratherbewriting.com (http://idratherbewriting.com), you're really using HTTP protocol (`http://`) to submit a GET request to the resource available at idratherbewriting.com. 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` . The web itself is an example of RESTful style architecture.

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 a URI, 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. And the responses can be cached in order to increase the performance.

No WSDL files, but some specs exist

REST APIs don't use a WSDL file to describe 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. (This provides a great opportunity for technical writers! Hooray!)

Some more formal specifications — for example, Swagger and RAML — have been developed to describe REST APIs. When you describe your API using the Swagger or RAML specification, Swagger or RAML will produce documentation that describes how to interact with the API (listing out the resources, parameters, and other details).

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 Swagger or RAML documentation outputs to include all the details users would need to work with your API (for example, how to pass authorization keys, workflows and interdependencies between endpoints, and so on).

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 the API. As you explore REST APIs, you will find that they differ greatly from one to another (especially their documentation formats!), but they all share the common patterns outlined here.

Other REST formats: OData

Finally, I want to mention one sub-type of REST APIs: OData (Open Data). OData follows the RESTful API style but with a more specific format, especially with regards to the query parameters in the URIs. Many of the OData URIs follow a specific pattern.

For example, with OData APIs, you add query parameters in the URI prefaced by `$`. Common parameters are `format`, `orderby`, `filter`, and `top`.

Here's a sample OData request:

```
http://services.odata.org/OData/OData.svc/Products?$orderby=Price&$format=json
```

Responses can be in JSON or XML.

With most REST APIs, you don't use `$` before the query parameters like you do with OData. Query parameters are usually separated by `?`, and each REST API has its own variety of unique parameters.

OData is championed by Microsoft and used by Azure (the Microsoft cloud services). To learn more about OData, see the http://www.odata.org/ (http://www.odata.org/).

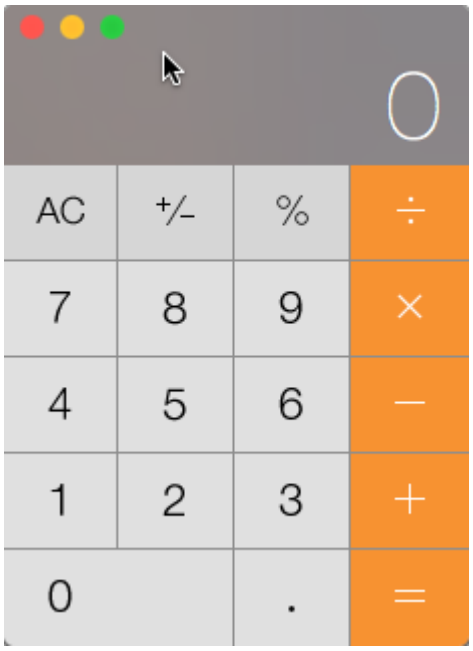
Exploring a REST API marketplace

An API is an interface between systems

An API provides an interface between two systems. It's like a cog that allows two systems to interact with each other.

(http://bit.ly/1DexWMo)

Consider 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.



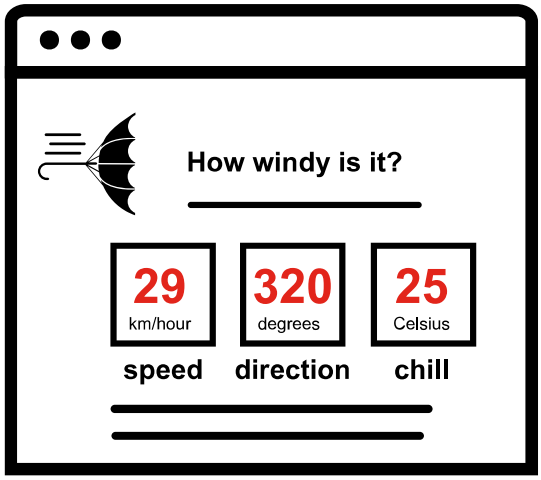
APIs often work in similar ways. But instead of interacting within the same system, those functions call remote services to get their information. REST APIs make the calls using web protocols — similar to how addresses you type in a browser return a web page.

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 both powered by APIs underneath.

Our course scenario: Weather forecast API

In this course, we're going to use an API 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 biking site. You want to allow users who come to your site to see what the weather is like for the week. You want something like this:



You don't have your own meteorological service, so you're going to 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?" Click the button to see wind details.

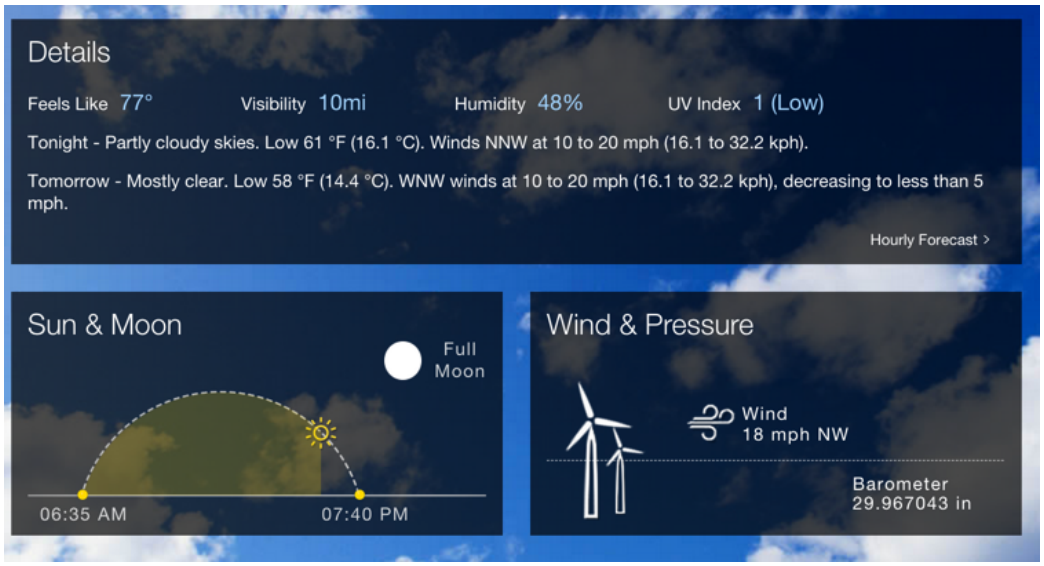
Check wind conditions

Wind conditions for Santa Clara

Wind chill:
Wind speed:
Wind direction:

When you request this data, an API is going out to a weather service, retrieving the information, and displaying it to you.

Of course, the above example is extremely simple. You could also build an attractive interface like this:

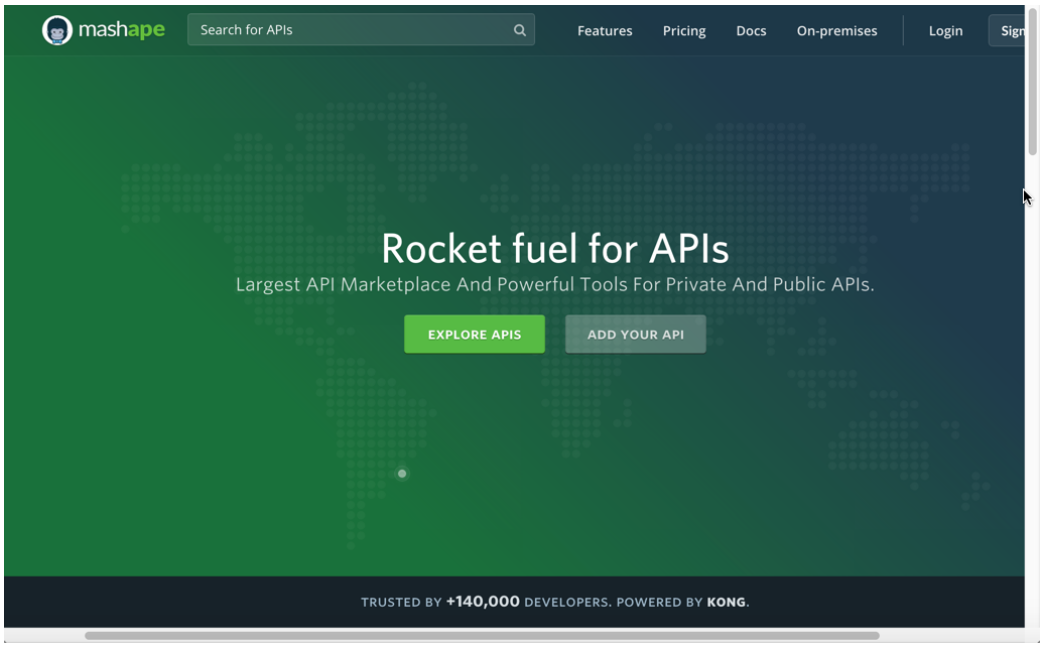


(https://weather.yahoo.com/united-states/california/santa-clara-2488836/)

Browse the available APIs on Mashape

Activity

Mashape 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.



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

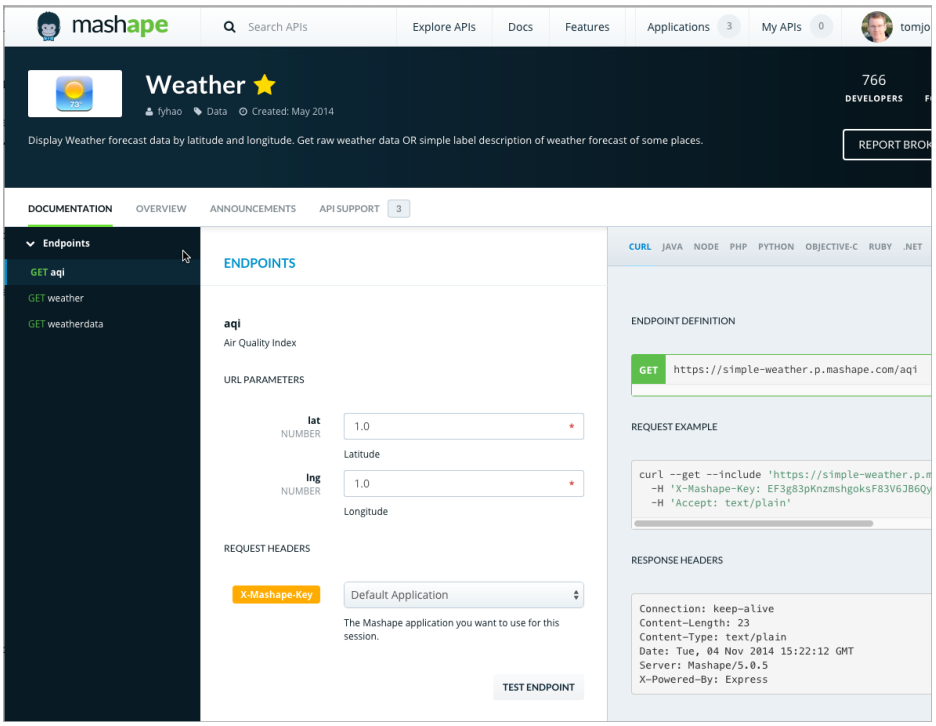
Explore the APIs available on Mashape:

1. Go to mashape.com (http://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 developers build.

3. Look for an API called "Weather," by fyhao. Although there are many weather APIs, this one seems to have a lot of reviews and is free.



(https://www.mashape.com/fyhao/weather-13)

Spend a little time exploring and getting to know the features and information this weather API provides. Click **Test Endpoint** and see what kind of information comes back.

Answer these basic questions about this weather forecast API:

- What does the API do?
- How many endpoints does the API have?
- What does each endpoint do?
- 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.

☑ **Tip:** Sometimes people use 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

Authorization for making API calls

Almost every API has a method in place to authenticate requests. You usually have to provide your API key in requests in order to get a response. Authorization allows API publishers to do the following:

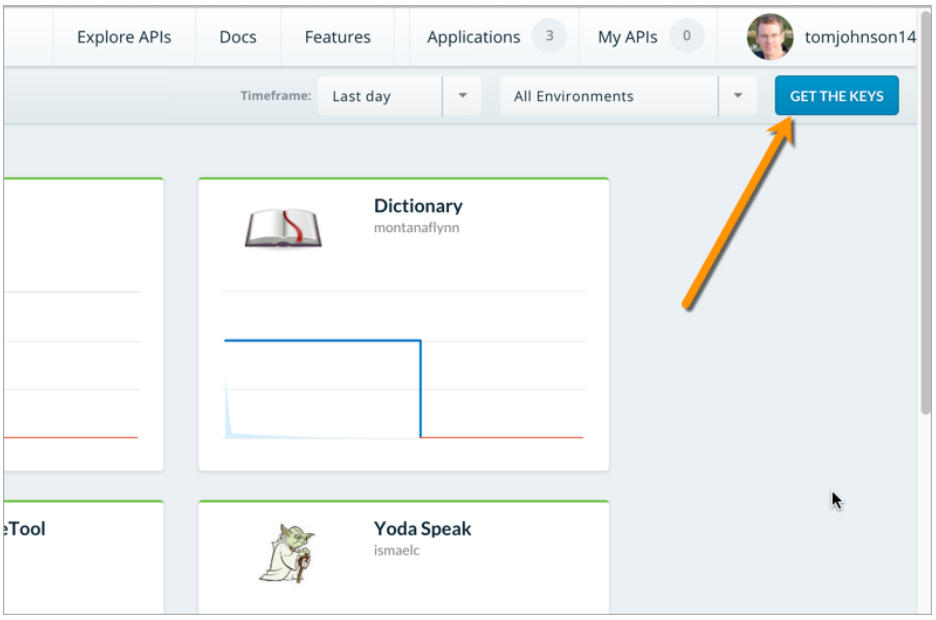
- License access to the API
- Rate limit the number of requests
- Control availability of certain features within the API

☑ **Tip:** Keep in mind how users authorize calls into an API — this is something you usually cover in API documentation. Later in the course we will dive into authorization methods in more detail (page 0).

Get the Mashape authorization keys

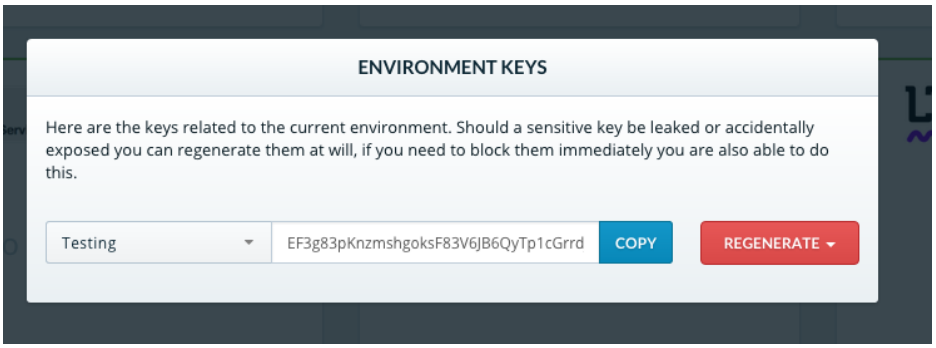
In order to get the authorization keys to use the Mashape API, you'll need to sign up for a Mashape account.

1. On Mashape.com (http://mashape.com), click **Sign Up Free** in the upper-right corner and create an account.
2. Click **Applications** on the top navigation bar, and then select **Default Application**.
3. In the upper-right corner, click **Get the Keys**.



Note: If you don't see the Get the Keys button, make sure you click **Applications > Default Application** on the top navigation bar first.

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



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

Text editor tips

When you're working with code, you use a text editor 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)
- 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 WebStorm when I'm working with Jekyll projects.)

Submit requests through Postman

GUI clients make REST calls a little easier

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 calls with specific names
- More easily enter information in the right format
- See the response in a prettified JSON view or a raw format

Common GUI clients

Some popular GUI clients include the following:

- Postman (https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en) (Chrome app)
- Advanced REST Client (https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo) (Chrome browser extension)
- REST Console (https://chrome.google.com/webstore/detail/rest-console/cokgbflfommojglbmbpenpphppikmonn)
- Paw (https://luckymarmot.com/paw) (Mac, $30)

Learn by doing, then deep dive

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. I think this reflects how people use software and documentation.

If it seems like I'm glossing over some things now, like what a GET method is or a resource URI, hang in there. When we deep dive into that point later, and things will be a lot clearer.

Make a call in Postman

1. If you haven't already done so, download and start the Postman app for Chrome (https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en).

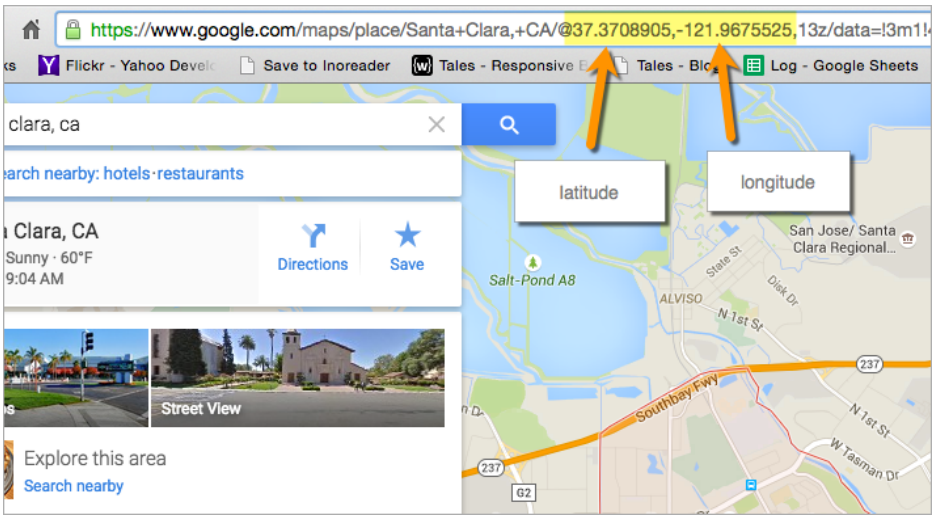
   > **Note:** There's also a Postman *extension* for Chrome, but you want the Chrome app. It's more functional.

2. You'll make a REST call for the first endpoint (`aqi`). Select **GET** for the method.

3. Insert the endpoint into the box next to GET: `https://simple-weather.p.mashape.com/aqi`

4. Click the **Params** button and insert `lat` and `lng` parameters with specific values (other than `1`).

   For example, `lat: 37.354108` and `lng: -121.955236` . You can find latitude and longitude values from the URL in Google Maps when you go to a specific location.



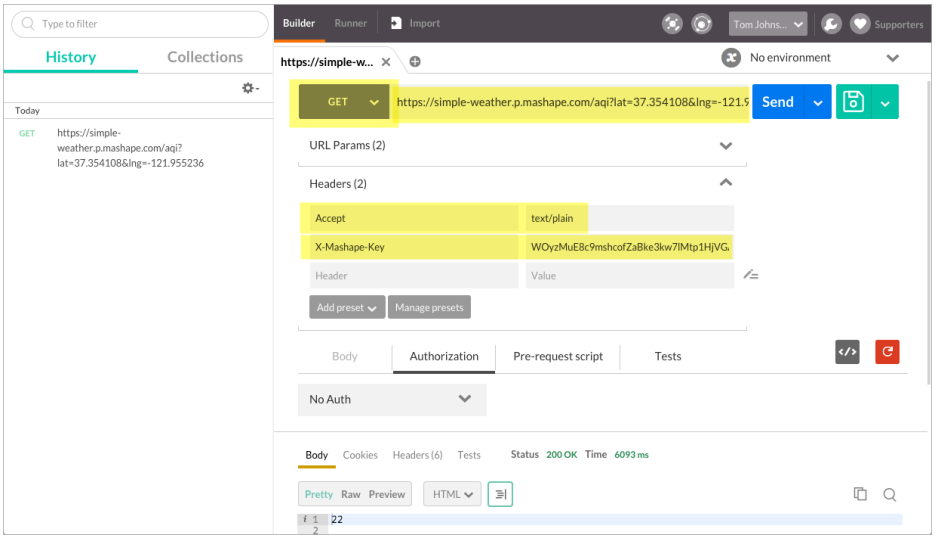
You can also find coordinates for a location using mapcoordinates.net (http://www.mapcoordinates.net/en).

When you add these parameters, they will dynamically be added as a query string to the endpoint URI. The query string is the string followed by the `?` in the endpoint URI.

5. Expand the **Headers** section and insert the key value pairs: `Accept: application/json` and `X-Mashape-Key: {api key}`. (Omit the colons, and swap in your own API key.)

6. Insert the endpoint URL (with the query string parameters) into the field next to GET.

It should look like this:



7. Click **Send**.

   The response appears. In this case, the response is text only. You can switch the format to HTML, JSON, XML, or other formats.

   > **Note:** Usually the responses are more detailed JSON. Notice that the header set the `Accept` type as `text/plain`.

Save the call

1. In Postman, click the **Save** button (the floppy disk next to Send).
2. Create a new collection (e.g., weather) by typing the collection name in the **or create a new one** box.
3. Type a name for the request (e.g., aqi).
4. At the bottom of the dialog box, click **Add to Collection.**

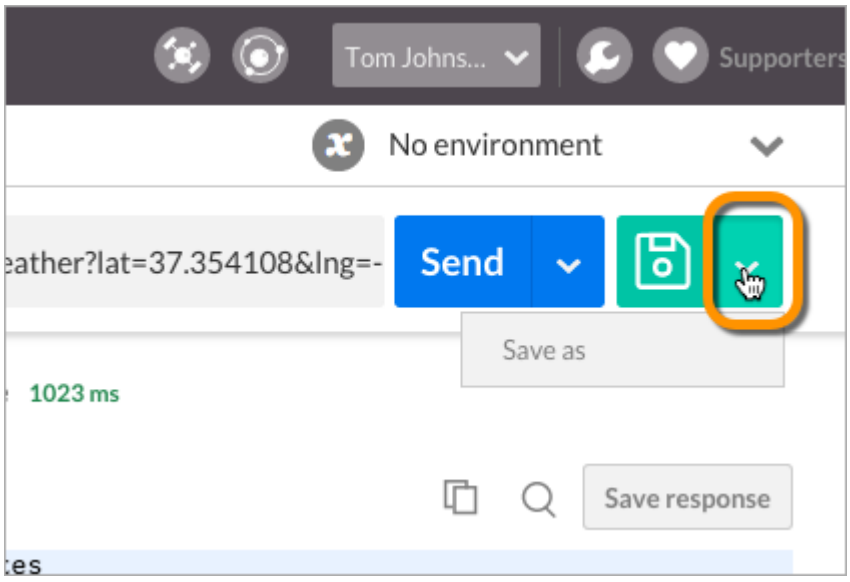
Saved endpoints appear in the left column under Collections.

Make calls for the other endpoints

Enter details into Postman for the other two endpoints:

- weather
- weatherdata

When you save these other endpoints, click the arrow next to Save and choose **Save as**. Otherwise you'll overwrite the settings of the existing call.



(Alternatively, click the **+** button and create new tabs each time.)

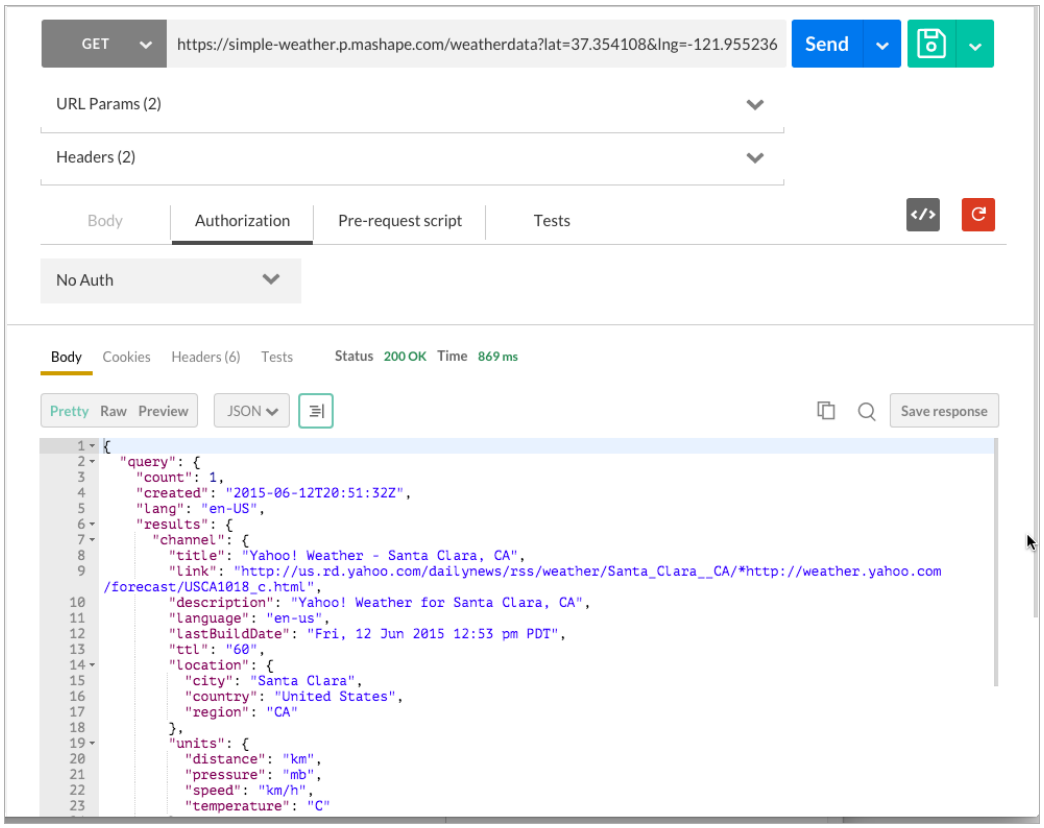
> **Note:** The Accept header tells the browser what format you will accept the response in. The Accept header for the `weatherdata` endpoint is `application/json`, whereas the first two are `application/text`.

View the format of the weatherdata response in pretty JSON

Activity

The first two endpoint responses include text only. The weatherdata endpoint response is in JSON format.

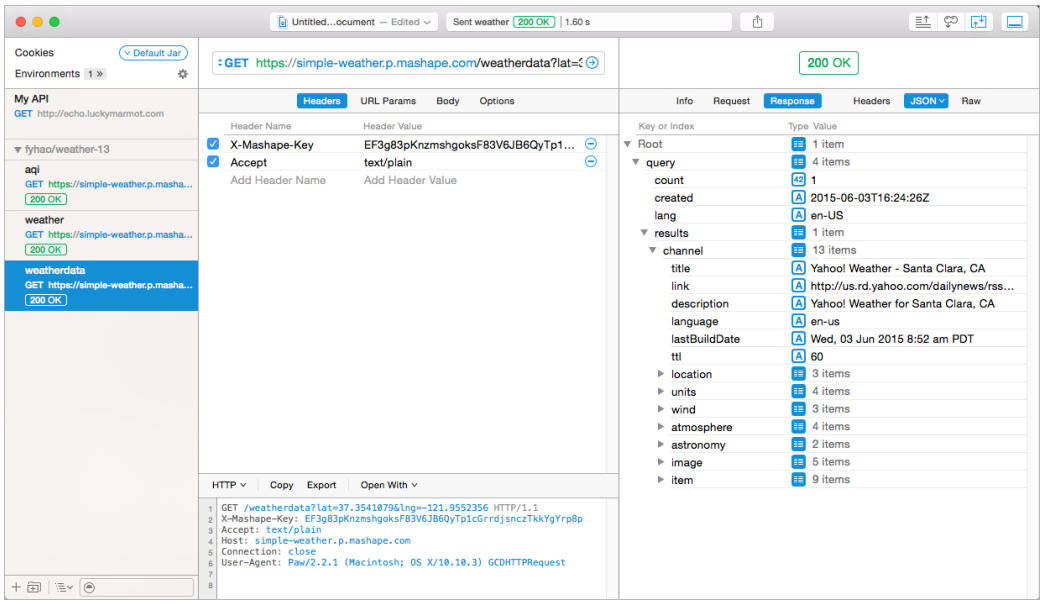
In Postman, run the weatherdata call. Then toggle the options to **Pretty** and **JSON**.



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

> ☑ **Tip:** To "prettify" content means to un-minify it and format it with white space that is readable.

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



Paw also allows you to easily see the request headers, response headers, URL parameters, and other data.

Installing cURL

It's better that you install cURL before the course so that you aren't bogged down with technical issues when you're trying to focus on the course material. cURL is usually available by default on Macs but requires some installation on Windows.

Installing cURL

Follow these instructions for installing cURL.

Mac

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

1. Open Terminal (press **Cmd + spacebar** to open Spotlight, and then type "Terminal").

2. In Terminal type `curl -V` . The response should look something like this:

```
curl 7.37.1 (x86_64-apple-darwin14.0) libcurl/7.37.1 Sec
ureTransport zlib/1.2.5
Protocols: dict file ftp ftps gopher http https imap ima
ps ldap ldaps pop3 pop3s rtsp smtp smtps telnet tftp
Features: AsynchDNS GSS-Negotiate IPv6 Largefile NTLM NT
LM_WB SSL libz
```

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

To make a test API call, submit this:

```
curl --get -k --include "https://simple-weather.p.mashape.com/a
qi?lat=37.354108&lng=-121.955236" -H "X-Mashape-Key: EF3g83pKnz
mshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p" -H "Accept: text/plai
n"
```

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

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 Zendesk article: Installing and using cURL (https://support.zendesk.com/hc/en-us/articles/203691436-Installing-and-using-cURL#install).

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 7.37.1 (x86_64-apple-darwin14.0) libcurl/7.37.1 SecureTran
sport zlib/1.2.5
Protocols: dict file ftp ftps gopher http https imap imaps lda
p ldaps pop3 pop3s rtsp smtp smtps telnet tftp
Features: AsynchDNS GSS-Negotiate IPv6 Largefile NTLM NTLM_WB S
SL libz
```

To make a test API call, submit this:

```
curl --get -k --include "https://simple-weather.p.mashape.com/a
qi?lat=37.354108&lng=-121.955236" -H "X-Mashape-Key: EF3g83pKnz
mshgoksF83V6JB6QyTp1cGrrdjsnczTkkYgYrp8p" -H "Accept: text/plai
n"
```

Note: 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.)

☑ **Tip:** 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-cant-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.
- By adding `-k` in the cURL command, you bypass cURL's security certificate. In this case

Make a cURL call

About cURL

While Postman is convenient, it's hard to represent just how to make the calls. Plus, different users probably use different GUI clients, or none at all. Instead of describing how to make REST calls using a GUI client, the most conventional method is to explain how to make the calls using cURL.

cURL is a command-line utility that lets you excecute HTTP requests with different parameters and methods. In this section, you'll use cURL to make these same requests.

Prepare the call in cURL format

1. Go back into the Weather API (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/aqi?lat=1.0&lng=1.0' -H 'X-Mashape-Key: {api key}' -H 'Accept: text/plain'
```

3. If you're on Windows, change the single quotation marks to double quotation marks, and add `-k` as well to work around security certificate issues.

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

4. Swap in your own API key.

> **Note:** In the instruction here, `{api key}` will be used instead of an actual API key. You should replace that part with your own API key. Omit the curly braces `{ }`.

Make the call in cURL (Mac)

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

> **Tip:** If you plan on working in Terminal a lot, use iTerm (https://www.iterm2.com/) instead of Terminal.)

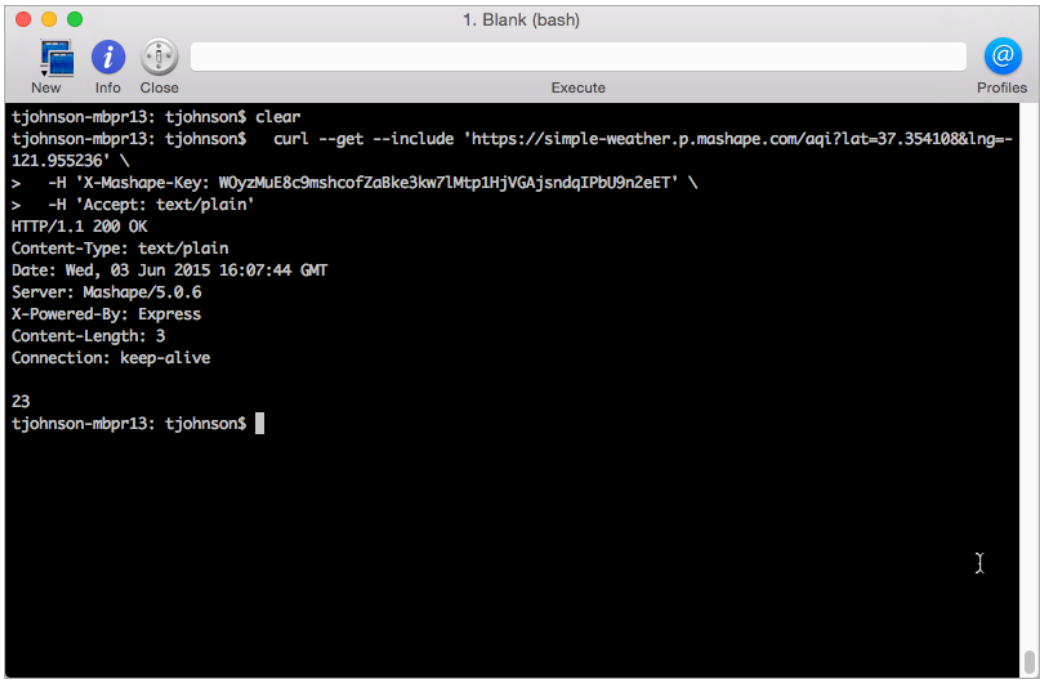
2. Paste the call you have in your text editor into the command line.

   My call looks like this:

```
  curl --get --include 'https://simple-weather.p.mashap
e.com/aqi?lat=37.354108&lng=-121.955236' -H 'X-Mashape-K
ey: {api key}' -H 'Accept: text/plain'
```

3. Press your **Enter** key.

You should see something like this as a response:



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 call in cURL (Windows 7)

1. Copy the cURL call from your text editor.

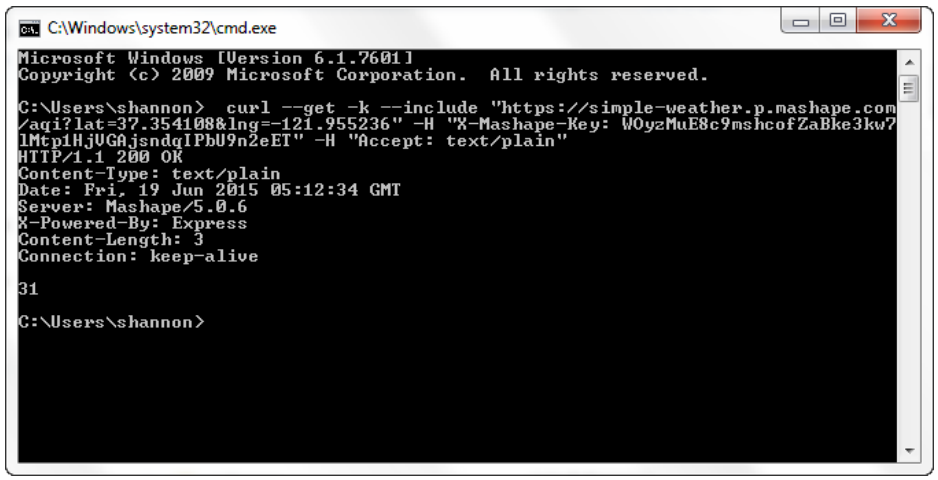
2. Go to **Start** and type **cmd** to open up the commandline. (If you're on Windows 8, see these instructions for accessing the commandline (http://pcsupport.about.com/od/windows-8/a/command-prompt-windows-8.htm).)

3. Right-click and then select **Paste** to insert the call. My call looks like this:

```
curl --get -k --include "https://simple-weather.p.mashape.com/aqi?lat=37.354108&lng=-121.955236" -H "X-Mashape-Key: {api key}" -H "Accept: text/plain"
```

Warning: Make sure you use double quotes and include the `-k`.

The response looks like this:



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 the 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 a path.

Understand cURL more

cURL is a cross-platform way to show requests and responses

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

> ⚠ **Important:** Almost every API shows how to interact with the API using cURL.

REST APIs follow the same request/return 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 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). 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 only see 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
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Date: Fri, 19 Jun 2015 05:58:50 GMT
Etag: "359670651"
Expires: Fri, 26 Jun 2015 05:58:50 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (rhv/818F)
X-Cache: HIT
x-ec-custom-error: 1
Content-Length: 1270
```

The header contains the metadata about the response. All of this information is transferred to the browser whenever you make requests to URLs (that is, when you go to web pages), but the browser doesn't show you this information from you. You can see it 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 when working with REST endpoints:

HTTP METHOD	DESCRIPTION
POST	Create a resource
GET	Read a resource
PUT	Update a resource
DELETE	Delete a resource

Note: When 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/aq
i?lat=37.354108&lng=-121.955236' \
  -H 'X-Mashape-Key: {api key}' \
  -H 'Accept: text/plain'
```

cURL has shorthand names for the various options 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
`--get`	The HTTP method to use. (This is actually unnecessary. You can remove this and the request returns the same response.)
`--include`	Whether to show the headers in the response. Also represented by `-i`.
`-H`	Submits a custom header. Include an additional `-H` for each header you're submitting.

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

Query strings

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. 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
`-i` or `--include`	Include the response headers in the response.	`curl -i http://www.example.com`
`-d` or `--data`	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.	`curl -d "data-to-post" http://www.example.com`

CURL COMMAND	DESCRIPTION	EXAMPLE
`-H` or `--header`	Submit the request header to the resource. This is very common with REST API requests because the authorization is usually included here.	`curl -H "key:12345" http://www.example.com`
`-X POST`	The HTTP method to use with the request (in this example, `POST`). If you use `-d` in the request, cURL automatically specifies a POST method. With GET requests, including the HTTP method is optional, because GET is the default method used.	`curl -X POST -d "resource-to-update" http://www.example.com`
@filename	Load content from a file	`curl -X POST -d @my-body.json http://www.example.com`

See the cURL documentation (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:MI
A}" 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 \
```

The `Accept` header instructs the server to process the post body as JSON.

Test your memory

Fill in the blanks to see how much you remember:

- `–i` means...
- `–H` means...
- `–X POST` means...
- `–d` means...

See the cURL parameters (http://127.0.0.1:4033//docapis/docapis_answers#curlParameters) on the answer page to check your responses.

More Resources

To learn more about cURL with REST documentation, see REST-esting with cURL (http://blogs.plexibus.com/2009/01/15/rest-esting-with-curl/).

☑ **Tip:** 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. You won't constantly be needing to add things "to your path."

Using methods with cURL (petstore example)

Posting to the body

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/), but without actually using the Swagger UI (which is something we'll explore later). For now, we just need an API that we can create, update, and delete content from. (You're just getting familiar with cURL here.)

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. Don't worry so much about understanding the petstore API.

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.

> ☑ **Tip:** A lot of APIs require you to post requests containing JSON messages in the body. This is often how you configure something. The list of JSON key-value pairs that the API accepts is called the "model schema" 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`.

> **Note:** Use unique numbers and names that others aren't likely to also use.

3. Save the file in this directory: `Users/{your username}`.

4. In your Terminal, browse to the directory where you saved the mypet.json file. (Usually the default directory is `Users/{your username}` — hence the previous step.)

   If you've never browsed directories using the command line, note these essential commands:

   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, find your current directory by typing `cd` (or just look at the prompt path). 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. Once your Terminal 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 "htt
p://petstore.swagger.io/v2/pet"
```

The response should look something like this:

```
{"id":51231236,"category":{"id":4,"name":"testexecutio
n"},"name":"fluffernutter","photoUrls":["string"],"tag
s":[{"id":0,"name":"string"}],"status":"available"}
```

☑ **Tip:** 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, now multiplet pets). Todd Fredrich explains idempotency by comparing it to a pregnant cow (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 "htt
p://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" "htt
p://petstore.swagger.io/v2/pet/51231236"
```

   The response contains your pet name and other information:

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

   You can format the JSON by pasting it into a JSON formatting tool (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" "htt
   p://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" "htt
   p://petstore.swagger.io/v2/pet/5123123"
   ```

   You should see this error message:

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

Conclusion

This example has 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 usage. Quality assurance teams often construct advanced test scenarios that iterate through a lot of cURL requests.

From Postman to cURL and back again

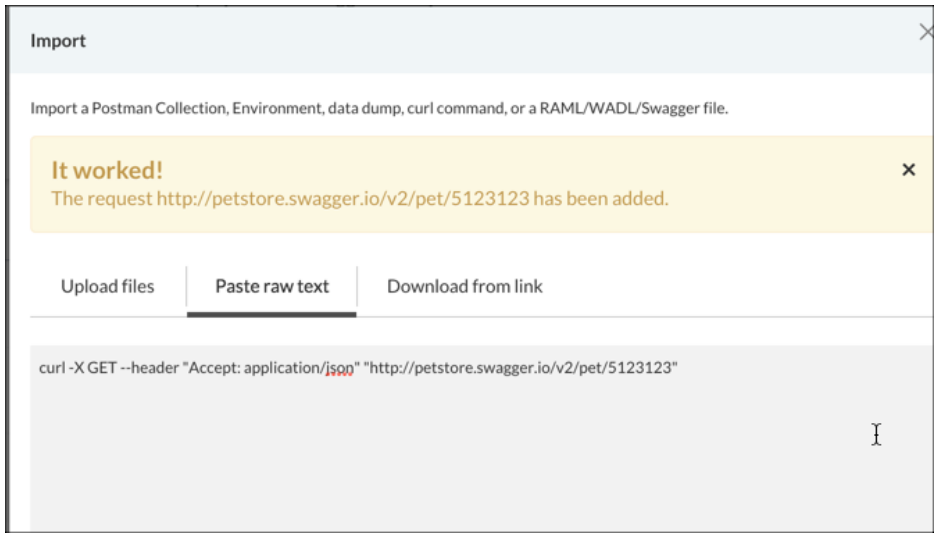
You can import cURL commands into Postman, and you can export Postman commands to cURL format.

To import cURL into Postman:

1. Open a new tab in Postman and click the **Import** button.

2. Select **Paste Raw Text** and insert your cURL command:

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

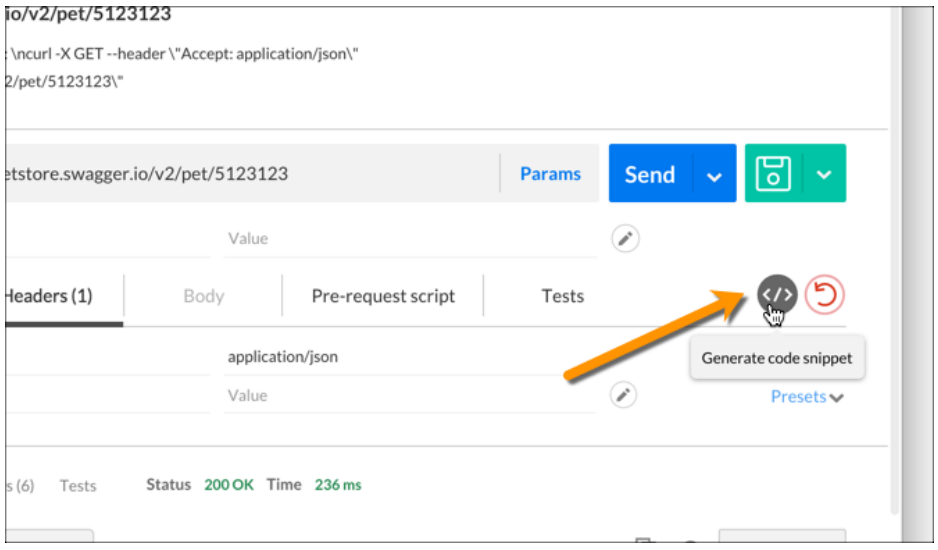


Make sure you don't have any extra spaces at the beginning.

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

To export Postman into cURL:

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



2. Select **Curl** from the menu.
3. Copy the code snippet.

```
curl -X GET -H "Accept: application/json" -H "Cache-Cont
rol: no-cache" -H "Postman-Token: e40c8069-21db-916e-9a9
4-0b9a42b39e1b" 'http://petstore.swagger.io/v2/pet/51231
23'
```

You can see that Postman adds some extra header information into the request. This extra header information is unnecessary and can be removed.

Analyze the JSON response

Prettify the JSON response

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

```
{"query":{"count":1,"created":"2015-06-03T16:24:26Z","lang":"e
n-US","results":{"channel":{"title":"Yahoo! Weather - Santa Cla
ra, CA","link":"http://us.rd.yahoo.com/dailynews/rss/weather/Sa
nta_Clara__CA/*http://weather.yahoo.com/forecast/USCA1018_c.htm
l","description":"Yahoo! Weather for Santa Clara, CA","languag
e":"en-us","lastBuildDate":"Wed, 03 Jun 2015 8:52 am PDT","tt
l":"60","location":{"city":"Santa Clara","country":"United Stat
es","region":"CA"},"units":{"distance":"km","pressure":"mb","sp
eed":"km/h","temperature":"C"},"wind":{"chill":"16","directio
n":"0","speed":"0"},"atmosphere":{"humidity":"67","pressure":"1
014.8","rising":"0","visibility":"16.09"},"astronomy":{"sunris
e":"5:46 am","sunset":"8:23 pm"},"image":{"title":"Yahoo! Weath
er","width":"142","height":"18","link":"http://weather.yahoo.co
m","url":"http://l.yimg.com/a/i/brand/purplelogo//uh/us/news-we
a.gif"},"item":{"title":"Conditions for Santa Clara, CA at 8:5
2 am PDT","lat":"37.35","long":"-121.95","link":"http://us.rd.y
ahoo.com/dailynews/rss/weather/Santa_Clara__CA/*http://weathe
r.yahoo.com/forecast/USCA1018_c.html","pubDate":"Wed, 03 Jun 20
15 8:52 am PDT","condition":{"code":"30","date":"Wed, 03 Jun 20
15 8:52 am PDT","temp":"16","text":"Partly Cloudy"},"descriptio
n":"\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. Hig
h: 22 Low: 13<br />\nThu - AM Clouds/PM Sun. High: 22 Low: 13<b
r />\nFri - AM Clouds/PM Sun. High: 24 Low: 14<br />\nSat - AM
Clouds/PM Sun. High: 24 Low: 15<br />\nSun - Partly Cloudy. Hig
h: 26 Low: 16<br />\n<br />\n<a href=\"http://us.rd.yahoo.com/d
ailynews/rss/weather/Santa_Clara__CA/*http://weather.yahoo.com/
forecast/USCA1018_c.html\">Full Forecast at Yahoo! Weathe
r</a><BR/><BR/>\n(provided by <a href=\"http://www.weather.co
m\" >The Weather Channel</a>)<br/>\n","forecast":[{"code":"3
0","date":"3 Jun 2015","day":"Wed","high":"22","low":"13","tex
t":"AM Clouds/PM Sun"},{"code":"30","date":"4 Jun 2015","da
y":"Thu","high":"22","low":"13","text":"AM Clouds/PM Sun"},{"co
de":"30","date":"5 Jun 2015","day":"Fri","high":"24","low":"1
4","text":"AM Clouds/PM Sun"},{"code":"30","date":"6 Jun 201
5","day":"Sat","high":"24","low":"15","text":"AM Clouds/PM Su
n"},{"code":"30","date":"7 Jun 2015","day":"Sun","high":"26","l
ow":"16","text":"Partly Cloudy"}],"guid":{"isPermaLink":"fals
e","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 "prettify" 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/weathe
r/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA101
8_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//u
h/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/wea
ther/Santa_Clara__CA/*http://weather.yahoo.com/forecast/USCA101
8_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 />\n
Partly Cloudy, 16 C<BR />\n<BR /><b>Forecast:</b><BR />\nWed -
AM Clouds/PM Sun. High: 22 Low: 13<br />\nThu - AM Clouds/PM Su
n. High: 22 Low: 13<br />\nFri - AM Clouds/PM Sun. High: 24 Lo
w: 14<br />\nSat - AM Clouds/PM Sun. High: 24 Low: 15<br />\nSu
n - Partly Cloudy. High: 26 Low: 16<br />\n<br />\n<a href=\"ht
tp://us.rd.yahoo.com/dailynews/rss/weather/Santa_Clara__CA/*htt
p://weather.yahoo.com/forecast/USCA1018_c.html\">Full Forecast
at Yahoo! Weather</a><BR/><BR/>\n(provided by <a href=\"htt
p://www.weather.com\" >The Weather Channel</a>)<br/>\n",
               "forecast":[
                  {
                     "code":"30",
                     "date":"3 Jun 2015",
                     "day":"Wed",
                     "high":"22",
                     "low":"13",
                     "text":"AM Clouds/PM Sun"
                  },
                  {
                     "code":"30",
                     "date":"4 Jun 2015",
                     "day":"Thu",
                     "high":"22",
                     "low":"13",
                     "text":"AM Clouds/PM Sun"
                  },
                  {
                     "code":"30",
                     "date":"5 Jun 2015",
                     "day":"Fri",
```

```
                  "high":"24",
                  "low":"14",
                  "text":"AM Clouds/PM Sun"
               },
               {
                  "code":"30",
                  "date":"6 Jun 2015",
                  "day":"Sat",
                  "high":"24",
                  "low":"15",
                  "text":"AM Clouds/PM Sun"
               },
               {
                  "code":"30",
                  "date":"7 Jun 2015",
                  "day":"Sun",
                  "high":"26",
                  "low":"16",
                  "text":"Partly Cloudy"
               }
            ],
            "guid":{
               "isPermaLink":"false",
               "content":"USCA1018_2015_06_07_7_00_PDT"
            }
         }
      }
   }
}
```


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.

Some APIs return information in both JSON and XML. But if you're trying to parse through the response and render it on the web, 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 or Boolean, 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 objects can contain strings, numbers, booleans, arrays, or other objects.

With numbers or booleans (true or false values), you don't use quotation marks.

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

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

Mixing it up

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
   }
]
```

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 `[ ]`.

☑ **Tip:** It's important to understand the difference between objects and arrays because it determines how you access the data to pull out and display the information. More on that later.

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.

Note that the escaping (`\` tags) in the `description` is to keep the JSON formatting valid.

More information

For more information on understanding the structure of JSON, see json.com (https://www.json.com/).

Logging JSON responses to the console

Making use of the JSON response

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 in displaying the REST response on a web page.

Parse and display REST JSON response

Mashape provides some sample code (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.

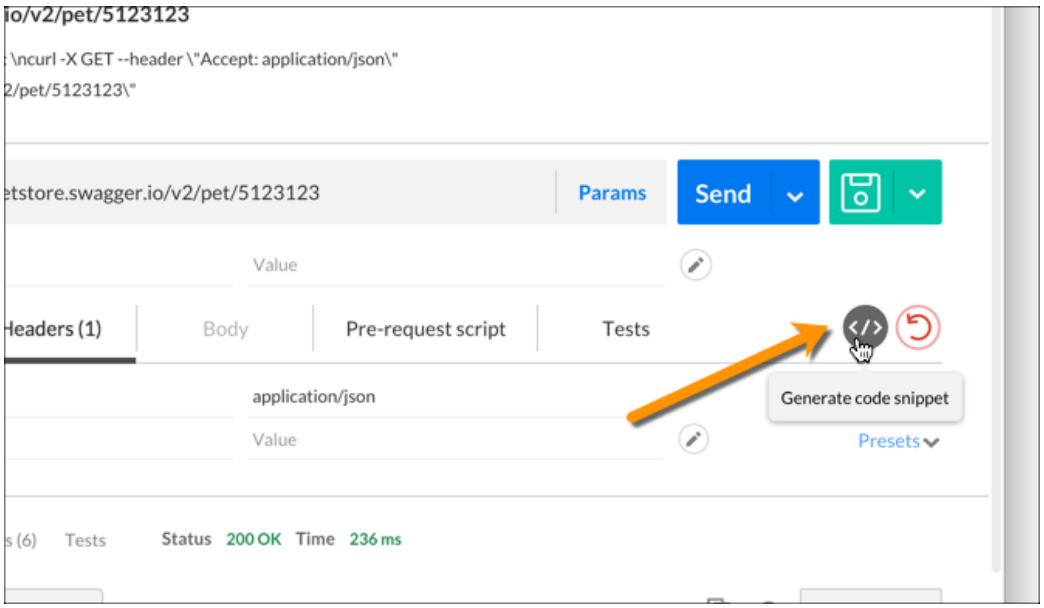
1. Start with a basic HTML template with jQuery referenced, like this:

```
<html>
<body>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.1
1.1/jquery.min.js"></script>

</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.



1. Select **JavaScript > jQuery AJAX**.

2. Copy the code sample.

3. Insert the Postman code sample between `<script>` tags in your template.

4. The Postman code sample needs one more parameter: `dataType`. Add `"dataType": "json"` as parameter in `settings`.

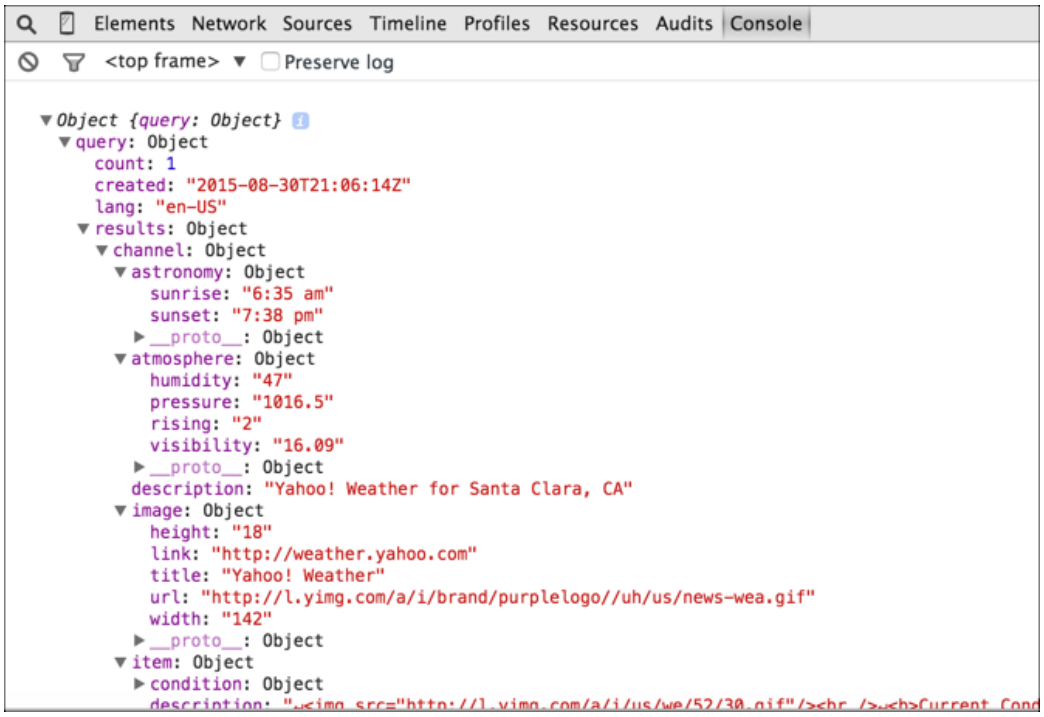
Your code should look like this:

```
<html>
<body>
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://simple-weather.p.mashape.com/weatherdata?la
t=37.354108&lng=-121.955236",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "{api key}"
  }
}

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

1. Start Chrome and open the JavaScript Console by going to **View > Developer > JavaScript Console**.

2. Open the weatherdata.html file in Chrome (**File > Open File**).

The weatherdata response should be logged to the JavaScript console. You can inspect the payload by expanding the sections.



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 code samples is the `ajax` method from jQuery.

In brief, the `ajax` method takes an argument, such as `settings`.

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

This `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).

Some important values are the `url`, which is the URI or endpoint you are submitting the request to. Another 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.

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 or not. 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. Find the section that appears here: object > query > results > channel > item > description.

Access the JSON values

Accessing JSON values

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.

> **Note:** 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 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.

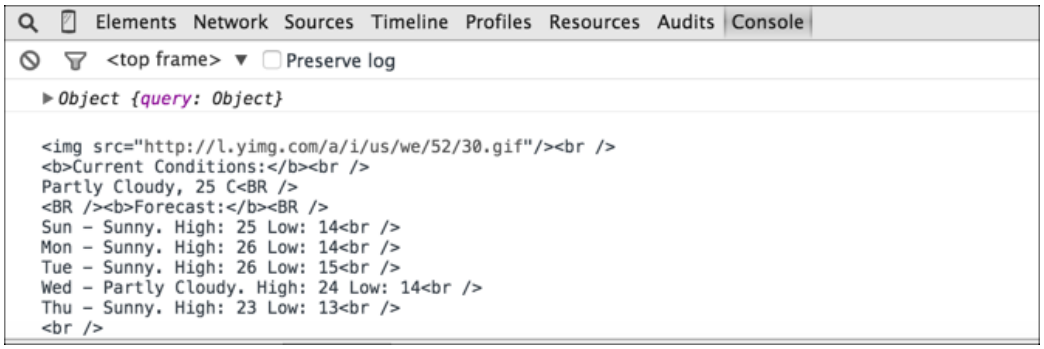
To pull out the description element from the JSON response and display it on the page, add this to your code sample, right below the `console.log(response)` part:

```
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:



Printing the JSON 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 (whichever you prefer).

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.descripti
   on";
   ```

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

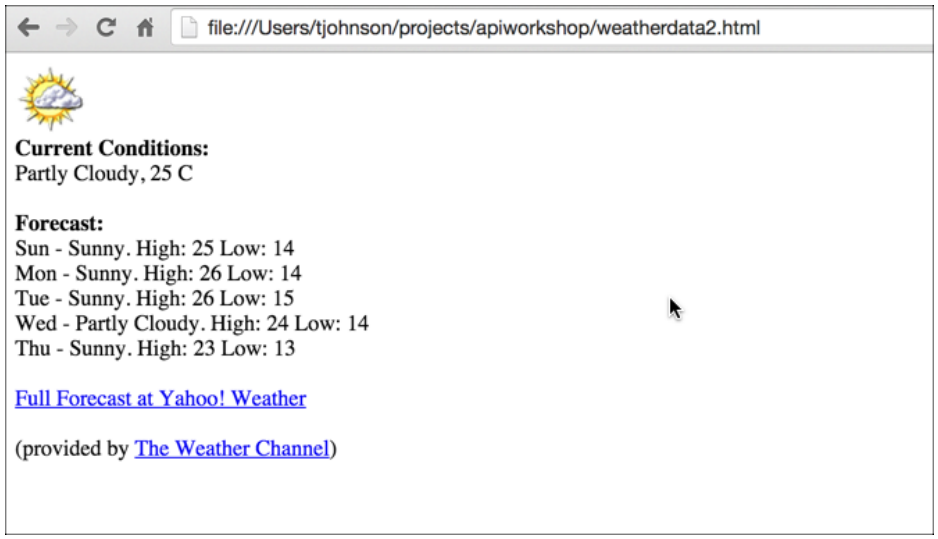
   ```
   $("#weatherDescription").append(content);
   ```

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

   Your entire code should look as follows:

```
<html>
<body>

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

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

$.ajax(settings)

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

  var content = response.query.results.channel.item.desc
ription;
  $("#weatherDescription").append(content);
});
</script>

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

Here's the result:



Now change the display to access the wind speed instead.

Diving into dot notation

How dot notation works

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`, I 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:

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

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

`glove` is the third item in the array.

> **Note:** With programming, you usually start counting at 0.

Activity 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.j
s"></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 custom-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`.
    - green
    - nike
    - goldenrod
    - Sarah

Check your work by looking at the Dot Notation section (page 0) on the answers page.

Showing wind conditions on the page

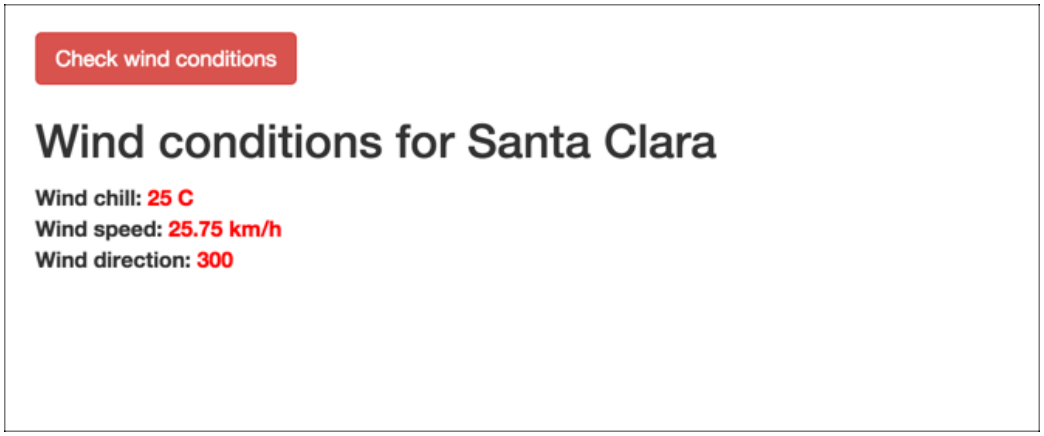
At the beginning of the course, I showed an example of embedding the wind speed and other details on a website. Now let's revisit this code example and see how it's put together. Copy the following code into a basic HTML page, customize the `{api key}` value, and view it in the browser:

```
<html>
<head>
<script src="http://code.jquery.com/jquery-2.1.1.min.js"></scri
pt>
<link rel="stylesheet"  href='https://maxcdn.bootstrapcdn.com/b
ootstrap/3.3.4/css/bootstrap.min.css' rel='stylesheet' type='te
xt/css'>

  <title>Sample Query to get the wind</title>
<style>
   #wind_direction, #wind_chill, #wind_speed, #temperature, #sp
eed {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?la
t=37.354108&lng=-121.955236",
    "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "{api key}"
  }
}

  $.ajax(settings)

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

    $("#wind_speed").append (response.query.results.channel.win
d.speed);
    $("#wind_direction").append (response.query.results.channe
l.wind.direction);
    $("#wind_chill").append (response.query.results.channel.win
d.chill);
    $("#temperature").append (response.query.results.channel.un
its.temperature);
```

```
    $("#speed").append (response.query.results.channel.units.sp
eed);
  });
}
</script>


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

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

<b>Wind chill: </b><span id="wind_chill"></span> <span id="temp
erature"></span></br>
<b>Wind speed: </b><span id="wind_speed"></span> <span id="spee
d"></span></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, it's wrapped inside a function called `checkWind` . When the button is clicked, it fires the `checkWind()` function through the `onclick` method. * When `checkWind` runs, it pulls out the wind chill, speed, and direction and writes them to several ID tags on the page. Units for each of these values is 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:



New API endpoint to document

Shift perspectives: Now you're the technical writer

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 now a technical writer working with the weather API team. The team is asking you to document a new endpoint.

You have a new endpoint to document

The project manager calls you over and says 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/) 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 any information like this at all, 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.

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 miliseconds 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 riptide 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.

> **Note:** 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 and Markdown syntax (https://help.github.com/articles/github-flavored-markdown/).

Documenting resource descriptions

The terminology to describe a "resource" varies

When it comes to the right terminology to describe the resource, practices vary. Exactly what are the things that you access using a URL? Here are some of the terms used in API docs:

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

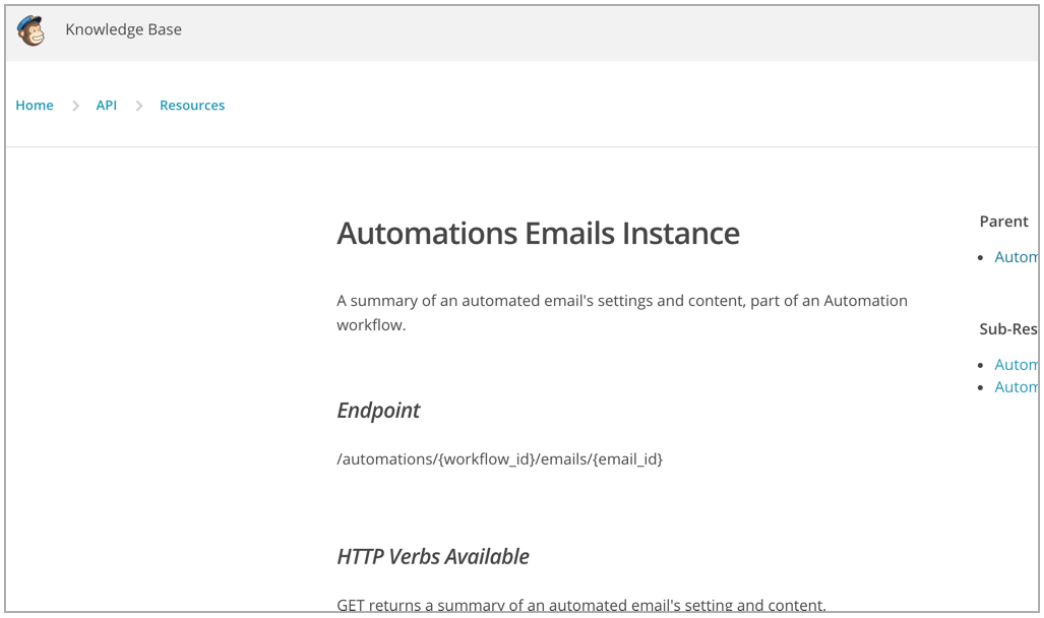
Some docs get around the situation by not calling them anything explicitly.

You could probably choose the terms that you like best. My favorite is to use *resources* and *endpoints*. 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 called a resource.

☑ **Tip:** A URI (Uniform Resource Identifier) describes what something is, whereas a URL (Uniform Resource Location) tells you where to locate it. Hence the resource itself is a URI that you access using a URL.

Some examples

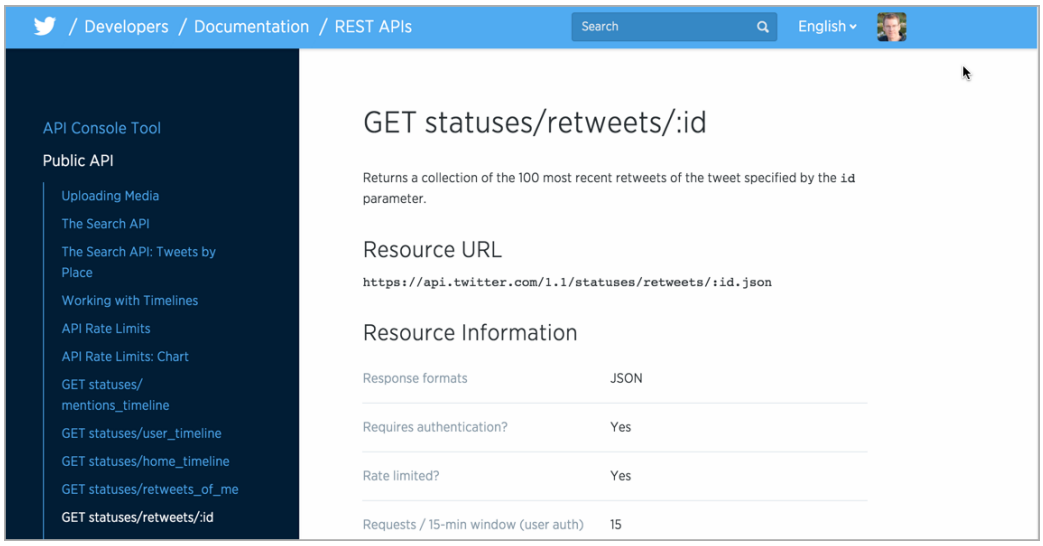
Take look at Mailchimp's API for an example (http://kb.mailchimp.com/api/resources/automations/emails/automations-emails-instance).



(http://kb.mailchimp.com/api/resources/automations/emails/automations-emails-instance)

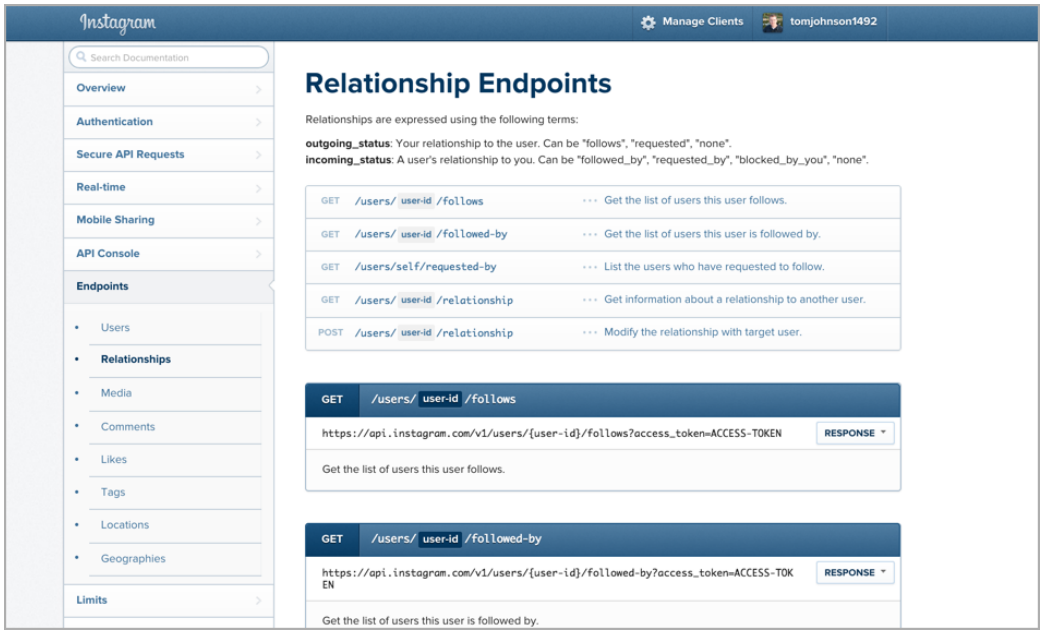
With Mailchimp, the resource might be "Automations Emails Instance." The endpoint to access this resource is `/automations/{workflow_id}/emails/{email_id}`.

In contrast, look at Twitter's API. This page is called GET statuses/retweets/:id (https://dev.twitter.com/rest/reference/get/statuses/retweets/%3Aid). To access it, you use the Resource URL `https://api.twitter.com/1.1/statuses/retweets/:id.json` (`https://api.twitter.com/1.1/statuses/retweets/:id.json`).



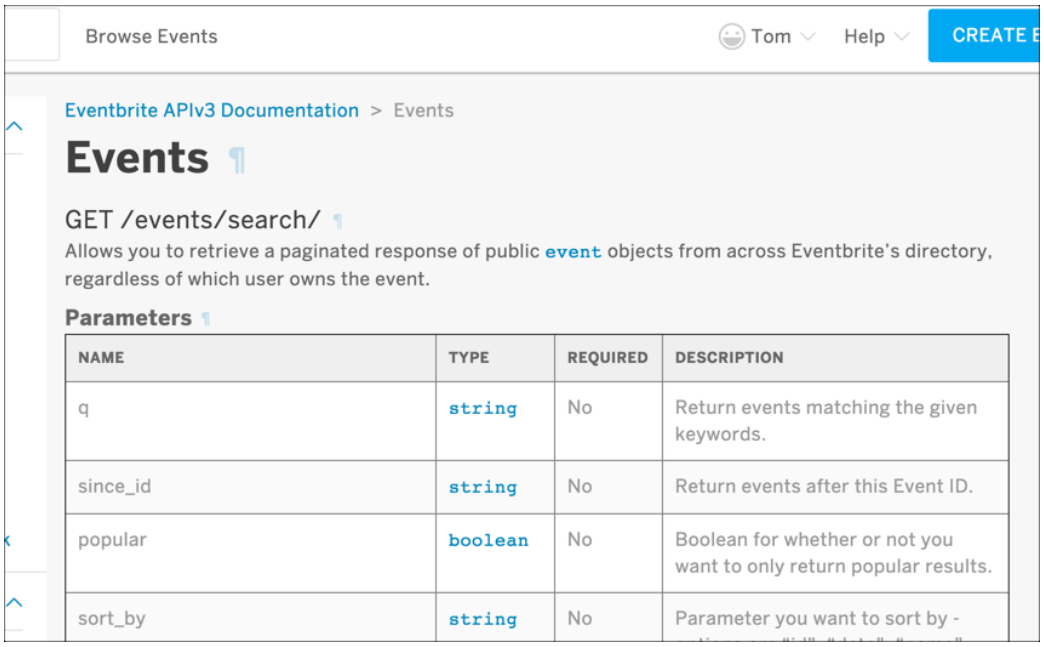
(https://dev.twitter.com/rest/reference/get/statuses/retweets/%3Aid)

Here's the approach by Instagram. Their doc calls it "endpoints" in the plural -- e.g., "Relationship endpoints," with each endpoint listed on the relationship page.



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

The EventBrite API shows a list of endpoints, but when you go to an endpoint, what you're really seeing is an object. On the object's page you can see the variety of endpoints you can use with the object.



(page 89)

☑ **Tip:** Remember the distinction between resources and endpoitns. A resource (or "object") 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.

When I worked at Badgeville, our endpoints somewhat looked like this:

```
api_site.com/{apikey}/users
// gets all users

api_site.com/{apikey}/users/{userId}
// gets a specific user

api_site.com/{apikey}/rewards
// gets all rewards

api_site.com/{apikey}/rewards/{rewardId}
// gets a specific reward

api_site.com/{apikey}/users/{userId}/rewards
// gets all rewards for a specific user

api_site.com/{apikey}/users/{userId}/rewards/{rewardId}
// gets a specific reward for a specific user

api_site.com/{apikey}/users/{userId}/rewards/{missionId}
// gets the rewards for a specfic mission related to a specifi
c user

api_site.com/{apikey}/missions/{missionid}/rewards
// gets the rewards available for a specific mission
```

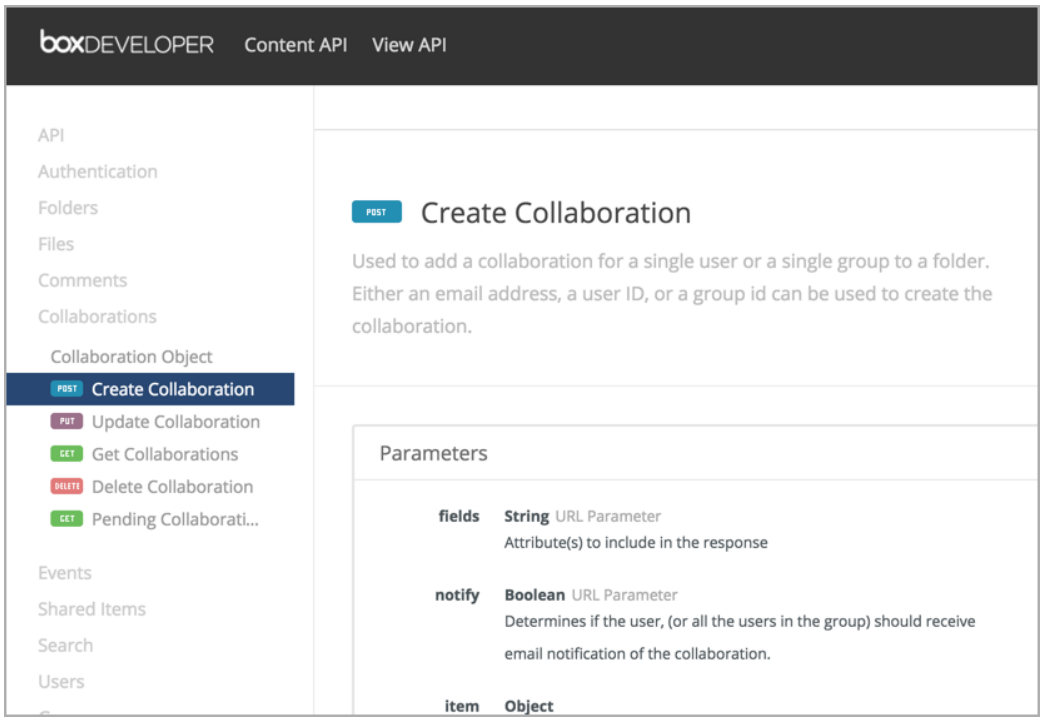
Depending on how you construct the endpoint paths determines the response. A rewards resource had various endpoints that returned different types of information related to rewards.

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

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

The same resource can have multiple endpoints

The Box API has a good example of how the same resource can have multiple endpoints and methods.



The Box example has 5 different endpoints or methods you can call. Each of these methods lets you access the Comments resource or object in different ways. Why call it an object? When you get the Comments resource, the JSON is an object.

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

When describing the resource, start with a verb

Review the surf report wiki page (page 0) containing the information about the endpoint, and try to describe the endpoint in the length of one or two tweets (140 characters).

The resource description usually starts with a verb and is a fragment. Here are some examples:

- Delicious API (https://github.com/SciDevs/delicious-api/blob/master/api/posts.md#v1postsupdate)

- Foursquare API (https://developer.foursquare.com/docs/venues/menu)
- Box API (https://box-content.readme.io/#add-a-comment-to-an-item)

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

After drafting the outline, 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."

I think these description are too short. But developers like concision. If shortening the surfreport description, you could also write:

```
surfreport
Provides surf condition information.
```

Recognize the difference between reference docs versus user guides

One thing to keep in mind is the difference betweeen 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, you 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).

Re-using the resource description

The description of the endpoint is likely something you'll re-use in different places:

- Product overviews
- Tutorials
- Code samples

As a result, put a lot of effort into crafting it.

Documenting the endpoint definitions and methods

Terminology for the endpoint definition varies

Now let's document the endpoints. When you list the endpoint definitions / URLs, the term you use to describe this section also varies. Here are some terms you might see:

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

My preferred term is "endpoint."

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 example, the endpoint is just `/surfreport/{beachId}`. You don't have to list the full URL every time. Doing so distracts the user from focusing on the path that matters.

In your user guide, you explain the full code path (which usually includes an API key) in an introductory section.

Represent parameter values with curly braces

If you have parameter values, represent them through curly braces. For example, here's an example from Mailchimp's API:

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

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

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

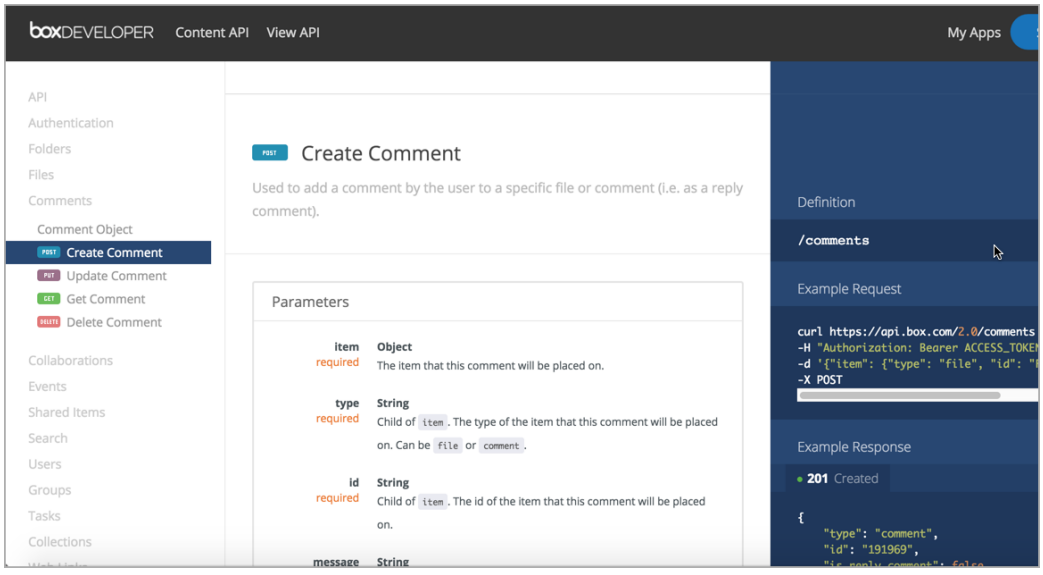
```
/campaigns/:campaign_id/actions/send
```

You can see this convention in the EventBrite API (https://www.eventbrite.com/developer/v3/).

In general, if the placeholder name is ambiguous as to whether it's a placeholder or what you're really supposed to write in the path, clarify it.

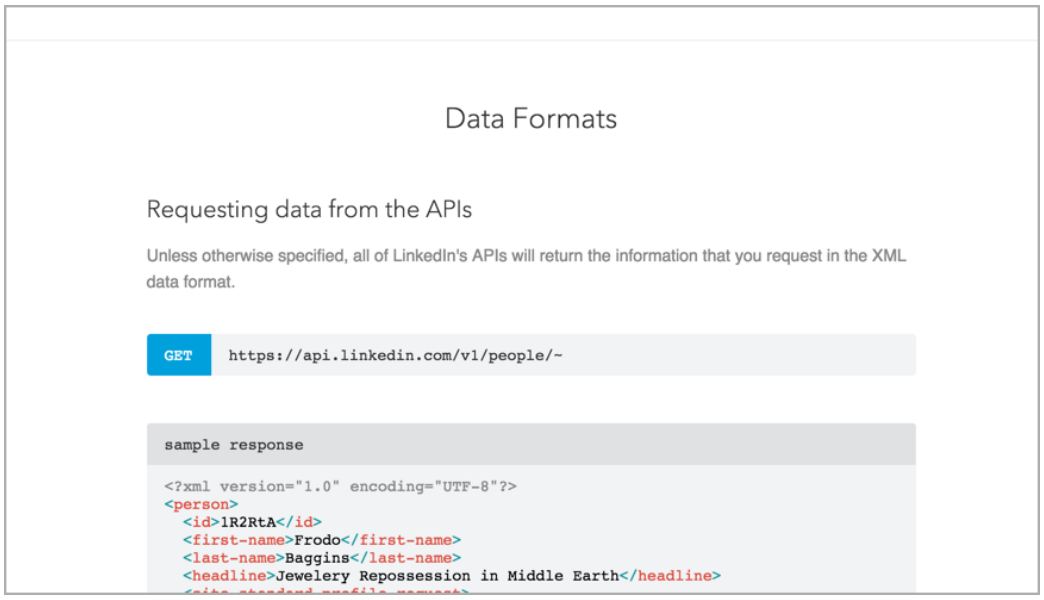
You can list the method beside the endpoint

It's common to list the method next to the endpoint. Since there's not much to say about the method itself, it makes sense to group it with the endpoint. Here's an example from Box's API:



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

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



(https://developer.linkedin.com/docs/rest-api)

☑ **Tip:** Sometimes the method is referred to as the "verb". GET, PUT, POST, and DELETE are all verbs or actions, after all.

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}`

If you had different endpoints for the same resource, you might have more to say here. But with this example, the bulk of the description is with the resource.

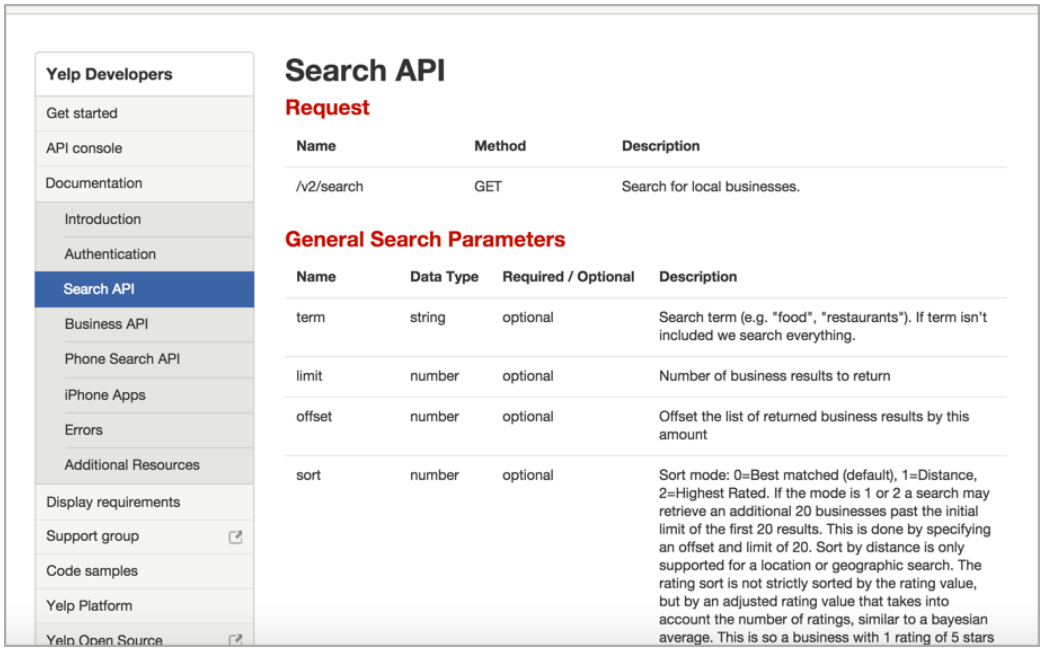
Documenting parameters

Parameters are ways to configure the endpoint

Parameters refer to the various ways the endpoint can be configured to influence the result. Many times parameters are listed out in a simple table like this:

PARAMETER	REQUIRED?	DATA TYPE	EXAMPLE
format	optional	string	json

Here's an example from Yelp's documentation:



(https://www.yelp.com/developers/documentation/v2/search_api)

You can format the values in a variety of ways. If using a definition list or other non-table format, you probably have styles that make these values easily readable.

Data types indicate the format for the values

It's important to list the data type for each parameter because APIs may not process the parameter correctly if it's not formatted in the right way. These data types are the most common:

- **string**: An alphanumeric sequence of letters and possibly 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. In Java, for example, it's important to note the data type allowed because the program allocates 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 Bangalore, for example, `12.9539974` and `77.6309395`, the response is `Not Supported - IN - India - IN-KA`. 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. Usually the order in which parameters are passed to the endpoint does not matter.

For example:

```
/surfreport/{beachId}?days=3&units=metric&time=1400
```

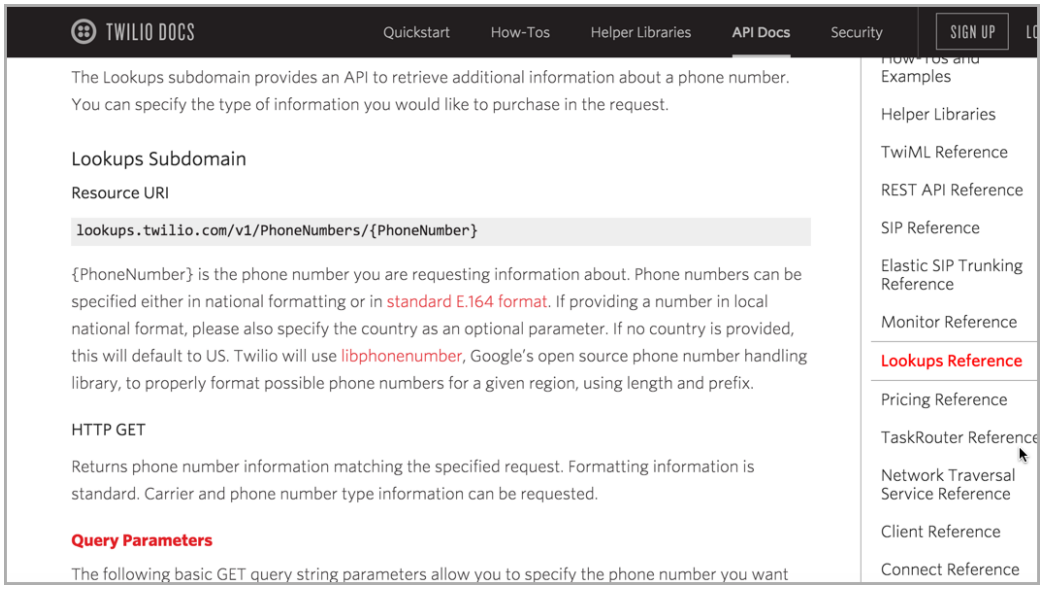
and

```
/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:



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

The `{PhoneNumber}` value is described in the description of the endpoint rather than in another section that lists the query parameters you can pass to the endpoint.

Here are a few other details to remember when describing parameters:

- Note whether the parameter has a maximum or minimum value.
- Note whether the parameters are optional or required.

☑ **Tip:** When you test the 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.

Passing parameters in the JSON body

Not all APIs use browser URLs to submit requests. Sometimes REST APIs allow only server-to-server communication. With this approach, you might pass parameters in the body of the POST in a JSON format.

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

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

This kind of submission is common when you have a lot of parameters available. Sometimes the parameters submitted can be quite numerous, such as 20 or 30 key-value pairs.

Time values usually follow ISO or Unix formats

Time values usually follow either the ISO-8601 format (http://en.wikipedia.org/wiki/ISO_8601), or the Unix timestamp format (http://en.wikipedia.org/wiki/Unix_time). You can get the Unix time using a converter (http://www.unixtimestamp.com/).

The Unix time is the number of milliseconds to the current date from January 1, 1970. This number gives you an easy integer to insert into a resource URL, but it's not very readable. In contrast, the ISO-8601 format is more readable but more susceptible to formatting or time zone errors.

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.	Integer
units	Optional	Options are either `imperial` or `metric`. 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. `metric` is the default.	string

PARAMETER	REQUIRED	DESCRIPTION	TYPE
time	Optional	If you include the time, then only the current hour will be returned in the response.	integer. Unix format (ms since 1970) in UTC.

Documenting sample requests

The sample request clarifies how to use the endpoint

Although you've already listed the endpoint and parameters, you also include a sample request that shows the endpoint that integrates parameters in an easy-to-understand way.

Here's an example from the NYTimes API:

SEARCH API V2
Requests
g a Search Query
Examples
Back to Top
All APIs
API Console

Article Search API v2

With the Article Search API, you can search New York Times articles from Sept. 18, 1851 to today, retrieving headlines, abstract lead paragraphs, links to associated multimedia and other article metadata.

Note: In URI examples and field names, *italics* indicate placeholders for variables or values. Brackets [ ] indicate optional items. Parentheses ( ) are not a convention — when URIs include parentheses, interpret them literally.

The Article Search API at a Glance	
Base URI	`http://api.nytimes.com/svc/search/v2/articlesearch`
HTTP method	GET
Response formats	JSON (`.json`), JSONP (`.jsonp`)
Quick links	Requests | Constructing a Search Query | Examples

Requests

URI STRUCTURE

Article Search requests use the following URI structure:

```
http://api.nytimes.com/svc/search/v2/articlesearch.response-format?[q=search
term&fq=filter-field:(filter-term)&additional-params=values]&api-key=####
```

PARAMETERS

(http://developer.nytimes.com/docs/read/article_search_api_v2)

In this case, adding the parameters looks kind of confusing because there's a square bracket in there after the query string.

Additionally, it's hard to tell at a glance which are the parameters and which are the sample inserted values.

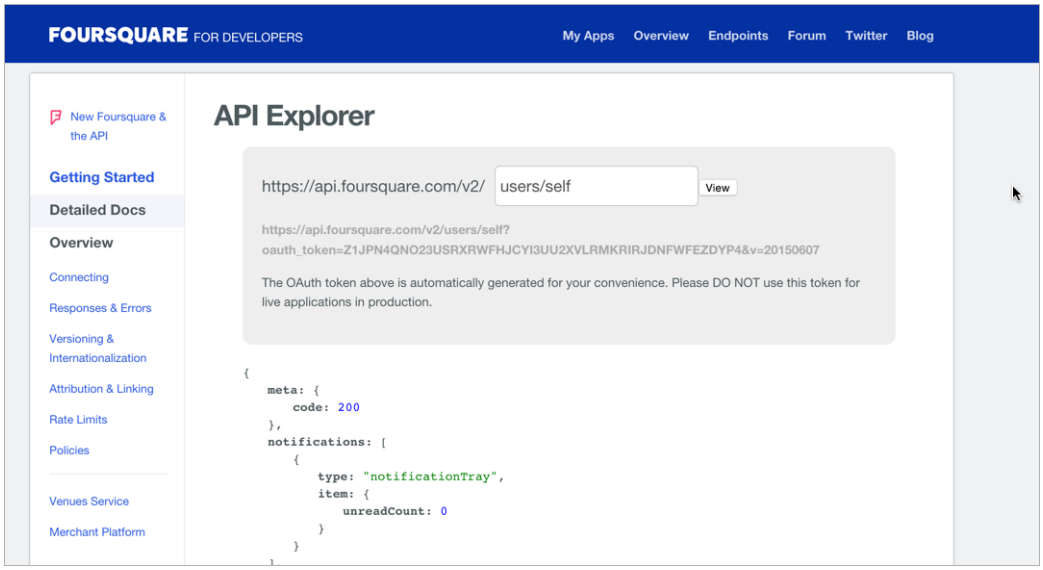
For example, `q=search term` is not what you'd enter. `q` is the parameter, and `searchterm` is the placeholder added that represents a real search term.

Similarly, only the `fq` is the parameter. The `filter-field:(filter-term)` is just placeholder text. If you read the description of the parameter, you see that you need to use a "Filtered search query using standard Lucene syntax." Well, shoot, it would be nice to see a few examples of Lucene syntax to help get going!

Fortunately, the NYTimes API provides numerous request examples (http://developer.nytimes.com/docs/read/article_search_api_v2#examples) showing various filter queries. If you have a lot of different ways to pass parameters to the endpoint, then definitely show several requests.

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:



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

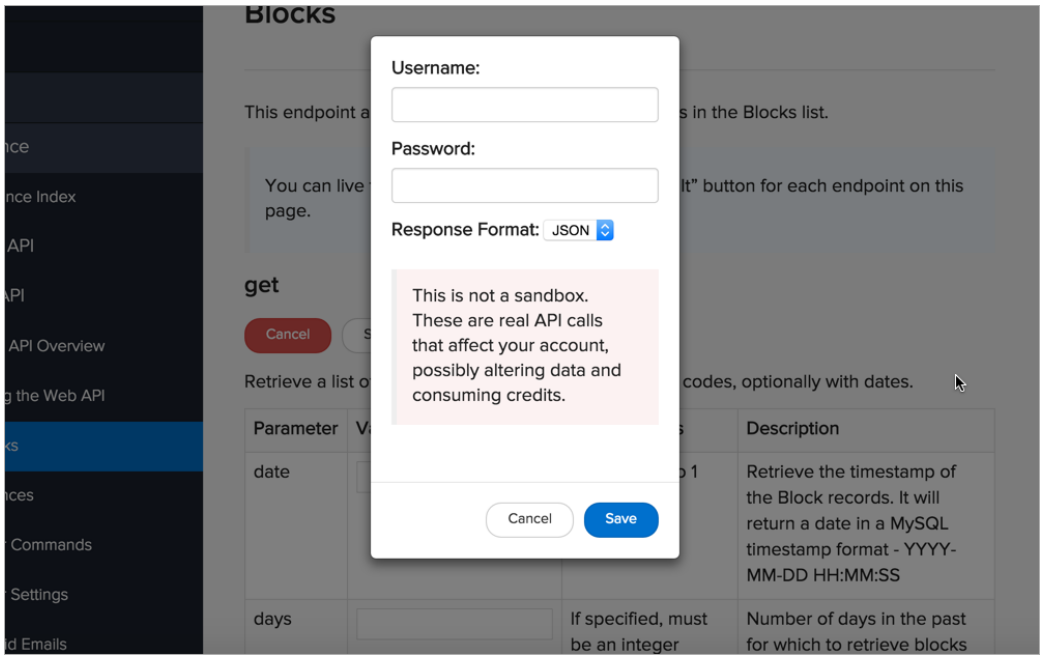
The API Explorer lets you insert your own values, your own API key, and other parameters into a request so you can see the responses directly in the Explorer. Being able to see your own data maybe makes the response more real and immediate.

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.

API Explorers can be dangerous in the hands of a newbie

Additionally, the API Explorer can be a dangerous addition to your site. What if a novice user trying out a DELETE method accidentally removes data? And 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 up their data. With IBM's Watson 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)

Some platforms such as Swagger (http://swagger.io/) and Readme.io (http://readme.io) can integrate an API Explorer functionality directly into your documentation.

As far as integrating other API Explorer tooling, this is a task that should be relatively easy for developers. All the Explorer does it 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.

If different requests return different responses, show multiple responses

One of the problems with showing a sample request and sample response is that different requests return different responses. Depending on the parameters you include, the response will be tailored to those parameters.

If the requests and responses vary dramatically, consider including multiple response examples. How many different requests and responses should you show? There's probably no easy answer, but probably no more than 3. You decide what makes sense for your API.

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

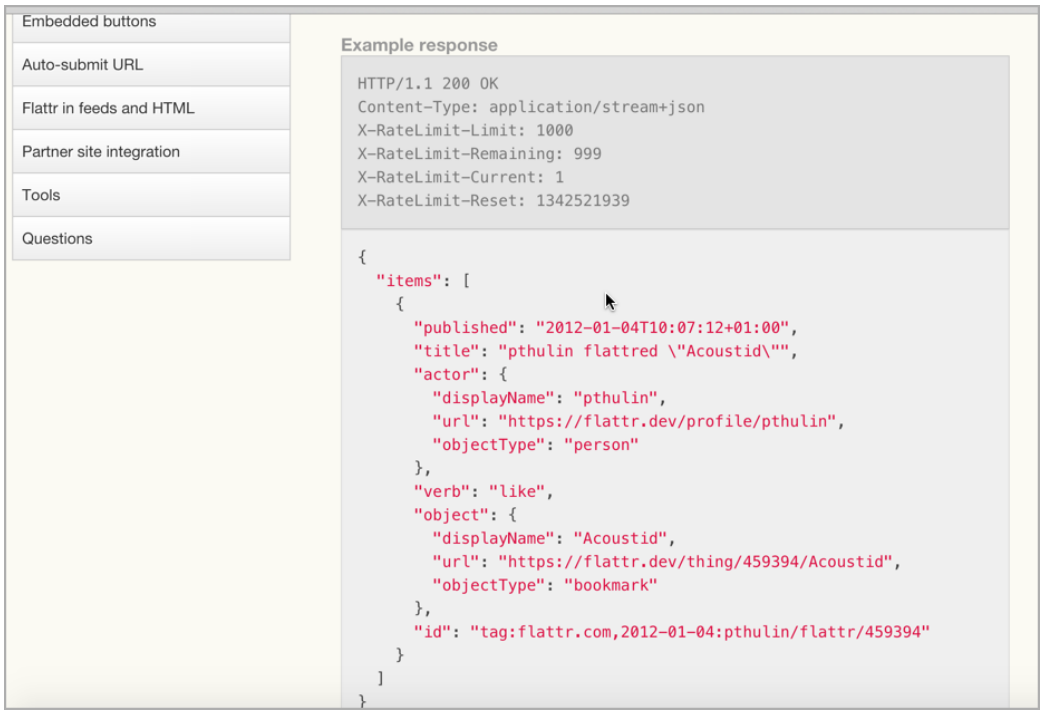
```
curl --get --include 'https://simple-weather.p.mashape.com/
surfreport/123?units=imperial&days=1&time=1433772000'
  -H 'X-Mashape-Key: {api key}'
  -H 'Accept: application/json'
```

Documenting sample responses

Provide a sample response for the endpoint

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.

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://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).

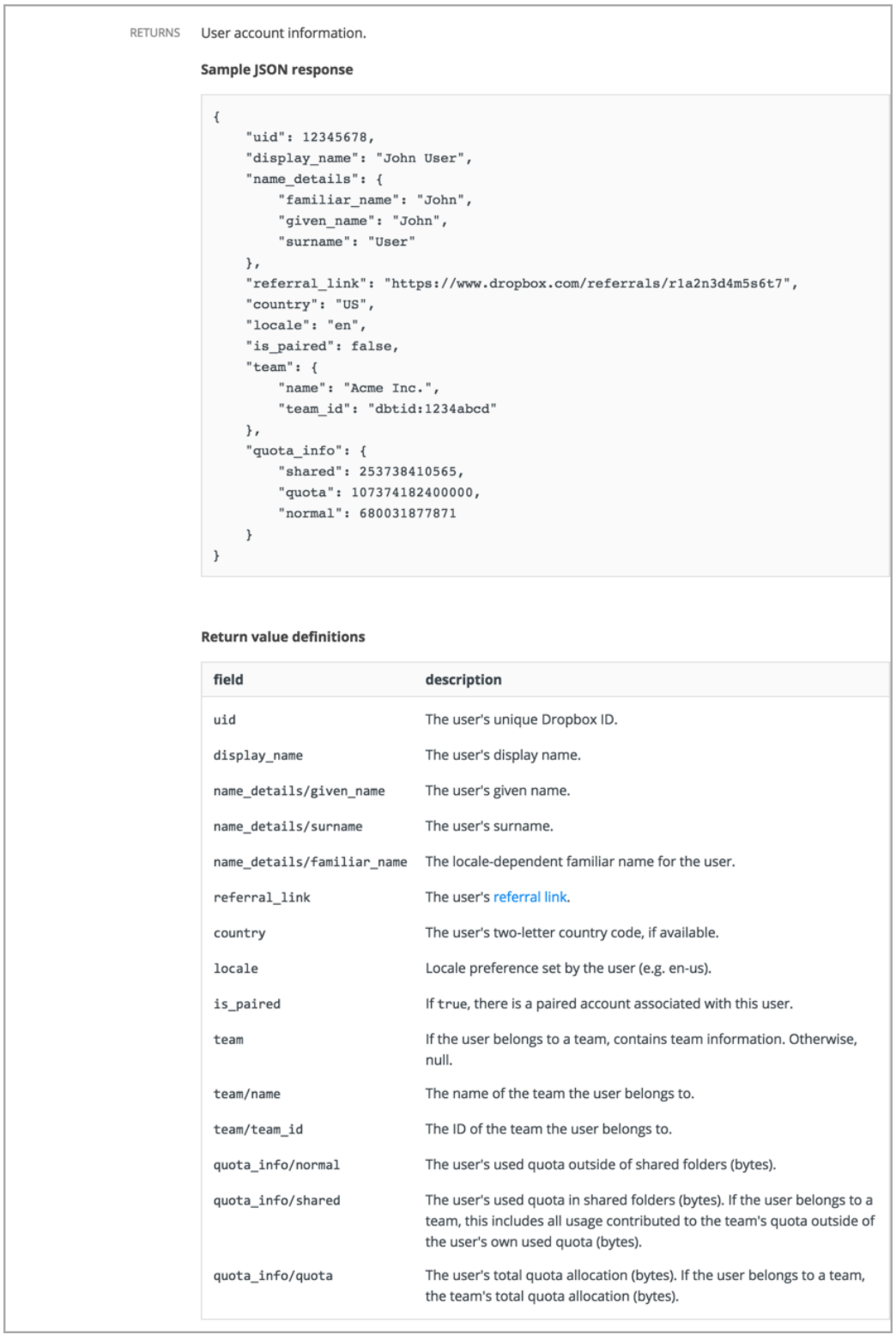
If the labels in the response are abbreviated or non-intuitive, however, you definitely should document the responses. Developers sometimes abbreviate the responses to increase performance by reducing the amount of text sent.

Plus, 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 to? Some air quality indexes are on a scale from 1 to 10. 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 by using a slash. For example, `team/name` provides the documentation for the `name` object within the `team` object.



RETURNS User account information.

Sample JSON response

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

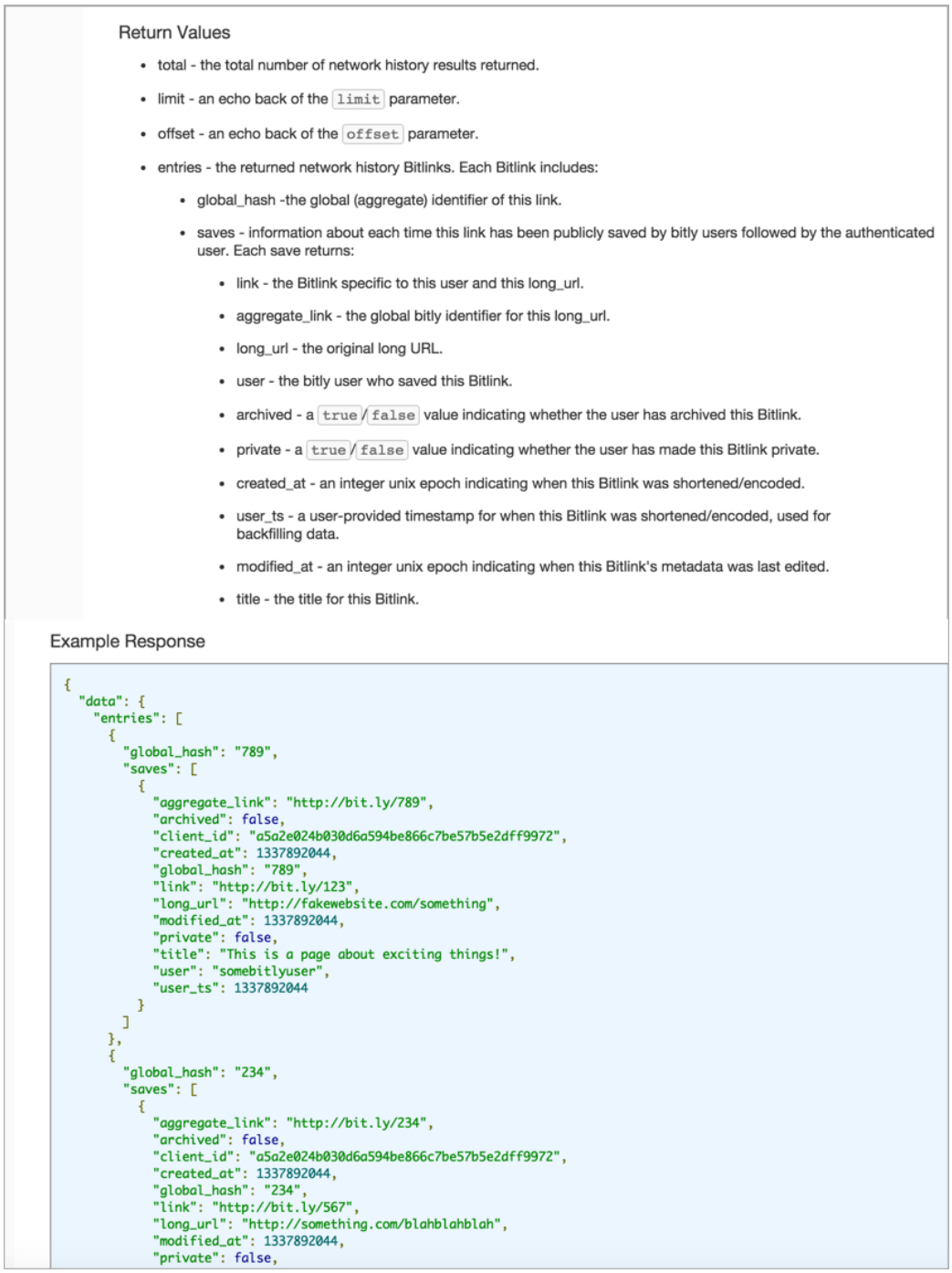
Return value definitions

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

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

☑ **Tip:** Notice how the response values are in a monospace 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.

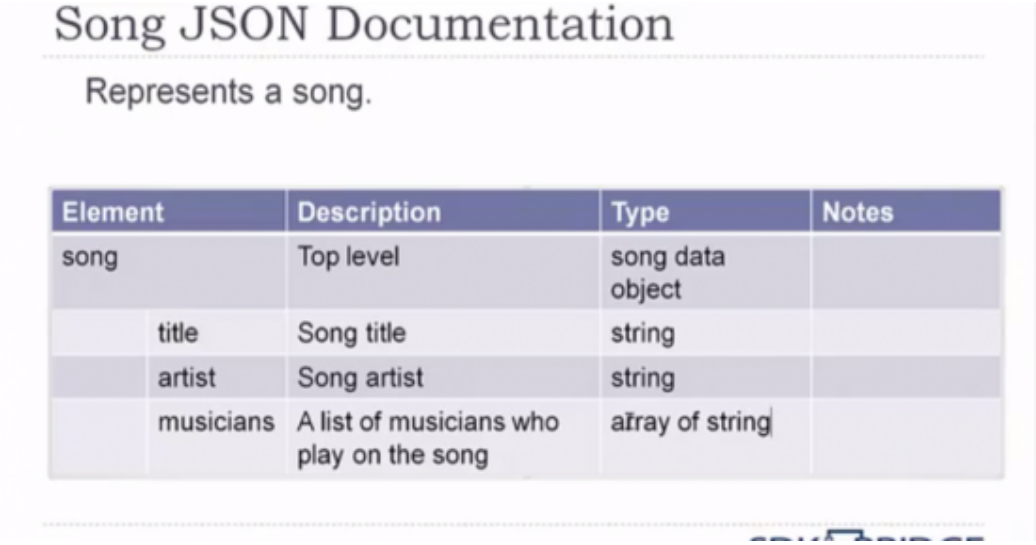
Example Response


```
{
  "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,
```


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

Personally, I think the indented approach with different levels of bullets is an eyesore.

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

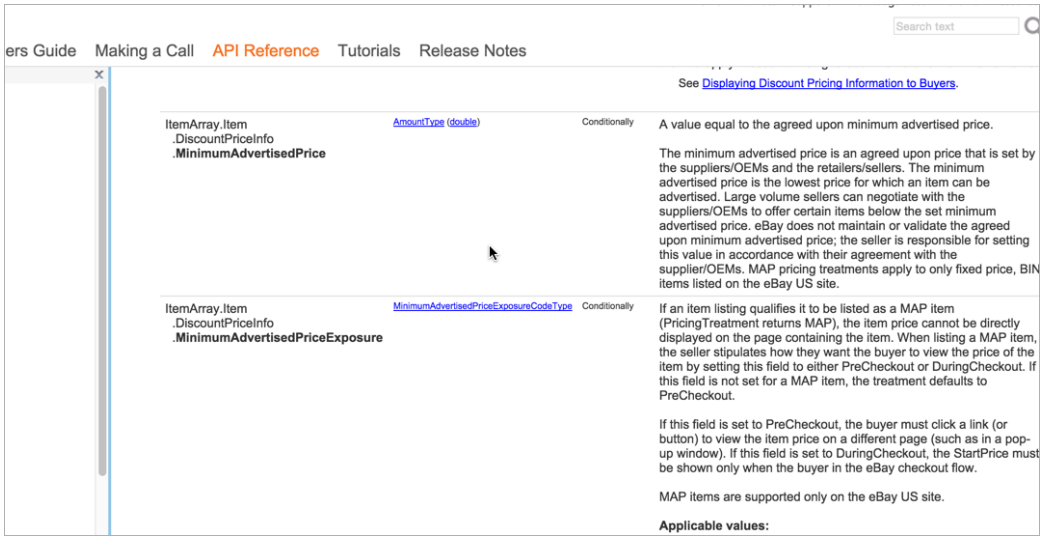
Represents a song.

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/)

(However, 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:



ers Guide Making a Call API Reference Tutorials Release Notes

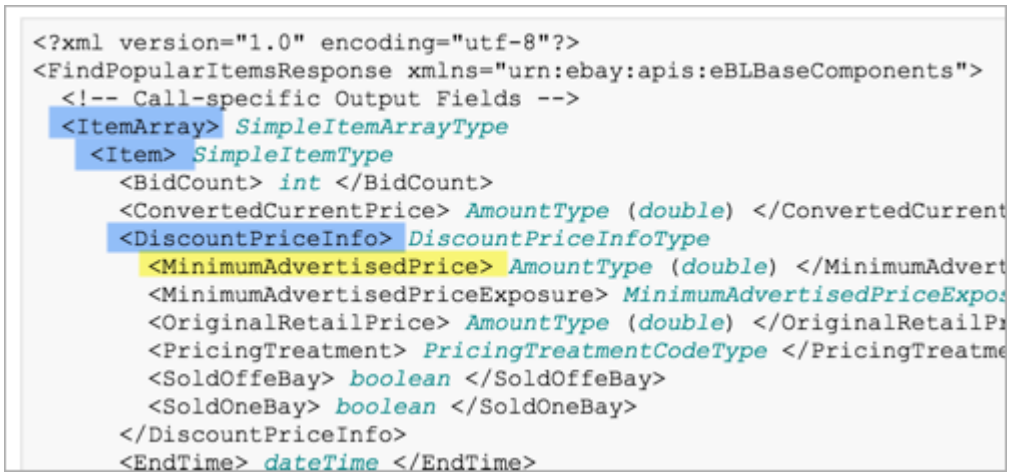
Search text

See Displaying Discount Pricing Information to Buyers.

ItemArray.Item .DiscountPriceInfo **.MinimumAdvertisedPrice**	AmountType (double)	Conditionally	A value equal to the agreed upon minimum advertised price. The minimum advertised price is an agreed upon price that is set by the suppliers/OEMs and the retailers/sellers. The minimum advertised price is the lowest price for which an item can be advertised. Large volume sellers can negotiate with the suppliers/OEMs to offer certain items below the set minimum advertised price. eBay does not maintain or validate the agreed upon minimum advertised price; the seller is responsible for setting this value in accordance with their agreement with the supplier/OEMs. MAP pricing treatments apply to only fixed price, BIN items listed on the eBay US site.
ItemArray.Item .DiscountPriceInfo **.MinimumAdvertisedPriceExposure**	MinimumAdvertisedPriceExposureCodeType	Conditionally	If an item listing qualifies it to be listed as a MAP item (PricingTreatment returns MAP), the item price cannot be directly displayed on the page containing the item. When listing a MAP item, the seller stipulates how they want the buyer to view the price of the item by setting this field to either PreCheckout or DuringCheckout. If this field is not set for a MAP item, the treatment defaults to PreCheckout. If this field is set to PreCheckout, the buyer must click a link (or button) to view the item price on a different page (such as in a pop-up window). If this field is set to DuringCheckout, the StartPrice must be shown only when the buyer in the eBay checkout flow. MAP items are supported only on the eBay US site. **Applicable values:**

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

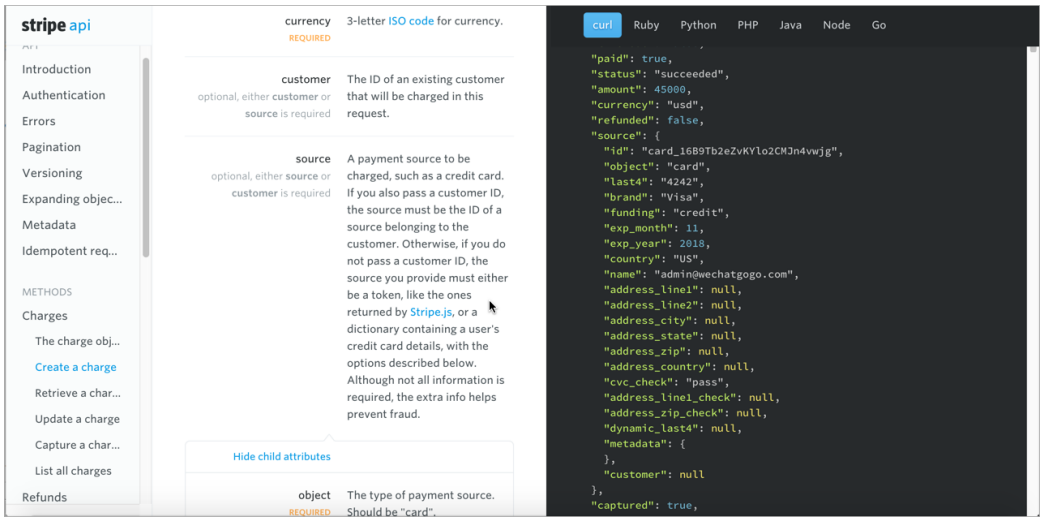
(http://developer.ebay.com/Devzone/shopping/docs/CallRef/FindPopularItems.html)

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.

> **Note:** 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.

Information design choice: 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:



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

A lot of APIs have modeled their design after Stripe's. (For example, see 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/) 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/). 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, javascript.

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.

Some APIs embed 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) and click an endpoint — it appears to be generated dynamically).

When I worked at Badgeville, we had a test/demo system we used to generate the responses. It was important that the test system has 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.

Create a section for a sample request 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 a
re okay, not great."
                }
                "3pm": {
                    "tide": -1,
                    "wind": 10,
                    "watertemp": 65,
                    "surfheight": 1,
                    "recommendation": "Not a good day for s
urfing."
                }
            }
        }
    ]
}
```

The following table describes each item in the response.*

RESPONSE ITEM	DESCRIPTION
beach	The beach you selected based on the beach ID in the request. The beach name is the official name as described in the National Park Service Geodatabase.
{day}	The day of the week selected. A maximum of 3 days get returned in the response.
{time}	The time for the conditions. This item is only included if you include a time parameter in the request.
{day}/{time}/tide	The level of tide at the beach for a specific day and time. Tide is the distance inland that the water rises to, and can be a positive or negative number. When the tide is out, the number is negative. When the tide is in, the number is positive. The 0 point reflects the line when the tide is neither going in nor out but is in transition between the two states.

RESPONSE ITEM	DESCRIPTION
{day}/{time}/wind	The wind speed at the beach, measured in knots per hour or kilometers per hour depending on the units you specify. Wind affects the surf height and general wave conditions. Wind speeds of more than 15 knots per hour make surf conditions undesirable, since the wind creates white caps and choppy waters.
{day}/{time}/watertemp	The temperature of the water, returned in Farenheit or Celsius depending upon the units you specify. Water temperatures below 70 F usually require you to wear a wetsuit. With temperatures below 60, you will need at least a 3mm wetsuit and preferably booties to stay warm.
{day}/{time}/surfheight	The height of the waves, returned in either feet or centimeters depending on the units you specify. A surf height of 3 feet is the minimum size needed for surfing. If the surf height exceeds 10 feet, it is not safe to surf.

RESPONSE ITEM	DESCRIPTION
{day}/{time}/recommendation	An overall recommendation based on a combination of the various factors (wind, watertemp, surfheight). Three responses are possible: (1) "Go surfing!", (2) "Surfing conditions are okay, not great", and (3) "Not a good day for surfing." Each of the three factors is scored with a maximum of 33.33 points, depending on the ideal for each element. The three elements are combined to form a percentage. 0% to 59% yields response 3, 60% - 80% and below yields response 2, and 81% to 100% yeilds response 3.

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

Documenting code samples

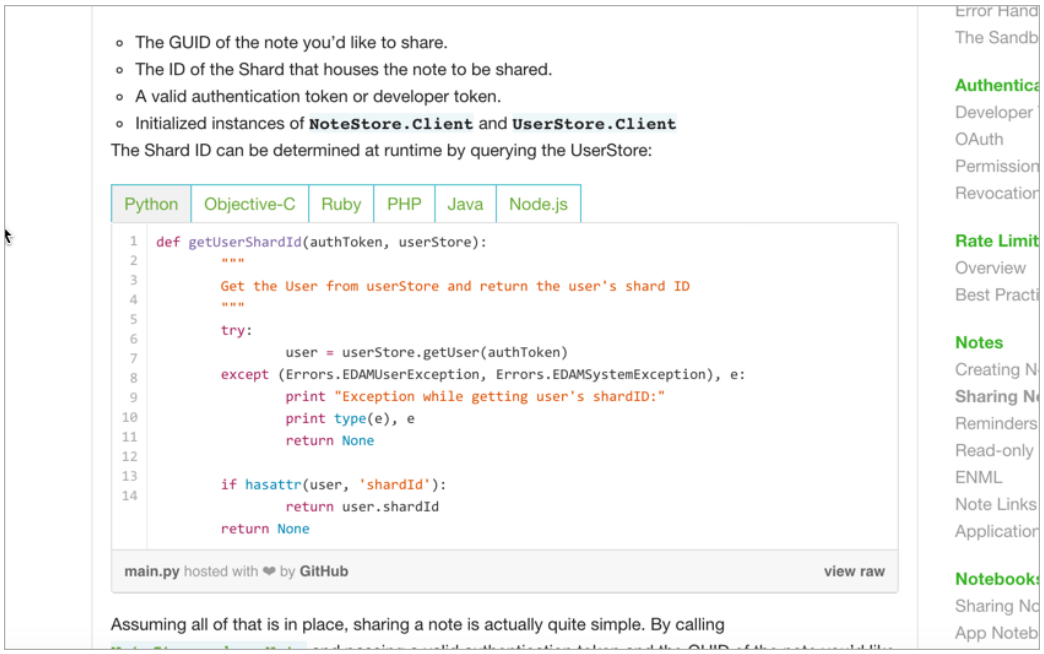
REST APIs are language agnostic

One ingenius aspect of REST APIs 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.

Which language should you provide code samples in?

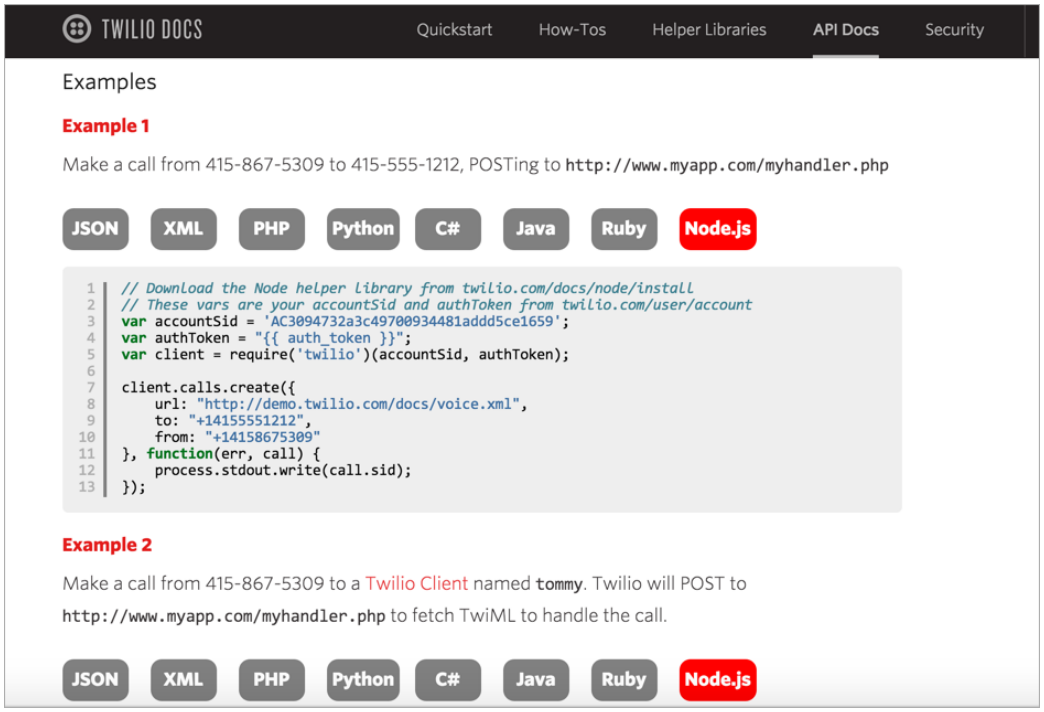
Because you can't really know which language your end users will be developing in, it's kind of fruitless to try to provide code samples in every language. Many APIs just show the format for submitting requests and a sample response, and they assume that developers will know how to submit HTTP requests 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:



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

And another from Twilio:



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

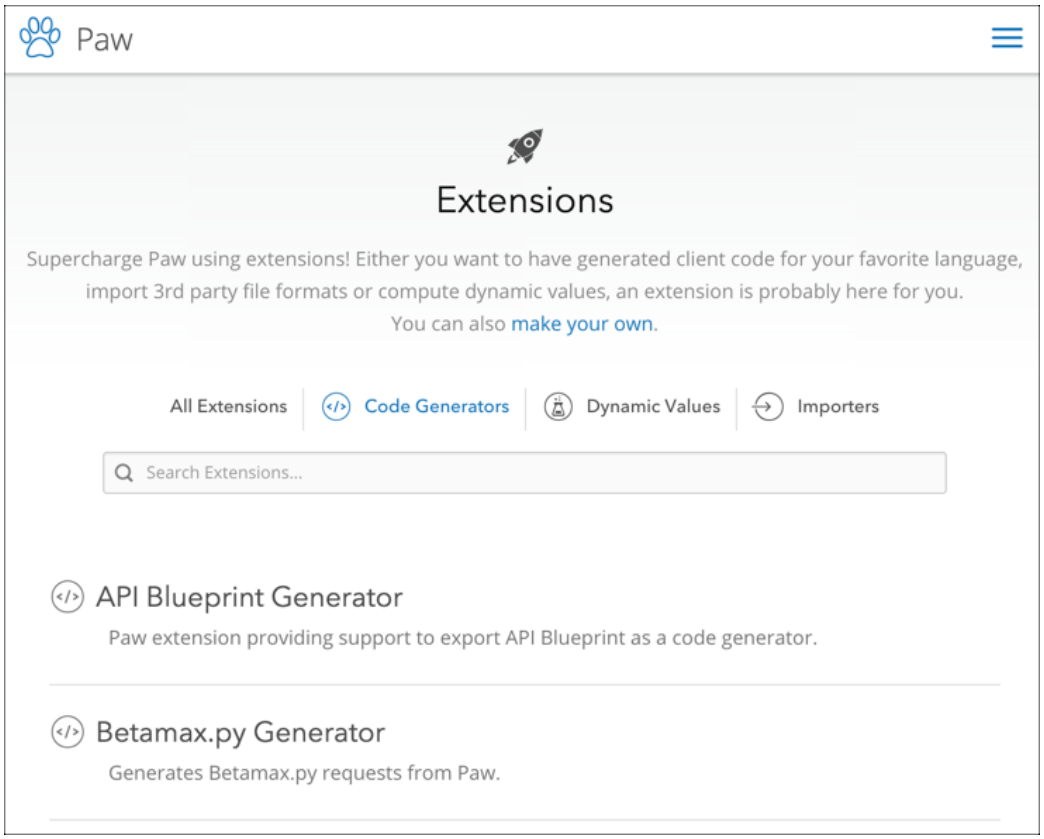
However, don't feel so intimidated by this smorgasbord of code samples. Some API doc tools might actually automatically generate these code samples because the patterns for making REST requests in different programming languages

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

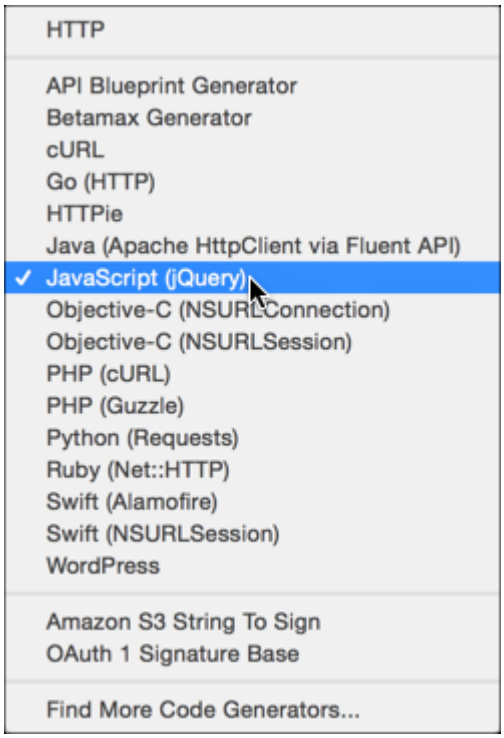
You can auto-generate code samples from both Postman and Paw, if needed.

Paw has more than a dozen code generator extensions:



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

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



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

Note: 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

Activity

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

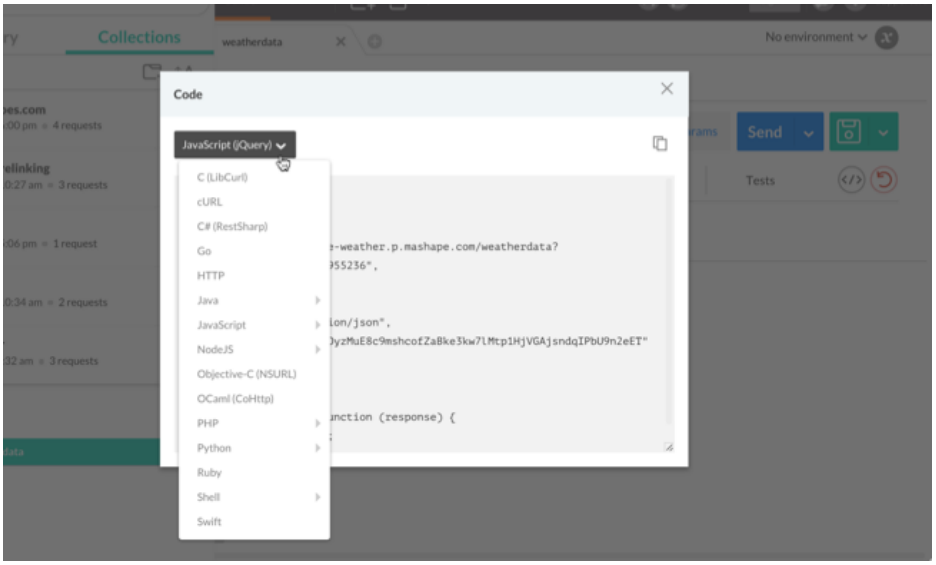
To generate a JavaScript code snippet from Postman:

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

Create a new HTML file with the basic HTML elements:

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

</body>
</html>
```

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"></scri
pt>
<script>
var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://simple-weather.p.mashape.com/weatherdata?la
t=37.354108&lng=-121.955236",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "{api key}"
  }
}

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

</body>
</html>
```

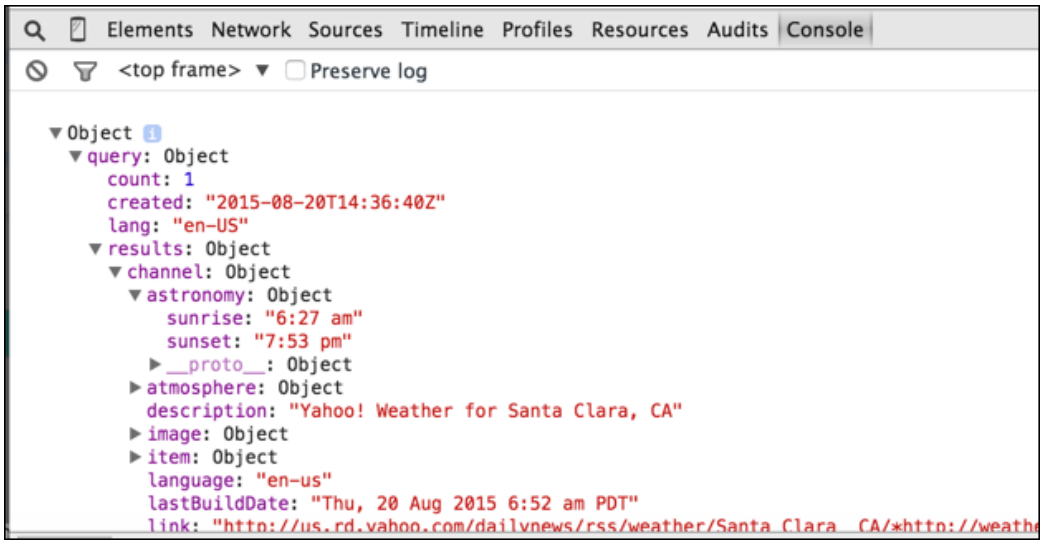
For some reason, the JavaScript code sample is missing the `dataType` parameter. Add `"dataType": "json",` in the list of settings:

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

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

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.

Open the file up in your Chrome browser.

Open the JavaScript Developer Console by going to **View > Developer > JavaScript Console**. Refresh the page. You should see the object logged to the console.



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"></scri
pt>
<script>
var settings = {
  "async": true,
  "crossDomain": true,
  "dataType": "json",
  "url": "https://simple-weather.p.mashape.com/weatherdata?la
t=37.354108&lng=-121.955236",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "{api key}"
  }
}

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

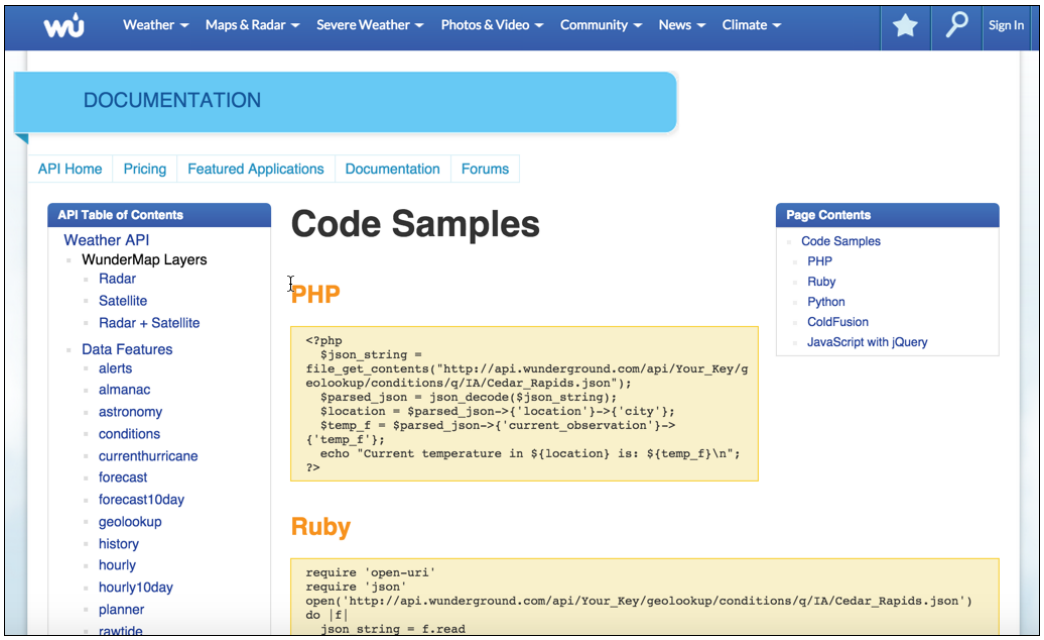
A lot of times, developers will create an SDK (software development kit) that accompanies a REST API. The SDK helps developers implement the API using specific tooling.

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 tooking 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.

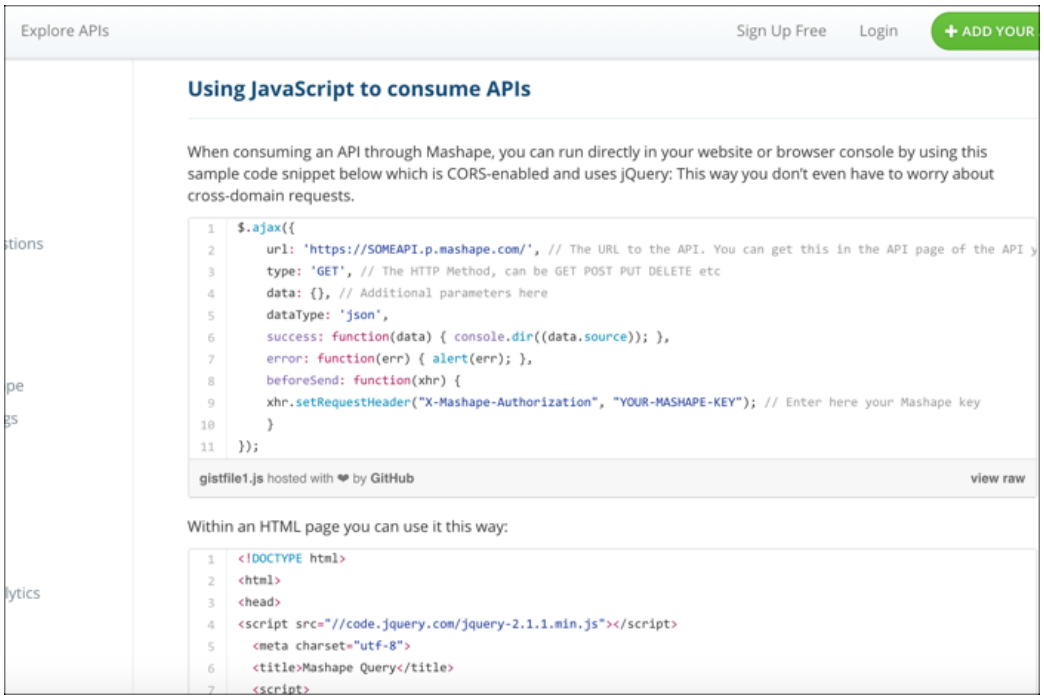
General code samples

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:



(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:



You already worked with this code earlier, so it shouldn't be new. It's most 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.

Your turn to practice: Create a code sample and documentation for surfreport

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.

```html
<!DOCTYPE html>
<head>
<script src="http://code.jquery.com/jquery-2.1.1.min.j
s"></script>
<script>
var settings = {
  "async": true,
  "crossDomain": true,
  "dataType": "json",
  "url": "https://simple-weather.p.mashape.com/surfreport/2
5",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "{api key}"
  }
}

$.ajax(settings).done(function (response) {
  console.log(response);
  $("#surfheight").append(response.query.results.channel.su
rf.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 probably wouldn't 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

Full working example

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

Sample request

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

Sample response


```
{
    "surfreport": [
        {
            "beach": "Santa Cruz",
            "monday": {
                "1pm": {
                    "tide": 5,
                    "wind": 15,
                    "watertemp": 80,
                    "surfheight": 5,
                    "recommendation": "Go surfing!"
                },
                "2pm": {
                    "tide": -1,
                    "wind": 1,
                    "watertemp": 50,
                    "surfheight": 3,
                    "recommendation": "Surfing conditions a
re okay, not great."
                },
                "3pm": {
                    "tide": -1,
                    "wind": 10,
                    "watertemp": 65,
                    "surfheight": 1,
                    "recommendation": "Not a good day for s
urfing."
                }
            }
        }
    ]
}
```


The following table describes each item in the response.

RESPONSE ITEM	DESCRIPTION
beach	The beach you selected based on the beach ID in the request. The beach name is the official name as described in the National Park Service Geodatabase.
{day}	The day of the week selected. A maximum of 3 days get returned in the response.
{time}	The time for the conditions. This item is only included if you include a time parameter in the request.
{day}/{time}/tide	The level of tide at the beach for a specific day and time. Tide is the distance inland that the water rises to, and can be a positive or negative number. When the tide is out, the number is negative. When the tide is in, the number is positive. The 0 point reflects the line when the tide is neither going in nor out but is in transition between the two states.

RESPONSE ITEM	DESCRIPTION
{day}/{time}/wind	The wind speed at the beach, measured in knots per hour or kilometers per hour depending on the units you specify. Wind affects the surf height and general wave conditions. Wind speeds of more than 15 knots per hour make surf conditions undesirable, since the wind creates white caps and choppy waters.
{day}/{time}/watertemp	The temperature of the water, returned in Farenheit or Celsius depending upon the units you specify. Water temperatures below 70 F usually require you to wear a wetsuit. With temperatures below 60, you will need at least a 3mm wetsuit and preferably booties to stay warm.
{day}/{time}/surfheight	The height of the waves, returned in either feet or centimeters depending on the units you specify. A surf height of 3 feet is the minimum size needed for surfing. If the surf height exceeds 10 feet, it is not safe to surf.

RESPONSE ITEM	DESCRIPTION
{day}/{time}/recommendation	An overall recommendation based on a combination of the various factors (wind, watertemp, surfheight). Three responses are possible: (1) "Go surfing!", (2) "Surfing conditions are okay, not great", and (3) "Not a good day for surfing." Each of the three factors is scored with a maximum of 33.33 points, depending on the ideal for each element. The three elements are combined to form a percentage. 0% to 59% yields response 3, 60% - 80% and below yields response 2, and 81% to 100% yeilds response 3.

Error and status codes

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

STATUS CODE	MEANING
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.

```html
<!DOCTYPE html>
<head>
<script src="http://code.jquery.com/jquery-2.1.1.min.j
s"></script>
<script>
var settings = {
  "async": true,
  "crossDomain": true,
  "dataType": "json",
  "url": "https://simple-weather.p.mashape.com/surfreport/2
5?days=1&units=metric",
  "method": "GET",
  "headers": {
    "accept": "application/json",
    "x-mashape-key": "{api key}"
  }
}

$.ajax(settings).done(function (response) {
  console.log(response);
  $("#surfheight").append(response.query.results.channel.su
rf.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 a template by way of a script. I'll talk more about publishing in the next course, Publishing API Documentation.

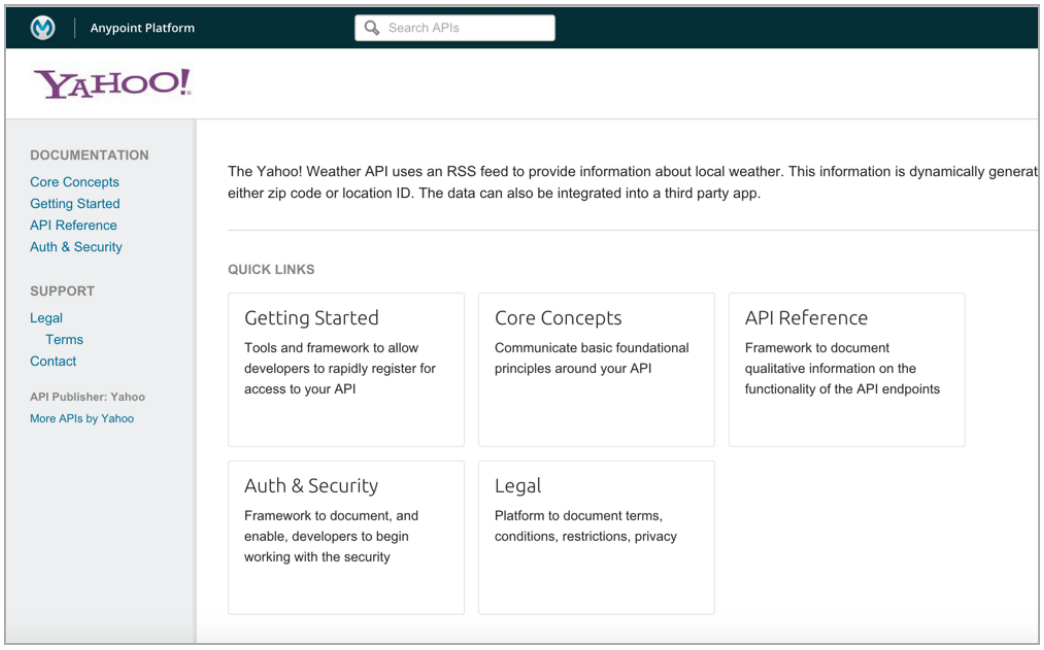
Creating the user guide

User guides versus reference documentation

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.

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 becauase it's such a simple API.)

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



(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) has a good example of these other user-guide sections essential to API documentation. It does a good show showing how API documentation is more than just a collection of endpoints.

Code tutorials

While your API shows the endpoints available, you need to explain how to use the API to accomplish real tasks your users have. A lot of times, real tasks involve using *multiple* endpoints in various dependency sequences.

Sometimes the endpoints require certain data to be set up to return the right values, and so on. These tutorials will span across the endpoints to focus on business tasks.

The usual user guide stuff

Beyond the sections outlined above, you should include the usual stuff that you put 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 provides 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

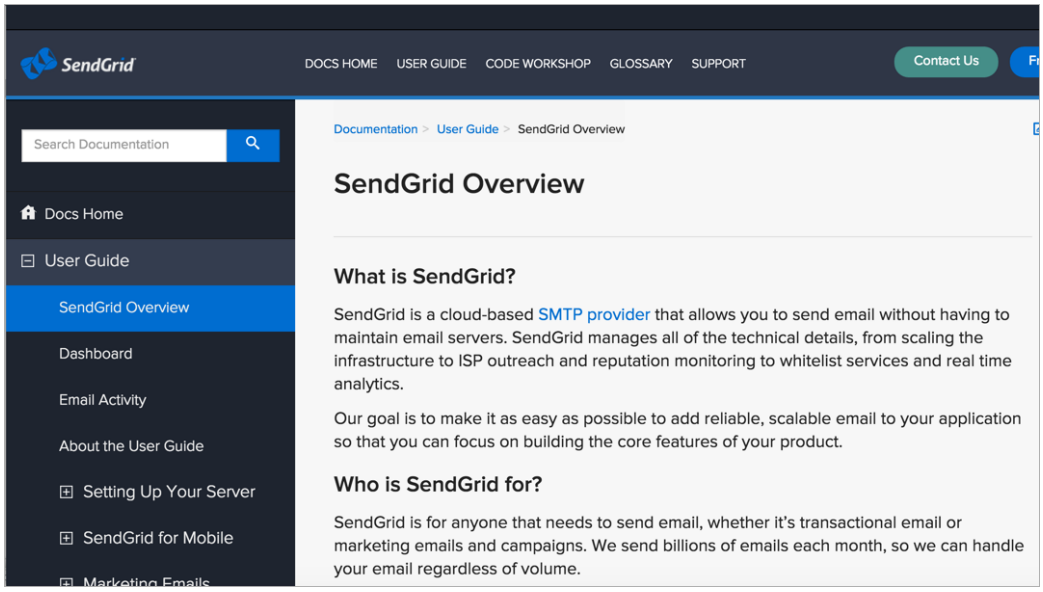
Provide how-to's for these tasks just like you would with any user guide. Seeing the tasks users can do with an API may be a little less familiar because you don't have a GUI to click through. But the basic concept is the same -- ask what will users want to do with this product, what can they do, and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

Writing the overview section

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.

Here's the Sendgrid overview:



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

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?

Your overview should probably go on the homepage of the API, or be a link from the page. 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 perceie the audience.

Writing the Getting Started section

About 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.

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/
```

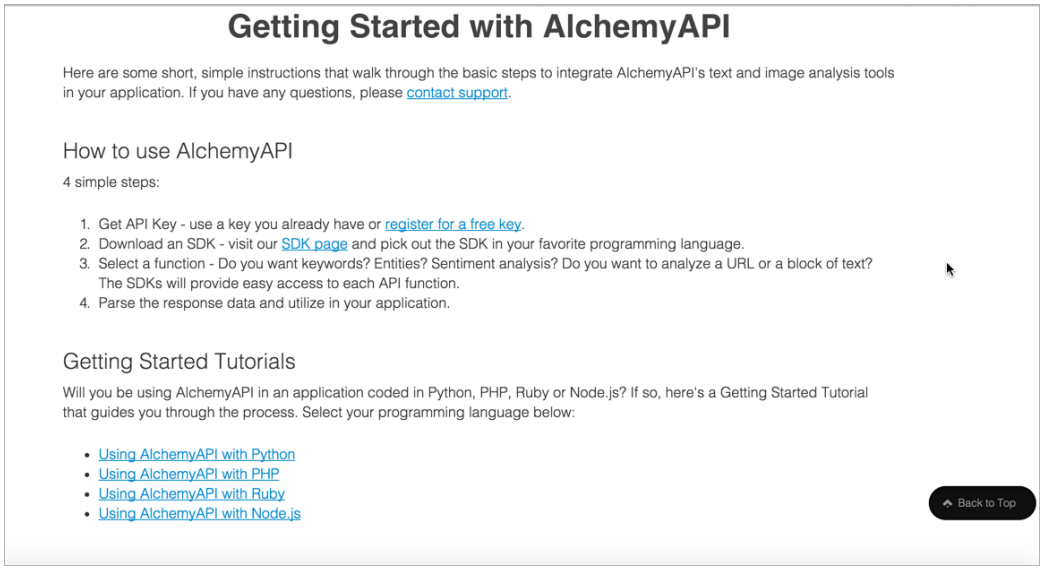
And when you combine the domain root with a sample endpoint (or resource root), it looks like this:

```
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

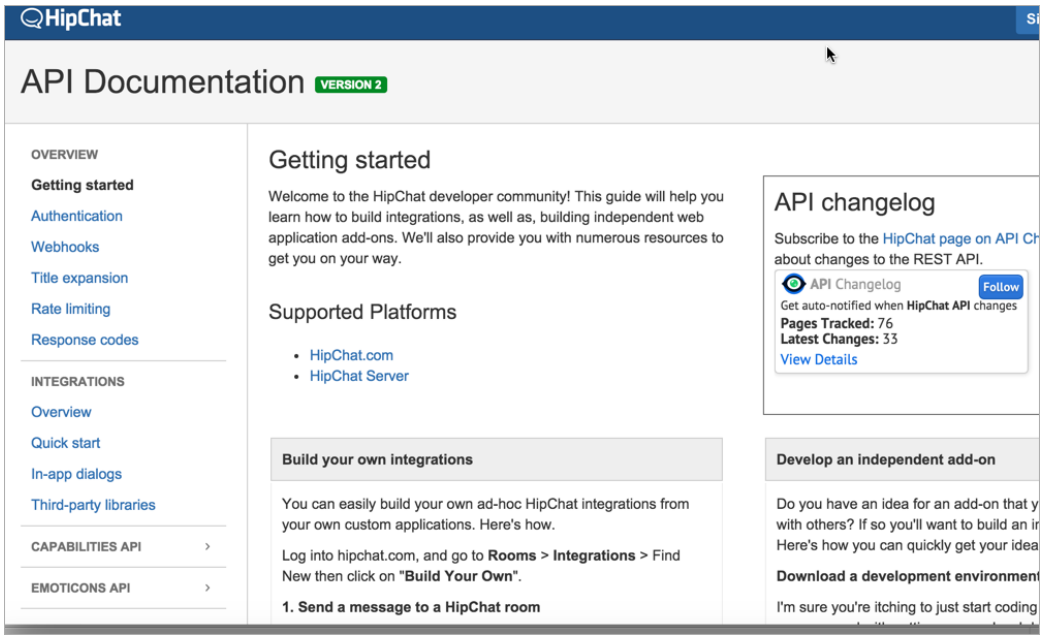
Will you be using AlchemyAPI in an application coded in Python, PHP, Ruby or Node.js? If so, here's a Getting Started Tutorial that guides you through the process. Select your programming language below:

- Using AlchemyAPI with Python
- Using AlchemyAPI with PHP
- Using AlchemyAPI with Ruby
- Using AlchemyAPI with Node.js

Back to Top

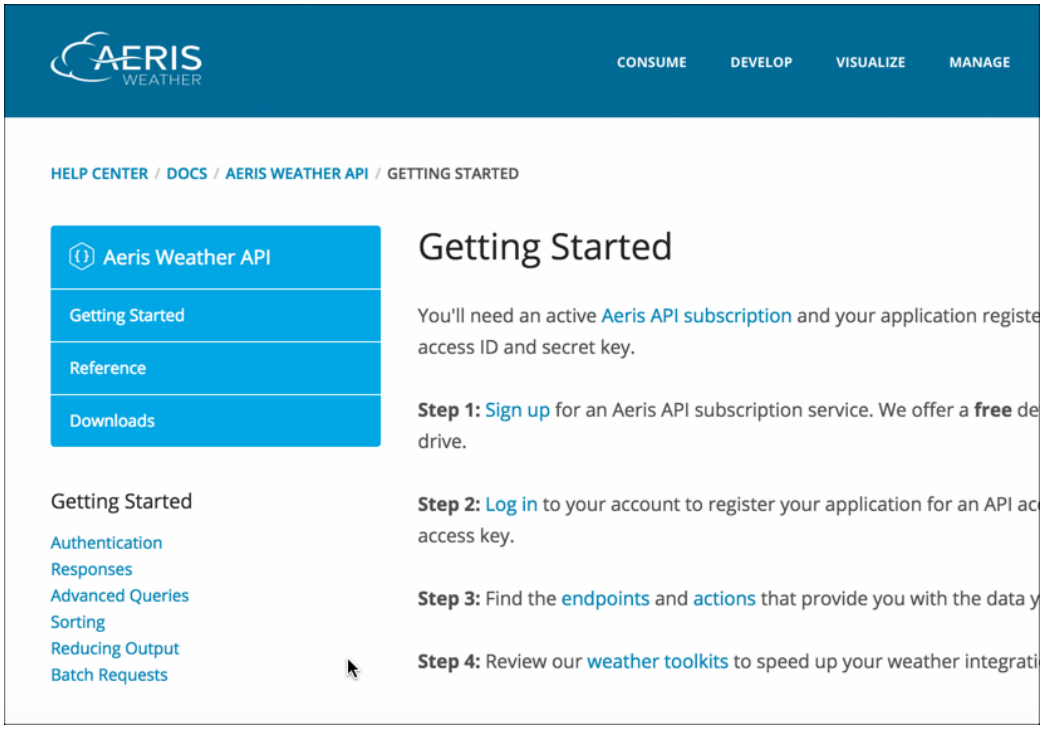
(http://www.alchemyapi.com/developers/getting-started-guide)

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



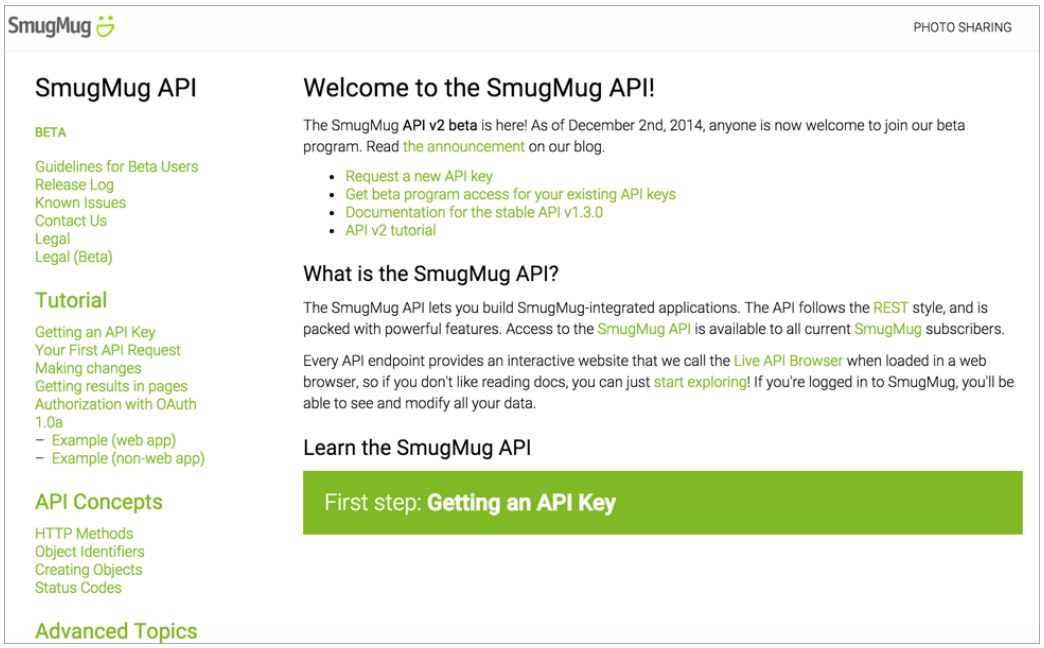
(https://www.hipchat.com/docs/apiv2)

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



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

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



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

If you compar the various Getting Started sections, you'll see that some are detailed and some rae 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 section 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.

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.

Documenting authentication and authorization

Authentication and authorization overview

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. In the sample weather API, the authentication consists of a custom header with the Mashape API key. But this API is a read-only (GET requests) API that delivers non-sensitive information (weather).

In other APIs, security might be much more important and strict due to sensitive data, the need for companies to monitor or block requests, to prove identity, to ensure the requests aren't tampered with by malicious third parties, and more.

In this section, you'll learn more about authentication and what you should focus on in documentation.

Defining authorization and authentication?

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 call.

Consequences if an API lacks authentication and authorization

There are many different ways to enforce authentication and authorization with the API calls. 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 any kind of 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 think about ways to make the API 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 URI 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 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 requesters who are exceeding 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.

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 is one of example of documentation that does this.

Basic authorization

A simple 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 receiver 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:

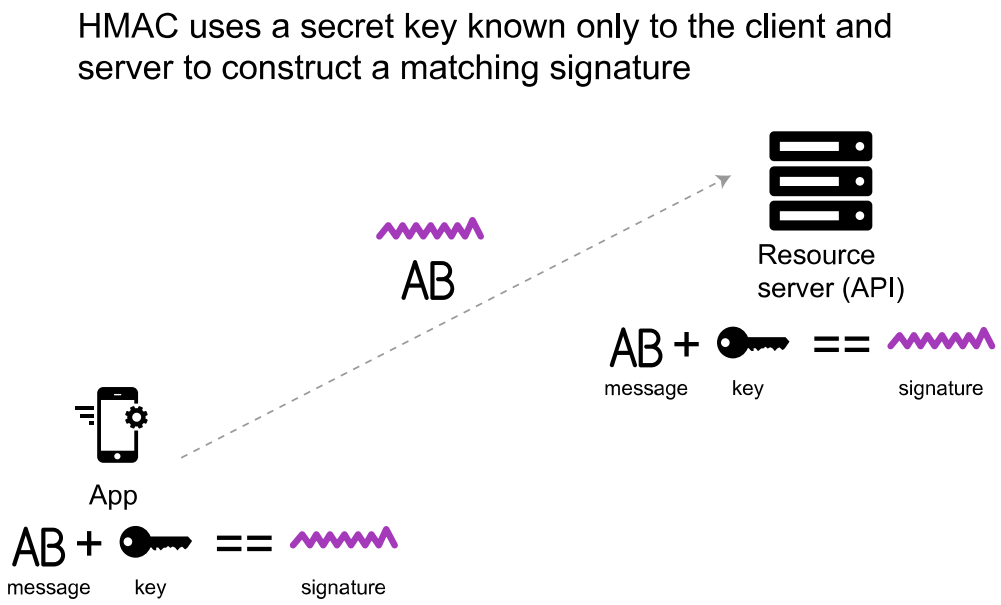
```
Authorization: Basic RnJlZDpteXBhc3N3b3Jk
```

HMAC (Hash-based message authorization code)

The HMAC approach works like this. 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 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. HMAC security is one of the stronger methods of securing API calls.

OAuth 2.0

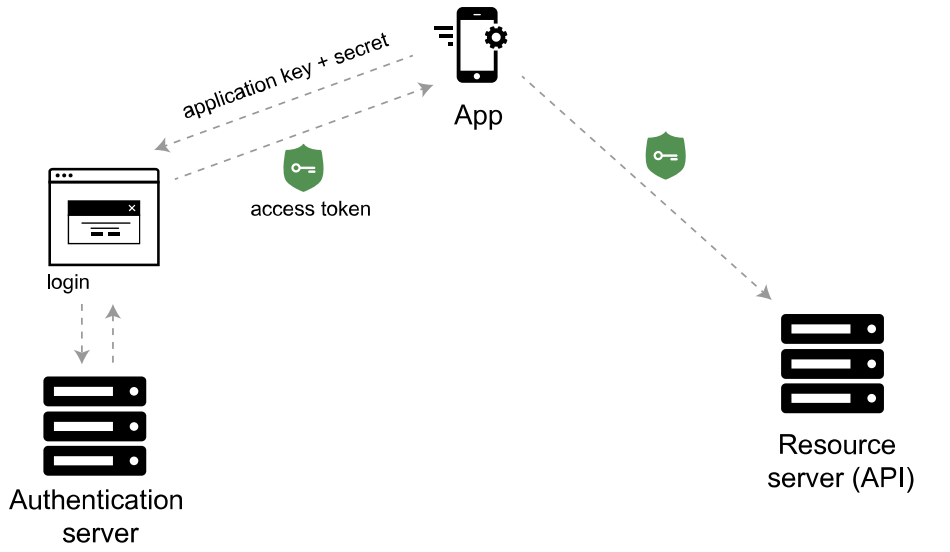
One popular method for authenticating and authorizing users is to use OAuth 2.0. This approach relies upon an authentication server to communicate with the API server in order to grant access. You often see OAuth 2.0 when you use services like Twitter, Google, or Facebook in order to log into a site.

There are a few varieties of OAuth — "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 information (such as Hubble images).

In contrast, three-legged OAuth is used when you need to protect sensitive data. There are three groups interacting in this scenario:

- Authentication server
- Resource server (API server)
- User

Here's the basic workflow of OAuth:



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 user then makes a request to the API server (known as the "resource 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/)

- 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 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:

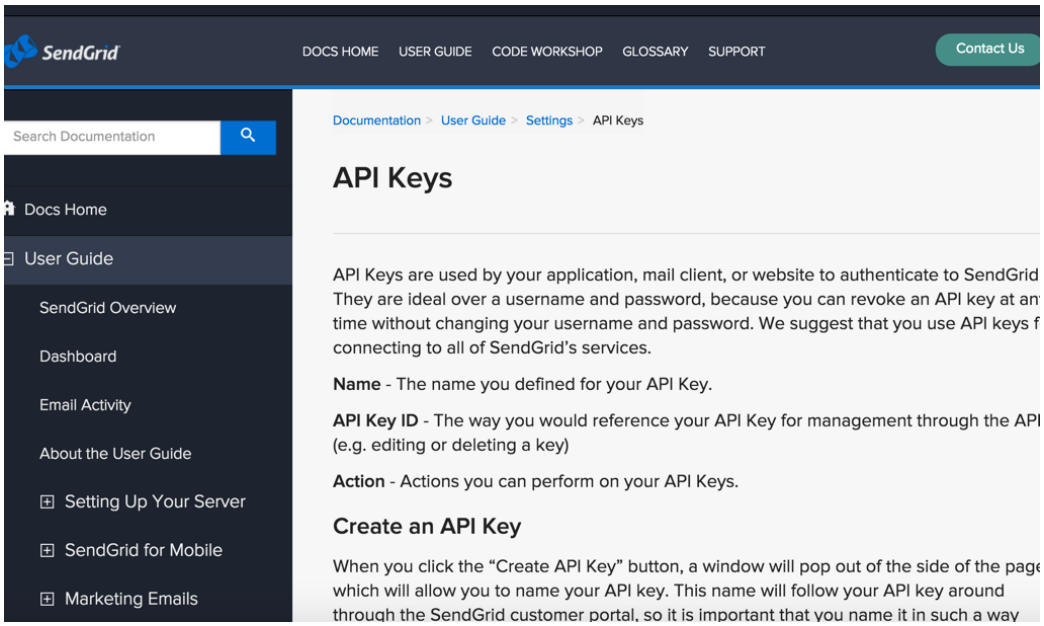
- How to get API keys
- How to authenticate requests
- Error messages related to invalid authentication
- Rate limits with API requests
- Potential costs surrounding API request usage
- Token expiration times

If you have public and private keys, you should explain where each key should be used, and that private keys should not be shared.

API keys section of 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:



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

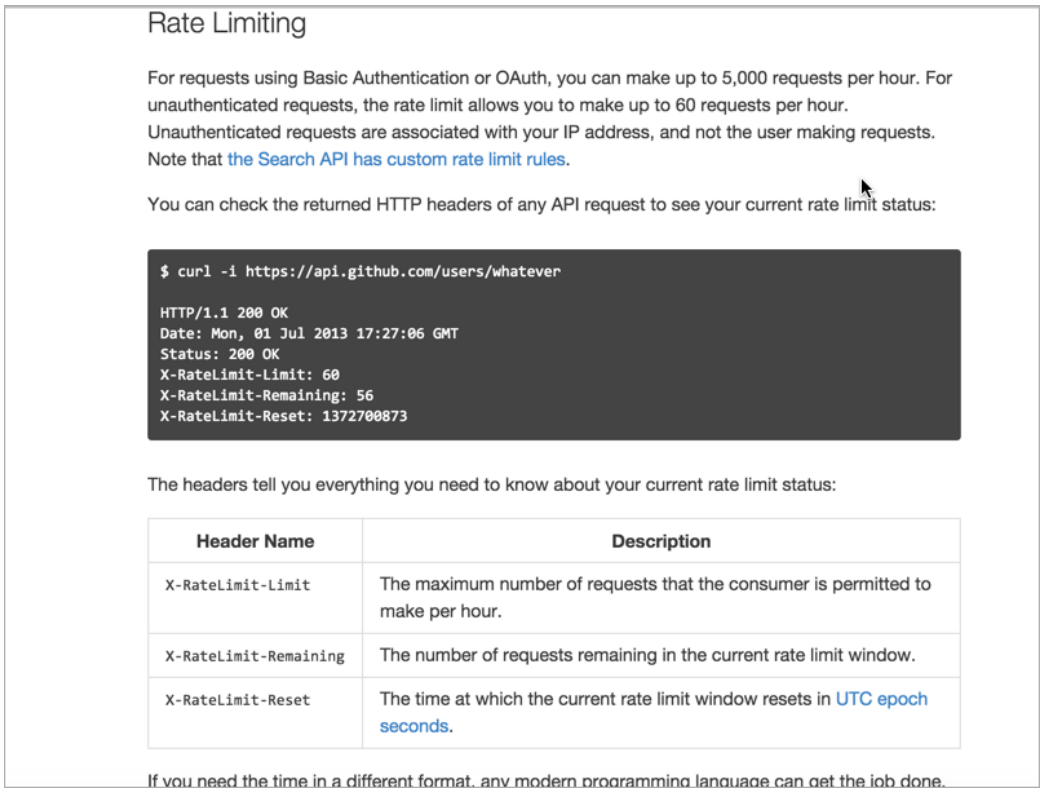
If different license tiers provide different access to the API calls you can make, these licensing tiers should be explicit in your authorization section or elsewhere.

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 lmits.

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)

Documenting response and error codes

Response codes let you know the status of the request

Remember when we submitted the cURL call back in an earlier lesson (page 0)? 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/aq
i?lat=37.354108&lng=-121.955236' \-H 'X-Mashape-Key: {api ke
y}' \
  -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 make 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.

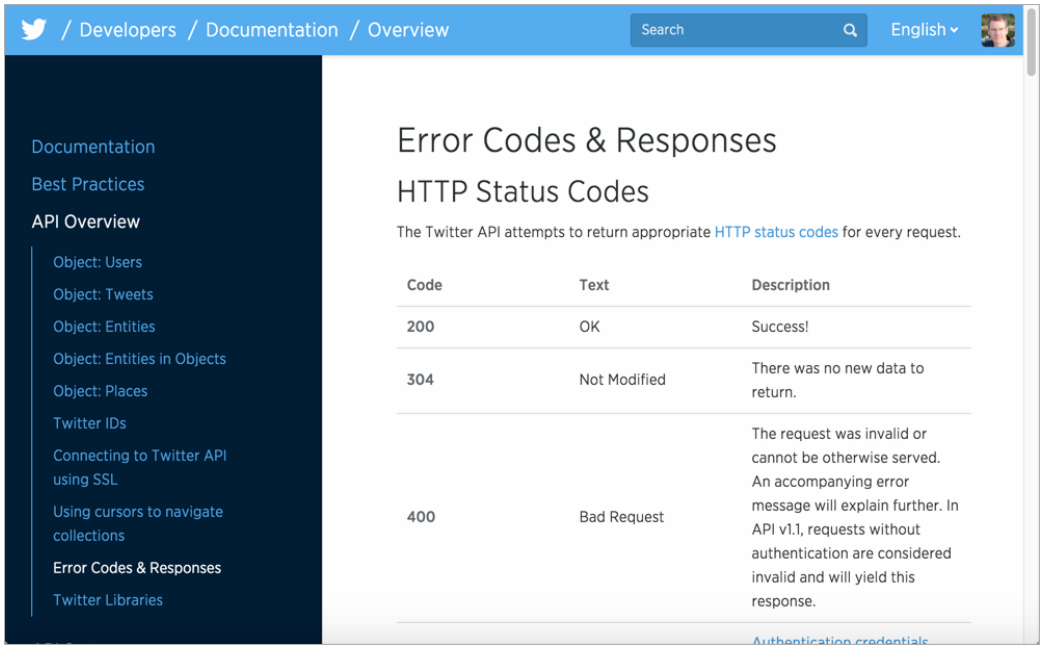
Common status codes follow standard protocols

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 univerally 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).

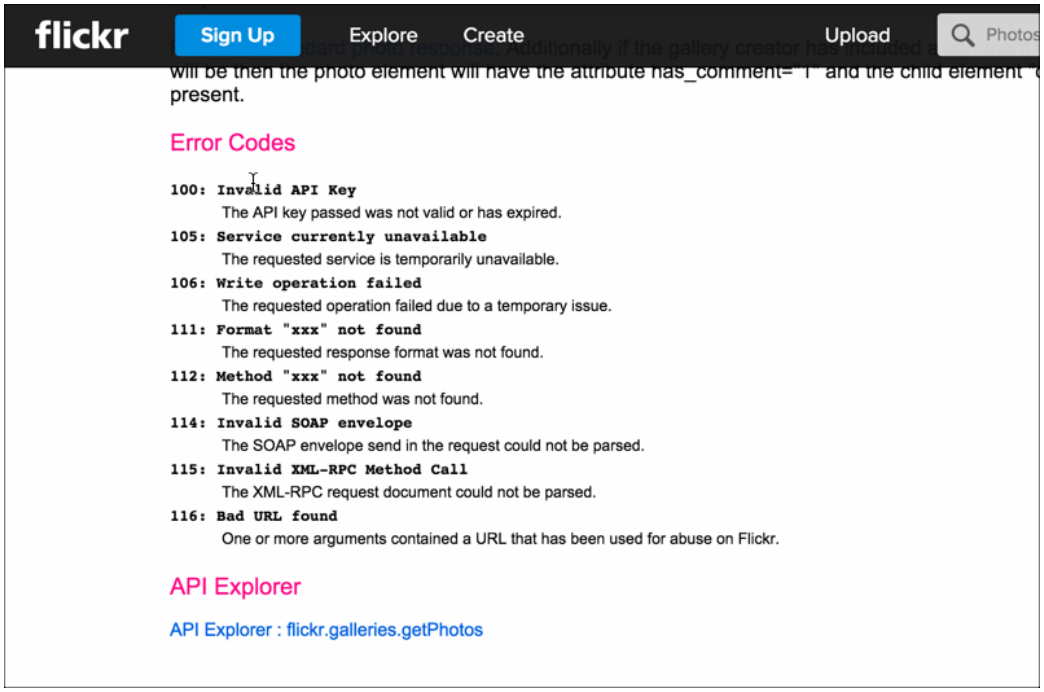
Where to list the HTTP response and error codes

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:



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

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



(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.

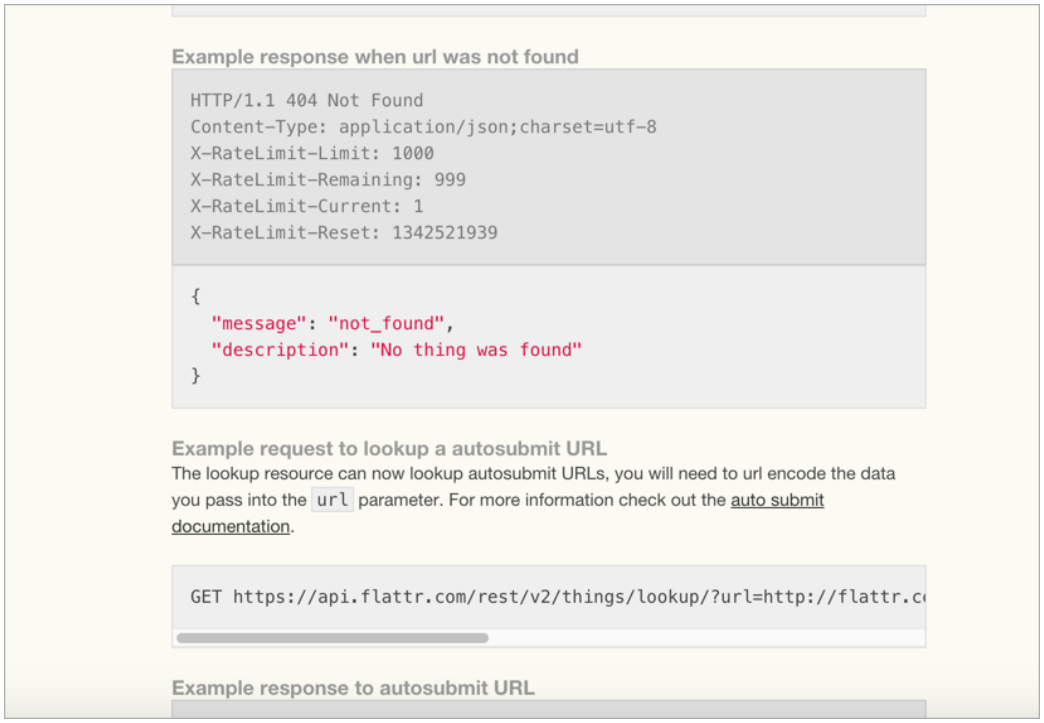
Where to get error codes

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 all, the API might return a special status code. You would especially need to document this custom code. Listing out all the error codes is an reference in the "Troubleshooting" section of your API documentation.

When endpoints have specific status codes

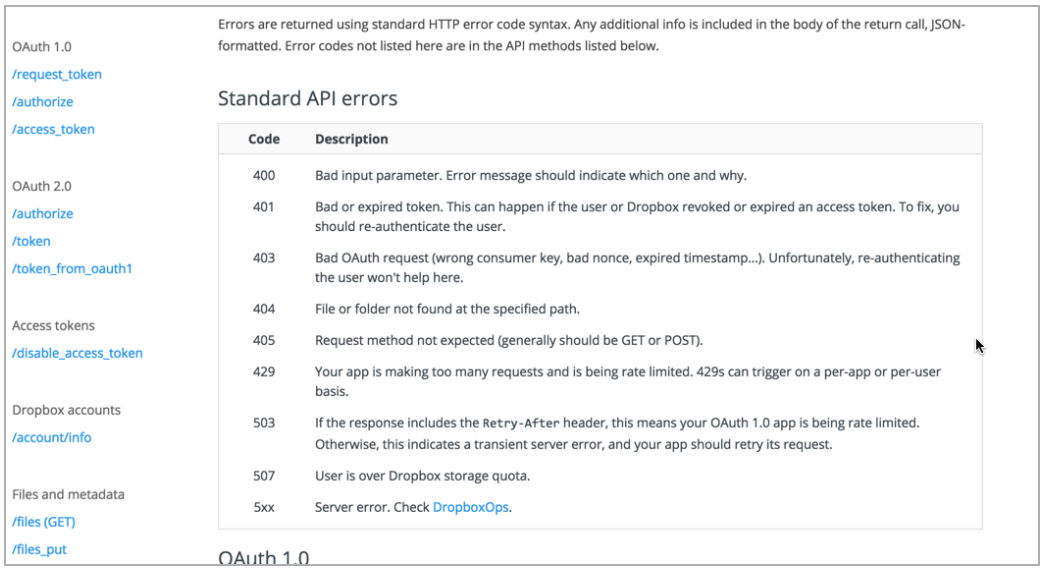
In the Flattr API, sometimes endpoints return particular status codes. For example, when you "Check if a thing exists," the response includes `HTTP/1.1 302 Found` when the object is found. This is a standard HTTP response. If it's not found, you see a 404 status code.



(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:



(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.

> ☑ **Tip:** 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.

Sample status code list

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 are subtle

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 remove the error

Error codes and 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

About code samples

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

The following sections list some best practices around code samples.

Code samples are like candy for developers

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

You are not the audience

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

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

It might be important to remind developers that users often have technical talent in different areas. For example, a user might be an expert in Java but only mildly familiar with JavaScript.

Focus on the why, not the what

In any code sample, you should focus your explanation on the *why*, not the *what*. Explain why you're doing what you're doing, not the detailed play-by-play of what's going on.

Here's an example of the difference:

- **what**: In this code, several arguments are passed to jQuery's `ajax` method. The response is assigned to the data argument of the callback function, which in this case is `success`.

- **why**: Use the `ajax` method from jQuery because it allows cross-origin resource sharing (CORS) for the weather resources. In the request, you submit the authorization through the header rather than including the API key directly in the endpoint path.

Focus your explanation on your company's code only

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-pastable

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 looking 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.

From code samples to real tasks in user guides

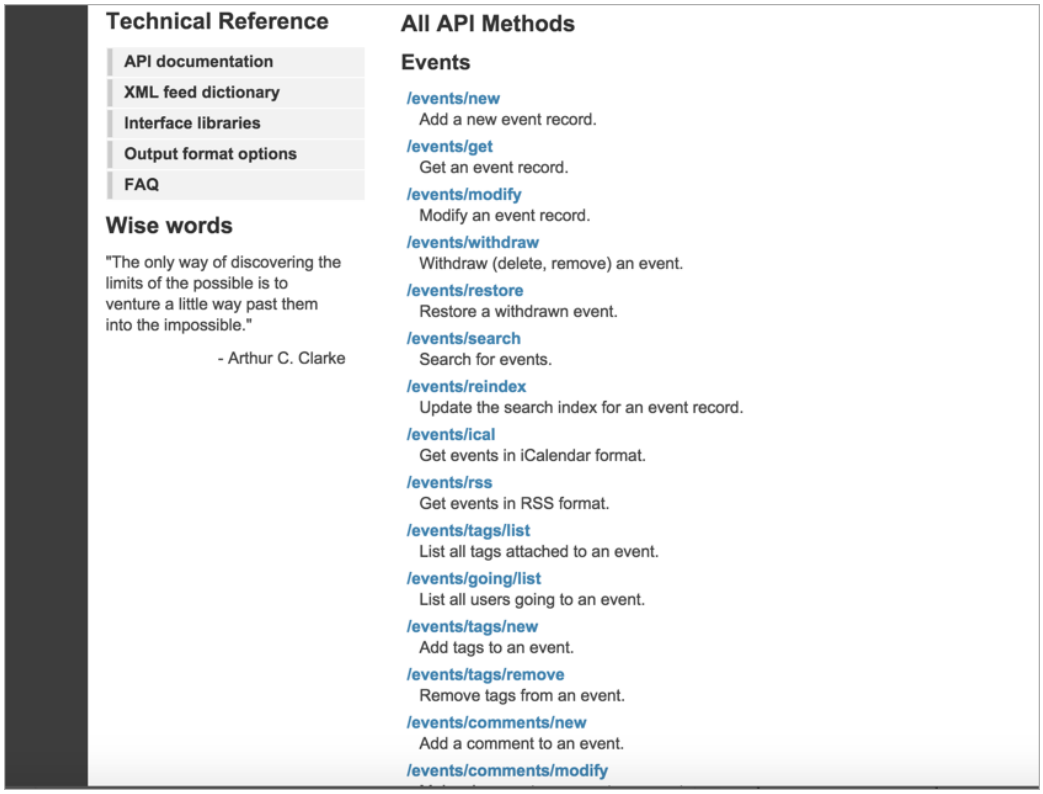
Getting into code samples leads us more toward user guide tasks than reference tasks. However, keep in mind that code samples are a bear to maintain. When your API pushes out a new release, will you check all the code samples to make sure the code doesn't break with the new API (this is called regression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

Quick reference guide

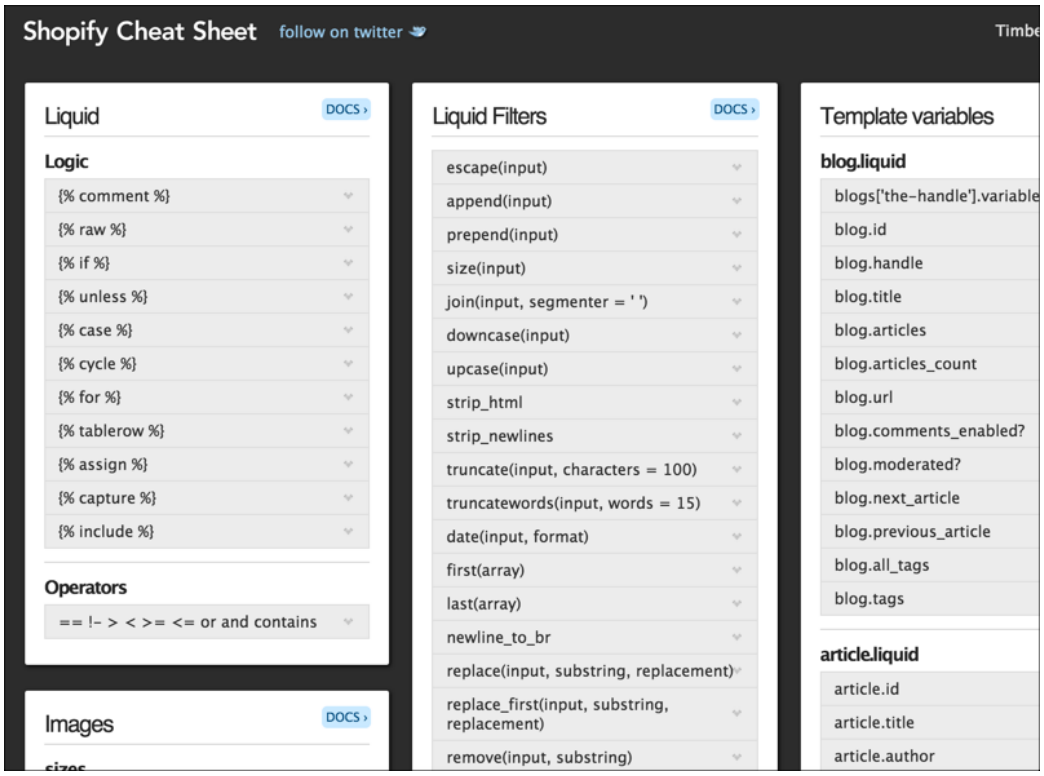
For those power users who just want to glance at the content to understand it, provide a quick reference guide. For example, here's a quick reference guide from Eventful's API:



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

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.

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:



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

You can also make a visual illustration showing the API endpoints and how they relate to one another. I once created this one page endpoint diagram at Badgeville, and 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 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 interdependencies of endpoints, and more.

EventBrite example: Get Event information and display it on a page

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

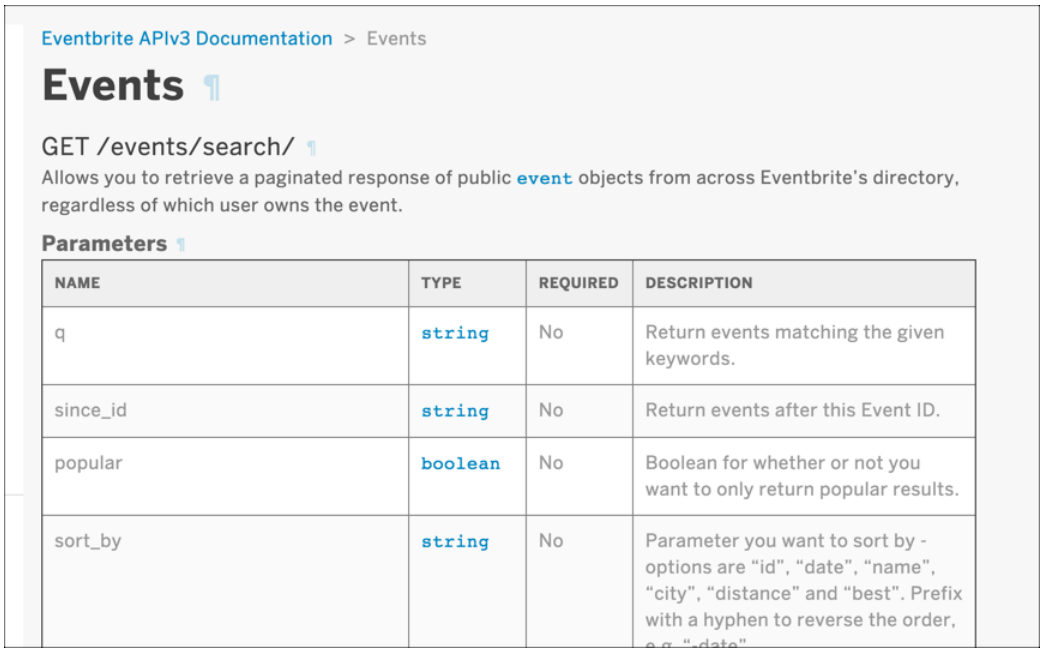
To make any kind of requests, you'll need a tokaen, which you can learn about in the Authentication section (https://www.eventbrite.com/developer/v3/reference/authentication/). Although it's best to pass an Oauth token in the header, for simplicity purposes you can just get a token to make direct calls.

If you want to sign up for your own token, register your app here (https://www.eventbrite.com/myaccount/apps/). 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/) 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,

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

> **Note:** 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) seems to provide what we need.

> **Note:** The Eventbrite docs convention is to use `:id` instead of `{id}` to represent values you pass into the endpoint.

3. Construct the request

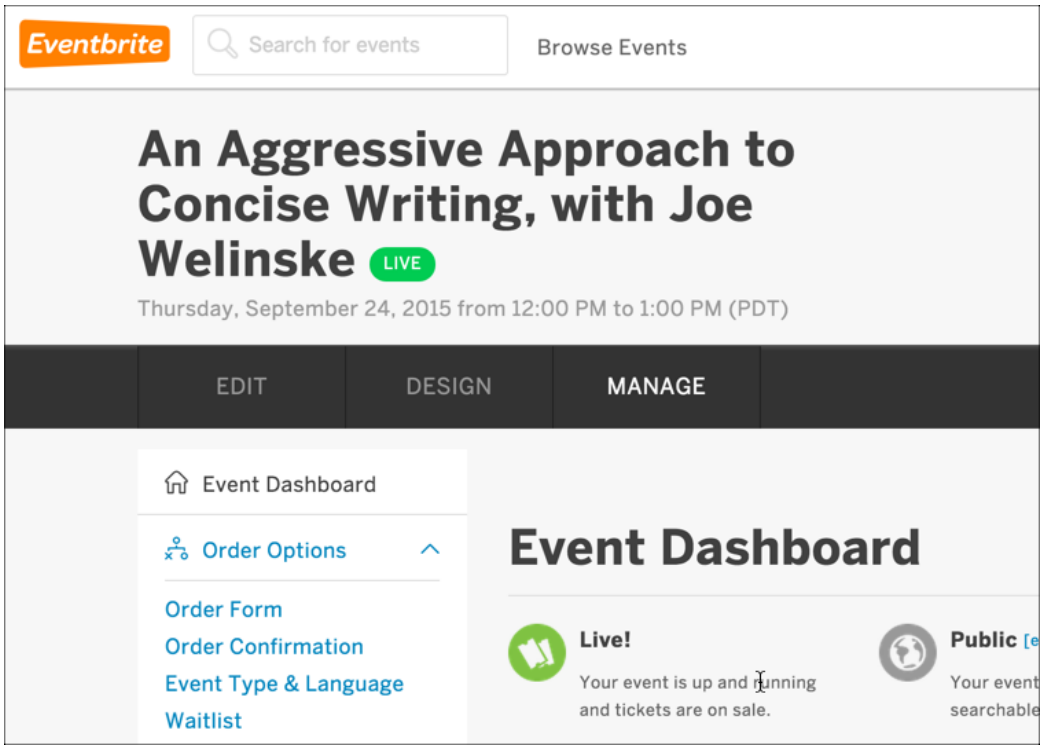
Reading the quick start page, the sample request format is here:

```
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:



(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, wit
h Joe Welinske",
        "html": "An Aggressive Approach to Concise Writing, wit
h Joe Welinske"
    },
    "description": {
        "text": "Webinar Description \nWriting concisely is on
e of the fundamental skills central to any mobile user assistan
ce. The minimal screen real estate can\u2019t support large amo
unts of text and graphics without extensive gesturing by the us
ers. Using small font sizes just makes the information unreadab
le unless the user pinches and stretches the text.   Even outsi
de of the mobile space, your ability to streamline your conten
t improves the likelihood it will be effectively consumed by yo
ur target audience.   This session offers a number of examples
and techniques for reducing the footprint of your prose while m
aintaining a quality message. The examples used are in the cont
ext of mobile UA but can be applied to any technical writing si
tuation. \nAbout Joe Welinske Joe Welinske specializes in helpi
ng your software development effort through crafted communicati
on. The best user experience features quality words and images
in the user interface. The UX of a robust product is also enhan
ced through comprehensive user assistance. This includes Help,
wizards, FAQs, videos and much more. For over twenty-five year
s, Joe has been providing training, contracting, and consultin
g services for the software industry. Joe recently published th
e book, Developing User Assistance for Mobile Apps. He also tea
ches courses for Bellevue College, the University of Californi
a, and the University of Washington. Joe is an Associate Fello
w of STC. ",
        "html": "<P><SPAN STYLE=\"font-size: medium;\"><STRON
G>Webinar Description<\/STRONG><\/SPAN><\/P>\r\n<P>Writing conc
isely is one of the fundamental skills central to any mobile us
er assistance. The minimal screen real estate can\u2019t suppor
t large amounts of text and graphics without extensive gesturin
g by the users. Using small font sizes just makes the informati
on unreadable unless the user pinches and stretches the text.<B
R> <BR>Even outside of the mobile space, your ability to stream
line your content improves the likelihood it will be effectivel
y consumed by your target audience.<BR> <BR>This session offer
s a number of examples and techniques for reducing the footprin
t of your prose while maintaining a quality message. The exampl
es used are in the context of mobile UA but can be applied to a
ny technical writing situation.<\/P>\r\n<P><SPAN STYLE=\"font-s
ize: medium;\"><STRONG>About Joe Welinske<\/STRONG><\/SPAN><B
R>Joe Welinske specializes in helping your software developmen
```

```
t effort through crafted communication. The best user experienc
e features quality words and images in the user interface. The
UX of a robust product is also enhanced through comprehensive u
ser assistance. This includes Help, wizards, FAQs, videos and m
uch more. For over twenty-five years, Joe has been providing tr
aining, contracting, and consulting services for the software i
ndustry. Joe recently published the book, Developing User Assis
tance for Mobile Apps. He also teaches courses for Bellevue Col
lege, the University of California, and the University of Washi
ngton. Joe is an Associate Fellow of STC.<\/P>"
    },
    "id": "17920884849",
    "url": "http://www.eventbrite.com/e/an-aggressive-approac
h-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/17
920884849/",
    "logo": null
}
```

5. Pull out the information you need

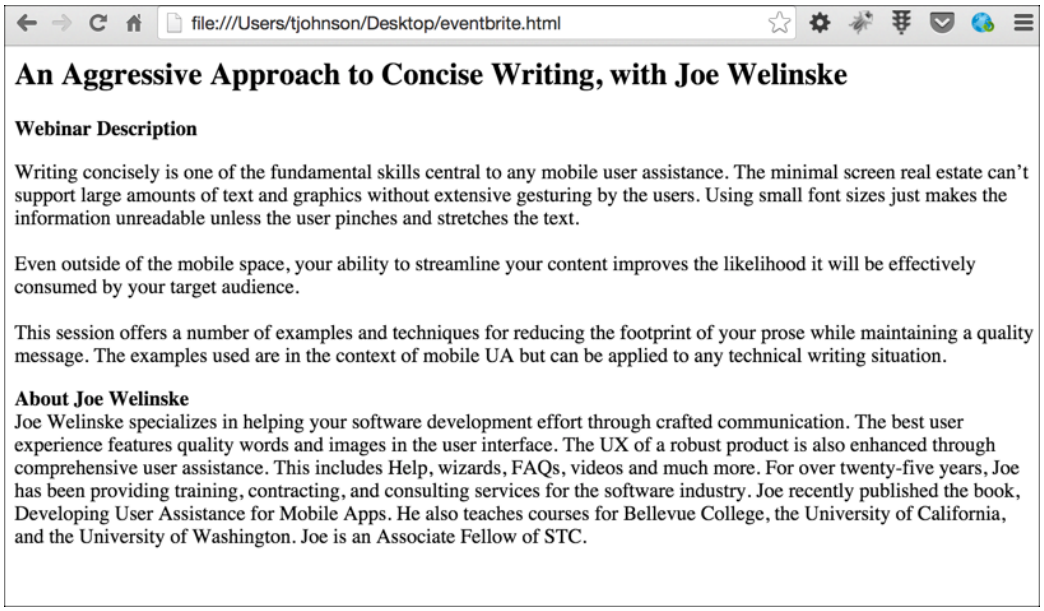
The information has a lot more than we need. We just want to display the event's title and description on our site. To do this, we use some simple jQuery code to pull out the information and append it to a tag on our web page:

```
<html>
<body>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.1
1.1/jquery.min.js"></script>

<script>
  var settings = {
    "async": true,
    "crossDomain": true,
    "url": "https://www.eventbriteapi.com/v3/events/1792088484
9/?token={api key}",
    "method": "GET",
    "headers": {}
  }

  $.ajax(settings).done(function (data) {
    console.log(data);
    var content = "<h2>" + data.name.text + "</h2>" + data.desc
ription.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.

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

Here's the result:



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.

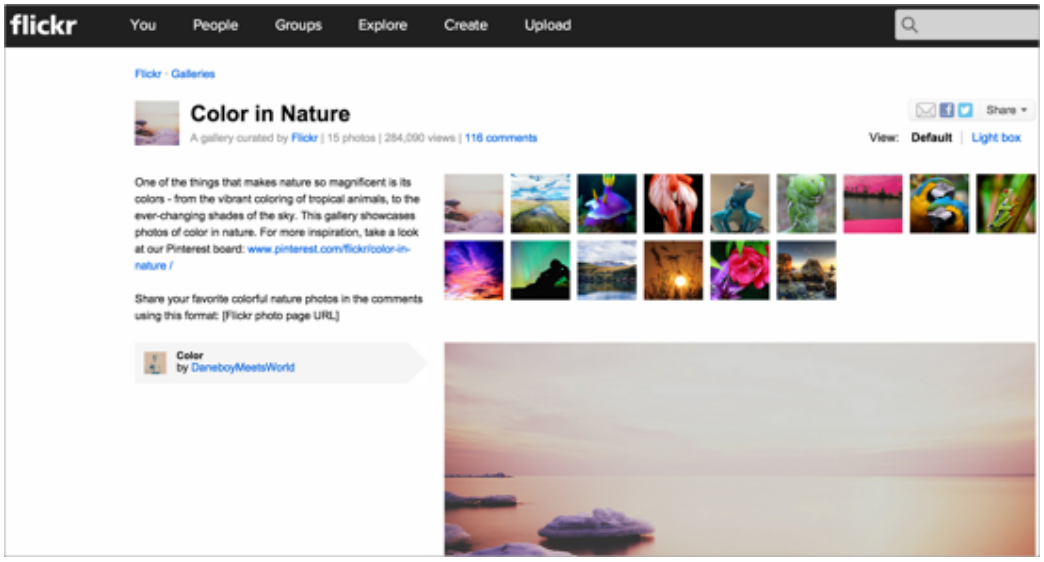
We then rename the content we want with a variable (`var content`) and use jQuery's `append` method to assign it to a specific tag (`eventbrite`) on the page.

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

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:



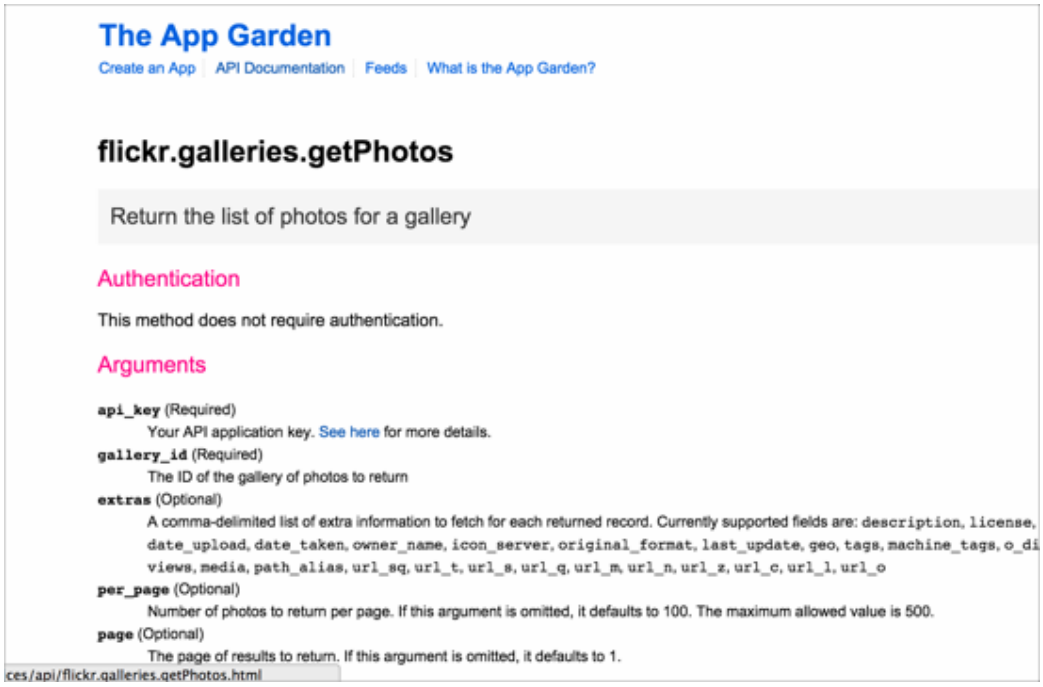
(https://www.flickr.com/photos/flickr/galleries/72157647277042064/)

1. Get an API key to make requests

Before you can make request with the Flickr API, you'll need a API key, which you can read more about here (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

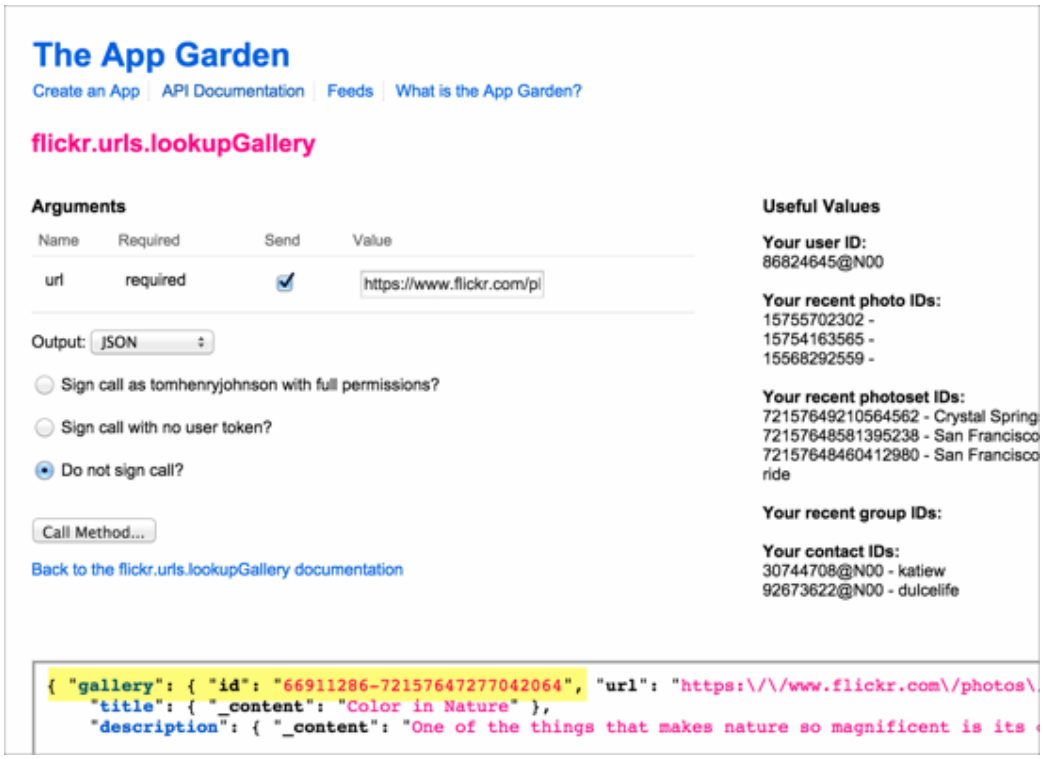
From the list of Flickr's API methods (https://www.flickr.com/services/api/), the flickr.galleries.getPhotos (https://www.flickr.com/services/api/flickr.galleries.getPhotos.html) endpoint, which is listed under the galleries resource, is the one that will get photos from a gallery.



(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) endpoint listed in the URLs resource section to get the gallery ID from a gallery URL:



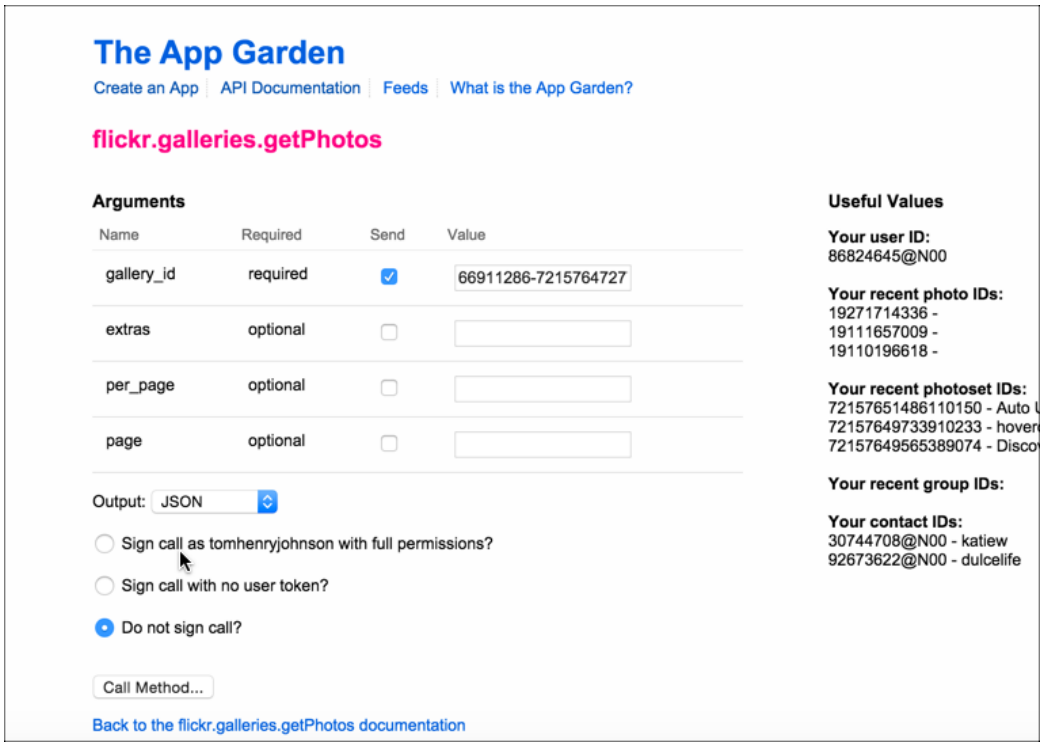
(https://www.flickr.com/services/api/explore/flickr.urls.lookupGallery)

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.

2. 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.



(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:

The URL below the response shows the right syntax for using this method:

```
https://api.flickr.com/services/rest/?method=flickr.galleries.getPhotos&api_key={api key}&gallery_id=66911286-72157647277042064&format=json&nojsoncallback=1
```

Note: 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:

Tip: I'm using the JSON Formatting extension for Chrome (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) 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|p
> ```

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.

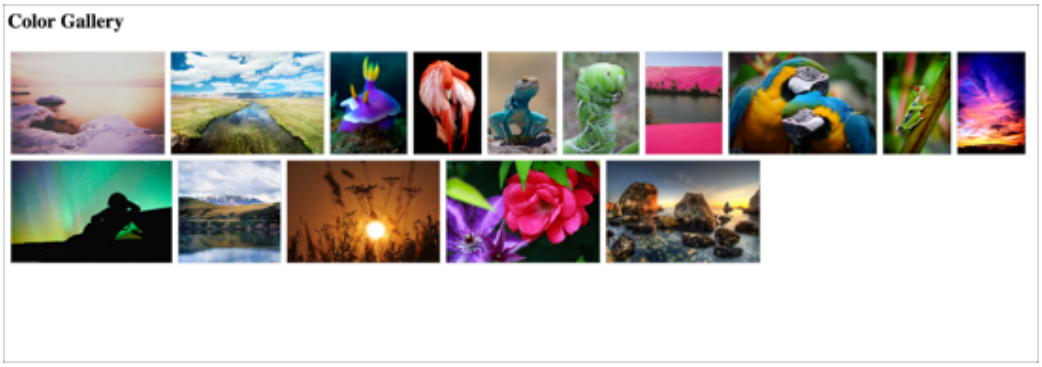
5. Pull out the information you need

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/1.1
1.1/jquery.min.js"></script>


<script>

var settings = {
  "async": true,
  "crossDomain": true,
  "url": "https://api.flickr.com/services/rest/?method=flickr.g
alleries.getPhotos&api_key={api key}&gallery_id=66911286-721576
47277042064&format=json&nojsoncallback=1",
  "method": "GET",
  "headers": {}
}

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



$("#galleryTitle").append(data.photos.photo[0].title + " Galler
y");
        $.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 + ", " + secre
t);

//  https://farm{farm-id}.staticflickr.com/{server-id}/{id}_{se
cret}.jpg

$("#flickr").append('<img src="https://farm' + farmId + '.stati
cflickr.com/' + serverId + '/' + id + '_' + secret + '.jp
g"/>');

});
});
```

```
</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: 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/) 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/) 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 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('<img src="https://farm' + farmId + '.stati
cflickr.com/' + serverId + '/' + id + '_' + secret + '.jp
g"/>');
```

☑ **Tip:** 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.

Documenting REST APIs

Newsletter signup

Get new posts delivered straight to your inbox.

Subscriber count: 2,476

Sponsors

Text links

- Wordpresswriter.com (http://www.wordpresswriter.com/)
- College writing tips (http://www.campusgrumbleconnect.com/)
- Onecom (http://one.dealspotr.com/onecom)
- Host Papa (http://hostpapa.dealspotr.com/hostpapa)
- Just Host (http://justhost.dealspotr.com/justhost)
- Communications-Major.com (http://www.communications-major.com/)

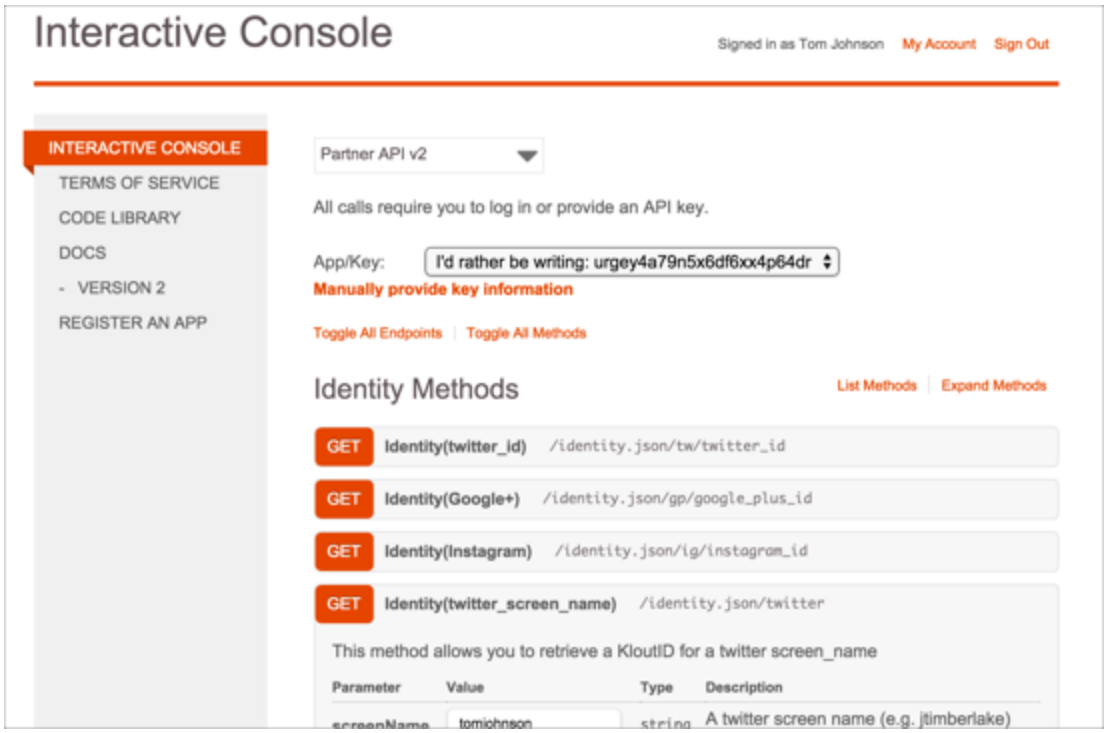
Recent Comments

Klout example: Retrieve Klout score

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.



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

1. Get an API key to make requests

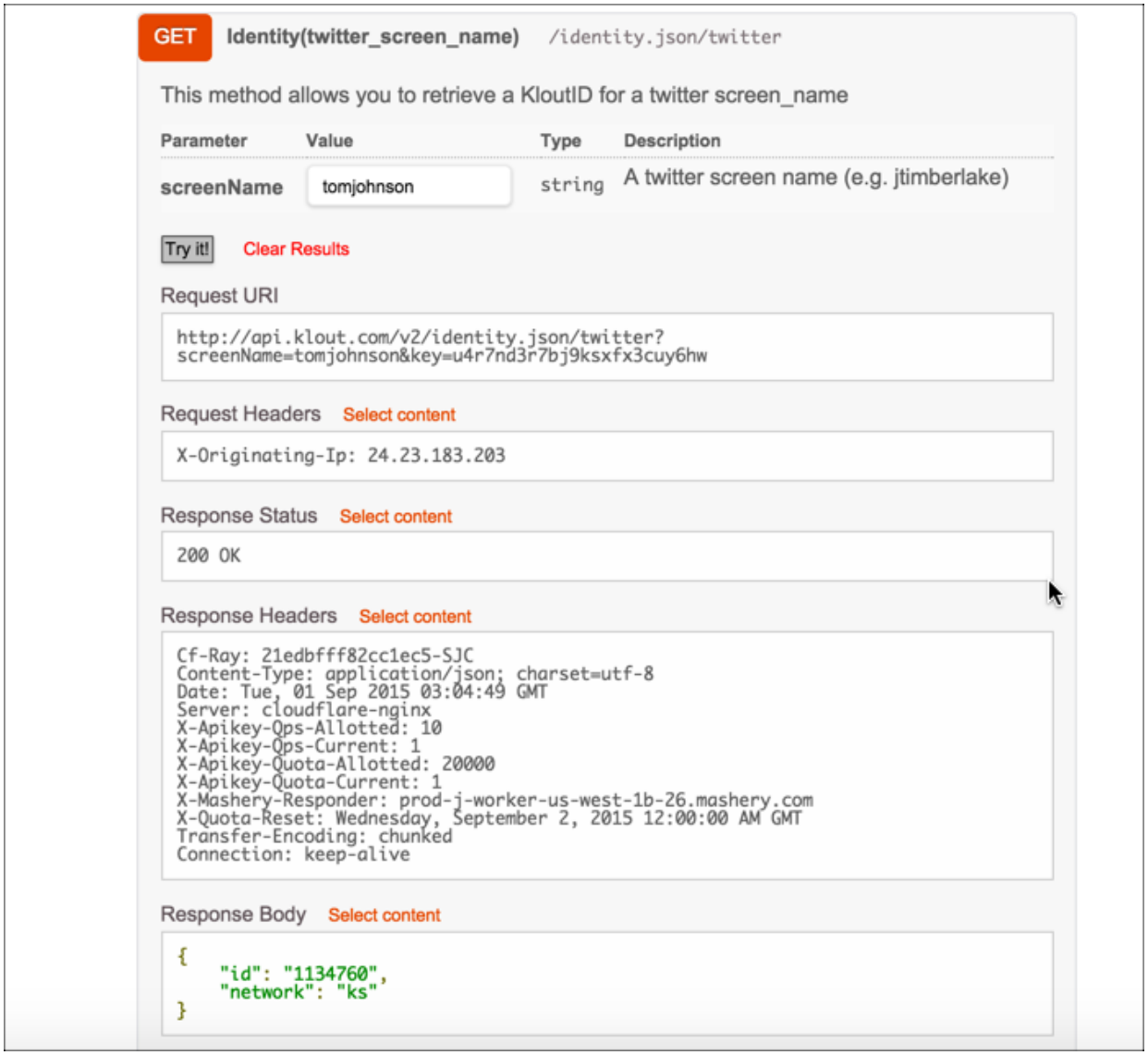
To use the API, you have to to register an "app," which allows you to get an API key. Go to My API Keys (http://developer.klout.com/apps/mykeys) page to register your app and get the keys.

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.



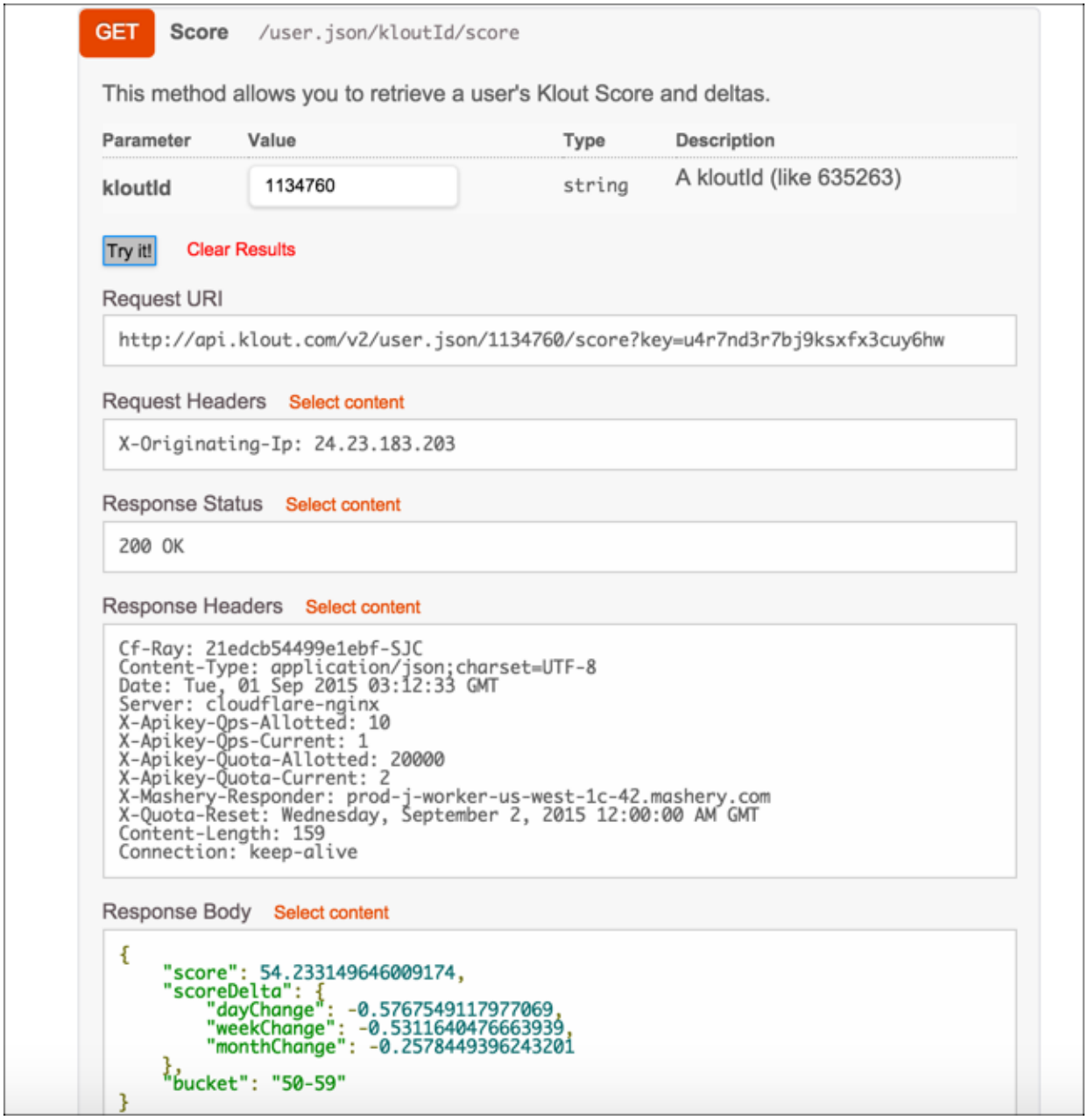
Instead of using the API console, you can also submit the request via your browser by going to the request URL:

```
http://api.klout.com/v2/identity.json/twitter?screenName=tomjohnson&amp;key={your api key}
```

Note: 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.



My score is `54` . Klout's interactive console makes it easy to get responses for API calls, but you could equally submit the request URI in your browser.

```
http://api.klout.com/v2/user.json/1134760/score?key={your api key}
```

After submitting the request, here is what you would see:

```
{
   "score": 54.233149646009174,
   "scoreDelta": {
   "dayChange": -0.5767549117977069,
   "weekChange": -0.5311640476663939,
   "monthChange": -0.2578449396243201
},
   "bucket": "50-59"
}
```


Now suppose you want to know who you have influenced (your influencees) and who influences you (your influencers). After all, this is what Klout is all about. Influence is measured by the action you drive.

To get your influencers and influencees, you need to use the `influence` endpoint, passing in your Klout ID.

3. Analyze the response

And here's the influence resource's response:

```
{
    "myInfluencers": [{
        "entity": {
            "id": "441634251566461018",
            "payload": {
                "kloutId": "441634251566461018",
                "nick": "jekyllrb",
                "score": {
                    "score": 50.41206120210041,
                    "bucket": "50-59"
                },
                "scoreDeltas": {
                    "dayChange": -0.05927708546307997,
                    "weekChange": -0.739829931907181,
                    "monthChange": -0.7917151139830239
                }
            }
        }
    }, {
        "entity": {
            "id": "33214052017370475",
            "payload": {
                "kloutId": "33214052017370475",
                "nick": "Mrtnlrssn",
                "score": {
                    "score": 22.45014953758632,
                    "bucket": "20-29"
                },
                "scoreDeltas": {
                    "dayChange": -0.3481056157609004,
                    "weekChange": -2.132213372307284,
                    "monthChange": -2.315034722843535
                }
            }
        }
    }, {
        "entity": {
            "id": "177892199475207065",
            "payload": {
                "kloutId": "177892199475207065",
                "nick": "TCSpeakers",
                "score": {
                    "score": 28.23034124231384,
                    "bucket": "20-29"
                },
                "scoreDeltas": {
                    "dayChange": 0.00154327588529668,
```

```
                "weekChange": -0.6416866188503434,
                "monthChange": -4.226666088333872
            }
        }
    }
}, {
    "entity": {
        "id": "91760850663150797",
        "payload": {
            "kloutId": "91760850663150797",
            "nick": "JohnFoderaro",
            "score": {
                "score": 39.39045702175103,
                "bucket": "30-39"
            },
            "scoreDeltas": {
                "dayChange": -0.6092388403641991,
                "weekChange": -0.699356032047298,
                "monthChange": 5.34513233077341
            }
        }
    }
}, {
    "entity": {
        "id": "1057244",
        "payload": {
            "kloutId": "1057244",
            "nick": "peterlalonde",
            "score": {
                "score": 42.39625419500191,
                "bucket": "40-49"
            },
            "scoreDeltas": {
                "dayChange": -0.32068173129262334,
                "weekChange": 0.14276611846587173,
                "monthChange": -0.9354253686809457
            }
        }
    }
}],
"myInfluencees": [{
    "entity": {
        "id": "537311",
        "payload": {
            "kloutId": "537311",
            "nick": "techwritertoday",
            "score": {
```

```
                "score": 49.99313854987996,
                "bucket": "40-49"
            },
            "scoreDeltas": {
                "dayChange": -0.10510042996928348,
                "weekChange": -0.568647896457648,
                "monthChange": 0.3425617785475197
            }
        }
    }
}, {
    "entity": {
        "id": "91760850663150797",
        "payload": {
            "kloutId": "91760850663150797",
            "nick": "JohnFoderaro",
            "score": {
                "score": 39.39045702175103,
                "bucket": "30-39"
            },
            "scoreDeltas": {
                "dayChange": -0.6092388403641991,
                "weekChange": -0.699356032047298,
                "monthChange": 5.34513233077341
            }
        }
    }
}, {
    "entity": {
        "id": "33214052017370475",
        "payload": {
            "kloutId": "33214052017370475",
            "nick": "Mrtnlrssn",
            "score": {
                "score": 22.45014953758632,
                "bucket": "20-29"
            },
            "scoreDeltas": {
                "dayChange": -0.3481056157609004,
                "weekChange": -2.132213372307284,
                "monthChange": -2.315034722843535
            }
        }
    }
}, {
    "entity": {
        "id": "45598950992256021",
```

```
            "payload": {
                "kloutId": "45598950992256021",
                "nick": "DavidEgyes",
                "score": {
                    "score": 40.40572793362214,
                    "bucket": "40-49"
                },
                "scoreDeltas": {
                    "dayChange": 0.001934309078080787,
                    "weekChange": 2.233816485488269,
                    "monthChange": 1.4901401977594801
                }
            }
        }
    }, {
        "entity": {
            "id": "46724857496656136",
            "payload": {
                "kloutId": "46724857496656136",
                "nick": "fabi_ator",
                "score": {
                    "score": 30.32498605174672,
                    "bucket": "30-39"
                },
                "scoreDeltas": {
                    "dayChange": -0.005890177199574964,
                    "weekChange": -0.6859163242901047,
                    "monthChange": -5.293301673692355
                }
            }
        }
    }],
    "myInfluencersCount": 5,
    "myInfluenceesCount": 5
}
```


The response contains an array containing 5 influencers and an array containing 5 influencees. (Remember the square brackets denote an array; the curly braces denote an object. Each array contains a list of objects.)

5. Pull out the information you need

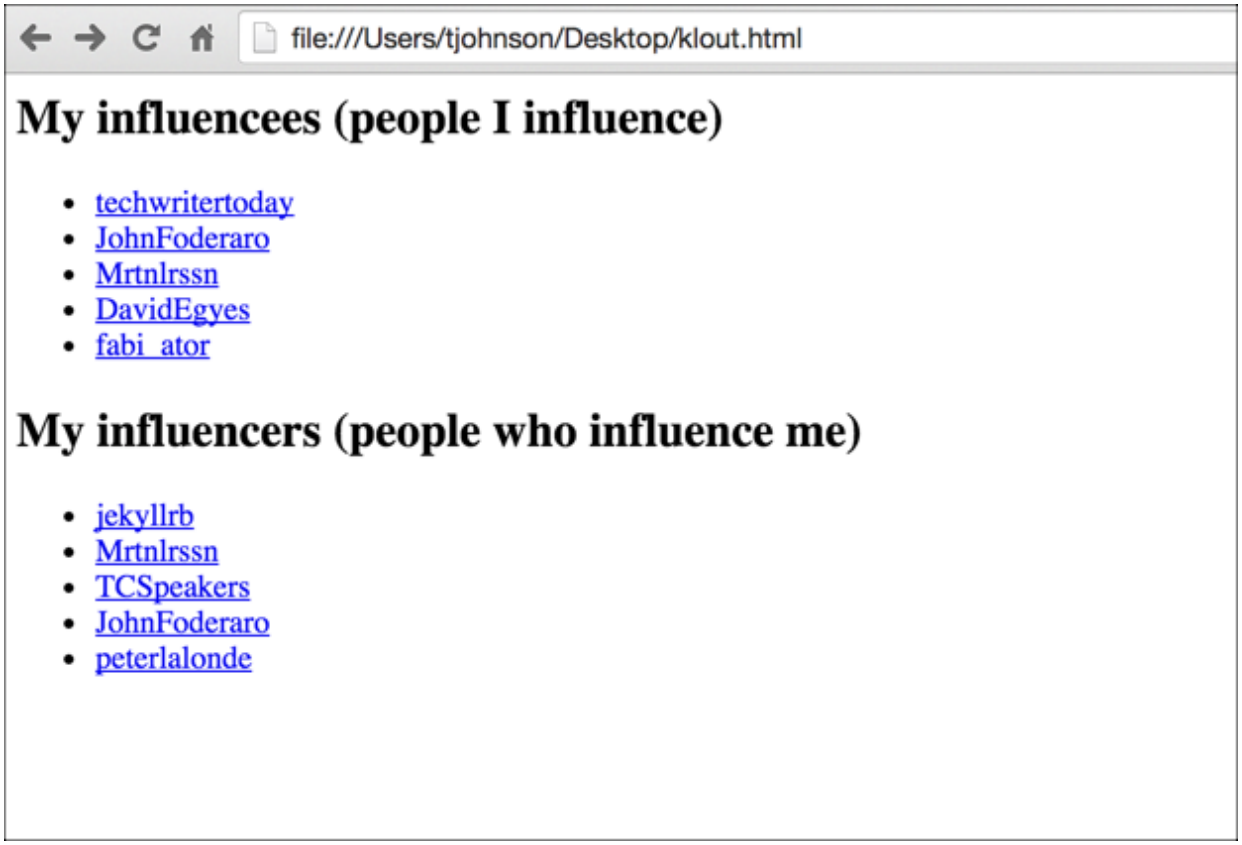
Suppose you just want a short list of Twitter names with their links.

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.j
s"></script>

<script>
  var settings = {
    "async": true,
    "crossDomain": true,
    "url": "http://api.klout.com/v2/user.json/1134760/influence?key={api ke
y}&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/'+in
f.entity.payload.nick + '">' + inf.entity.payload.nick + '</a></li>');
     });
    $.each( data.myInfluencers, function( i, inf ) {
       $("#kloutinfluencers").append('<li><a href="http://twitter.com/'+in
f.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"

</body>
</html>
```

Note: Remember to swap in your own API key in place of `{api key}`.

The result looks like this:



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/influenc
e?key={api key}&callback=?. No 'Access-Control-Allow-Origin' header is prese
nt 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).

(I'm not entirely sure why Klout gives this error unless you use CORS...)

Course Completion

You finished!

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

Summary of what you learned

During this 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 display the 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 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 my next course. See Publishing API documentation (page 0) to get started.

Feedback

If you have feedback about the course, please send it to me.