## **WORKING WITH APIS**

### **Introduction**

One of the most powerful things a web developer can do is fetching data from a server and displaying it creatively on their site. In many cases, the server solely exists for that specific site. The server could contain blog posts, user data, high scores for a game or anything else. In other cases, the server is an open service that serves data to anyone that wants to use it (i.e. weather data or stock prices). In either case, the methods of accessing and then using that data are essentially the same.

### **APIs**

Servers that are created for serving data for external use (in websites or apps) are often referred to as APIs or [‘Application Programming Interfaces’](https://www.youtube.com/watch?v=s7wmiS2mSXY).

There are multiple ways of requesting data from an API, but all of them basically do the same thing. For the most part, APIs are accessed through URLs, and the specifics of how to query these URLs change based on the specific service you are using. For example, the OpenWeatherMap API has several types of data that you can request. To get the current weather in a specific location, you need to request data from this URL:

api.openweathermap.org/data/2.5/weather?q=London,uk

You’ll want to switch out the city for the location you’re requesting. The specifics for using any API are usually documented on the service’s website. [Check here for the OpenWeatherMap API documentation](https://openweathermap.org/current).

If you haven’t already, go ahead and paste the weather URL above into your browser…(we’ll wait).

Unless the implementation of that specific API has changed, you probably get an error like this:

{"code":401, "message": "Invalid API key. Please see http://openweathermap.org/faq#error401 for more info."}

This brings us to another point about APIs. In most cases, you have to sign up and get an API key to use them. Obtaining the API key is as simple as signing up on their website and using it is usually as easy as pasting it into the URL:

http://api.openweathermap.org/data/2.5/weather?q=London,uk&APPID=1111111111

(exactly how to include the key changes from service to service)

Services like the OpenWeatherMap use API keys to track who is requesting the data they serve, and how much data they are requesting. The reason they do this is so that people can’t take advantage of their service. Running servers, especially large ones costs money, and while each current weather request (or whatever) is relatively cheap, if the amount of requests gets too high the cost could be significant. Imagine using that API to create an amazing weather app that gets used all over the world….you could easily have thousands of people accessing that data every minute!

By signing up for a service and getting an API key you are letting the service track how much you are actually using. In many cases services are limited as to how much data they can request for free. With the weather app example, their free plan only allows you to make 60 requests per minute and also limits what types of data you can access ([details here if you’re interested](https://openweathermap.org/price)). So, if your app became successful, you would probably need to pay for a better account.

Luckily for us, the majority of our apps are only going to be used by us and the people that view our portfolios. So we’ll get by *just fine* with free services.

Once you get a key (try this now if you like!) and waited for its activation (see [Do I need to activate my API key?](https://openweathermap.org/faq)) you can paste the URL into the browser again (including your key of course) and hopefully, you’ll see a proper response:

{"coord":{"lon":-77.73,"lat":38.77},"weather":[{"id":800,"main":"Clear","description":"clear sky","icon":"01d"}],"base":"stations","main":{"temp":75.74,"pressure":1017,"humidity":57,"temp\_min":71.6,"temp\_max":78.8},"visibility":16093,"wind":{"speed":3.87,"deg":291},"clouds":{"all":1},"dt":1504188900,"sys":{"type":1,"id":2886,"message":0.0053,"country":"US","sunrise":1504175992,"sunset":1504222878},"id":4775660,"name":"New Baltimore","cod":200}

### **Fetching Data**

So how do we actually get the data from an API into our code?

A couple of years ago the main way to access API data in your code was using an XMLHttpRequest. This function still works in all browsers, but unfortunately, it is not particularly nice to use. The syntax looks something like this:

// Just getting XHR is a mess! if (window.XMLHttpRequest) { // Mozilla, Safari, ... request = new XMLHttpRequest(); } else if (window.ActiveXObject) { // IE try { request = new ActiveXObject('Msxml2.XMLHTTP'); } catch (e) { try { request = new ActiveXObject('Microsoft.XMLHTTP'); } catch (e) {} } } // Open, send. request.open('GET', 'https://url.com/some/url', true); request.send(null);

Ouch. That was painful.

Developers, feeling the pain of having to write that stuff out, began writing 3rd party libraries to take care of this and make it much easier to use. Some of the more popular libraries are [axios](https://github.com/mzabriskie/axios) and [superagent](https://github.com/visionmedia/superagent), both of which have their strengths and weaknesses.

More recently, however, web browsers have begun to implement a new native function for making HTTP requests, and that’s the one we’re going to use and stick with for now. Meet fetch:

// URL (required), options (optional) fetch('https://url.com/some/url') .then(function(response) { // Successful response :) }) .catch(function(err) { // Error :( });

In case you’ve forgotten, scroll back up and look at how you would use XHR to do the same thing. While you’re admiring how nice and clean that code is, notice the .then() and .catch() functions there. Do you remember what those are? (PROMISES!)

Let’s change up our API for this example. We’re going to walk through an example using fetch with the [giphy](https://giphy.com/) API to display a random gif on a webpage. The API requires you to sign up and get a free API key, so go ahead and [do that here](https://developers.giphy.com/docs/).

Giphy has several methods for searching and finding gifs which you can read about in their documentation. Today we’re just going to use the ‘translate’ endpoint because it’s the simplest one for our purposes. You can find the appropriate URL in their documentation by scrolling down [here](https://developers.giphy.com/docs/). What it tells us is that the correct URL is api.giphy.com/v1/gifs/translate and that it requires 2 parameters, your api\_key and a search term. If you put it all together correctly (with YOUR API key) you should get something like this:

'https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats' // of course we're searching for cats

Go ahead and try that URL (with YOUR API key) in a browser. If everything goes well you should get a relatively long string of data and no errors.

### **CORS**

A side note before we start putting this into our code. For security reasons, by default, browsers restrict HTTP requests to outside sources (which is exactly what we’re trying to do here). There’s a very small amount of setup that we need to do to make fetching work. Learning about this is outside our scope right now, but if you want to learn a bit about it this [Wikipedia article](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing) and this [Javascript.info article](https://javascript.info/fetch-crossorigin) are good starting points.

Whether or not you took the detour to learn all about Cross Origin Resource Sharing (CORS) the fix is simple. With fetch, you are able to easily supply a JavaScript object for options. It comes right after the URL as a second parameter to the fetch function:

fetch('url.url.com/api', { mode: 'cors' });

Simply adding the {mode: 'cors'} after the URL, as shown above, will solve our problems for now. In the future, however, you may want to look further into the implications of this restriction.

### **Let’s Do This**

For now, we’re going to keep all of this in a single HTML file. So go ahead and create one with a single blank image tag and an empty script tag in the body.

<!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <title>Document</title> </head> <body> <img src="#" /> <script> </script> </body> </html>

In the script tag, let’s start by selecting the image and assigning it to a variable so that we can change the URL once we’ve received it from the Giphy API.

<script> const img = document.querySelector('img'); </script>

Adding fetch with our URL from above is also relatively easy:

<script> const img = document.querySelector('img'); fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}) .then(function(response) { console.log(response.json()); }); </script>

You should now be able to open the HTML file in your browser, and while you won’t see anything on the page, you *should* have something logged in the console. The trickiest part of this whole process is deciphering how to get to the data you desire from the server’s response. In this case, inspecting the browser’s console will reveal that what’s being returned is *another* Promise… to get the data we need another .then() function.

<script> const img = document.querySelector('img'); fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}) .then(function(response) { return response.json(); }) .then(function(response) { console.log(response); }); </script>

Now we have a JavaScript object and if you inspect it closely enough you’ll find that the data we need (an image URL) is nested rather deeply inside the object:



To get to the data we need to drill down through the layers of the object until we find what we want!

<script> const img = document.querySelector('img'); fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}) .then(function(response) { return response.json(); }) .then(function(response) { console.log(response.data.images.original.url); }); </script>

Running the file should now log the URL of the image. All that’s left to do is set the source of the image that’s on the page to the URL we’ve just accessed:

<script> const img = document.querySelector('img'); fetch('https://api.giphy.com/v1/gifs/translate?api\_key=YOUR\_KEY\_HERE&s=cats', {mode: 'cors'}) .then(function(response) { return response.json(); }) .then(function(response) { img.src = response.data.images.original.url; }); </script>

If all goes well, you should see a new image on the page every time you refresh!

If you’ve gotten lost along the way, check out [this jsbin project](http://jsbin.com/canofar/edit?html,output). Besides the glorious styling, this is what your version should look like.

While we are pushing this API key to the frontend, this isn’t something you should do with any key that is not free, anything on the client is public knowledge. Handling keys that are unsafe to push to the frontend will be taught in later sections if you haven’t been exposed in the Ruby course.

### **Assignment**

1. Read the fetch documentation [here](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch). It’s not all that complicated to use, but we’ve only really scratched the surface at this point.
2. Check out [this list](https://github.com/abhishekbanthia/Public-APIs) of free, open APIs and let your imagination go wild.
3. Expand on our little project here by adding a button that fetches a new image without refreshing the page.
4. Add a search box so users can search for specific gifs. You should also investigate adding a .catch() to the end of the promise chain in case Giphy doesn’t find any gifs with the searched keyword. Add a default image, or an error message if the search fails.