The Fetch API is part of ECMAScript 6 — which means that most browsers still aren’t supporting it. If you want to use this API, you will need to use what is called a “polyfill”.

*A polyfill, or polyfiller, is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide*natively*.*

To use the fetch API you’ll need to include two different polyfills —[one for the fetch method](https://github.com/github/fetch) and [one for Promises](https://github.com/stefanpenner/es6-promise). Once you have those installed, you can start using the fetch method.

Let’s look at a simple example of the fetch method first .

A GET request to just one API. Here’s what the code would look like:

fetch('http://api.exampledomain.com/api/search').then(  
 function(response){  
 return response.json();  
 }  
);

Pretty simple right (at least compared to XMLHttpRequest!)? Like I mentioned earlier, the fetch method returns a Promise object. Promise objects have a .then() method which takes two parameters — a success callback function and failure callback function. In this example, when the fetch method successfully returns from making the API request, the success callback function is called which has a Response object as it’s parameter. The Response object also has several methods available to it, one of which is a .json() method which also returns Promise. So, if you want to do anything with a response from an API request and that response is in a JSON format, you’ll need to do this:

fetch('http://api.exampledomain.com/api/search').then(  
 function(response){  
 return response.json();  
 }  
).then(function(jsonData){

//handle json data processing here  
});

As you can see, if you’ve got multiple API requests you want to make and you want to handle the data in a specific order, or you don’t want to display anything on the screen until all the data is returned from the various API requests, you’ll be chaining quite a few .then() functions together. This is what I ended up doing in the first iteration of my Shopsifter code and it was getting pretty ugly and unmaintainable. Luckily there is a better way!

**Promise.all()**

Say I’m searching for “blue suede shoes” on [Shopsifter](http://shopsifter.co/" \t "_blank). When I hit enter on the home page after typing my query, what I call the “merchants” page loads. This page makes a request to multiple APIs to see which stores have that product available. I didn’t want anything to load on the page until all the requests from the various APIs had resolved themselves. This is where the Promise.all() method comes in handy. From the Mozilla Developer Network documentation:

*The****Promise.all(iterable)****method returns a promise that resolves when all of the promises in the iterable argument have resolved, or rejects with the reason of the first passed promise that rejects.*

Here’s an example of how to use the Promise.all() method:

var apiRequest1 = fetch('api.example1.com/search').then(function(response){   
 return response.json()  
});

var apiRequest2 = fetch('api.example2.com/search').then(function(response){  
 return response.json()  
});

var combinedData = {"apiRequest1":{},"apiRequest2:{}};  
Promise.all([apiRequest1,apiRequest2]).then(function(values){  
 combinedData["apiRequest1"] = values[0];  
 combinedData["apiRequest2"] = values[1];  
 return combinedData;  
});

The Promise.all() method takes an array of Promise objects. When all of those objects have resolved then the Promise.all() function resolves and the callback function passed into the .then() function is called. The callback function is an array of values that match up with the array of Promise objects that were passed into the Promise.all() function. In the case of Shopsifter, the API calls I made returned back data in different forms, so using the Promise.all() function allowed me to create a cohesive dataset that could be used by my components.