**Requests with ES6**

In the previous lesson, we spent a lot of time dealing with asynchronous data (remember AJAX/ Asynchronous JavaScript And XML?). Many of our web page interactions rely on asynchronous events, so managing these events is essential to good web development.

To make asynchronous event handling easier, *promises* were introduced in JavaScript in ES6:

* [Mozilla Development Network: Promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)

A promise is an object that handles asynchronous data. A promise has three states:

* *pending* : when a promise is created or waiting for data.
* *fulfilled* : the asynchronous operation was handled successfully.
* *rejected* : the asynchronous operation was unsuccessful.

The great thing about promises is that once a promise is fulfilled or rejected, you can chain an additional method to the original promise.

In this lesson, we will explain how to use fetch(), which uses promises to handle requests. Then, we will simplify requests using async and await.

We'll use the Datamuse API for GET requests and Rebrandly URL Shortener API for POST requests. For you to complete the lessons on POST, make sure you create a Rebrandly API Key by following the instructions in the article below:

* [Codecademy Articles: Rebrandly URL Shortener API](https://www.codecademy.com/articles/rebrandly-signup) .

**fetch() GET Requests I**

The first type of requests we’re going to tackle are GET requests using fetch()

* [MDN: Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API).

The fetch() function:

* Creates a request object that contains relevant information that an API needs.
* Sends that request object to the API endpoint provided.
* Returns a promise that ultimately resolves to a response object, which contains the status of the promise with information the API sent back.

In the next exercise we’ll go over the boilerplate code for using fetch() and walk through what each step does!

**fetch() GET Requests II**

We are going to reconstruct the boilerplate code necessary to create a GET request using the fetch() function step-by-step.

Feel free to refer to the fetch() GET diagram at any point while completing this exercise:

* [fetch() GET diagram](https://s3.amazonaws.com/codecademy-content/courses/intermediate-javascript-requests/diagrams/fetch+GET+diagram.svg)

# fetch() GET Requests III

In the previous exercise, you wrote the boilerplate code for a GET request using fetch() and .then(). In this exercise, you’re going to use that code and manipulate it to access the Datamuse API and render information in the browser.

* [Datamuse API](https://www.datamuse.com/api/).

If the request is successful, you'll get back an array of words that sound like the word you typed into the input field.

You may get some errors as you complete each step. This is because sometimes we've split a single step into one or more steps to make it easier to follow. By the end, you should be receiving no errors.

# fetch() GET Requests IV

Great job making it this far!

In the previous exercise, you created the query URL, called the fetch() function and passed it the query URL and a settings object. Then, you chained a .then() method and passed it two functions as arguments — one to handle the promise if it resolves, and one to handle network errors if the promise is rejected.

In this exercise, you’ll now take the information that was returned with the promise and manipulate the webpage!

# fetch() POST Requests I

In the previous exercise, you successfully wrote a GET request using the fetch API and handled Promises to get word suggestions from Datamuse. Give yourself a pat on the back (or two to treat yourself)!

Now, you're going to learn how to use fetch()to construct POST requests!

Take a look at the diagram to the right. It has the boilerplate code for a POST request using fetch().

Notice that the initial call takes two arguments: an endpoint and an object that contains information needed for the POST request. The rest of the request is identical to the GET request.

# fetch() POST Requests II

We are going to reconstruct the code from the previous exercise step-by-step until we have written a complete POST request using fetch()and .then().

Feel free to refer to the fetch() POST diagram at any point while completing this exercise:

* [fetch() POST diagram](https://s3.amazonaws.com/codecademy-content/courses/intermediate-javascript-requests/diagrams/fetch+POST+diagram.svg)

# fetch() Post Requests III

In the previous exercise, you created the boilerplate code for making a POST request using fetch() and .then(). In this exercise, you’re going to update that boilerplate code to allow you to shorten a URL using the Rebrandly URL Shortener API.

* [Rebrandly API](https://developers.rebrandly.com/).

If you haven't already created a Rebrandly API key, read through the Rebrandly sign up guide:

* [Codecademy Articles: Rebrandly URL Shortener API](https://www.codecademy.com/articles/rebrandly-signup).

If you reset the exercise at any point, you will have to paste in your API key again at the top!

# fetch() POST Requests IV

In the previous exercise you’ve positioned yourself to make the POST request by providing the endpoint and the object containing all the necessary information. In this exercise you’ll handle the response.

The request returns a Promise which will either be resolved or rejected. If the promise resolves, you can use that information and the ok status. Let's implement that knowledge into your code!

If you reset the exercise at any point, you will have to paste in your API key again at the top!

# fetch() POST Requests V

You’re almost done with the POST request you started a few lessons back!

In fact, this time you’ll add another .then() to the chain to finally make the information available to your webpage!

If you reset the exercise at any point, you will have to paste in your API key again at the top!

**async GET Requests I**

Let’s take a minute to appreciate what you’ve accomplished so far:

* used fetch() to make GET and POST requests.
* check the status of the responses coming back
* catch errors that might possibly arise
* taking successful responses and rendering it on the webpage

That is fantastic! It’s the basis of how the internet works!

In the following exercises, we’re going to take what you’ve learned about chaining Promises and make it simpler using functionality introduced in ES8: async and await. You read that right, you did the hard part already, now it’s time to make it easier.

The structure for this request will also be slightly different. Notice the new keywords async and await, as well as the try and catch statements.

We’ll be going over how to write the boilerplate code for async GET requests in the next lesson.

# async GET Requests II

We are going to walk through and recreate the boilerplate code necessary to create a GET request using the async and await.

Here are some key points to keep in mind as we walk through the code:

* Using an async function that will return a promise.
* await can only be used in an asyncfunction. await allows a program to run while waiting for a promise to resolve.
* In a try...catch statement, code in the try block will be run and in the event of an exception/error, the code in the catchstatement will run.

Feel free to refer to the async/await GET diagram at any point while completing this exercise:

* [async/await GET diagram](https://s3.amazonaws.com/codecademy-content/courses/intermediate-javascript-requests/diagrams/async+await+GET+diagram.svg)

# async GET Requests III

In the previous exercise, we created the boilerplate code for making a GET request using async and await.

In this exercise, you’re going to build on previously created boilerplate code to get nouns that describe the inputted word from the Datamuse API:

* [Datamuse API Documentation](https://www.datamuse.com/api/).

# async POST Requests I

Now that you’ve made an async GET request, let’s start on getting you familiar the asyncPOST request.

As with the other GET and POST requests that you’ve been making, an async POST request requires more information. Take a look at the diagram.

We still have the same structure of using tryand catch as before. But, in the fetch() call, we now have to include an additional argument that contains more information like method and body.

We’ll be explaining the why's and how's of the boilerplate code for async POST requests in the next lesson.

# async POST Requests II

Now we’re going to piece together a POST request using async and await.

Feel free to refer to the async/await diagram below at any point while completing this exercise:

* [async/await POST diagram](https://s3.amazonaws.com/codecademy-content/courses/intermediate-javascript-requests/diagrams/async+await+POST+diagram.svg)

# async POST Requests III

Since you’ve created the boilerplate code for a POST request, the next step is to incorporate that experience and logic into making a real request.

In this exercise, you'll need to retrieve your Rebrandly API key to access the Rebrandly API.

* [Rebrandly API Keys](https://app.rebrandly.com/account/api-keys).

If you reset the exercise at any point, you will have to paste in your API key again at the top!

# Review Requests II

Let's recap on the concepts covered in the previous exercises:

1. GET and POST requests can be created a variety of ways.
2. Use AJAX to asynchronously request data from APIs. fetch() and async/await are new functionalities developed in ES6 (promises) and ES8 respectively.
3. Promises are a new type of JavaScript object that represent data that will eventually be returned from a request.
4. fetch() is a web API that can be used to create requests. fetch() will return promises.
5. We can chain .then() methods to handle promises returned by fetch().
6. The .json() method converts a returned promise to a JSON object.
7. async is a keyword that is used to create functions that will return promises.
8. await is a keyword that is used to tell a program to continue moving through the message queue while a promise resolves.
9. await can only be used within functions declared with async.