**Introduction to Requests**

Have you ever wondered what happens after you click a “Submit” button on a web page? For instance, if you are submitting information, where does the information go? How is the information processed? The answer to the previous questions revolves around *HTTP requests*.

There are many types of HTTP requests. The four most commonly used types of HTTP requests are GET, POST, PUT, and DELETE. In this lesson, we’ll cover GET and POST requests. If you want to learn more about the different HTTP requests, we recommend the following documentation:

* [Mozilla Developer Network: HTTP methods](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)

With a GET request, we’re retrieving, or *getting*, information from some source (usually a website). For a POST request, we’re *posting* information to a source that will process the information and send it back.

In this lesson, we will explain how to make GET and POST requests by using JavaScript’s **XHR** object. We’ll also incorporate query strings into our requests.

We’ll use the Datamuse API for GET requests and the Rebrandly URL Shortener API for POST requests. To complete the exercise 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) .

**Instructions**

Click the button on the web page to see what the code in **main.js** does.

You’ll see *JSON* being displayed on the page. JSON, JavaScript Object Notation, will be the format in which our data is sent to us. That button you click produces a GET request. That’s right, you’ve just made a GET request!

Go to the next exercise to learn more about requests.

const jsonButton = document.querySelector('#generate');

const buttonContainer = document.querySelector('#buttonContainer');

const display = document.querySelector('#displayContainer');

const collection = ["Another", "More", "Next", "Continue", "Keep going", "Click me", "A new one"];

const generateJson = () => {

  const xhr = new XMLHttpRequest();

  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {

    if (xhr.readyState === XMLHttpRequest.DONE) {

  renderResponse(xhr.response);

      changeButton();

    }

  }

  xhr.open('GET', 'https://jsonplaceholder.typicode.com/users');

  xhr.send();

}

const formatJson = (resJson) => {

  resJson = JSON.stringify(resJson);

  let counter = 0;

  return resJson.split('')

  .map(char => {

    switch (char) {

      case ',':

        return `,\n${' '.repeat(counter \* 2)}`;

      case '{':

        counter += 1;

        return `{\n${' '.repeat(counter \* 2)}`;

      case '}':

        counter -= 1;

        return `\n${' '.repeat(counter \* 2)}}`;

      default:

        return char;

    }

  })

  .join('');

}

const renderResponse = (jsonResponse) => {

  const jsonSelection = Math.floor(Math.random() \* 10);

  display.innerHTML = `<pre>${formatJson(jsonResponse[jsonSelection])}</pre>`;

}

const changeButton = () => {

  const newText = Math.floor(Math.random() \* 7);

  jsonButton.innerHTML = `${collection[newText]}!`;

}

jsonButton.addEventListener('click', generateJson);

**HTTP Requests**

One of JavaScript’s greatest assets is its non-blocking properties, or that it is an *asynchronous* language.

Websites, like newspaper websites, take advantage of these non-blocking properties to provide a better user experience. Generally, a site’s code is written so that users don’t have to wait for a giant image to load before being allowed to read the actual article—rather, that text is rendered first and then the image can load in the background.

JavaScript uses an *event loop* to handle asynchronous function calls. When a program is run, function calls are made and added to a stack. The functions that make requests that need to wait for servers to respond then get sent to a separate queue. Once the stack has cleared, then the functions in the queue are executed.

Web developers use the event loop to create a smoother browsing experience by deciding when to call functions and how to handle asynchronous events. We’ll be exploring one system of technologies called Asynchronous JavaScript and XML, or AJAX.

To read more about the event loop, read the MDN documentation:

* [MDN Documentation: Event Loop](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop)

**Instructions**

**1.**

To get a glimpse of how the event loop works, take a look at the code in the code editor.

We’ll be using setTimeout(), which will pass a function call to the queue. The first argument is a callback and the second argument is the number of milliseconds the program must wait before the callback can be run.

The other console.log() calls are run from the stack.

**2.**

Interesting right?

What if we change the 2500 in setTimeout() to be 0? Essentially the callback doesn’t need to wait before it can be called. Do you think that this change will affect the order?

console.log('First message!');

setTimeout(() => {

   console.log('This message will always run last...');

}, 0);

console.log('Second message!');

# XHR GET Requests I

Asynchronous JavaScript and XML (AJAX), enables requests to be made after the initial page load. Initially, AJAX was used only for XML formatted data, now it can be used to make requests that have many different formats.

[MDN Documentation: Extensible Markup Language (XML)](https://developer.mozilla.org/en-US/docs/XML_introduction).

Similarly, the XMLHttpRequest (XHR) API, named for XML, can be used to make many kinds of requests and supports other forms of data.

Remember, we use GET to retrieve data from a source. Take a look at the boilerplate code in the diagram to see how to make an XHR GET request.

We’ll construct this template from scratch in a different exercise and walk through what each step does.



**XHR GET Requests II**

We are going to reconstruct XHR GET request boilerplate code step-by-step until we have written a complete GET request.

Feel free to refer to the XHR GET diagram at any point while completing this exercise:

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

**Instructions**

**1.**

First, we need to create the XMLHttpRequest object using the new operator. Save this object in a const called xhr.

Note: While the code will work regardless of how you name your variables, it is a common practice to name this object xhr.

Hint

The XMLHttpRequest object is used in JavaScript to create and send requests. To create a new instance of an object, you would use the new keyword. Like so:

const xhr = new XMLHttpRequest();

**2.**

Next, save the following URL to a const called url. Make sure the URL is wrapped in quotes so that it is a string.

https://api-to-call.com/endpoint

Hint

The line of code looks like:

const url = 'https://api-to-call.com/endpoint';

**3.**

Set the responseType property of the xhr object to equal 'json'.

JSON is JavaScript Object Notation, which is how the response is going to be formatted.

Hint

To access responseType of xhr, use dot notation. The line of code looks like:

xhr.responseType = 'json';

**4.**

Set the xhr.onreadystatechange event handler equal to an anonymous arrow function. Leave the function empty.

Hint

The .onreadystatechange of xhr will contain an event listener which will execute when the readyState property changes. The syntax will look like:

xhr.onreadystatechange = () => {};

**5.**

In the code block of the function you created in the previous step, add this conditional statement:

if (xhr.readyState === XMLHttpRequest.DONE) {

}

The purpose of this conditional statement checks to see if the request has finished.

**6.**

In the code block of the conditional statement, return the response property of xhr.

Hint

To access the response property, we can use dot notation like so: xhr.response. This response will contain the data that we’re getting back from the request.

The syntax will look like:

return xhr.response;

**7.**

Below the function you created in the previous two steps, call the .open() method on the xhr object and pass it 'GET' and url as arguments.

.open() creates a new request and the arguments passed in determine the type and URL of the request.

Hint

The syntax will look like:

xhr.open('GET', url);

**8.**

On the following line, call the .send() method on the xhr object. Do not pass it any arguments.

Nice work! You’ve written the boilerplate code for an AJAX GET request using an XMLHttpRequest object.

Hint

The syntax for this line of code will be:

xhr.send();

const xhr = new XMLHttpRequest();

const url =  'https://api-to-call.com/endpoint';

xhr.responseType = 'json';

xhr.onreadystatechange = () => {

if (xhr.readyState === XMLHttpRequest.DONE) {

return xhr.response;

}

xhr.open('GET', url);

xhr.send();

};

**XHR GET Requests III**

By this point, you have an idea of how to write the boilerplate code for an AJAX request using an XHR object.

In this exercise, you will incorporate that boilerplate code to make a GET request to the Datamuse API to search for words that rhyme!

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

**Instructions**

**1.**

At the top of **main.js**, create a const named url and save to it the following URL (as a string):

https://api.datamuse.com/words?

You’ll be using this URL to direct your request.

Hint

The syntax will look something like:

const nameOfVariable = 'a string of sorts';

**2.**

Underneath const url, create another const named queryParams and assign it to 'rel\_rhy='

'rel\_rhy=' is the start of a parameter for the query string. This parameter will narrow your search to words that rhyme.

Hint

The syntax will look like:

const nameOfVariable = 'a string of sorts';

**3.**

In the function getSuggestions(), create a const named wordQuery and assign it to be inputField.value.

inputField.value grabs what is in the inputField and assigns it to the variable wordQuery.

You’ll be working in getSuggestions() for the rest of the exercise.

Hint

Make sure you’re inside the code block of getSuggestions().

**4.**

Now create another const named endpoint, and assign equal to a string that concatenates url, queryParams, and wordQuery.

endpoint will store the value of the entire URL and query string.

Hint

There are many ways of concatenating strings; take a look at the following examples:

const word1 = ‘hello’;

const word2 = ‘world!’;

console.log(word1 + ' ' + word2);

// ‘hello world!’

You can also interpolate it using a template literal, but remember to use backticks!

console.log(`${word1} ${word2}`);

// ‘hello world!’

**5.**

You can now start on the XMLHttpRequest object. Declare a const named xhr and use the new operator to create the XMLHttpRequest object.

Hint

Make sure you’re still inside the code block of getSuggestions(). The syntax will look like:

const xhr = new XMLHttpRequest();

**6.**

Set the responseType of xhr to 'json'.

When data is sent back, it will be in JSON format.

Hint

Make sure you’re in the code block for getSuggestions().

To access the .responseType property of xhr, use dot notation like: xhr.responseType. Then assign it a value of ‘json’. The syntax for this would look like:

xhr.responseType = 'json';

**7.**

Assign an anonymous arrow function to the onreadystatechange event handler of xhr.

Hint

You can think of the onreadystatechange event handler as another property. Use dot notation to access it and assign it to an anonymous arrow function.

To access onreadystatechange using dot notation, use the following syntax:

xhr.onreadystatechange

Then, to assign it an anonymous arrow function:

xhr.onreadystatechange = () => {

// Event handler code

}

**8.**

Include the following code inside the code block of the anonymous arrow function you just created:

if (xhr.readyState === XMLHttpRequest.DONE) {

renderRawResponse(xhr.response)

}

The renderRawResponse() helper function can be viewed at **public/helperFunctions.js**.

**9.**

Below the anonymous arrow function you just created (but still inside of getSuggestions()), call the .open() method on the XHR object and pass it 'GET' and endpoint as respective arguments. This method call will create a new request using the two arguments: 'GET' sets the method and url sets the destination.

Underneath .open(), call the .send() method on xhr and pass it no arguments. The .send() method will send the request to the server.

Then run your code.

Type in a word in the input field and click the submit button. If all went well, the response field in the browser will display the raw response from the API!

Stuck? Get a hint

**10.**

Now delete renderRawResponse(xhr.response) and replace it with renderResponse(xhr.response) and run your code.

Then type in a word in the input field and click the submit button.

Great, now it looks nice and formatted!

// Information to reach API

const url = 'https://api.datamuse.com/words?';

const queryParams = 'rel\_rhy=';

// Selecting page elements

const inputField = document.querySelector('#input');

const submit = document.querySelector('#submit');

const responseField = document.querySelector('#responseField');

// AJAX function

const getSuggestions = () => {

  const wordQuery = inputField.value;

  const endpoint = `${url}${queryParams}${wordQuery}`;

  const xhr = new XMLHttpRequest();

  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {

    if (xhr.readyState === XMLHttpRequest.DONE) {

      renderResponse(xhr.response);

    }

  }

  xhr.open('GET', endpoint);

  xhr.send();

}

// Clear previous results and display results to webpage

const displaySuggestions = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  getSuggestions();

}

submit.addEventListener('click', displaySuggestions);

# XHR GET Requests IV

In the previous exercise, you made a GET request to the Datamuse API to find words that rhyme. In this exercise, we will create a request to set a topic and find adjectives that describe the input word using query strings.

A query string contains additional information to be sent with a request. The Datamuse API allows us to retrieve more specific data with query strings attached to the request URL.

* [Wiki: query string](https://en.wikipedia.org/wiki/Query_string)

A query string is separated from the URL using a ? character. After ?, you can then create a parameter which is a key value pair joined by a =. Examine the example below:

'https://api.datamuse.com/words?key=value'

If you want to add an additional parameter you will have to use the & character to separate your parameters. Like so:

'https://api.datamuse.com/words?key=value&anotherKey=anotherValue'

Let’s incorporate this into our code!

**Instructions**

**1.**

Let’s do something else besides finding words that rhyme. Have const queryParams store the value 'rel\_jjb='. This will search for words that describe another word.

Run the code. Then, type in a word and click the submit button on the web page.

Hint

You’ll find queryParams near the top of **main.js**. Assign it 'rel\_jjb='

const queryParams = 'rel\_jjb='

**2.**

Since we want to retrieve more specific results, we should create another parameter. Create another const additionalParams underneath queryParams, and assign it '&topics='.

Reminder: the & character at the start of the string is used to separate our parameters. The = character will join the key 'topics' to a value.

Hint

Hint: The syntax will look like:

const someVar = 'assigned string'

**3.**

Now, if you were wondering why there’s a second input field, that’s exactly what we’re going to hook up now! The word typed in here will be the value portion of our second parameter.

The second parameter will filter the response using the word typed into the second input field. In the next step, we’ll incorporate this parameter in with our query string.

In the code block of getSuggestions(), under wordQuery, declare a const topicQuery, and assign it to the value of topicField.

Hint

To access the value of topicField use dot notation like so:

const topicQuery = topicField.value

**4.**

In getSuggestions(), change the value of endpoint to a concatenated string of url, queryParams, wordQuery, additionalParams, and topicQuery.

Run the code. Then enter a word and a topic and click submit.

Our request will have returned a response of adjectives that are related to a topic! Feel free to play around with variables and parameters to get more word suggestions!

Hint

To concatenate a string you have a few options, here’s the syntax for two techniques below:

const word1 = ‘hello’;

const word2 = ‘world!’;

console.log(word1 + ' ' + word2);

// ‘hello world!’

You can also interpolate it using a template literal, but remember to use backticks!

console.log(`${word1} ${word2}`);

// ‘hello world!’

// Information to reach API

const url = 'https://api.datamuse.com/words?';

const queryParams = 'rel\_jjb=';

const additionalParams = '&topics=';

// Selecting page elements

const inputField = document.querySelector('#input');

const topicField = document.querySelector('#topic');

const submit = document.querySelector('#submit');

const responseField = document.querySelector('#responseField');

// AJAX function

const getSuggestions = () => {

  const wordQuery = inputField.value;

  const topicQuery = topicField.value;

  const endpoint = `${url}${queryParams}${wordQuery}${additionalParams}${topicQuery}`;

  const xhr = new XMLHttpRequest();

  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {

    if (xhr.readyState === XMLHttpRequest.DONE) {

      renderResponse(xhr.response);

    }

  }

  xhr.open('GET', endpoint);

  xhr.send();

}

// Clear previous results and display results to webpage

const displaySuggestions = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  getSuggestions();

}

submit.addEventListener('click', displaySuggestions);

# XHR POST Requests I

Reminder: If you haven’t already signed up for an API Key from Rebrandly, please read this Rebrandly sign up guide.

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

Great! By this point you’ve signed up for an API key, and you know the essence of making a GET request. We will be making a POST request using the Rebrandly API.

The major difference between a GET request and POST request is that a POST request requires additional information to be sent through the request. This additional information is sent in the body of the post request.

We’ll walk through the code from the diagram and construct our own POST request in the next lesson.

**Instructions**

Move on to the next exercise when you’re ready!



# XHR POST Requests II

We are going to reconstruct the code from the previous exercise step-by-step until we have written a complete AJAX POST request.

Feel free to refer to the XHR POST diagram at any point while completing this exercise:

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

**Instructions**

**1.**

Create a new XMLHttpRequest object using the new operator, and save it in a const called xhr.

The XMLHttpRequest object is used in JavaScript to interact with servers.

Stuck? Get a hint

**2.**

Next, save the following URL to a const called url. Make sure the URL is wrapped in quotes so that it is a string.

https://api-to-call.com/endpoint

The URL will direct the request to the correct server.

**3.**

Create a new const called data, and save this line of code to it:

JSON.stringify({id: '200'});

JSON.stringify() will convert a value to a JSON string. By converting the value to a string, we can then send the data to a server.

**4.**

Set the responseType property of the xhr object to be 'json'.

Stuck? Get a hint

**5.**

Set the xhr.onreadystatechange event handler equal to an anonymous arrow function. Leave the function empty until the next step.

.onreadystatechange will contain the event handler that will be called when xhr‘s state changes.

Hint

To assign xhr.onreadystatechange to an anonymous function, use the following syntax:

xhr.onreadystatechange = () => {}

**6.**

In the code block of the function you created in the previous step, add a conditional statement that checks to see if the readyState of xhr is equal to XMLHttpRequest.DONE.

Hint

Make sure you’re inside the code block of the anonymous arrow function.

The if conditional will check to see if the request has finished.

To access the readyState of the xhr object, use dot notation like so: xhr.readyState.

When comparing two objects use ===.

The syntax will look like:

if(xhr.readyState === XMLHttpRequest.DONE){

}

**7.**

In the code block of the conditional statement, return the response property of xhr. The response property will contain the data that we’re getting back from the POST request.

Hint

To access the response property, we can use dot notation like so: xhr.response.

**8.**

Below the function you created in the previous two steps, call the .open() method on the xhr object and pass it 'POST' and url as arguments.

.open() creates a new request and the arguments passed in determine what type of request is being made and where it’s being made to.

Hint

The syntax will look like:

xhr.open('POST', url)

**9.**

On the following line, call the .send() method on the xhr object and pass data as an argument.

.send() will send the request to the server.

Nice work! You’ve written the boilerplate code for an AJAX POST request using an XMLHttpRequest object.

Hint

The syntax will look like:

xhr.send(data)

const xhr = new XMLHttpRequest();

const url = 'https://api-to-call.com/endpoint';

const data = JSON.stringify({id: '200'});

xhr.responseType = 'json';

xhr.onreadystatechange = () => {

  if(xhr.readyState === XMLHttpRequest.DONE){

    return xhr.response;

  }

}

xhr.open('POST', url);

xhr.send(data);

# XHR Post Requests III

Reminder: If you haven’t already signed up for an API Key from Rebrandly, please read the article:

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

In this exercise, you’ll be making a POST request to the Rebrandly API to shorten a URL.

Get ready! You’re now going to incorporate the previous lesson’s boilerplate code into making an actual POST request!

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

**Instructions**

**1.**

Copy your Rebrandly API Key, and assign it to the const apiKey at the top of your code.

**2.**

Within the code block of shortenUrl(), create a const called urlToShorten, and save inputField.value to it. urlToShorten will now save the value of the input field

Note: for the remainder of this exercise’s instructions we will be working inside the code block of shortenUrl()!

Hint

Make sure you’re inside the code block for shortenUrl().

**3.**

Create a const called data, and save the following code to it:

JSON.stringify({destination: urlToShorten});

We’re including this information because the API expects to see an object with a key destination that has a value of a URL.

Hint

Make sure you’re inside the code block for shortenUrl().

data goes under urlToShorten.

**4.**

Create a new XMLHttpRequest object using the new operator, and save it to a const called xhr.

Hint

The syntax will look like:

const xhr = new XMLHttpRequest();

**5.**

Set the responseType property of the xhr object to be 'json'.

Hint

To access the responseType property of the xhr object, use dot notation.

The syntax to use dot notation looks like: xhr.someProperty.

**6.**

Save an empty anonymous arrow function to the onreadystatechange event handler of the xhr object. This function will not take in any parameters.

Inside the anonymous function’s code block, include the following code inside of its code block:

if (xhr.readyState === XMLHttpRequest.DONE) {

renderRawResponse(xhr.response);

}

The renderRawResponse function can be viewed at **public/helperFunctions.js**.

Hint

The syntax will look like:

xhr.onreadystatechange = () => {

if (xhr.readyState === XMLHttpRequest.DONE) {

renderRawResponse(xhr.response);

}

}

**7.**

Below the anonymous function you just created, call the .open() method on xhr, and pass it 'POST' and url as respective arguments.

Hint

The syntax will look something like:

xhr.open('HTTP Request Type', url)

**8.**

To access the Rebrandly API, we need a header with two key-value pairs. In the header, you must include values for 'Content-type' and an 'apikey'.

To set the header, we have to include the following code below our .open() method.

xhr.setRequestHeader('Content-type', 'application/json');

xhr.setRequestHeader('apikey', apiKey);

Hint

Make sure you’re inside the code block for shortenUrl().

**9.**

On xhr, call the .send() method, and pass it data as an argument.

Hint

Make sure you’re inside the code block for shortenUrl().

**10.**

Enter this URL into the input field, and click the shorten button in the web page.

https://medium.com/@codecademy/breaking-the-coding-language-barrier-bf24652c3c60

Notice the object that came back!

Now replace renderRawResponse(xhr.response) with renderResponse(xhr.response). Run the code.

Paste the URL again and click “Shorten”.

Isn’t it much cleaner?

Hint

Make sure you’re in shortenUrl().

The code for renderResult() function can viewed at **public/helperFunctions.js**.

// Information to reach API

const apiKey = '<Your API Key>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

const inputField = document.querySelector('#input');

const shortenButton = document.querySelector('#shorten');

const responseField = document.querySelector('#responseField');

// AJAX functions

const shortenUrl = () => {

  const urlToShorten = inputField.value;

  const data = JSON.stringify({destination: urlToShorten});

  const xhr = new XMLHttpRequest();

  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {

    if (xhr.readyState === XMLHttpRequest.DONE) {

      renderResponse(xhr.response);

    }

  }

  xhr.open('POST', url);

  xhr.setRequestHeader('Content-type', 'application/json');

  xhr.setRequestHeader('apikey', apiKey);

  xhr.send(data);

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

# Review Requests I

You’ve done an amazing job navigating through making XHR GET and POST requests! Take some time to review the core concepts before moving on to the next lesson.

1. JavaScript is the language of the web because of its asynchronous capabilities. AJAX, which stands for Asynchronous JavaScript and XML, is a set of tools that are used together to take advantage of JavaScript’s asynchronous capabilities.
2. There are many HTTP request methods, two of which are GET and POST.
3. GET requests only request information from other sources.
4. POST methods can introduce new information to other sources in addition to requesting it.
5. GET requests can be written using an XMLHttpRequest object and vanilla JavaScript.
6. POST requests can also be written using an XMLHttpRequest object and vanilla JavaScript.
7. Writing GET and POST requests with XHR objects and vanilla JavaScript requires constructing the XHR object using new, setting the responseType, creating a function that will handle the response object, and opening and sending the request.
8. To add a query string to a URL endpoint you can use ? and include a parameter.
9. To provide additional parameters, use & and then include a key-value pair, joined by =.
10. Determining how to correctly write the requests and how to properly implement them requires carefully reading the documentation of the API with which you’re working.

**Instructions**

Play around with the browser and code to make GET and POST requests. If you’re going to make POST requests, make sure you assign apiKey your Rebrandly API key and run the code!

If you want to challenge yourself:

* Build shortenUrl() or getSuggestions() from scratch.
* Manipulate the object that is returned to display something different in the browser.
* Use a different API to make a GET or POST request.
* Create query strings to yield different results.

// NOTE: wordSmith functions from lines 4 - 39

// NOTE: byteSize functions from lines 41 - 76 (remember to add your API key!)

// information to reach API

const dataMuseUrl = 'https://api.datamuse.com/words?';

const queryParams = 'rel\_jjb=';

// selecting page elements

const inputField = document.querySelector('#input');

const submit = document.querySelector('#submit');

const responseField = document.querySelector('#responseField');

// AJAX function

const getSuggestions = () => {

  const wordQuery = inputField.value;

  const endPoint = dataMuseUrl + queryParams + wordQuery;

  const xhr = new XMLHttpRequest();

  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {

    if (xhr.readyState === XMLHttpRequest.DONE) {

      renderWordResponse(xhr.response);

    }

  };

  xhr.open('GET', endPoint);

  xhr.send();

}

// clear previous results and display results to webpage

const displaySuggestions = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  };

  getSuggestions();

};

submit.addEventListener('click', displaySuggestions);

// information to reach Rebrandly API

const apiKey = '<Your API Key>';

const rebrandlyUrl = 'https://api.rebrandly.com/v1/links';

// element selector

const shortenButton = document.querySelector('#shorten');

// AJAX functions

const shortenUrl = () => {

  const urlToShorten = inputField.value;

  const data = JSON.stringify({destination: urlToShorten});

  const xhr = new XMLHttpRequest();

  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {

    if (xhr.readyState === XMLHttpRequest.DONE) {

      renderByteResponse(xhr.response);

    }

  };

  xhr.open('POST', rebrandlyUrl);

  xhr.setRequestHeader('Content-type', 'application/json');

  xhr.setRequestHeader('apikey', apiKey);

  xhr.send(data);

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  };

  shortenUrl();

};

shortenButton.addEventListener('click', displayShortUrl);

# Introduction to 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) .

**Instructions**

Click the button on the web page to see what the code in **main.js** does. Notice the usage of fetch(), async, and await.

We’ll also be working with JSON, so study the structure of the response on the web page!

Go to the next exercise to learn more about requests.

const jsonButton = document.querySelector('#generate');

const buttonContainer = document.querySelector('#buttonContainer');

const display = document.querySelector('#displayContainer');

const collection = ["Another", "More", "Next", "Continue", "Keep going", "Click me", "A new one"];

const generateJson = async () => {

  try {

    const response = await fetch('https://jsonplaceholder.typicode.com/users');

    if(response.ok){

      const jsonResponse = await response.json();

      renderResponse(jsonResponse);

      changeButton();

    }

  } catch(error) {

    console.log(error);

  }

};

const formatJson = (resJson) => {

  resJson = JSON.stringify(resJson);

  let counter = 0;

  return resJson.split('')

  .map(char => {

    switch (char) {

      case ',':

        return `,\n${' '.repeat(counter \* 2)}`;

      case '{':

        counter += 1;

        return `{\n${' '.repeat(counter \* 2)}`;

      case '}':

        counter -= 1;

        return `\n${' '.repeat(counter \* 2)}}`;

      default:

        return char;

    }

  })

  .join('');

};

const renderResponse = (jsonResponse) => {

  const jsonSelection = Math.floor(Math.random() \* 10);

  display.innerHTML = `<pre>${formatJson(jsonResponse[jsonSelection])}</pre>`;

};

const changeButton = () => {

  const newText = Math.floor(Math.random() \* 7);

  jsonButton.innerHTML = `${collection[newText]}!`;

};

jsonButton.addEventListener('click', generateJson);

# 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)

**Instructions**

**1.**

First, call the fetch() function and pass it this URL as a string:

https://api-to-call.com/endpoint

This first argument determines the endpoint of the request.

Stuck? Get a hint

**2.**

Chain a .then() method to the end of the fetch() function and pass it the success callback arrow function as its first argument. The success callback function takes one parameter, response.

.then() will fire only after the promise status of fetch() has been resolved.

Hint

The syntax for chaining a .then() looks like:

fetch('https://api-to-call.com/endpoint').then(response => {});

The callback function will be called with the response once one is received.

**3.**

Inside of the response callback function, check the ok property of response inside of a conditional statement. In the code block of the conditional statement, return response.json().

The reason we’re testing the ok property of the response object is that it will be a Boolean value. If there were no errors response.ok will be true and then your code will return response.json().

Hint

Your code will look like:

fetch('https://api-to-call.com/endpoint').then(response => {

if (response.ok) {

return response.json();

}

});

**4.**

Below the curly braces of the conditional statement, create a new error with this code:

throw new Error('Request failed!');

Your code will throw this error when response.ok is falsy.

Hint

Your code will look like:

fetch('https://api-to-call.com/endpoint').then(response => {

// Conditional statement for `response.ok`

throw new Error('Request failed!');

});

**5.**

Add a second argument to .then(), it will be an arrow function that will handle our failures. Separate the first callback function from the second with a comma. The second callback function takes a single parameter, networkError.

In the code block of the second callback function, log networkError.message to the console.

If we could not reach the endpoint at all, e.g., the server is down, then we would get this networkError.

Hint

Your complete code will look like:

fetch('https://api-to-call.com/endpoint').then(response => {

// Code to handle a successful response

}, networkError => {

console.log(networkError.message);

})

**6.**

Chain another .then() method to the end of the first .then() method.

Pass the new .then() method a callback function that takes jsonResponse as its parameter and return jsonResponse.

The second .then()‘s success callback won’t run until the previous .then() method has finished running. It will also not run if there was an error was thrown previously.

Hint

Your complete code will look like:

fetch('https://api-to-call.com/endpoint').then(response => {

// Code to handle a successful response

}, networkError => {

// Code to handle a network error

}).then(jsonResponse => {

return jsonResponse;

})

fetch('https://api-to-call.com/endpoint').then(response => {

  if (response.ok) {

    return response.json();

  }

  throw new Error('Request failed!');

}, networkError => {

  console.log(networkError.message);

}).then(jsonResponse => {

  return jsonResponse;

});

# 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.

**Instructions**

**1.**

At the top of **main.js**, create a const called url. Assign url to the following URL as a string:

https://api.datamuse.com/words

Hint

Wrap the URL in quotes, this will be the URL of the API that you will be accessing.

**2.**

Below url, create another const and call it queryParams. Assign it a value of '?sl='

queryParams will be the start of your query string and will narrow your search to words that sounds like your word.

**3.**

Inside the getSuggestions() function, create a const called wordQuery and assign it inputField.value.

You’ll need wordQuery to store the value of what is being typed into the input field.

You will be working inside getSuggestions() for the remainder of this exercise.

Hint

Remember to write your line of code inside the code block for getSuggestions().

**4.**

Now it’s time to add a query string to the URL with all the necessary parameters.

Create another const called endpoint, assign it value of a string that is url, queryParams, and wordQuery concatenated in that order.

Hint

Make sure you’re still in the code block of getSuggestions(). You have a couple of ways to concatenate a string:

const word1 = 'hello'

const word2 = 'world!'

console.log(word1 + ' ' + word2) // 'hello world!'

// You can also interpolate it using a template literal, but remember to use backticks!

console.log(`${word1} ${word2}`) // ‘hello world!’

**5.**

Call the fetch() function and pass in endpoint as an argument. For this API to work within the provided browser, add {cache: 'no-cache'} as the second argument.

Hint

Make sure you’re in the code block of getSuggestions().

You do not need to save the return value of this function call.

**6.**

Chain a .then() method to the fetch() function. Pass it a success arrow callback function as an argument. The callback function should take response as its single parameter.

Hint

You will use anonymous arrow function as the callback. The syntax will look like:

fetch(endpoint, {cache: 'no-cache'}).then(response => {})

**7.**

Inside the success callback function, create a conditional statement that checks if the ok property of the response object evaluates to a truthy value. If so, call the function renderJsonResponse() and pass in response as an argument. Then, run your code.

Then type in a word to and click the submit button to view the JSON that came back. The status of the promise return from fetch() will be resolved.

The code for renderJsonResponse() can be viewed at **public/helperFunctions.js**.

Stuck? Get a hint

**8.**

Delete renderJsonResponse(response) and replace it with return response.json().

By returning response.json(), the next function that is .then() chained to it will receive a Promise with JSON data.

**9.**

Below the condition’s code block, add this code to raise an exception if the request failed: throw new Error('Request failed!');

Hint

This line of code is not part of the conditional statement to check response.ok. It goes under that block of code but still inside the .then() callback, e.g.:

// Previous code

if(response.ok){

return response.json()

}

throw new Error('Request failed!');

**10.**

Separate the success callback function and the error callback function with a comma. The error callback function will also be an arrow function that takes one parameter, networkError. In the code block of the arrow function, log networkError.message to the console.

Hint

The syntax will look like:

.then((response) => {

// Previous code

},(networkError) => {

// code goes here

});

// Information to reach API

const url = 'https://api.datamuse.com/words';

const queryParams = '?sl=';

// Selects page elements

const inputField = document.querySelector('#input');

const submit = document.querySelector('#submit');

const responseField = document.querySelector('#responseField');

// AJAX function

const getSuggestions = () => {

  const wordQuery = inputField.value;

  const endpoint = `${url}${queryParams}${wordQuery}`;

  fetch(endpoint, {cache: 'no-cache'}).then(response => {

    if (response.ok) {

      return response.json();

    }

    throw new Error('Request failed!');

  }, networkError => {

    console.log(networkError.message)

  })

}

// Clears previous results and display results to webpage

const displaySuggestions = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  getSuggestions();

};

submit.addEventListener('click', displaySuggestions);

**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!

**Instructions**

**1.**

At the end of the .then() method, chain another .then() method.

Pass .then() an anonymous arrow callback function that takes jsonResponse as its single parameter.

Hint

The syntax for your code will look like:

fetch(url, {cache: 'no-cache'}).then(

// Previous code

}).then(jsonResponse => {} );

**2.**

Inside the callback function, call the function renderRawResponse() and pass in jsonResponse as an argument. Run the code.

In the input field, you can type in a word and click the submit button.

If all went well, you should see an array of words that the Datamuse API responded with!

You can view the purpose of renderRawResponse at **public/helperFunctions.js**.

Hint

Make sure you are in the code block of the second .then()‘s success callback function.

**3.**

Time to clean up that response a bit. Delete renderRawResponse(jsonResponse) and replace it with renderResponse(jsonResponse).

Run your code.

Try the webpage again with another word!

Hint

Remember to remove renderRawReponse(jsonResponse). This code goes inside the second .then() callback function’s code block.

// Information to reach API

const url = 'https://api.datamuse.com/words';

const queryParams = '?sl=';

// Selects page elements

const inputField = document.querySelector('#input');

const submit = document.querySelector('#submit');

const responseField = document.querySelector('#responseField');

// AJAX function

const getSuggestions = () => {

  const wordQuery = inputField.value;

  const endpoint = `${url}${queryParams}${wordQuery}`;

  fetch(endpoint, {cache: 'no-cache'}).then(response => {

    if (response.ok) {

      return response.json();

    }

    throw new Error('Request failed!');

  }, networkError => {

    console.log(networkError.message)

  }).then(jsonResponse => {

    renderResponse(jsonResponse);

  })

}

// Clears previous results and display results to webpage

const displaySuggestions = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  getSuggestions();

};

submit.addEventListener('click', displaySuggestions);

**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.

**Instructions**

Move on to the next exercise when you’re ready!



# 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)

**Instructions**

**1.**

Call the fetch() function, pass it the URL below as a string as its first argument, and pass it an empty object ({}) as its second argument.

https://api-to-call.com/endpoint

We’re calling fetch() and providing an endpoint. In the next step we’ll fill in the empty object with the necessary information.

**2.**

The settings object you passed to the fetch() function will contain two properties: method, with a value of 'POST', and body, with a value of JSON.stringify({id: '200'}).

This second argument determines that this request is a POST request and what information will be sent to the API.

Hint

The syntax for the fetch() call will look like:

fetch('https://api-to-call.com/endpoint', {

method: 'POST',

body: JSON.stringify({id: '200'})

})

**3.**

Chain a .then() method to the fetch() function and pass it the success callback function as its first parameter. Pass in response as an argument for the callback function. Leave the code block of the callback function empty for now.

The code inside .then() will execute when the promise returned from fetch() is resolved.

Hint

The syntax will look like:

fetch('https://api-to-call.com/endpoint').then(response => {});

**4.**

Inside the callback function’s code block, check the ok property of the response object inside of a conditional statement. In the code block of the conditional statement, return response.json().

When returned, this information will be passed on to the next .then() callback function.

Hint

To access the ok property of response use dot notation like so: response.ok. Then use an if conditional statement to execute code if the ok property is truthy. The syntax would look something like:

if(response.ok) {

return response.json()

}

**5.**

Below the curly braces of the conditional statement, create a new error with this code:

throw new Error('Request failed!');

This error will be raised if we get back some status error.

Hint

Make sure you’re outside of the conditional statement but still within the callback function.

**6.**

Create the failure callback function. This function takes a single parameter, networkError. Separate the first callback function from the second with a comma. This function is still inside of the .then() method.

In the code block of the function you just made, log networkError.message to the console.

Hint

This error will be raised when there is a problem with the network. For instance, the network could be down or experiencing heavy traffic and can’t provide service to the user. Your code should resemble:

fetch('https://api-to-call.com/endpoint').then(response => {

// Code for success callback

}, networkError => {

console.log(networkError.message);

})

**7.**

Chain another .then() method to the end of the first .then() method.

In the new .then() method, create an arrow callback function that takes jsonResponse as its parameter.

Then in the code block return jsonResponse.

The purpose of this step is to view the JSON that was returned from the previous .then().

Hint

Your code should resemble:

fetch('https://api-to-call.com/endpoint').then(response => {

// Code for success callback

}, networkError => {

// Code for failure callback

}).then(jsonResponse => {

return jsonResponse;

})

fetch('https://api-to-call.com/endpoint', {

  method: 'POST',

  body: JSON.stringify({id: "200"})

}).then(response => {

  if(response.ok){

    return response.json();

  }

  throw new Error('Request failed!');

}, networkError => {

  console.log(networkError.message);

}).then(jsonResponse => {

  console.log(jsonResponse);

})

# 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!

**Instructions**

**1.**

Assign apiKey to your Rebrandly API key as a string.

If you do not assign the correct key, you will not see the proper results in the steps afterwards.

Stuck? Get a hint

**2.**

Inside the code block of shortenUrl(), create a const named urlToShorten and assign it inputField.value. urlToShorten will keep the value of what is being typed into the input field.

Please note, you will be working inside shortenUrl() for the remainder of this exercise.

Hint

Make sure you’re working inside shortenUrl().

**3.**

Underneath urlToShorten, create another const named data, and assign it to the result of calling the method JSON.stringify() with {destination: urlToShorten} as an argument.

The reason for creating data is to prepare the information needed to send in the body.

Hint

First create a const named data

const data

Then assign data to the value of the result of calling JSON.stringify() and passing in {destination: urlToShorten}

const data = JSON.stringify({destination: urlToShorten})

**4.**

Below data, call the fetch() function. Pass it url as its first argument and an empty object as its second argument.

Hint

The syntax for this step would look something like:

fetch(url, {})

**5.**

Now it’s time to add some properties to the empty object that you just created. Create a property with the key method and the value 'POST'.

Hint

Make sure you’re in the empty object you created in the previous step.

An object with a single property (key-value pair) should look like this:

{ exampleKey: 'example value' }

**6.**

In the same object, create another property. The key for this property is headers and the value will be another object.

Assign headers the value of another object listed below:

{

'Content-type': 'application/json',

'apikey': apiKey

}

Hint

Separate the properties using commas.

In this step you have to create an object inside an object! Take a look at the following example:

let someObject = {

key1: value1,

key2: {

insideKey2: valueInsideKey2,

alsoInsideKey2: value2InsideKey2

}

}

**7.**

In that same object that has the properties method and headers, add another property. The key is named body and the value will be data.

Setting up this information now will make chaining .then() in the next exercise much easier!

Hint

Make sure you’re adding it to the object that has method and headers. You’re not in the headers object.

Also, check to see that you’ve separated the properties with commas. In this step, you’re creating another key-value pair.

The syntax for the object will look similar to:

let someObject = {

method: value1,

headers: {

insideKey2: valueInsideKey2,

alsoInsideKey2: value2InsideKey2

},

key3: value3

}

// Information to reach API

const apiKey = '<Your API Key>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

const inputField = document.querySelector('#input');

const shortenButton = document.querySelector('#shorten');

const responseField = document.querySelector('#responseField');

// AJAX functions

const shortenUrl = () => {

  const urlToShorten = inputField.value;

  const data = JSON.stringify({destination: urlToShorten})

  fetch(url, {

    method: 'POST',

    headers: {

      'Content-type': 'application/json',

      'apikey': apiKey

    },

    body: data

  })

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild)

  }

  shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

**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!

**Instructions**

**1.**

Chain a .then() method to the end of the fetch() function you wrote in the previous exercise. As its first argument, pass it an arrow function as a callback that takes response as its single parameter.

Hint

The syntax for chaining a .then() will look something like:

fetch(url, {

property: value,

property: value

}).then(parameter => {});

**2.**

Inside the block of the function you made in Step 1, use a conditional statement to check the value of the ok property of response. If it evaluates to a truthy value, call renderJsonResponse() and pass in response.

Run the code.

Now, if you post a URL, you should see the object that was sent back!

renderJsonResponse() is a helper function found in **public/helperFunctions.js**.

**3.**

Great, now that you see the raw object, you will need to pass the JSON to the next .then(). Delete renderJsonResponse(response) and replace it with return response.json()

**4.**

Below the curly braces of the conditional statement, throw a new Error in case response.ok is falsy.

The message the error should raise is ‘Request failed!’.

Hint

The syntax to raise throw a new Error and include ‘Request failed!’ as a message is below:

throw new Error('Request failed!');

**5.**

Outside of the code block from the first callback function you wrote, add another arrow callback function that takes networkError as a single parameter.

console.log() the networkError.message inside of the callback function you just wrote.

By adding this second callback, you’re safeguarding yourself in the rare event that the network returns an error!

Hint

The syntax will look like:

fetch(/\* previous code\*/).then((response) => {

// Previous code

},(networkError) => {

// code goes here

});

// Information to reach API

const apiKey = '<Your API Key>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

const inputField = document.querySelector('#input');

const shortenButton = document.querySelector('#shorten');

const responseField = document.querySelector('#responseField');

// AJAX functions

const shortenUrl = () => {

  const urlToShorten = inputField.value;

  const data = JSON.stringify({destination: urlToShorten})

  fetch(url, {

    method: 'POST',

    headers: {

      'Content-type': 'application/json',

      'apikey': apiKey

    },

    body: data

  }).then(response => {

    if (response.ok) {

      return response.json();

    }

    throw new Error('Request failed!');

  }, networkError => {

    console.log(networkError.message)

  })

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild)

  }

  shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

**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!

**Instructions**

**1.**

Chain a .then() method to the end of the original .then() method.

Hint

Make sure you’re chaining the second .then() to the original .then(); you’ll have to be careful about checking that parentheses match.

The syntax will resemble:

.then(response => {

//Previous code

}, networkError => {

// Previous code

).then();

**2.**

Pass in an anonymous arrow function as an argument for .then(). This callback function will take jsonResponse as its single parameter.

Inside the callback function, call renderRawResponse() and pass in jsonResponse.

Run the code.

Put in any URL in the text field, and then click the shorten button on the webpage. Make sure you include the entire link, including ‘http://‘ or ‘https://‘.

You’ll see the JSON of the response sent back from the API.

The renderRawResponse() helper function can be viewed at **public/helperFunctions.js**.

Hint

By knowing the structure of this object that was sent back, you’re able to pull out the relevant information. In this case, the relevant information is value of the key called shortUrl. You can access that property by calling jsonReponse.shortUrl which is what happens in our helper function in the next step!

**3.**

Delete renderRawResponse(jsonResponse). In its place, call renderResponse() and passing in jsonResponse.

Run the code.

Put a URL into the text field again and then click the Shorten button on the web page. Notice the difference?

The renderResponse() helper function can be viewed at **public/helperFunctions.js**.

// Information to reach API

const apiKey = '<Your API Key>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

const inputField = document.querySelector('#input');

const shortenButton = document.querySelector('#shorten');

const responseField = document.querySelector('#responseField');

// AJAX functions

const shortenUrl = () => {

  const urlToShorten = inputField.value;

  const data = JSON.stringify({destination: urlToShorten})

  fetch(url, {

    method: 'POST',

    headers: {

      'Content-type': 'application/json',

      'apikey': apiKey

    },

    body: data

  }).then(response => {

    if (response.ok) {

      return response.json();

    }

    throw new Error('Request failed!');

  }, networkError => {

    console.log(networkError.message)

  })

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild)

  }

  shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

# 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 async function. 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 catch statement 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)

**Instructions**

**1.**

In **main.js**, create an arrow function using the async keyword and save it to a const getData.

The async keyword will ensure that the function returns a promise.

Check the hint for syntax help.

Hint

The syntax for using async in an arrow function is as follows:

const getData = async () => {};

**2.**

In the code block for getData, we should add a try statement with an empty code block.

Below the try statement’s code block, add a catch(error) statement.

The code in the try block will send a request and handle the response. The catch statement will then take care of an error if it is thrown.

Check the hint for syntax help.

Hint

The syntax will look like:

const getData = () => {

try {

// Code will eventually go here

} catch (error) {}

};

**3.**

Inside the catch code block, log error to the console.

Since this exercise is an example, we’re using console.log() to view the error. Generally, developers create a more sophisticated way of handling the error, like redirecting their users to another page, but logging is fine for us at the moment.

Stuck? Get a hint

**4.**

Now we have to consider what to do inside of the code block of the try statement. We should start by using the await keyword before calling fetch(). Pass in the following URL, as a string, to the function as its first argument:

https://api-to-call.com/endpoint

We’ll have to save the returned promise in a variable called response using the const keyword. response will save the the response of endpoint once that information has been sent back.

Check the hint for syntax help.

Hint

Pass in the URL as a string. The syntax will look like:

try {

const response = await fetch('https://api-to-call.com/endpoint')

} catch (error) {

//Code to handle errrors

}

**5.**

Under response, create a conditional statement that checks if the ok property of the response object evaluates to a truthy value.

Hint

The syntax will look like:

try {

const response = await fetch('https://api-to-call.com/endpoint');

if (response.ok) {

}

} catch (error) {

//Code to handle errrors

}

**6.**

Inside the code block of the conditional statement, await the resolution of calling the .json() method on response.

Save the returned object to a variable called jsonResponse using the keyword const.

Since .json() is an asynchronous method we have to await until the promise status is resolved. Then we store the value to know what data the JSON holds.

Hint

The syntax will look like:

const jsonResponse = await response.json();

**7.**

Return jsonResponse directly below the code you wrote in the previous step.

Normally, we’d want to use the information from jsonResponse in a different manner. In this exercise, we’re practicing how to write the boilerplate code.

**8.**

Below the conditional statement, throw a new Error. The message the error should raise is ‘Request failed!’.

Hint

The syntax will look like:

try {

const response = await fetch('https://api-to-call.com/endpoint');

if (response.ok) {

// Code to execute

}

throw new Error('Request failed!');

} catch (error) {

// Code to handle errrors

}

const getData = async () => {

  try {

    const response = await fetch('https://api-to-call.com/endpoint');

    if (response.ok) {

      const jsonResponse = await response.json();

      return jsonResponse;

    }

    throw new Error('Request failed!');

  } catch (error) {

    console.log(error);

  }

}

# 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/).

**Instructions**

**1.**

Under the comment “AJAX function”, create a new arrow function called getSuggestions() using the async keyword.

You’ll be coding inside the arrow function of const getSuggestions for the remainder of this exercise.

Hint

The syntax would look like:

const getSuggestions = async () => {}

**2.**

Inside the code block of the async arrow function, create a const variable named wordQuery and assign it inputField.value.

**3.**

Create another const called endpoint, assign it value of a string that is url, queryParams, and wordQuery concatenated in that order.

Hint

Make sure you’re still in the code block of getSuggestions(). You have a couple of ways to concatenate a string:

const word1 = ‘hello’;

const word2 = ‘world!’;

console.log(word1 + ' ' + word2);

// ‘hello world!’

You can also interpolate it using a template literal, but remember to use backticks!

console.log(`${word1} ${word2}`);

// ‘hello world!’

**4.**

Add a try statement with an empty code block. Outside the code block for try, add a catch(error) statement with a code block that logs error to the console.

Hint

The syntax will look something like:

try {

// code goes here

} catch (error) {

console.log(error)

}

**5.**

Inside the try code block, using const, create a variable named response and assign it to await the result of calling fetch() with endpoint as the first argument. For this API to work within the provided browser, add {cache: 'no-cache'} as the second argument.

Hint

First create a variable named response using const:

const response

Then assign response to await the resolution of fetch(endpoint, {cache: 'no-cache'})

const response = await fetch(endpoint, {cache: 'no-cache'})

**6.**

Below the variable response from the previous step, create a conditional statement that the checks if the ok property of the response evaluates to a truthy value.

Inside the code block of the conditional statement, await response.json() and save it to a variable called jsonResponse using the const keyword.

Hint

Use an if statement to check the ok property of response. Then inside the statement, create a const jsonResponse and assign it to await the resolution of response.json().

The general syntax will look like:

if(variable.property){

const someVariable = await variable.method()

}

**7.**

Call the function renderRawResponse() and pass in jsonResponse. Then, run the code.

In the response field, type in a word and click the submit button on the web page.

You should now see an array of objects that contain nouns that describe the word you typed in!

You can view the purpose of the renderRawResponse helper function at **public/helperFunctions.js**.

Hint

Make sure you’re still in the conditional statement, write your code under jsonResponse.

**8.**

Now that you can see the body of response, you should clean it up to display on the webpage.

Delete renderRawResponse(jsonResponse) and replace it with renderResponse(jsonResponse). Run the code. Then type in another word and click the submit button.

Great, now you have an organized list of words and you finished your GET request!

You can view the purpose of renderResponse function at **public/helperFunctions.js**.

// Information to reach API

const url = 'https://api.datamuse.com/words?';

const queryParams = 'rel\_jja=';

// Selecting page elements

const inputField = document.querySelector('#input');

const submit = document.querySelector('#submit');

const responseField = document.querySelector('#responseField');

// AJAX function

// Code goes here

const getSuggestions = async () => {

  const wordQuery = inputField.value;

  const endpoint = `${url}${queryParams}${wordQuery}`;

  try {

    const response = await fetch(endpoint, {cache: 'no-cache'});

    if(response.ok){

      const jsonResponse = await response.json();

      renderResponse(jsonResponse);

    }

  } catch (error) {

    console.log(error);

  }

}

// Clear previous results and display results to webpage

const displaySuggestions = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  getSuggestions();

}

submit.addEventListener('click', displaySuggestions);

# 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)

**Instructions**

**1.**

At the top of **main.js** create an async arrow function and save it to a const getData().

The async keyword will ensure that the function returns a promise.

Hint

To create an async arrow function use the following syntax:

const getData = async () => {};

**2.**

In the code block for getData, we should add a try statement with an empty code block.

In case things go wrong, we need some code to handle that. Below the try code block, add a catch statement and pass in error as an argument.

Then, in the catch statement code block, log error to the console.

Hint

The syntax will look something like:

try {

// Code to handle request

} catch (error) {

console.log(error)

}

**3.**

We now have to consider what we want to do inside of the try code block. Since we are making a POST request, we should start by using the await keyword before calling the fetch() function.

We will have to save the returned promise in a variable called response using the const keyword.

Hint

The syntax will look like:

try {

const response = await fetch()

} catch (error) {

// Code to handle error

}

**4.**

In the fetch() call that we just made, pass in the following URL to the function as a string for the first argument:

https://api-to-call.com/endpoint

Then as our second argument, let’s pass in an empty object for now.

Hint

Remember to wrap the URL in quotes.

**5.**

Let’s now fill in the request options in our second argument.

First, add the method property and the value is 'POST'. Then we have to include a body property and the value is JSON.stringify({id: 200}).

Remember to separate the properties with a comma.

Hint

The general syntax to create the properties will be:

const response = await fetch(url, {

property1: value1,

property2: value2

});

**6.**

After the code block of response, we should create an if statement that checks the ok property of the response object.

Inside the code block of the conditional statement, await the resolution of calling the .json() method on response. Save the returned object to a variable called jsonResponse using the keyword const.

Hint

The syntax will be:

if (response.ok) {

const jsonResponse = await response.json()

}

**7.**

Now that we have what we want, we should return jsonResponse directly below the code written in the previous step.

Like with previous boilerplate exercises, we’re practicing writing code. Normally, we would want to do more beyond this step of returning jsonResponse.

**8.**

Below the if conditional, throw a new Error() with the message 'Request failed!'

Nice work, didn’t that feel very similar to making a GET request?

Hint

The syntax will look like:

if(response.ok){

// Previous code

}

throw new Error('Request failed!');

Nice work! Didn’t that feel very similar to making a GET request?

const getData = async () => {

  try {

    const response = await fetch('https://api-to-call.com/endpoint', {

      method: 'POST',

      body: JSON.stringify({id: 200})

    })

    if(response.ok){

      const jsonResponse = await response.json();

      return jsonResponse;

    }

    throw new Error('Request failed!');

  } catch(error) {

    console.log(error);

  }

}

# 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!

**Instructions**

**1.**

At the top of **main.js**, assign apiKey to your Rebrandly API key.

**2.**

Under the comment “AJAX functions”, create a new arrow function and assign it to a const shortenUrl() using the async keyword.

Hint

The syntax will look like:

const shortenUrl = async () => {}

**3.**

Inside the code block of the arrow function of shortenUrl create two consts:

One named urlToShorten and assign it inputField.value.

The other named data and assign it the value of calling JSON.stringify() and passing in {destination: urlToShorten}.

Please note, we will be working inside shortenUrl() for the remainder of the exercise.

**4.**

Add a try statement and leave the code block empty for now. After the try code block, create a catch statement and pass in error.

In the code block of catch(error), log error to the console.

Hint

The syntax will look something like:

try {

// Empty for now

} catch (error) {

console.log(error)

}

**5.**

Inside the try code block, using const, create a variable named response and assign it to await the value of calling calling fetch().

Hint

Break this step into smaller bits. “using const, create a variable named response“

const response

“assign it to await the value of calling calling fetch()“

const response = await fetch();

**6.**

In the fetch() call, pass in url as the first argument and an empty object as the second argument.

In that empty object you just created. It will have the following three properties:

* method with a value of 'POST'
* body with a value of data
* headers with a value of the object below:
* {
* 'Content-type': 'application/json',
* 'apikey': apiKey

}

Hint

Remember to separate your properties using commas!

The basic structure of the object will look like:

{

key1: value1,

key2: value2,

key3: {

'Content-type’: 'application/json’,

'apikey': apiKey

}

}

**7.**

Below the variable response from the previous step, create a conditional statement that the checks if the ok property of response evaluates to a truthy value.

Hint

Make sure you’re not inside the code block of the 2nd argument for fetch(). This step is asking you to use an if statement to check response.ok The general syntax will look like:

const response = await fetch(/\*Code for fetch\*/)

if(response.ok){}

**8.**

Inside the code block of the conditional statement, await response.json() and save it to a variable called jsonResponse using the const keyword.

**9.**

Call the function renderRawResponse() and pass in jsonResponse. Then, run the code.

In the response field, you can paste in a URL and click the shorten button.

You should now see an object containing all the information the Rebrandly API sent back!

You can view the purpose of the renderRawResponse() helper function at **public/helperFunctions.js**.

**10.**

Now it’s time to clean up the response sent back.

Delete renderRawResponse(jsonResponse) and replace it with renderResponse(jsonResponse). Run the code. Then paste in the URL again and click the shorten button.

Notice the formatted response.

Great job using Rebrandly to shorten your URL!

You can view the purpose of the renderRawResponse() helper function at **public/helperFunctions.js**.

// information to reach API

const apiKey = '<Your API Key>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

const inputField = document.querySelector('#input');

const shortenButton = document.querySelector('#shorten');

const responseField = document.querySelector('#responseField');

// AJAX functions

// Code goes here

const shortenUrl = async () => {

  const urlToShorten = inputField.value;

  const data = JSON.stringify({destination: urlToShorten});

  try {

    const response = await fetch(url, {

      method: 'POST',

      body: data,

      headers: {

        'Content-type': 'application/json',

        'apikey': apiKey

      }

    });

    if(response.ok){

      const jsonResponse = await response.json();

      renderResponse(jsonResponse);

    }

  } catch (error) {

    console.log(error);

  }

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

# 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.

**Instructions**

Congratulations on learning all about AJAX requests using fetch(), async, and await! These concepts are fundamental and will help you develop more robust web apps!

Play around the with browser and code to make GET and POST requests. If you’re going to make POST requests, make sure you assign apiKey your Rebrandly API key and run the code!

If you want to challenge yourself:

* Rewrite the requests from scratch.
* Replace the helper methods with your own code.
* Use different APIs to make GET/POST requests using async/await.

// NOTE: wordSmith functions from lines 4 - 39

// NOTE: byteSize functions from lines 48 - 81 (remember to add your API key!)

// information to reach API

const dataMuseUrl = 'https://api.datamuse.com/words?';

const queryParams = 'rel\_jjb=';

// selecting page elements

const inputField = document.querySelector('#input');

const submit = document.querySelector('#submit');

const responseField = document.querySelector('#responseField');

// AJAX function

const getSuggestions = async () => {

  const wordQuery = inputField.value;

  const endpoint = dataMuseUrl + queryParams + wordQuery;

  try{

    const response =  await fetch(endpoint);

    if(response.ok){

      let jsonResponse = await response.json();

      renderWordResponse(jsonResponse);

    }

  }

  catch(error){

    console.log(error);

  }

}

// clear previous results and display results to webpage

const displaySuggestions = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  getSuggestions();

}

submit.addEventListener('click', displaySuggestions);

// information to reach Rebrandly API

const apiKey = '<Your API Key>';

const rebrandlyEndpoint = 'https://api.rebrandly.com/v1/links';

// element selector

const shortenButton = document.querySelector('#shorten');

// AJAX functions

const shortenUrl = async () =>{

  const urlToShorten = inputField.value;

  const data = JSON.stringify({destination: urlToShorten});

  try{

    const response =  await fetch(rebrandlyEndpoint, {

      method: 'POST',

      body: data,

      headers: {

        "Content-type": "application/json",

        'apikey': apiKey

      }

    })

    if(response.ok){

      const jsonResponse = await response.json();

      renderByteResponse(jsonResponse);

    }

  }

  catch(error){

    console.log(error);

  }

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

  event.preventDefault();

  while(responseField.firstChild){

    responseField.removeChild(responseField.firstChild);

  }

  shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);