Sending Data with Fetch

- Due No Due Date
- Points 1
- Submitting a website url

[(https://github.com/learn-co-curriculum/phase-1-sending-data-with-fetch)](https://github.com/learn-co-curriculum/phase-1-sending-data-with-fetch) [(https://github.com/learn-co-curriculum/phase-1-sending-data-with-fetch/issues/new)](https://github.com/learn-co-curriculum/phase-1-sending-data-with-fetch/issues/new)

Learning Goals

- Use `fetch()` to send data to a remote host
- Handle the response from a successful request
- Handle errors from an unsuccessful request

Introduction

If you think about it, `fetch()` is a little browser in your browser. You tell `fetch()` to go to a URL by passing it an argument, e.g. `fetch("https://flatironschool.com")` , and it makes a network request. You chain calls to `fetch()` with `then()` . Each `then()` call takes a callback function as its argument. Based on actions in the callback function, we can display or update content in the DOM.

This is a lot like browsing the web: you change the URL in the URL bar, or you follow a link, and those actions tell the browser to go somewhere else and get the data. A technical way to describe that is: "The browser implements an HTTP `GET` to retrieve the content at a URL." It's also 100% technically correct to say " `fetch()` uses an HTTP `GET` to retrieve the content specified by a URL."

The browser also provides a helpful model for understanding what *sending* data from the browser looks like. We know this as an HTML *form*. Technically speaking, HTML forms "use an HTTP `POST` to send content gathered in `<input>` elements to a specified URL." It's also 100% technically correct to say " `fetch()` uses an HTTP `POST` to send content gathered in a JavaScript `Object` ."

HTML forms are still widely used, but with `fetch()` , we have more detailed control of the request. Using `fetch()` , we can actually *override* the normal behavior of an HTML form, capture any user input, package it up with the appropriate request information and send it out.

Our focus in this lesson will be learning how to send data using `fetch()` .

Using JSON Server to Mimic a Backend Database

To start up JSON Server, run `json-server --watch db.json` in your terminal. **Note**: Running this command will instruct `json-server` to use a `db.json` file in your terminal's current directory, so make sure to run this command from the same directory as this lab.

Once the server is running, you'll see a list of available resource paths in the terminal:

```
Resources
  http://localhost:3000/dogs
  http://localhost:3000/cats
  http://localhost:3000/users
  http://localhost:3000/robots
```

These endpoints each provide different sets of data. Since it is mimicking a RESTful API, sending a request to '**http://localhost:3000/dogs (http://localhost:3000/dogs)**' will return all records in the database for dogs, while '**http://localhost:3000/dogs/1 (http://localhost:3000/dogs/1)**' will return the dog with the id of 1.

Some example data is already present, stored in `db.json`. If the JSON server is running, you can also visit any of the above resources in a browser to see the data.

The tests in this lab do not need JSON Server to be running, but if you would like to run tests while also running the server, open a second tab in your terminal.

> **Note**: For users of the **Live Server VSCode extension (https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer)**, if the page is reloading when you initiate a fetch request, you'll need to set up some additional configuration for Live Server to play nicely with `json-server`. Follow the steps in **this gist (https://gist.github.com/ihollander/cc5f36c6447d15dea6a16f68d82aacf7)** (you'll only need to do this once), then come back to this lesson.

Analyze Data Sent in an HTML Form

Let's take a look at an HTML `<form>` (see `sample_form.html` in this repo):

```
<form action="http://localhost:3000/dogs" method="POST">
  <label> Dog Name: <input type="text" name="dogName" id="dogName" /></label
  ><br />
  <label> Dog Breed: <input type="text" name="dogBreed" id="dogBreed" /></label
  ><br />
  <input type="submit" id="submit" value="Submit" />
</form>
```

When we use the `<form>` element's default POST behavior in combination with a backend server, the key components for sending the submitted data to the server are:

- The destination URL as defined in the `action` attribute of the `<form>` tag
- The HTTP verb to use as defined in the `method` attribute of the `<form>` tag
- The key / value data obtained from the inputs in the fields `dogName` and `dogBreed`

We should expect that our "mini-browser," `fetch()` , will need those same bits of information in order to send a Post request to the server.

> **Note**: with JSON Server and our HTML form, we already have what we need to submit our form the conventional way, without using JavaScript. To try this out, make sure the JSON server is running and open `sample_form.html` in the browser. If you enter a dog name and breed in the input fields and click "Submit," your information should successfully POST to the JSON server database, `db.json` .

Construct a POST Request Using `fetch()`

Sending a POST request with `fetch()` is more complicated than what we've seen up to this point. It still takes a `String` representing the destination URL as the first argument, as always. But as we will see below, `fetch()` can also take a JavaScript `Object` as the *second* argument. This `Object` can be given certain **properties** (https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch#Parameters) that can be used to change `fetch()` 's default behavior.

```
fetch(destinationURL, configurationObject);
```

The `configurationObject` contains three core components that are needed for standard POST requests: the HTTP verb, the headers, and the body.

Add the HTTP Verb

So far, comparing to an HTML form, we've only got the destination URL ('**http://localhost:3000/dogs** (http://localhost:3000/dogs)' in this case). The next thing we need to include is the HTTP verb. By default, the verb is GET, which is why we can send simple GET requests with *only* a destination URL. To tell `fetch()` that this is a POST request, we need to add a `method` property to our `configurationObject`:

```
const configurationObject = {
  method: "POST",
};
```

Add Headers

The second piece we need to include is some *metadata* about the actual data we want to send. This metadata is in the form of ***headers*** **(https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers)**. Headers are sent just ahead of the actual data payload of our POST request. They contain information about the data being sent.

One very common header is `"Content-Type"` **(https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type)**. `"Content-Type"` is used to indicate what format the data being sent is in. With JavaScript's `fetch()`, **JSON** **(https://www.json.org/)** is the most common format we will be using. We want to make sure that the destination of our POST request knows this. To do this, we'll include the `"Content-Type"` header:

```
const configurationObject = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
};
```

Each individual header is stored as a key/value pair inside an object. This object is assigned as the value of the `headers` property as seen above.

When sending data, the server at the destination URL will send back a response, often including data that the sender of the `fetch()` request might find useful. Just like `"Content-Type"` tells the destination server what type of data we're sending, it is also good practice to tell the server what data format we *accept* in return.

To do this, we add a second header, **`"Accept"`** **(https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept)**, and assign it to `"application/json"` as well:

```
const configurationObject = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
  },
};
```

There are many other **headers** **(https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers)** available for particular uses. Some are used to send credentials or user authentication keys. Others are used to send cookies containing user info. `"Content-Type"` and `"Accept"` are two that we'll see the most throughout the remainder of this course.

A server that expects requests with specific headers may reject requests that don't include those headers.

Add Data

We now have the destination URL, our HTTP verb, and headers that include information about the data we're sending. The last thing to add is the *data* itself.

Data being sent in `fetch()` must be stored in the `body` of the `configurationObject` :

```
const configurationObject = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json"
  },
  body: /* Your data goes here */
};
```

There is a catch here to be aware of — when data is exchanged between a client (your browser, for instance), and a server, the data is sent as *text*. Whatever data we're assigning to the `body` of our request needs to be a string.

Use `JSON.stringify()` to Convert Objects to Strings

When sending data using `fetch()` , we often send multiple pieces of information in one request. In our code, we often organize this information using objects. Consider the following object, for instance:

```
{
  dogName: "Byron",
  dogBreed: "Poodle"
}
```

This object contains two related pieces of information, a dog's name and breed. Let's say we want to send the data in this object to a server. We can't simply assign it to `body` , as it isn't a string. Instead, we convert it to JSON. The object above, converted to JSON would look like this:

```
"{"dogName":"Byron","dogBreed":"Poodle"}"
```

Here, using JSON has enabled us to preserve the key/value pairs of our object within the string. When sent to a server, the server will be able to take this string and convert it back into key/value pairs in whatever language the server is written in.

Fortunately, JavaScript comes with a built-in method for converting objects to strings, `JSON.stringify()` . By passing an object in, `JSON.stringify()` will return a string, formatted and ready to send in our request:

```
const configurationObject = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
  },
  body: JSON.stringify({
    dogName: "Byron",
    dogBreed: "Poodle",
  }),
};
```

Send the POST Request

We've got all the pieces we need. Putting it all together, we get:

```
const configurationObject = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
  },
  body: JSON.stringify({
    dogName: "Byron",
    dogBreed: "Poodle",
  }),
};

fetch("http://localhost:3000/dogs", configurationObject);
```

With the JSON server running, if you open up `sample_form.html` or `index.html` , you can test out the code above in the console. Try it and take a look in `db.json` : you should see that Byron the Poodle has been successfully persisted to our database.

We can make our code a bit more general by splitting out the body of our request into a variable:

```
const formData = {
  dogName: "Byron",
  dogBreed: "Poodle",
};

const configurationObject = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
  },
  body: JSON.stringify(formData),
};

fetch("http://localhost:3000/dogs", configurationObject);
```

Using the `formData` and `configurationObject` variables helps make our code more readable and flexible, but, of course, we could instead just pass an anonymous object as the second argument to `fetch()` :

```
fetch("http://localhost:3000/dogs", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
  },
  body: JSON.stringify({
    dogName: "Byron",
    dogBreed: "Poodle",
  }),
});
```

All three approaches yield the same results!

Note: As a security precaution, most modern websites block the ability to use `fetch()` in console while on their website, so if you are testing out code in the browser, make sure to be on a page like `index.html` or `sample_form.html` .

Handling What Happens After

Just like when we use `fetch()` to send GET requests, we have to handle responses to `fetch()` . As mentioned before, servers will send a **Response** (https://developer.mozilla.org/en-US/docs/Web/API/Response) that might include useful information. To access this information, we use a series of calls to `then()` which are given function *callbacks*.

Building on the previous implementation we might write the following:

```
const formData = {
  dogName: "Byron",
  dogBreed: "Poodle",
};

const configurationObject = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
  },
  body: JSON.stringify(formData),
};

fetch("http://localhost:3000/dogs", configurationObject)
  .then(function (response) {
    return response.json();
  })
  .then(function (object) {
    console.log(object);
  });
```

Notice that the first `then()` is passed a callback function that takes in `response` as an argument. This is a **`Response`** (https://developer.mozilla.org/en-US/docs/Web/API/Response) object, representing what the destination server sent back to us. This object has a built-in method, `json()` , that converts the *body* of the response from JSON to a plain old JavaScript object. The result of `json()` is returned and made available in the *second* `then()` . In this example, whatever `response.json()` returns will be logged in `console.log(object)` .

Let's go ahead and send the example above to our JSON server in the console; once the request is successfully resolved, you should see the following log:

```
{dogName: "Byron", dogBreed: "Poodle", id: 6} // Your ID value may be different
```

The JSON server is sending back the data we sent, along with a new piece of data, an `id` , created by the server.

When Things Go Wrong: Using `catch()`

When something goes wrong in a `fetch()` request, JavaScript will look down the chain of `.then()` calls for something very similar to a `then()` called a `catch()` and, if it exists, execute it. This allows us to write code to "handle" the error. Say for instance, we forgot to add the HTTP verb to our POST request, and the `fetch()` defaults to GET. By including a `catch()` statement, JavaScript doesn't fail silently:

```
const formData = {
  dogName: "Byron",
  dogBreed: "Poodle",
};

// method: "POST" is missing from the object below
const configurationObject = {
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
  },
  body: JSON.stringify(formData),
};

fetch("http://localhost:3000/dogs", configurationObject)
  .then(function (response) {
    return response.json();
  })
  .then(function (object) {
    console.log(object);
  })
  .catch(function (error) {
    alert("Bad things! Ragnarők!");
    console.log(error.message);
  });
```

If you try the code above in the console from `index.html` or `sample_form.html` , you should receive an alert window pop-up and a logged message:

```
Failed to execute 'fetch' on 'Window': Request with GET/HEAD method cannot have body.
```

While `catch()` may not stop *all* silent errors, it is useful to have as a way to gracefully handle unexpected results. We can use it, for instance, to display a message in the DOM for a user, rather than leave them with nothing.

Challenge

It's time to practice writing your own POST request using `fetch()` . In `index.js` , write a function, `submitData` , that takes two strings as arguments, one representing a user's name and the other representing a user's email.

The first two tests mirror the behavior of the JSON server. As you write your solution, keep the server running to test your code. Open `index.html` in a browser to gain access to your `submitData` function in console.

Note: The tests in this lab need access to the `fetch()` request inside `submitData` . In order to give them access, write your solution so that `submitData` *returns* the `fetch()` . This will not change the behavior of your `fetch()` .

Test 1 - Send Data

In `submitData` , write a valid POST request to `http://localhost:3000/users` using `fetch()` . This request should include:

- The destination URL

- Headers for 'Content-Type' and 'Accept', both set to 'application/json'
- A body with the name and email passed in as arguments to `submitData` . These should be assigned to `name` and `email` keys within an object. This object should then be stringified.

Test 2 - Handle the Response

On a successful POST request, expect the server to respond with a `Response` (https://developer.mozilla.org/en-US/docs/Web/API/Response) object. Just like we saw earlier in the dog example, the `body` property of this response will contain the data from the POST request along with a newly assigned *id*.

Use a `then()` call to access the `Response` object and use its built-in `json()` method to parse the contents of the `body` property. Use a *second* `then()` to access this newly converted object. From this object, find the new id and append this value to the DOM.

If JSON Server is running and `index.html` is open in the browser, you can test your code in the console: calling `submitData()` in the console should cause an id number to appear on the page.

Test 3 - Handle Errors

For this final test, after the two `then()` calls on your `fetch()` request, add a `catch()` .

When writing the callback function for your `catch()` , expect to receive an object on error with a property, `message` , containing info about what went wrong. Write code to append this message to the DOM when `catch()` is called.

Test 4 - Return the Fetch Chain

An amazing feature of `fetch()` is that if you *return* it, *other* functions can tack on *their own* `then()` and `catch()` calls. **For this lab, you will need to return the `fetch()` chain from our `submitData` function to ensure that the tests run correctly.**

Conclusion

In this lab, we learned how to use `fetch` requests to post data to a server. This allows us to override the default behavior of an HTML `<form>` element and have greater control over the behavior of our form when it is submitted. It also enables us to improve our users' experience by rendering content without reloading the page.

Specifically, we learned:

- In order to submit a `POST` request, we call `fetch()` and pass it two arguments: the URL we're submitting the request to, and an object containing details about the request.
- This 'configuration' object needs to contain three properties: the `method` ("POST" in this case), `headers` that provide some information about our request, and the `body` of the request (i.e., the content we want to post).
- The body must be "stringified" before it can be passed in the request.
- Just as with a `GET` request, the `POST` request returns a `Response` object that can be accessed and used to update the DOM using chained `then` calls.
- In addition to the `then`s, you can also chain a `catch()` function that will "handle" unsuccessful requests.

With this information, you can now use `fetch()` to send both `GET` and `POST` requests! This means we have the tools we need to stitch together server updates (reads **and** updates) with DOM updating and event handling. We're almost ready to build the "Simple Liker" from scratch!