Using fetch()

(https://github.com/learn-co-curriculum/phase-1-using-fetch) (https://github.com/learn-co-curriculum/phase-1-using-fetch/issues/new)

Learning Goals

- Explain how to fetch data with `fetch()`
- Working around backwards compatibility issues
- Identify examples of the AJAX technique on popular websites

Introduction

When it comes to making engaging web sites, we often find ourselves needing to send a lot of data (text, images, media, etc.) so that the page is exciting.

But browsers won't show anything until they've processed all of that data. As a result, they show nothing. The screen stays blank and users experience "waiting."

Too much waiting means visitors will click away and never come back. Web users expect sites to load quickly **and** to stay updated. Research shows that 40 percent of visitors to a website will leave if the site takes more than 3 seconds to load. Mobile users are even *less* patient.

To solve this problem and help provide lots of other really great features, we developed a technique called ***AJAX***.

In AJAX we:

1. Deliver an initial, engaging page using HTML and CSS which browsers render *quickly*
2. *Then* we use JavaScript to add more to the DOM, behind the scenes

AJAX relies on several technologies:

- Things called `Promise` s
- Things called `XMLHttpRequestObject` s
- A **serialization format** (https://en.wikipedia.org/wiki/Serialization) called JSON for "JavaScript Object Notation"

- **asynchronous Input / Output** (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing)
- **the event loop** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop)

Part of what makes AJAX complicated to learn is that to understand it *thoroughly*, you need to understand *all* these components. For the moment, however, we're going to gloss over all these pieces in this lesson. It just so happens that modern browsers have *abstracted* all those components into a single function called `fetch()`. While someone interviewing to be a front-end developer will be expected to be able to explain all those components above (which we *will* cover later), while we're getting the hang of things, we're going to simplify our task by using `fetch()`.

Let's learn to use `fetch()` to apply the AJAX technique: a way to load additional data *after* information is presented to the user.

Explain How to Fetch Data with `fetch()`

The `fetch()` function retrieves data. It's a global *method* on the `window` object. That means you can use it simply by calling `fetch()` and passing in a path to a resource as an argument. To use the data that is returned by the `fetch()`, we need to chain on the `then()` method. We can see what this looks like below:

```
fetch("string representing a URL to a data source")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    // Use the data from the response to do DOM manipulation
  });
```

Now let's add some multi-line (`/*...*/`) comments (which JavaScript will ignore) to describe what's happening:

```
/*
  Here we are calling `fetch()` and passing a URL to a data source as the
  argument. The function call returns a Promise object that represents what the data
  source sent back. It does *not* return the actual content. (More about this
  later.)
```

```
*/
fetch("string representing a URL to a data source")
  /*
    Next, we call the `then()` method on the Promise object returned by calling
    `fetch()`. `then()` takes one argument: a callback function.
    (More on Promises later!)

    Inside the callback function, we do whatever processing we need on the
    object, in this case, converting it from JSON using the built-in `json()`
    method. (Another commonly-used method is `text()`, which will convert the
    response into plain text.) The `json()` method returns a Promise, which we
    return from our callback function.

    Note that we *have to return* the content that we've gotten out of the
    response and converted from JSON in order to use the data in the next then()
    method call.

    This first callback function is usually only one line: returning the
    content from the response after converting it into the format we need.
  */
  .then(function (response) {
    return response.json();
  })
  /*
    This time, the `then()` method is receiving the object that we returned from the
    first call to `then()` (our parsed JSON object). We capture the object in the
    parameter `data` and pass it into a second callback function, where we will
    write code to do DOM manipulation using the data returned from the server
  */
  .then(function (data) {
    // Use the actual data to do DOM manipulation
  });
```

> **Top Tip:** As always, we can name the parameters being used in our callback functions anything we like, but you will often see `response` (or `resp`) and `data` used.

Filling Out the Example

Let's fill out our base skeleton.

First, we'll provide a `String` argument to `fetch()`. As it happens, `http://api.open-notify.org/astros.json` will provide a list of the humans in space. You can paste this URL into a browser tab and see that the data uses a JSON structure.

JSON is a way to send a collection of data in the internet, formatted as a `String`. It just so happens that this string is written in a way that would be valid JavaScript syntax for an `Object` instance. Thus the name "JavaScript Object Notation", or JSON ("jay-sawn"). Programmers find it very easy to think about JavaScript `Object`s, so they often send "stringified" versions of `Object`s as responses.

The `then()` method takes a callback function as an argument. Here is where you tell JavaScript to parse the network response, which is formatted as a special JSON string, into actual JavaScript objects. When you first start using `fetch()`, most of your first `then()`s are going have a callback function that looks like this:

```
function(response) {
  // take the response, which is a JSON-formatted **string**,
  // and parse it into an actual JavaScript **object**
  return response.json();
}
```

The final `then()` is when you actually get some data (the parsed object returned from the first `then()`) passed in. You can then do something with that data. The easiest options are:

- `alert()` the data
- `console.log()` the data
- hand the data off to another function.

We'll go for the `console.log()` approach:

```
function(data) {
  console.log(data)
}
```

> **STRETCH:** But you *should* be able to imagine that you could do some DOM manipulation instead.

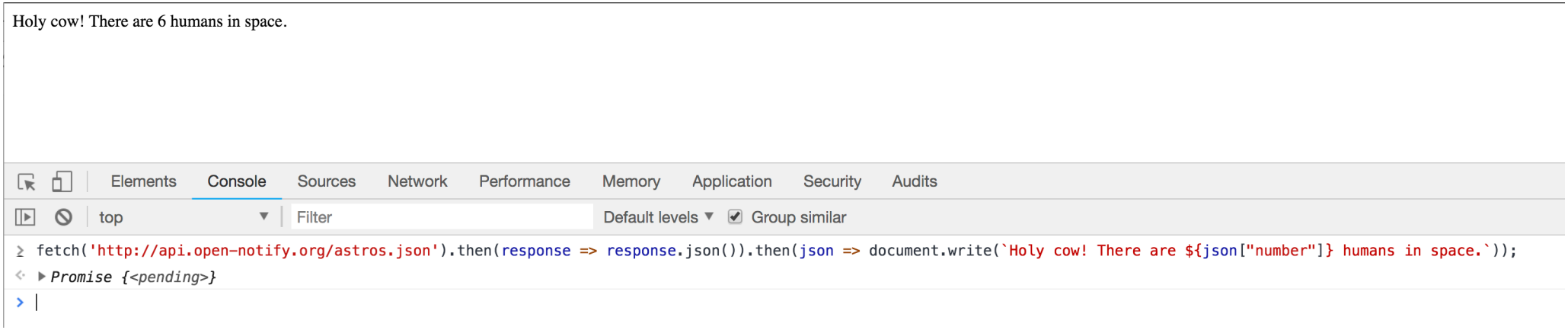
Here's a completed example:

```
fetch("http://api.open-notify.org/astros.json")
  .then(function (response) {
    console.log(response);
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  });
```

Let's perform a demonstration. Navigate to **http://open-notify.org** (http://open-notify.org) in an **incognito** tab. We need to go incognito to make sure that none of your browsing history interferes with this experiment.

Open up DevTools and paste the following into the console:

```
fetch("http://api.open-notify.org/astros.json")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
    console.log(`Holy cow! There are ${data["number"]} humans in space.`);
  });
```



You might notice in the DevTools console that this chained method call returned a `Promise`. We'll cover that later.

Working Around Backwards Compatibility Issues

As you can see, `fetch()` provides us with a short way to fetch and work with resources. In older code you might see `jquery.ajax` or `$.ajax` or an object called an `XMLHttpRequestObject`. You might also see libraries like `axios` used in newer code. These are distractions at this point in your education. After working with `fetch()` you'll be able to more easily integrate these special topics.

Identify Examples of the AJAX Technique on Popular Websites

The AJAX technique opens up a lot of uses!

- It allows us to pull in dynamic content. The same "framing" HTML page remains on screen for a cooking website. The recipe on display updates *without* page load. This approach was pioneered by GMail whose nav area is swapped for mail content swiftly — thanks to AJAX.
- It allows us to get data from multiple sources. We could make a website that displays the current weather forecast and the current price of bitcoin side by side! This approach is used by most sites to render ads. Your content loads while JavaScript gets the ad to show and injects it into your page (sometimes AJAX can be used in a way that we don't *entirely* like).

Conclusion

Many pages use AJAX to provide users fast and engaging sites. It's certainly not required in all sites. In fact, using it could be a step backward if simple HTML would suffice. However, as sites have more and more material, the AJAX technique is a great tool to have.

Using `fetch()`, we can include requests for data wherever we need to in our code. We can `fetch()` data on the click of a button or the expansion of an accordion display. There are many older methods for fetching data, but `fetch()` is the future.

Resources

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