Lesson Tuesday

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / Fetch API

Text

Now that we have experience creating and using `XMLHttpRequest` objects to make API calls, let's simplify our code even further. While `XMLHttpRequest` objects get the job done, there are other methods that use `XMLHttpRequest` objects under the hood while making our lives easier as developers.

In this lesson, we'll focus on the Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API), a Web API that will allow us to make a basic API call with a single line of code.

Note that the Fetch API is called an API because it provides a simple interface we can use in our applications. Remember, that's all an API is: an application programming interface. It is *not* called the Fetch API because it is used to make API calls.

Before we get into the details of the Fetch API, we should pause and remind ourselves of a few best practices. Now that we're using a new tool to make API calls, we'll also be using new properties and methods to handle errors and process the API response that won't be the same as with the `XMLHttpRequest` object. This lesson is meant to familiarize you with these new features. However, you

should be sure to do your part to use breakpoints, the *Network* tab, or Postman to examine the response objects that are returned, and double check your error handling by actually causing errors! Remember to try out these three test cases:

- A bogus input to cause a 404 Not Found
- A bad API key to get a 401 Unauthorized
- A bad API request URL to cause a 400 Bad Request

Using the Fetch API

Here's a basic GET request to the OpenWeather API with `fetch()`:

```
fetch(`http://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${process.env.API_KEY}`);
```

That's it.

What's special about the `fetch()` method is that it returns a promise. In other words, we can use `fetch()` instead of manually creating both promises and `XMLHttpRequest` objects. That is very cool!

Because the `fetch()` method returns a promise, we can use `Promise.prototype.then()` to handle the response.

```
fetch(`http://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${process.env.API_KEY}`);
  .then(function(response) {
    console.log(respones);
  });
```

This is so much easier, though there will be a bit of a tradeoff when it comes to handling the response.

Adding `fetch()` to our OpenWeather API Project

We'll learn how to use the Fetch API by adding it to our OpenWeather API project. Then at the end of the lesson, we'll share the completed code.

Let's start by updating our `weather-service.js` file:

src/weather-service.js

```
export default class WeatherService {
  static getWeather(city) {
    return fetch(`http://api.openweathermap.org/data/2.5/we
ather?q=${city}&appid=${process.env.API_KEY}`)
      .then(function(response) {
        if (!response.ok) {
          const errorMessage = `${response.status} ${respon
se.statusText}`;
          throw new Error(errorMessage);
        } else {
          return response.json();
        }
      })
      .catch(function(error) {
        return error;
      });
  }
}
```

Remember that our function needs to return a promise. Since `fetch()` itself returns a promise, we can just return that with the first line `return fetch(...);`.

That's the easy part.

Error Handling

However, there's a bit of a tradeoff when handling the `fetch()` response. `fetch()` almost never rejects a promise. There has to be a network error, like no internet connection, for the promise to be rejected. That's not very helpful for us, because we need some kind of error handling if the response is not 200 OK. When we used a promise without `fetch()`, we were able to reject responses that weren't 200 OK, but now we can't do that without taking some extra steps.

For that reason, we're updating our API call logic to include throwing an Error, and catching it! When we throw an error, any `.catch()` block will be triggered, and this will in turn trigger a rejection. Here's the code that throws the error:

```
if (!response.ok) {
  const errorMessage = `${response.status} ${response.statu
sText}`;
  throw new Error(errorMessage);
}
```

Fortunately, the response object that `fetch()` returns includes an `ok` property that we can use to determine whether the API call was successful or not. While the Fetch API uses an `XMLHttpRequest` object under the hood, they don't contain all of the same properties. We highly recommend adding a breakpoint to inspect the fetch `response` object.

If the API call has a 200 level status, the `ok` property will be set to `true`. If it doesn't, it will be set to `false`. So if `!response.ok`, our code will throw an error.

We create a custom error message using the fetch response object's `status` and `statusText` properties. These two properties are the exact same as the `XMLHttpRequest.status` and

`XMLHttpRequest.statusText` properties. Then, we pass our custom error message as the argument to the `Error` constructor.

Whenever an error is thrown, control immediately goes to the catch block, just like with `try...catch` blocks. In our catch block, we'll simply return the `error`:

```
.catch(function(error) {
  return error;
});
```

Here, the `error` parameter and returned value represents the error object we created earlier:

```
const errorMessage = `${response.status} ${response.statusT
ext}`;
throw new Error(errorMessage);
```

The error object is automatically passed to the catch block when we throw the error.

Then, when we return the `error` in our catch block:

```
.catch(function(error) {
  return error;
});
```

this value becomes available in `index.js`. We'll review the new code in `index.js` after we talk about the code that handles resolving the fetch request successfully.

Note that it's generally a good idea to include the `Promise.prototype.catch()` method when we are working with promises for the purpose of error handling. Even if we don't manually throw an error in our application, it's possible that our code will throw an error anyway, especially if there's a typo or something is broken. You can always return an `Error` object from a promise in the exact fashion we use above — as long as you write code to handle it as well. We'll be covering that in a moment.

Handling a Successful Fetch

We handle a successful API call in the `else` block:

src/weather-service.js

```
return fetch(`http://api.openweathermap.org/data/2.5/weathe
r?q=${city}&appid=${process.env.API_KEY}`)
  .then(function(response) {
    if (!response.ok) {
      const errorMessage = `${response.status} ${response.s
tatusText}`;
      throw new Error(errorMessage);
    } else {
      return response.json();
    }
  })
  .catch(....)
```

Specifically, we call the method `Response.json()` (https://developer.mozilla.org/en-US/docs/Web/API/Response/json). This code is new, and specific to the Fetch API. Let's break this down.

`fetch()` returns a promise in the form of a `Response` object (https://developer.mozilla.org/en-US/docs/Web/API/Response). It's this `Response` object that contains the properties `ok`, `status`, and `statusText`, as well as methods like `.json()`.

`Response.json()` specifically parses the response to JSON. We can't use `JSON.parse()` here, because the API response data is a stream that our code must read and convert to JSON. A stream of data is data that's broken into many small pieces that are processed individually. To learn more about streams, visit this documentation on MDN (https://developer.mozilla.org/en-US/docs/Web/API/Streams_API).

So, when we call `response.json()`, it's processing data that is streaming and being retrieved *now* but the data transfer won't be complete until *later*. This is more async code! Making the API request by calling `fetch()` is the first async operation, then reading the data stream from the response is the second async operation.

So our method returns a promise which returns a second promise. Even if the `catch` block is triggered, our method will still return a promise because the return of `Promise.prototype.then()` is itself a promise — even if we are just running sync code like our error handling.

Ultimately, our method will return one of two things:

- The data of a successful response (ideally), or
- An error message if the API call goes wrong.

Updating `index.js`

Next, we need to update our code in `index.js`.

src/index.js

```
import 'bootstrap';
import 'bootstrap/dist/css/bootstrap.min.css';
import './css/styles.css';
import WeatherService from './weather-service.js';

// Business Logic

function getWeather(city) {
  WeatherService.getWeather(city)
    .then(function(response) {
      if (response.main) {
        printElements(response, city);
      } else {
        printError(response, city);
      }
    });
}

// UI Logic

function printElements(response, city) {
  document.querySelector('#showResponse').innerText = `The
humidity in ${city} is ${response.main.humidity}%.
  The temperature in Kelvins is ${response.main.temp} degre
es.`;
}

function printError(error, city) {
  document.querySelector('#showResponse').innerText = `Ther
e was an error accessing the weather data for ${city}:
  ${error}.`;
}

function handleFormSubmission(event) {
  event.preventDefault();
  const city = document.querySelector('#location').value;
  document.querySelector('#location').value = null;
  getWeather(city);
}

window.addEventListener("load", function() {
```

```
  document.querySelector('form').addEventListener("submit",
handleFormSubmission);
});
```

Let's start by looking at the code directly related to the API call itself:

src/index.js

```
function getWeather() {
  WeatherService.getWeather(city)
    .then(function(response) {
      if (response.main) {
        printElements(response, city);
      } else {
        printError(response, city);
      }
    });
}
```

We start by calling `WeatherService.getWeather(city)`. Since this method returns a promise, we can immediately call `Promise.prototype.then()` below it without saving the returned promise to a variable.

Inside of the `Promise.prototype.then()`, we only have one callback function that's meant to be called when the API call resolves.

```
.then(function(response) {
  if (response.main) {
    printElements(response, city);
  } else {
    printError(response, city);
  }
});
```

If we wanted to include a function for a rejection, we'd add a second callback function:

```
.then(function(response) {
  if (response.main) {
    printElements(response, city);
  } else {
    printError(response, city);
  }
}, function(rejection) {
  // code to handle rejection
});
```

However, this second callback function would only be triggered if there is a network error, like no internet connection. And, the error that we throw and our catch block in the `WeatherService.getWeather()` method will already handle network connection errors, so we don't need another callback function to handle rejections.

So, what do we need to do in our callback function? We need to determine if our `response` is the OpenWeather API data or the error, and call either `printElements` or `printError`, depending. There's many ways to configure this; we've opted to use a conditional statement to see if `response.main` is truthy, and if so, call the `printElements` function. Remember that evaluating if a variable is truthy, means seeing if is exists and has any value at all. If it does, we know we're working with the OpenWeather API response data. In all other cases we call `printError`.

Notice that the arguments we pass into both `printElements` and `printError` have changed. Also, how we access that data in each function has changed:

src/index.js

```
function printElements(response, city) {
  document.querySelector('#showResponse').innerText = `The
humidity in ${city} is ${response.main.humidity}%.
  The temperature in Kelvins is ${response.main.temp} degre
es.`;
}

function printError(error, city) {
  document.querySelector('#showResponse').innerText = `Ther
e was an error accessing the weather data for ${city}:
  ${error}.`;
}
```

That's because we've both simplified our error handling, and our configuration with `fetch()` doesn't have the exact same limitations as we had when we used an `XMLHttpRequest` object with a promise. The lesson here is that what tool you choose to work with will impact how you pass data between function calls.

Returning More Complex Error Data

Notably, for our error message, we're only passing in the `status` and `statusText` details that we could get from the Fetch API's `Response` object. Remember this code?

src/weather-service.js

```
static getWeather(city) {
  return fetch(...)
    .then(function(response) {
      if (!response.ok) {
        // we created our custom error message here
        const errorMessage = `${response.status} ${respons
e.statusText}`;
        throw new Error(errorMessage);
      }
      ...
    })
    .catch(....);
}
```

Well, we can return the OpenWeather API's own custom error messages, too, by taking an extra step. Check out what this looks like:

src/weather-service.js

```
static getWeather(city) {
  return fetch(...)
    .then(function(response) {
        if (!response.ok) {
          return response.json()
            .then(function(apiErrorMessage) {
              const errorMessage = `${response.status} ${re
sponse.statusText}
              ${apiErrorMessage.message}`;
              throw new Error(errorMessage);
            });
        }
        ....
    })
    .catch(....);
}
```

Remember that the API response from OpenWeather, whether it's weather data or an error message, is in a stream. So to access the data, we first need to call `Response.json()` to turn it into JSON. `Response.json()` is async and returns a promise itself, so we call `Promise.prototype.then()` on it, passing in a callback function to handle the response. It's at this point we can craft a custom error message with both the data from the request, like `status` and `statusText`, and the API's response. Then, we throw the error.

If your head is spinning, that's a very normal reaction! This additional step adds more complexity and isn't necessary; it's completely option to include error messages from both the Fetch API's `Response` object and the API's own custom error messages. Instead, you can pick just one or the other.

Indentation and Spacing for Promises

Take a look at the `getWeather` function once more, and note the indentation:

src/index.js

```
function getWeather() {
  WeatherService.getWeather(city)
    .then(function(response) {
      if (response.main) {
        printElements(response, city);
      } else {
        printError(response, city);
      }
    });
}
```

We put `Promise.prototype.then()` on a new line. We did the same in our static method in `weather-service.js` as well. This spacing is common for readability, especially when chaining multiple promises

together. For instance, we might see something like this out in the wild:

```
// Pseudo-code!

promise
  .then(doSomething)
  .then(doSomethingElse)
  .then(doOneLastThing)
  .catch(takeCareOfThatError)
```

As we can see, putting each `.then()` on a newline makes our code easier to read. Finally, the catch block handles errors that occur anywhere inside this chain of promises. We will cover this process further in a future optional lesson on chaining promises (https://www.learnhowtoprogram.com/intermediate-javascript/asynchrony-and-apis/further-exploration-chaining-promises).

Final Thoughts

While the `fetch()` method is very useful, some developers prefer using `XMLHttpRequest` objects. You may choose to use either on this section's independent project. Ultimately, even if you prefer using `fetch()`, it's still important to have a good understanding of `XMLHttpRequest` objects and promises because `fetch()` relies on both.

Here's a quick guide to consider which one you might want to use in different situations:

- Use `XMLHttpRequest` objects and promises if you want full control over being able to reject a promise.
- Use `fetch()` if you don't want to worry about dealing with `XMLHttpRequest` objects and want any advantages that come

with streaming the data instead of waiting for the full response.

Just make sure you get plenty of opportunities to practice both! The more practice you get with different ways of dealing with asynchrony, the stronger you'll be as a coder.

Example GitHub Repo for API Project with `fetch()` (https://github.com/epicodus-lessons/section-6-js-api-call-with-webpack/tree/4_api_call_with_fetch)

The above link takes you to the branch called `4_api_call_with_fetch`.

Previous (/intermediate-javascript/asynchrony-and-apis/api-refactor-dino-ipsum)
Next (/intermediate-javascript/asynchrony-and-apis/async-and-await)

disable dark mode

(http://www.epicodus.com)