Lesson | Wednesday

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / Further Exploration: Chaining Promises

Text

Note: This lesson is a further exploration. You will not be expected to chain promises together for this section's independent project. However, we recommend that you follow along with this lesson closely because you'll learn more about coding best practices including using services, handling errors, and other things we can do to clean up our code.

In this lesson, we'll update our weather API project to chain together multiple promises. We've already chained together two promises with the Fetch API, but let's take it to another level by first making an API call to the OpenWeather API, waiting on the results of that call, and *then* using the result of that call to make a second API call to the Giphy API.

Processing data through multiple APIs is a real world use case and this is a great opportunity to practice chaining promises as well. In the process, we'll also have a chance to look at error handling for a more complex example, too.

Chaining Promises with Multiple API Calls

We'll refactor our OpenWeather API project starting from branch `4_api_call_with_fetch`, which just uses the Fetch API, but not `async` functions.

🗁 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)

Our updated application will display a weather description for the user and then show the user a GIF based on that description. There will be a lot of changes to the code, including a lot of refactoring.

We already have code for our `WeatherService`, so what should we do for our Giphy code? Do you think it would be better to create a separate service in a different file called `GiphyService` or would it be better to turn `WeatherService` into `ApiService` and put all the API calls in there?

Think about it for a moment... and think about why you arrived at your choice. You will have to make decisions about separating code *all of the time*. Sometimes there will be a right or a wrong approach. And sometimes both approaches will have their advantages and disadvantages. But regardless of the case, you should always be prepared to think through how you structure your code.

If you think the best approach is two separate services, you are correct! The Giphy API and the OpenWeather API do two separate things. We want to keep our code separate and modular. We might even want to use each of these APIs elsewhere in our application in ways that have nothing to do with each other.

Also, there is a trend in programming towards **microservices**. This is a software design pattern where applications are built around lots of smaller services that communicate with each other. In contrast, there's the **monolithic** approach which can have lots and lots of closely entangled code — which means that when things break, they really break.

Microservices allow code to be loosely coupled, which means every service isn't too dependent on other code. By separating out our API calls in two different services, they are fully decoupled and don't know about each other. If there's an error in one of our services, it won't break the other service (though it will certainly break any code that does depend on the broken service).

Adding `giphy-service.js`

Let's create a separate directory in `src` for `services` and add both our `weather-service.js` file and a new `giphy-service.js` file. VSCode should automatically offer to update the paths in any import statements that use `weather-service.js` in your application. Click *yes* to do so or manually update the paths as needed.

Here's the code for our new `GiphyService`. It will be very similar to code we've written previously for using the Fetch API with promises:

src/services/giphy-service.js

```
export default class GiphyService {
  static async getGif(query) {
    return fetch(`http://api.giphy.com/v1/gifs/search?q=${q
uery}&api_key=${process.env.GIPHY_API_KEY}&limit=5`)
      .then(function(response) {
        if (!response.ok) {
          const errorMessage = `${response.status} ${respon
se.statusText}`;
          throw new Error(errorMessage);
        }
        return response.json();
      })
      .catch(function(error) {
        return error;
      });
  }
}
```

Most of this code should look familiar but there are a couple of important things to note. First, we use a generic parameter called `query` in our static method. We could use this API call *anywhere* in our application for things other than weather, so we don't want to call it something constricting like `currentWeather`.

Next, we name our environmental variable `GIPHY_API_KEY`. We'll also update the environmental variable name for the OpenWeather API to `OPEN_WEATHER_API_KEY`. Now that we are working with multiple APIs in our application, we need to make sure the names for each are descriptive. Make sure to update your `.env` file accordingly.

Updating `weather-service.js`

Let's make sure that we update the name of the environmental variable in our `WeatherService` as well:

src/services/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.OPEN_WEATHER_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;
      });
  }
}
```

So now we have two services. Each returns either a promise that will resolve with API data *or*, if something goes wrong, an `Error` object.

Chaining Promises in `index.js`

Now we are ready to update `index.js` to handle our promises and chain them together. There will be quite a few updates to this file, including some refactoring of exiting code. We'll also be making some updates to HTML classes in `index.html` and our updated UI methods will reflect that.

src/index.js

```
// Import statements updated to reflect new paths and also
import GiphyService.
import 'bootstrap';
import 'bootstrap/dist/css/bootstrap.min.css';
import './css/styles.css';
import WeatherService from './services/weather-service.js';
import GiphyService from './services/giphy-service.js';

// Business Logic

// we update the name of this function
function getAPIData(city) {
  WeatherService.getWeather(city)
    .then(function(weatherResponse) {
      if (weatherResponse instanceof Error) {
        const errorMessage = `there was a problem accessing
the weather data from OpenWeather API for ${city}:
        ${weatherResponse.message}`;
        throw new Error(errorMessage);
      }
      const description = weatherResponse.weather[0].descri
ption;
      printWeather(description, city);
      return GiphyService.getGif(description);
    })
    .then(function(giphyResponse) {
      if (giphyResponse instanceof Error) {
        const errorMessage = `there was a problem accessing
the gif data from Giphy API:
        ${giphyResponse.message}.`;
        throw new Error(errorMessage);
      }
      displayGif(giphyResponse, city);
    })
    .catch(function(error) {
      printError(error);
    });
}

// UI Logic
```

```
// the parameter has changed for this function, as
// has the message it prints to the DOM
function printWeather(description, city) {
  document.querySelector('#weather-description').innerText
= `The weather in ${city} is ${description}.`;
}

// printError() is now much more simple, since we handle
// creating the error message in the getAPIData() function
function printError(error) {
  document.querySelector('#error').innerText = error;
}

// we have a new function that displays the gif
function displayGif(response, city) {
  const url = response.data[0].images.downsized.url;
  const img = document.createElement("img");
  img.src = url;
  img.alt = `${city} weather`;
  document.querySelector("#gif").append(img);
}

// we have a new function that clear previous results.
function clearResults() {
  document.querySelector("#gif").innerText = null;
  document.querySelector('#error').innerText = null;
  document.querySelector('#weather-description').innerText
= null;
}

function handleFormSubmission(event) {
  event.preventDefault();
  // we call our new function (below) to clear previous res
ults
  clearResults();
  const city = document.querySelector('#location').value;
  document.querySelector('#location').value = null;
  // we update the name of the function that makes the API
call
  getAPIData(city);
}
```

```
window.addEventListener("load", function() {
  document.querySelector('form').addEventListener("submit",
handleFormSubmission);
});
```

Let's start by focusing on the most important code in our new `getAPIData` function. In the process, we'll also explain our new UI functions as well.

src/index.js

```
function getAPIData(city) {
  WeatherService.getWeather(city)
    .then(function(weatherResponse) {
      if (weatherResponse instanceof Error) {
        const errorMessage = `there was a problem accessing
the weather data from OpenWeather API for ${city}:
        ${weatherResponse.message}`;
        throw new Error(errorMessage);
      }
      const description = weatherResponse.weather[0].descri
ption;
      printWeather(description, city);
      return GiphyService.getGif(description);
    })
    .then(function(giphyResponse) {
      if (giphyResponse instanceof Error) {
        const errorMessage = `there was a problem accessing
the gif data from Giphy API:
        ${giphyResponse.message}.`;
        throw new Error(errorMessage);
      }
      displayGif(giphyResponse, city);
    })
    .catch(function(error) {
      printError(error);
    });
}
```

We start by calling our `WeatherService` static method. Because this returns a promise, we can use `Promise.prototype.then()` with it. In fact, we chain two `.then()` methods, and then a `.catch()`:

- The first `Promise.prototype.then()` to handle the response from the fetch to OpenWeather, and initiating the next API call to Giphy.
- The second `Promise.prototype.then()` to handle the response from the fetch to Giphy.
- A `Promise.prototype.catch()` to handle any errors that occur from OpenWeather, Giphy, or a network error.

Let's first discuss the new error handling.

Error Handling

If there's an error processing our API call in the static method (`WeatherService.getWeather()` or `GifService.getGif`) and `response.ok` is set to false, we'll throw an error, and the `Promise.prototype.catch()` block will return that error to `index.js` so that we can eventually display it to the UI. While we could just directly display the error message to the UI in our API call service logic, that would be bad separation of logic.

Then, back in `index.js` in our first `Promise.prototype.then()`, we can check to see if an error has been returned to us with the following conditional:

```
if (weatherResponse instanceof Error) {
  // do something with error
}
```

As we discussed in JavaScript Exception Handling with `try...catch` (https://www.learnhowtoprogram.com/intermediate-javascript/asynchrony-and-apis/javascript-exception-handling-with-

try-catch), `instanceof` is actually an operator and it's very helpful for checking to see whether or not something is a certain type of object.

In this case, if `weatherResponse` is an instance of an `Error` object, we need to throw another error. This one will revert control to the *nearest* catch block, which actually comes at the end of our chained promise. That catch block will call a `printError()` function which will display an error for the user.

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

So here's what happens when the API call doesn't return a 200 response:

- In our static API call method, control switches over to a conditional which then throws an error.
- The catch block in our static method catches the error, then returns an `Error` object from the method.
- When the static method is complete, control switches over to the next `Promise.prototype.then()`, the first one in the `getAPIData` function in `index.js`. The method checks if `weatherResponse` is an instance of an `Error` object. Since it is, we throw *another* error which will then switch control to the catch block of our chained promises in the `getAPIData` function and pass along the error message.
- Finally, the catch block will call the `printError()` function, which handles printing the error message to the webpage.

If it seems like we are throwing this error all over the place, we are. But that's common in coding — think about a time when you've made an error in a JavaScript application that uses webpack. The

application will fail to compile and there will be a **stack trace**. It's often not just one error, but an entire cascade of errors that the first one triggers.

In this particular application, there are several advantages to the approach we're taking here.

First, we wouldn't want to call `printError()` in our API call services. Printing errors has to do with our UI and services shouldn't know or care about our UI. On the other hand, it makes complete sense for our service to either return data or throw an error. We could use this service anywhere in any application and handle the error as needed. For instance, if we were also using this service in a backend Node application, we wouldn't want to display an error to a user — instead, we might print the error message to a server error log.

Here's another big advantage: the chained promises in `index.js` are handling multiple API calls and we can use the same catch block to handle errors from either. We just need to have a slightly different error message so the user can see exactly what went wrong. In the case of the Weather API response, the message is:

```
there was a problem accessing the weather data from OpenWea
ther API for ${city}: ${weatherResponse.message}
```

In the case of the Giphy response, that message is:

```
there was a problem accessing the gif data from Giphy API:
${giphyResponse.message}.
```

If you want to see these error messages for yourself when you read the code, cause an error, like inputting a non-existent city.

So now we've set up our error handling so that messages related to the UI are in the correct place — and we can use the same catch block for errors related to both API calls, keeping our code DRY.

Successful Calls

Now let's take a look at what will happen if all goes well with our OpenWeather API call:

src/index.js

```
function getAPIData(city) {
  WeatherService.getWeather(city)
    .then(function(weatherResponse) {
      ...

      const description = weatherResponse.weather[0].descri
ption;
      printWeather(description, city);
      return GiphyService.getGif(description);
    })
    ...
}
```

First we parse the `description` property from the response data. Then we call the `printWeather` function to print the weather description for the user-inputted city. Why do it here instead of later? Well, there's no reason for users to wait until the Giphy API call is resolved for them to get *some* data. That's a huge advantage of asynchronous code and something you'll see often online — sites like Facebook and Twitter will give you some data *now* even as they are loading more data to show you *later*. We don't want to wait for everything to load all at once. Also, if the Giphy API call were to fail, we'd still get data from the OpenWeather API.

Now for the *big* gotcha that trips students up. `Promise.prototype.then()` is a method. We know that, right? Well, the basic rule about JavaScript methods and functions is that you

always need to return something or else their return will be `undefined`. This applies to `Promise.prototype.then()`, too, which is why we return the next API call to Giphy:

```
return GiphyService.getGif(description);
```

Because the syntax of chaining promises together looks confusing at first, it can be really easy to forget to add the `return`. **You *must* return a value from `Promise.prototype.then()` if you want to chain another promise to it.**

When we make the API call to Giphy, it goes through the same process as the API call to OpenWeather:

- If there's an error, our code will go through the same process as it does with the OpenWeather API.
- If the call is successful, we'll call the `displayGif` function.

```
function getAPIData(city) {
  WeatherService.getWeather(city)
    .then(function(weatherResponse) {
      ...
    })
    .then(function(giphyResponse) {
      if (giphyResponse instanceof Error) {
        const errorMessage = `there was a problem accessing
the gif data from Giphy API:
        ${giphyResponse.message}.`;
        throw new Error(errorMessage);
      }
      displayGif(giphyResponse, city);
    })
    ...
}
```

We don't need to return anything from the second `Promise.prototype.then()` method because there are no further promises chained to it.

In fact, this final method just has **side effects**. A method or function that alters something elsewhere in the code (instead of or in addition to returning a value) is said to have side effects. Side effects are common with functions related to the UI — though when it comes to business logic, they should be avoided, just like we did in our API service logic.

And that's it for the code. The most complex change is the error handling. And while it can seem daunting at first, if you think carefully about keeping code separate and using callbacks to make the code more modular, it's not so bad. We'll run into a lot more trouble (and generally bad code) if we just throw a lot of code inside `Promise.prototype.then()` without extracting as much as possible into separate functions.

HTML

One last thing: we've updated the HTML as well. There's really nothing to say about it other than the fact that we had to add some HTML tags and adjust come ids to account for handling GIFs:

src/index.html

```
<html lang="en-US">
<head>
  <title>Weather</title>
</head>
<body>
  <div class="container">
    <h1>Get Weather Conditions From Anywhere!</h1>
    <p>To get the current description of the weather condit
ions for a location, please enter the city name or the city
and state separated by a comma. Here are three examples:</p
>
    <pre>
      Portland
      Atlanta, Georgia
      cairo
    </pre>
    <p>We will also show you a gif that corresponds to the
weather's description.</p>
    <form>
      <label for="location">Enter a location:</label>
      <input id="location" type="text" name="location">
      <button type="submit" class="btn-success" id="weather
Location">Get Current Temperature and Humidity</button>
    </form>
    <p id="weather-description"></p>
    <p id="error"></p>
    <div id="gif"></div>
  </div>
</body>
</html>
```

And that's all of our updated code. Even though this lesson is optional and chaining promises isn't required for the independent project, we highly recommend trying to chain promises in your code. Doing so will give you a better understanding of promises, error handling, and other important JavaScript concepts.

Example GitHub Repo for API Project with Chained Promises (https://github.com/epicodus-lessons/section-6-js-api-call-with-webpack/tree/6_chained_promises)

The above link takes you to the branch called `6_chained_promises`.

Previous (/intermediate-javascript/asynchrony-and-apis/bike-index-cryptocurrency-analytics-app-api-of-choice-two-day-project-part-1)
Next (/intermediate-javascript/asynchrony-and-apis/pull-requests-and-submitting-great-work)

Lesson 28 of 33
Last updated more than 3 months ago.

disable dark mode

(http://www.epicodus.com)