Lesson Tuesday

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / Async and Await

Text

In this lesson, we'll cover async functions, a relatively new JavaScript feature that was added in ES 2017. (Remember that ES6 is ECMAScript 2015.) While you aren't required to use async functions for the independent project, they are a really useful piece of JavaScript functionality and definitely a tool you should know about and be familiar with.

An async function allows us to write asynchronous code as if it were synchronous. This can make our code more concise. And while promises go very well with API calls, there are plenty of other situations where using async functions will handle asynchronous operations better than a promise will. For example, if we wanted to do a series of five or six things in a row, many but not all of them async, code with promises would be very verbose while async functions would be easier to read and reason about.

In fact, even the code from the last lesson chains together two async operations. Throw another API call into the mix that also uses `fetch()` and we are talking about four async operations in a row when we consider the streaming data as well. It's very feasible that we'd do a few synchronous things in between(such as parsing data from the first API call).

`async` Functions

Before we update our weather API project to use async functions, let's take a look at an example. We can create an async function with the `async` keyword. Try the following code in the console:

```
> async function thisIsAsync() { return "This is async"; }
> thisIsAsync();
Promise {<resolved>: "This is async"}
```

All we did here is add the `async` keyword to a basic function that returns a string. When we call the function, it returns a resolved promise. As this example shows, async functions are using promises under the hood, too!

`async` and `await`

The real power of async functions lies with the `await` keyword. When `await` is used within an async function, our code will stop executing until the line of code that includes `await` is completed. Let's take a look at an example in the context of an API call that uses `fetch()`:

```
async function makeApiCall() {
  const response = await fetch("http://some-api-call.com");
  const jsonifiedResponse = await response.json();
  return jsonifiedResponse;
}
```

We start by adding the `async` keyword to our `makeApiCall()` function. Now we can use the `await` keyword as needed inside of the async function to wait for asynchronous actions to complete.

Note that we can't use the `await` keyword outside of an `async` function. If we try to, we'll get the following error: `Uncaught SyntaxError: await is only valid in async function`.

Next, we set the value of the `response` variable to be equal to the response of the API call. If we did this without the `await` keyword, `response` would be undefined. That's because the variable is assigned before the async `fetch()` call is complete.

Once we add the `await` keyword, though, the value of `response` won't be assigned until the `fetch()` call is resolved.

We also need to `await` the completion of the `Response.json()` method because it's an async operation, too. Once that's done, we're ready to return the final `jsonifiedResponse`. The `await` keyword has given us the ability to write asynchronous code as if it were synchronous. This results in very concise code that is easy to read and understand.

Adding `async` and `await` to our OpenWeather API Project

Now that we've looked at a basic example, let's update our weather application to use `async` and `await`.

We'll start by updating our service logic:

src/weather-service.js

```
export default class WeatherService {
  static async getWeather(city) {
    try {
      const response = await fetch(`http://api.openweatherm
ap.org/data/2.5/weather?q=${city}&appid=${process.env.API_K
EY}`);
      const jsonifiedResponse = await response.json();
      if (!response.ok) {
        const errorMessage = `${response.status} ${respons
e.statusText}
        ${jsonifiedResponse.message}`;
        throw new Error(errorMessage);
      }
      return jsonifiedResponse;
    } catch(error) {
      return error;
    }
  }
}
```

First, we update our static `getWeather` method to be async. Next, we need to wrap our code in a `try...catch` block. We use a `try...catch` block to handle errors, because `async` functions don't have the capability of resolving or rejecting the promise it uses under the hood.

Inside the `try` block, we'll make our API call with `fetch()` and then use the `await` keyword to wait for the API call to complete. Then we call `reponse.json()`, and `await` this action. We do this next so that we can use the API response for weather data and for its error messages. The rest of the service code has only minor changes to variable names — we still throw an error if the `response` property isn't `ok`, and if the response property is `ok`, we return the "jsonified" response.

Next, let's take a look at our user interface logic:

src/index.js

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

// Business Logic

async function getWeather(city) {
  const response = await WeatherService.getWeather(city);
  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);
});
```

All of the updates we've made are within the `getWeather` function. Now this function is `async`, and inside we `await` the response of the API call.

```
async function getWeather(city) {
  const response = await WeatherService.getWeather(city);
  if (response.main) {
    printElements(response, city);
  } else {
    printError(response, city);
  }
}
```

Then, instead of using `Promise.prototype.then()`, we directly run the `response` variable through a conditional to determine whether to print the weather data or print an error message.

And that's it!

Async functions can be a concise and elegant way to write code. However, it's important to think carefully about the code we put inside async functions. We are essentially forcing any code inside an async function to run synchronously. That means any asynchronous actions will block the flow of code until it is complete. If we put too much code inside an async function, we risk overriding JavaScript's non-blocking capabilities, potentially making our code more inefficient. That's not an issue in the code above but it would be if we wrapped our entire application in an async function.

You are welcome (but not required) to use async functions for this section's independent project. You are also encouraged to practice writing async functions during the classwork. However, make sure

you have a clear understanding of promises first. After all, async functions use promises under the hood.

Example GitHub Repo for API Project with `fetch()` (https://github.com/epicodus-lessons/section-6-js-api-call-with-webpack/tree/5_fetch_with_async_await)

The above link takes you to the branch called `5_fetch_with_async_await`.

Previous (/intermediate-javascript/asynchrony-and-apis/fetch-api)
Next (/intermediate-javascript/asynchrony-and-apis/further-exploration-iifes)

Lesson 25 of 33
Last updated more than 3 months ago.

disable dark mode

(http://www.epicodus.com)