Lesson Monday

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / Separating Promise Logic

Text

Now that we've learned about static methods, let's refactor our weather API application to use one. In the process, we'll separate our API call business logic into a separate file.

Here's the code so far on the branch called `2_xhr_api_call_with_promises`:

📂**Example GitHub Repo for API Project with Promises (https://github.com/epicodus-lessons/section-6-js-api-call-with-webpack/tree/2_xhr_api_call_with_promises)**

We'll just add one additional file to hold our API logic. We'll call this file `weather-service.js` and the file will hold a class called `WeatherService`.

Why are we using the term **service** here? A service is a piece of reusable code that can be shared across an application. In this case, we are separating out our API call so we can make this code available wherever it's needed. In a very large application, we might need to include our service in many different places. That isn't the case here, but using this design pattern will make our application more scalable.

Let's take a look at the code in this file:

src/weather-service.js

```
export default class WeatherService {
  static getWeather(city) {
    return new Promise(function(resolve, reject) {
      let request = new XMLHttpRequest();
      const url = `http://api.openweathermap.org/data/2.5/w
eather?q=${city}&appid=${process.env.API_KEY}`;
      request.addEventListener("loadend", function() {
        const response = JSON.parse(this.responseText);
        if (this.status === 200) {
          resolve([response, city]);
        } else {
          reject([this, response, city]);
        }
      });
      request.open("GET", url, true);
      request.send();
    });
  }
}
```

We `export a default` class because we will only be exporting `WeatherService`. We also don't include a constructor, however we can still instantiate a new WeatherService instance if we needed to:

```
const myWeatherService = new WeatherService();
```

Next, we use the `static` keyword to define a static method called `getWeather()` which takes a `city` as a parameter.

The code inside this static method is almost exactly the same as it was before. The only difference is that we need to *return* our promise. There's no need to save the promise in a variable. So, `getWeather` is just a method that returns a promise object.

The big gotcha that we see for many students trying to separate out logic is that they forget the `return` keyword. As we know, functions need to return something or they will be `undefined`. We've seen many situations where students thought they were getting `undefined` in their code because they weren't handling asynchrony properly — but the real reason was because a function didn't return anything!

Now let's take a look at the updated code for `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) {
  let promise = WeatherService.getWeather(city);
  promise.then(function(weatherDataArray) {
    printElements(weatherDataArray);
  }, function(errorArray) {
    printError(errorArray);
  });
}

// UI Logic

function printElements(data) {
  document.querySelector('#showResponse').innerText = `The
humidity in ${data[1]} is ${data[0].main.humidity}%.
  The temperature in Kelvins is ${data[0].main.temp} degree
s.`;
}

function printError(error) {
  document.querySelector('#showResponse').innerText = `Ther
e was an error accessing the weather data for ${error[2]}:
${error[0].status} ${error[0].statusText}: ${error[1].messa
ge}`;
}

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);
});
```

First we need to make sure to import our `WeatherService`. We make our API call by doing the following:

src/index.js

```
function getWeather(city) {
  let promise = WeatherService.getWeather(city);
  promise.then(function(weatherDataArray) {
    printElements(weatherDataArray);
  }, function(errorArray) {
    printError(errorArray);
  });
}
```

It should be really clear why our static `WeatherService.getWeather()` method needs to return a promise — otherwise, the `promise` variable would be undefined when we use it.

Because the variable holds a promise, we can call `Promise.prototype.then()` on it.

An Alternate Organization of `index.js`

Consider the following alternate organization of our logic in `index.js`. This is just as acceptable as the initial solution, and could be considered more favorable since there's fewer lines of code without sacrificing good separation of logic and readability.

How you decide to organize the code in your own applications will depend on the functionality: how you need to process the data involved and how many API calls you need to make. Regardless of how you choose to organize your code, your business logic and UI logic should remain clearly separated.

src/index.js

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

// UI Logic

function handleFormSubmission(event) {
  event.preventDefault();
  const city = document.querySelector('#location').value;
  document.querySelector('#location').value = null;
  let promise = WeatherService.getWeather(city);
  promise.then(function(weatherDataArray) {
    printElements(weatherDataArray);
  }, function(errorArray) {
    printError(errorArray);
  });
}

function printElements(data) {
  document.querySelector('#showResponse').innerText = `The
humidity in ${data[1]} is ${data[0].main.humidity}%.
  The temperature in Kelvins is ${data[0].main.temp} degree
s.`;
}

function printError(error) {
  document.querySelector('#showResponse').innerText = `Ther
e was an error accessing the weather data for ${error[2]}:
${error[0].status} ${error[0].statusText}: ${error[1].messa
ge}`;
}

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

Summary

In this lesson, we've made some minor changes to our code that make a big difference in terms of separating logic and keeping our code concise and clean. You'll be expected to separate code related to API calls into its own file for this section's independent project.

Here's the project with separated logic on the branch called `3_separate_logic_with_xhr_and_promises`:

📂Example GitHub Repo for API Project with Promises (https://github.com/epicodus-lessons/section-6-js-api-call-with-webpack/tree/3_separate_logic_with_xhr_and_promises)

Previous (/intermediate-javascript/asynchrony-and-apis/static-methods-and-properties)

Next (/intermediate-javascript/asynchrony-and-apis/sop-and-cors)

Lesson 20 of 33

Last updated more than 3 months ago.

disable dark mode

Epicodus (http://www.epicodus.com)