Exercise 16 – AJAX and JSON

Objectives

To work with AJAX and JSON to create a dynamic image feed.

Overview

In this exercise, you will be working with the SlideShow object you created in a previous chapter. We are going to download data from a simple web service and parse it from JSON back to an array.

This exercise is scheduled to run for around **30 minutes**.

Exercise Set Up

1. In VSCode, open the file called **index.html** and **index.js** from the **Starter** and **Starter/src** folders inside **Ex16** of the **Exercises** folder.
2. This project has been set up to use Webpack and Babel, so from the *Ex16 Starter* folder, run the command **npm i** to install the required packages.
3. Fire up the application with the command **npm start**. It should open your browser and display the image gallery (minus any images!).
4. Open another command window and navigate the prompt to **Ex16→Starter→Data.**
5. Run the command **json-server images.json.**
6. Open your browser and navigate to **http://localhost:3000** and click on the link to open **images**.

What you should see is a JSON object that has the images array in it. We will use this as the basis of the data we will retrieve for the exercise.

json-server is a "full fake REST API with zero coding". Essentially, it allows us to use a properly formed JSON file and serve that as a result of an HTTP request. It is a readily available package, installed globally on your machines from NPM.

Part 1 - Using Fetch

The **SlideShow** class in the **index.js** file has been stripped back, so that only functions that we do not need to modify are present (apart from the constructor), as has the call to the constructor at the bottom of the page. You will find the HTML remains largely intact from the code produced for the previous Slide Show exercise. The default text for the **<figcaption>** has been modified to suggest that the gallery is loading.

The first thing that we need to do is to rebuild the constructor for the slide show. The array of images will no longer be passed directly into **SlideShow**. The slide show will retrieve it when it is constructed. Therefore, the only parameters that are needed are for the 4 elements that were used previously.

1. Modify the constructor signature so that it accepts 4 parameters:
   a) **theImage**;
   b) **imageCaption**;
   c) **next**;
   d) **prev**.
2. Set these instance variables (notionally private) to the *values passed in*.
3. Add another instance variable called **_startPosition** and initialise this as **0**.
4. Add a final instance variable called **_theGallery** and initialise this as *an empty array*.

All of the values that the **SlideShow** class needs to operate have now been set. The next stage is to hook up the **onclick** event of the buttons again.

5. Still in the **constructor**, set the **onclick** property of both **_next** and **_prev** to be an *arrow function* that takes *no parameters* and *calls* the *appropriate function* (**_nextImage** or **_previousImage**).
6. *Disable* both of these buttons by setting their **disabled** property to **true**.
7. Initialise the **src** property of **_theImage** to be the gif called **waiting** in the images folder.
   (Note: the relative paths are important!)
8. Save your code and check the output in the browser. You should see the buttons, the **figcaption** and the revolving gif.

We now know how our gallery will display before it knows about any images. The next stage is to make a call to our **json-server** to deliver the array of images via a HTTP request. Our class is going to need a function we can call to load the images.

9. Under the **constructor**, but *still within the class definition*, add a function called **_ajaxLoad**. The function should:
   a) Define a **const** of **url** set to the string **http://localhost:3000/images**;
   b) Make a **fetch** call to **url** and **then**:
      i) Take a value called **response** and *through an arrow function*:
      ii) Check **if** the **response.code** is **200** and:
         (1) **return response.json()** if it is;
         (2) And if not, set **_imageCaption.innerText** to **"The gallery cannot be found"** and **_theImage.src** to an *empty string*.
   c) And **then** take a value called **galleryData** and *through an arrow function*:
      i) Set **_theGallery** to **galleryData**;
      ii) Set **_startPosition** to a *random number* within the **length** of **_theGallery**;
      iii) Call the **_displayImage** function.
   d) And catch any errors, setting **_imageCaption.innerText** to **"The gallery cannot be found"**.

Notice that we check the status code of the response. This is because the Promise returned will resolve as long as it receives a status code - this could be a 404, a 503, etc. What we are doing is ensuring that we only use good data!

Now we have a function to load the images array from an external source, we need to call the function somewhere.

10. Add a call to _ajaxLoad at the bottom of the constructor.

In the solution file, we have added a **setTimeout** around the function call. This is set for **1500ms** and delays executing the function for this time to simulate a network delay.

11. Save your code. You should see the gallery displayed as before but the **next** and **prev** buttons are still disabled!
12. In the **_displayImage** function, set the **_next** and **_prev** disabled properties to **false** before any other code executes.
13. Again, save your code and check that the gallery works as required.

It would be sensible to test that our error checking works. We will do this by providing a bad url to simulate a 404 error and by stopping our json-server so the promise rejects.

14. Change the value of **url** in the **_ajaxLoad** function to something that is not what it is currently!
15. Save your code and observe the expected result - the image gallery cannot be found. Check the 404 error on the console.
16. On the command line running **json-server**, hit **CTRL+C** to stop the server running.
17. Refresh your browser and observe the expected result - the image gallery cannot be found. Check the GET error on the console.

Part 2 - Using async/await

In this part of the exercise, you are going to replace the call to the **_ajaxLoad** function that uses **fetch** and **Promises** explicitly to with an **async** function. For this to work with **Babel**, we need to install another dependency via **npm** and also modify the **webpack.config.js** file to recognise it.

1. Close the previous webpack-dev-server command line (you can keep json-server running).
2. Install babel-polyfill using the command:

```
npm i babel-polyfill --save
```

3. Edit webpack.config.js by adding the **entry** line before the **module** object:

```
module.exports = {
  entry: ['babel-polyfill', './src/index.js'],
  module: {
    …
  }
}
```

4. Spin up the **webpack-dev-server** again with the command:

```
npm start
```

All of this work is to ensure that Babel knows how to transpile the async/await code we are about to write. Without this polyfill being applied, we would not be able to use our code across all browsers.

Now to get started with our asynchronous function!

5. Define a function called _ajaxLoadUsingAsync, ensuring that you have the async keyword before it. The function should:
   a) Define a const called url set to http://localhost:3000/images;
   b) Define a const called response that awaits a fetch call to url;
   c) Define a const called galleryData that awaits the result of response.json();
   d) Sets _theGallery to be galleryData;
   e) Sets _startPosition to be a random number between 0 and the length of _theGallery;
   f) Calls the _displayImage function.
6. Replace the call to _ajaxLoad in the constructor with a call to _ajaxLoadUsingAsync.
7. Save your code and check the output in the browser. You should still have a fully functioning gallery.