Camosun College

ICS 211 - Web Applications

Lab 6 - Rendering Data From a JSON-based API

Due Date: Lab Demo Due by Nov. 2 @ 2:20 PM

Lab Quiz (10 Questions) Due by Nov. 2 @ 11:59 PM

Background and Theory

RESTful JSON APIs

REST (Representational State Transfer) is a stateless architectural style of building an API on top of the HTTP Protocol so that the API can be used over the Internet. This architectural style is part of the service-oriented, distributed, API-first architecture of modern web applications. In such an application, the *front-end* is completely separate from the *back-end*. The *front-end* consumes data from the *back-end* via the RESTful API. The *front-end* is typically client-side JavaScript running in a user's browser whereas the *back-end* is running on a remote server. The client-side JavaScript sends and receives data asynchronously from the *back-end* using the API. The *back-end* is said to be working as a *web service*. In most cases, the data the web service sends to the client is in JSON format. We will build RESTful APIs next semester in 221. For this course, we'll only worry about consuming and rendering the JSON data we get from the *web service*.

In this lab, we'll use a "fake" web service. This Server will be local on your machine and will serve a JSON file but lacks a full back-end database-driven implementation.

Stateful Components

React Components have another attribute associated with them besides `this.props`, `this.state`. Props are **immutable** within a Component but State is **mutable**. A Component with a state variable can update the value by calling `this.setState()`. Updating a state variable causes the Component's `render()` method to be re-called. You should limit the number of Stateful Components in your React Application. Stateless Components are simpler and easier to code. To make a Component Stateful, you have to implement its Constructor as state variables need to be initialized there. **See the lecture slides on Stateful Components for an example on how this is achieved.**

ES6 Arrow Functions

JavaScript and Node.js make heavy use of Callbacks. Callbacks are functions that are executed after an anonymous operation completes. Callbacks are typically anonymous functions. ES6 introduced a more concise

way of writing these functions called *Arrow Functions*. Arrow functions have the following formats:

- No params, return implied (concise body): `() => return_value`
- Single param, return implied (concise body): `param=> return_value`
- Multiple param, return implied (concise body): `(p1, p2)=> return_value`
- Single param, no return value (block body): `param=> { ... }`

If a *block body* Arrow Function does need to return a value, the `return` must be explicitly written. It isn't implied like in the concise body. **For some examples of Arrow Functions, see the associated lecture slides.**

Component Lifecycle Events

A React Component has several Events that occur over its lifetime. They can be split into *Mounting Events*, *Updating Events*, and the single *Unmounting Event*. Lifecycle methods allow you to fine-tune the behaviour of your Component. They are also often used for integrating other libraries and frameworks into React. For example, in this lab, you will use the `componentDidMount()` lifecycle method to integrate the Fetch API into your React Component.

ES6 Promises

Asynchronous Functions and Callbacks have long been the bane of JavaScript Developers especially once you start nesting them. The code gets quite complicated especially when integrated with error handling. To help solve this issue, ES6 introduced Promises. A Promise Object is always in one of three states: *Pending*, *Fulfilled*, or *Rejected*. Asynchronous Functions that implement the Promise API will create a Promise Object that is initially in the *Pending* state. When the async function successfully completes, the Promise Object becomes *Fulfilled*, and its `then()` method is executed. The result of the function is passed into this method. On the other hand, if the async function fails in some way, the Promise Object becomes *Rejected* and an Error object is passed to its `catch()` method. Promises can be chained as long as the `then()` method of the previous promise returns something. **For more on Promises and some examples, see the associated lecture slides.**

The `fetch()` API

XHR has long been the bane of AJAX Developers. It's an old API and is cumbersome to use. Some wrapper APIs around the XHR API have been developed to make it easier to code AJAX calls. JQuery has `ajax()` and another popular wrapper around XHR is axios. More recently, a new API has been developed to replace XHR called `fetch()`. The `fetch()` API is a cleaner, more powerful, and modern API compared to XHR. All modern browsers now support the `fetch()` API (Internet Explorer is an exception). Older browsers do not. For those browsers, there are libraries that "emulate" the `fetch()` API using XHR so you only need one codebase for all browsers new and old. **See the lecture slides for an example on how to use this API.**

The JavaScript `map()` method

`map()` is a method that you can run on an array. It takes each element of an array and applies a function to it that you pass into `map()`. It also works with JSX. Instead of a function, you pass in the JSX you want generated for each element. `map()` is typically used with the `fetch()` API. `fetch()` retrieves the data and `map()` "JSX-izes" it for rendering. **See the lecture slides on the Fetch API for an example on how to use map().**

Tasks

Task 1 - Setting Up Your JSON Server

1. For this lab, you will use a "fake" RESTful API called JSON Server. This won't be remote, it will be a local Node.js server that will serve a JSON object that you create in a file. Normally, you would write an entire *back-end* application that would implement all your CRUD operations with a database and generate the JSON when requested. But that's next semester!! To install the JSON Server, run the following command:

   ```
   npm install -g json-server
   ```

   You may have already done this if you did the example from the lecture.

2. The next thing you need to do is write the JSON Object that the Server will serve. In the last two labs, you have been working on a *Favourite Sites* application. The *Favourite Sites* have been hard-coded in your *client-side* code. What if instead those sites were previously entered in a form (future lab) and stored on a remote server. Your Application would need to make an AJAX Request to the Server to fetch the data. We'll simulate such a scenario by making a JSON Object of your Favourite Sites. Follow the template below:

   ```
   {
     "sites": [
       { "id": 1, "name": "First Site Name", "link": "Site Link",
         "color": "First Site Color" },
       { "id": 2, "name": "Second Site Name", "link": "Second Site Link",
         "color": "Second Site Color" },
       { "id": 3, "name": "Third Site Name", "link": "Third Site Link",
         "color": "Third Site Color" }
     ]
   }
   ```

   Replace the *Site Names* with the actual Site Names you used. Replace the *Site Links* with the actual links you used. The `color` attribute in this example is the CSS property that was passed in as props in the last lab. You may have chosen a different CSS property. **Use whatever one that you used**. Store this file as `lab6.json`.

3. To start your JSON Server to listen for API Requests, in the same directory as `lab6.json`, issue the following command:

   ```
   json-server --watch lab6.json
   ```

   When it's up and running, it will output a message of where the Resources (REST endpoint) are located. Most likely, it will say `http://localhost:3000/sites`. Make a note of that, you will need it later!

The rest of this lab involves refactoring your solution from Lab 5 to use the JSON data instead of the hard-coded data in the FavSites Component. Copy your Lab 5 Solution into a new HTML file. This will be your starting point.

Task 2 - Make the FavSites Component Stateful

1. The first thing you need to do to make the *FavSites* Component stateful, is to implement a constructor. **Refer to the lecture notes on Stateful Components to see how to do this. Another good resource is the Fetch API Example from the lecture notes on the Fetch API.**

2. In the Constructor of *FavSites*, ensure a `sites` state variable is initialized to an empty array.

Task 3 - Fetch the Data from the JSON Server

Add the `componentDidMount()` lifecycle method to your *FavSites* Component. This method should use the *fetch()* function to fetch the JSON object from your JSON Server. **Use the url you noted in Task 1 as the API endpoint to fetch**. Your fetch API call should also call `this.setState` to update the state variable you initialized in your constructor. **Refer to the lecture slides on the Fetch API** to properly implement this step. Don't forget to also add the `handleHTTPErrors(response)` method to your *FavSites* Component.

Task 4 - Use the JavaScript `map` Function to "JSX-ize" the Data

Now you need to refactor *FavSite*'s `return` statement in its `render()` method to "JSX-ize" the data. After the fetch call, a JavaScript array will be stored at `this.state.sites`. You want to use the `map()` function inside the `<ol>` tags to iterate over each element of the array, instantiating an instance of your `Site` component, passing in the data as props. **There is an example of doing this near the end of the lecture notes on the Fetch API**. You will only need a single `<Site .../>` statement as the `map()` function will generate the JSX you pass in for each element in the array. Don't forget that React requires a `key` attribute on elements inside a `map()` function.

Task 5 - Refactor Both the Title and Site Components as Arrow Functions

Since both the Title and Site Components are **Stateless** Components, they should be made into *Arrow Functions*. **Use the block body form of Arrow Functions for these.** That means you must explicitly `return()` something from the function. Note that for your *Site* Component, now that it is no longer a class, `this` has no meaning. So `this.props` becomes just `props`. **Refer to the lecture notes on Arrow Functions.**

Lab Submission (see top of lab for due date)

1. Demo to the Lab Instructor (10 marks):
    - That you have both `http-server` and `json-server` running
    - That your website STILL works!
    - That your *FavSites* Component is Stateful.
    - That your *FavSites* Component called the fetch API inside the `compountDidMount()` lifecyle method.
    - That your *FavSites* Component uses the JavaScript `map()` function to render the *Site* Component.
    - That both your *Title* and *Site* Component Classes are refactored as Arrow Functions.
    - For full marks, all have to be implemented correctly.

2. Do the Lab 6 Quiz in D2L (10 marks).