Project Instructions

This Github repository is your starter code for the project. Clone the specific branch 'refresh-2019' or download the ZIP file, although feel free to start from scratch! It is the same as the starter code we began with in Lesson 2. Install and configure Webpack just as we did in the course. Feel free to refer to the course repo as you build this one, and remember to make frequent commits and to create and merge branches as necessary!

The goal of this project is to give you practice with:

- Setting up Webpack
- Sass styles
- Webpack Loaders and Plugins
- Creating layouts and page design
- Service workers
- Using APIs and creating requests to external URLs

We have divided the instructions into the following stages, as explained below:

Stage 1 - Getting Started - Setting up the Project

It would be good to first get your basic project up and functioning. Fork the project Github repo, and then clone or download the zip file locally. Remember that once you clone, you will still need to install everything:

```
cd <project directory>
npm install
```

Follow the steps from the course up to Lesson 4, but ***do not add Service Workers just yet***. We won't need the service workers during *development*, and having extra caches floating around just means there's more potential for confusion. Just for your quick reference, we installed the following loaders and plugins so far:

```
# Choose the necessary installation for your development mode
npm i -D @babel/core @babel/preset-env babel-loader
npm i -D style-loader node-sass css-loader sass-loader
npm i -D clean-webpack-plugin
npm i -D html-webpack-plugin
npm i -D mini-css-extract-plugin
npm i -D optimize-css-assets-webpack-plugin terser-webpack-plugin
```

Stage 2 - Setting up the API

If you started this project on or before July 7, 2020, you will be using the Aylien API for this project. The Aylien API has you install a node module to run certain commands through. It will simplify the requests we need to make from our node/express backend.

If you started this project after July 7, 2020, you will be using the MeaningCloud Sentiment Analysis API for this project.

The project rubric item for "API" criteria says:

> *The app should make a successful call to the API on form submission. If this is successful, the API keys and response handling were done correctly. You can check that the API keys are found in server.js or a similar node file. It is not acceptable for an API key to be set in a client-facing file (like index.js)*

> *Information from the API response must show up in the view. It is not enough for the response to be logged to the console or saved in js, etc.*

Step 1: Sign up for an API key

For the Aylien API: You will need to go to the user signup page. Signing up will get you an API key. Don't worry, at the time of this course, the API is free to use up to 1000 requests per day or 333 intensive requests. It is free to check how many requests you have remaining for the day.

For the MeaningCloud API: You can find the API here. Once you create an account with MeaningCloud, you will be given a license key to start using the API. This API does not require an SDK, so you can skip ahead to step 4 in the instructions.

Step 2: Install the SDK (Aylien API only)

Next, you'll need to get the Software Development Kit (SDK) for Node.js. SDK is usually a program that brings together various tools to help you work with a specific technology. For instance, the Aylien SDK brings together a bunch of tools and functions that will make it possible to interface with their API from our server. Aylien SDKs are available for all the major languages and platforms, such as Node, Python, PHP, Go, Ruby, and many others.

Install the SDK in your project, as per the instructions mentioned for **Node.js SDK** at Text Analysis API Documentation.

Step 3: Require the SDK package (Aylien API only)

Your `server/index.js` file must have these things:

```
// Require the Aylien npm package
var aylien = require("aylien_textapi");
```

Step 4: Environment Variables

Next, in `server/index.js`, you need to declare your API credentials, which will look something like this:

```
// You could call it aylienapi, or anything else
var textapi = new aylien({
  application_id: "your-api-id",
  application_key: "your-key"
});
```

If you are using the MeaningCloud API, the process will look pretty similar to the Aylien API process, but you don't need to use an `application_id`.

...But there's a problem with this. We are about to put our personal API keys into a file, but when we push, this file is going to be available PUBLICLY on Github. Private keys, visible publicly, are never a good thing. So, we have to figure out a way to make that not happen.

The way we will do that is with environment variables. Environment variables are pretty much like normal variables in that they have a name and hold value. The environment variables only belong to your local system and won't be visible when you push your code to a different environment like Github. Follow the steps below to configure environment variables:

1. Use npm to install the dotenv package - `npm install dotenv` This will allow us to use environment variables we set in a new file
2. Create a new `.env` file in the root of your project.
3. Fill the `.env` file with your API keys like this:

   ```
   API_ID=**************************
   API_KEY=**************************
   ```

4. Add this code to the very top of your `server/index.js` file:

   ```
   const dotenv = require('dotenv');
   dotenv.config();
   ```

5. If you want to refer the environment variables, try putting a prefix `process.env.` in front of the variable name in the `server/index.js` file, an example might look like this:

   ```
   console.log(`Your API key is ${process.env.API_KEY}`);
   ```

   The step above is just to help you understand how to refer an environment variable from your code. In `server/index.js`, your updated API credential settings should look like this:

   ```
   // You could call it aylienapi, or anything else
   var textapi = new aylien({
      application_id: process.env.API_ID,
      application_key: process.env.API_KEY
   });
   ```

6. Go to your `.gitignore` file, in the project root, and add `.env`. It will make sure that we don't push our environment variables to Github! If you forget this step, all of the work we did to protect our API keys would become pointless.

Step 5: Using the API

We're ready to go! The Aylien API has a lot of different endpoints you can take a look at the Aylien API endpoints. You can see how using the SDK simplifies the requests we need to make. You can also check out the documentation of the MeaningCloud API here. MeaningCloud also has several other APIs, which we won't be using for this project, but feel free to take a look around if you're curious!

Now it's up to you to create the various requests and make sure your server is set up appropriately. For example, ensure that the "dependencies" in `package.json` have a suitable entry for Aylien, such as, `"aylien_textapi": "^0.7.0",`, where the version may vary with time.

Stage 3 - Project Enhancement

At the current stage, make enhancement in your project code to ensure most of the requirements as mentioned in the project rubric are met. In addition, parse the response body to dynamically fill content on the page.

Only the rubric requirements related to "Offline Functionality" and "Testing" criteria should remain for the next stages.

Stage 4 - Unit Testing using Jest Framework

You must have read the rubric item for "Testing" criteria, that says:

> *Check that the project has Jest installed, that there is an npm script to run Jest, and that the tests all pass. Every src/client/js file should have at least one test.*

Jest is a framework for testing JavaScript projects. We are interested in the unit-testing of our project. The Jest framework provides us the ability to create, and run unit tests. In general, unit testing means to test the functionality of each unit/component of a project. But, in our case, we will write tests for desired functions defined in the src/client/js directory. The tests will check if the functions are behaving expectedly when provided an input. Let's learn to add Jest to your project to handle unit-testing.

How does it work?

1. Install Jest by using `npm install --save-dev jest`
2. Write the custom JS in your src/client/js directory, responsible for the server, and form submission task. **For example**, assume that the `/src/client/js/formHandler.js` file has the following function to be tested:

```
function handleSubmit(event) {
    event.preventDefault()
    // check what text was put into the form field
    let formText = document.getElementById('name').value
    Client.checkForName(formText)
    console.log("::: Form Submitted :::")
}
export { handleSubmit }
```

3. **You have to ensure that all your custom functions in src/client/js directory can handle error responses if the user input does not match API requirements.** You will write tests in `<function_name>.test.js` or `<function_name>.spec.js` file, to be present in a `__test__` folder. For each functionality, consider writing a separate test file. The `__test__` folder should be present in the project directory.

   In each test file, the general flow of the test block should be:

   - Import the js file to test
   - Define the input for the function. Note that, to keep it simple, we will not validate the input being provided to the test cases.
   - Define the expected output
   - Check if the function produces the expected output

   For the example function shown above, `/src/client/js/formHandler/handleSubmit()`, you can write a test file `testFormHandler.spec.js` in the `__test__` directory, having a test block as:

```
// Import the js file to test
import { handleSubmit } from "../src/client/js/formHandler"
```

```
// The describe() function takes two arguments - a string description, ar
// A test suite may contain one or more related tests
describe("Testing the submit functionality", () => {
    // The test() function has two arguments - a string description, and
    test("Testing the handleSubmit() function", () => {
           // Define the input for the function, if any, in the form of v
           // Define the expected output, if any, in the form of variable
           // The expect() function, in combination with a Jest matcher,
           // The general syntax is `expect(myFunction(arg1, arg2, ...)).
           expect(handleSubmit).toBeDefined();
})});
```

You must be wondering about the matchers, and other syntactical information about test blocks. At this point, you must refer to the external resources:

- Jest - Getting started - Provides a basic overview, with the help of an example.
- Jest - matchers - Read carefully to identify the suitable matcher for each of your functions.
- Jest - testing asynchronous code - If you have code that runs asynchronously.
- A tutorial for beginners - A good explanatory tutorial.

4. Configure an npm script named "test" in `package.json` to run your tests from the command line:

```
"scripts": {
    "test": "jest"
}
```

Also, ensure that the "devDependencies" in `package.json` have a suitable entry for Jest and others, such as, `"jest": "^25.3.0",`, where the version may vary with time.

5. Run the `npm run test` command.

Stage 5 - Service Workers

The rubric item for "Offline Functionality" criteria says:

> *The project must have set up service workers in webpack.*

Go to the webpack config file, and add the setup for service workers. Test that the site should be available even when you stop your local server.

Stage 6 - Deployment

A great step to take with your finished project would be to deploy it! Unfortunately its a bit out of scope for me to explain too much about how to do that here, but check out Netlify or Heroku for some really intuitive free hosting options.