Lab Week 8 - Unit & E2E Testing

Start Assignment

Due Sunday by 11:59pm **Points** 3 **Submitting** a website url

In this lab, we are going to be walking you through two of the many ways to test your applications. The first style of testing we'll be looking at is End-to-End (E2E) testing. This style tries to replicate a user's workflow from start to finish (i.e. from end-to-end). As such, it will be more focused on interacting with the webpage. The second style we are going to look at you might be more familiar with already: Unit testing. When you unit test, you test individual parts of your code in encapsulated units to make sure each part of your code is functioning as it should. Debugging on a small scale without many moving parts is much easier than inside of a large moving application.

Resources

- **https://jestjs.io/docs/en/getting-started** (https://jestjs.io/docs/en/getting-started)
- **https://pptr.dev/** (https://pptr.dev/)
- **https://pptr.dev/#?product=Puppeteer&version=v11.0.0&show=api-pageselector-1** (https://pptr.dev/#?product=Puppeteer&version=v11.0.0&show=api-pageselector-1)
- **https://github.com/smooth-code/jest-puppeteer** (https://github.com/smooth-code/jest-puppeteer)

Set Up

- Fork Lab 8 repo here: **https://github.com/CSE110-FA22/Lab8_Starter** (https://github.com/CSE110-FA22/Lab8_Starter)
- (https://github.com/CSE110-F2021/Lab8_Starter) Run this terminal command

```
npm -v
```

  - (checks if you have the Node Package Manager aka npm installed -- this is a tool that allows you to access and download resources such as external library code written by other people in the form of software packages that npm locates within a giant registry)
    - If you don't have it on your local machine, install it here (https://www.npmjs.com/get-npm) **https://www.npmjs.com/get-npm** (https://www.npmjs.com/get-npm)

- run this command:

```
npm install --save-dev jest babel-jest @babel/core @babel/preset-env puppeteer jest-puppeteer
```

    - This installs Jest, the Javascript testing framework we'll be using, and Babel, a Javascript compiler that allows us to use ES6 modules over CommonJS.
    - Note: "--save-dev" includes these packages only during development. Since puppeteer and jest are just for testing, there's no need to include the extra dependencies when you deploy for projects for production
  - Add the new *node_modules* directory to your **.gitignore** inside your top-level directory to prevent code bloat committed to your repository.
- If the installation does not automatically produce a **package.json** file for you, you can create a blank one and fill it out with the data in the section below

Configuring Jest & Puppeteer

- First, configure your npm script to run Jest. In your package.json, add a *scripts* section with the following:

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

This will tell your node interpreter to run the "jest" command when we run "npm run test"

- Configure your node project
  - Inside of your **package.json** file, inside the JSON object at the top level (e.g. right before the last curly brace '}'), add the following in order to run the "jest" command with the "jest-puppeteer" framework, with verbose output

```
"jest": {
  "preset": "jest-puppeteer",
  "verbose": true
}
```

your package.json file should end up like this:

```
{
  "devDependencies": {
    "@babel/core": "^7.16.0",
    "@babel/preset-env": "^7.16.0",
    "babel-jest": "^27.3.1",
    "jest": "^27.3.1",
    "jest-puppeteer": "^6.0.0",
    "puppeteer": "^11.0.0"
  },
  "scripts": {
    "test": "jest"
  },
  "jest": {
    "preset": "jest-puppeteer",
    "verbose": true
  }
}
```

Intro

Software testing has become an absolutely integral part of the software development pipeline. Every software company integrates many different forms of software testing procedures and frameworks in order to ensure the quality of the software they are producing.

The most recent developments in the practice of testing is the increased use of automation testing, in which scripts are written to automatically test your new and existing software based on certain events, i.e. code getting pushed to the remote repository, or test runs of the whole repo that are triggered nightly.

In your final lab, we are going to be walking you through two of the many ways to test the front-end of your applications, namely **unit tests** and **end-to-end tests**. For unit testing we will use the **Jest** framework, and for end-to-end tests we will use **Puppeteer** to operate the browser and **Jest** again to perform the tests.

Check Your Understanding

Fill your answer in your **README.md**

1) Where would you fit your automated tests in your Recipe project development pipeline? Select one of the following and explain why.

1. Within a Github action that runs whenever code is pushed
2. Manually run them locally before pushing code
3. Run them all after all development is completed

Part 1 - Expose: E2E Testing with Jest-Puppeteer (8 points)

This first style of testing we'll be looking at is **end-to-end testing,** also named **UI Testing or E2E testing.** End-to-end testing is a way for developers to automate test cases that involve emulating user actions from start to finish (end to end). We will be using an api called Puppeteer which provides developers with tools that enable you to interact with a web page by navigating to its url and interacting with its DOM.

Check your understanding:

2) Would you use an end to end test to check if a function is returning the correct output? (yes/no)

Setting up your webapp to be tested (only if needed)

If you are having any issues connecting to our GitHub pages website, we have included the entire site for you locally in the starter code, you can run it by

1. First, opening **index.html** in VS Code, then running Live Server.
2. Next, updating the URL in **lab8.test.js** to your local URL from your Live Server

Open the __tests__/lab8.test.js file

This is where you will write your Puppeteer tests. The describe() function creates a test suite for your SPA app. A test suite is essentially a collection of tests that all execute to test the same area of code. Defining test suites allows you to encapsulate all of the test logic so that tests from one suite don't affect the results of tests in another suite.

We've included the first two tests for your that are mostly complete so that you can generally see how they operate. You first interact with the webpage using **Puppeteer** (that's the **Page** variable you're seeing, along with some other things), then you use **Jest** to create
tests (e.g. **expect(something).toBe(something)** - this is but one example, **Jest** has a whole suite of things you can use to test with).

We've outlined 8 steps for you to complete in **lab8.test.js** to complete the testing suite, follow the instructions for each to complete each of the tests.

Some helpful resources:

Puppeteer Docs: **https://pptr.dev/** (https://pptr.dev/)

Puppeteer Page Selector Docs: **https://pptr.dev/#?product=Puppeteer&version=v11.0.0&show=api-pageselector-1** (https://pptr.dev/#?product=Puppeteer&version=v11.0.0&show=api-pageselector-1)

Jest Docs: **https://jestjs.io/docs/getting-started** (https://jestjs.io/docs/getting-started)

To run your tests, use the command:

```
npm run test
```

NOTE: An empty test case will default to a pass on a test run.

Part 2 - Explore: Unit Testing with Jest (2 points)

The next style we are going to look at you might be more familiar with already: Unit testing. Unit testing involves testing individual parts of your code in encapsulated units to make sure each part of your code is functioning as it should.

Pros:

- Debugging on a small scale without many moving parts is much easier than inside of a large moving application.
- Execute quickly
- Changing other app features likely won't affect the non-related unit tests

Cons:

- Cannot test how these individual components interact with each other on an application/feature level.

Check Your Understanding

Fill in your answers in your **README.md**

3) Would you use a unit test to test the "message" feature of a messaging application? Why or why not? For this question, assume the "message" feature allows a user to write and send a message to another user.

4) Would you use a unit test to test the "max message length" feature of a messaging application? Why or why not? For this question, assume the "max message length" feature prevents the user from typing more than 80 characters.

Testing with Jest

To run the tests, run the command `**npm test ./__tests__/sum.test.js**` from your top-level directory. It should pass right now since we outlined a basic function for you in **sum.test.js**

In the **__tests__** directory, open the **sum.test.js** file. Modify the existing test in sum.test.js file to the following:

```
test('adds 1 + 2 to equal 3', () => {
    expect(1 + 2).toBe(3);
});
```

Here, the **test()** function takes in 2 arguments: first, a short description of the test; second, a function of what the test should run. This test used **expect** and **toBe** to test that two values were exactly identical. To learn about the other things that Jest can test: https://jestjs.io/docs/en/using-matchers (https://jestjs.io/docs/en/using-matchers) .

Run:

```
npm test ./__tests__/sum.test.js
```

There should still be 1 passed test now.

We can also add unit tests to test out a function in our program. In `/code-to-unit-test`, there is a `sum.js` file containing a simple function that adds two numbers. In `sum.test.js`:

You can import functions from other files that you want to test. For example, we can import the **sum()** function from sum.js like this:

```
const sum = require('../code-to-unit-test/sum.js');
```

Now, we can write *another* test calling the **sum()** function to test its output:

```
test('adds 1 + 2 to equal 3', () => {
    expect(sum(1,2)).toBe(3);
});
```

Run

```
npm test ./__tests__/sum.test.js
```

There should be 2 passed tests!

Your Turn

Now, just like in **sum.test.js**, you are going to be creating tests in **unit.test.js**. You will notice that we have required a variable **functions** from **unit-test-me.js**. You can access all of the functions from **unit-test-me.js** using this variable by doing something like **functions.isPhoneNumber()**. You will be creating unit tests for each of the functions within **code-to-unit-test/unit-test-me.js** in the file **unit.test.js**

- Run your tests using

```
npm test ./__tests__/unit.test.js
```

- Create **2** tests that should be **true**, and **2** tests that should be **false** for ***each function*** *(4 tests x 5 functions = 20 tests).*
    - For clarification, all of your tests should ***pass***, but you should **expect** two of your tests to return a **true** response and two of your tests to return a **false** response
        - e.g. expect(2+2).toBe(4) and expect(4+4).toBe(10)

Part 3 - Expand (No points)

Some extra tasks you could accomplish that could help you out with your project:

- Create a smaller build pipeline for this lab that will automatically run these tests when you commit
- Use Percy https://percy.io/ (https://percy.io/) - This will let you pix diff different versions of your website, let you know if anything changes / breaks
- Set up a lighthouse tester with a performance budget on this lab to measure performance
- Set up some form of accessibility testing in your smaller build pipeline for this lab

Canvas Submission:

- Link to your repo
    - **README.md** with your and your partner's) name(s) *and the "Check Your Understanding" question answers*
    - A **screenshot** of all your tests results from running **`npm test`**. This should include all tests from **sum.test.js**, **unit.test.js**, and **lab8.test.js**
    - All of your code