CS 253 Mark Holliday

Project 3: Unit testing a railway network

(written by Dr. Andrew Scott and Dr. Mark Holliday)

Due: 5 PM on Monday, November 24

Contents

Section 1. Overview

In this project you extend Project Two by adding unit tests and code coverage. The unit tests including coverage will use the `jest` testing framework, https://jestjs.io.

Since the project is an extension of Project One and Two, if something was not correct in your Project One solution or Project Two solution, you need to fix it in your Project Three solution. If any feature of Project One or Project Two is not correct in your Project Three solution, then that is considered an error in Project Three.

Section 2. Setup

Subsection 2.1. Jest configuration

Create a `project3` directory that has in it a `README.md` Markdown file and your source files (`client_network.js`, `client_routeSummary.js`, `client_route.js`, `railway_network.js`, `railway_routeSummary.js`, and `railway_route.js`).

1. Run the command

```
npm init
```

to create your `package.json` file. The `npm` (Node Package Manager) program will ask you several questions as part of its configuring the `package.json` file.

- at the `test command:` prompt, enter `jest` (no quotes are needed)

When the questions complete, if you do `less package.json` part of what you will see is below.

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

2. At this point add an extra property to the `scripts` property of your `package.json` so it looks like below.

```
  "scripts": {
    "test": "jest",
    "test:coverage": "jest --coverage"
  }
```

3. Then run the command

```
npm install --save-dev jest
```

to install the `jest` test framework. There are two dashes before the word `save` without any space between them. The installation of `jest` causes the `node_modules` subdirectory to be created to hold the jest package. The `--save` flag causes the current jest version to be added as a dependency in the package.json file.

4. Do `ls -la` and you will see that a `node_modules` directory has been created. That directory is large as can be seen by the `du -s` command; when I ran `du -s` on my machine the response was `35220` units of 512 bytes each, so approximately 17 megabytes. The `du` means *disk usage* and the `-s` means *summary*.

5. Create a file of functions to be tested and a file for the test suite for the first file. Then run the command `npm run test:coverage` and the test file will run and output to the console which tests passed and the code coverage.

6. Do a `ls -l` and you will see a new subdirectory named `coverage` is created that is not very large (84 512 byte blocks when the functionality test below was run on my machine). Use `cd coverage/lcov-report` to go into the subdirectory `lcov_report` of the `coverage` directory which contains a webpage containing a set of webpages that is a graphical report of the code coverage including a listing of all the lines of each file being tested.

7. If you are developing the software on a local machine (not agora) you can view the graphical report of the code coverage. In particular, ifyou are using Windows you can use File Explorer to navigate to your WSL files (the bottom of the left side has an icon for Linux and a sub-icon for Ubuntu) Navigate to `Linux > Ubuntu > home > username > project3 > coverage > lcov-report` (with `username` replaced by your WSL username) and click on the `index.html` file to open the graphical report in a browser. This is also file (with `holliday/jest2` replaced by your username and project directory) `file://///wsl.localhost/Ubuntu/home/holliday/jest2/coverage/lcov-report/index.html`. If you click on the name of your source code file on the left, a new webpage appears showing a list of all the code in that file.

Subsection 2.2. Functionality test

To confirm that jest is working correctly use it on the following simple example which is from https://jestjs.io/docs/getting-started.

1. Create a `sum.js` file.

```
function sum(a, b) {
    return a + b;
}
module.exports = sum;
```

2. Create a `sum.test.js` file.

```
const sum = require ('./sum');

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

3. Run the command

```
npm run test:coverage
```

and the Jest will print a message similar to below.

```
> jest2@1.0.0 test:coverage
> jest  --coverage

 PASS  ./sum.test.js
  an assortment of tests
    ✓ adds 1 + 2 to equal 3 (1 ms)

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 sum.js   |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.149 s
Ran all test suites.
```

Subsection 2.3. Make source files node module

You will have eight JavaScript source files that you have from Project 2 that you need to test.

- `client_bestJourneys.js`
- `client_network.js`
- `client_routeSummary.js`

- `client_route.js`
- `railway_bestJourneys.js`
- `railway_network.js`
- `railway_routeSummary.js`
- `railway_route.js`

You need to make each of those source files a node module by exporting all functions you want to be public so that you can test them.

Each of these source files will have a corresponding Jest test file. A particular test file needs to import the functions from the source file it is testing using the `require()` function. The test files and source files need to be in the same directory or you need to use `..` in specifying the relative file path in your `require()` function.

For example, your `client_network.js` file will need to be imported by your `client_network.test.js` testing file.

Section 3. Requirements

In the `project3_rubric.pdf` besides specifying general properties your program should have, there is in the `Correctness` section a subsection that is specific to correctness requirements for unit testing and project 3. Below are some other requirements.

Subsection 3.1. General requirements

- **Servers do not stop running when the respond to an API call (that is, an HTTP request); they continue to run until explicitly killed by a ctrl-C signal.**

- You can comment out the `main()` functions of `railway.js` and `network.js` prior to conducting the unit tests,

- It is acceptable to just use one railway network in your unit tests, but it must be a complex network. So `uk.json` is a good choice and `simpleton.json` is a very bad choice.

- If you go to the Jest website, https://jestjs.io/docs/using-matchers, you can see all the matchers and modifiers that jest supports in its `expect` assertions.

- your tests should be *specific and informative*. This means that a unit test should be designed so that if it fails, the person testing has very specific information about what went wrong.

- When comparing a JavaScript object the `toEqual` matcher is useful since it checks for deep equality meaning that it recursively checks every field of an object or array. Other matchers are likely to be useful.

Subsection 3.2. Checking exceptions thrown

For this project you should improve the functions you have written for `railway.js` and `network.js` so that the gracefully handle invalid and valid input rather than just falling over. You must also optimize them for speed.

If you think having an exception thrown is the correct response for an invalid input, then you must write unit tests to check for that exception. Below is an example of using the `toThrow` jest matcher within a test case to check for exceptions.

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Division by zero');
  }
  return a / b;
}

describe('divide', () => {
  it('should throw an error when dividing by zero', () => {
    expect(() => divide(10, 0)).toThrow('Division by zero');
  });

  it('should return the correct result', () => {
    expect(divide(10, 2)).toBe(5);
  });
});
```

Notice that the test when an exception is expected is the `divide(10, 0)` case. For that test the argument to `expect` is

```
expect(() => divide(10, 0))
// not
expect(divide(10, 0))
```

In Jest, when you expect a function to throw an error, you need to wrap the function call (in this case, `divide(10, 0)`) in a function (using an anonymous function) because `toThrow` is an assertion on the function execution, not on the result of the function itself.

Section 4. Administrative

- The project is due at 5:00 PM Monday, the 24th of November. You can submit a project multiple times before the time and day it is due. I will just grade the last version. No late projects are accepted.
- **Separate from your solution each of you will also submit a brief attestation of, in your view, your contributions and the contributions of your partner. Submit your attestation on your Canvas Project 3 submission location either as a file or in the Comment section.**
- You will work in teams of two students; your partners will be assigned. If you have problems working with your partner find out quickly and request to work alone.
- You may talk with other people and AIs about the concepts involved in doing the project, but anything involving actual code needs to just involve your team. In other words, you can not show your team's code to people outside of your team or AIs and you can not look at the code of another teams or of an AI.

- Your project will be graded based on how it runs on agora when I test it. See the `project_rubric.pdf` file for how the project will be grading. As noted in the rubric file you do not need to JSDoc your `describe` or `test` (that is, `it`) functions since their first argument is a description of what the function does. However, that argument should be a specific and detailed description of what that test function is doing.
- JSDoc is very similar to javadoc. See the following links for more details
  - https://en.wikipedia.org/wiki/JSDoc
  - http://usejsdoc.org/
  - https://github.com/jsdoc3/jsdoc
- Your code must not be placed on Git repositories that are public.
- You will use your GitHub Classrooms repository for this project to submit your project. In particular, when you are ready to submit your project, create a new branch of your code that has your submitted version that is named `submission`. I will download the version of your code on that branch to agora and grade it on how it runs on agora.