Lesson Monday

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Setting Up Jest

Text

There are many JavaScript testing frameworks. Some of the most popular include **Jest**, **Jasmine**, and **Mocha**. These libraries have many similarities, and after learning one, you can quickly learn others. We use Jest because it is the testing framework of choice for React. It's also a popular testing framework for JavaScript in general.

We'll continue building out our shape tracker application. Here's the project so far:

📂Example GitHub Repo for Shape Tracker (https://github.com/epicodus-lessons/section-5-shape-tracker)

Make sure that you are referencing the code from the branch called `1_functioning_environment`. This is the default branch, so running `git clone...` with the URL of the repo homepage will automatically clone down the branch called `1_functioning_environment`. If you need a review on how to navigate between branches, review the

lesson on accessing code from different branches (https://www.learnhowtoprogram.com/lessons/accessing-code-from-different-branches).

At this point, our application has a fully functioning environment along with user interface logic. However, there's hardly any business logic other than a `Triangle` constructor function and a `Triangle.prototype.checkType*()` method that returns `"I can't answer that yet!"`. Now we're ready to use Jest and TDD to build out our business logic.

Jest Installation and Setup

First, we'll need to add Jest to our project:

```
$ npm install jest@24.9.0 --save-dev
```

Once again, we are pinning the version so it will play nicely with our configuration. This is a development dependency because testing is part of the development process.

Next, we need to add a new npm script called `"test"` to run Jest in our `package.json`:

```
"scripts": {
  "build": "webpack --mode=development",
  "start": "npm run build && webpack-dev-server --open --mo
de=development",
  "lint": "eslint src --ext .js",
  "test": "jest"
}
```

Note that we could call our script something other than `"test"`, but `"test"` makes sense here, because we're invoking our test runner, Jest.

At this point, we'll get an error if we run `$ npm run test`. That's because we haven't added any tests yet. Before we can add tests, we need to set up a folder and file for tests that Jest can recognize.

Adding A New Test Directory

All Jest tests must live inside of a directory called `__tests__` in the root directory of our project. Go ahead and create that directory now in the Shape Tracker project. The files and folders in the root should now look like this:

```
shape-tracker/
├── __tests__/
├── dist/
├── node_modules/
├── src/
├── .eslintrc
├── .gitignore
├── package-lock.json
├── package.json
├── README.md
└── webpack.config.js
```

It is common convention to add a double underscore `__` at the beginning and end of `__tests__` with Jest. We will store all our test files in this directory. For now, we just need one: `triangle.test.js`.

```
shape-tracker/
├── __tests__/
│   └── triangle.test.js
```

With the name of the test file, we follow another common naming convention, which looks like this:

```
// This is pseudocode.
nameOfFileToBeTested.test.js
```

In other words, the test file name should match the name of the file that has the code we are testing — the difference in the file naming structure is that we add `.test` to the name as well.

Now if we run our tests with `$ npm run test`, Jest will automatically look in the `triangle.test.js` file:

```
$ npm run test

> shape-tracker@1.0.0 test
> jest

 FAIL  __tests__/triangle.test.js
  ● Test suite failed to run

    Your test suite must contain at least one test.

      at node_modules/@jest/core/build/TestScheduler.js:24
2:24
      at asyncGeneratorStep (node_modules/@jest/core/build/
TestScheduler.js:131:24)
      at _next (node_modules/@jest/core/build/TestSchedule
r.js:151:9)
      at node_modules/@jest/core/build/TestScheduler.js:15
6:7
      at node_modules/@jest/core/build/TestScheduler.js:14
8:12
      at onResult (node_modules/@jest/core/build/TestSchedu
ler.js:271:25)

Test Suites: 1 failed, 1 total
Tests:       0 total
Snapshots:   0 total
Time:        2.909s
Ran all test suites.
```

As you might guess, this is a bad fail because we aren't testing anything yet. Jest states the following: `Your test suite must contain at least one test.`

Another Way to Run Jest

Note that we can also run `jest` directly in the terminal to run our tests. To do this, we'll use npx, which is node's package runner (https://nodejs.dev/learn/the-npx-nodejs-package-runner). In the

root of any directory that has a `__tests__` folder with a test file within, we can run this command:

```
$ npx jest@24.9.0
```

When we enter this command, node package runner does three things:

- Accesses the internet to look in its registry of all node packages.
- Locates the package called `jest` at version 24.9.0.
- Then invokes Jest in our project, which leads to the same effect as calling `$ npm run test`

We can also use npx to run other packages like webpack and ESLint, as long as our project has the correct configuration files in place. For example, instead of `$ npm run test`, we could run

```
$ npx eslint src/*.js
```

However, we can just as easily use npm scripts to invoke the packages installed locally to our project, and we don't even need the internet to use them (as long as the packages are already installed)! Because of this, it's not necessary to use npx.

We are almost ready to start writing our tests. However, before we do, we'll need to set up another tool called Babel. We'll do that in the next lesson.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/red-green-refactor-workflow)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/setting-up-babel)

disable dark mode

(http://www.epicodus.com)