

Unit Testing Lab

Our environment could be considered complete. But many people would say that you can't call it complete until unit testing is incorporated.

In this lab, we're going to set up unit tests using jasmine.

Setting up Jasmine

The first step would normally be to create an npm script entry called "test" that runs jasmine. But we already did that when we initialized npm. Let's verify.

1. Edit your package.json file. Look for the "test" entry. Make it say this:

```
"test": "node_modules/jasmine/bin/jasmine"
```

2. Next, install Jasmine with npm

```
npm install --save-dev jasmine
```

3. Open a terminal window and run it to initialize the jasmine environment

```
npm run test init
```

4. Notice that this has created a folder called spec and a config file called jasmine.json. Take a look at this file. See that line where it mentions spec_files? That means that any file that ends in "spec.js" will be considered a test file and will be run by jasmine. Let's create one!

5. Create a test file in the spec folder called romanNumeralConverter.spec.js. Make it say this:

```
describe('Roman numeral converter', () => {  
});
```

describe is a grouping of tests. You'll put all tests that are related to one another in a *describe* group.

6. Rerun Jasmine:

```
npm run test
```

7. Look in the terminal window. It may have seen the group of roman numeral tests but there were no tests/specs to run.

Let's create one!

Writing a simple test

8. Inside the *describe* grouping, create a test:

```
it('will return 1 given i', () => {  
  let actual = convert('i');  
  expect(actual).toEqual(1);  
});
```

9. One more time, run Jasmine. This time it will fail because the `convert()` function doesn't exist. So we'll work on that.

10. Create a new folder called `src`.

11. In there create a JavaScript file called `roman_numeral_converter.js`. Put these lines in it:

```
function convert(romanNumeral) {  
  throw "Not yet implemented";  
}  
export default convert;
```

12. Run Jasmine again. It should still fail because `convert` isn't defined but hey, you have Jasmine running and you have a test! Now let's automate it.

Making tests part of the build process

At this awkward time in JavaScript history, we have two ways of importing things, one for the server and a different way in the browser. Our production JavaScript files are used in the browser but they're being tested on the server. In this section, we're going to teach the server how to read these client-side files.

13. Just to prove the issue, add this to the top of `roman_numeral_converter.spec.js`:

```
import convert from '../src/roman_numeral_converter';
```

14. Until the day that node surprises us and begins to support ES2015 modules, you'll get an error. Grrr! Let's fix that now.

15. Install `esm` which is a node module that allows ES 2015 modules:

16. Change the `package.json` scripts line by adding `"-r esm"`:

```
"test": "node -r esm node_modules/jasmine/bin/jasmine"
```

Your test still fails, but it isn't because `convert` isn't recognized. It is because we told it to throw in the `convert` method. If that's the case, let's move on.

Making tests part of the build process

It might make sense for us to re-run all unit tests as part of every build and before deploying our site. This will ensure that we have regression testing to protect us from new code breaking existing working code.

17. Open `package.json` and find your `"build"` script entry. Add `"npm run test"` to it.

```
"build": "npm run test && webpack",
```

18. Run this script and notice that before the compile your tests are also run. Since your tests are currently failing, the build throws. Nice.