What Is Unit Testing?

freecodecamp.org/learn/front-end-development-libraries-v9/lecture-understanding-the-different-types-of-testing/what-is-unit-testing

As you design your applications, you will often have a series of small functions responsible for one thing. This is known as the "single responsibility principle." When you have a series of small functions, it is best to test these functions to ensure that everything works as expected. In this lesson, we will take a look at how to create a unit test using the popular Jest testing framework.

In this example, we are going to create a function that is responsible for returning a newly formatted string:

```
export function getFormattedWord(str) {
  if (!str) return '';
  return str.charAt(0).toUpperCase() + str.slice(1);
}
```

This `getFormattedWord` function has a parameter called `str` and will first check if the `str` is empty. If so, then an empty string is returned. Otherwise, a new string is returned where the first letter is capitalized. We are exporting this function so we can use it in a test file.

In a separate `getFormattedWord.test.js` file, we can write some tests to verify that the function is doing what it is supposed to be doing.

At the top of the `getFormattedWord.test.js` file, we need to first import the function like this:

```
import { getFormattedWord } from "./getFormattedWord.js";
```

Then we need to install the Jest package by using `npm i jest`. Then we can add a test to check that a word made up of lowercase characters, `hello`, is properly formatted to `Hello`, which starts with a capital `H`:

```
test('capitalizes the first letter of a word', () => {
  expect(getFormattedWord('hello')).toBe('Hello');
});
```

The `expect` function is used to test a value. It's combined with a matcher, which is a function that checks whether the value behaves as expected. In this case the matcher is `toBe()`. Jest has a variety of matchers to help you test for truthiness, strings, numbers, and more.

One way to run your tests is by adding an npm script to your `package.json` file. Here's an example script for the Jest framework:

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

Then just run `npm run test` in your terminal to run your tests. If your tests pass, you should see something in the terminal like this:

```
PASS ./getFormattedWord.test.js
  ✓ capitalizes the first letter of a word (1 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.103 s, estimated 1 s
Ran all test suites.
```

To see what a failing test looks like, we can intentionally break the test by updating the function to the following:

```
export function getFormattedWord(str) {
  if (!str) return '';
  return "This is incorrect";
}
```

Now when you run the `npm run test` command, there will be an error message because the test was expecting a different result:

```
FAIL ./getFormattedWord.test.js
  × capitalizes the first letter of a word (1 ms)

  ● capitalizes the first letter of a word

  expect(received).toBe(expected) // Object.is equality

  Expected: "Hello"
  Received: "This is incorrect"

    2 |
    3 | test('capitalizes the first letter of a word', () => {
    4 |   expect(getFormattedWord('hello')).toBe('Hello');
                                          ^

    5 | });
    6 |

    at Object.toBe (getFormattedWord.test.js:4:37)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.121 s, estimated 1 s
Ran all test suites.
```

Now we can update our function back to the original here:

```
export function getFormattedWord(str) {
  if (!str) return '';
  return str.charAt(0).toUpperCase() + str.slice(1);
}
```

Add another test to check for cases where no argument is provided to `getFormattedWord`:

```
test("returns an empty string when no argument is provided", () => {
  expect(getFormattedWord()).toBe("");
});
```

There are other tests you can add to make things more robust, but these first couple of tests are a good introduction to unit testing.

When it comes to testing JavaScript applications some common testing frameworks include Jest, Mocha, and Vitest.

Unit testing is important because it will help you catch more bugs in your programs, and ensure that everything works as expected. It also can serve as a form of documentation for your application because it is meant to represent the expected behavior for your code.

Questions

Which of the following refers to the principle where a function should only be responsible for one thing?

Single responsibility principle

Correct!

Multiple responsibility principle

Single request principle

Multiple response principle

Which of the following is NOT a commonly used testing framework in the JavaScript ecosystem?

JUnit

Correct!

Jest

Vitest

Mocha

What can you add to a `package.json` file to run your tests in the terminal?

```
"test": {
  "test": "run
test"
},
```

```
"scripting
s": {
  "jest"
},
"ru
n":
{

"te
st"
},
```

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

Correct!