Writing Unit Tests (CodeGrade)

Learning Goals

- Follow a test-driven development process for writing code
- Write a unit test using Jest

Introduction

In this lesson, we'll walk through the steps of writing a unit test for a function, then in the next lesson we'll give you an exercise to write your own unit test and get it to pass by following a test-driven development workflow.

As before, we'll use Jest as our test framework. We'll be writing the tests in the `src/__tests__/utils.test.js` file and the application code in the `src/utils.js` file.

Fork and clone this lesson, then run `npm install` to install the dependencies.

Unit Test Example

In this example, we'll follow a test-driven development approach to writing tests and implementing a solution in code. We'll follow these steps:

1. Understand the feature we're building
2. Translate the feature into a test specification
3. Write and implement code that passes the test
4. Clean up and refactor
5. Repeat!

Feature Request

Imagine we're building a word puzzle game. One piece of functionality our game needs is the ability to calculate a score for a given word. In our word game, a user will enter a word, and they'll receive 1 point for each vowel and 2 points for each consonant.

For example, if a user enters the word "test", they'll receive 7 points:

```
t   e   s   t
2 + 1 + 2 + 2 = 7
```

Now that we understand the feature, let's start writing a test!

Write the Test

Given the feature above, a rough outline of the code we'd like to have would be:

```
I expect calling the function pointsForWord("test") will return 7
```

Let's translate that into some test code. As before, we can use `describe` and `it` to provide context around what functionality the test is intended for:

```
import { pointsForWord } from "../utils";

describe("pointsForWord", () => {
  it("calculates the total points for a word (1 point per vowel, 2 per consonant)", () => {
    // TODO: write test code
  });
});
```

We can add our test code based on understanding the input and expected output for this test case:

```
import { pointsForWord } from "../utils";

describe("pointsForWord", () => {
  it("calculates the total points for a word (1 point per vowel, 2 per consonant)", () => {
    const word = "test";

    const points = pointsForWord(word);

    expect(points).toBe(7);
  });
});
```

The key here is that we're using the `expect` function and passing in an "expected" value to test, and we're using the **`.toBe` matcher (https://jestjs.io/docs/expect#tobevalue)** to check that `points` is exactly `7`.

> **Note**: when writing tests, you'll need to determine the right Jest matcher function to test values, depending on the data type you're testing. The Jest documentation on **Using Matchers (https://jestjs.io/docs/using-matchers)**, along with the Jest API documentation on **Expect (https://jestjs.io/docs/expect)**, are good resources to bookmark when you're deciding what matcher is appropriate.

At this point, we should run the tests with `npm test`. Woohoo, a failing test! We're now in the "red" stage of our "red-green-refactor" process. Let's get it to green.

Write Code

Now that we've got a test to work with, let's implement this functionality. Remember, at this point we're just trying to get the **minimum** amount of code required to get the test to pass. There are a number of ways you could approach this. Just go with whatever approach you're most comfortable with — no need to get fancy! Remember, we'll have time to refactor later.

Here's one approach we could take:

```
// src/utils.js
export function pointsForWord(word) {
  let points = 0;
  for (const char of word) {
    if (["a", "e", "i", "o", "u"].includes(char)) {
      points += 1;
    } else {
      points += 2;
    }
  }
  return points;
}
```

While there are some issues with this code, it does the trick and passes our test. Running `npm test` now should verify: our tests are green!

This is also a good time to commit our code, since we've got a working version, so that we can go back to a known working state in the event our refactor doesn't go as planned.

Refactor

Now that we've got working code, we can confidently refactor, knowing that if anything goes wrong we can always go back to our earlier working implementation.

See if you can come up with a better approach!

After each refactor, run the tests to verify that they're still passing.

Here's one possible refactor, which uses a regular expression to check for vowels:

```
export function pointsForWord(word) {
  let points = 0;
  for (const char of word) {
    points += /[aeiou]/.test(char) ? 1 : 2;
  }
```

```
    return points;
  }
```

Repeat

Now that we've successfully implemented this feature, we can repeat the process when we have more information about different use cases our code should handle. Right now, all we can say with confidence is that calling `pointsForWord` with a value of `"test"` will produce a result of `7` . What about other ways users might use this function? Here, it's helpful to think of "edge cases" where a user might try to use our function in unexpected ways. For example:

- Handling a string that has both uppercase and lowercase letters
- Handling an empty string
- Handling a string that has non-alphanumeric characters (i.e. numbers, symbols, emoji)
- Handling a value that isn't a string

You don't necessarily have to write tests for all possible use cases: only the use cases that may actually occur when a user interacts with your application. For now, let's focus on the first case: handling a string that has both uppercase and lowercase letters.

We'll go through the whole TDD process once more:

Write the test inside the `describe` callback:

```
it("handles uppercase and lowercase input", () => {
  const word = "tEsT";

  const points = pointsForWord(word);

  expect(points).toBe(7);
});
```

Run the test to verify that the test isn't passing. Red!

Next, write the code to pass the test (in this case, using the `i` flag for the regular expression to indicate a case-insensitive match for either uppercase or lowercase values):

```
export function pointsForWord(word) {
  let points = 0;
  for (const char of word) {
    points += /[aeiou]/i.test(char) ? 1 : 2;
  }
  return points;
}
```

Run the test to verify that the test is passing. Green!

Note that our original test is still passing, so we know that our new code didn't break the original test case.

Feel free to refactor at this point too.

Conclusion

Now you've seen the full test-driven development process in action! The tests for this lesson should be relatively simple, since we can clearly define our test cases and our the kind of functions we're testing lend themselves nicely to unit testing. In the coming lessons, we'll show more advanced features of Jest and demonstrate how to write tests for more challenging functions.

Resources

- **Jest (https://jestjs.io/)**