Test-Driven Development Codealong (CodeGrade)

 (https://github.com/learn-co-curriculum/js-tdd-codealong)  (https://github.com/learn-co-curriculum/js-tdd-codealong/issues/new)

Learning Goals

- Follow a test-driven development process for writing code
- Understand the file structure required to test applications using Jest
- Use the Arrange-Act-Assert pattern to structure test code
- Read and understand the output from running tests with Jest

Introduction

To get a sense of how you might use test-driven development to build a new feature in an application, let's work through an example together.

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

Writing a Test

Identifying the Desired Behavior

Before we can write any tests, we need to think about what our program needs to do. In other words, what is the desired behavior of our code?

Let's say you run a popular social networking site. We'll call it MyFace, a name inspired by nothing in particular.

Your users will fill out their profile information with what year they were born. You will subsequently need to display how old they are on their profile page.

We could conceive of needing to write a function `currentAgeForBirthYear` to accomplish this task.

Our function will need to take in the user's birth year, subtract that from the current year, and return the user's age.

If the year is currently 2022 and I was born in 1984, when I call the function `currentAgeForBirthYear` and provide it my birth year, 1984, by passing that year as an argument, `currentAgeForBirthYear(1984)`, I expect it to return `38`.

Expressing that narrative in code is called a test!

Coding Our Test

In an ideal world, we could code this requirement with something like:

```
I expect calling the function currentAgeForBirthYear(1984) will return 38
```

To express this requirement in code, we use a **testing framework**.

Different programming languages have different testing frameworks. JavaScript has several popular libraries, including Mocha, Jasmine and Jest. The most popular choice for React developers is Jest, so that's what we'll be using in this course.

Jest (https://jestjs.io/) is a JavaScript Testing Framework designed to allow programmers to describe the behavior and outcomes of their programs in a very natural language similar to the above example.

Let's learn a bit about Jest.

Understanding the Test in the `__tests__` Directory

In this lesson, our code is organized in the following file structure:

- `src/__tests__` : a folder containing all our test files
- `src/__tests__/utils.test.js` : a file containing tests for the `utils.js` file
- `src/utils.js` : a file for common utility functions that we can use in other parts of our imaginary application

All of our tests will be located within the `__tests__` directory.

Our actual code, our programs, our solutions to the challenges in the lab, the stuff that makes our tests pass, are all coded outside of the `__tests__` directory, generally in the `src` directory or its subfolders.

When we run our tests, `src/__tests__/utils.test.js`, that code will import the variables in `src/utils.js` and try to execute `currentAgeForBirthYear(1984)` with the expectation that it returns `38`. If so, the test will pass. Anything else will make it fail.

```
// src/__tests__/utils.test.js
import { currentAgeForBirthYear } from "../utils";

describe("currentAgeForBirthYear", () => {
  it("returns the age of a person based on the year of birth", () => {
    const birthYear = 1984;
    const ageOfPerson = currentAgeForBirthYear(birthYear);
    expect(ageOfPerson).toBe(38);
  });
});
```

Let's break this code down.

How your test loads your variables:

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

The first line of the test ensures that we can access the variables and functions defined in the `currentAgeForBirthYear` file.

> **Note**: this lesson uses **Babel (https://babeljs.io/setup#installation)** to enable the `import` / `export` syntax in our application when we run the tests. Babel is included and configured with Create React App, so you don't have to worry about setting it up yourself. Since this lesson wasn't set up using Create React App, we needed to also include some additional configuration in the `.babelrc` and `package.json` files.

The `describe` function in Jest:

```
describe("currentAgeForBirthYear", () => {});
```

`describe` is a function provided by Jest. It is a way of grouping related tests together so that when we run our tests, it's clear which tests describe the `currentAgeForBirthYear` function.

The `it` function in Jest:

After describing the subject of our test, the function `currentAgeForBirthYear`, we use the Jest function `it` to state an expectation of our function.

```
it("returns the age of a person based on their year of birth", () => {});
```

The `it` function takes two arguments:

- a **string** where we describe what we're testing
- a **callback function** where we write the actual test code

Jest also lets you use the `test` function in the same way as the `it` function, depending on what style you prefer. So we could also write the same test as:

```
test("returns the age of a person based on their year of birth", () => {});
```

Testing our function:

The next three lines are our actual test code and the most important part of the test file. It is within this callback function passed to the `it` function that we test the functionality of our function.

In order to actually test our code, we need to use the function that this test relies on, that this test is designed to exercise. So the first real lines of code in our test are:

```
const birthYear = 1984;
const ageOfPerson = currentAgeForBirthYear(birthYear);
```

What we're doing here is calling a function, `currentAgeForBirthYear(birthYear)`, the very function we're supposed to define and implement, passing it a known value, `1984`, and assigning the return value of the function to a variable called `ageOfPerson`.

What do you think the value of `ageOfPerson` should be if the function `currentAgeForBirthYear` is called with `1984` as the argument?

The next line of code poses that exact question with an expected outcome. Using Jest, we say, quite colloquially:

```
expect(ageOfPerson).toBe(38);
```

What this line of code means is that we `expect` the value of the variable `ageOfPerson` to be `38`.

`expect` is another function that is provided by Jest. We pass it a value, and it returns an object that has a number of **matchers (https://jestjs.io/docs/using-matchers)** like `.toBe` that let us check if the value is what we want it to be.

Our test loads our code, uses our code in the manner desired, and compares the result of our code with a known outcome so that we know our code behaves as we expected.

We could imagine another specification of the `currentAgeForBirthYear` function as another `it` function within the opening `describe` function:

```
it("returns the current year for a person born in year 0", () => {
  const birthYear = 0;
  const ageOfPerson = currentAgeForBirthYear(birthYear);
  expect(ageOfPerson).toBe(2022);
});
```

A test is always going to be about setting up a state with a known result, and comparing that known result (or *expectation*) to the behavior of your program, thus ensuring that your program behaves as you expected.

Arrange-Act-Assert

One common way to structure tests is by following a pattern known as **arrange-act-assert**. This pattern helps our test clearly communicate our intent, and provides a consistent way to structure all our tests.

- **Arrange**: set up any data that need to be tested
- **Act**: act upon the system being tested
- **Assert**: make claims about what we expect the outcome to be

You can see this structure clearly with comments in our test code:

```
it("returns the current year for a person born in year 0", () => {
  // Arrange
  const birthYear = 0;

  // Act
  const ageOfPerson = currentAgeForBirthYear(birthYear);

  // Assert
  expect(ageOfPerson).toBe(2022);
});
```

While the test above is relatively simple, following this pattern makes the story we're telling with our tests clear.

Running Our Tests

Now that we can read our test code in our `__tests__` directory, let's actually run the tests. We're going to execute our test program, which is going to:

- Load our real program
- Try to use it in a certain manner we defined in our tests
- Report on the results.

To do all this, simply run the `npm test` command in your terminal.

The `npm test` command will run a script defined in the `package.json` file. In this case, it's running the `jest` command:

```
// package.json
{
  "scripts": {
    "test": "jest"
  }
}
```

Jest will automatically run any tests that are either:

- In a `__tests__` directory, or
- Have `.test` in the filename.

Jest runs JavaScript code, so everything in our test files must be valid JavaScript.

Jest also has a great **watch mode (https://jestjs.io/docs/cli#--watch)** feature that will automatically re-run tests related to changed files. You can enable it by updating the test script in the `package.json` file:

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

When running Jest in watch mode, you can interact with the test suite in the terminal. Check out this video for a quick overview of some watch mode features:

Use Jest's Interactive Watch Mode (https://egghead.io/lessons/javascript-use-jest-s-interactive-watch-mode)

Understanding Test Output

When you run the tests with the `npm test` command (or `npm t` for short), you'll see the results of the test in your console. Jest will report on what is working and what is broken and why.

> **Note**: If you see "No tests found related to files changed since last commit." in the test output, press the `a` key to run all the tests.

When you run this lab's test suite with `npm test`, before writing any solution code, you'll see output similar to:

```
> react-hooks-tdd-codealong@1.0.0 test
> jest

 FAIL  src/__tests__/utils.test.js
```

```
    currentAgeForBirthYear
      × returns the age of a person based on the year of birth (2 ms)

    ● currentAgeForBirthYear › returns the age of a person based on the year of birth

      TypeError: (0 , _utils.currentAgeForBirthYear) is not a function

        4 |   it("returns the age of a person based on the year of birth", () => {
        5 |     const birthYear = 1984;
      > 6 |     const ageOfPerson = currentAgeForBirthYear(birthYear);
          |                          ^
        7 |     expect(ageOfPerson).toBe(38);
        8 |   });
        9 | });

        at Object.<anonymous> (src/__tests__/utils.test.js:6:25)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.873 s
Ran all test suites.
```

Let's break this down.

```
 FAIL  src/__tests__/utils.test.js
  currentAgeForBirthYear
    × returns the age of a person based on the year of birth (2 ms)
```

Those lines are summaries of what we are testing and what failed. They correspond directly to the strings provided to `describe` and `it`, and they are simply there to provide context.

```
● currentAgeForBirthYear › returns the age of a person based on the year of birth

  TypeError: (0 , _utils.currentAgeForBirthYear) is not a function

    4 |   it("returns the age of a person based on the year of birth", () => {
    5 |     const birthYear = 1984;
  > 6 |     const ageOfPerson = currentAgeForBirthYear(birthYear);
      |                         ^
    7 |     expect(ageOfPerson).toBe(38);
    8 |   });
    9 | });

    at Object.<anonymous> (src/__tests__/utils.test.js:6:25)
```

This actually describes why our test failed.

TypeError:

The above line raises the line of code in our test suite that created the failure and error. The rest of the output describes the error:

```
TypeError: (0 , _utils.currentAgeForBirthYear) is not a function
```

Before writing any code, our test suite is failing because of a line of code within it:

```
const ageOfPerson = currentAgeForBirthYear(birthYear);
```

This line tried calling a function, `currentAgeForBirthYear`, which your test expected to have been defined. You have yet to define it, however, resulting in a `TypeError`.

Tip: We can run our test suite as many times as we want, it's totally free. In fact, we suggest that every time you make a change to your code and think it might solve something in the test, run the test suite again. Run the test suite a lot. Get instant feedback. Read the errors; they are clues.

It's totally cool to have errors — a big part of programming is simply getting past the current error your test suite raises and getting to a new error. Progressing through errors until your tests pass is a very normal development cycle.

Reading Errors And Making Our Tests Pass

So, we conceptually understand what we're trying to build: a function called `currentAgeForBirthYear`, that when given an argument of a year of birth, `currentAgeForBirthYear(1984)`, returns the age of a person, `38`. Our test suite actually tries to execute this code and compares the result of it to the desired outcome, failing until the expectation and the outcome are equal.

The first error thrown by the test suite is that our code, defined in `src/utils.js`, should have defined a function called `currentAgeForBirthYear`, but did not, resulting in a `TypeError` when we attempt to invoke the function.

Let's fix this error by defining a function in `src/utils.js` called `currentAgeForBirthYear`.

Add the following to the file, `src/utils.js`:

```
export function currentAgeForBirthYear() {}
```

Save the file and go back to your terminal and run the `npm test` command. You'll see output including:

- currentAgeForBirthYear › returns the age of a person based on the year of birth

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

Expected: 38
Received: undefined

   5 |     const birthYear = 1984;
   6 |     const ageOfPerson = currentAgeForBirthYear(birthYear);
>  7 |     expect(ageOfPerson).toBe(38);
     |                         ^
   8 |   });
   9 | });
```

This failure isn't a syntax error related to an undefined function. Instead, this error is telling us that we expected the return value of the function `currentAgeForBirthYear(1984)`, stored in the variable `ageOfPerson` to equal 38, but in actuality, the function returned the value `undefined`.

That's perfect. Our test is showing a **mismatched expectation**. We need to add actual logic to that function to solve the problem.

How do we calculate the difference between the year currently and the year provided to the function as an argument `birthYear`? You might simply subtract the birth year from the current year, 2022 in our case (at the time of writing).

```
export function currentAgeForBirthYear(birthYear) {
  return 2022 - birthYear;
}
```

Run `npm test` again and you should see the test suite passing. Great job!

Refactor

The current implementation is that the test suite and our solution are brittle and can produce a false positive. Our tests hard-code the current year at the time they were written (in 2022). If the current year isn't 2022, our code will break but our tests will still pass. As long as we are relying on hard coded notions of the current year, our code and tests aren't honest.

We could take advantage of JavaScript's built-in `Date` object instead:

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

describe("currentAgeForBirthYear", () => {
  it("returns the age of a person based on the year of birth", () => {
    // Arrange
    const currentYear = new Date().getFullYear();
    const birthYear = 1984;
    const answer = currentYear - birthYear;

    // Act
```

```
    const ageOfPerson = currentAgeForBirthYear(birthYear);

    // Assert
    expect(ageOfPerson).toEqual(answer);
  });
});
```

That test would use the year at the moment the test was executed to compute the accurate result and compare it to the result of calling the function.

To make that pass you would have to implement your solution in `utils.js` as:

```
export function currentAgeForBirthYear(birthYear) {
  return new Date().getFullYear() - birthYear;
}
```

That would be a better implementation of `currentAgeForBirthYear` as it is more abstract.

Conclusion

In this lesson, we went through an example of how to follow a test-driven development approach to writing code. We demonstrated how to structure files in our application so that we can run tests using Jest; discussed some of the specific syntax used in writing Jest tests (such as `describe`, `it`, and `expect`); and learned how to run our tests and decipher the output.

In the next lesson, we'll discuss different **types** of tests, and how to know what type of test to write as you're building new features for your application.

Resources

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

This tool needs to be loaded in a new browser window

Load Test-Driven Development Codealong (CodeGrade) in a new window