Competency 130 – JavaScript TDD

Module Plan

Leader: L Shumlich

Date: 2019/02

Overview & Purpose

The subject of developer-written unit tests is a long and complicated one. There are many things that need to be identified:

- What is the purpose of developer-written unit tests?
- What are the advantages and disadvantages?
- How technically do we do unit testing?

Lecture - TDD the basics

Not all code can be unit tested properly but my rule is that, if we don't unit test a piece of code, we need a darn good reason why and not the other way around.

I unit test and strongly promote Test Driven Development (write your test first) for many reasons and here are just a few:

- The quality of the code increases significantly
- The maintainability of the code increases significantly
- The structure of the code improves significantly

For more info search coupling and cohesion in software engineering:

- https://en.wikipedia.org/wiki/Cohesion_(computer_science)

Some reference reading:

- https://medium.freecodecamp.org/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9
- https://medium.com/mobile-quality/test-driven-development-d16fd216d45c
- http://agiledata.org/essays/tdd.html ⇒ Spend some time on this one
- https://medium.com/welldone-software/an-overview-of-javascript-testing-in-2018-f68950900bc3

If we write our tests first, our code becomes better. In all cases follow these steps:

  - write a stub method / function
  - write a test that fails
  - write the method to satisfy the test, repeat and refactor
  - add more tests
  - add more code
  - repeat and refactor until done

We'll practise in the coming exercises.

Exercise - Running our first test

Let's run our first test:

- from the command line navigate to "evolveu/objects" directory. This was created in our last exercise.
- Run the following command: npm test

    - this runs all the tests in our project. In our case there is only one. This test was installed as part of the create-react-app
    - as we write tests and save our code, the running process will keep running the changed tests
    - review the options
- npm test -- --coverage
    - this runs all the tests and give us the code coverage. Larry will be walking around and looking at this all the time
    - this also creates an html report: /evolveu/objects/coverage/lcov-report/index.html
- https://facebook.github.io/create-react-app/docs/running-tests
    - way more info

Exercise - Writing our first test

We are going to start using new JavaScript features that became available ES6, ES7, and ES8. We are also going to develop the simplest of tests. We will write code for our basic math functions and create a component for React.

The steps for Test Driven Development (this is just the coding part of TDD):

    - write a stub method / function
    - write a test that fails
    - write the method to satisfy the test, repeat and refactor

Let's get started:

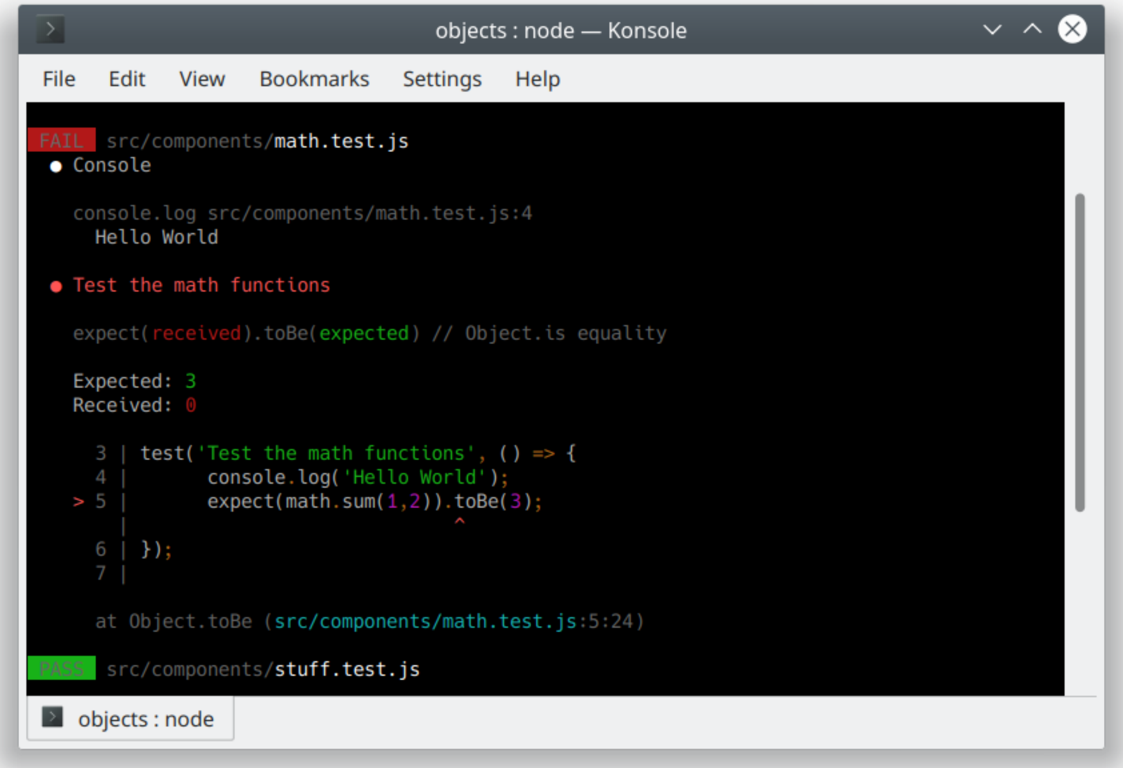
- in directory "evolveu/objects" start the test runner
    - npm test
- create two new files in "evolveu/objects/src/components" math.js and math.test.js
- write a stub method / function. A stub literally means create the method signature (method name and parameters) and return a dummy value. We will start by writing the sum function first.

```
function sum(a, b) {
      return 0;
}

export default { sum };
```

- write a test that fails. This test will fail because the function returns 0 all the time

```
import math from "./math";

test('Test the math functions', () => {
      console.log('Hello World');
      expect(math.sum(1,2)).toBe(3);
});
```

- the results from the test runner



```
FAIL  src/components/math.test.js
 ● Console

    console.log src/components/math.test.js:4
      Hello World

 ● Test the math functions

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

    Expected: 3
    Received: 0

      3 | test('Test the math functions', () => {
      4 |         console.log('Hello World');
    > 5 |         expect(math.sum(1,2)).toBe(3);
        |                               ^
      6 | });
      7 |

      at Object.toBe (src/components/math.test.js:5:24)

PASS  src/components/stuff.test.js
```

- what nice feedback
- now that we have a failing test, write the code to make it pass
- repeat these steps for our 4 favorite functions: + - / *
- remember: stub, failing test, pass, repeat

Exercise - Using the Components

Now that we have some well tested functions, let's attach them to a React Component. Using all your JavaScript, React, and TDD knowledge do the following:

- Create a React component called MathComp.js in the components directory. Start first with it having a <h1> saying "Hello World from MathComp"
- Connect it to the main App. When one of the icons are pushed, that MathComp will be displayed
- Create a test for your MathComp; copy the App.test.js. This test can be written after. Why?

- Slowly modify your MathComp.js to start implementing your functions in your math.js. Remember to do one line and test. When it quits working you may need to backup to the last spot it last worked and start again. As you implement your functionality, you have the confidence that your “Problem Solving Code” (PSC) is working properly because your tests are working.

Congratulations! This is another major step.

Lecture - Reading the Documentation

There is some really good documentation available. There are many locations, but this is the source:

- https://jestjs.io/
- select the “Docs” tab
- we have done the “Getting Started” section a slightly different way but achieved the same results
- start reading the “Using Matchers” tab
- browse the rest; we will need to use some of these features