TDD in JavaScript

Section 1: Tooling and Introduction to Mocha

1.1 Visual Studio Code, NPM, Mocha and Chai

Visual Studio is a very powerful editor with an integrated terminal. Any editor can be used to write code such as Atom, Sublime, Notepadd++, etc. However, Visual Studio has grown in popularity with developers. You can install the VS Code editor by visiting Microsoft's VS Code website (https://code.visualstudio.com/). This is a free cross platform editor.

NPM is an acronym or Node Package Manager. This is a Package Manager for JavaScript which allows us to pull packages/libraries down into our npm based projects. When installing Node.js on our machines this installs the NPM as well. Node.js can be installed by visiting their website (https://nodejs.org/en/). You can search for npm packages/ libraries by visiting https://www.npmjs.com/.

Mocha and Chai are two npm packages/libraries that we can install into our JavaScript (Node.js) projects using the npm commands in our terminal. We can learn more about these libraries from their website (https://mochajs.org/ and https://www.chaijs.com/).

To create a new NPM/Node project we can run the following command in the terminal (ensuring we are within the project directory within the terminal i.e. using cd to navigate to the project path before running the following command):

```
$ npm init
```

Note: We can use the -y flag to autofill all of the project setup questions.

Once the Node project has been created we should see a new package.json file which outlines the NPM project we are working with. We can now run the following commands to install the Mocha and Chai packages:

```
$ npm install --save-dev mocha
```

```
$ npm install --save-dev chai
```

Important Note: the --save-dev flag will save the packages as development dependencies and will not download the package when we deploy the live application. We can also run one command instead of two separate commands to install multiple packages at once as seen below. This will also apply the same flag parameters to both package installations.

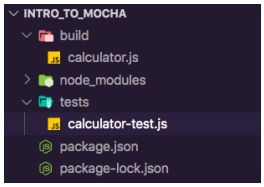
```
$ npm install --save-dev mocha chai
```

The package.json file will now update adding the two packages as a devDependencies and a new node_modules directory will appear in the root of our project directory. This node_modules will contain the installed libraries along with their library dependencies.

We now have all of the tooling setup and can now see how to run some basic tests in mocha along with the chai assertion library.

1.2 Introduction to Mocha

Typically in the root of your project directory you would have a src or a build folder which will hold the live production files/codes. In the root project directory will also be a tests folder containing your Mocha test code of your live/build code. Below is an example of what this directory structure would look like for a very basic project.



We can see that the -test.js file relates to the JavaScript file it will test. This is a common naming pattern where the test file name corresponds to the file it is testing.

The first thing we would do in the -test.js file is to import the chai assertion library to work with our Mocha tests. To do this in Node.js is to use the require function to import a library and then assign it to a variable we can use to call the various methods from the library object.

Below is a simple assertion example where we check whether the string foo is not equal to the string bar.

```
var assert = require("chai").assert;

describe("Simple assert", function() {
    it("foo !== bar", function() {
        assert('foo' !== 'bar', 'foo is equal to bar');
    });
});
```

We start off with a describe( ) function and this simply describes the block of the related test i.e. the test suite name. The first argument is a string which describes the test suite while the second argument is an anonymous callback function.

The it( ) function is the actual test we want to run. The first argument is the description of the test and the second argument is another anonymous callback function which will run the assertion test.

Therefore, the describe( ) is the block that contains test while the it( ) is the actual description of the test we are running in the block.

The assert( ) function will perform an assertion test. The first argument is the assertion i.e. JavaScript to run while the second argument is a string if the test failed.

Note: There is no code above the assert because in this simple example we are not putting anything else together to prepare the test. This is a simple example of assertion.

We now have a test simple assertion test created but cannot do anything with it. We would now need to modify the package.json and add a new script inside the scripts block in order to use the Mocha command to run the test file.

```
"scripts": {
    "test": "./node_modules/mocha/bin/mocha"
},
```

We have a script called test. This will run mocha library when we call “npm test” in the terminal.

In the terminal while cd into the project directory we can run the test script by simply running npm test followed by the path to the test file we want to run.

```
$ npm test ./tests/calculator-test.js
```

This is the same as if we wrote the command in the terminal directly:
./node_modules/moch/bin/mocha calculator-test.js

The Mocha library will run the file and output the results in the terminal. The example on the right is a successful test case. We can see the block description is printed which is the string description we passed as the first argument to the describe( ) function.
Below this is the test description which is the string description we passed as the first argument to the it( ) function. This will have a tick next to it if it passed the assertion test.
Finally at the bottom, Mocha displays the total number of passing/failing tests and how long it took to run the test suite.

```
Simple assert
  ✓ foo !== bar


1 passing (6ms)
```

If the assertion failed the second argument in the assert( ) function would display as the failure description text. This can be seen in the example below:

```
var assert = require("chai").assert;

describe("Simple assert", function() {
    it("foo !== bar", function() {
        assert('foo' == 'bar', 'foo is equal to bar');
    });
});
```

```
Simple assert
  1) foo !== bar


0 passing (9ms)
1 failing

1) Simple assert
     foo !== bar:
   AssertionError: foo is equal to bar
    at Context.<anonymous> (tests/calculator-test.js:5:7)
    at processImmediate (internal/timers.js:456:21)
```

This example demonstrates how we can use Mocha and Chai libraries in our projects to perform testing of our project code.

Disclaimer: There are other testing libraries/frameworks that we can use that do the same thing as Mocha and Chai for example Jest, Jasmine, Karma and Puppeteer to name a few and we are not limited to Mocha and Chai.

1.3 Different Types of Mocha and Chai Tests

There are many different types of assertion tests we can perform using the Chai assertion library with Mocha. Below provides an overview of Mocha with Chai.

```
describe("Numeric Calculations", function() {
  it("Addition: Result should be 10", function() {
    assert.equal(5 + 5, 10);
  });

  it("Subtract: Result should be 5", function() {
    assert.equal(10 - 5, 5);
  });
});
```

```
Numeric Calculations
  ✓ Addition: Result should be 10
  ✓ Subtract: Result should be 5
```

In the example to the left we expect to see the result of the equation which is the first argument to be equal to 10 which is the second argument parameter to the .equal( ) method.

We can also have multiple tests within the describe block i.e. multiple tests within a test suite (e.g. an add and subtract tests). The results are also grouped together when outputted to the terminal.

```
  it("Less than", function() {
    assert.isBelow(2, 5);
  });
```

```
Numeric Calculations
  ✓ Addition: Result should be 10
  ✓ Subtract: Result should be 5
  1) Less than


3 passing (11ms)
1 failing

1) Numeric Calculations
     Less than:

    AssertionError: expected 7 to be below 5
    + expected - actual

    -7
    +5

    at Context.<anonymous> (tests/calculator-test.js:19:14)
    at processImmediate (internal/timers.js:456:21)
```

The .isbelow( ) method checks whether a value is below the value. This takes in two arguments where the first argument is the value we are testing the assertion and the second argument is the value we expect to be below.

If the assertion fails this will be logged to the console as seen in the example to the left where 7 is not a value below 5.

```
it("Less than", function() {
  assert.isBelow(7, 5);
});
```

```
  it("Less than", function() {
    assert.isBelow(7, 5, "Number is below 5");
  });
```

```
1) Numeric Calculations
     Less than:

    Number is below 5
    + expected - actual

    -7
    +5

    at Context.<anonymous> (tests/calculator-test.js:19:14)
    at processImmediate (internal/timers.js:456:21)
```

The .equal( ) and .isBelow( ) methods can also take in a third string argument. This allows us to print our own custom error message instead of the default. The example on the left demonstrates this where the AssertError: text is now replaced by the text we added as the third argument to the .equal( ) method.

```
describe("String Manipulations", function() {
  it("String should be 'test'", function() {
    myString = "test"
    assert.equal(myString, "test", "The string is not equal to test");
  });
});
```

The .equal( ) method can also test other data types to equal whatever you expect the value to be. For example we can test a string to equal a string text or a boolean to equal true or false etc. The example on the left provides an example where we test whether the variable myString value is equal to test.

```
  it("Variable is a data type string", function() {
    myString = "testing"
    assert.typeOf(myString, "string");
  });
```

```
  it("Length is 4'", function() {
    myString = "test"
    assert.lengthOf(myString, 4);
  });
```

The .typeof( ) method tests whether the variable passed in as the first argument is the data type expected which his passed in as the second argument to the method.
The .lengthOf( ) method tests whether the length of a variable passed in as the first argument is the expected length which is passed in as the second argument to the method.

END