TDD in JavaScript

Section 2: Test Driven Development

2.1 What is TDD

There may be variations on the concept of Test Driven Development (TDD) depending on the article(s) you may read on this topic but this document will introduced you to a very simple and practical explanation on TDD.

There is generally a workflow to TDD whereby you first write your test based on a use case. This step is done before you write any kind of production code. Normally you will have use cases that you want the software to go through and they can be isolated down to particular areas and those areas is where you start writing your test.

An example would be writing a calculator app. We would create a test and in that test we would create an object for a calculator (which we will not have yet because we are writing the test first) — we may call this calculator object e.g. calc.

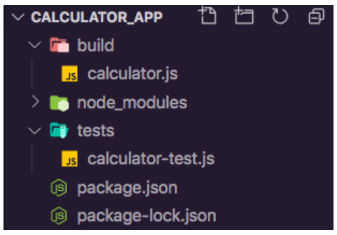
On that calc object we may then create our first test e.g. calc.sum(a, b) which takes in two numbers as parameters. This will write the test and the test would fail because there is no calculator object and this is perfectly fine as this is the first step. This will now force us to write the associated code and we can then see the test will pass once we have the bare minimum amount of logic to make the test pass.
For example, we can pass two 5 as the two parameters which will equal 10 and test the .sum(a, b) would return 10 back because that will pass our test as the bare minimum.

Once we have the code working we can then refactor the code and do whatever manipulation we want to the production code and ensure the test will continue passing at that point forward.

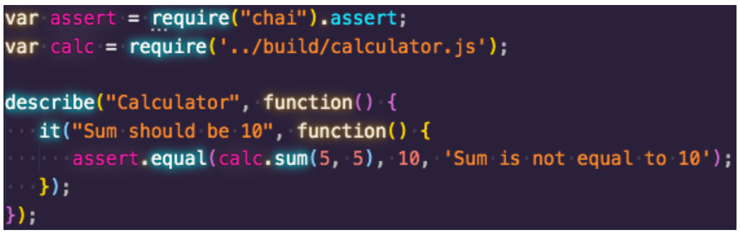
Therefore, the test should be very self-explanatory as to what we want to test i.e. it displays the use case which is very easy to read. Initially it will fail because there is nothing behind the test which will force us to write the code which is why it is called Test Driven Development. We write enough to make the test initially pass. Finally, we then go back to the code and do any refactoring i.e. put in more logic, clean the syntax or re-write the code for performance — building out the production code al the while ensuring the test is passing against that particular use case.

2.2 Writing a TDD Test Case (Example)

Below is an example code for writing a TDD test case for a calculator app. Initially we will have an empty project that is setup with npm to install mocha and chai packages in the project. The project would typically have a tests directory which will hold all of our test files while the build or src directory would have our actual application (production) code.



```
CALCULATOR_APP
  build
    calculator.js
  node_modules
  tests
    calculator-test.js
  package.json
  package-lock.json
```



```
var assert = require("chai").assert;
var calc = require('../build/calculator.js');

describe("Calculator", function() {
  it("Sum should be 10", function() {
    assert.equal(calc.sum(5, 5), 10, 'Sum is not equal to 10');
  });
});
```

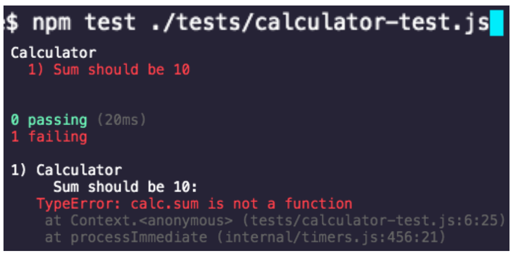
In the calculator-test.js file we would import the assert library object from chai and assign it to a variable which we can call on for our assertions/tests.

We would also create a calc object and assign its value to the calculator.js file which would contain the actual production code we would want to write/test.

The describe function would have a description for our test suite and will wrap all of the various it( ) test cases. In the above example, the test suite will hold all the test case for our calculator app.

The first thing we would probably want as our test case is a sum function. So we would write the description of what the test case is testing followed by the anonymous callback function to perform the assertion. In the example of sum we would want to use assert.equal method to test the sum scenario.

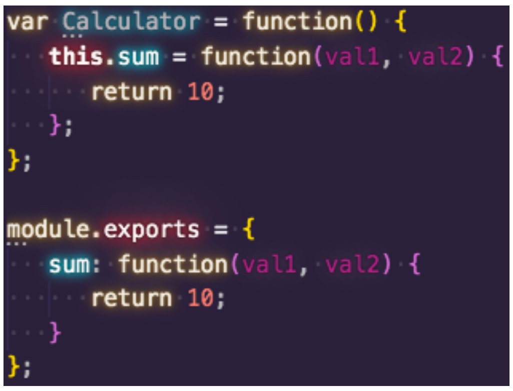
The first argument to the assert.equal( ) method is the actual argument i.e. the function we want to test. We would want to access our calculator object and create a function which we know does not exist yet; however, we are now thinking through the use case of how we want this function to present itself in a readable manner.
This can be seen in the above example where we write as the first actual argument of calc.sum(5, 5) which is how we envision the function to look like on our calc object. We know the .sum( ) will have two parameters which we have hard coded as 5 and 5 and we expect this function to return 10 which is the second argument to the .equal( ) method i.e. the expected argument. The third string argument is optional if we want to display a different text should the test fail.



```
$ npm test ./tests/calculator-test.js
Calculator
  1) Sum should be 10


0 passing (20ms)
1 failing

1) Calculator
     Sum should be 10:
   TypeError: calc.sum is not a function
    at Context.<anonymous> (tests/calculator-test.js:6:25)
    at processImmediate (internal/timers.js:456:21)
```

This creates our first TDD test case. If we now try to run this test we should run into our first failing test which is expected.

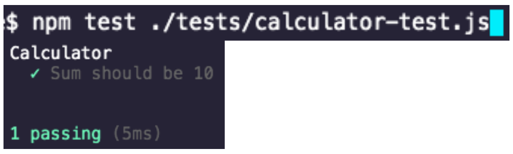
The calc.sum does not exist which is true because our calculator.js file is an empty file and is not exporting any object/functions to use in our test file.



```
var Calculator = function() {
  this.sum = function(val1, val2) {
    return 10;
  };
};

module.exports = {
  sum: function(val1, val2) {
    return 10;
  }
};
```

Our goal now is to build the minimum amount of code just to make this particular test case pass. Below is the example code we can write in the calculator.js file to build this .sum function and export the function out.

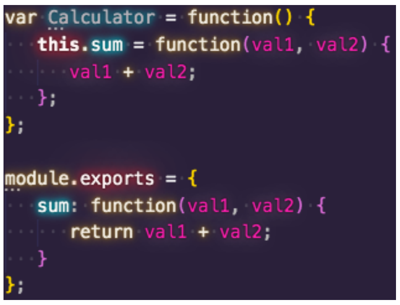
In this example we are hardcoding the return of 10 as the bare minimum to make our test case pass. We would now export this function.



```
$ npm test ./tests/calculator-test.js
Calculator
  ✓ Sum should be 10

1 passing (5ms)
```

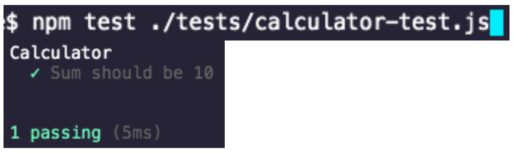
If we now re-run the test suite we should have a passing test case.

We can now go back to our calculator.js file and refactor the code and build out the more complex logic to make the function actually work. In this example it is quite easy to refactor the code but with real world code it would be much more involved to build out logics for your application code.



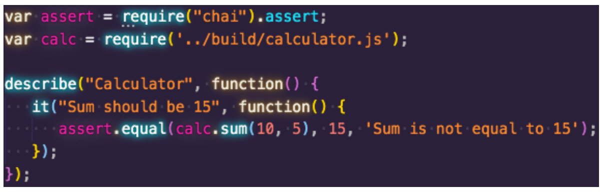
```
var Calculator = function() {
    this.sum = function(val1, val2) {
        val1 + val2;
    };
};

module.exports = {
    sum: function(val1, val2) {
        return val1 + val2;
    }
};
```

From this point on every time we refactor the code it should continue to pass the test case we wrote for this test case.

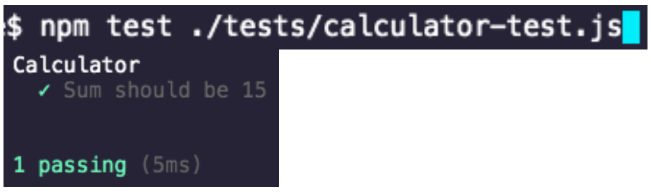


```
$ npm test ./tests/calculator-test.js
Calculator
  ✓ Sum should be 10

1 passing (5ms)
```

We could also change the parameters in the test case to check that it continues to pass with different values. For example 10 + 5 should equal to 15 and we can pass these parameters to the .sum( ) function and the function should return the correct and expected value.



```
var assert = require("chai").assert;
var calc = require('../build/calculator.js');

describe("Calculator", function() {
    it("Sum should be 15", function() {
        assert.equal(calc.sum(10, 5), 15, 'Sum is not equal to 15');
    });
});
```



```
$ npm test ./tests/calculator-test.js
Calculator
  ✓ Sum should be 15

1 passing (5ms)
```

This is an example of a TDD pattern that can be used when using a TDD approach to building applications. To conclude and recap the TDD pattern:

1. Create a test case and fail the test i.e. the object or function does not exist but we have the use case that is explicit.
2. In the production code we would build out the minimal amount of code i.e. the signature for the function and the return value i.e. whatever the test case expects to be returned by the function. The test case should now pass.
3. The final step is to refactor the production code to remove the hard coded return values and add the function logic so that the function is functional and has the necessary logic required to make the function operate in the way it is suppose to work.

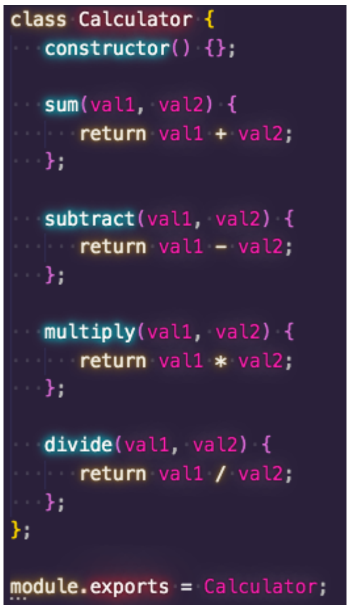
2.3 Test Initialisation and Cleanup

Continuing from the previous examples, we can change the Calculator function variable into a class by adding the class keyword and the constructor function. Once the class has been created with the various methods which would not return a value we can export the whole class itself. This example resembles a more real life code compared to the previous example code where we were exporting different functions.

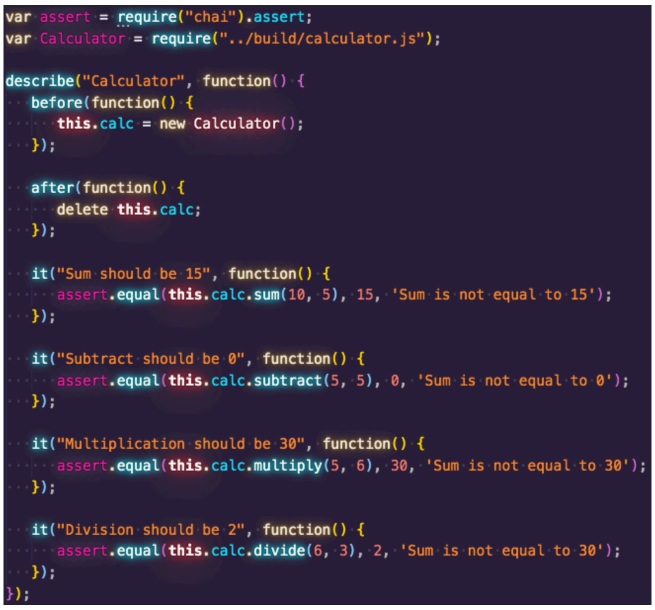
In the calculator-test.js file we would now import this Calculator class and make an instance of this class object before the test runs and delete it after every test case as an initialisation and cleanup before running each test cases.

Mocha and Chai testing library provides us with a before( ) and after( ) functions which we can use to run some code before and after every test case. These both take in a function as an argument parameter.

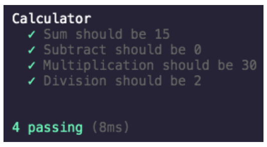
In the before( ) method we can create an instance of the Calculator object and assign it to this.calc variable. We can therefore use this.calc to call on the class methods for testing. In the after( ) method we can delete this.calc object as a cleanup so that the code will not error whenever it tries to recreate the new class object for the next test.



```
class Calculator {
  constructor() {};

  sum(val1, val2) {
      return val1 + val2;
  };

  subtract(val1, val2) {
      return val1 - val2;
  };

  multiply(val1, val2) {
      return val1 * val2;
  };

  divide(val1, val2) {
      return val1 / val2;
  };
};

module.exports = Calculator;
```



```
var assert = require("chai").assert;
var Calculator = require("../build/calculator.js");

describe("Calculator", function() {
  before(function() {
    this.calc = new Calculator();
  });

  after(function() {
    delete this.calc;
  });

  it("Sum should be 15", function() {
    assert.equal(this.calc.sum(10, 5), 15, 'Sum is not equal to 15');
  });

  it("Subtract should be 0", function() {
    assert.equal(this.calc.subtract(5, 5), 0, 'Sum is not equal to 0');
  });

  it("Multiplication should be 30", function() {
    assert.equal(this.calc.multiply(5, 6), 30, 'Sum is not equal to 30');
  });

  it("Division should be 2", function() {
    assert.equal(this.calc.divide(6, 3), 2, 'Sum is not equal to 30');
  });
});
```



```
Calculator
  ✓ Sum should be 15
  ✓ Subtract should be 0
  ✓ Multiplication should be 30
  ✓ Division should be 2

4 passing (8ms)
```

We now have a working test case again but using a class object. This demonstrates the ability to use the before( ) method to do whatever is required to setup the test before running the test cases and the delete( ) method to clean up after each e.g. reset variables so that they do not bleed over to the next test i.e. to start afresh for each test case.

END