Chapter 100: Unit Testing JavaScript

Section 100.1: Unit Testing Promises with Mocha, Sinon, Chai and Proxyquire

Here we have a simple class to be tested that returns a `Promise` based on the results of an external `ResponseProcessor` that takes time to execute.

For simplicity we'll assume that the `processResponse` method won't ever fail.

```
import {processResponse} from '../utils/response_processor';

const ping = () => {
  return new Promise((resolve, _reject) => {
    const response = processResponse(data);
    resolve(response);
  });
}

module.exports = ping;
```

To test this we can leverage the following tools.

1. `mocha`
2. `chai`
3. `sinon`
4. `proxyquire`
5. `chai-as-promised`

I use the following `test` script in my `package.json` file.

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register --require
./test/unit/test_helper.js  --recursive test/**/*_spec.js"
```

This allows me to use `es6` syntax. It references a `test_helper` that will look like

```
import chai from 'chai';
import sinon from 'sinon';
import sinonChai from 'sinon-chai';
import chaiAsPromised from 'chai-as-promised';
import sinonStubPromise from 'sinon-stub-promise';

chai.use(sinonChai);
chai.use(chaiAsPromised);
sinonStubPromise(sinon);
```

`Proxyquire` allows us to inject our own stub in the place of the external `ResponseProcessor`. We can then use `sinon` to spy on that stub's methods. We use the extensions to `chai` that `chai-as-promised` injects to check that the `ping()` method's promise is `fullfilled`, and that it `eventually` returns the required response.

```
import {expect}       from 'chai';
import sinon          from 'sinon';
import proxyquire     from 'proxyquire';

let formattingStub = {
  wrapResponse: () => {}
```

```
}

let ping = proxyquire('../../../src/api/ping', {
  '../utils/formatting': formattingStub
});

describe('ping', () => {
  let wrapResponseSpy, pingResult;
  const response = 'some response';

  beforeEach(() => {
    wrapResponseSpy = sinon.stub(formattingStub, 'wrapResponse').returns(response);
    pingResult = ping();
  })

  afterEach(() => {
    formattingStub.wrapResponse.restore();
  })

  it('returns a fullfilled promise', () => {
    expect(pingResult).to.be.fulfilled;
  })

  it('eventually returns the correct response', () => {
    expect(pingResult).to.eventually.equal(response);
  })
});
```

Now instead let's assume you wish to test something that uses the response from `ping`.

```
import {ping} from './ping';

const pingWrapper = () => {
  ping.then((response) => {
    // do something with the response
  });
}

module.exports = pingWrapper;
```

To test the `pingWrapper` we leverage

1. `sinon`
2. `proxyquire`
3. `sinon-stub-promise`

As before, `Proxyquire` allows us to inject our own stub in the place of the external dependency, in this case the `ping` method we tested previously. We can then use `sinon` to spy on that stub's methods and leverage `sinon-stub-promise` to allow us to `returnsPromise`. This promise can then be resolved or rejected as we wish in the test, in order to test the wrapper's response to that.

```
import {expect}   from 'chai';
import sinon      from 'sinon';
import proxyquire from 'proxyquire';

let pingStub = {
  ping: () => {}
};
```

```
let pingWrapper = proxyquire('../src/pingWrapper', {
  './ping': pingStub
});

describe('pingWrapper', () => {
  let pingSpy;
  const response = 'some response';

  beforeEach(() => {
    pingSpy = sinon.stub(pingStub, 'ping').returnsPromise();
    pingSpy.resolves(response);
    pingWrapper();
  });

  afterEach(() => {
    pingStub.wrapResponse.restore();
  });

  it('wraps the ping', () => {
    expect(pingSpy).to.have.been.calledWith(response);
  });
});
```

Section 100.2: Basic Assertion

At its most basic level, Unit Testing in any language provides assertions against some known or expected output.

```
function assert( outcome, description ) {
    var passFail = outcome ? 'pass' : 'fail';
    console.log(passFail, ': ', description);
    return outcome;
};
```

The popular assertion method above shows us one quick and easy way to assert a value in most web browsers and interpreters like Node.js with virtually any version of ECMAScript.

A good unit test is designed to test a discreet unit of code; usually a function.

```
function add(num1, num2) {
    return num1 + num2;
}

var result = add(5, 20);
assert( result == 24, 'add(5, 20) should return 25...');
```

In the example above, the return value from the function `add(x, y)` or `5 + 20` is clearly `25`, so our assertion of `24` should fail, and the assert method will log a "fail" line.

If we simply modify our expected assertion outcome, the test will succeed and the resulting output would look something like this.

```
assert( result == 25, 'add(5, 20) should return 25...');

console output:

> pass: should return 25...
```

This simple assertion can assure that in many different cases, your "add" function will always return the expected

result and requires no additional frameworks or libraries to work.

A more rigorous set of assertions would look like this (using `var result = add(x,y)` for each assertion):

```
assert( result == 0, 'add(0, 0) should return 0...');
assert( result == -1, 'add(0, -1) should return -1...');
assert( result == 1, 'add(0, 1) should return 1...');
```

And console output would be this:

```
> pass: should return 0...
> pass: should return -1...
> pass: should return 1...
```

We can now safely say that **`add(x,y)`**... **should return the sum of two integers**. We can roll these up into something like this:

```
function test__addsIntegers() {

    // expect a number of passed assertions
    var passed = 3;

    // number of assertions to be reduced and added as Booleans
    var assertions = [

        assert( add(0, 0) == 0, 'add(0, 0) should return 0...'),
        assert( add(0, -1) == -1, 'add(0, -1) should return -1...'),
        assert( add(0, 1) == 1, 'add(0, 1) should return 1...')

    ].reduce(function(previousValue, currentValue){

        return previousValue + current;

    });

    if (assertions === passed) {

        console.log("add(x,y)... did return the sum of two integers");
        return true;

    } else {

        console.log("add(x,y)... does not reliably return the sum of two integers");
        return false;

    }
}
```