Unit Testing

Principles of unit testing

- predictable
- pass or fail
- self-documenting
- single responsibility
- useful error messages
- not integration tests

Predictable

given the same input you'll get the same output

```
describe("Add", function () {

    it("should return the sum of both operands", function () {

        // would not want to pass a random number to add()
        expect(calculator.add(1, 1)).to.be(2);
    });
});
```

Pass or fail:

the outcome of the test is pass or fail and obvious

```
describe("Add", function () {

    it("should return the sum of both operands", function () {
        // expect is an assertion that returns true or false
        expect(calculator.add(1, 1)).to.be(2);
    });
});
```

Self-documenting:

the test should describe what it's meant to do and someone else can quickly determine what is being tested

```
describe("Add", function () {
    /*
     * BDD style testing makes the test's purpose obvious
     * by reading like natural language
     */
    it("should return the sum of both operands", function () {

        expect(calculator.add(1, 1)).to.be(2);
    });
});
```

Single responsibility:

Each test verifies only one thing at a time - if you want to test more things, write more tests

```
describe("Add", function () {

    it("should return the sum of both operands", function () {

        // this test only tests outcome of calling add()
        expect(calculator.add(1, 1)).to.be(2);
    });
});
```

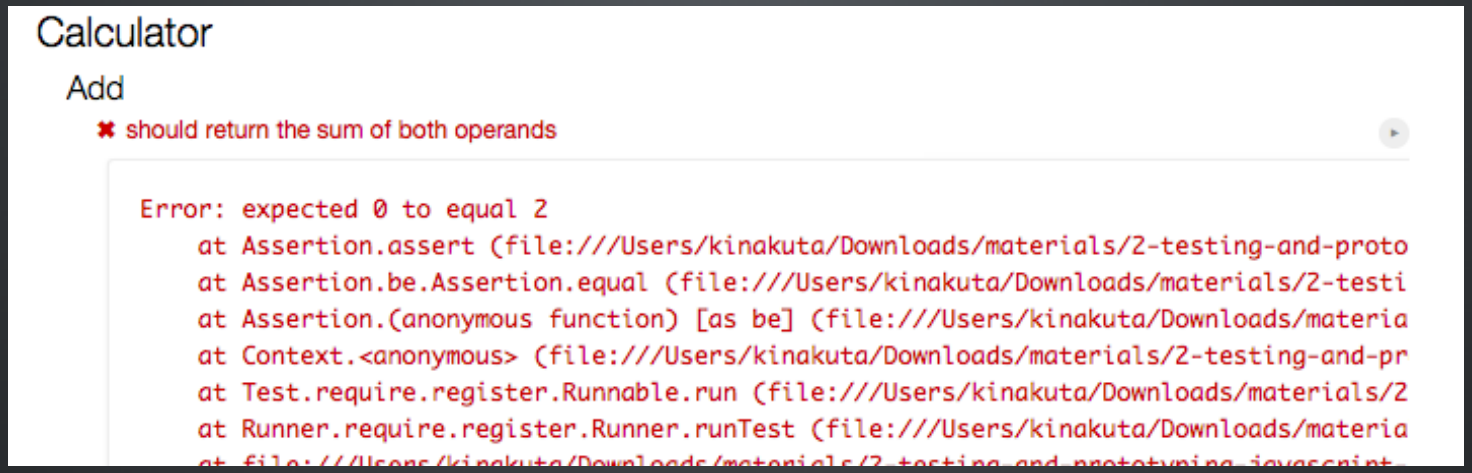
Useful error messages:

descriptive errors should make it easy to tell what went wrong when a unit test fails

```
describe("Add", function () {

    it("should return the sum of both operands", function () {

        expect(calculator.add(1, 1)).to.be(2);
    });
});
```



Not integration tests:

Unit tests are not supposed to be testing multiple components at once.

```
describe("Add", function () {

    it("should return the sum of both operands", function () {

        // only testing one piece of the (Calculator) component
        expect(calculator.add(1, 1)).to.be(2);
    });
});
```

Mocha test framework

what is Mocha?

Mocha is a framework that supports test suites, specs, and multiple test paradigms (both behavior driven development and test driven development.) It offers front-end and back-end (Node) integration, simple asynchronous support, versatile timeouts, slow test identification, and varied test reporters. It supports various assertion libraries: chai, should, expect, et. al.

Setting up a mocha test driver page:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Mocha test harness</title>
    <link rel="stylesheet" href="js/lib/mocha.css" />
</head>
<body>
    <div id="mocha"></div>
    <script src="js/libs/mocha.js"></script>
    <script src="js/libs/chai.js"></script>
    <script>
        var expect = chai.expect;
        mocha.setup('bdd');
        window.onload = function () {
            mocha.run();
        }
    </script>
    <script src="js/spec/hello.spec.js"></script>
</body>
</html>
```

The describe method is used to organize your tests into suites - they can be nested indefinitely

```
describe('View tests', function () {

    describe('App view', function () {

    });

    describe('Nav view', function () {

    });
});
```

Mocha, like other testing frameworks, provides hooks - places to run code before and after tests:

before:

```
before(function () { /* runs once before the suite */ });
```

beforeEach:

```
beforeEach(function () { /* runs before each test */ });
```

afterEach:

```
afterEach(function () { /* runs after each test */ });
```

after:

```
after(function () { /* runs once after the suite */ });
```

example of a basic unit test:

```
// suite description
describe("Calculator", function () {

    var calculator;

    // run this before each test
    beforeEach(function () {
        calculator = new Calculator();
    });

    // section description
    describe("Add", function () {

        // actual test/specification
        it("should return the sum of both operands", function () {
            // using Chai's expect to assert something as true/false
            expect(calculator.add(1, 1)).to.equal(2);
        });
    });
});
```

Mocha is configurable:

```
mocha.setup({
    ui: "bdd",              // other options: "tdd", "exports", or "QUnit"
    reporter: "HTML",       // see http://mochajs.org/#reporters for options
    globals: ["jQuery"],    // to tell mocha this is an accepted global
    timeout: 3000,          // timeout in milliseconds
    bail: false,            // determines if mocha will quit after a failure
    slow: 2000,             // default: 75 - milliseconds before a test is "slow
    ignoreLeaks: true       // useful when making JSONP calls
    grep: ""                // string or regex on which to filter
});
```

Mocha supports a notion of "pending" tests:

```
describe("Calculator", function () {
    var calculator;

    beforeEach(function () {
        calculator = new Calculator();
    });

    describe("Add", function () {

        it("should return the sum of the two operands", function () {
            expect(calculator.add(1, 1)).to.equal(2);
        });

        // pending tests have no callback implementation
        it("should return NaN if passed 0 arguments");
        it("should return NaN if passed 1 argument");
    });
});
```

Mocha supports a simple way to test asynchronous tests using an extra parameter called "done" by convention:

```
describe("Calculator", function () {

    it("makes an asynchronous call", function (done) {
        expect(myAsychronousObject.fetch(function (data) {
            // do some check against the data returned here
            done();
        });
    });
});
```

Test timing and slow tests:

```
describe("Test timing", function () {
    it("should be a fast test", function (done) {
        expect("hi").to.equal("hi");
        done();
    });
    it("should be a medium test", function (done) {
        setTimeout(function () {
            expect("hi").to.equal("hi");
            done();
        }, 40);
    });
    it("should be a slow test", function (done) {
        setTimeout(function () {
            expect("hi").to.equal("hi");
            done();
        }, 100);
    });
    it("should be a timeout failure", function (done) {
        setTimeout(function () {
            expect("hi").to.equal("hi");
            done();
        }, 2001);
    });
});
```

The timeout value is configurable on a per-suite basis:

```
describe("View", function () {

    // override the timeout for the suite
    this.timeout(5000);
    it("should show in 3500ms", function (done) {
        view.show().then(function () {
            done();
        });
    });
});
```

The timeout value is also configurable on a per-test basis:

```
describe("View", function () {

    it("should show in 3500ms", function (done) {

        // timeout configured for a single test
        this.timeout(5000);
        view.show().then(function () {
            done();
        });
    });
});
```

Example of test failures:

```
describe("Test failures", function () {

    // Chai assertion failure
    it("should fail on assertion", function () {
        expect("hi").to.equal("goodbye");
    });

    // un-handled exception failure
    it("should fail on unexpected exception", function () {
        throw new Error();
    });
});
```

Isolating and excluding tests with .only and .skip

```
describe("Test failures", function () {

    describe("isolating tests", function () {

        it.only("should return the sum of the two operands", function () {
            expect(calculator.add(1, 1)).to.equal(2);
        });

        it("should return the difference of the two operands, function () {
            expect(calculate.subtract(2, 1)).to.equal(1);
        });
    });

    describe("excluding tests", function () {

        it.skip("should return the product of the two operands", function () {
            expect(calculate.multiply(2, 2)).to.equal(4);
        });

        it("should return the quotient of the two operands", function () {
            expect(calculate.divide(4, 2)).to.equal(2);
        });
    });
});
```

the Chai assertion library

chaining objects and assertions in Chai

```
/*
 * Chai's BDD interface exposes objects to chain together
 * to make assertions more readable
 */
expect("foo").to.be.a("string");
```

modifying the assertion

```
// using "not" to modify the assertion
expect("foo").to.not.be.an("object");
```

equality checks for objects

```
// this fails as equal checks same reference
expect({ foo: bar }).to.equal({ foo: bar });

// this one passes as deep equal checks object has same properties
expect({ foo: bar }).to.deep.equal({ foo: bar });
```

grouping with and:

```
// "and" is used to group multiple assertions
expect("foo")
    .to.be.a("string").and
    .to.equal("foo").and
    .to.have.length(3);
```

basic value assertions:

ok:

```
expect("foo").to.be.ok;
expect(true).to.be.ok;
expect(false).to.not.be.ok;
```

exist:

```
expect(false).to.exist;
expect(null).to.not.exist;
expect(undefined).to.not.exist;
```

true:

```
expect("foo").to.not.be.true;
expect(true).to.be.true;
```

comparing values:

- equal: strict (===) equality
- eql: deep equality - equivalent to .deep.equal
- above, least, below, most, within, closeTo, match

basic value assertions:

a:

```
// can use "a" or "an" depending on grammatical sense
expect({ foo: bar }).is.an("object");
```

instanceof:

```
var User = function () {},
    Role = function () {};

// checking that something is a specific object type
expect(new User()).is.an.instanceof(User);
expect(new Role()).is.not.an.instanceof(Role);
```

property:

```
// check both for property existence as well as its value
expect({ name: "James"}).to.have.property("name", "James");
```

Sinon.js

a test mock, stub, and spy library that allows the artificial isolation of components and testing of specific behaviors without interaction with the rest of the application

When a component has dependencies, it can be difficult to test it in isolation. Additionally, those external dependencies can slow down test execution, be non-deterministic, and can sometimes be impediments to testing the component at all.

Test doubles is the term used to collectively label the techniques used to observe or replace application behaviors to mitigate the hurdles created by these external dependencies. There are three primary forms of test doubles: spies, stubs, and mocks.

Spies

Spies are functions that wrap methods being tested so the inputs and outputs of those methods can be inspected when testing it. They are useful when we want to know how and when a function is called inside an application.

assign the spy to a variable then make assertions on the spy

```
describe('enhancedDate', function () {

    describe('getDate', function () {

        var getDayNameSpy;
        var getMonthNameSpy;

        before(function () {
            getDayNameSpy = sinon.spy(enhancedDate, 'getDayName');
            getMonthNameSpy = sinon.spy(enhancedDate, 'getMonthName');
        });

        it('uses own methods when requesting formatted output', function () {
            expect(getDayNameSpy).to.have.been.called;
            expect(getMonthNameSpy).to.have.been.called;
        });

    });

});
```

Stubs

Stubs are a type of spy that replaces the functionality of a method with new behavior. This is particularly useful for test isolation, by essentially replacing external dependencies with pre-programmed behavior.

Mocks

Mocks are a combination of spies and stubs that additionally verify expected function behavior during execution.

stubs allow the replacement of the normal behavior of a method

```
// these are contrived examples to demonstrate how to build a stub
describe('calculator', function () {

    var multiplyStub;
    var divideStub;

    before(function () {
        multiplyStub = sinon.stub(calculator, 'multiply', function (a, b) {
            return a + b;
        });
        divideStub = sinon.stub(calculator, 'divide').returns(5);
    );

    it('multiply returns the sum of two numbers', function () {
        expect(multiplyStub(1, 2)).to.equal(3);
    });

    it('divide always returns same value', function () {
        expect(divideStub(4, 5)).to.equal(5);
    });

```

Backbone.js Testing

Plan, architect, and develop tests for Backbone.js applications using modern testing principles and practices.

Ryan Roemer

[PACKT] open source