Testing Node.js Applications

Agenda

- What is testing and why do it?
- Testing Terms & Concepts
- Types of Tests
- Test Driven Development
- Test Frameworks
- Jasmine
- Implementing tests in a Node.js app

Why are we writing tests?

- Quickly verify that your code works the way you think it does
- Quickly isolate and fix bugs
- Deliver quality software to your client (less bugs/broken features)

How do tests help you as a developer?

In a project starting from scratch (a greenfield project), you can move quickly because there are very few features. With less features, it's easy to just "test in the browser" by hitting URLs and submitting forms. With this method you are manually verifying that your software works.

How do tests help you as a developer?

- As your codebase grows you will add more features
- Requirements change and you will need to change the behavior of existing features
- Sometimes changing one feature can break another

How do tests help you as a developer?

This means that every time you make a change, you might have to manually test several features which can be time consuming. The larger your codebase, the more manual testing you will have to do. This is where automated tests come in.

How do tests help you as a developer?

Automated tests allow you to run a single command over the command line and verify that multiple features are working as expected.

How do tests help you as a developer?

When a new bug is found it can be closed with a fix and an accompanying test for that specific scenario, verifying that the fix actually did eliminate the bug. This is known as regression testing.

How do tests help you as a developer?

- Testing forces you to think critically about your code, resulting in better design
- Testing encourages writing decoupled code, resulting in better design
- Tests are great documentation

Downsides of Testing

Testing doesn't come without a cost. Writing and maintaining tests can be time consuming, but the idea is that the tests save you time in the long run.

Don't even bother writing tests if you aren't going to maintain them.

Can't I just build my app without tests?

- You can and when creating an MVP, this is definitely advisable just to get started
- Larger software development shops and consultancies write test suites, which is why we teach you the basics of how to write them!
- This will be hard to learn, but is a great skill to get familiar with and something you should have in your resume

Testing Terms and Concepts

Tests are usually split into several different suites which are made up of several tests cases.

Testing Terms and Concepts

- Test Case - Verifies a small piece of functionality works
- Test Suite - Verifies several small pieces of functionality work. Made up of several test cases.

Test suites group similar test cases allowing you to organize your tests.

Testing Terms and Concepts

It is useful to organize your tests into suites so you can focus on one piece of your application at a time. Suites give you the ability to run a particular set of tests and ignore others.

Testing Terms and Concepts

You might have a test suite for verifying user login works, and a separate test suite for verifying blog post create/edit/update works. Within each suite, there are several test cases.

Testing Terms and Concepts

TestSuite: BlogPostsController

- TestCase: GET blog post responds 200 OK
- TestCase: DELETE blog post deletes blog post
- TestCase: UPDATE blog post properly updates a blog post

Types of Tests

There are several different types of tests you can write, each with a different purpose.

- Unit Tests: Test a very small unit, usually a function or method
- Functional Tests/Integration Tests: Test a piece of functionality. May hit the database or the web server.
- Acceptance Tests: Tests your application as a user would use it. Hits URLs, clicks buttons, verifies correct text is displayed on screen.

Unit Tests

The unit test is the most popular and most important kind of test. Because it tests small units, you can easily isolate a bug. With functional tests, you know something is broken but it's hard to tell where sometimes.

- Never hit the DB, server, filesystem etc.
- Very fast
- Best for finding exactly where a bug is

Functional Tests

A functional test might test an entire feature. Submitting the login form might run code from several classes consisting of several units.

Even if you have unit tests covering everything, it is useful to write functional tests to make sure the app actually works as expected and assert that the right data is going in the database or some other area that can't be touched by a unit test.

Functional Tests

Functional tests are also good for making sure you have your "wiring" code correct. All your units might be working in isolation but you might wire them together wrong.

Functional Tests

If every piece of a car engine has been individually tested, that doesn't mean the car will run when you put the pieces together. The pieces have to be put together correctly and even if they are put together correctly, there may be some unforeseen error in their interaction.

Acceptance Tests

Acceptance tests are from the user perspective, there is no knowledge of the inner workings of the code in acceptance tests. You don't "new up" classes or mock dependencies. You simply hit URLs, click on buttons, and assert that the page contains the text you expected it to.

Acceptance Tests

- Easiest to write
- Not good for finding out exactly what has gone wrong
- Very slow

Acceptance Tests

If a codebase is untestable due to the code being highly coupled (spaghetti code), people will start with acceptance tests so they can begin refactoring with at least some confidence.

Test Driven Development

- TDD uses Unit Tests
- TDD or Test Driven Development is the practice of writing the test code before the implementation code
- In TDD, you do everything in small bite sized steps
- The basic cycle of TDD is red, green, refactor

The Cycle of TDD

- Think - What code comes to next to accomplish your goal?
- Red - Write a failing test
- Green - Do the bare minimum to make the test pass
- Refactor - Clean up your code
- Repeat

Testing Frameworks

You could write code that tests your production code without using a testing framework but it would be tedious and verbose.

Example:

```
class TestCalculator {
  let calculator = new Calculator();
  if (calculator.add(5,5) === 10) {
    console.log('success');
  } else {
    console.log('fail');
  }
}
```

Testing Frameworks

While that would work, it would quickly get out of hand. There are many great testing frameworks that will take care of the leg work for you.

Testing Frameworks

- Jasmine - JavaScript
- PHPSpec - PHP
- RSpec - Ruby

Jasmine

- Tests are all about making "assertions"
- Assertions state that something is true or not true
- Assertions are never questions

Jasmine

Two important but not unique concepts in Jasmine are expectations and matchers.

Expectations are built with the `expect` function which takes a value called the actual. It is chained with a Matcher function, which takes the expected value.

```
let someVar = true;
expect(someVar).toBe(true);
// someVar is the ACTUAL value
// true is the EXPECTED value
```

Jasmine

Every matcher has a specific use case and makes comparisons in different ways.

```
expect(true).toBe(true) // compares using === (identical) operator
expect(1).toEqual('1')  // compares using the == operator
expect(1).toBe('1')     // FAILS
expect(1).toEqual('1')  // Passes
```

Jasmine

Some of the other matchers include:

```
expect(someVar).toBeNull()              // compares against null
expect(someVar).toBeDefined()           // compares against 'undefined'
expect(someArray).toContain(someItem)   // used for asserting an item is in an array
```

Jasmine

Take five minutes to review the Jasmine docs and see some of the other matchers.

http://jasmine.github.io/2.5/introduction.html#section-Included_Matchers

Jasmine

You may have noticed the "describe" and "it" syntax in the Jasmine docs.

Jasmine uses "describe" to create a test suite and group related test cases. Each test suite contains multiple tests cases (called specs in jasmine). Specs are created using the "it" function.

Jasmine

Let's set up Jasmine in an empty project and look at some example code

```
cd ~/Desktop
mkdir app
cd app
npm init -f
npm install --save-dev jasmine
```

Jasmine

We now have Jasmine installed. Let's add scripts to our package.json to initialize it.

```
// Inside of package.json
  "scripts": {
    "test-init": "jasmine init",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Then run the command:

```
$ npm run test-init
```

Jasmine

Let's take a look at what Jasmine just did for us:

- Created directory "spec" for holding test code
- Created file spec/support/jasmine.json and provided default config

Jasmine

The jasmine.json file allows us to configure Jasmine.

- specs (test files) should be located in the spec directory
- spec files should end in spec or Spec.js
- test helpers will be located in the helpers directory which we don't need right now

```
{
  "spec_dir": "spec",
  "spec_files": [
    "**/*[sS]pec.js"
  ],
  "helpers": [
    "helpers/**/*.js"
  ]
}
```

Jasmine

Now let's setup the script in package.json to run the tests.

```
// Inside of package.json
"scripts": {
    "test-init": "jasmine init",
    "test": "jasmine"
},
```

Then run the command:

```
$ npm run test  // or you can just say npm test
```

Jasmine

Now that we have Jasmine set up. Let's go through a TDD cycle.

Our goal will be to create a calculator class that can add two numbers. We will do everything in very small steps.

Jasmine

Create a calculator class that can add two numbers

Step 1: Create the spec

```
// spec/CalculatorSpec.js
describe("Calculator", function() {
  var Calculator = require('../lib/Calculator.js');

  it("should add two numbers", function() {
    // nothing here yet
  });
});
```

Jasmine

Create a calculator class that can add two numbers

Step 2: Run the tests

Error: Cannot find module '../lib/Calculator.js'

We have achieved "RED"

Jasmine

Create a calculator class that can add two numbers

Step 3: Create the Calculator.js file

```
// lib/Calculator.js
'use strict'
class Calculator {

}

module.exports = Calculator;
```

Jasmine

Create a calculator class that can add two numbers

Step 4: Run the tests

We have achieved "GREEN" but we don't have any assertions yet

Jasmine

Create a calculator class that can add two numbers

Step 5: Add an assertion

```
// spec/CalculatorSpec.js
describe("Calculator", function() {
  var Calculator = require('../lib/Calculator.js');
  var calculator = new Calculator;

  it("should add two numbers", function() {
    expect(calculator.add(5,5)).toBe(10);
  });
});
```

Run the tests `$ npm test`

We have achieved "RED" again

Jasmine

Create a calculator class that can add two numbers

Step 6: Define the method

```
// spec/Calculator.js
class Calculator {
  add() {
    return 10;
  }
}
```

Run the tests `$ npm test`

We have achieved "GREEN" again but we faked it so we need to refactor

Jasmine

Create a calculator class that can add two numbers

Step 7: Refactor

```
// spec/Calculator.js
class Calculator {
  add(a,b) {
    return a + b;
  }
}
```

Run the tests `$ npm test`

We have achieved "GREEN" again and the method works so we are done.

Jasmine

The code was very simple so it might feel like the steps we took were too small but this is how TDD is supposed to be done.

Jasmine

Exercise

Try implementing a subtract function that subtracts two numbers. Use TDD when implementing the feature.

Jasmine

Jasmine can generate some example specs for us. Let's go ahead and generate the examples.

```
//package.json
"scripts": {
    "test-init": "jasmine init",
    "test-examples" "jasmine examples",
    "test": "jasmine"
},
```

then run

```
`$ npm run test-examples`
```

Jasmine

Let's take a look at what Jasmine just did for us:

- Created lib/jasmine_examples directory with some classes
- Created spec/jasmine_examples directory with specs
- Created helpers/jasmine_examples directory for holding test helpers

Jasmine

Go ahead and the run the tests.

```
$ npm test
```

Jasmine

Take a few minutes to look over the specs in `spec/jasmine_examples` and classes in `lib/jasmine_examples`.

Try and understand what is going on.

Jasmine

Note that you can create your own matchers in Jasmine. Custom matchers make your test code easier to understand.

See the example in:

`spec/helpers/jasmine_example/SpecHelper.js`

Jasmine

Looking at open source code and their tests is a great way to learn how to test.

We aren't going to review these tests in class but I recommend you check out the tests for the node.js framework we are using (express).

https://github.com/expressjs/express/tree/master/test

Jasmine

Now that we have a basic understanding of Jasmine, let's set it up in our blogger app.

Jasmine

Jasmine used to be difficult to use within a node.js app but any version of Jasmine greater than 2.0.0 works fine with node so we will just use the regular Jasmine package.

Within your blogger app...

```
npm install jasmine --save-dev
```

Then add the initialization and testing commands into the package.json scripts

Jasmine

Our blogger app doesn't have a lot of business logic yet. We basically have models and routes which means there isn't much to unit test here. Instead of writing unit tests, let's write some acceptance tests.

Jasmine

- Add the test below
- Fire up your app: `$ npm run start`
- In a new tab in Terminal run `$ npm test`

```
// spec/routes/postsSpec.js
describe("post routes", function() {

  var request = require('request');

  it("should list blog posts", function(done) {
    request("http://localhost:3000/posts", function(error, response, body){
      expect(body).toMatch("Great Post 1");
      expect(body).toMatch("Great Post 2");
      done();
    });
  });
});
```

Jasmine

Notice that we did not require any classes from our application code. We are doing acceptance testing here, simply requesting a URL and asserting the page contained what we expected it to.

We did require the request class but we didn't write that code and we need it to make the request.

Writing Testable Code

I highly recommend you watch the "Clean Code Talks", presented by the creator of Angular.js.

I specifically recommend the talk on Test Driven Development.

It talks about writing testable code which is very important to writing tests.

https://www.youtube.com/watch?v=wEhu57pih5w

Resources

Coupling
Cycle of TDD
Unit Testing
Types of Tests
Blogger App