Unit testing and test driven development

with AngularJS

Contents of today's session

- Unit testing
  - What is unit testing?
  - What tools to use
  - How to write tests
- Test driven development
  - What is test driven development?
  - Advantages and disadvantages
- Live code demo
  - Test driven development of a simple AngularJS app

What is unit testing?

Unit testing is a software testing method by which individual units of source code are tested.

Tools for unit testing with AngularJS

- Open source testing framework for Javascript
- Tests are written with Jasmine

Tools for unit testing with AngularJS

- Open source testing framework
- Runs the tests in the browser
- Reports the results
- Configured through a js config file

Tools for unit testing with AngularJS

- Javascript task runner
- Lots of modules available in NPM
- Configured with Gruntfile.js

Writing Jasmine specs

http://jasmine.github.io/2.0/introduction.html

Jasmine

Powered by PIVOTAL LABS

Jump To: ajax.js boot.js custom_equality.js custom_matcher.js introduction.js node.js python_egg.py ruby_gem.rb upgrading.js

introduction.js

Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests. This guide is running against Jasmine version 2.0.0.

Standalone Distribution

The releases page has links to download the standalone distribution, which contains everything you need to start running Jasmine. After downloading a particular version and unzipping, opening `SpecRunner.html` will run the included specs. You'll note that both the source files and their respective specs are linked in the `<head>` of the `SpecRunner.html`. To start using Jasmine, replace the source/spec files with your own.

Suites: `describe` Your Tests

A test suite begins with a call to the global Jasmine function `describe` with two parameters: a string and a function. The string is a name or title for a spec suite – usually what is being tested. The function is a block of code that implements the suite.

Specs

Specs are defined by calling the global Jasmine function `it`, which, like `describe` takes a string and a function. The string is the title of the spec and the function is the spec, or test. A spec contains one or more expectations that test the state of the code. An expectation in Jasmine is an assertion that is either

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

Writing Jasmine specs

A ‘describe’ block describes a unit:

```
1  'use strict';
2
3  describe('The unit', function() {
4
5      // Tests
6
7  });
```

Writing Jasmine specs

Inside each describe are ‘it’ blocks:

```
1   'use strict';
2
3   describe('The unit', function() {
4
5       it('Should test something', function() {
6           // Testing stuff
7       });
8
9       it('Should test something else', function() {
10          // Testing more stuff
11      });
12
13  });
```

Writing Jasmine specs

Each it contains expectations:

```
4   it('Should do something', function() {
5
6       expect(true).toBe(true);
7
8       expect('true').toBeTruthy();
9
10      expect({foo: 'bar'}).toEqual({foo: 'bar'});
11
12      expect(false).not.toBe(true);
13
14      expect(foo).toBeUndefined();
15
16  });
```

Writing Jasmine specs

The 3 A's:

```
3   it('Should test something', function() {
4       // Arrange
5       // Act
6       // Assert
7   });
```

```
4   describe('The multiplyBy function', function() {
5       it('Should multiply controller.mult by the input and return the result', function() {
6
7           // Arrange
8           controller.mult = 5;
9
10          // Act
11          var t1 = controller.multiplyBy(4);
12          var t2 = controller.multiplyBy(6);
13
14          // Assert
15          expect(t1).toBe(20);
16          expect(t2).toBe(30);
17
18      });
19  });
```

Writing Jasmine specs

beforeEach can be used to arrange the same state for multiple tests:

```
4   describe('The controller in error state', function() {
5
6       beforeEach(function() {
7           controller.state = 'error';
8           controller.initialize();
9       });
10
11      it('Should set the the correct message for error code 123', function() {
12         controller.error.errorCode = 123;
13         expect(controller.errorMsg()).toBe('Dataset is empty!')
14      });
15
16      it('Should set the the correct message for error code 456', function() {
17         controller.error.errorCode = 456;
18         expect(controller.errorMsg()).toBe('Invalid date format!')
19      });
20
21      it('Should set the the correct message for unspecified error codes', function() {
22         controller.error.errorCode = 'foo';
23         expect(controller.errorMsg()).toBe('Uspecified error! (code \'foo\'');
24         controller.error.errorCode = 'bar';
25         expect(controller.errorMsg()).toBe('Uspecified error! (code \'bar\'');
26      });
27
28  });
```

Writing Jasmine specs

Use a spy to test function calls:

```
4   it('Should call a few functions, but only once!', function() {
5
6       spyOn(controller, 'checkState');
7       spyOn(controller, 'setMsg');
8
9       controller.doSomething();
10
11      expect(controller.checkState).toHaveBeenCalled();
12      expect(controller.setMsg).toHaveBeenCalledWith(123, 'foo');
13
14      expect(controller.checkState.calls.count()).toEqual(1);
15      expect(controller.setMsg.calls.count()).toEqual(1);
16
17  });
18
```

Writing Jasmine specs

More espionage:

```
4   it('Should explain different methods of spying', function() {
5
6       // Simple spy
7       spyOn(controller, 'foo');
8
9       // Spy on the function and let the call through
10      spyOn(controller, 'foo').and.callThrough();
11
12      // Spy on the function with and return a specific value
13      spyOn(controller, 'foo').and.returnValue({foo: 'bar'});
14
15      // Spy on the function and delegate to the supplied function
16      spyOn(controller, 'foo').and.callFake(function(input) {
17          return input * 15;
18      });
19
20  });
21
```

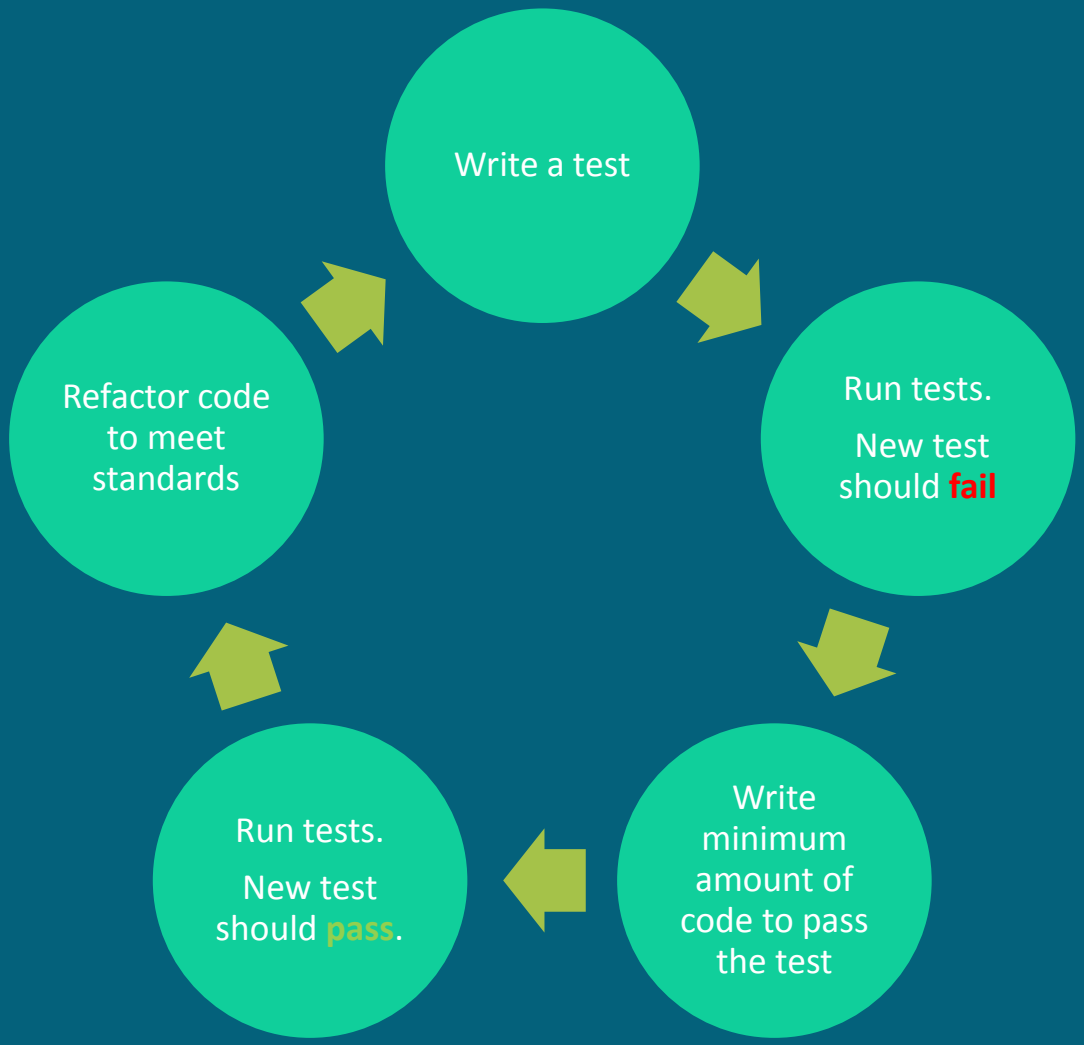
Writing quality tests

- Test components separately
- Keep your units small
- Keep your tests small
- Use mocks whenever possible
- Code coverage =/= test quality

What is test driven development?

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle.

The TDD development cycle



Pros and cons of TDD

Advantages

- Final product is fully tested
- Forces modular and well tested code
- Code is easier to maintain and refactor
- Helps with finding bugs early in development

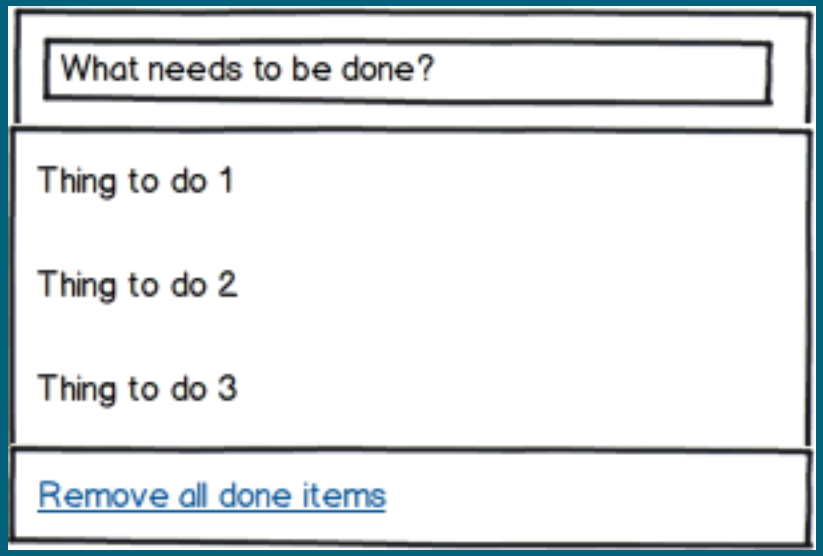
Disadvantages

- Longer development cycle
- More refactoring when the design changes during development

Live coding demo

Simple todo list

- Data in local storage
- Adding new items
- Mark items as done
- Remove items
- Remove all done items
- Edit item title



Live coding demo – project setup

https://github.com/lvandiest/todo-list

- Bower dependencies
- Node modules
- Gruntfile
- Karma config