

Justifying test-driven JavaScript development

- JavaScript is a first-class citizen in our products.
- Modern web applications are predominantly written in JavaScript with some markup.
- JavaScript usage is growing, even on the server-side.
- Production quality code should be tested.
 - Unit, integration, and functional/acceptance testing.
- *Don't practice reckless development!*

Quick review of test-driven development

- Use unit tests to drive development and design.
- Write the test first, then the code.
 - See the test fail, then make it pass.
 - Importance of spiking before test-first development.
- Test coverage of your code remains high because of test-first approach.
- A fast test suite is typically run frequently.

Benefits of test-driven development

- Design tool.
- Helps build confidence.
- Executable documentation of the code base.
 - Tests infer the intent of the code.
- Code base is continually executed when test suites are run in continuous integration environments.
 - Avoid code rot.

The test-driven development cadence

Write code to make
the test pass



Refactor code
and tests

Start with a failing
test

The importance of “spiking”

- Test-driven development is grounded in the assumption that you know your tools and what you are building.
- When unsure about how the solution should proceed, use **spike solutions** to learn more about what you’re attempting to do.
- Spike solutions are *not* production code.
- Spike solutions are typically thrown away. Value is in the problem domain learning that takes place.

Karma

- JavaScript test runner that integrates with a number of browser runners.
- Dependent on node.js, distributed as a node package.
- Command line tool, but also integrated into JetBrains WebStorm IDE.

→ calculator git:(master) x `karma start`

INFO [karma]: Karma v0.10.8 server started at <http://localhost:9876/>

INFO [launcher]: Starting browser PhantomJS

INFO [PhantomJS 1.9.2 (Mac OS X)]: Connected on socket TbzZHmxXJQ3aKLGcIIe1

PhantomJS 1.9.2 (Mac OS X): Executed 12 of 12 SUCCESS (0.022 secs / 0.003 secs)

Phantom.js

- Headless WebKit browser runner, scriptable with a JavaScript API
- Native support for various web standards
 - DOM, Canvas, and SVG
 - CSS selectors
 - JSON

Introducing Jasmine 2.0

- Testing framework
 - Suites possess a hierarchical structure
 - Tests as specifications
 - Matchers, both built-in and custom
 - Spies, a test double pattern

Jasmine suite

- Group specifications together using nested **describe** function blocks.
- Also useful for delineating context-specific specifications.

```
describe("render", function() {  
    . . .  
});
```

Jasmine specification

- Specifications are expressed with an **it** function.
- Use descriptive descriptions. Descriptions should read well in the test run report.

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

```
  it("tabName should be set to 'tr'", function() {  
    . . .  
  });
```

```
});
```

Jasmine expectations

- Expectations are expressed with the **expect** function.

```
describe("render", function() {  
  it("tabName should be set to 'tr'", function() {  
    expect(view.tagName).toBe('tr');  
  });  
});
```

Jasmine setup using `beforeEach`

```
describe("PintailConsulting.ToDoListView", function() {  
    var view;  
  
    beforeEach(function(){  
        view = new PintailConsulting.ToDoListView();  
    });  
  
    it("sets the tagName to 'div'", function() {  
        expect(view.tagName).toBe('div');  
    });  
});
```

Jasmine tear down using `afterEach`

```
describe("PintailConsulting.ToDoListView", function() {  
    var view;  
  
    beforeEach(function(){  
        view = new PintailConsulting.ToDoListView();  
    });  
  
    afterEach(function(){  
        view = null;  
    });  
  
    it("sets the tagName to 'div'", function() {  
        expect(view.tagName).toBe('div');  
    });  
});
```

Sharing variables using `this` keyword

- Each specification's `beforeEach/it/afterEach` is given an empty object that can be referenced with `this`.

```
beforeEach(function() {  
    this.view = new PintailConsulting.ToDoListView();  
});
```

```
it("tagName should be set to 'div'", function() {  
    expect(this.view.tagName).toBe('div');  
});
```

Jasmine matchers

- not
- toBe
- toEqual
- toMatch
- toBeDefined
- toBeUndefined
- toBeNull
- toBeTruthy
- toBeFalsy
- toContain
- toBeLessThan
- toBeGreaterThan
- toBeCloseTo
- toThrow

Jasmine custom matchers

```
beforeEach(function() {  
  this.addMatchers({  
    toBeLessThan: function(expected) {  
      var actual = this.actual;  
      var notText = this.isNot ? " not" : "";  
  
      this.message = function () {  
        return "Expected " + actual + notText +  
          " to be less than " + expected;  
      }  
      return actual < expected;  
    }  
  });  
});
```


Loose matching with `jasmine.any`

- Accepts a constructor or “class” name as an expected value.
- Returns **true** if the constructor matches the constructor of the actual value.

```
var spy = jasmine.createSpy(My.Namespace, 'foo');  
foo(12, function(x) { return x * x; });  
expect(spy).toHaveBeenCalledWith  
  (jasmine.any(Number), jasmine.any(Function));
```

Partial matching

- Use the `jasmine.objectContaining` function for partial matching.
- Allows writing expectations where you only care about certain key/value combinations on the actual result.

```
expect(result).toEqual(jasmine.objectContaining({  
  someProperty: 'foobar',  
  someOtherProperty: 12  
}));
```

Jasmine spies

- Test double pattern.
- Interception-based test double mechanism provided by the Jasmine library.
- Spies record invocations and invocation parameters, allowing you to inspect the spy after exercising the SUT.
 - Very similar to mock objects.
- Jasmine 2.x spy syntax is quite different than version 1.3.

Jasmine spy creation

- **spyOn(object, functionName)**: Create a spy on an existing object and function name, as a string.
- **createSpy(identity)**: Creates a “bare” spy. No implementation is tied to this type of spy.
- **createSpyObj(identity, propertiesArray)**: Creates a mock with multiple spies. An array of strings, each string representing a spy.

Jasmine spy usage

Spying and verifying invocation

```
var spy = spyOn(dependency, "render");  
systemUnderTest.display();  
expect(spy).toHaveBeenCalled();
```

Spying, verifying invocation and argument(s)

```
var spy = spyOn(dependency, "render");  
systemUnderTest.display("Hello");  
expect(spy).toHaveBeenCalledWith("Hello");
```

Jasmine spy usage

Spying, verifying number of invocations and arguments for each call

```
var spy = spyOn(L, "circle").and.callThrough();
mapView.processResults(earthquakeJsonResults);
expect(spy).toHaveBeenCalled()
expect(circleConstructorSpy.calls.count()).toBe(2);
expect(circleConstructorSpy.argsForCall[0][0])
    .toEqual([56.6812, -155.0237])
```

Jasmine spy usage

- **and.callThrough()**: Allows the invocation to passthrough to the real subject.
- **and.returnValue(result)**: Return a hard-coded result.
- **and.callFake(fakeImplFunction)**: Return a dynamically generated result from a function.
- **and.throwError(error)**: Throws an error with supplied message.

Jasmine spy tracking features

- `calls.any()`: Returns a boolean result, `true` if the spy was called, `false` if it was not called.
- `calls.count()`: Return the number of times a spy was called.
- `calls.argsFor(index)`: Returns arguments for call specified by index.
- `calls.allArgs()`: Returns the arguments to all calls.

Jasmine spy tracking features

- **calls.all()**: Returns the `this` context and the arguments for all calls.
- **calls.mostRecent()**: Returns the `this` context and the arguments for the most recent call.
- **calls.first()**: Returns the `this` context and the arguments for the first call.
- **calls.reset()**: Clears all tracking on the spy.

Earthquakes map demonstration

- Pulls live data feed from USGS, formatted in GeoJSON format.
- Plots each earthquake event as a red circle.
- Binds a popover annotation to the earthquake event circle, showing detailed information.
- Existing solution is fully tested with Jasmine specs.
- Uses Leaflet.js for mapping. jQuery is used for AJAX.

Earthquakes map demonstration

User story: Earthquake events are colored per their magnitude

Acceptance criteria:

1. Show that earthquakes of magnitude less than 1.0, the circle boundary is #217DBB and the fill color is #3498DB.
2. Show that earthquakes of magnitude less than 2.0 but greater than or equal to 1.0, the circle boundary is #148F77 and the fill color is #1ABC9C.
3. Show that earthquakes of magnitude less than 3.0 but greater than or equal to 2.0, the circle boundary is #25A25A and the fill color is #2ECC71.
4. Show that earthquakes of magnitude less than 4.0 but greater than or equal to 3.0, the circle boundary is #C29D0B and the fill color is #F1C40F.
5. Show that earthquakes of magnitude less than 5.0 but greater than or equal to 4.0, the circle boundary is #BF6516 and the fill color is #E67E22.
6. Show that earthquakes of magnitude greater than 5.0, the circle boundary is #BA140A and the fill color is #EB1A0C.

karma-coverage

- Test coverage plugin for karma
- <https://github.com/karma-runner/karma-coverage>

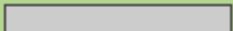
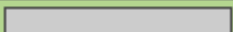
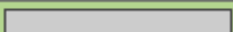
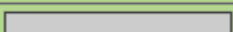
npm install karma-coverage --save-dev

- Run karma with coverage configured (*karma.conf.js*)
- Generate reports using istanbul report
 - Reports saved to the *coverage* subdirectory

Code coverage report

Code coverage report for **All files**

Statements: **100%** (139 / 139) Branches: **100%** (8 / 8) Functions: **100%** (44 / 44) Lines: **100%** (139 / 139)

File ▲	Statements	Branches	Functions	Lines
scripts/ 	100.00% (24 / 24)	100.00% (0 / 0)	100.00% (8 / 8)	100.00% (24 / 24)
scripts/data-builders/ 	100.00% (2 / 2)	100.00% (0 / 0)	100.00% (1 / 1)	100.00% (2 / 2)
scripts/models/ 	100.00% (6 / 6)	100.00% (0 / 0)	100.00% (5 / 5)	100.00% (6 / 6)
scripts/views/ 	100.00% (107 / 107)	100.00% (8 / 8)	100.00% (30 / 30)	100.00% (107 / 107)

Generated by [Istanbul](#) at Thu Mar 06 2014 23:21:26 GMT-0600 (CST)

Unit testing tips

- Strive for one assertion per example.
 - Allows all assertions to execute.
 - Each assertion runs in a clean SUT setup.
- Avoid making live AJAX calls in your unit tests/specs.
 - Spy/intercept the low-level AJAX invocations (jQuery.ajax)
 - Use fixture data for testing AJAX callbacks.

How do we sustain test-driven development?

- Practice, practice, practice!
- Test-first development takes time to learn and become proficient with.
- Pair programming, even in remote situations.
 - Screenhero, Hangouts, Skype
- Continuous integration server.
 - Run your test suites often, preferably on every commit.

Functional/acceptance testing

- Very important part of the testing portfolio.
- Many tools support testing web-based user interfaces today.
 - Selenium, Geb, Capybara, Cucumber{Ruby|jvm|js}, SpecFlow (.NET), Protractor.js, Concordian, spock
- You should strongly consider adding functional/acceptance testing in your testing portfolio.
- Covers areas of code that unit testing cannot cover.

Tool references

- <http://jasmine.github.io/2.0/introduction.html>
- <http://karma-runner.github.io/0.12/index.html>
- <http://phantomjs.org/>
- <https://github.com/karma-runner/karma-jasmine>

Recommended reading

- Secrets of the JavaScript Ninja - John Resig and Bear Bibeault
- JavaScript: The Good Parts - Douglas Crockford
- Test-Driven JavaScript Development - Christian Johansen