JavaScript Test-Driven Development with Jasmine and Karma

Christopher Bartling

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

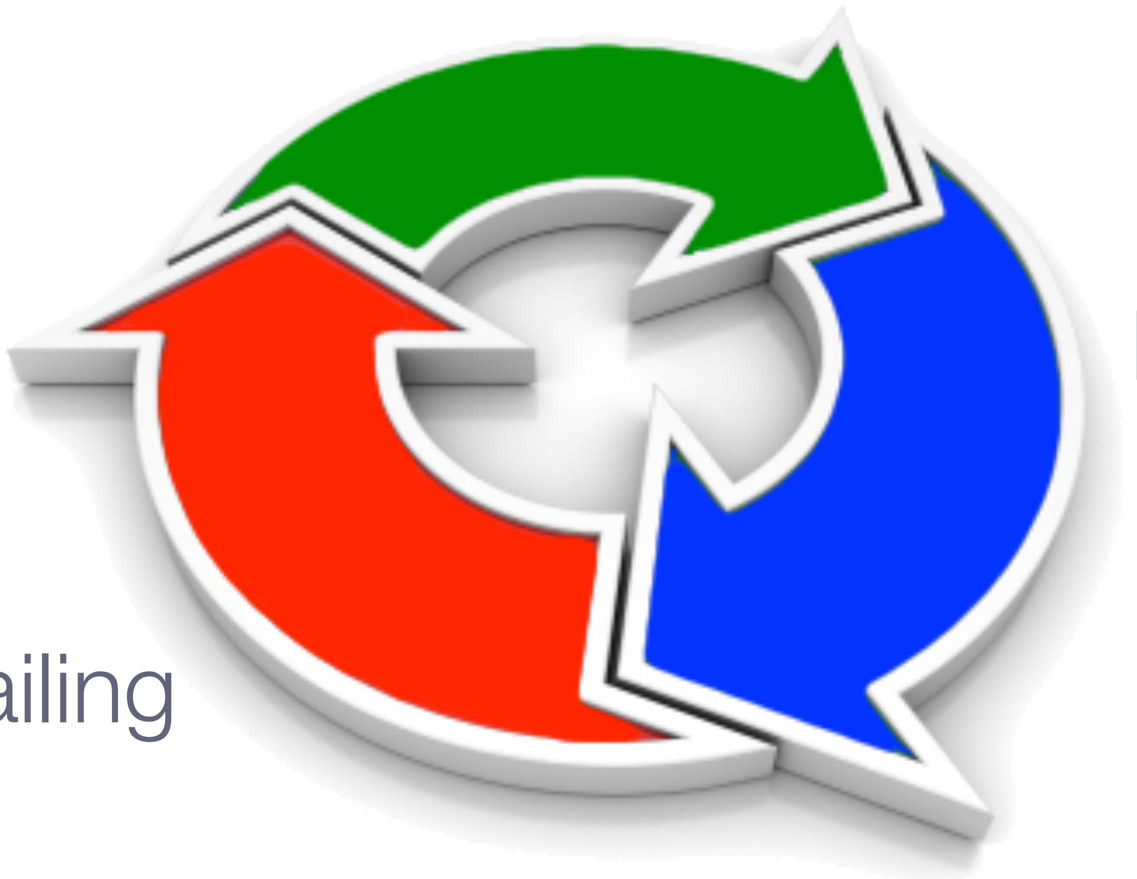
- Use unit tests to drive development and design.
- Write the test first, then the code.
  - See the test fail, then make it pass.
  - Importance of spiking before test-first development.
- Test coverage of your code remains high because of test-first approach.
- A fast test suite is typically run frequently.

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) ✗ 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 TbzZHmxXJQ3aKLGcIIel
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

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

Jasmine suite

```
describe("A specification suite", function() {

    …

});
```

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

Jasmine specification

```
describe("A specification suite", function() {

  it(“contains spec with an expectation", function() {
    expect(view.tagName).toBe(‘tr’);
  });

});
```

- Specifications are expressed with the **it** function.
  - The description should read well in the report.
- Expectations are expressed with the **expect** function.

Jasmine matchers

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

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');
   });
});
```

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;
    }
  });
});
```

Demonstration

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.
- More information at https://github.com/pivotal/jasmine/wiki/Spies.

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(Leaflet, “circle”).andCallThrough();
mapView.processResults(earthquakeJsonResults);
expect(spy).toHaveBeenCalled()
expect(circleConstructorSpy.callCount).toBe(2);
expect(circleConstructorSpy.argsForCall[0][0])
   .toEqual([56.6812, -155.0237])
```

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));
```

Jasmine spy usage

- `andCallThrough()`: Allows the invocation to passthrough to the real subject.
- `andReturn(result)`: Return a hard-coded result.
- `andCallFake(fakeImplFunction)`: Return a dynamically generated result from a function.
- `createSpy(identity)`: Manually create a spy.
- `createSpyObj(identity, propertiesArray)`: Creates a mock with multiple property spies.

Jasmine asynchronous support

- Use `runs` and `waitsFor` blocks and a latch function.
- The latch function polls until it returns true or the timeout expires, whichever comes first.
- If the timeout expires, the specification fails with a message.
- Kind of clunky to use.

Jasmine asynchronous example

```
describe("an async spec", function() {
    var done;

    beforeEach(function() {
        done = false;
        var doStuff = function() {
            // simulate async stuff and wait 10ms
            setTimeout(function() { done = true; }, 10);
        };
        runs(doStuff);
        waitsFor(function() { return done; },
            ‘The doStuff function should be done by now.’,
            100);
    });

    it("did stuff", function() {
        expect(done).toBe(true);
    });
});
```

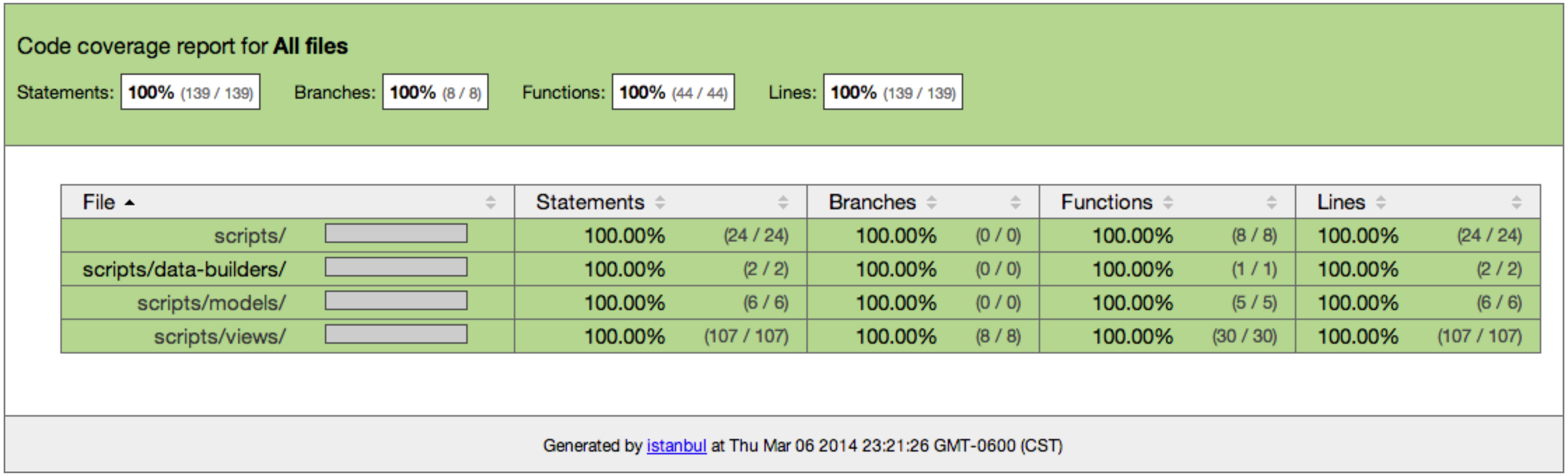
karma-coverage

- 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!
  - Code katas,
- 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.
    - Geb, Capybara, Cucumber{Ruby|jvm|js}, 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://phantomjs.org
- http://karma-runner.github.io/
- http://gruntjs.com/
- http://bower.io/
- http://pivotal.github.io/jasmine/
- http://yeoman.io/

Recommended reading

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

Learning resources

- Let's Code: Test-Driven JavaScript
  - http://www.letscodejavascript.com/
- Egghead.io
  - http://egghead.io/

Code kata resources

- http://katas.softwarecraftsmanship.org/
- http://codekata.pragprog.com/
- http://projecteuler.net/
- http://codekatas.org/

Presentation GitHub repository

- https://github.com/cebartling/ncaa-basketball-tournament
- The **web-client** directory contains this entire sample Backbone.js-based application.

Thank you!

- Christopher Bartling
    - @cbartling
    - chris@pintailconsultingllc.com