Angular

Capital One

developintelligence.com

Last Day Topics

- Animation
- Jasmine Unit Testing
- Karma Automated Test Runner
- End-to-end Testing with Angular Scenario Runner
- End-to-end Testing with Protractor
- Build Tools / Scaffolding
- Angular Architecture

Animation

Animation

- Angular gives our application hooks that can be utilized for animation
- We can interact with these hooks via CSS and JavaScript
  - CSS3 Transitions
  - JavaScript Animations
  - CSS3 Animations

Animation [cont.]

- The ngAnimate module was taken out of the Angular core in 1.2
  - angular-animate.js
  - bower install angular-animate
- Needs to be injected

```
var app = angular.module('demo', ['ngAnimate']);
```

$animate Service

- Automatically supports some of Angular’s built-in directives
- No manual configuration needed on our part to utilize animation support
- We can use the $animate service for our own animations

Built-in Support

- $animate works by listening to events on the directives
  - ngRepeat: enter, leave, move
  - ngView, ngInclude, ngSwitch, ngIf: enter, leave
  - ngClass, ngShow, ngHide : add, remove

Built-in Support [cont.]

- $animate interacts with the directive by adding classes based on the directive events

- All the directives that fire events have them added by $animate
- ngRepeat, ngView, ngInclude, ngSwitch, ngIf
  - enter: .ng-enter (start) … .ng-enter-active (transition)
  - leave: .ng-leave (start) … .ng-leave-active (transition)
  - move: .ng-move (start) … .ng-move-active (transition)

Built-in Support [cont.]

- Structure example

```
/** Defined transition & starting opacity  **/
.structure-animation.ng-enter {
  -webkit-transition: 0.5s linear all;
  transition: 0.5s linear all;
  opacity: 0;
}

/** Transition ending opacity **/
.structure-animation.ng-enter-active {
  opacity:1;
}
```

Built-in Support [cont.]

- Other directives …

- class: class-animation
  - add: .class-animation-add
  - remove: .class-animation-remove
- When dealing with classes Angular will append the -add and -remove for us, but we really don't need them
  - We can simply specify a base class then change it up by adding another class

Built-in Support [cont.]

- Other directives … adding class example
  - This will give us a pulse from white to yellow back to white
  - The transition would take place when we programmatically add the class “class-animation” to an element
  - We could use this transition when we click the “clear transaction” button our our sell page

```
/** Defined transition & color of white  **/
.class-animation-add {
       transition: background-color 0.2s ease;
       background-color: rgb(255,255,255);
}

/** Transition ending color of yellow **/
.class-animation-add-active {
       background-color: rgb(255,255,0);
}
```

Built-in Support [cont.]

- Other directives …

- ngShow, ngHide
  - .ng-hide-add
  - .ng-hide-remove

Built-in Support [cont.]

- Other directives … ngShow / ngHide example

```
/** Defined transition & start with opacity of 0 with ng-hide
class added to our hidden show-animation div **/
.show-animation.ng-hide{
       opacity: 0;
}

/** When we are actively removing mg-hide then transition to 1
**/
.show-animation.ng-hide-remove-active {
       -webkit-transition: 0.5s linear all;
       transition: 0.5s linear all;
       opacity: 1;
}
```

Built-in Support [cont.]

- Upon update of the DOM the directives gain an additional class with a -activate added
  - .ng-enter, gets .ng-enter-active
- The additionally added class triggers our animation
  - $animate uses the appropriate CSS
- After the animation is all done the class are removed
  - i.e. .ng-enter and .ng-enter-active

Animations with JavaScript

- Let's take a look at the code in Lab 12
  - Specifically the animations.js file
  - We can comment out the CSS transition information so that JavaScript will have to handle the transitioning for us

Events

Events

- $broadcast
- Dispatches an event down the child scopes and into their children
- Registered scope listeners will be able to have their callback functions run

```
$rootScope.$broadcast(‘semanticEvent’, arguments);
$rootScope.$on('semanticEvent', function () {
  //Do what is needed
}
```

Events

- $emit
- Dispatches an event up the parent scopes
- Registered scope listeners will be able to have their callback functions run

```
$scope.$broadcast(‘semanticEvent’, arguments);
$scope.$on('semanticEvent', function () {
  //Do what is needed
}
```

Testing

Test: Why?

- JavaScript can be wonky   :)
  - No compiler help   :(
  - We want to be confident about what we have written
- Javascript is heavy-weight
  - Rich Internet applications
  - Server-side JS
- Automate testing
  - Increase efficiency
  - Increase app coverage

Test: What?

- We need to have a testing strategy
- Aimless testing is useless
    - no confidence in the tests
- Test everything
    - more time spent on testing than code

Test: What?

- Unit tests
  - Verify the smallest unit of work in an application
  - Makes a single assumption
  - Runs in isolation - testing different inputs
- Integration tests
  - Group testing of modules
  - Makes sure components work well together

Test: What? [cont.]

- System tests
  - Examine the complete application
  - Includes: load testing, scalability and security
- Functional tests (E2E)
  - Validation of the User Interface - Real life scenarios
  - Testing large swaths of the application
  - Quality assurance testing

Regression

- Make changes to the code base
  - Run tests & make sure you are still all good
- When you find a bug
  - Write a test that will show the bug
  - Fix the code
  - Run the test to make sure the bug is fixed

Good Unit Tests

- Automated & Repeatable
  - Others need to be able to run the tests for code changes
- Easy to understand
  - Others need to be able to understand the test
  - Easy for others to add more test cases
- Incremental
  - Tests should be updated if there is a code defect
- Easy to run & Fast
  - Click a button or execute a command to run them
  - We don’t want tests that will take a long time to run

Jasmine

- Testing framework
  - Defines our test syntax
  - Defines how the tests are written

Jasmine

- Behavior Driven style for writing our tests
  - We don’t write functions, tests and asserts
  - We describe behaviors and set our expectations
- Behavior Driven Development
  - Agile at its core
  - User stories as the foundation for writing tests
  - **Given** a player, when the **game** is paused then **he** should drop the controller
  - **given** - The initial context
  - **when** - Some event occurs
  - **then** - Ensure some outcome

Jasmine [cont.]

- Jasmine translation :)
  - Pretty easy to make the jump from thinking in behaviors to unit testing them
  - Each acceptance criteria is a test unit
  - Each test unit is called a **spec** (i.e. specification)

```
describe(‘Player’, function() {
  describe(‘when game has paused’, function() {
    it(‘should drop the controller’, function() {

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

Suite

- Groups of specifications that relate to the same implementation code
  - Created with describe()
  - We can nest suites for better grouping our specifications
- **describe**(label, function() {…})
  - Groups related tests together
  - Think of them as modules

```
describe('Unit test: DemoController', function() {
  describe('save method', function() {
    // Specs go in here
  });
});
```

Specifications i.e. Tests

- Specifications contain at least one expectation, testing the state of the code in question
- Labels an individual test
- Created with **it**()
  - **it**(label, function() {…})
    - **label**: String of a title/description of the **spec**
    - **function:** One or more expectations

- If the test has all true expectations it is passing
- If the test has one or more expectations that are false it is considered to be failing

Expectations

- Report whether the specification passes
- Created with **expect**(actual)
  - Expectations that evaluate to true or false
  - Used to compare actual values against expected values
  - If the test has all true expectations it is passing
  - If the test has one or more expectations that are false it is considered to be failing

Matchers

- Used to perform comparisons between the passed “expected” and the actual
  - expect(expectedValue).toBe(actualValue)

```
it(‘should expect things’, function() {
  expect(true).toBe(true);
});
```

Setup and Teardown

- Often you will have repetitive code setting up or tearing down your test scenario
- Jasmine provides beforeEach() and afterEach() functions, which get called before and after each spec within a suite
- When suites are nested, all beforeEach()/ afterEach() function is called in the order of nesting

```
it('should expect things', function() {
  expect(true).toBe(true);
});
```

Jasmine Statements [cont.]

- **beforeEach**(function(){…})
  - We don’t have to setup test conditions manually for every test
  - Code that runs prior to each **it** in the **describe**
  - Setup

```
describe(‘A spec suite’ function() {
  var greeting;
  beforeEach(function() { greeting = ‘Hello ’; });
  it(‘should say hello kamren’, function() {
    expect(greeting + ‘Kamren’).toEqual(‘Hello Kamren’);
  });
  it(‘should say hello class’, function() {
    expect(greeting + ‘class’).toEqual(‘Hello Kamren’);
  });
});
```

Jasmine Statements [cont.]

- **afterEach**(function() {…})
  - We don’t have to teardown test conditions manually after every test
  - Code that runs after each **it** in the **describe**
  - Teardown

```
describe(‘A spec suite’ function() {
  var counter;
  afterEach(function() { counter = 0; });
  it(‘should say increment’, function() {
    count = count + 1;
    expect(count).toEqual(1);
  });
  it(‘should say hello class’, function() {
    expect(count).toEqual(0);
  });
});
```

Alternate Variable Sharing

- it, beforeEach and afterEach function all share the same scope
- Instead of variables within the suite scope, properties can be set and accessed on the ‘this’ object within these methods

```
describe(‘Suite’, function() {
  beforeEach(function() {
    this.a = 1;
  });
  afterEach(function() {
    this.a = 0;
  });
});
```

Jasmine Matchers

- **toBe**(null / true / false)
  - ===

```
describe('toBe' function() {
  it('passes on equality', function() {
    var aObject = {a: '1'};
    var bObject = {a: '1');
    var aArray = [1, 2];
    var bArray = [1, 2];
    expect(true).toBe(true);
    expect(aObject).toBe(aObject);
    expect(aObject).not.toBe(bObject);
    expect(aArray).toBe(aArray);
    expect(aArray).not.toBe(bArray);
    expect("Hello World").toBe("Hello World");
  });
})
```

Jasmine Matchers [cont.]

- **toEqual**(value)
  - Used the most
  - Good for comparing simple literals and variables
  - Good for comparing object content

```
describe(‘toEqual’ function() {
  it(‘passes on equality’, function() {
    expect(’42’).toEqual(’42’);
    expect(’42’).not.toEqual(’23’);
    expect([‘a’, ‘b’]).toEqual([‘a’, ‘b’]);
    expect({a: 1, b: 2}).toEqual({a: 1, b:2});
  });
})
```

Jasmine Matchers [cont.]

- **toMatch**(regular expression / string)

```
describe('toMatch' function() {
  it('compares based on regexp', function() {
    expect('Hello World').toMatch(/world/i);
    expect('Hello World').toMatch('Hello');
    expect('Hello World').not.toMatch('goodbye');
  });
})
```

Jasmine Matchers [cont.]

- **toBeDefined**()
  - Checking for existence

```
describe(‘toBeDefined’ function() {
  it(‘passes if subject is not undefined’, function() {
    expect({}).toBeDefined();
    expect(undefined).not.toBeDefined();
  });
})
```

Jasmine Matchers [cont.]

- **toBeUndefined**()
  - Checking for existence

```
describe(‘toBeUndefined’ function() {
  it(‘passes if subject is undefined’, function() {
    expect(undefined).toBeUndefined();
    expect(undefined).toBe(undefined);
    expect({}).not.toBeUndefined();
  });
})
```

Jasmine Matchers [cont.]

- **toBeNull()**

```
describe(‘toBeNull’ function() {
  it(‘passes if subject is null’, function() {
    expect(null).toBeNull();
    expect(null).toBe(null);
    expect({}).not.toBeNull();
  });
})
```

Jasmine Matchers [cont.]

- **toBeTruthy()**

```
describe(‘toBeTruthy’ function() {
  it(‘passes if subject is truthy’, function() {
    expect(true).toBeTruthy();
    expect(1).toBeTruthy();
    expect(‘Hello World’).toBeTruthy();
    expect([]).toBeTruthy();
    expect({}).toBeTruthy();
    expect(false).not.toBeTruthy();
  });
})
```

Jasmine Matchers [cont.]

- **toBeFalsy()**

```
describe(‘toBeFalsy’ function() {
  it(‘passes if subject is falsy’, function() {
    expect(false).toBeFalsy();
    expect(NaN).toBeFalsy();
    expect(null).toBeFalsy();
    expect(undefined).toBeFalsy();
    expect(false).not.toBeFalsy();
  });
})
```

Jasmine Matchers [cont.]

- **toContain**(string)

```
describe(‘toContain’ function() {
  it(‘passes if expected is contained in actual array’,
function() {
    expect([4, 2]).toContain(2);
    expect(“Hello World”).toContain(“World”);
    expect([4, 2]).not.toContain(1);
  });
})
```

Jasmine Matchers [cont.]

- **toBeLessThan**(number)

```
describe('toBeLessThan' function() {
  it('passes if actual is less than expected', function() {
    expect(5).toBeLessThan(2);
    expect(5).not.toBeLessThan(7);
  });
})
```

Jasmine Matchers [cont.]

- **toBeGreaterThan**(number)

```
describe(‘toBeGreaterThan’ function() {
  it(‘passes if actual is greater than expected’, function() {
    expect(2).toBeGreaterThan(5);
    expect(2).not.toBeGreaterThan(1);
  });
})
```

Jasmine Matchers [cont.]

- **toBeCloseTo**(number, precision)

```
describe(‘toBeGreaterThan’ function() {
  it(‘passes if precision is met’, function() {
    expect(3.1415).toBeCloseTo(3.14);
    expect(3.1415).not.toBeCloseTo(3.15);
    expect(3.14159).toBeCloseTo(3.14160, 2);
  });
})
```

Jasmine Matchers [cont.]

- **toThrow()**

```
describe(‘toBeGreaterThan’ function() {
  it(‘passes if actual is greater than expected’, function() {
    var object = {
      doSomething: function() {
        throw new Error("Unexpected error!")
      }
    };
    expect(object.doSomething).toThrow(new Error("Unexpected
      error!"));
  });
})
```

Cutom Matchers [cont.]

- While fairly robust, sometimes the provided matchers don’t provide everything we need
- Custom matchers can be registered with Jasmine for use in specs
- A custom matcher is nothing more than an object that contains a ‘compare’ function, which is passed the ‘actual’, and returns an object with a ‘pass’ and ‘message’ properties
- Custom matchers must be registered from within it() or beforeEach() functions

Cutom Matchers [cont.]

- Example

```
jasmine.addMatchers({
  toBeFunction: function() {
    return {
      compare: function(actual) {
        var pass = typeof actual === “function”;
        var not = pass ? ‘’ : “ not”;
        return {
          pass: pass,
          message: actual + “ is” + not + “ a function”
        };
      }
    }
  }
});
```

Jasmine Matchers [cont.]

- Matching anything
- Jasmine can match anything via jasmine.any() function, which accepts a Constructor

```
expect(42).toEqual(jasmine.any(Number));
```

Angular Interaction

Angular Interaction

- To interact with our angular code base we need some some help outside of Jasmine
- Google has given some nice tools
  - ngMock
    - A library created for mocking
    - Defines simulated objects that act as real objects
  - https://docs.angularjs.org/api/ngMock
  - We need to include angular-mocks.js in our test runner

```
bower install angular-mocks#~1.3
```

Module Setup

- First we need to setup our mock angular module
- angular.mock.module()
    - Sets up our Angular mock module
    - Takes a string of the module to mock
    - This can be used during setup of a suite

```
describe('Suite' function() {
  beforeEach(angular.mock.module('aModule'));
});
```

```
describe('Suite' function() {
  beforeEach(module('aModule'));
});
```

Injecting Dependencies

- Second we inject our dependencies
- angular.mock.inject()
  - During testing we are in charge of injecting dependencies
  - This is what angular does at run time in our app normally
  - We specify the test functionality we want to verify

```
beforeEach(angular.mock.inject(function($controller) {
  //Use the $controller
}));
```

```
beforeEach(inject(function($controller) {
  //Use the $controller
}));
```

Injecting Dependencies [cont.]

- We will want to set up injections in our beforeEach for our test suite
  - This will allow us to have access to them across all tests in the suite
- We will use an _dependency_ name format
  - Makes sure the injector ignores the name when injecting

```
describe('Suite' function() {
  var createdService;

  beforeEach(module('aModule'));
  beforeEach(inject(function(_createdService_) {
    createdService = _createdService_;
  });
});
```

Injecting Dependencies [cont.]

- We have seen explicit injection
- We can also use implicit injection

```
describe(‘Suite’ function() {
  var CreatedService;

  beforeEach(module(‘aModule’));
  beforeEach(inject(function($injector) {
    CreatedService = $injector.get(‘CreatedService’);
  });
});
```

$httpBackend Service

- We want our Unit Tests to run fast with no external dependencies
  - Say no thanks to Ajax
  - Say yes to $httpBackend
- $httpBackend will not be used in normal development
  - Usually we will use higher level $http or $resource
- In testing we mock $httpBackend, a fake backend
  - Used to verify the requests and responses
  - No server needed
  - https://docs.angularjs.org/api/ngMock/service/$httpBackend

Request Expectation

- $httpBackend.expect
  - Used to make assertions about the application requests
  - Defines responses for the application requests
  - Sets up the expected HTTP method and URL we anticipate sending via our application

```
$httpBackend.expect(‘GET’, ‘some/url’, [data], [headers]);
```

```
$httpBackend.expect('GET', ‘/data/supplies.json');
```

Request Expectation [cont.]

- Helper methods
  - $http.expectGET(url, [headers]) //Retrieve
  - $http.expectDELETE(url, [headers]) //Delete
  - $http.expectPOST(url, [data], [headers]) //Update
  - $http.expectPUT(url, [data], [headers]) //Create
  - $http.expectHEAD(url, [headers]);

Request Expectation [cont.]

- Our $httpBackend.expect() creates an expectation
- It returns to us a requestHandler object containing a respond() method
  - Allows us to specify how to handle the matched request
  - We could setup response codes, data or headers

```
.respond([status,] data[, headers, statusText]);
```

```
$httpBackend.expect('GET', '/data/supplies.json')
  .respond(200, {
    initial: {
      lemonadeQuantity: 4,
      healthySnackQuantity: 2,
      treatQuantity : 2
    }
  });
```

Backend Definition

- $httpBackend.when
- Creates a backend the doesn’t assert if the request was made
- Doesn’t setup expectations for interacting with the backend
  - Used for more loose unit testing
  - Good for setting up commonality across all tests

Backend Definition [cont.]

- With $httpBackend.when we can make as many calls as we might need against the backend definition
  - .expect will only allow us to make 1 call
- Same syntax as the $httpBackend.expect

Flush

- httpBackend.flush()
- Allows us to explicitly flush pending tests
  - Preserves our asynchronous interaction but allows us to execute test code in a synchronous manner

Clean-up

- $httpBackend.verifyNoOutstandingExpectation()
  - Makes sure all of the requests in our $httpBackend expect were made
  - Throws an exception if they weren't
- $httpBackend.verifyNoOutstandingRequest()
  - Makes sure there aren't any requests that need to be flushed

- Good to put these in our afterEach testing teardown

Application Unit Testing Continued

Testing Routes

- We need to make sure our single page application traverses the correct paths
  - Are we going where we want and loading what we need?
  - Are we generating a 404 … file not found?
- Route testing will involve $state (use $route if using ng-route), $location and $rootScope
- We will also need a mock $httpBackend for template fetching
  - We can fake like we had a successful retrieval of our template

Testing Routes [cont.]

- After moving location we will need to interact with the $scope lifecycle
  - $rootScope.$digest() primarily used in unit testing
  - Simulates the scope life cycle
  - $watch is called on every $digest loop
  - The $digest loop re-runs on every detection of a dirty value
- Usually you won't call the $digest in production code
  - scope.$apply() will force a $digest() loop

Testing Filters

- Filters are built upon isolated functionality
- They limit or manipulate output
- To test filters we inject the $filter service into the tests
- We write expectations off the filter output

```
expect(filter('someFilter')('abc')).toEqual('AbC');
```

Testing Directives

- We want to test that the rendering is what we expect
  - This means we need to focus on the bindings
- We will create an element that will hold the directive
  - The created element looks like what we would normally place in the HTML

Testing Directives [cont.]

- We will also need to $compile the element
  - It will take our string of HTML and produce a template for us based off of whatever scope we give it

```
$compile(createdElement)(scope);
```

  - After compilation we run the digest loop
  - Puts our element into a fake DOM

Testing Directives [cont.]

- Lastly we need bind our properties to the scope
  - This is done in the $apply to force our digest loop to run
  - We use $apply usually because it will allow us to evaluate a function given to it

Testing Templates

- Unit testing templates doesn’t work so well
  - Templates are tied directly to views
  - We would need to make expectations on the completely rendered view

Testing Templates [cont.]

- We can test the router configuration using $route.routes or $state.get(‘aState’)

```
expect($route.routes[‘/‘].controller).toBe(‘HomeController’);
expect($route.routes[‘/‘].templateUrl)
  .toEqual(‘templates/home.html’);
```

```
expect($state.get(‘home’).controller).toBe(‘HomeController’);
expect($state.get(‘home’).templateUrl)
  .toEqual(‘templates/home.html’);
```

Testing Templates [cont.]

- We can test that the template did load correctly
  - Test the routes via $location.path() changes
  - Setup an expect for the template URL from $httpBackend
  - Allow us to be notified when a template's html file is loaded

Jasmine Lab

- Use Jasmine for testing
  - **bower install jasmine**
  - Don’t put it in your dependencies
  - Unzip the latest standalone **zip** inside the **dist** folder to find **SpecRunner.html**
    - Copy that file and put it in your **test** folder which is on the same level as your **src** folder
    - Notice where the Player.js and Song.js application files are and where the corresponding spec PlayerSpec.js is located
    - Remove any references to Player and Song
    - Replace with your source and tests
    - Link to appropriate Jasmine files in bower repository
      - ../bower_components/jasmine/lib/jasmine-core/…

Multiple Browser Tests

Karma

- Test runner used to ease test automation
  - Manages test environments
  - Staying on top of testing is simpler
  - Finds all the tests in our code
  - Supports unit and E2E testing
  - http://karma-runner.github.io/0.12/index.html

Karma [cont.]

- Make a single browser instance or multiple browser instances
  - Manages target browsers
  - Runs our tests against the browsers environments on our machine
  - Uses Web Sockets (socket.io) to interact with the browser
  - Great realtime solution for tests assertions
- Captures test results
  - Manages test reporting

npm init

- Before getting started with Karma we need to prime our working directory
- **npm init** will setup your **package.json** file
  - A little script that asks you some questions
  - A **package.json** file is created so others will only have to do an **npm install** to get all the dependencies this project needs up and running

```
npm init
```

npm init [cont.]

- Questions to answer
  - **name**: By default it will take the folder name
  - **version**: Start it off at 0.0.0
  - **description**: What your module will be about
  - **entry point**: The file used for this module once it is all done
  - **test command**: A command for running unit tests
  - **git repository**: Location
  - **keywords**: Keywords about what this package does
  - **license**: ISC - Simplified MIT

Karma Install

- Install Karma CLI
  - Lets us run Karma easier

```
npm install -g karma-cli
```

- Install Karma locally
  - Install in application folder
  - Will install in devDependencies folder

```
npm install karma --save-dev
```

Karma Install [cont.]

- Install Karma plugins
  - Pick the components we need for the project
  - Install these in our same application folder
  - Need to be installed for every project

Karma Plugins

- Install testing framework plugins:
  - Jasmine, Mocha, QUnit …

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

  - After installation these will appear in the **package.json** file under **devDependencies**

Karma Plugins [cont.]

- Install browser launchers:
  - Chrome, Firefox, IE, Safari …

```
npm install karma-chrome-launcher --save-dev
```

```
npm install karma-firefox-launcher --save-dev
```

```
npm install karma-phantomjs-launcher --save-dev
```

  - After installation these will appear in the **package.json** file under **devDependencies**

Karma Configuration

- Once we have Karma installed we need to configure it
  - Karma needs to know where our testing files and application files are located
- **Karma** gives us an easy installation script just like **npm** for our **package.json** file
  - We will use the script to create a **karma.config.js** file
  - This file is what karma looks for to get the testing environment up and running

```
karma init
```

Karma Config Questions

- Which testing framework? **Jasmine**
- Do you want to use Require.js? **No**
- Do you want to capture any browsers automatically? **Chrome** (hit enter twice)
- What is the location of your source and test files?
  - Set relative to the path relative to the current folder
  - First load **angular.js**
  - Second load **angular-mocks.js** & angular modules
  - Third load application source code
  - Last load the source code for actual unit tests

Karma Config Questions [cont.]

- Should any of the files included by the previous patterns be excluded?
  - Leave this empty as long as you were specific about the source and test files
- Do you want Karma to watch all the files and run the tests on change? **yes**

- Now you have a **karma.conf.js** file setup for unit testing

karma.conf.js Extras

- Other parameters could be assigned manually
  - **port**: Port to have Karma test runner server run on
  - **logLevel**: Specify the level of log to capture from the browser (e.g. console.log, console.info, console.debug …)
  - **singleRun**: Tells Karma to shutdown the server after a single run of the unit tests

Start Karma

- After installation and configuration it is time to start Karma up
  - Karma is up and running
  - Chrome is out there
  - We can see our results in the terminal

```
karma start karma.conf.js
```

```
Chrome 36.0.1985 (Mac OS X 10.9.4): Executed 1 of 1
SUCCESS (0 secs / 0.029 secs
```

Karma

- Great news!
  - Karma has a built in watch
  - Every time you save your JavaScript Karma will run the tests again

Application Unit Testing Continued

Testing Services

- Services usually have some sort of asynchronous activity with them
  - Could be an $http call
  - Could be a $resource

Testing Services [cont.]

- These services return promises that we have to negotiate

- If a service is returning an **$http** then we can utilize our $httpBackend
  - We will get a promise returned and then we need to negotiate the mocked data within the promises then-able architecture

Testing Services [cont.]

- These services return promises that we have to negotiate
- If a service is returning a **$resource** then we will again utilize our $httpBackend
  - We will get a deferred that we need to grab the $promise off of our deferred object
  - Then we need to negotiate the mocked data within the promises then-able architecture

Testing Services [cont.]

- To test the returned promise you will need to flush out the $httpBackend
  - $httpBackend.flush()
- That will allow you to be brought into the resolution of the promise where your expectations should be run

Testing Controllers

- We need to make sure the controller logic is functioning as expected
- We will need our controllers to have an instance of a known scope
  - This is achieved via creating a new child scope
  - $rootScope.$new()

Testing Controllers

- We will need to test against that scope
  - We will dynamically inject the $controller service and get a controller based on the newly created child scope

```
var actual = {
  lemonadeQuantity: 0,
  healthySnackQuantity: 0,
  treatQuantity: 0,
};
scope = $rootScope.$new();
$controller('SuppliesController', { $scope: scope });
expect(scope.actual).toEqual(actual);
```

Application Testing e2e

e2e Basics

- We care about content being rendered in the correct manner
- We will be setting expectations on the rendered HTML

Protractor

Protractor

- AngularJS Scenario Runner didn’t cut it :)
  - It tried to make tests more stable and deterministic
  - Simulating clicks and typing through JavaScript wasn’t good
  - Not a true user flow
  - Angular’s new e2e tool

Protractor [cont.]

- Protractor builds on Selenium WebDriver
  - Works at the OS level
  - Performs actual clicks and keystrokes
- Adds specific AngularJS functionality to WebDriver

Protractor [cont.]

- Protractor aims to minimize the waiting of Ajax applications
  - Single page applications only load one page
  - Data is fetched asynchronously after the page is loaded
  - We don’t have to wait for arbitrary time or events
  - When a button is clicked, a server call is made, Protractor waits for the server call to return before going on with more tests
  - Phew! We can just focus on the test writing. No conditions or timeouts for data loading or elements to fade

Protractor Installation

- Install protractor via npm
  - It installs the protractor cli and the webdriver-manager cli
  - WebDriver manager is a tool to help with getting Selenium up and running
  - Allows us easily to interact with WebDriver

```
npm install -g protractor
```

Selenium Interaction [cont.]

- Update webdriver-manager

```
webdriver-manager update
```

- Start the Selenium Server

```
webdriver-manager start
```

- Selenium download documentation
  - http://docs.seleniumhq.org/download/

Protractor

- Just like everything else we have seen so far we need to create a configuration file for protractor
  - protractor-conf.js
  - Tell it where the tests are and what page we will serving the application from

```
exports.config = {
  specs: [
       './e2e/**/*.spec.js'
  ],
  baseUrl: 'http://localhost:8080',
  capabilities: {
    'browserName': 'chrome'
  }
};
```

Protractor

- To run protractor
  - protractor protractor-conf.js
- To run with specific JS tests
  - protractor protractor-conf.js —specs='e2e/give.spec.js'

- You need to have your node server up and running for protractor to run properly

Protractor API

- Uses the same Jasmine syntax as unit testing
  - **describe** is for sets of tests
  - **it** if for each test
- Full API
  - https://github.com/angular/protractor/blob/master/docs/api.md
- Protractor doesn’t use Jasmine 2.0 syntax yet
  - Won’t work: .not().toEqual()
  - Will work: .toNotEqual()

Protractor API [cont.]

- **browser**:
  - Used for browser navigation to different pages
  - Wrapped around WebDriver so we have direct browser interaction
  - browser.get(‘url’)
  - browser.findElement(locator)
  - browser.isElementPresent(element)
  - browser.baseUrl
  - browser.sleep(time)
  - browser.setLocation(‘url’)
    - Based on the $location service

Protractor API [cont.]

- **element**:
  - Object used for finding and interacting with HTML elements
  - Takes a selector to find an element and returns the element back
  - element(locator) / $(cssSelector)
  - element.all(locator) / $$(cssSelector)
    - useful for intreating with ngRepeat
  - element.getText()
  - element.getInnerHtml()

Protractor API [cont.]

- **by**:
  - Object that has a bunch of element finding strategies/ selectors
  - WebDriver has built in the ability to find things by **id** or **CSS classes**
  - by.id('content')
  - by.css('.tab')
  - by.binding('person.email')
  - by.model('person.name')
  - by.repeater('item in items')

Protractor API [cont.]

- **by**:
  - Object that has a bunch of element finding strategies/ selectors
  - WebDriver has built in the ability to find things by **id** or **CSS classes**
  - Protractor added functionality to find AngularJS things by **model**, **binding** and **repeater**

Debugging

- We can’t use debugger; in protractor either
- We “should” be able to pause the interaction
  - browser.pause()
  - Will stop the browser in its tracks
  - Pause is still experimental

Protractor API

- Protractor gives us easy ways to interact with the DOM
- **.click()**
  - Actually clicks the element
- **.clear()**
  - Allows us to clear an input field
- **.sendKeys()**
  - Allows us to send keystrokes to the browser

- Need more:
  - https://github.com/angular/protractor/blob/master/docs/api.md

Protractor API

- Protractor gives us easy ways to interact with the DOM
- element.all(by.repeater(‘item in items’)).then()
  - Repeaters become easy to interact with
- .count()
  - The amount of indexes in a repeater

Testing Routes

- We need to think about how the user will transition through our application
- Write readable tests about what the user will experience

Testing Requests

- Protractor is all promised based
  - Can make it a bit tricky to debug
  - Everything is wrapped in a promise
  - So don’t forget your .then powers! :)
- There is very little need to worry about the asynchronous nature or your AngularJS
  - A browser.get() will make sure all of the our $http calls are resolved before moving into the testing code
  - Bindings based on async calls will be resolved before Jasmine comparisons take place

Testing Application

- Due to protractor’s use of promises we can succinctly test our application logic without having to tie into Angular’s mocks
- This allows us to transition from views, interact with the results of multiple services, and get see different controllers

Building

Grunt

- http://gruntjs.com/
- A task runner
- Like Ant, Make or Rake
- One of the JavaScript build tool solutions
- Built on top of node.js
- You can set up a grunt file that gives you a single, unified set of commands for every JavaScript project you run
- Uses http://livereload.com/

```
npm install -g grunt
```

Grunt

- Efficient Builds:
  - It runs builds we don’t have to think about it
- Consistent Builds:
  - It runs builds we don’t have to think about it
- Programmers more Effective:
  - It runs builds we don’t have to think about it
- Good community support
  - We can contribute to the Grunt effort :)

Grunt Packages

- Use npm to install any grunt packages we need
  - Make sure to install them locally to the project
    - --save : Packages appear as **dependencies**
    - --save-dev : Packages are saved as **devDependencies**
  - These will modify your package.json file
    - Create a package.json file with **npm init**

JSHint

- What Grunt tasks
  - JSHint - Check out those possible JS bugs
  - https://github.com/gruntjs/grunt-contrib-jshint
  - Load module:
    - grunt.loadNpmTasks(‘grunt-contrib-jshint’);
  - jshint-stylish node package for styling output
    - https://github.com/sindresorhus/jshint-stylish

```
jshint: {
  options: {
    jshintrc: '.jshintrc',
    reporter: require('jshint-stylish')
  },
  target1: ['Gruntfile.js', 'src/**/*.js']
}
```

Transcompilation

- Grunt will allow us to take LESS/Sass files and transpile that code to the CSS we need

- LESS/Sass will help you write your stylesheets easier
  - Include variables and looping
  - Built in common functions
  - Mixins

- **less** task … grunt-contrib-less
- **sass** task … grunt-contrib-sass

Copy

- Copies files from one location to another
  - So you don’t screw stuff up :)
  - https://github.com/gruntjs/grunt-contrib-copy
  - Load module:
    - grunt.loadNpmTasks('grunt-contrib-copy');

```
copy: {
  main: {
    files: [
      {expand: true, src: ['src/js/**'],
         dest: 'annotate/'}
    ]
  }
}
```

Concatenation

- The **concat** task allows us to put all our JavaScript files into a single source file
  - grunt-contrib-concat
  - https://github.com/gruntjs/grunt-contrib-concat
- Not only is it good to have small JavaScript files it is great to have less JavaScript files
  - Less files === less calls to the backend

Concatenation [cont.]

- Load module
  - grunt.loadNpmTasks(‘grunt-contrib-concat');

```
concat: {
  target1: {
    files: {
      'build/js/main.js': [
        'annotate/src/js/main.js'
       ]
    }
  }
}
```

Minification

- The **uglify** task will allow us to minify our JavaScript
  - grunt-contrib-uglify
  - https://github.com/gruntjs/grunt-contrib-uglify

- The smaller our JavaScript file the faster our application will load

```
uglify: {
  app: {
    src: 'build/js/main.js',
    dest: 'build/js/main.min.js'
  }
},
```

Clean

- Clean up after yourself
  - https://github.com/gruntjs/grunt-contrib-clean
- Load module:
  - grunt.loadNpmTasks('grunt-contrib-clean');

```
clean: {
  src: ['annotate/**']
}
```

Wiredep

- Will wire up Bower dependencies with your application
  - https://github.com/stephenplusplus/grunt-wiredep
  - Automatically inputting the needed js links in your HTML
- Load module
  - grunt.loadNpmTasks('grunt-wiredep');
- bower.json
  - You will need to make sure you have a bower.json file that will be loaded as you pull in bower packages
  - **bower init**
    - This command will help you create a bower file
  - Put this file in the same folder as the Gruntfile.js

Wiredep

- Configuration

```
wiredep: {
  app: {
    src: 'src/index.html'
  }
}
```

```
<body>
  <!-- All your code -->
  <!-- bower:js -->
  <!-- endbower –>
  <script src="js/main.js"></script>
</body>
```

ngAnnotate

- Dependency Injection is really important in Angular
  - Grunt can help us after writing our code to make sure we have all the necessary pieces dependency injected
  - Gets you ready to minify your code
  - https://www.npmjs.org/package/grunt-ng-annotate
- ngAnnotate has replaced ngMin for Angular dependency injection management
- Load Module:
  - grunt.loadNpmTasks(‘grunt-ng-annotate');

ngAnnotate

- Configuration
  - If we set singleQuotes to true the code replacement will be done with single quotes
    - Useful for hinting

```
ngAnnotate: {
  options: {
    singleQuotes: true
  },
  app: {
    files: {
      'annotate/src/js/main.js': ['annotate/src/js/main.js']
    }
  }
}
```

Watch

- We can have changes in our development files automatically cause things to happen
  - https://github.com/gruntjs/grunt-contrib-watch
- Load module:
  - grunt.loadNpmTasks('grunt-contrib-watch');
- Live reload plugin
  - Allows our browser to be auto-updated when we have a file change
  - Browser plugin needed along with the grunt setup
    - http://feedback.livereload.com/knowledgebase/articles/86242-how-do-i-install-and-use-the-browser-extensions

Watch

- Configuration

```
watch: {
  scripts: {
    files: [
      'src/index.html',
      'src/js/**',
      'build/**',
      'annotate/temp/**'
    ],
    options: {
      livereload: true
    }
  }
}
```

Angular Templates

- Concatenate and cache angular templates used in templateUrl and ng-include
  - https://www.npmjs.org/package/grunt-angular-templates
  - Takes the template html files and puts them into one JavaScript file
  - That file can be minified for even better results
- Load module
  - grunt.loadNpmTasks('grunt-angular-templates');

Angular Templates

- Configuration

```
ngtemplates:  {
  lemonadeApp:        {
    //cwd: Folder root where the templates are usually
    //  served up from
    cwd:      'src',
    src:      'templates/**.html',
    dest:     'build/js/templates.js'
  }
},
```

Grunt

- We don't want to specify what packages we have to load in our Gruntfile.js
  - That is silly
  - https://github.com/sindresorhus/load-grunt-tasks

- Load all our Grunt plugins
  - require('load-grunt-tasks')(grunt);
  - require('load-grunt-tasks')(grunt, {pattern: 'grunt-*'});
  - Both the above statements are equivalent

Grunt

- Execute Grunt via command line
  - **grunt**
- After executing Grunt looks for our **Gruntfile.js**
  - It is the entry point into our build process
- To watch your app with Grunt
  - **grunt watch**

package.json & bower.json

- To install all node packages specifided in the package.json
  - **npm install**
  - Install files into **node_modules** folder
- To install all bower packages specified in the bower.json
  - **bower install**
  - Install files into **bower_components** folder

- These commands will download all the specified packages

Lab 13

- Let’s go back and add Grunt to Lab 1, Lab 2, and Lab 3 and Lab 11
  - Lab 1 is for simplicity
  - Lab 2 and Lab 3 are to see the angular templates work across different routers (i.e. ngRoute and ui-router)
  - Lab 11 is to setup grunt for a more complex application

Gulp

- http://gulpjs.com/
- Another build
- The new kid on the block
- Does the same stuff as Grunt
- Streaming focused: Gives more control over flow
- Follows the CommonJS spec
- No watch plugin needed it is built at the core of Gulp
- Every plugin has a single action/responsibility

```
npm install -g gulp
```

Broccoli

- https://github.com/broccolijs/broccoli
- Another build
- The newest kid on the block
- Does the same stuff
- Similar to Gulp
  - Doesn't use pipes like gulp
  - Applies filters to a tree of files
- Produces the most concise code
- It has a very young code base and not as big of following as Gulp

```
npm install -g gulp
```

Yeoman

- http://yeoman.io/
- Scaffolds projects for you
- Accomplishes its goals through generators
- Installing yeoman automatically installs Bower and Grunt for you!

```
npm install -g yo
```

- Install a generator

```
npm install -g generator-angular
```

- Scaffold Angular: In the directory you want

```
yo angular
```

Angular Seed

- https://github.com/angular/angular-seed
- Seed project for Angular applications
- Gives an opinionated start to Angular application development

ngbp: Formerly ng-boilerplate

- https://github.com/ngbp/ngbp
- A kickstarter for Angular
- It is a build management tool

Architecture

Architecture

- Move to modules
  - Think about what buckets/sections you could group code into
  - Controller hierarchy could be a helpful clue
- Buckets don't need to just include .js files
  - Put your html / css / js that is related together
- A good bucket could be common code
  - Filters / directives
- Create an Angular module for these buckets
  - These buckets will dependency injected into your main module
  - Can't remember the path?? use a **constant** service to store the path to you partials

End