Angular

Capital One

developintelligence.com

Topics for Today

- Web Development Overview
- JavaScript Necessities
- Angular Foundation
- Model-View-Controller & Model-View-Whatever
- Data-binding
- Scope, Modules, Controllers
- Intro to Routing with ngRoute
- Dependency Injection
- Browser Services
- Bower

Second Day Topics

- Upgraded Router
- Filters
- Looping
- Forms
- Error Messaging
- Directives
- Angular @ Startup
- Digest Loop
- Custom Directives

Third Day Topics

- Controllers vs .Services
- Services: Factory & Service
- Services: Constant & Value
- Services: Provider
- Ajax with $http
- Ajax with $resource
- Ajax with Restangular
- Promises & $q
- Thinning out Controllers

Last Day Topics

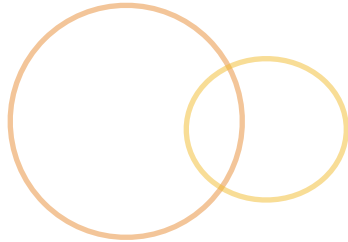
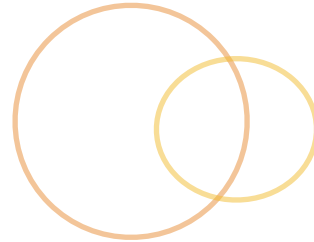
- Animation
- Jasmine Unit Testing
- Karma Automated Test Runner
- End-to-end Testing with Angular Scenario Runner
- End-to-end with Protractor
- Angular Architecture
- Build Tools
- Scaffolding

Web Overview

Separation of Concerns

- Design our applications using distinct sections in mind
  - Sections that will control the organization and structure
  - Sections that will control the presentation
  - Sections that will control the logic and behavior
- Separate Documents
  - Easier to maintain
  - Smaller document size

HTML


- Hypertext Markup Language
- Section that will control the organization
  - Tags for content
  - Semantic layout
  - Document Object Model (DOM)
- When building an application
  - Start with thinking through your content
  - Then begin building the structure

HTML Structure

- Simple HTML boilerplate

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>A Cool Title</title>
    <link rel="stylesheet" href="css/yourCSS.css">
  </head>
  <body>
    <!-- Our content here --!>
    <section id="content">
      <p class="title">Hello</p>
      <p class="title">World!</p>
    </section>
    <script src="js/yourJS.js"></script>
  </body>
</html>
```

CSS

- Cascading Style Sheets
- Sections that will control the presentation
  - Brings style to the content
  - Describes the HTML elements
- When building an application layer on the presentation

CSS Structure

- Cascading Style Sheets
  - Selector: Reference of element to style
  - Declaration: CSS property / value pair

```
/* CSS rules for the HTML elements */
body {
  font-family: “Verdana”;
}

#content {
  padding: 1em;
}

p .title {
  background-color: rgba(0,0,0,0.3);
}
```

JavaScript

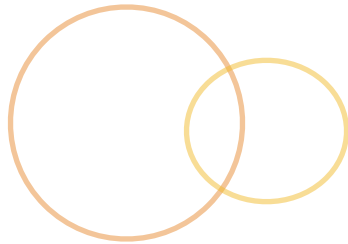
- Sections that will control the application logic and behavior
  - Can handle page routing
  - Handle user interaction
  - Form submission
  - Data crunching
- When building an application, layer on the behavior and control

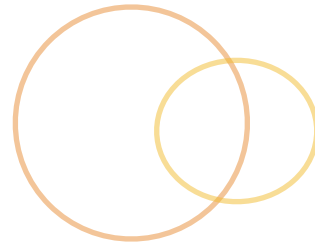
JavaScript Structure

- Sections that will control the application logic and behavior

```
/**
 * Function welcoming everyone.
 **/
var welcome = function(who) {
  console.log("Hello " + who);
};

//Call welcome function
welcome("everyone");
```

JavaScript Necessities





JavaScript Necessities

- Debugging
  - Browser debuggers
  - JSHint
  - Sublime: JSHint Gutter
- Functions
  - Scope: Function level
  - Hoisting
  - Immediately Invoked Function Expressions
- Modules
  - Module Pattern
  - Module Revealing Pattern

Debugging Tools

- Chrome developer tools
  - https://developer.chrome.com/devtools/index
- Firebug
  - http://getfirebug.com/
  - https://addons.mozilla.org/en-US/firefox/addon/firebug/
- Safari
  - https://developer.apple.com/library/safari/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/Introduction/Introduction.html

JSHint

- A powerful tool that will help you catch simple JavaScript programming errors
  - Similar to FindBugs
  - http://www.jshint.com/about/
- .jshintrc file
  - Modify to make yourself a happy coder :)
  - http://www.jshint.com/docs/options/

Sublime Text: Package Control

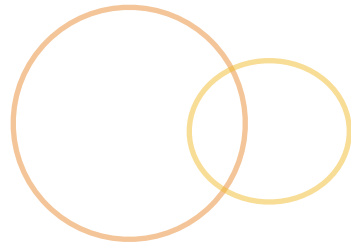
- The package manager for Sublime Text
  - Similar to FindBugs
  - https://sublime.wbond.net
- JSHint Gutter
  - https://sublime.wbond.net/search/jshint%20gutter
  - Nice plugin for JSHint
  - Allows for simple “hinting on save”
  - Easy access to the .jshintrc file

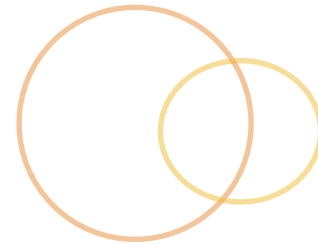
Function Scope

- **var** sets a variable in a local scope (i.e. the function doIt())

```
function doIt () {
  var y = 24;
  console.log("y inside: " + y);
}

doIt();
console.log("y outside: " + y);
```

Hoisting





- All code in JavaScript is **not** interpreted line by line in one pass (i.e. in a top down manner)
  - Remember any variable declared is attached to its scope

File 1

```
//What will print
aNumber = 42;
var aNumber;
console.log(aNumber);
```

File 2

```
//What will print
console.log(aNumber);
var aNumber = 42;
```

File 3

```
//What will print
console.log(aNumber);
```

Hoisting [cont.]

- All declarations are processed first
  - Includes all functions and variables
  - Then code is executed
- 2 phases of JavaScript processing
  - "compilation" phase processes declaration
  - "execution" phase process assignment

```
//How many statements
var aNumber = 42;
```

Hoisting [cont.]

- Functions Expression (i.e. assignment)

```
console.log("Speak: " + yelp());
var yelp = function () {
    return "yelp";
};
```

- Function Declaration: Hoisted

```
console.log("Speak: " + bark());
function bark() {
    return "bark";
};
```

Immediate Invoked

- Functions can immediate call themselves if need be
  - IIFE: Immediate Invoked Function Expressions
  - The function is wrapped in parenthesis because the parser then knows it is a function **expression** not **declaration**

```
//Douglas Crockford's suggestion
(function () {
  console.log("yelp");
}());

//This works fine also
(function () {
  console.log("yelp");
})();
```

```
//Syntax error: a declaration
function () {
  console.log("yelp");
}();

//You meant: an expression
var aFunction = function () {
  console.log("yelp");
}();
```

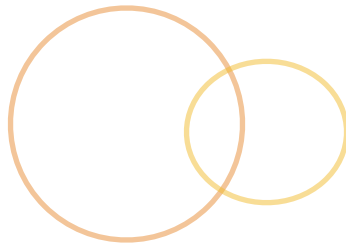
Module Pattern [cont.]

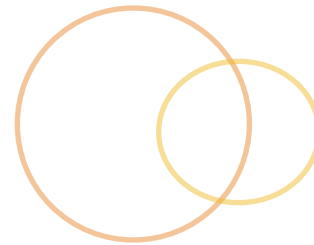
```
var aModule = (function() {
  var time = 8;

  return {
    doWork: function() {
      console.log("working " + time + " hours");
    },
    doMoreWork: function() {
       time += 2;
      console.log("working more " + time + " hours");
    }
  };
}());

console.log("time: " + aModule.time);
aModule.doWork();
aModule.doMoreWork();
```

Revealing Module

```
var aModule = (function () {
  var time = 8;
  function doWork() {
    console.log("working " + time + " hours");
  };
  function doMoreWork() {
      time += 2;
      console.log("working more " + time + " hours");
  };

  return {
    doWork: doWork,
    doMoreWork: doMoreWork
  };
}());

console.log("time: " + aModule.time);
aModule.doWork();
aModule.doMoreWork();
```

Single Page Application (SPA)





What Makes an App?

- Good user interaction
- No screen refreshes
- Data
- Interactive components

What Makes an App? [cont.]

- A desktop experience on the web
- A back-button that works well
- URL's that can be bookmarked

Coding an SPA

- Event driven
  - Users interact with HTML
  - JavaScript handles non-semantic browser events
    - Event listening
    - jQuery event delegation
  - Modules interact via semantic events
    - Creating custom events
    - jQuery trigger
- A lot of code is spent managing user events
  - Wouldn't it be better to code the business logic vs. the application boiler plate!

Coding an SPA [cont.]

- Heavy DOM manipulation
  - Changes in the HTML structure are managed via DOM parsing
  - You need to cozy up with your CSS selector skills
  - All the weight of finding elements and changing them on the fly is placed on the developer
- A lot of code is spent managing the DOM
  - Wouldn't it be better to code the business logic vs. DOM manipulation!

Coding an SPA [cont.]

- Stateful
  - The beauty of HTML is being able to give URLs to users that get them to a specific point in a website
  - This is hard to do in a Single Page Application
  - Routing, the back-button and application state need to all be managed by the developer
- A lot of code is spent managing the state of the application
  - Wouldn't it be better to utilize a routing component instead of rolling your own!

Angular Introduction

What is it?

- Client-side Framework
  - It is written in JavaScript
- Used for building heavy weight JS
  - Helps in organizing the architecture
  - Useful for making web applications vs web pages
  - Makes User Interface code updating easier

Prescriptive

- Angular has its recommended way to build apps
  - It is a framework not a library
- You will follow where others have tread
  - Instead of trailblazing =)

Application Focused

- Angular facilitates single page applications
  - No more jumping from page to page with full page refreshes
  - The page only loads once and pieces are fetched and dynamically injected into the page
  - Ajax on steroids, yet manageable =)

Manage the SPA

- Angular will:
  - Get the application it running
  - Simplify the user-event interaction
  - Abstract away the DOM manipulation
  - Handle the routing
  - Simplify the testing process

Angular is Declarative

- A typical single page application fetches small fragments of HTML
  - The HTML however doesn't really describe the purpose of these chunks
  - Your going to need HTML comments or a good class modular naming convention like SMACSS
- Angular is declarative
  - It specifies directly in the HTML what the snippet is about
  - It does this through augmenting HTML with its own custom tags
    - Angular calls these **directives**

Declarative [cont.]

Angular: Custom Directives

```
<tabs>
  <tab title="Lemonade">The Lemonade Stand</tab>
  <tab title="Sales">Sales</tab>
</tabs>
```

Plain HTML

```
<ul class="tabs">
  <li class="tab">The Lemonade Stand</li>
  <li class="tab">Sales</li>
</ul>
```

Angular Creates Data-binding

- Without data-binding
  - Typically we need to do an Ajax call
  - We get data back
  - We traverse the DOM
  - We update the HTML with the Data
- We need to know where our element is in the DOM and then we have to manually update it

Creates Data-binding [cont.]

- Without data-binding

HTML

```
<html>
  <body>
    <input id="feelingInput" type="text" />
    <button id="feelingSubmit">Apply feeling</button>
    <h1>I am feeling <span>happy</span></h1>
    <script src="jquery.js"></script>
    <script src="main.js"></script>
  </body>
</html>
```

JS

```
$('#feelingSubmit').on('click', function() {
  var inputValue = $('#feelingInput').val();
  $('h1 span').html(inputValue);
});
```

Creates Data-binding [cont.]

- With Angular's data-binding
  - Lots of boilerplate code can be eliminated to manage the DOM
  - We can "for the most part" get rid of that back and forth between the User Interface and the JavaScript
  - Angular's magic will handle it for us
- Angular allows us to more easily separate the DOM manipulation and our JavaScript logic
  - No need to worry so much about event listeners
  - No need to do heavy weight DOM manipulation

Creates Data-binding [cont.]

- With data-binding
  - hmmm… that was easy

```
<body>
    <input ng-model="feeling" type="text" />
    <h1>I am feeling {{feeling}}</h1>
</body>
```

Creates Data-binding [cont.]

- With data-binding
  - **ng-model:**
    - Angular directive that allows us to specify a model backing this view
    - It let's us bind the **feeling** attribute
    - Anything that goes into the input tag will be bound to the model object

```
<body>
    <input ng-model="feeling" type="text" />
    <h1>I am feeling {{feeling}}</h1>
</body>
```

Creates Data-binding [cont.]

- With data-binding
  - **{{feeling}}**: Template in our view showing the value of our ng-model directive **feeling**
    - Angular evaluates the templates (i.e. views) and changes them into the correct value
    - Views present the model backing them

```
<body>
    <input ng-model="feeling" type="text" />
    <h1>I am feeling {{feeling}}</h1>
</body>
```

Model - View - ??

Model-View-Controller

- Typical Controller in MVC
  - Grabs data from our data layer (i.e. the Model)
  - Uses services for manipulating and shaping the data
  - Backs the presentation layer (i.e. the View) with information to display to our users
    - Usually adds things on the request object
  - The View pulls that information and places it into the HTML
    - A view is usually a complete HTML page (e.g. a JSP)

Model-View-Whatever

- Angular Controller
  - Controller never has direct reference to the view
  - Controllers the relationship between the Model and the View
    - Via a $scope object
    - Keeps business logic encapsulated from the presentation
  - Core logic of your angular application

Model-View-Whatever [cont.]

- Model
  - The data that will be backing a view
  - It will be used in your User Interface
    - The Model doesn't interact with the view
    - It is mostly pure data

Model-View-Whatever [cont.]

- View
  - The User Interface
    - What the user sees and interacts with
    - The final rendered HTML
  - A display of the Model's state
    - Anytime we have a model change the view will be updated instantaneously
  - It's main purpose is knowing how to display data

Sample App

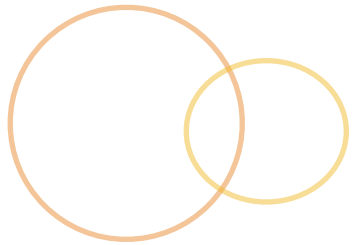
- To truly make this work we need:
  - Download angular.js … not angular.min.js (Grab a 1.2 release)
  - Include an application directive in the root of our code

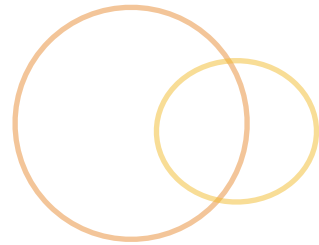
```
<html ng-app="demo">
  <head>
    <script src="angular.js"></script>
  </head>
  <body>
    <input ng-model="feeling" type="text" />
    <h1>I am feeling {{feeling}}</h1>
    <script src="main.js"></script>
  </body>
</html>
```

main.js

```
angular.module('demo', []);
```

The Flash





- Sometimes you might see our template handlebars flash on the screen
  - Browser renders the HTML before Angular has a chance to interact with it

- How else could we handle this?
  - Put angular.js in the document head
  - ng-cloak directive
  - ng-bind directive … we will look at this in a bit

ng-cloak Directive

- Hide the body until angular has been loaded
  - ng-cloak will remove itself when angular boots

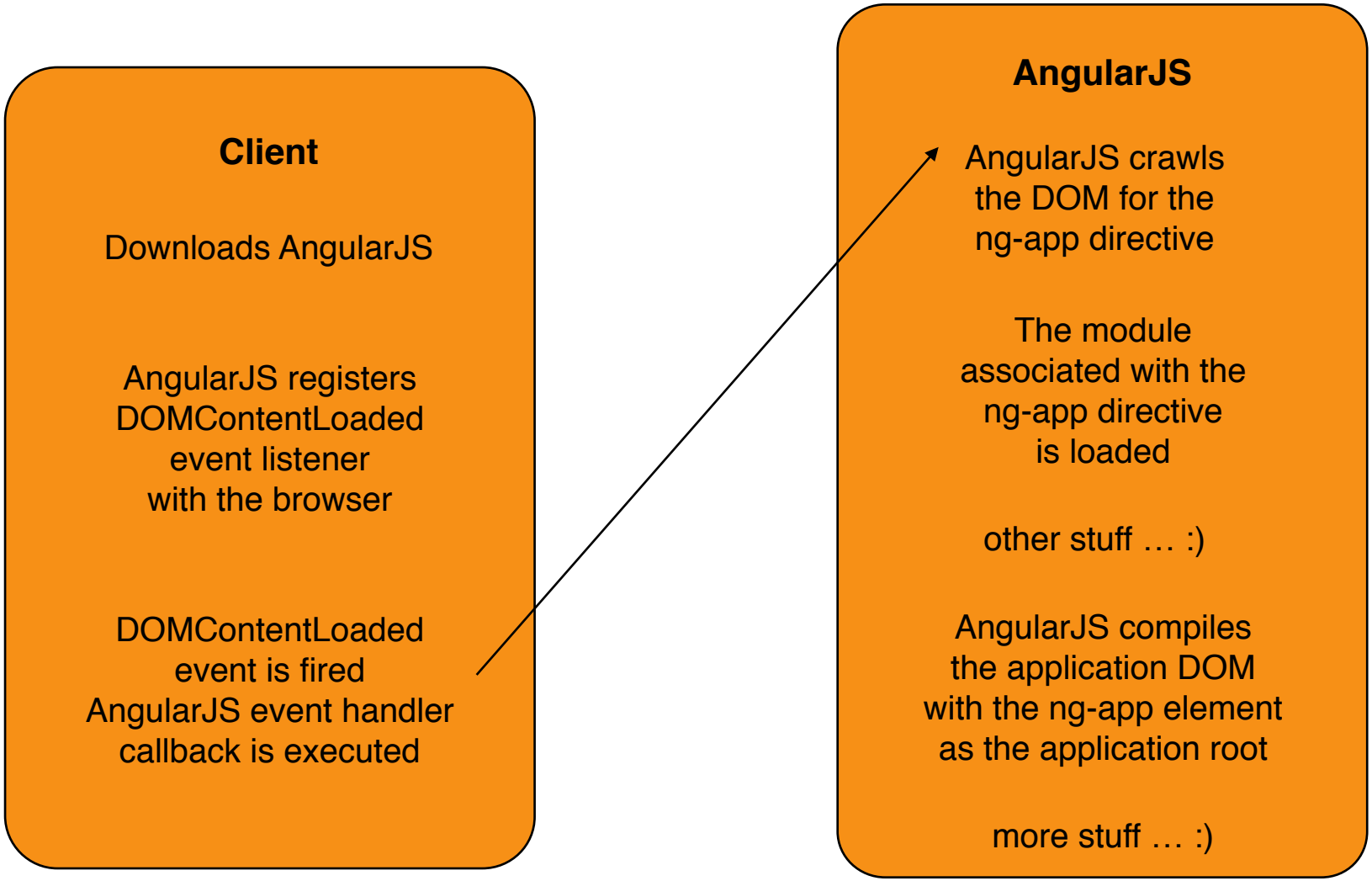
```
<html ng-app="demo">
  <head>
    <style>
      [ng-cloak] {
        display:none;
      }
    </style>
  </head>
  <body ng-cloak>
    <input ng-model="feeling" type="text" />
    <h1>I am feeling {{feeling}}</h1>
    <script src="angular.js"></script>
    <script src="main.js"></script>
  </body>
</html>
```

Application Directive

- **ng-app**:
  - Declares that all of the child elements encompassed by this directive are part of an Angular application

```
<html ng-app="demo">
…
</html>
```

Application Startup



Angular Modules

- **angular.module**
  - Modules will allow us to encapsulate code so that it is not just sitting on the global namespace
  - Angular's main way of creating apps
  - 'demo': The name of the module we are creating
    - Corresponds to ng-app
  - [ ]: List of dependencies we need for this Angular application
    - This is an array of strings
    - They are injected before this module is loaded

main.js

```
angular.module('demo', []);
```

Angular Modules [cont.]

- Modules are also used for application reusability
  - They facilitate a coding structure
  - They keep similar logic grouped together
  - They simplify testing

Angular Modules [cont.]

- Create an application module
- Create modules for features you need
- Create modules for reusable code
  - Directives and filters

Angular Modules [cont.]

- Configuration
- Run Blocks
- Constants
- Order

Angular Modules: Configuration

- Configuration blocks
  - **.config(function() {})**
- The configuration block only takes one parameter, the configuration function
  - We can dependency inject providers for services into that configuration function (i.e. **aMadeUpServiceProvider** will work)
  - We can't inject instances of services into the configuration (i.e. **aMadeUpService** will not work)
  - We can inject constant services because they don't ever change

Angular Modules: Configuration [cont.]

- Configuration takes place when the current module has been loaded
  - Configuration blocks are provided behind the scenes for things like factories and directives
    - We will look at these more in those sections

Angular Modules: Main Method

- Modules don't have a main method
  - Run blocks are as close as we can get
  - **.run(function() {})**
  - Run blocks only take one parameter the run initialization function

Angular Modules: Main Method [cont.]

- The **run** initialization function is executed once **all** modules have been loaded
  - Remember: the configuration function is executed once that specific module has been loaded
  - We can inject service instances into the run block (i.e. **AMadeUpService** will not work)
  - We can't inject service providers into the run block (i.e. **AMadeUpServiceProvider** will work)
  - We can inject constant services into the run block

Angular Modules: Main Method [cont.]

- First code executed in an Angular application
  - Unit testing run blocks is difficult
  - Where we could setup global event listeners

Angular Modules: Constants

- Constant services will be placed first in our module configuration
  - They will run before our configuration blocks for the specific module they are created on
  - **.constant(function() {})**
  - All we are going to say about constants for now :)

Angular Modules: Ordering

- Let's say we have a our main application module and a services module
- When will the constant, configuration and run blocks execute for each
  - Sub-module constant service (.constant)
  - Sub-module configuration block (.config)
  - Main module constant service (.constant)
  - Main module configuration block (.config)
  - Sub-module run block (.run)
  - Main module run block (.run)

Angular Modules

- We will see better how service providers & configuration blocks along with service instances & run blocks play off each other in the services section of the course

$scope object

- We talked about a model object
  - This is referenced in the **$scope** object
- A JavaScript object that holds properties the view will make use of
  - Controllers and Directives interact with the $scope
  - Controllers and Directives don't reference each other
- In our Controller/Whatever/ViewModel we can also interact with these properties

```
<body>
  <input ng-model="feeling" type="text" />
  <h1>I am feeling {{feeling}}</h1>
</body>
```

$scope object [cont.]

- **$scope** is utilized within Controllers
- **ng-controller:**
  - Directive that says all the children of this element belong to the declared controller

```
<body>
  <section ng-controller="DemoController">
    <input ng-model="feeling" type="text" />
    <h1>I am feeling {{feeling}}</h1>
  </section>
</body>
```

$scope object [cont.]

- Variables we set on the model $scope object in a controller can be presented in the view

main.js

```
function DemoController($scope) {
  $scope.who = “I”;
}
```

```
<body>
  <section ng-controller=“DemoController”>
    <input ng-model="feeling" type="text" />
    <h1>{{who}} am feeling {{feeling}}</h1>
  </section>
</body>
```

ng-bind Directive

- Hiding the flashes a second way
  - Instead of expression interpolation (i.e. {{ }})

```
<body>
  <section ng-controller="DemoController">
    <input ng-model="feeling" type="text" />
    <h1>
       <span ng-bind="sentence.who || '?'">?</span> am
         feeling <span ng-bind="sentence.feeling"></span>
    </h1>
  </section>
</body>
```

$scope object [cont.]

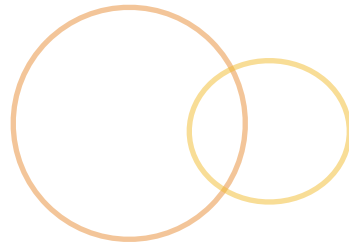
- Best practice
  - We should do bindings on object properties rather than as individual properties on the $scope object
  - Better to group properties together
  - Object references help Angular

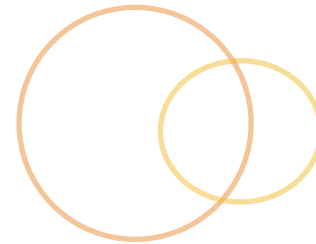
main.js

```
function DemoController($scope) {
  $scope.sentence = {
    who:'I',
    feeling: 'happy'
  };
}
```

```
<body>
  <section ng-controller="DemoController">
    <input ng-model="sentence.feeling" type="text" />
    <h1>{{sentence.who}} am feeling {{sentence.feeling}}</h1>
  </section>
</body>
```

Modules





- Not a good idea to pollute the global namespace
  - Whoops: That is how we just created our DemoController controller
- Remember how I said we really needed to define a module that corresponded to our ng-app directive
  - We will use angular.module to create an application namespace
  - Off that namespace we can add any controllers we need

main.js

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

Modules [cont.]

- We could get a reference to a module at a later point if needed even without assigning it to a variable

```
angular.module('demo');
```

  - This would get us a reference to the **demo** module
  - Looks similar to the way we setup the module, but we are only utilizing one parameter
  - Our module **getter**

Modules [cont.]

- How we should write a controller
- app.controller
  - ‘DemoController’: The name of the controller we are registering

main.js

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

app.controller('DemoController',
function($scope) {
  $scope.sentence = {
    who:'I',
    feeling: 'happy'
  };
});
```

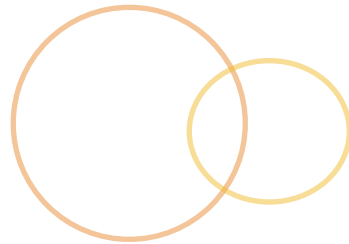
Modules [cont.]

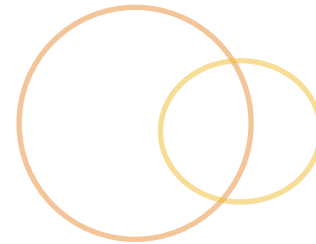
- Utilizes a factory function
  - First time Angular needs to initialize the controller, this function is executed and the result is returned
  - The result is held onto for future interactions

main.js

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

app.controller('DemoController',
function($scope) {
  $scope.sentence = {
    who:'I',
    feeling: 'happy'
  };
});
```

Scope





- What does the $scope refer to?

- $scope is used for
  - Methods on our controllers
  - Properties used in the view
  - Business functionality
- $scope is not the model
  - $scope refers to the model
  - A view updates properties on the model not the $scope
- The model is our JavaScript objects
  - Reuse any JavaScript objects we want

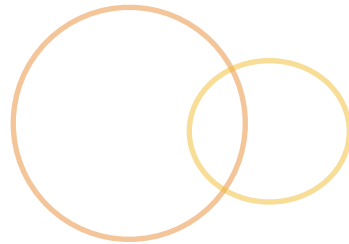
Live binding

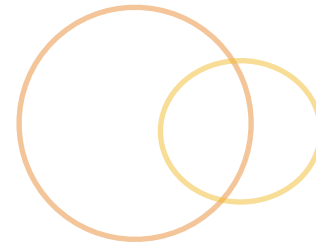
- Anything attached to the $scope is immediately useable in the view
- The $scope will be automatically updated whenever it is modified by the view

Controllers

- We use controllers to add additional functionality to the $scope
- Controllers can add custom actions for the view to the $scope

Directives





- Directives call these actions from within the view
- **ng-click**
  - Directive that is used to call an action on the controller via the view
  - Can take parameters if need be

```
<body>
  <section ng-controller="DemoController">
    <input ng-model="sentence.feeling" type="text" />
    <h1>{{sentence.who}} am feeling {{sentence.feeling}}
      {{sentence.punctuation}}</h1>
    <button ng-click="makeAQuestion()">Questionify</button>
    <button ng-click="makeAStatement()">Statementify</button>
    <button ng-click="commandMe('!')">Command Me</button>
  </section>
  <script src="main.js"></script>
</body>
```

Controllers

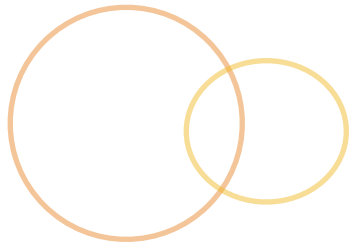
main.js

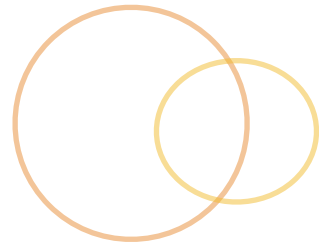
```
app.controller('DemoController', function($scope) {

  $scope.sentence = {
    who:'I',
    feeling: 'happy',
    punctuation: '.'
  };
  $scope.makeAQuestion = function() {
    $scope.sentence.punctuation = '?';
  };
  $scope.makeAStatement = function() {
    $scope.sentence.punctuation = '.';
  };
  $scope.commandMe = function(punctuation) {
    $scope.sentence.punctuation = punctuation;
  };

});
```

Controllers [cont.]

- Notice we are not doing DOM manipulation in the controller
- We keep that out of the controller
  - Focus on binding for the DOM updating

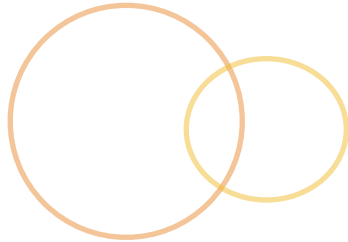
Lab 1

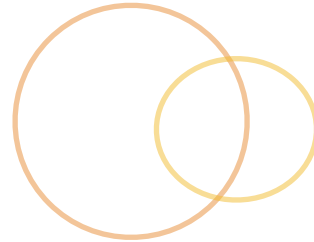




- Begin building a “Lemon-aide” application
- Create a sales page for selling products:
  - Large lemonade, Medium lemonade, Healthy snack, Treat
- Create a controller to handle button interaction
  - Update the individual product totals when the buttons are clicked
  - Update the transaction quantity when buttons are clicked
  - Update the transaction cost when the buttons are clicked
    - Decide how much you want the cost of each product to be
  - Clear the transaction when the “Clear Transaction” button is pressed
- Use ng-cloak to get rid of {{…}} flashing

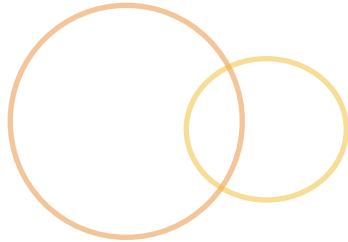
Scope

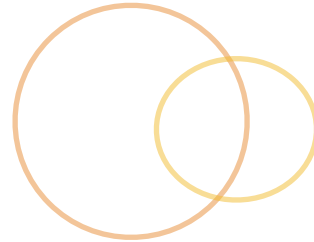




- Scopes have a hierarchy
  - They can be nested
  - They follow the structure of the DOM
- This is similar to the scopes within a function
  - We can have functions in functions in functions

Scope [cont.]





- **$rootScope**
  - This is the penultimate ancestor of all scopes in our Application
  - It is the “root scope” =)
  - Similar to the global window scope
- There is a binding setup between the $rootScope and our **ng-app** element during application startup

Nesting Controllers

- Can be nested
  - We can have parent controllers and child controllers
- A parent controller's $scope will be the parent of the $scope of the child controller
  - Gives the child access to properties/methods on the parent $scope

Nesting Controllers [cont.]

```
<body>
  <section ng-controller="DemoController">
    <input ng-model="sentence.feeling" type="text" />
    <h1>{{sentence.who}} am feeling {{sentence.feeling}}
      {{sentence.punctuation}}</h1>
    <button ng-click="makeAQuestion()">Questionify</button>
    <button ng-click="makeAStatement()">Statementify</button>
    <button ng-click="commandMe('!')">Command Me</button>
    <section ng-controller="ChildController">
      <h2>Isn't this fun? {{sentence.who}} think so.</h2>
      <button ng-click="answerYes()">Yes!</button>
      <button ng-click="answerNo()">Nope</button>
    </section>
  </section>
  <script src="main.js"></script>
</body>
```

Nesting Controllers [cont.]

- Child controllers can access their parents $scope
  - The $scope.sentence object is defined within the parent $scope

main.js

```
app.controller('ChildController', function($scope) {
  $scope.answerYes = function() {
    $scope.sentence.feeling = 'happy';
  };

  $scope.answerNo = function() {
    $scope.sentence.feeling = 'sad';
  };
});
```

Nesting Controllers [cont.]

- Nested $scope creation is done via prototypal inheritance
- So a child scope can have access to its parents scope
- Some directives create a new child scope
  - ng-controller is one example

Prototypal Inheritance

- Objects in JavaScript inherit from each other
- When an object instance is created a __proto__ reference link is created between **object** instance and **Object** constructor
  - No classes in JavaScript only objects
- The prototype chain is created all the way back to the Object
  - Originating object of all JavaScript Objects

Prototypal Inheritance [cont.]

- Setting up a supertype … **UrbanCenter**

```
function UrbanCenter(numOfPeople) {
  this.numOfPeople = numOfPeople;
  this.type = “Urban”;
}

UrbanCenter.prototype = {
  constructor: UrbanCenter,
  getNumOfPeople: function() {
    return this.numOfPeople;
  },
  toString: function() {
    return "Urban Center with " + this.numOfPeople + " people";
  }
};
```

Prototypal Inheritance [cont.]

- Setting up the subtype **City** via Object.create
- Object.create alternative to new constructor
  - This is setting the prototype of UrbanCenter to City

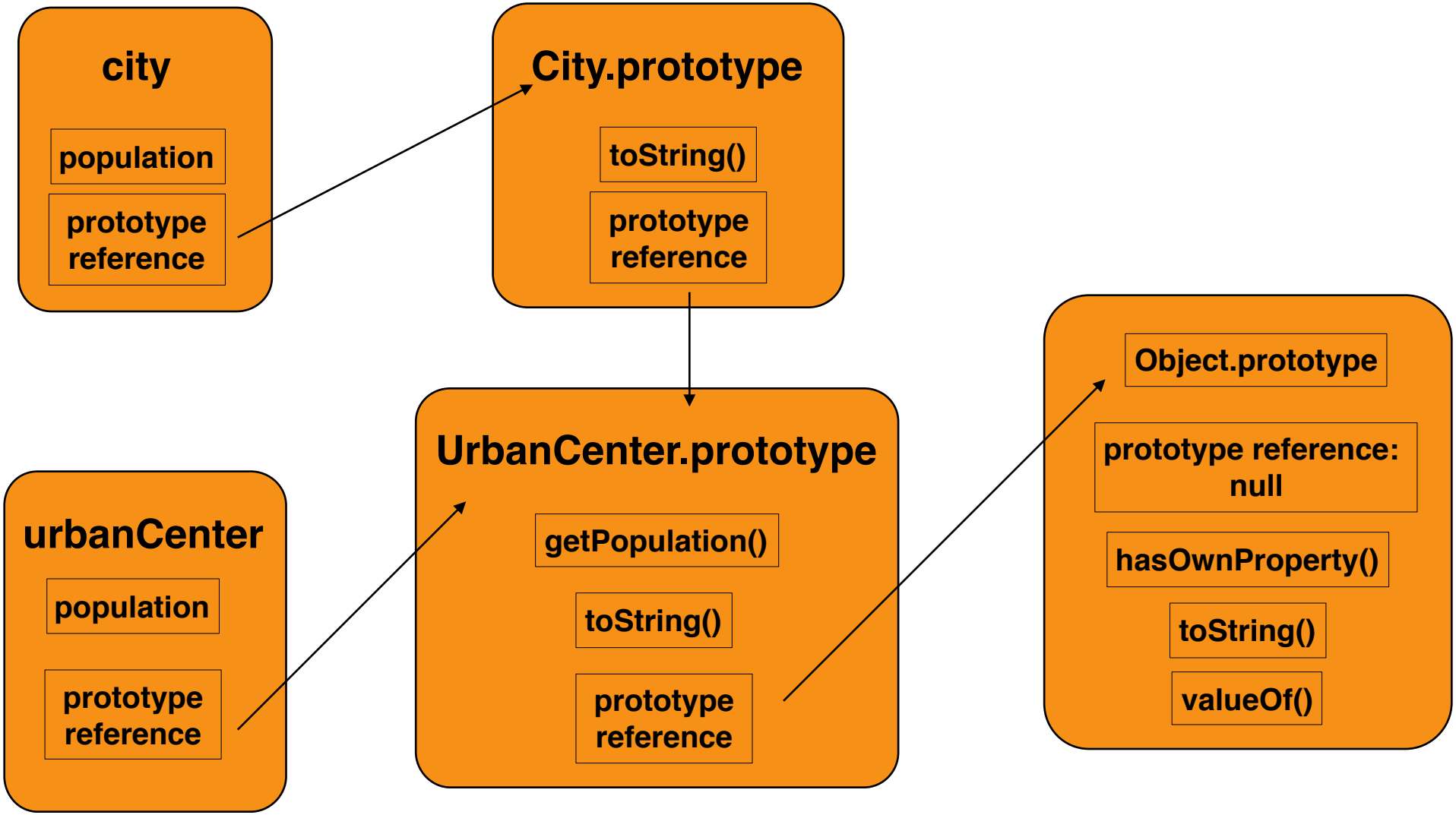
```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
}

City.prototype = Object.create(UrbanCenter.prototype);
City.prototype.constructor = City;
City.prototype.toString = function() {
  return "City with " + this.numOfPeople + " people";
};
```

Prototypal Inheritance [cont.]

- Let's create an Urban Center and a City

```
var urbanCenter = new UrbanCenter(300000);
console.log("Population: " + urbanCenter.getNumOfPeople());
console.log(urbanCenter.toString());
console.log(urbanCenter instanceof UrbanCenter);
console.log(urbanCenter instanceof Object);
console.log(urbanCenter.constructor === UrbanCenter);

var city = new City(350000);
console.log("Population: " + city.getNumOfPeople());
console.log(city.toString());
console.log(city instanceof City);
console.log(city instanceof UrbanCenter);
console.log(city instanceof Object);
console.log(city.constructor === City);
```

Inheritance [cont.]



Prototypal Inheritance [cont.]

- What does this mean for us?

- $scopes are searched for properties and methods
- If we can't find it on a local scope the parent scope will be checked all the way to the $rootScope
- Not on the $rootScope?
  - No updates to the view

Nesting Controllers

- It is possible to “shadow” properties in scopes
- Because of prototypal inheritance we can still access those properties on the parent scope
- $parent allows us access to the parent scope

```
<h1>{{$parent.title}}</h1>
```

Controller as …

- Useful if you find yourself nesting controllers within controllers within controllers and using $parent a lot
  - We are no longer attaching properties to the $scope object but rather to the instance of DemoController … **dc**

```
<section ng-controller="DemoController as dc”>
  <h1>{{dc.title}}</h1>
  <section ng-controller=“NestedController as nc">
    <h1>{{dc.title}}</h1>
  </section>
</section>
```

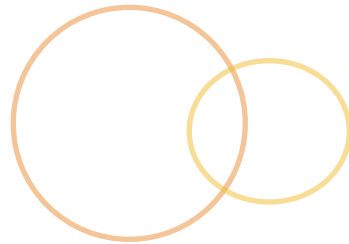
Controller as … [cont.]

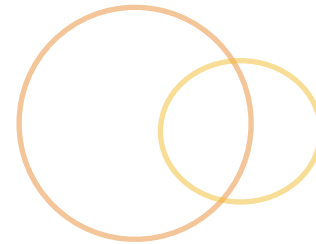
- With “Controller as” we don’t need the $scope in the controller
- We are attaching properties to the controller instance

```
app.controller('DemoController', function() {
  this.title = “Demo Controller”;
});
```

Routing Intro

Routing



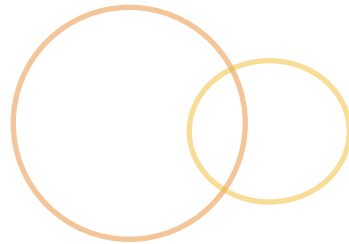


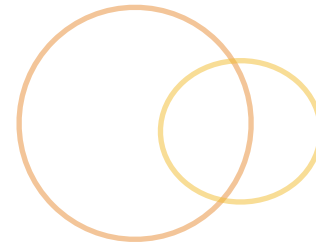
- To make a full fledged single page app we will need to include routing
- Angular's routing allows us to navigate between pages within an application
  - We will not need to do full page reloads to get our application to change pages
  - We can dynamically load sections of code by changing the URL

$routeProvider

- Handles the browser's history
- Allows bookmarking of specific pages (i.e. views)
- As of Angular 1.2 $routeProvider has been pulled out into an ngRoutes module that is not available in the angular.js core
- Grab angular-route.js (i.e. ngRoutes) from:
  - https://docs.angularjs.org/misc/downloading
  - https://code.angularjs.org/
  - Grab the right angular-route.js from the same version of angular.js you have

ngRoutes





- To include it in your application make sure you place it after the Angular JS file

```
<script src="js/libs/angular.js"></script>
<script src="js/libs/angular-route.js"></script>
```

ngRoutes [cont.]

- Because it is its own module we will now need to include it within the dependencies list in our angular module definition

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

ngRoutes [cont.]

- Remember configuration blocks are used to configure modules via service providers
- Once we have included the ngRoute module into the application we utilize the $routeProvider via dependency injection into our application configuration block

```
app.config(['$routeProvider', function($routeProvider) {
  //Configure our Routes
}]);
```

Configuring Routes

- We configure routes with the $routeProvider
- Two methods off of the $routeProvider
  - **when**: Adds a route
  - **otherwise:** Our catchall if no route is found

$routeProvider.when

- The **when** method allows us to add routes
- Takes 2 parameters
  - **path:** The URL where this route can be found
  - **routeObject:** Object literal defining the route properties

```
$routeProvider.when(path, routeObject);
```

$routeProvider.when

- **path:** The URL where this route can be found
- By default our routes are preceded by a sharp
  - http://www.yourdomain.com/#/
- Home route
  - Path: '/'
  - URL: http://www.yourdomain.com/#/
- Route one
  - Path: '/routeOne'
  - URL: http://www.yourdomain.com/#/routeOne
- Route two
  - Path: '/routeTwo'
  - http://www.yourdomain.com/#/routeTwo

$routeProvider.when

- **routeObject**: Object literal defining the route properties
  - template:
    - An HTML string to be converted to DOM elements

```
<section><h1>Route One</h1></section>
```

  - templateURL: The URL of an HTML template
    - templates/routeOne.html
  - controller: The controller backing this route

```
RouteOneController
```

$routeProvider.when

- Putting it together via configuration

```
app.config(['$routeProvider', function($routeProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'templates/home.html',
      controller: 'HomeController'
    })
    .when('/routeOne', {
      templateUrl: 'templates/routeOne.html',
      controller: 'RouteOneController'
    })
    .when('/routeTwo', {
      templateUrl: 'templates/routeTwo.html',
      controller: 'RouteTwoController'
    });
}]);
```

$routeProvider.otherwise

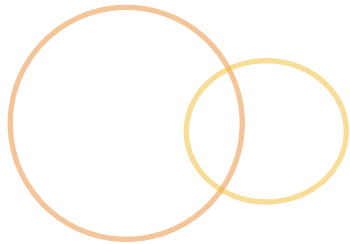
- The **otherwise** method sets up our catch all, when no route can be found
- Takes 1 parameter
  - **routeObject:** Object literal defining the route characteristics
    - Best suited with a **redirectTo**
    - Pointing us to an error page or simply back to the home page

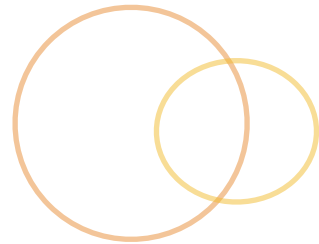
```
app.config(['$routeProvider', function($routeProvider) {
  $routeProvider
    .when('/', {
      templateUrl: 'templates/home.html',
      controller: 'HomeController'
    })
    .otherwise({
      redirectTo: '/'
    });
}]);
```

ng-view

- After we have the $routeProvider configured we need to declare in the HTML where we want our template switching to take place
- The **ng-view** directive gives us that ability
  - It is part of the ngRoute module
  - It signifies a placeholder element in which our templates (i.e. our $route views) will swap into

ng-view





- ng-view creates its own scope and places the view within it
  - It will remove the previous view and reinitialize the $scope for the new view
- When ng-view is all set it fires the **$viewContentLoaded** event

```
<body>
  <header>…</header>
  <section id="content" ng-view></section>
  <footer></footer>
</body>
```

Directive Priorities

- Priorities specify the order in which directives will be invoked
  - This is the order of directives attached to the same element
  - It doesn't matter which directive is declared first if there are different priorities
    - The directive with the highest priority gets called first

Directive Priorities [cont.]

- ng-view is considered a terminal directive
  - It runs at a 400 priority
  - Directives that have a lower priority than it will not be run
    - Hence terminal :)
  - Directives that have the same priority will be allowed to run

Event Listening

- jqLite: is Angular's stripped down version of jQuery
- One of the utilities brought in from jQuery is the ability to attach an event handler via **.on()**
  - **$on** in AngularJS is used in the same manner as .on() in jQuery except
  - However, no data is allowed to be passed through in the event and no selectors are allowed for event listening setup
  - **$on** is used by being attached to the $scope or $rootScope objects

Route Events [cont.]

- **$routeChangeSuccess**
  - Event that is fired when a route change has been successful
  - Event is fired from the $rootScope
- Used by ng-view
  - Signals to instantiate the controller
  - Signals to render the view

Route Events [cont.]

- **$routeChangeSuccess**
  - Takes 3 parameters
    - **event**: Angular event object
    - **nextRoute**: The current route object (i.e. where we changed to)
    - **previousRoute**:
      - The route object where the user came from
      - undefined if there was no previous route (i.e. you started on this route)

```
$scope.$on('$routeChangeSuccess',
  function(event, nextRoute, previousRoute) {
    //Do calculations or set variables
});
```

Route Events

- **$routeChangeStart**
  - Event that is fired before a route change from the $rootScope
  - Angular begins resolving all the necessary dependencies for the route at this point
    - This is when the view template is fetched
    - This is when dependencies on the route are resolved
  - Good place to check for authentication
  - Takes 3 parameters
    - **event**: Angular event object
    - **nextRoute**: The route object we are going to
    - **currentRoute**: The current route object

Route Events

- **$routeChangeError**
  - Event that is fired if there are problems
    - If a promise is rejected (i.e. fails) on one of the dependencies that are to be resolved
  - Event is fired from the $rootScope
  - Takes 3 parameters
    - **currentRoute**: Current route object
    - **previousRoute**: Previous route object
    - **rejectionError**: The error from the promise

Templates

- Via the templateUrl parameter on the routeObject we can specify a template to load
- Think of templates as partial HTML pages
- Those templates will be scoped to the backing controller and they can include model information within them

templates/routeOne.html

```
<section id="routeOne">
  <h1>{{page.title}}</h1>
</section>
```

Dependency Injection

Dependency Injection

- Angular uses Dependency Injection for everything
  - It assembles the application for us
  - Allows us to assemble the application differently for testing
- Dependency Injection allows us to use angular's services
  - Or we could swap for something we like better
  - We do not need to create instances of services
  - We can ask for the service when we need it

The Injector

- $injector service is used for managing Angular's Dependency Injection
  - It figures out which dependencies to use
  - It creates instances of dependencies
  - It instantiates all Angular components
    - Including modules, services, directives, controllers …
- Angular creates the $injector during its bootstrap process
  - In turn the $injector creates the $rootScope after the ng-app module is loaded

Application Startup … updated

Client

Downloads AngularJS

AngularJS registers DOMContentLoaded event listener with the browser

DOMContentLoaded event is fired AngularJS event handler callback is executed

AngularJS

AngularJS crawls the DOM for the ng-app directive

The module associated with the ng-app directive is loaded

The $injector is created. Which in turn creates the $rootScope and the $compile service

AngularJS $compile Service

$compile service links ng-app DOM element with the $rootScope

$compile starts compiling the application DOM starting at the ng-app element as the application root

more stuff … :)

The Injector [cont.]

- Behind the scenes $injector functions

- $injector.**annotate**(function or array)
  - .annotate is passed an array or a function
  - Returns an array of the dependency names that are being injected
    - method: $injector.annotate(function($scope){})
    - return: [“$scope”]
  - How Angular determines what needs to be injected into the function when it is called

The Injector [cont.]

- Behind the scenes $injector functions

- $injector.**invoke**(function($scope) {}): Invokes the function we have created with the supplied arguments
  - **.invoke** first utilizes the annotate function for the dependencies
- $injector.**has**('$scope'): returns true if the dependency is known to the $injector

The Injector [cont.]

- Behind the scenes $injector functions

- $injector.**get**('$scope'): returns an instance of the controller
  - Attempts to locate the dependency and returns an if it has been instantiated
- $injector.**instantiate**(ConstructorFunction): Takes a constructor function and creates a new object with the injected dependencies

Annotation Options

- 3 ways to annotate our functions with dependencies
  - Implicit Annotation
  - Inline Array Annotation
  - $inject Property Annotation
- We need to make sure the $injector will know the correct dependencies to inject

Implicit Annotation

- We have seen this in action already
  - Assume the function parameter names are the same name as the dependencies
  - Very straight forward
- The injector can figure out the services to inject by simply interrogating the names of the parameters
  - Not good to use if minification is needed
  - If $scope gets minified then the $injector.invoke() will try to give a dependency that doesn't exist :(

```
app.controller('RouteOneController', function($scope) {
  //Controller stuff to do
});
```

Inline Array Annotation

- A best practice way of creating controllers
- We pass an array of arguments instead of a function in our controller creation
- These array elements are injected

Inline Array Annotation [cont.]

- Why? Minifiers like this approach
  - Angular Dependency Injection uses reflection for application assembly
  - Strings in JavaScript are never minified

```
app.controller('RouteOneController', ['$scope', function($scope) {
  //Controller stuff to do
}]);
```

$inject Property Annotation

- Similar to inline Array Annotation
  - Can be useful for controllers to have the annotation information assigned directly to the controller
    - Could be cumbersome for services using the factory function
  - Does allow for minifiers to rename the parameters in the function

```
var RouteOneController = function(routeOneScope) {
  //Controller stuff to do
  //  routeOneScope will be understood as the $scope service
});

RouteOneController['$inject'] = ['$scope'];
```

Browser Services

$location Service

- Provides an interface for window.location object
  - https://docs.angularjs.org/api/ng/service/$location
- Similar to window.location except that it is AngularJS aware
  - $location understands the scope life-cycle
- Allows us to configure HTML5 routing mode
  - $locationProvider.html5Mode(true);
  - To get rid of the '#' in your URLs you will need to configure your server

$location Service [cont.]

- Parses the URL
- Allows for path changes
  - Makes redirection easy
- Examples:
  - $location.path() : Gets the current path
  - $location.path('/'): Sets the current path (i.e. it redirects you)
  - This is changing the URL it is not reloading the page

```
app.controller(['$location', function($location) {
  //Interact with the $location service
}]);
```

$location Service [cont.]

- $location.absUrl(): getter
  - http://localhost:8080/angular.html
- $location.protocol(): getter
  - http
- $location.host(): getter
  - localhost
- $location.port(): getter
  - 8080

```
http://localhost:8080/angular.html?user=42030
```

$location Service [cont.]

```
http://localhost:8080/angular.html?user=42030
```

- $location.search(): getter
  - {user: 42030}
- $location.search('user', '42030'): setter
  - http://localhost:8080/angular.html?user=42030
- $location.hash(): getter
  - Gets everything after the #
- $location.hash('foo'): setter
  - Adds #foo to the end of the URL

$location Service [cont.]

- $locationChangeStart
  - Event fired before a URL change
- $locationChangeSuccess
  - Event fired after the URL was changed

$location Service [cont.]

- $location.replace
  - Allows for a URL change without creating a new browser history entry
  - $location setters don't work exactly like window.location
    - Multiple $location interactions can happen as one
    - The .replace() will make the .path() and the .replace() act as one interaction
    - New $location interactions will create new browser history entries unless .replace is specified again

```
$location.path('/changedPath').replace()
```

$window Service

- Reference to JavaScripts window object
  - Used to avoid global window object
- As a service it can be:
  - Overridden
  - Removed
  - Mocked for testing

$window Service [cont.]

- If we want to reload the whole page we can't use the $location service
  - The $window service will allow us to do that

```
app.controller('$window', function($window) {
  //Completely reload the root
  $window.location.href = '';
});
```

$document Service

- A service that simply wraps the window.document object
  - $document[0].title would give us the title element for the page

ng-class Directive

- ng-class allows us to set a class dynamically on an element via an expression
  - It won't add duplicate classes
  - The class **selected** will be placed if the expression on the right evaluates to true

```
<section ng-class="{selected: isSelected}">
  <!– Fun stuff –!>
</section>
```

```
<section ng-class="{selected: 4 == aVarOnScope}">
  <!– Fun stuff –!>
</section>
```

npm

- https://www.npmjs.org/
- Node Package Manager
  - Used to install node.js programs
- With your node.js installation you get npm for free!
- Installing packages

```
npm install package_name
```

npm

- Updating packages
  - Looks for new versions and installs them
  - Resolves the dependencies that package needs
  - Phew! dependency management :)

```
npm update package_name
```

- Is it installed?

```
npm -version
```

Bower

- http://bower.io/
- Front-end package manager
  - Instead of us having to go to Angular's site to get things we can simply use bower
  - Bower takes care of finding the files we need and downloading them for us
  - Bower runs on top of **git**

  - Install Bower through npm

```
npm install -g bower
```

Bower

- Is it installed?

```
bower -v
```

- Grab a package for your local project
  - Defaults to Angular 1.2

```
bower install angular
```

  - Bower allows you to force packages

```
bower install angular#~1.3
```

Bower [cont.]

- We configure where bower will put its downloaded packages through a **.bowerrc** file
  - http://bower.io/docs/config/
  - **directory** property
    - Specify where we want our packages to go
    - If no directory is given Bower will place the packages in a **bower_components** folder in which the **bower** command was given

```
{
  "directory": "app/bower_components"
}
```

Lab 2

- On the sell page
  - Create a sub-controller for SellController that handles the Product interaction
- Make a single page application
  - Create an lemonaide template to show this is about aiding lemonade vendors :)
  - Get $routeProvider working with templates for the sell page and the lemonaide page
  - Include a footer navigation that will switch you between the templates
  - Get the footer links to show the appropriate highlighted link coordinated with the showing template
  - Use ng-bind instead of ng-cloak for {{…}} flashing
- Your node server will be needed for this lab

End