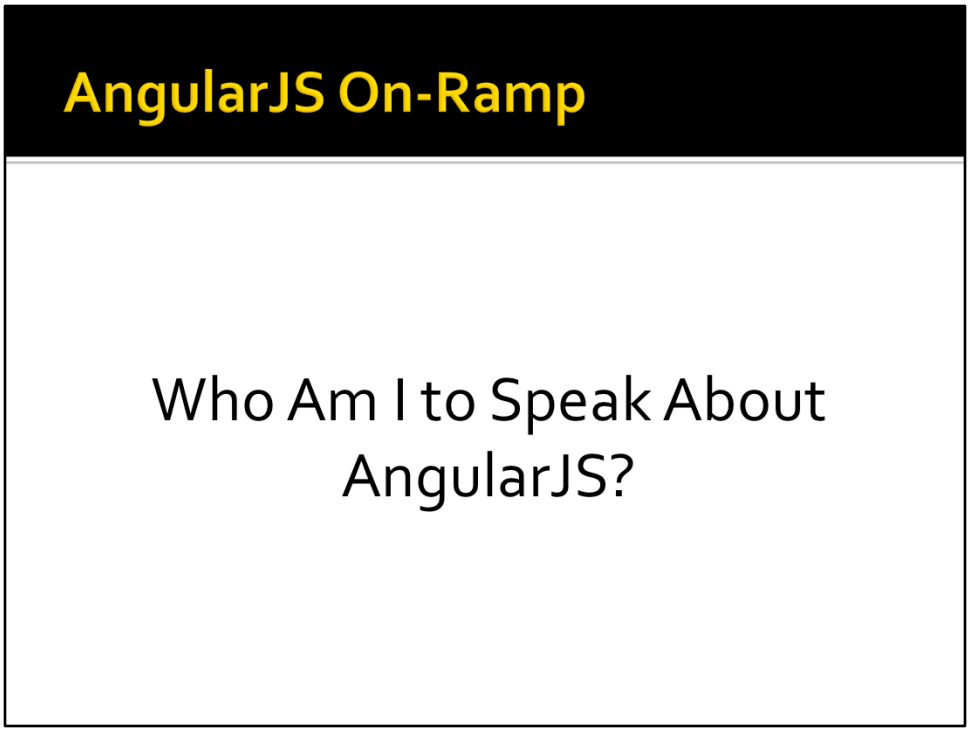
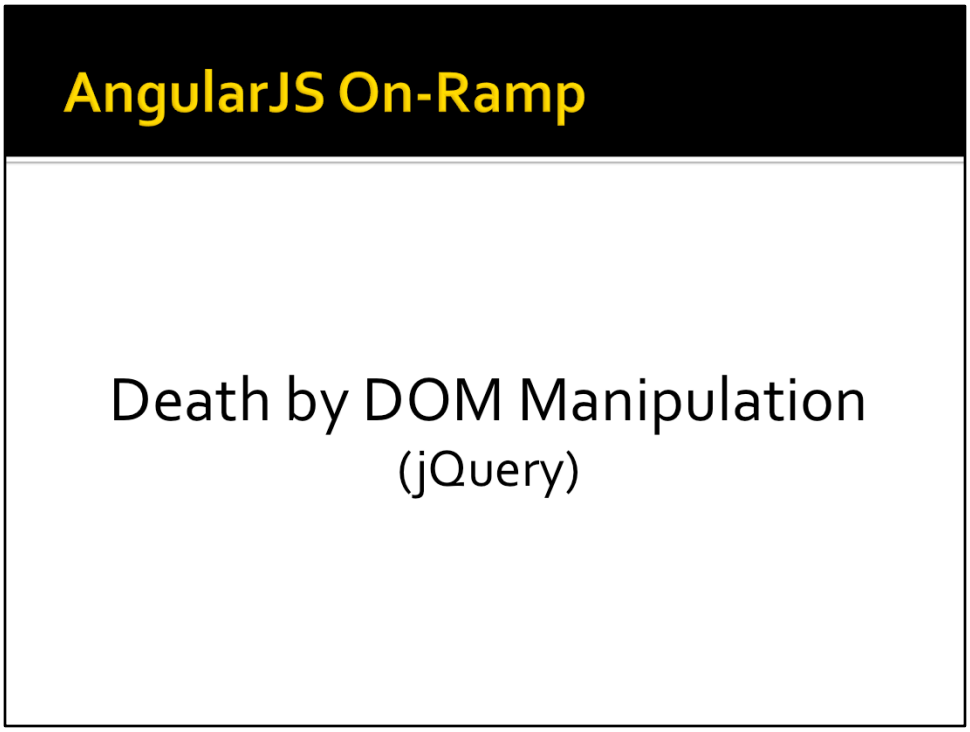
You can find the sample code at: https://github.com/MarkFreedman/AngularJSOnRamp



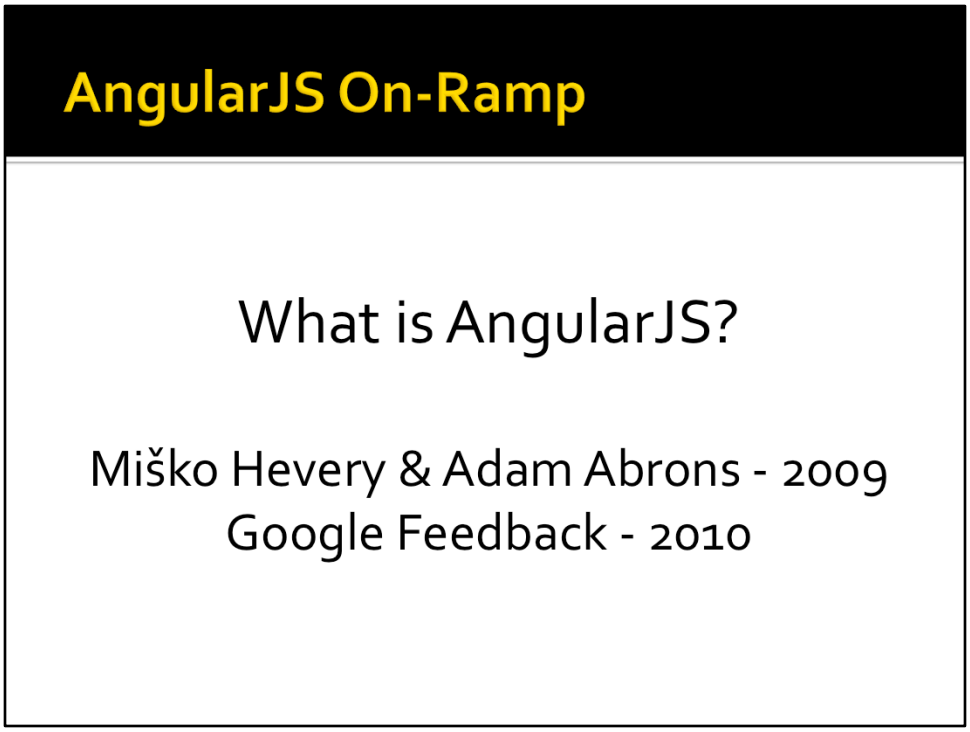
I've been developing web-based apps since the mid 90s, with heavy JavaScript development since 2011. I've been in software development for a few decades. I started using AngularJS in my apps since last year.



Who has been using jQuery, and how is that working out for you?

Has anyone been using AngularJS yet? How far have you gotten so far?

When you look at an HTML file, can you definitely tell me what you're using that DIV for?



It's often described as "what the web browser would have been, had it been designed for applications."

Frameworks like Angular JS are now feasible since JavaScript is no longer slow.

It's an ideal solution for SPAs.

MVC or MVVM.

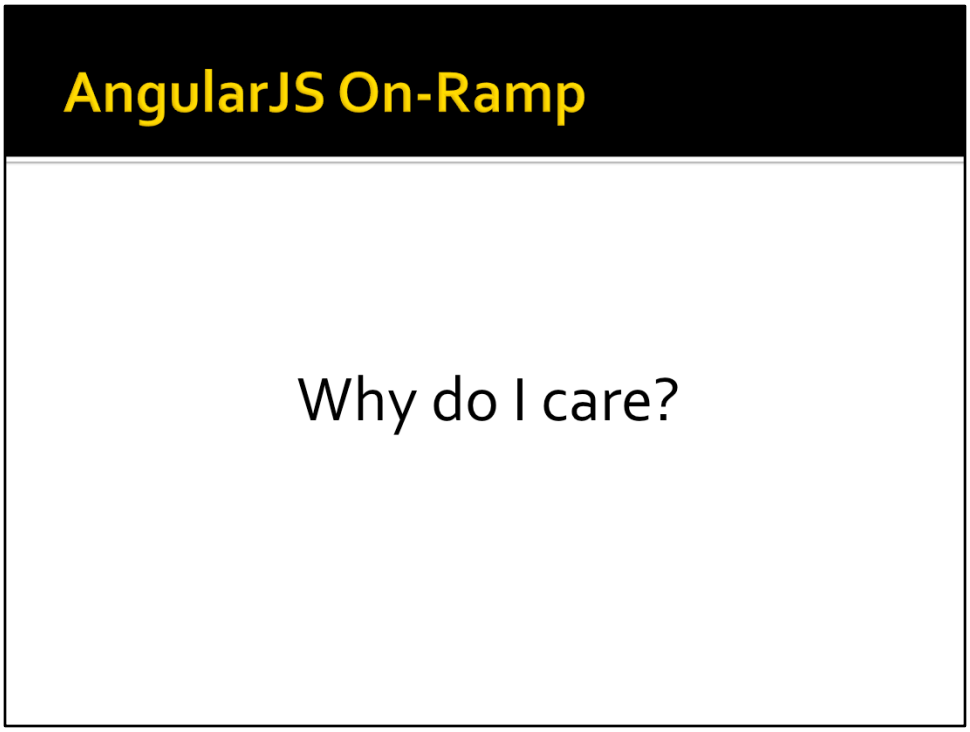
Two-way data binding.

It extends HTML with custom elements and attributes. Angular hints at the future of HTML.

It makes HTML self-documenting.

It follows the belief that declarative programming should be used for UI. Imperative for business logic.

Separation of concerns.

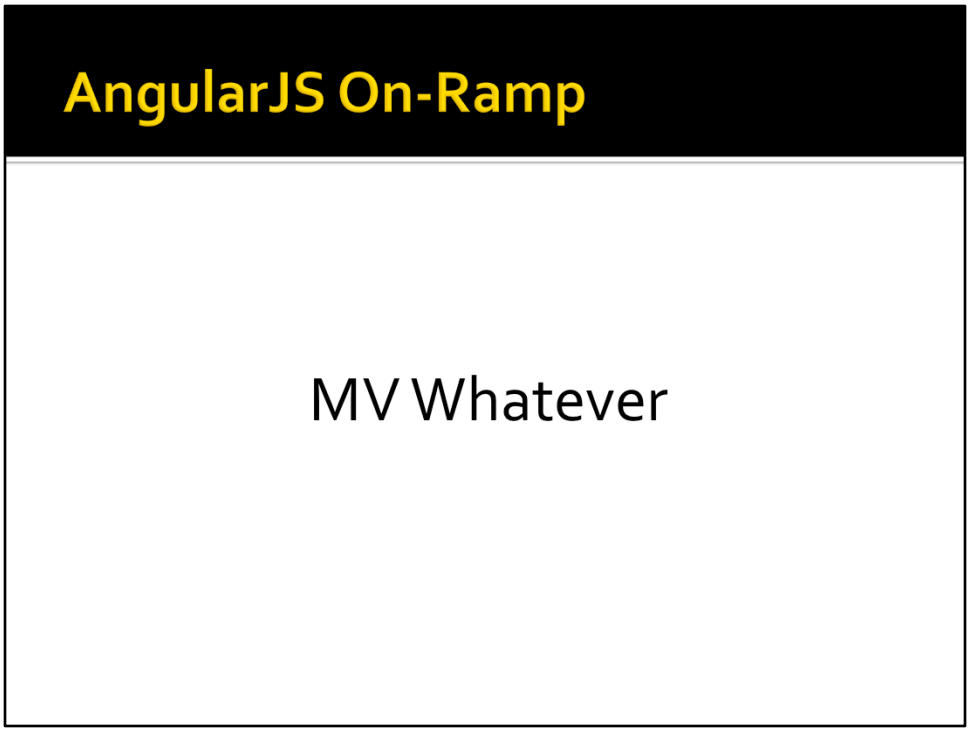


After the initial learning curve, your productivity will get a huge boost, especially when using some of the "seeding" options out there, such as John Papa's HotTowel.

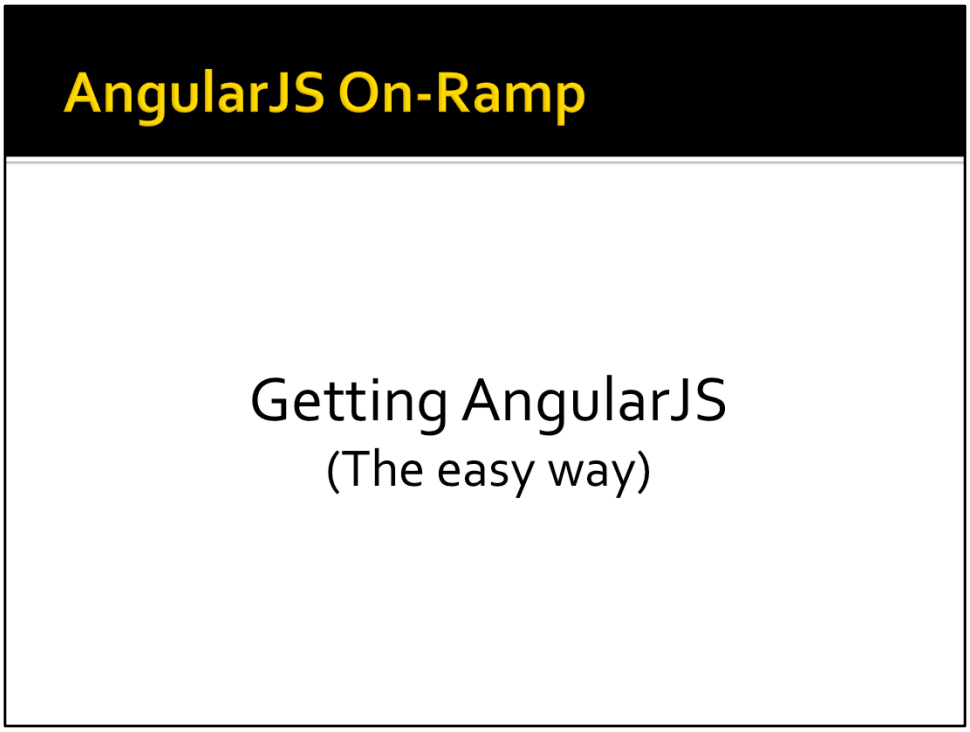
The JavaScript in the apps I've written are maybe 20% the size of what they were when I was using jQuery to manipulate everything.

Yes, you can finally, reliably unit test your JavaScript! AngularJS was build from the start to support testability, and includes mocking features to assist.

Most importantly – they have cool release names ☺



Although many consider it an MVC framework due to the view, controller, and mode ($scope) parts of it, many consider it an MVVM framework due to its two-way binding. I think it provides the best of both worlds.



The easiest way is to install HotTowel, which comes with a seed app structure.

But you can download and install manually, or use a CDN (content delivery network). Since I'm going to be showing the basics, I'll keep this simple, so we don't have to focus on the details of what a seeded environment such as HotTowel gives us.

AngularJS On-Ramp

Canonical "Hello World" Example

canonical.html

AngularJS On-Ramp
Canonical Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>Canonical Example</title>
  </head>
  <body>

    <div data-ng-app>
      <input type="text" data-ng-model="message">
      <h1>{{message + " World"}}</h1>
    </div>

    <script src="angular.min.js"></script>
  </body>
</html>
```

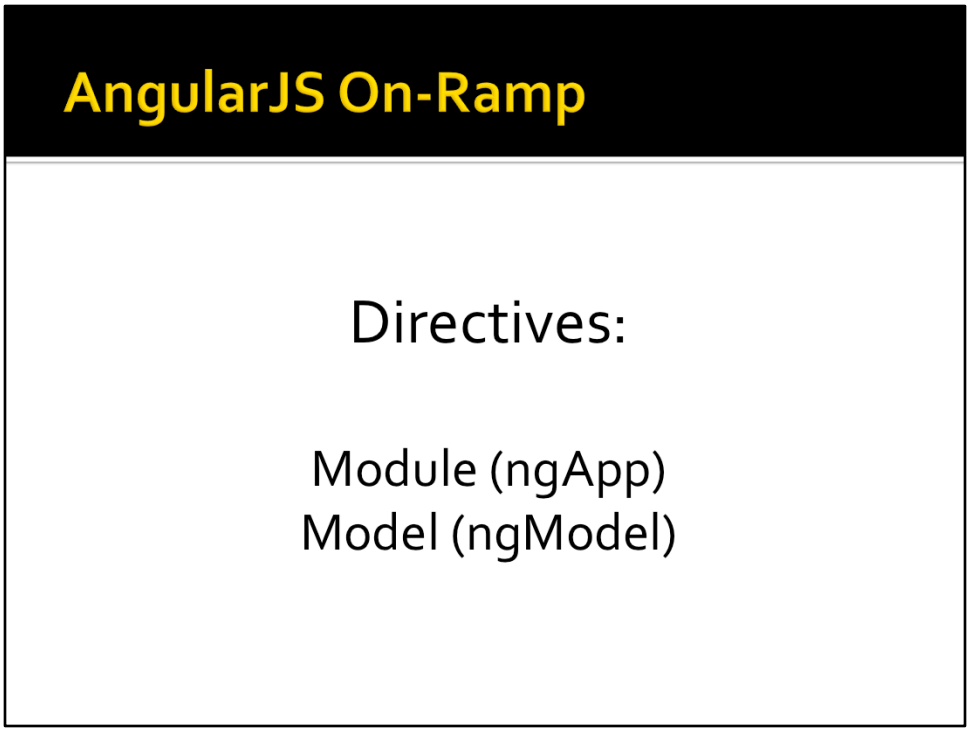
ng-app wraps the part of HTML you want handled by AngularJS. In this example, we're only wrapping a DIV, but normally, we'd wrap the entire page by placing it on the HTML element.

In order to conform to HTML 5's standards, and to not have Visual Studio complain, you can precede AngularJS's built-in attributes with "data-" (data-ng-app).

Data binding is done using the "handlebar" notation within your HTML.

Simple two-way data binding is done using the ngModel directive in conjunction with the handlebar notation.

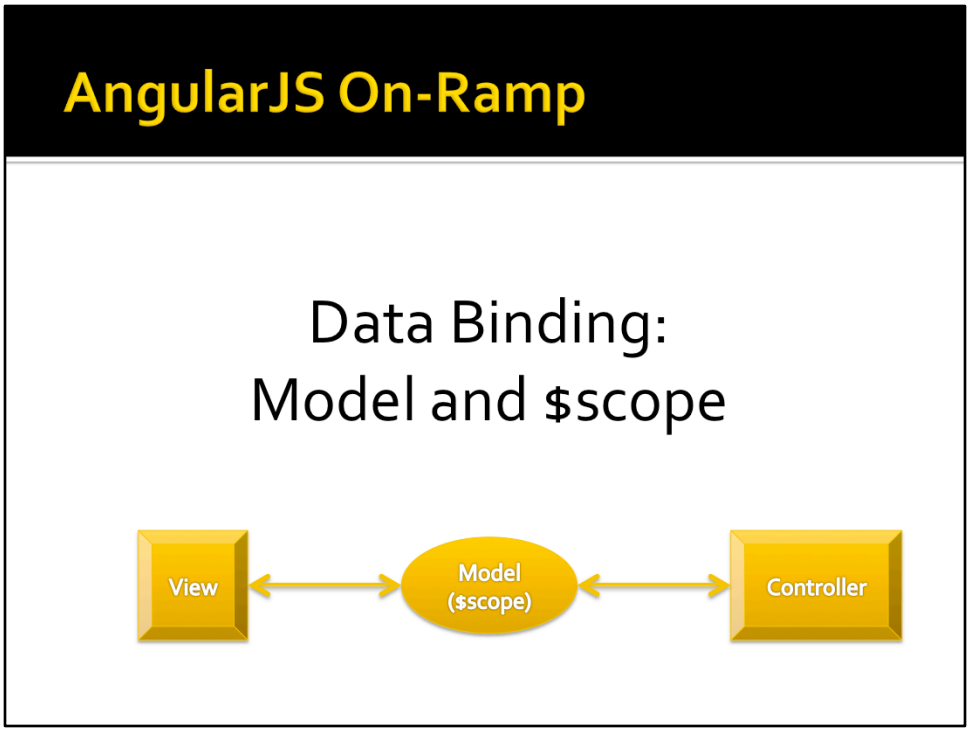
If you dig into the AngularJS JavaScript (I mean, *really* dig), you'll see that, among a lot of other logic, the ngModel directive includes logic to capture "model" updates on each keystroke.



All built-in directives are represented using the camel-case representations of their names. They get converted to HTML-standard naming, replacing camel-case with hyphen-separated, lowercase words. For example, the ngApp directive as coded in JavaScript becomes ng-app in HTML.

Due to semantic HTML rules, many people recommend using the "data-" equivalent attribute names. For example, either ng-app and data-ng-app can be used. Editors and IDEs such as Visual Studio will accept data-ng-app as "valid" HTM, and avoid unsightly squiggles in the editor, although either would work.

Wrap any HTML that you want handled (and pre-processed) by AngularJS by applying the built-in directive, ng-app, to an element. Often, we place this on the <HTML> tag, declaring that the entire page will be handled by AngularJS.



When you use the ngModel directive, you're actually creating a simple "scope" that your controller will use as its "model". Our simple example is good for "demo code." Although our first simple example implicitly creates both a scope (model) and controller, we can (and should) be explicit, and define our controller and scope.

AngularJS On-Ramp

Controllers

Dependency Injection

controller.html

AngularJS On-Ramp
Simple Controller

```
<!DOCTYPE html>
<html data-ng-app>
  <head>
    <title>Controller Example</title>
  </head>
  <body>

    <div data-ng-controller="studentController">
      <ul>
        <li data-ng-repeat="student in students">
          {{student.name}} is in classroom
          {{student.classroom}}, and earned a grade of
          {{student.grade}}.
        </li>
      </ul>
    </div>

    <script src="angular.min.js"></script>
    <script src="studentController.js"></script>
  </body>
</html>
```

As mentioned earlier, ngApp wraps the portion of the page you want controlled by AngularJS. In this example, we're wrapping the entire page. Normally, an app (or an AngularJS "module") will be the entire page. This "auto-bootstraps" an AngularJS application. More on modules, later.

In our first, canonical example, the controller was implied. Here, we're wrapping an area handled by a controller by using the ngController directive on a DIV element. This allows the scope (data model) to be shared with this portion of the view, and the controller, "mainController". Of course, we have to include the JavaScript file containing our controller.

If you notice, we're not using ngModel in this example, because the model we'll be using is defined inside of our controller, which I'll show you in a moment.

Inside of this DIV, we're defining an unordered list. On first appearance, it seems like we only have a single item in our list. But at closer inspection you'll see we're using another powerful AngularJS directive, ngRepeat. It does what the name implies.

In this scenario, ngRepeat will create an LI element for each student in our students object, which we're defining on our $scope in our controller. We'll look at that in a second, but if you notice, student is just a POJO with at least three attributes; name,

AngularJS On-Ramp
Simple Controller

```
function studentController($scope) {
  $scope.students = [
    {
      name: "John Doh",
      class: 6,
      grade: 93
    },
    {
      name: "Steve Smith",
      class: 5,
      grade: 72
    },
    {
      name: "Jane Doe",
      class: 7,
      grade: 87
    },
    .
    .
    .
  ];
}
```

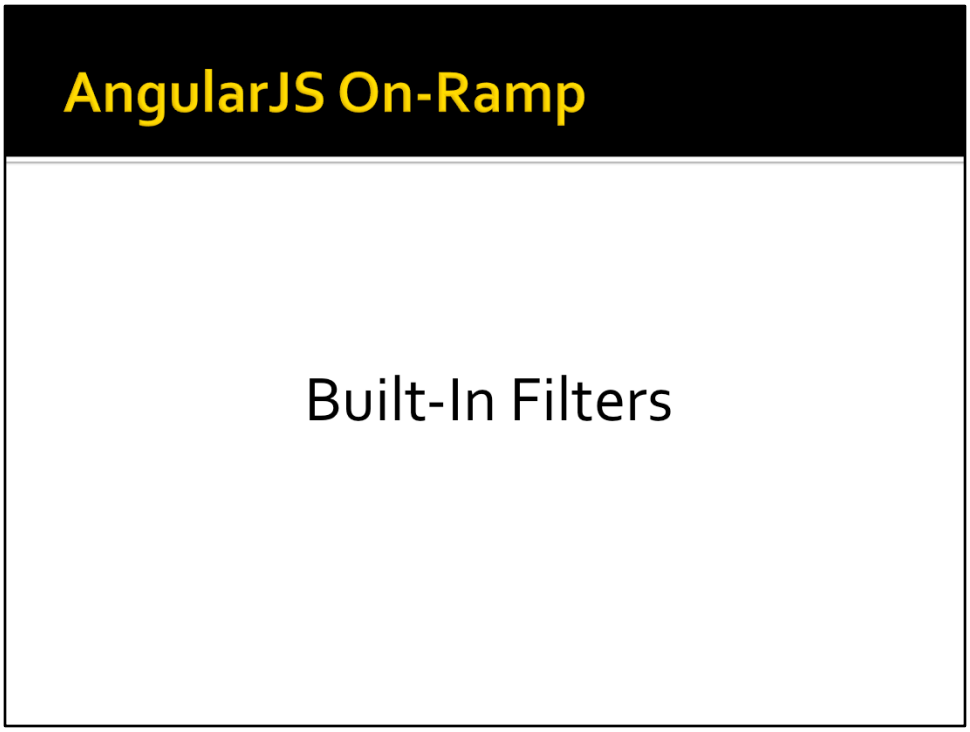
The controller name does not have to be the same name as our JavaScript file, of course. Often, we'll have several common controllers in a single JavaScript file.

Dependency injection is used throughout AngularJS. In this example, we're injecting our view model ($scope) into our controller. This is the glue between our view and the controller.

Anything we add to our $scope can be a POJO. Here, we're creating a students "repository" (or in our simple example, an array of students). We can do a lot more in our controllers, but in this example, all we're doing is initializing our view model.

You can see that each element name of our student objects match what we're outputting in our HTML.

You'll also notice that we did not have to prefix "students" in our ngRepeat directive on our page with $scope. $scope is implied.



builtInFilter.html

AngularJS On-Ramp
Built-In Filter

```
<!DOCTYPE html>
<html data-ng-app>
  <head>
    <title>Controller Example</title>
  </head>
  <body>

    <div data-ng-controller="studentController">
      <input type="text" data-ng-model="search.name">
      <ul>
        <li data-ng-repeat="student in students | filter:search | orderBy:'grade':true">
          {{student.name}} is in classroom
          {{student.classroom}}, and earned a grade of
          {{student.grade}}.
        </li>
      </ul>
    </div>

    <script src="angular.min.js"></script>
    <script src="studentController.js"></script>
  </body>
</html>
```

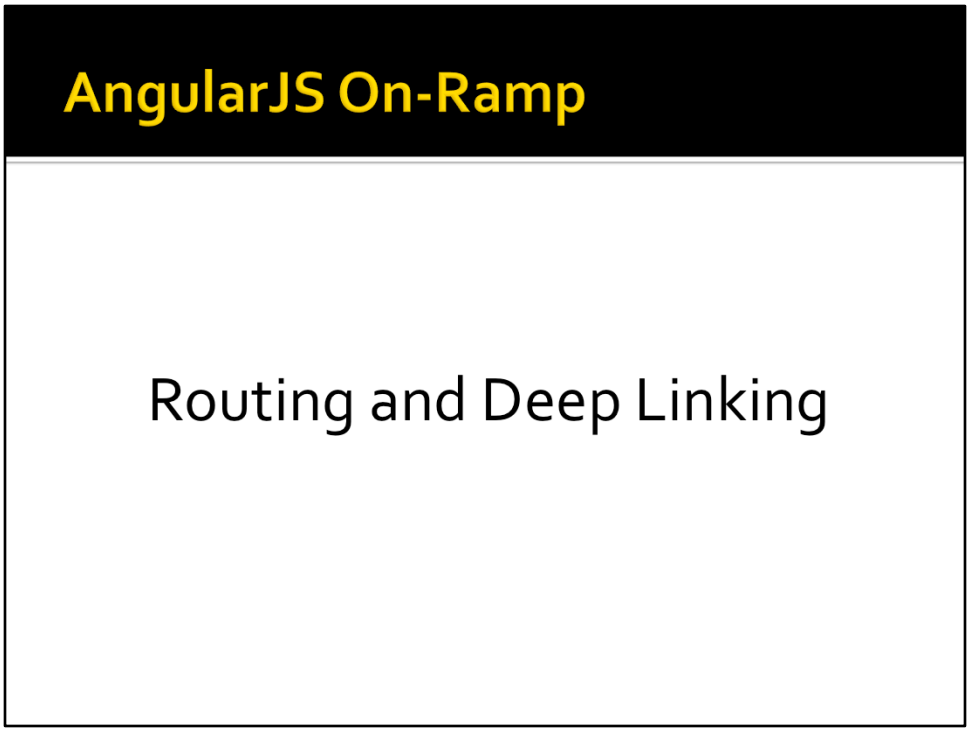
Now let's use a built-in filter to filter student names in our list.

We're using an INPUT element, and adding "search.name" to our view model. Even if we're using a controller, and initializing our $scope within the controller, we can still introduce model objects in our HTML, like we're doing here. I only recommend using this for one-offs, like filtering, though.

The most important new piece here, though, is the filter that we're piping our students array through. The "|" character is the "piping" character, which you may be familiar with using command line tools such as the Windows command prompt or PowerShell, or Bash, etc. "filter" is a built-in AngularJS filter. Don't get confused by the fact that this particular filter is named "filter". It's coincidental. AngularJS has several other pre-defined, built-in filters, such as "uppercase" and "lowercase" as well. As you can determine, these allow us to filter data through some sort of logic, to produce a modified result before displaying or otherwise making use of a value.

Also note that since "search.name" is specified for ngModel, the only field the list will be filtered on is the name field. If we wanted to filter across all fields, we would just use "search" for ngModel.

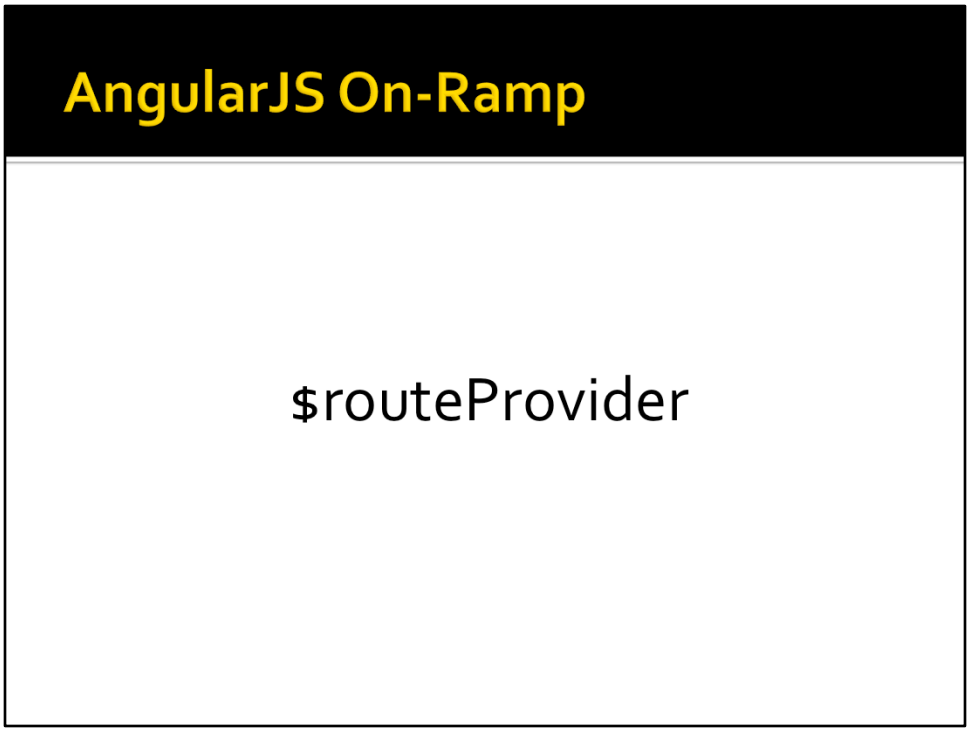
Also note that we did not have to do anything in JavaScript to support this. We'll



index.html

Unlike classic Web apps, where we handle routing on the server side, we have to resort to some advanced techniques to handle routing in SPAs. Thankfully, AngularJS has a route provider service that we can configure that can take care of routing and "deep linking" for us. Search engines will love us for it, and so will our customers.

Typically, in a classic Web app, when we dynamically change the view of our page, through AJAX and/or DOM manipulation, the URL of the original page does not change. Of course, this makes it impossible for users to bookmark a particular "state" of the page. But thanks to AngularJS routing, and the ngView directive, this is no longer an issue.



In order for routing to be supported, we need to inject the included AngularJS service, $routeProvider, into our module (app).

You've seen the ngApp directive used in our previous examples, but now we're actually going to set some configuration settings for our app (which is synonymous with "module") for setting up our routing scheme.

AngularJS On-Ramp
Configuring Routing

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

app.config(['$routeProvider', function ($routeProvider) {
  $routeProvider.
    when('/', {
        templateUrl: 'partials/home.html',
        controller: 'HomeController'
    }).
    when('/classrooms', {
        templateUrl: 'partials/classrooms.html',
        controller: 'ClassroomController'
    }).
    when('/students/:classroom', {
        templateUrl: 'partials/studentsClassrooms.html',
        controller: 'StudentController'
    }).
    when('/students', {
        templateUrl: 'partials/students.html',
        controller: 'StudentController'
    }).
    otherwise({
        redirectTo: '/'
    });
}]);
```

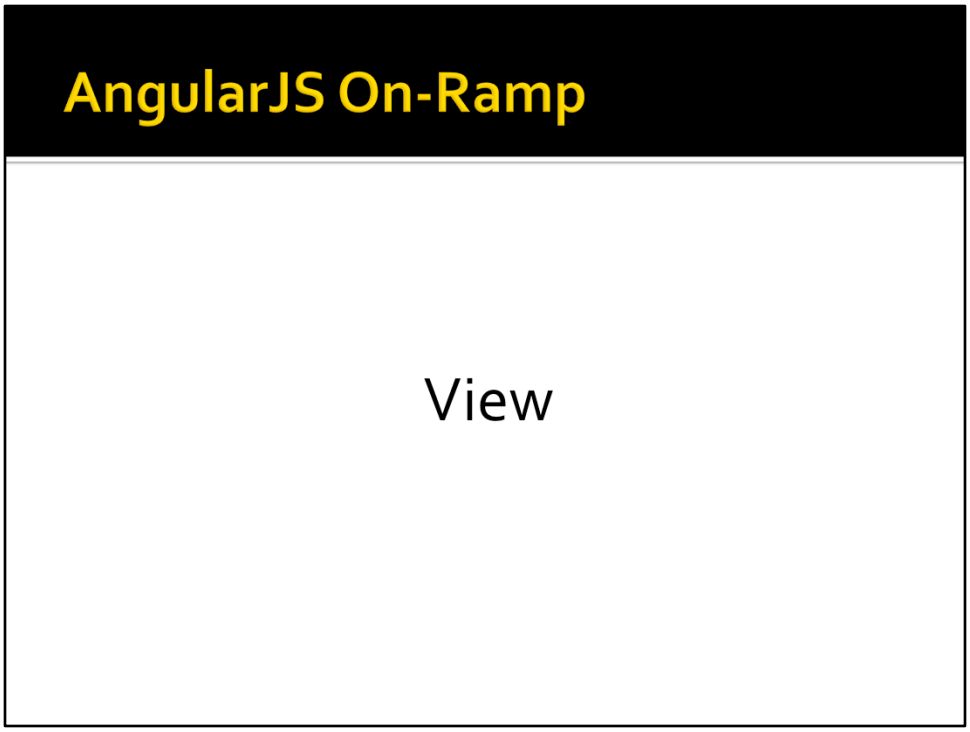
We're going to create a new JavaScript file for this sample application called app.js. Up front, we're declaring an "app" variable to point to our main module, and we're going to name our module (or app), "app". We'll look at how we reference this module in our HTML in a second.

In the meantime, notice that we are "injecting" "ngRoute" into the module. We inject libraries into an AngularJS module by passing in an array of string values representing these services. All AngularJS services started with a $. You can create your own services (which we will do later) as well.

Once we've created an instance of this module, we can start calling built-in methods. For modules, a key method is "config". In our config, we're simply setting up our routing scheme.

As shown earlier, we're injecting services to our AngularJS objects (functions) by passing them in as parameters. Here, we're passing in the $routeProvider, which is part of Angular's ngRoute library.

Here, we specify each route to use, and which view template file and controller to use for each route. We also specify a default route to use if an invalid route is attempted. We could alternatively use a "template" parameter instead of



Now let's look at the main page, the main "view", itself, and see how we incorporate our partials we defined in our app configuration.

AngularJS On-Ramp
Main Page (with ngView)

```
<!DOCTYPE html>
<html data-ng-app="app">
  <head>
    <title>School Roster</title>
  </head>
  <body>
    <h1>School Roster</h1>
    <div data-ng-view />

    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular.min.js"></script>
    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular-route.min.js"></script>
    <script src="app.js"></script>
  </body>
</html>
```

First, notice our familiar ngApp directive, with a twist. Here, we're also specifying a value for the attribute, which ties into the name we gave it in the module instantiation of our main JavaScript file. Since we didn't configure anything module-level in our earlier examples, there was no need for us to give the app a name. Most of the time, you will do so.

Further down the page, we're marking a DIV with the ngView attribute directive. This is where the partial view HTML will be inserted based on the URL specified.

Also, notice that we now have to include Angular's routing JavaScript file, as well as our main app.js file.

AngularJS On-Ramp
Partials (Views)

home.html

```
<a href="#/classrooms">List of Classrooms</a><p/>
<a href="#/students">List of Students</a>
```

classrooms.html

```
<h2>Classrooms</h2>
<h3 data-ng-repeat="classroom in classrooms | orderBy:'classroom'">
  <a href="#/students/{{classroom}}">{{classroom}}</a>
</h3>
```

students.html

```
<h2>Students</h2>
Filter: <input type="text" data-ng-model="search.name">
<ul>
  <li data-ng-click="displayStudentInfo(student)"
      data-ng-repeat="student in students | filter:search | orderBy:'grade':true">
    {{student.name}} is in classroom
    {{student.classroom}}, and earned a grade of
    {{student.grade}}.
  </li>
</ul>
```

studentsClassroom.html

```
<h2>Students by Classroom - Room {{classroom}}</h2>
<ul>
  <li data-ng-repeat="student in students | filter:classroom | orderBy:'name'">
    {{student.name}} earned a grade of {{student.grade}}.
  </li>
</ul>
```

Here are the partial views we'll be using.

As with other frameworks, since partial views are snippets of HTML to appear inside a placeholder, you don't need the HTML or BODY tags (and you must make sure you don't use them).

Notice for the "home" page (as defined by "/" in our routing scheme) that we're preceding our href values with a hash. This is because since we're not truly leaving the page when changing views, AngularJS takes advantage of the built-in hash support of HTML. This signifies a link on the same page. But AngularJS acts upon that by injecting the partial views into the ngView directive on this page.

In the classroom.html partial, we're passing in a parameter, classroom, as a deeper link in the URL. This will use the route with the parameter that we defined earlier.

Also note that we're adding a click handler to each LI tag we're generating, by specifying the ngClick directive. AngularJS has several event handler directives we can take advantage of. Here, we're specifying that when the LI is clicked, the displayStudentInfo() function defined on the scope will be called.

AngularJS On-Ramp Controllers

```
app.controller("HomeController", function () {});

app.controller("ClassroomController", function ($scope) {
  $scope.classrooms = [513, 602, 722];
});

app.controller("StudentController", function ($scope, $routeParams) {
  $scope.students = [
    {
      name: "John Doh",
      classroom: 602,
      grade: 93
    },
    {
      name: "Steve Smith",
      classroom: 513,
      grade: 72
    },
    .
    .
    .
  ];

  $scope.displayStudentInfo = function(student) {
    alert(student.name);
  }

  if ($routeParams.classroom) $scope.classroom = $routeParams.classroom;
});
```

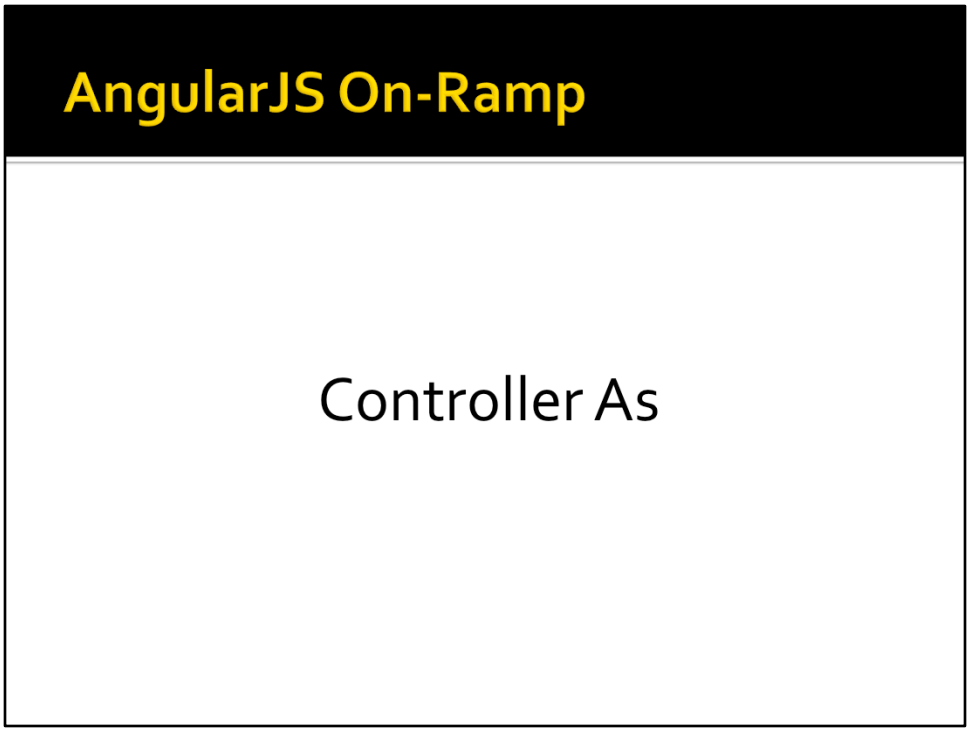
Here are the controllers we also included in the app.js file. Notice that in order to use controllers we specify in our routing configuration, we need to use the controller function on our module (app.controller). There are several ways to package and hook up our controllers; this is one simple way.

Note that we're not doing anything special in our HomeController, but we still need to define it if we've declared it in our route config. We could have left off the controller parameter of our home route config altogether, but I like to keep placeholders there, in case I ever want to add controller functionality later.

Notice that we're injecting $routeParams into our StudentContoller. We need to do that here, since we could be expecting a parameter passed in from the route.

You may have noticed that we're using this same controller for two routes; one with a parameter, and one without. The "if" statement at the bottom checks to see if we passed in a parameter, and if so, we're setting a new variable on our scope for classroom, off of $routeParams.classroom.

Let's go back and revisit partial view studentsClassroom.html to see how we're using this.



controllerAs.html

In a recent version, AngularJS added a “controller as” option to avoid confusion with nested controllers and $scopes. I find this more naturally ties a controller to a view via a view model.

AngularJS On-Ramp
Controller As

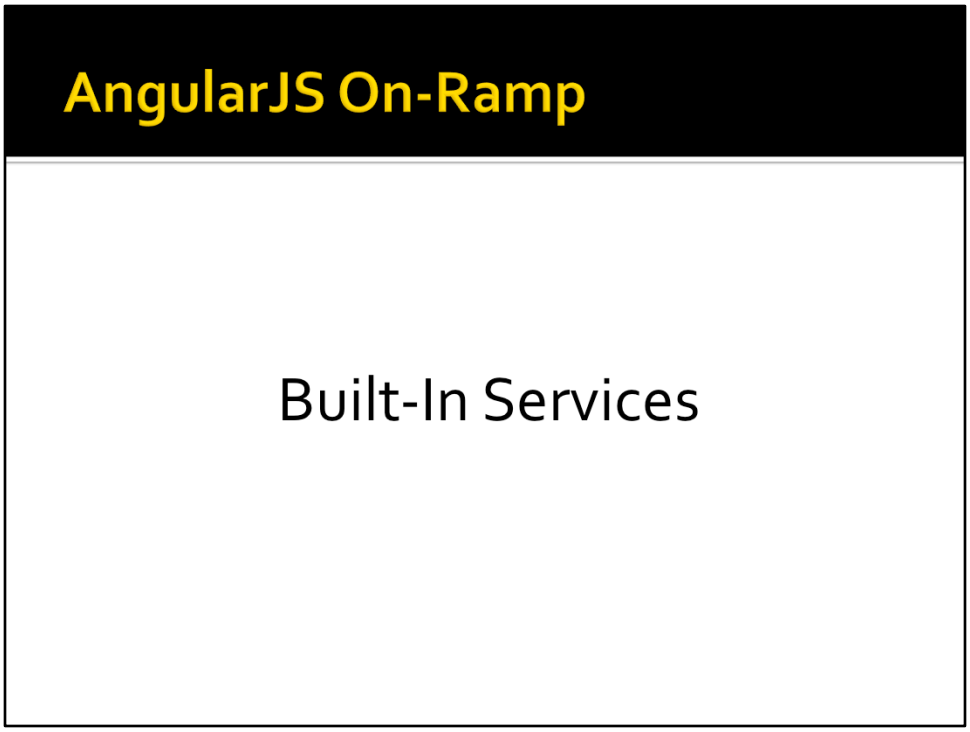
```
<!DOCTYPE html>
<html data-ng-app>
  <head>
    <title>Controller Example</title>
  </head>
  <body>

    <div data-ng-controller="studentController as vm">
      <ul>
        <li data-ng-repeat="student in vm.students">
          {{student.name}} is in classroom
          {{student.classroom}}, and earned a grade of
          {{student.grade}}.
        </li>
      </ul>
    </div>

    <script src="angular.min.js"></script>
    <script src="studentControllerAs.js"></script>
  </body>
</html>
```

Basically, instead of just specifying a controller name, follow the name with “as someScopeName” (I normally use “vm” for “viewModel”).

AngularJS On-Ramp
Controller As

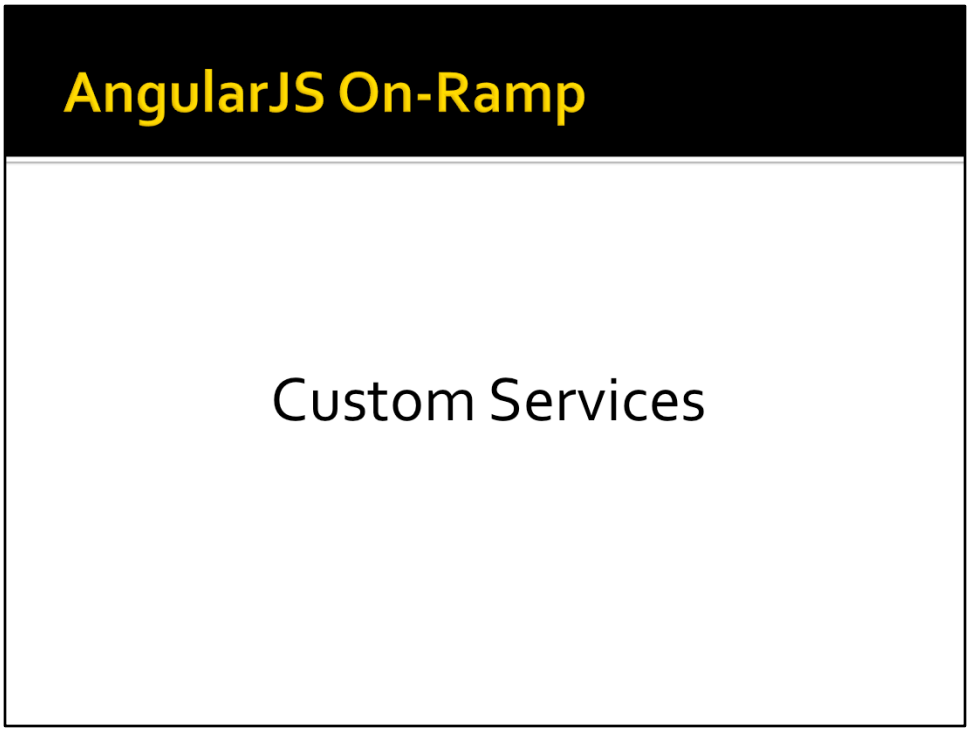
```
function studentController() {
  var vm = this;

  vm.students = [
    {
      name: "John Doh",
      class: 6,
      grade: 93
    },
    {
      name: "Steve Smith",
      class: 5,
      grade: 72
    },
    {
      name: "Jane Doe",
      class: 7,
      grade: 87
    },
    .
    .
    .
  ];
}
```

Then, in the controller JavaScript code, don't bother injecting $scope. All you need to assign objects to the scope is to assign it to "this". But because "this" can be confusing in JavaScript depending upon context, I recommend assigning "this" to the same label you gave the "controller as" setting; in this case, "vm".



Just like in a typical multi-tiered application, we'd prefer to separate our business logic from our controllers. We like to limit our controllers to do what is necessary to glue our views and models together, and to orchestrate the communication between layers of our applications.

AngularJS provides the same capabilities. We've seen an example of injecting AngularJS pre-built services into our controllers; services like routing (ngRoute / `$routeProvider`). Other built-in services include `$http` (which we'll try to look at later) and `$animate`, for animation functionality.



indexWithService.html

As I mentioned before, we like to keep our business logic separate from our controller logic, which should mainly be the glue between our views and models. AngularJS makes it easy to inject services into our controllers, as well as other services.

There are a few ways to create services in AngularJS, but for this intro, we're going to focus on the most common way, which is to create a factory.

AngularJS On-Ramp customerRepository Service

```
app.factory("studentRepository", function () {
  var factoryInstance = {};
  var students = [
    {
      name: "John Doh",
      classroom: 602,
      grade: 93
    },
    {
      name: "Steve Smith",
      classroom: 513,
      grade: 72
    },
    .
    .
    .
  ];

  factory.getStudents = function () {
    return students;
  }

  return factoryInstance;
});
```

As I mentioned before, we like to keep our business logic separate from our controller logic, which should mainly be the glue between our views and models. AngularJS makes it easy to inject services into our controllers, as well as other services.

There are a few ways to create services in AngularJS (namely, factories, services, and providers. But for this intro, we're going to focus on the most common way, which is to create a factory.

Although in our simple example, we're only using our students data source in one controller, we can clearly see the need for using it in several places within an app, across different controllers. So we're going to create a customer repository service.

We'll be calling the factory method on our app module to define a factory we'll be using for our studentRepository service. The first parameter is a string holding the name of our factory, and the second defines the object (or JavaScript function) representing our factory.

Within this object, we're hard-coding our student repository. Again, in a real app, we'd be retrieving this from a web or REST service, but we'd still be storing it in a students variable, which we'll be making use of in the methods we define for our

AngularJS On-Ramp
Modified Controller

```
app.controller("StudentController",
    function ($scope, $routeParams, studentRepository) {

  $scope.students = studentRepository.getStudents();

  $scope.displayStudentInfo = function(student) {
    alert(student.name);
  }

  if ($routeParams.classroom) $scope.classroom = $routeParams.classroom;
});
```

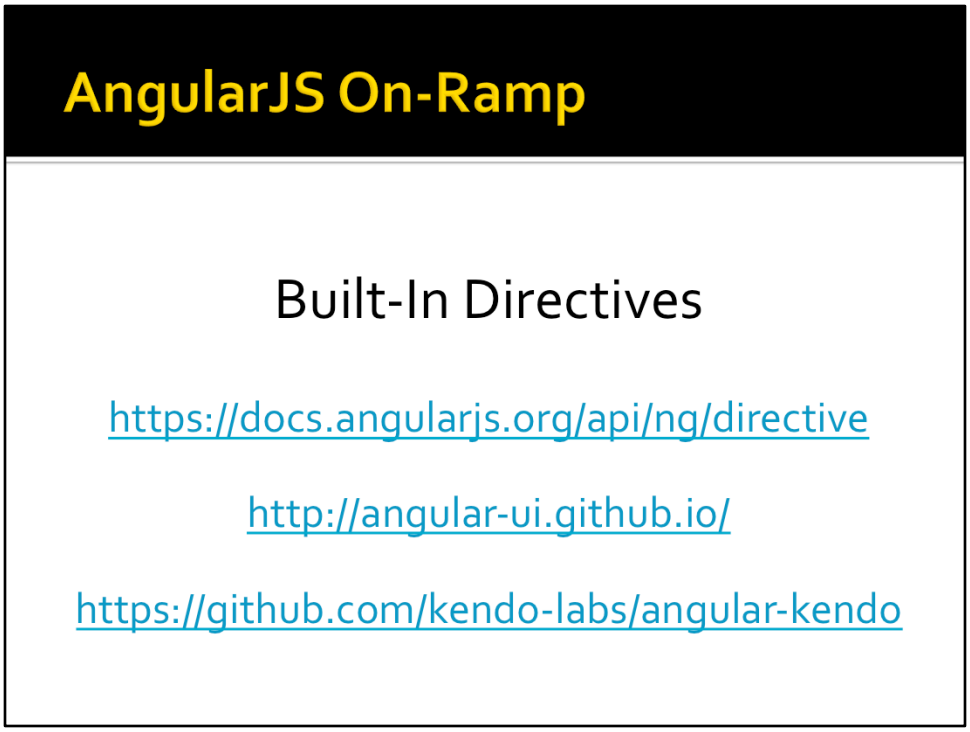
In our modified controller, we are now injecting our service factory, studentRepository. We can pass all injected objects in any order we'd like, but the best practice is to pass in built-in AngularJS objects first, followed by your own.

Next, instead of retrieving the students directly (or, in our case, hard-coding them), we're making use of our injected service. You may see the benefit here of Angular's DI. Where in a real application, you may be passing in a service which retrieves this data from a server, when you want to unit test, you can pass in a mock version of your service. Our version of the service is quite mock-like in our example, of course.

In our revised HTML page, we're just pulling in the new version of our JavaScript file.

That's it.

When we run the app, we can see we're getting the same results as before. But now our services are loosely coupled from our controllers.



We've seen several built-in AngularJS directives so far, such as ngApp, ngModel, ngClick, and ngRepeat. AngularJS contains a ton of these directives, and you can go through the list on their site.

As a matter of fact, before you decide on writing your own directives, take a close look at this list to see if something is already there for you to use.

In addition, there are several independently created directives available from the AngularUII site, including pre-built directives wrapping Bootstrap UI controls.

Not only that, but if you're a user of Kendo UI, there's an AngularJS directive library for that as well.

If you can avoid reinventing the wheel, do so.

But if what you want is not available yet, you'll have plenty opportunity to create your own…

indexWithDirective.html

Directives is the most powerful feature of AngularJS. It is the "killer" feature.

When you create custom directives, what you are essentially doing is expanding HTML and the DOM. This is what makes your HTML self-describing.

But it's the most difficult part of it as well. If you want to dive into directives – and you do – you may want to avoid doing what I did. One of my first attempts involved creating nested directives. I threw myself into the deep end, and at times felt like I was drowning. I still don't grok it, but it forced me to learn a lot rather quickly.

But let's start with a relatively simple example, just to get a taste of what you can do.

AngularJS On-Ramp
Directive Declaration

```
app.directive("collapsible", function () {
    return {
        restrict: "E",
        template: "<div data-ng-click='toggleVisibility()'>" +
                    "<span data-ng-show='visible'>Hide</span>" +
                    "<span data-ng-hide='visible'>Show</span> {{label}}" +
                    "<div data-ng-show='visible' data-ng-transclude>{{data}}</div>" +
                  "</div>",
        replace: true,
        scope: {
            data: "@",
            label: "@"
        },
        controller: function ($scope) {
            $scope.visible = true;

            $scope.toggleVisibility = function () {
                $scope.visible = !$scope.visible;
            };
        },
        transclude: true
    };
});
```

We're going to create a new directive by calling the directive method of our module. We're calling it "collapsible."

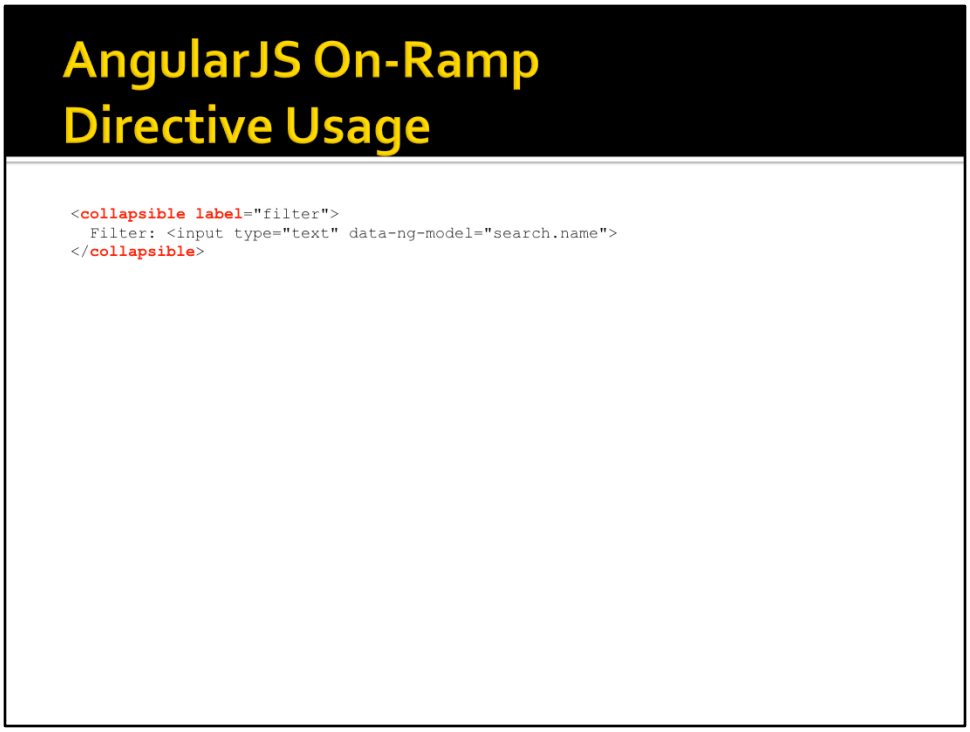
We create a directive by returning an object with all the settings we need for its functionality.

The "restrict" option lets us specify where we want the directive to be usable. This is an "element" directive. In other words, we're creating a new HTML element. We can combine directives. So if we used "AE", we'd be able to use it as a stand-alone element, or as an element's attribute (such as "<DIV collapsible></DIV>". The default is "A" – attribute.

The "template" option holds the HTML that will be generated by Angular when it sees this directive. Just like with routing, we can use "templateUrl" instead.

The "replace" option tells Angular to completely replace our directive with the templated HTML. Otherwise, the original directive will remain in the HTML, and since HTML ignores unknown elements, it should be fine.

By default, directives work on the current scope, using the current controller. If we want your directives to be reusable, and act as stand-alone components, we can use



We're going to make our student search filter collapsible.

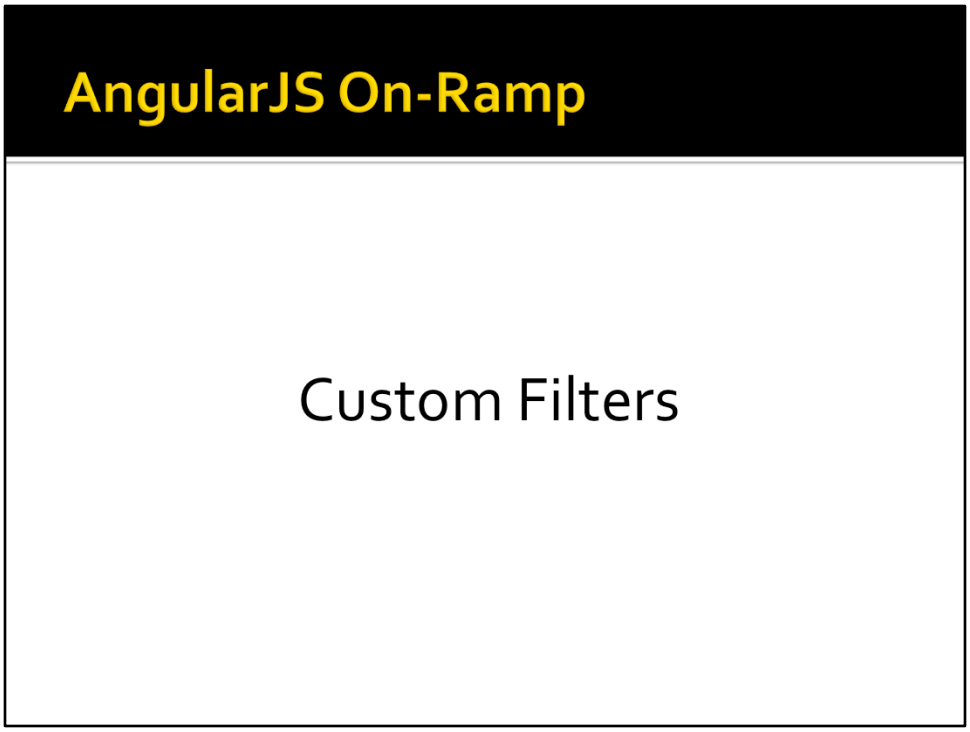
All we need to do is wrap our filtering HTML with the new "collapsible" element we just created. We created it once, and now we can just simply use it anywhere we need. All the click, hide, and show logic is handled inside the directive.

We just extended the HTML language, and anyone reading our HTML can clearly infer what this tag will do. If we did it the old fashioned way, with something like jQuery, we'd maybe see some <DIV> tags wrapping our filter, and would have to dig into our JavaScript to see how we may be modifying it, if at all. The only hint may have been an extra class given to the DIV. Here, our intentions are quite clear.

AngularJS On-Ramp
Custom Directives

E: <my-directive></my-directive>
A: <div my-directive>
C:
M: <!-- directive: my-directive -->

As mentioned, there are four types of directives: (E)lement, (A)ttribute, (C)lass, and Co(M)ment. I bet you can guess what each means.



indexWithFilter.html

We've seen some of AngularJS's built-in filters. You can create your own filters as well.

We'll build a simple example that spaces out text.

AngularJS On-Ramp
Custom "spacer" Filter

```
app.filter("spacer", function() {
  return function(text) {
    text = text.split("").join(" ");
    return text;
  };
});
```

Like we did with directives, we just call a method, "filter" on our module.

Filters return a function that handles the filtering. This function takes a parameter of the values to apply the filter to; in this case "text".

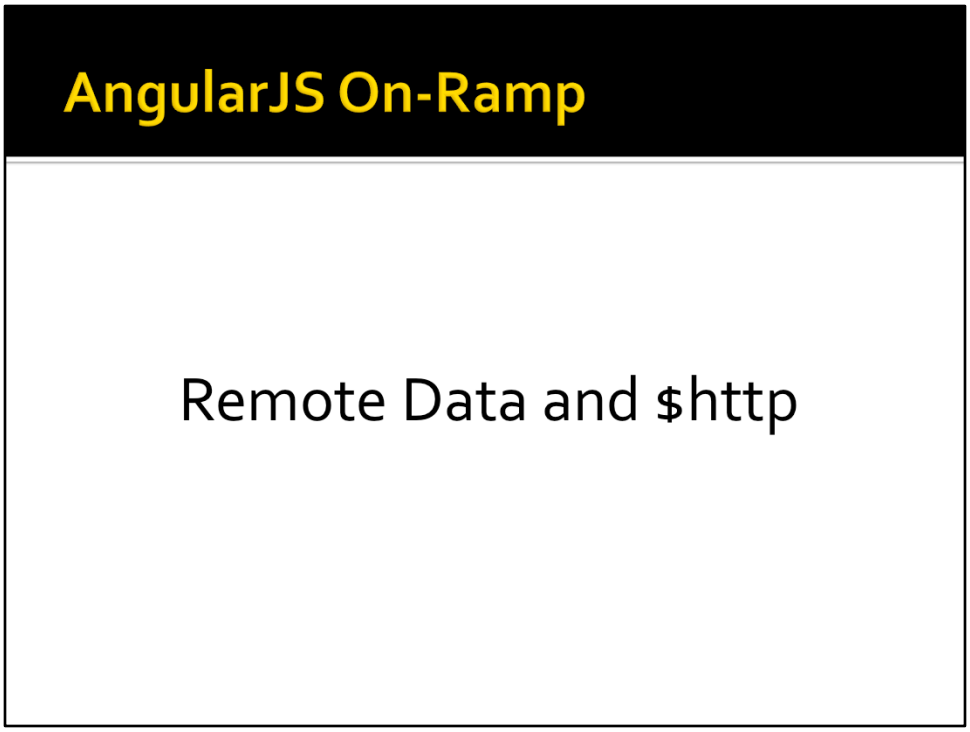
The function returns the result of the transformation.

Filters can take parameters, like the built-in "orderBy" filter, by passing them into the outer function call. We don't need any here.

AngularJS On-Ramp
Using the "spacer" Filter

```
<ul>
  <li data-ng-click="displayStudentInfo(student)"
      data-ng-repeat="student in students |
                      filter:search | orderBy:'grade':true">
    {{student.name | spacer}} is in classroom
    {{student.classroom}}, and earned a grade of
    {{student.grade}}.
  </li>
</ul>
```

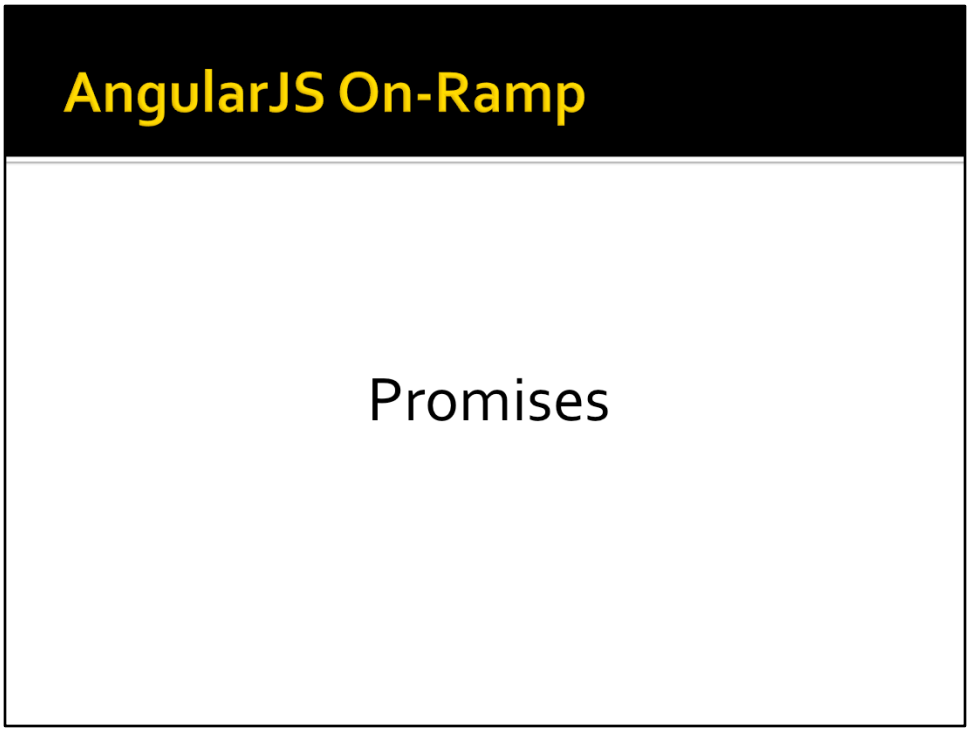
Then we just use it like any built-in AngularJS filter. That's it.



indexWithHttp.html

Up to this point, we've only grabbed our data from hard-coded objects. Normally, though, we'd be accessing our data over the Web, either via web services, REST, OData, and related calls.

AngularJS has a built-in http service (called, $http – the dollar sign prefix again). This is similar to jQuery's built-in remote access functionality.



For the most part, accessing remote data via AngularJS is handled asynchronously, making use of “promises” to handle responses. It also makes it easy for us to chain several promises together, but that’s a topic for a more advanced session

Let’s get rid of our hard-coded data, and at least place our data in JSON files in order to access it via $http.

AngularJS On-Ramp
Using $http

```
app.factory("studentRepository", function ($http) {
  var factory = {};

  factory.getStudents = function () {
     return $http.get("http://localhost/students.json");
  }

  return factory;
});
```

In our studentRepository service, we had hard-coded our students data source. But in this version, we are first injecting the $http service to our service.

What this will get us is a promise, which we'll have to act upon afterwards.

There are several ways you can make use of $http, but calling its "get" method is the simplest, and all we really need to load all of the students from our repository.

AngularJS On-Ramp
Handling the Promise

```
app.controller("StudentController", function($scope, $routeParams, studentRepository) {
  studentRepository.getStudents().then(function (result) {
    $scope.students = result.data;
  });

  $scope.displayStudentInfo = function(student) {
    alert(student.name);
  }

  if ($routeParams.classroom) $scope.classroom = $routeParams.classroom;
});
```

As I mentioned, when we call getStudents, what we get back is a promise. When the data returns to us, we handle the response through the promise's "then" function.

We grab the actual data (in this case, the JSON object) from the "data" attribute of the result.

And that's basically it. Like before, we're assigning it to "students" on our scope, and AngularJS automatically refreshes the UI using two-way model binding as we've seen before.

Overall, we've written maybe 20% of the JavaScript code we'd write before AngularJS. We've taken advantage of Angular's declarative nature to decorate our HTML, and have left the custom imperative business logic mainly in our services, which we've injected into our controllers.

AngularJS On-Ramp

Learning Resources

- AngularJS:
  https://angularjs.org/
- How do I think in AngularJS…:
  http://stackoverflow.com/questions/14994391/how-do-i-think-in-angularjs-if-i-have-a-jquery-background
- Create an Angular App in Seconds with Hot Towel:
  http://www.johnpapa.net/hot-towel-angular/
- Angular App Structuring Guideline:
  http://www.johnpapa.net/angular-app-structuring-guidelines/
- AngularJS - ng-conf 2014:
  http://ng-conf.ng-learn.org/
- AngularJS on Google+:
  https://plus.google.com/+AngularJS/posts