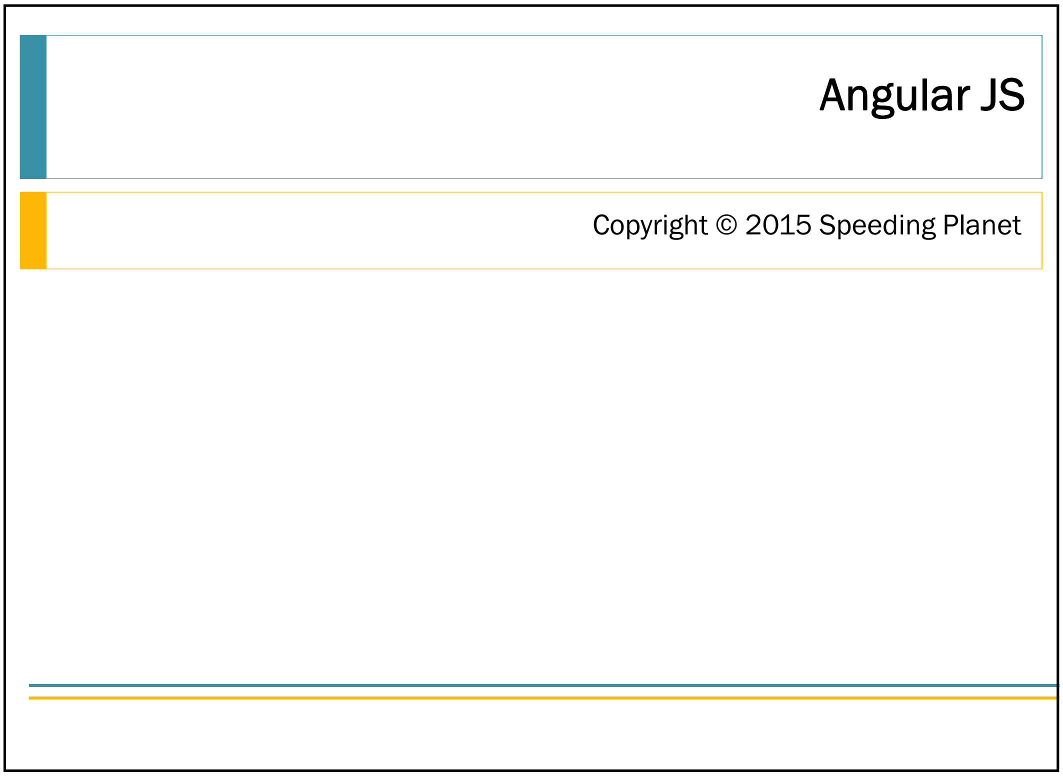
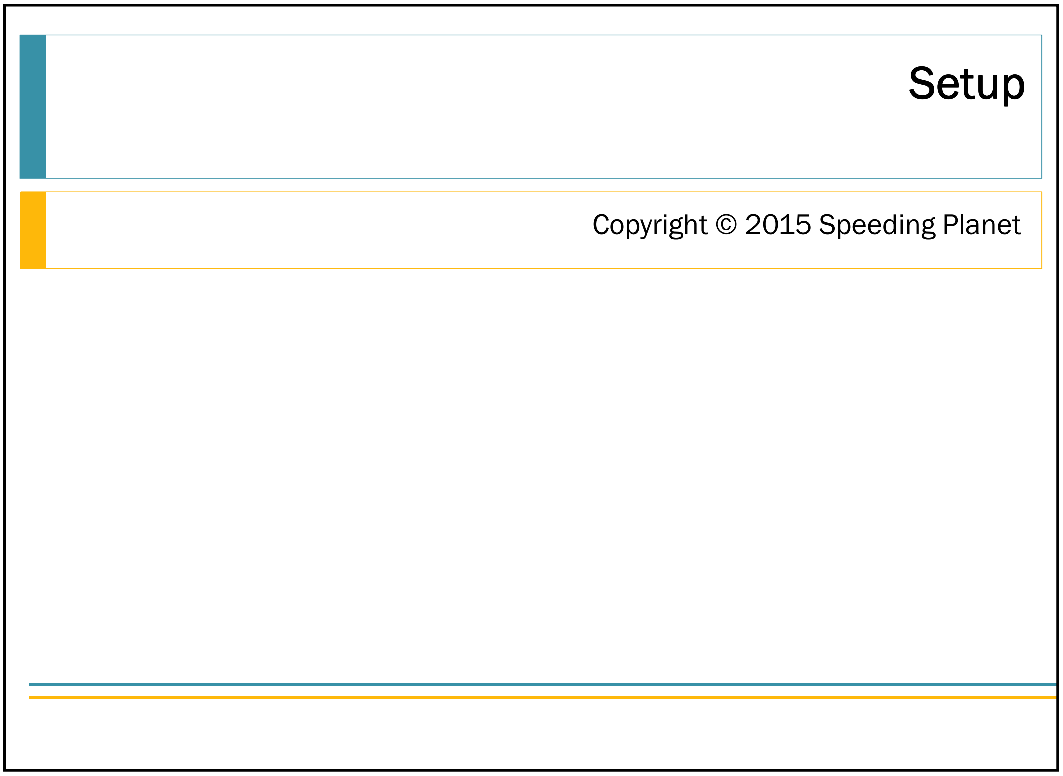
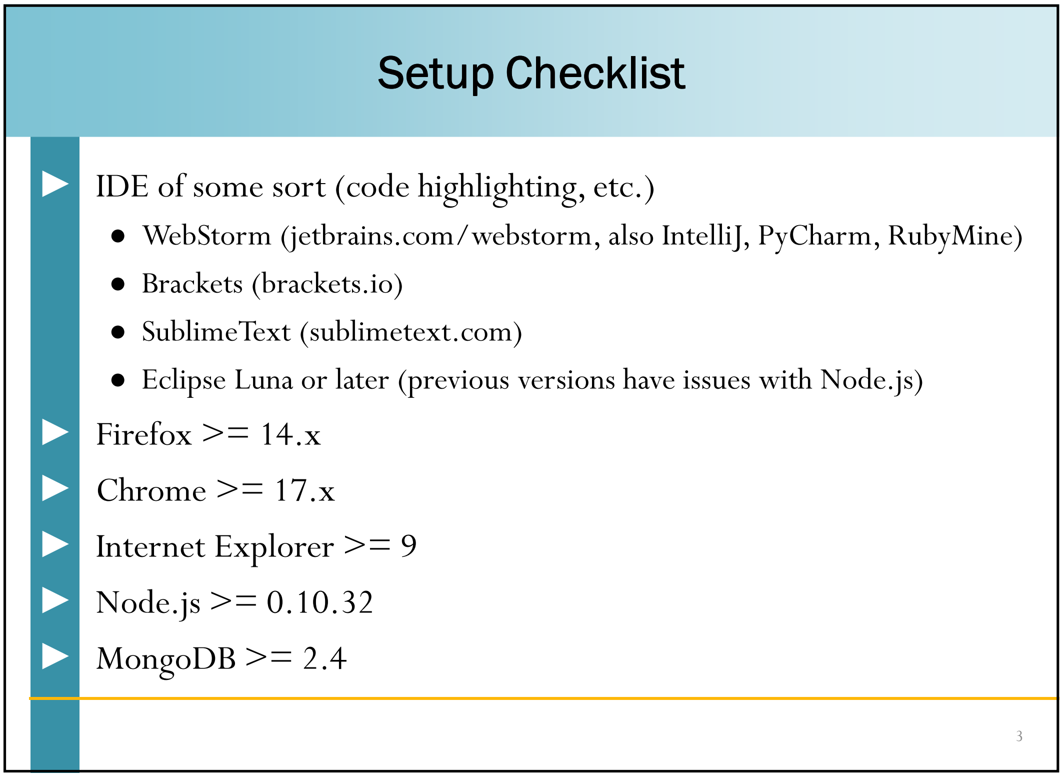


Angular JS

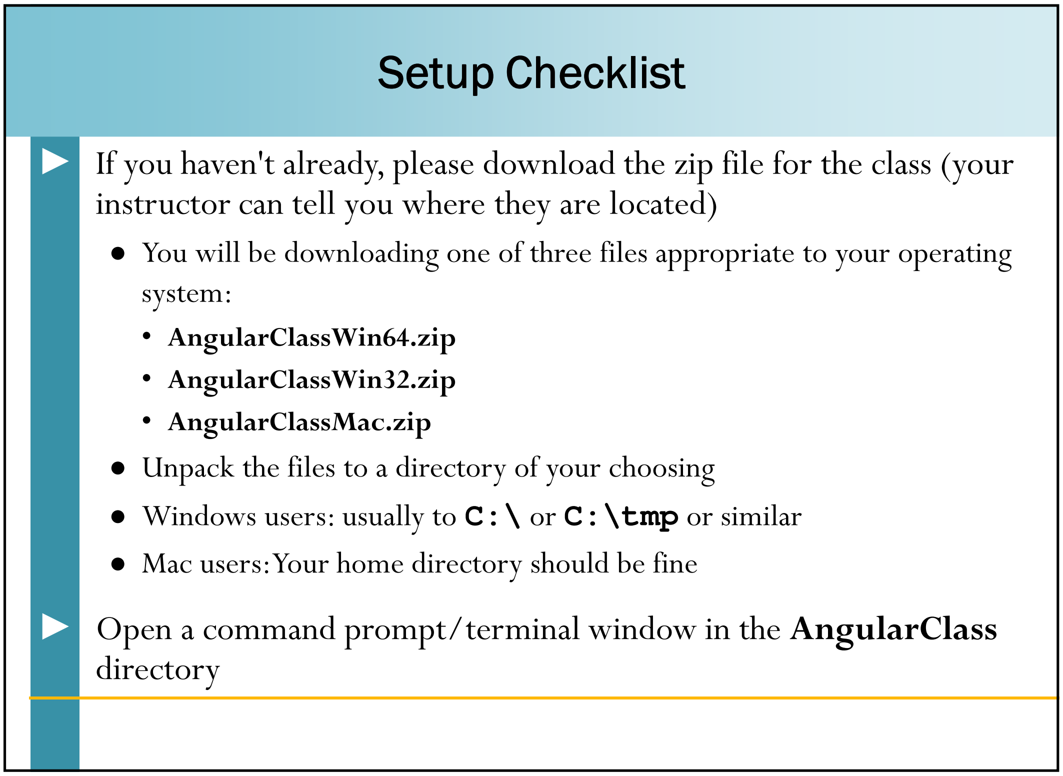


Setup



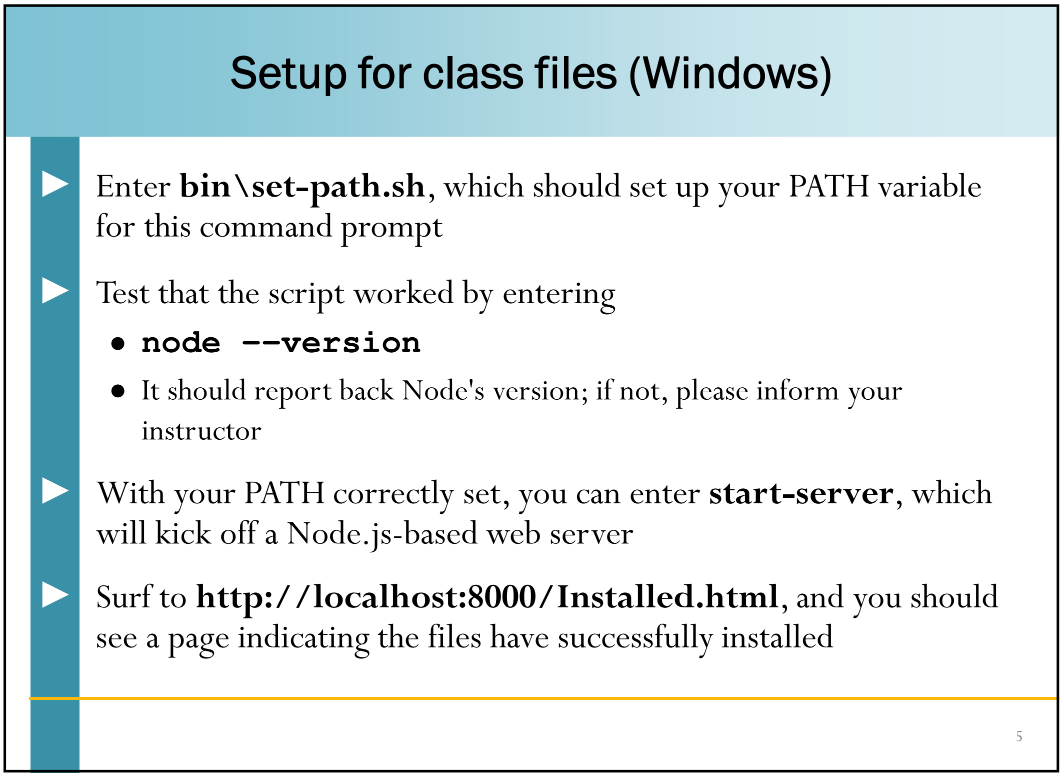
Setup Checklist

- IDE of some sort (code highlighting, etc.)
  - WebStorm (jetbrains.com/webstorm, also IntelliJ, PyCharm, RubyMine)
  - Brackets (brackets.io)
  - SublimeText (sublimetext.com)
  - Eclipse Luna or later (previous versions have issues with Node.js)
- Firefox >= 14.x
- Chrome >= 17.x
- Internet Explorer >= 9
- Node.js >= 0.10.32
- MongoDB >= 2.4



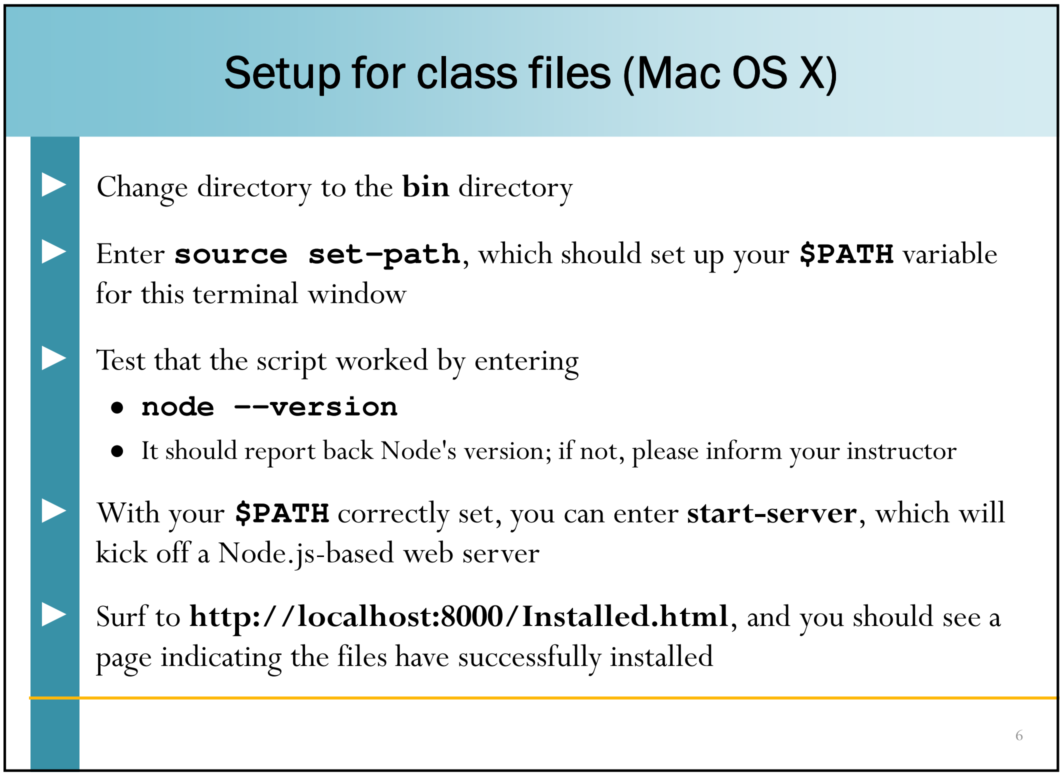
Setup Checklist

- If you haven't already, please download the zip file for the class (your instructor can tell you where they are located)
  - You will be downloading one of three files appropriate to your operating system:
    - **AngularClassWin64.zip**
    - **AngularClassWin32.zip**
    - **AngularClassMac.zip**
  - Unpack the files to a directory of your choosing
  - Windows users: usually to **`C:\`** or **`C:\tmp`** or similar
  - Mac users: Your home directory should be fine
- Open a command prompt/terminal window in the **AngularClass** directory



Setup for class files (Windows)

- Enter **bin\set-path.sh**, which should set up your PATH variable for this command prompt
- Test that the script worked by entering
  - **`node --version`**
  - It should report back Node's version; if not, please inform your instructor
- With your PATH correctly set, you can enter **start-server**, which will kick off a Node.js-based web server
- Surf to **http://localhost:8000/Installed.html**, and you should see a page indicating the files have successfully installed



Setup for class files (Mac OS X)

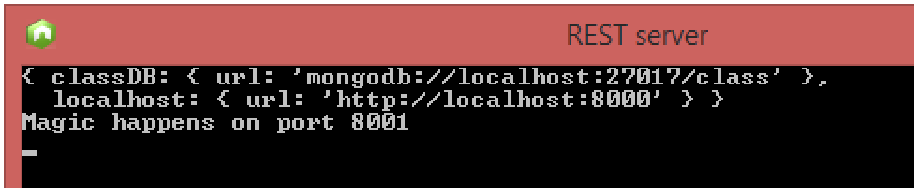
- Change directory to the **bin** directory
- Enter **`source set-path`**, which should set up your **`$PATH`** variable for this terminal window
- Test that the script worked by entering
  - **`node --version`**
  - It should report back Node's version; if not, please inform your instructor
- With your **`$PATH`** correctly set, you can enter **start-server**, which will kick off a Node.js-based web server
- Surf to **http://localhost:8000/Installed.html**, and you should see a page indicating the files have successfully installed

Database setup

- Test that your PATH has been correctly set by entering **`mongo --version`**
  - This should report back `MongoDB shell version: 2.6.7` or later
  - Or something similar, depending on the version
- Enter **`mongo-start`**, which will kick off an instance of the MongoDB database
- Enter **`mongo-load-class`**, which will load the class data into the Mongo database

REST setup scripts

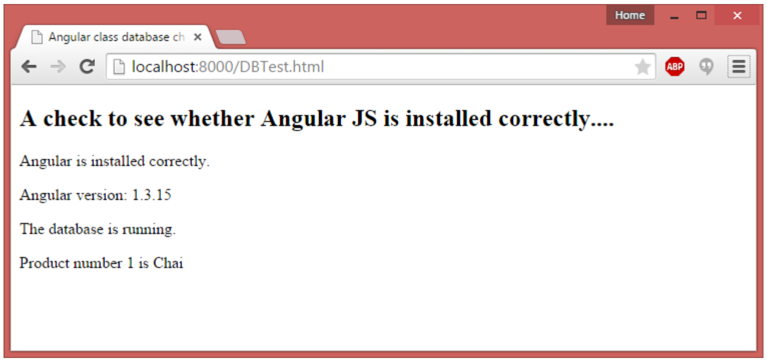
- Enter **start-rest**, which will kick off a RESTful interface to the Mongo database
- You should see a message like this:



```
{ classDB: { url: 'mongodb://localhost:27017/class' },
  localhost: { url: 'http://localhost:8000' } }
Magic happens on port 8001
_
```

RESTful check

- Surf to **http://localhost:8000/DBTest.html**
- You should see a page indicating that the web server and the RESTful server are both running



A check to see whether Angular JS is installed correctly....

Angular is installed correctly.

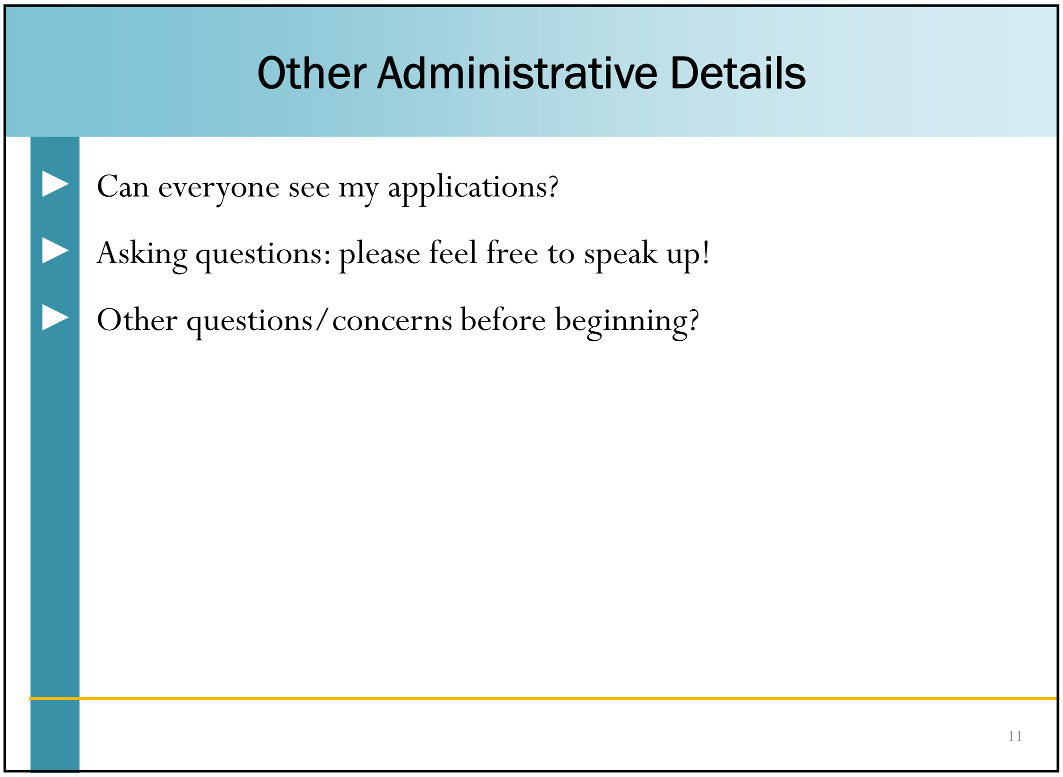
Angular version: 1.3.15

The database is running.

Product number 1 is Chai

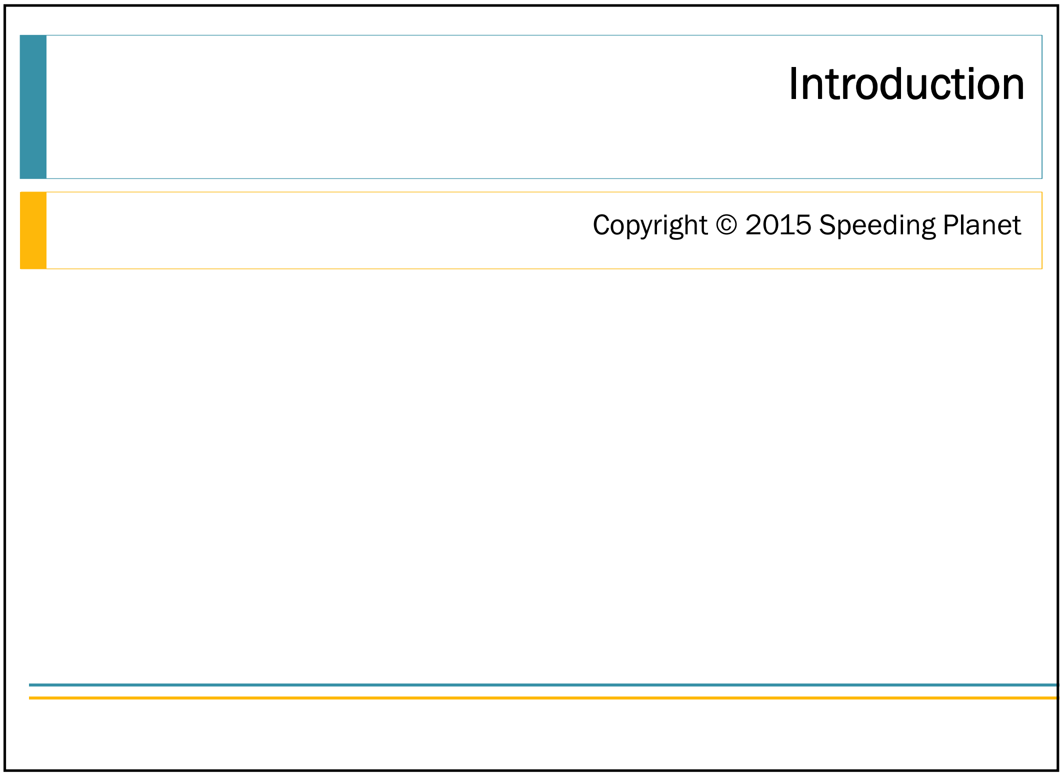
Assumptions

- You are familiar with basic HTML and CSS
  - HTML file structure, spans, divs, tables, etc.
  - CSS in-line styles, classes, in-document styles, linking separate CSS files
- You are familiar with JavaScript, particularly with functions and object literals
  - Of course, the basics of the language, variables, control-structures and so on help as well!
- You are conversant with the Model-View-Controller (MVC) design pattern

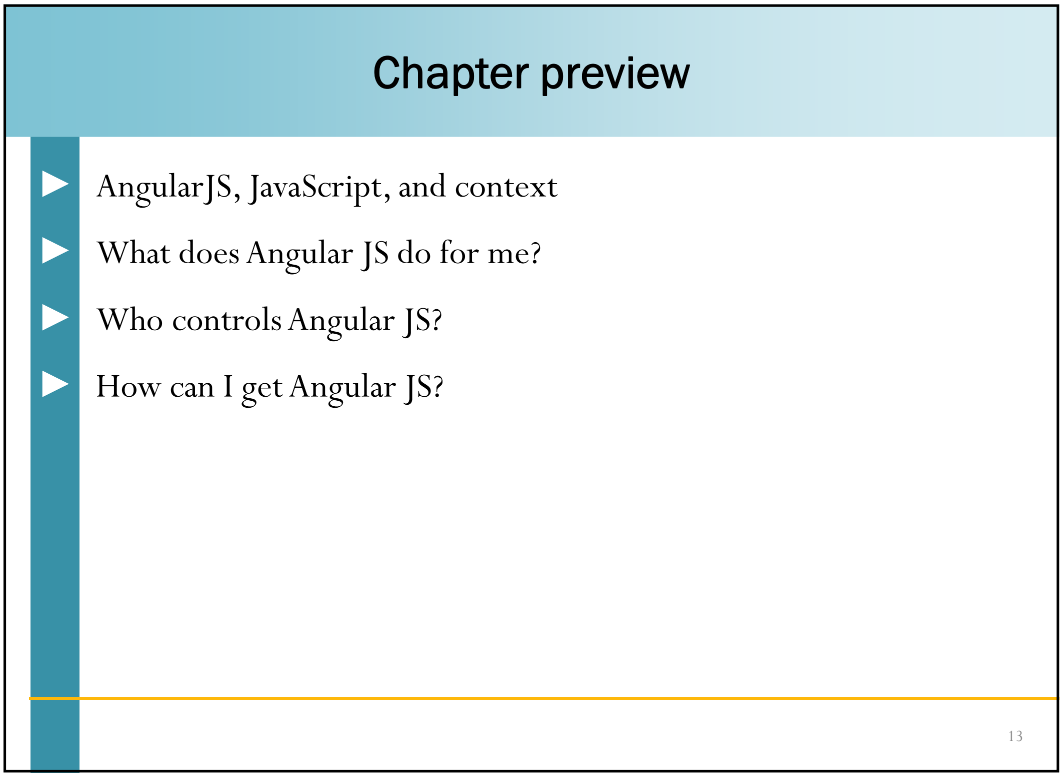


Other Administrative Details

- Can everyone see my applications?
- Asking questions: please feel free to speak up!
- Other questions/concerns before beginning?



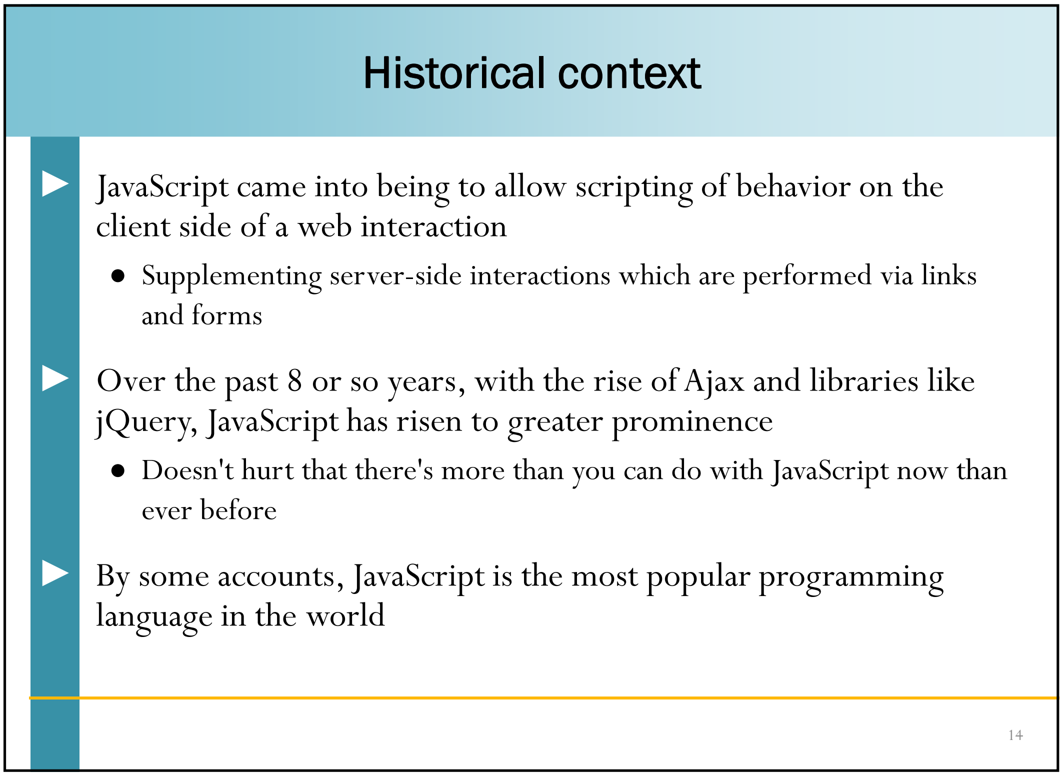
Introduction



Chapter preview

- AngularJS, JavaScript, and context
- What does Angular JS do for me?
- Who controls Angular JS?
- How can I get Angular JS?

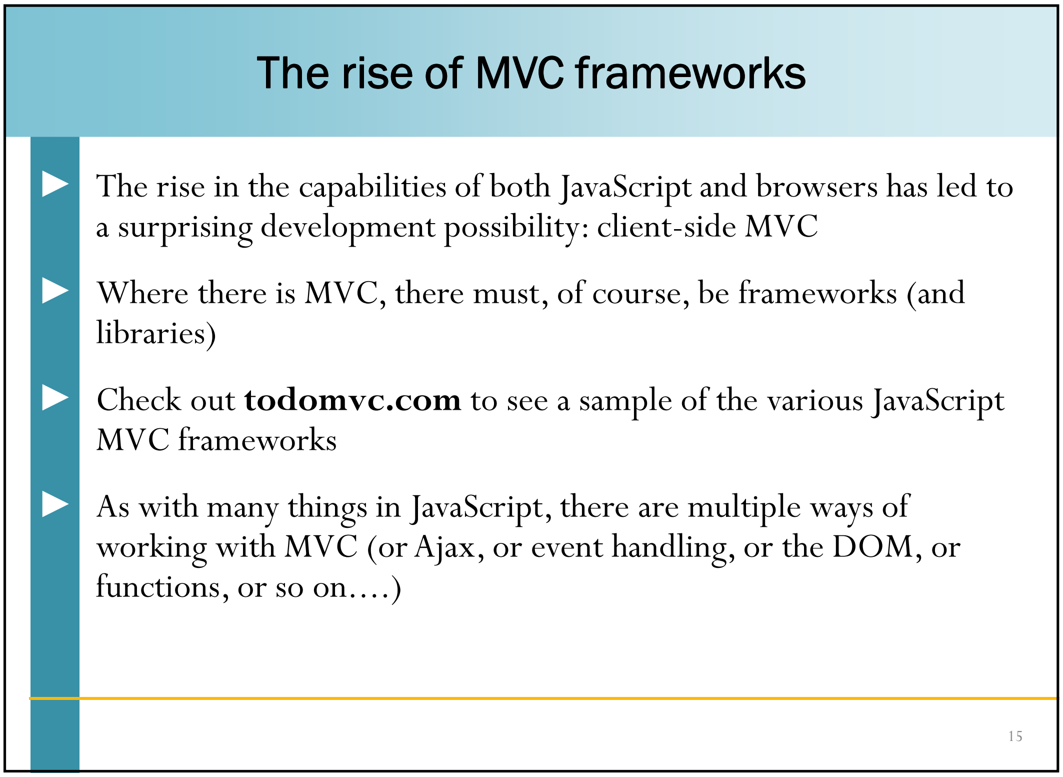
13



Historical context

- JavaScript came into being to allow scripting of behavior on the client side of a web interaction
  - Supplementing server-side interactions which are performed via links and forms
- Over the past 8 or so years, with the rise of Ajax and libraries like jQuery, JavaScript has risen to greater prominence
  - Doesn't hurt that there's more than you can do with JavaScript now than ever before
- By some accounts, JavaScript is the most popular programming language in the world

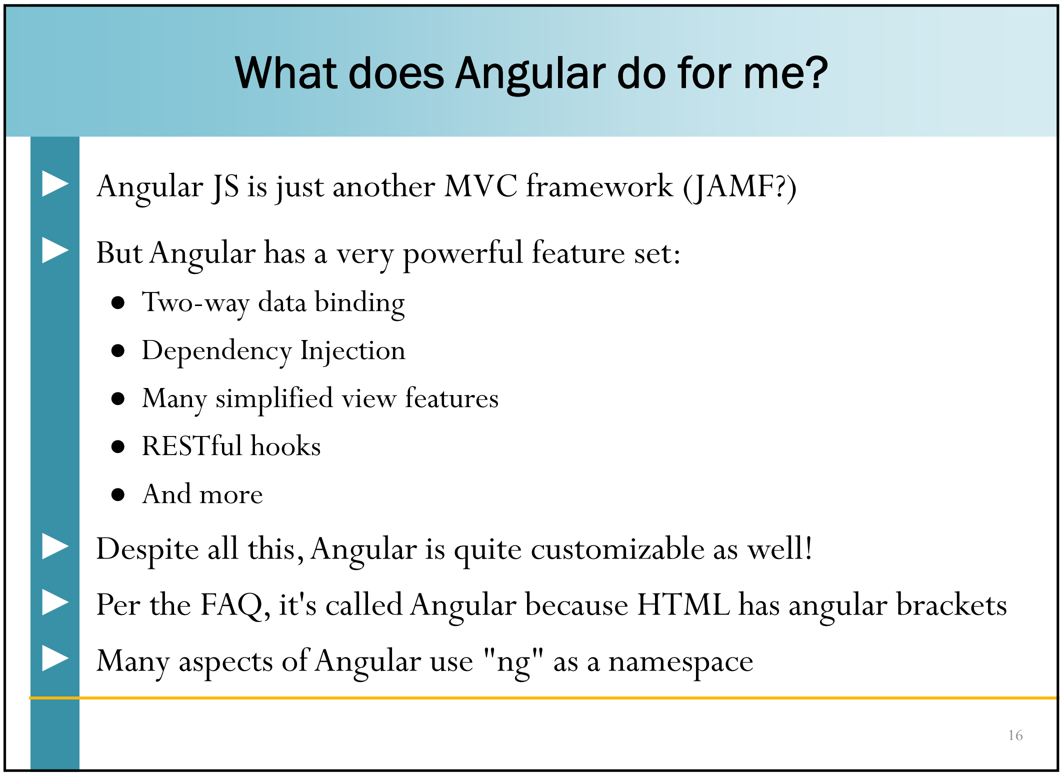
14



The rise of MVC frameworks

- The rise in the capabilities of both JavaScript and browsers has led to a surprising development possibility: client-side MVC
- Where there is MVC, there must, of course, be frameworks (and libraries)
- Check out **todomvc.com** to see a sample of the various JavaScript MVC frameworks
- As with many things in JavaScript, there are multiple ways of working with MVC (or Ajax, or event handling, or the DOM, or functions, or so on….)

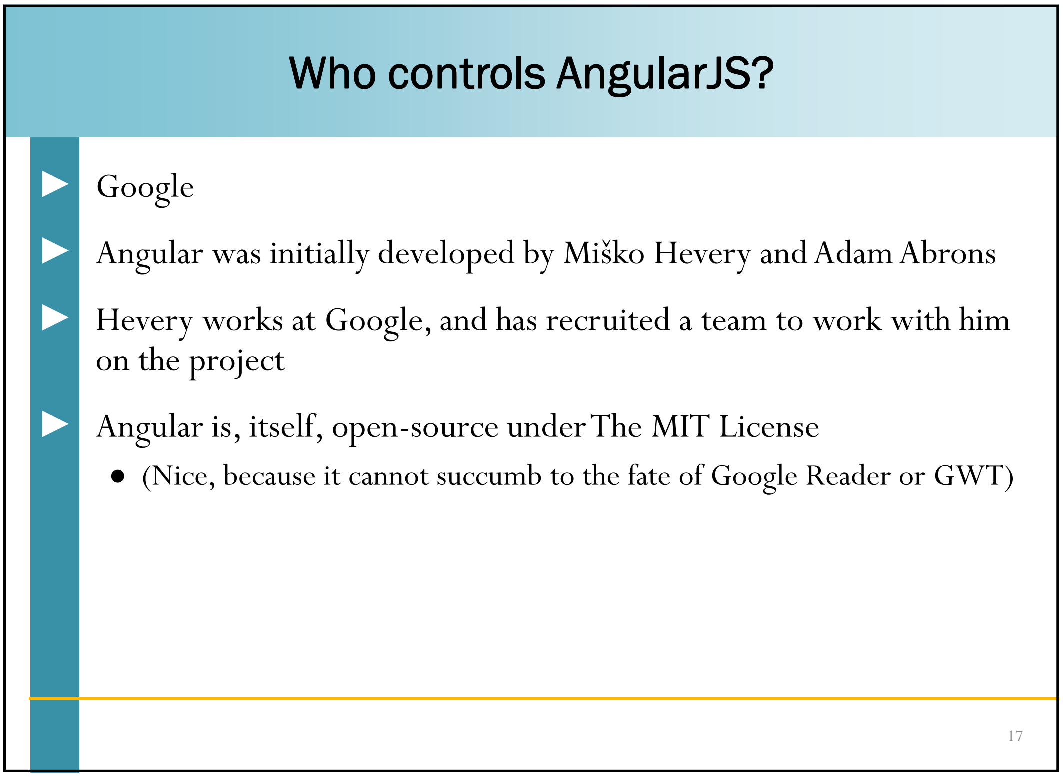
15



What does Angular do for me?

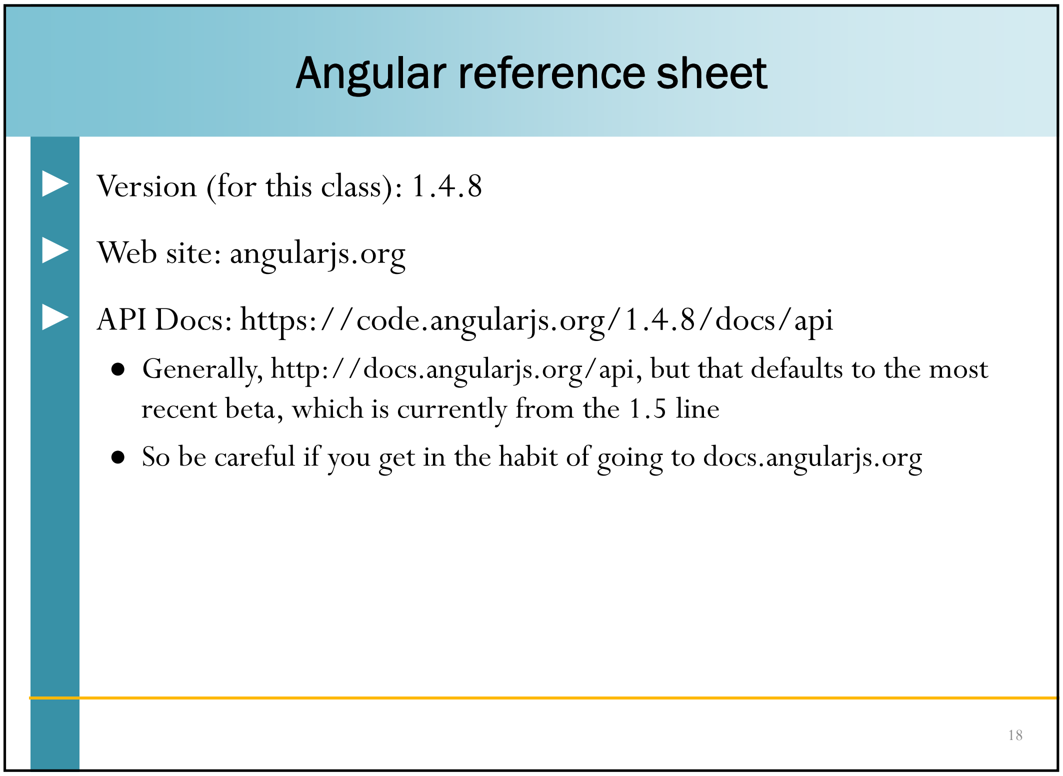
- Angular JS is just another MVC framework (JAMF?)
- But Angular has a very powerful feature set:
  - Two-way data binding
  - Dependency Injection
  - Many simplified view features
  - RESTful hooks
  - And more
- Despite all this, Angular is quite customizable as well!
- Per the FAQ, it's called Angular because HTML has angular brackets
- Many aspects of Angular use "ng" as a namespace

16



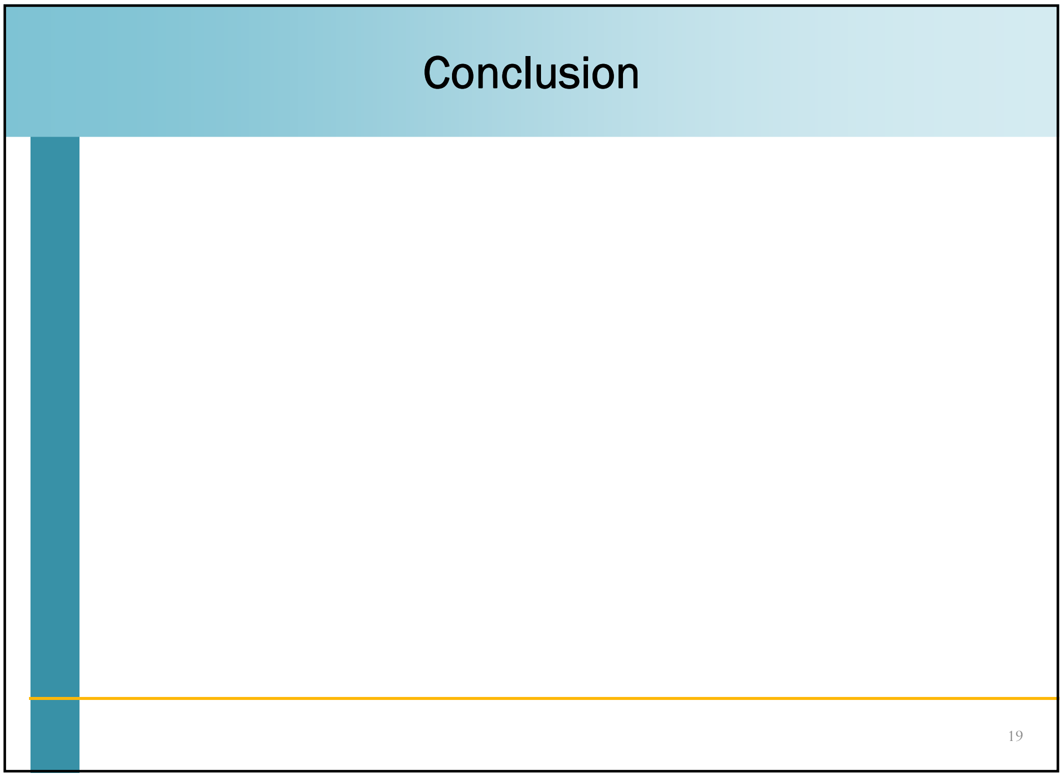
Who controls AngularJS?

- Google
- Angular was initially developed by Miško Hevery and Adam Abrons
- Hevery works at Google, and has recruited a team to work with him on the project
- Angular is, itself, open-source under The MIT License
  - (Nice, because it cannot succumb to the fate of Google Reader or GWT)

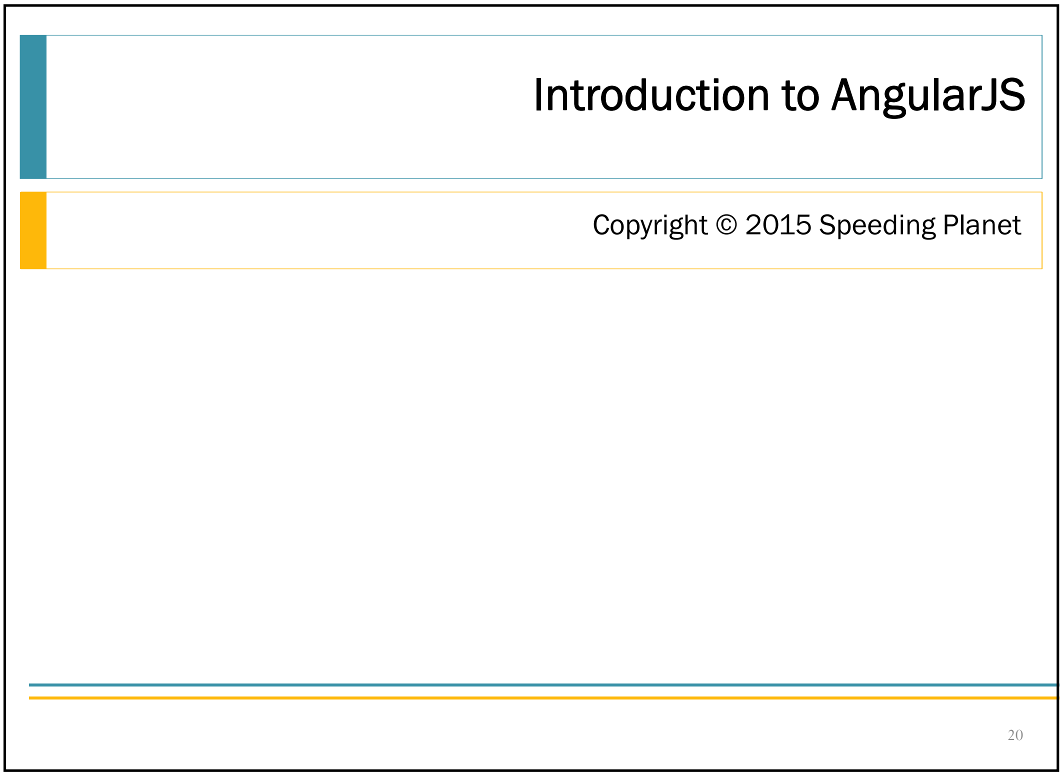


Angular reference sheet

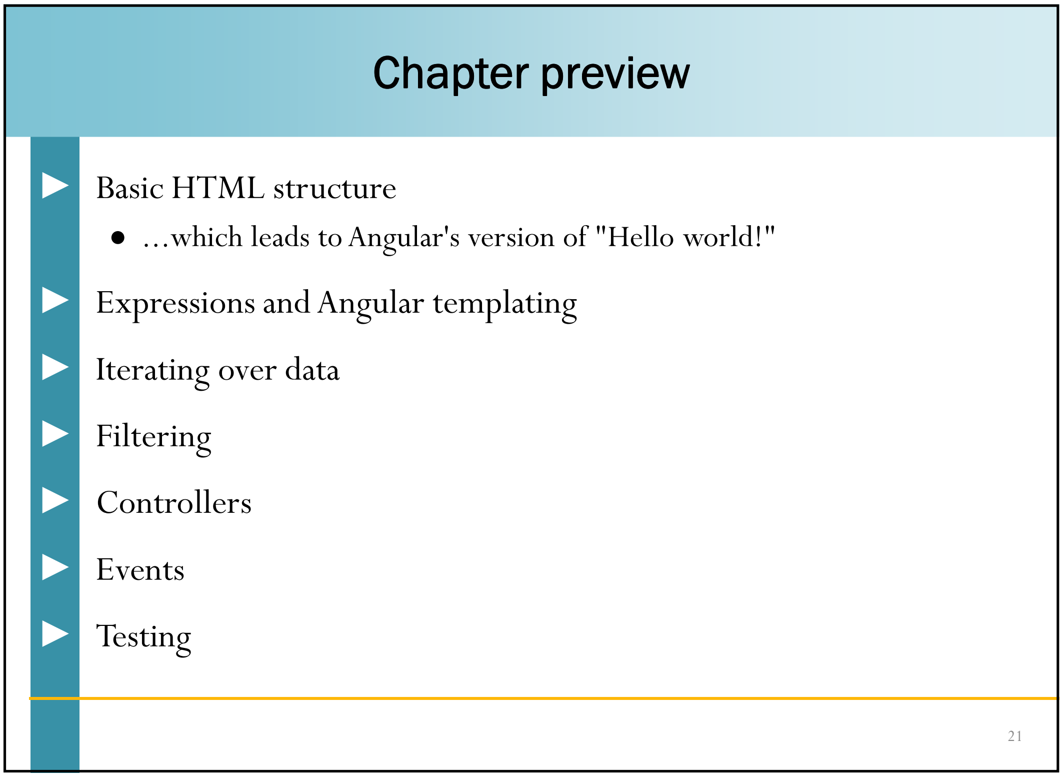
- Version (for this class): 1.4.8
- Web site: angularjs.org
- API Docs: https://code.angularjs.org/1.4.8/docs/api
  - Generally, http://docs.angularjs.org/api, but that defaults to the most recent beta, which is currently from the 1.5 line
  - So be careful if you get in the habit of going to docs.angularjs.org



Conclusion



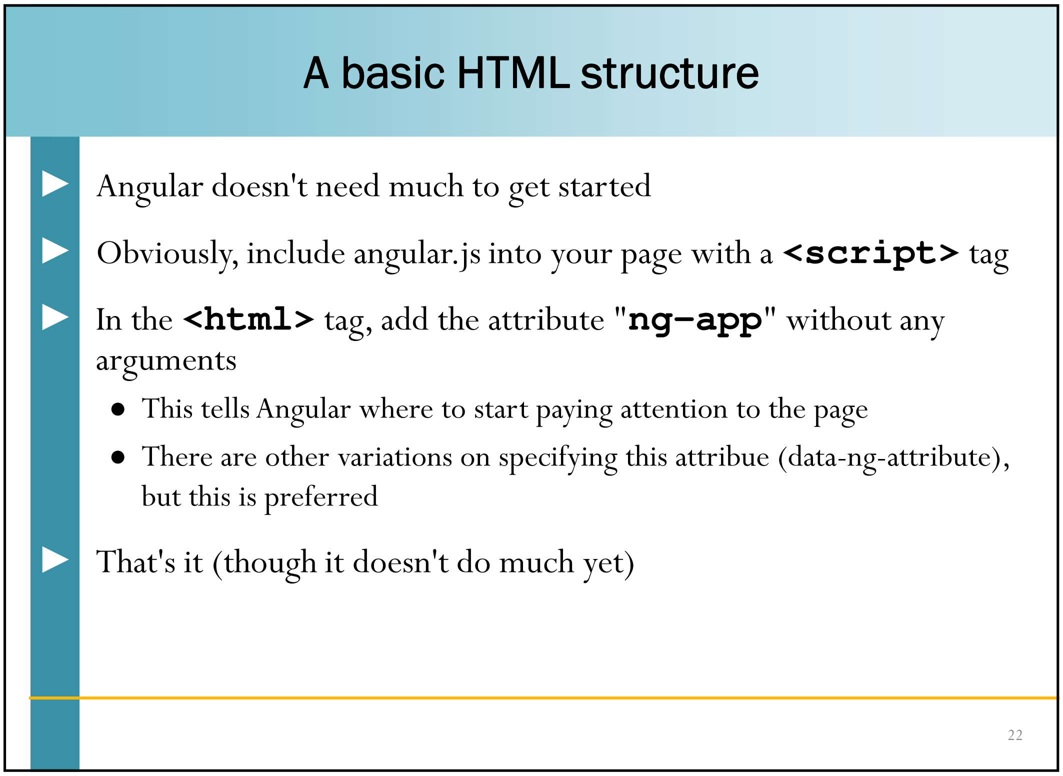
Introduction to AngularJS



Chapter preview

- Basic HTML structure
  - …which leads to Angular's version of "Hello world!"
- Expressions and Angular templating
- Iterating over data
- Filtering
- Controllers
- Events
- Testing

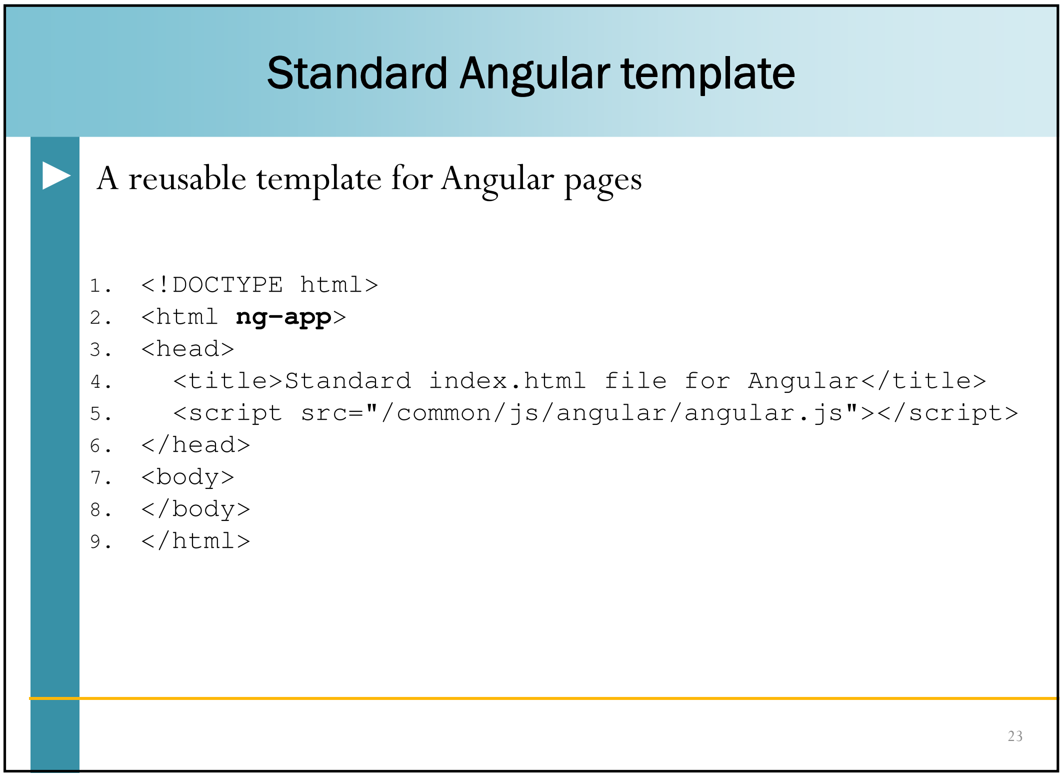
21



A basic HTML structure

- Angular doesn't need much to get started
- Obviously, include angular.js into your page with a **`<script>`** tag
- In the **`<html>`** tag, add the attribute "**`ng-app`**" without any arguments
  - This tells Angular where to start paying attention to the page
  - There are other variations on specifying this attribue (data-ng-attribute), but this is preferred
- That's it (though it doesn't do much yet)

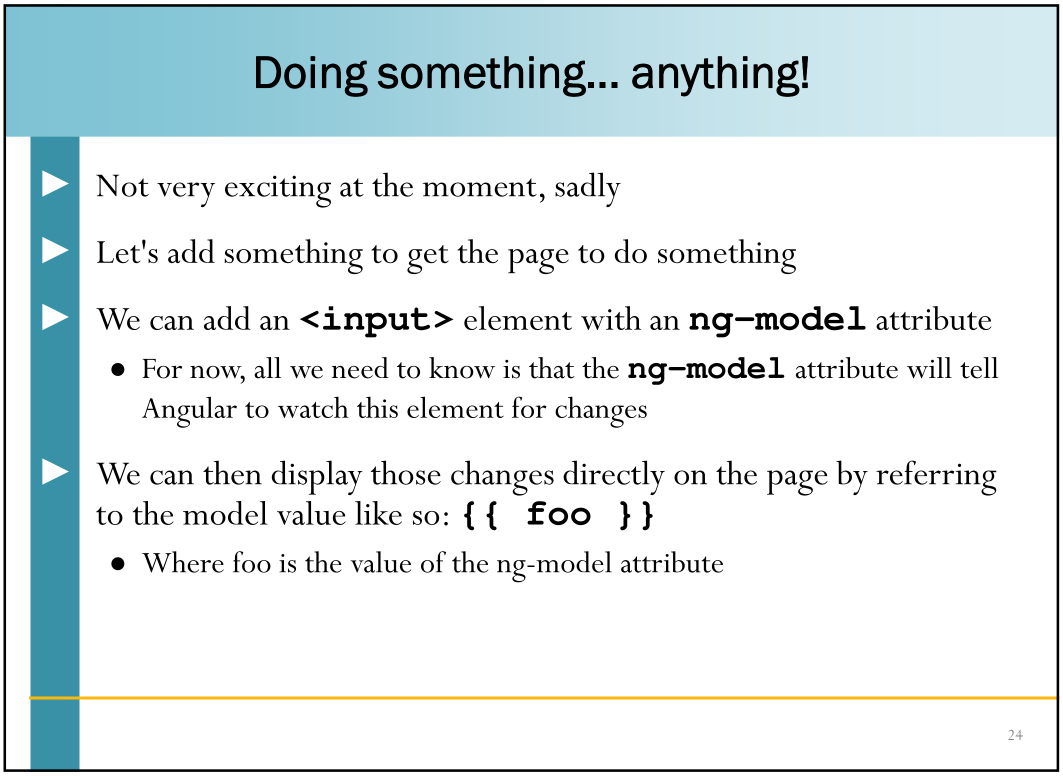
22



Standard Angular template

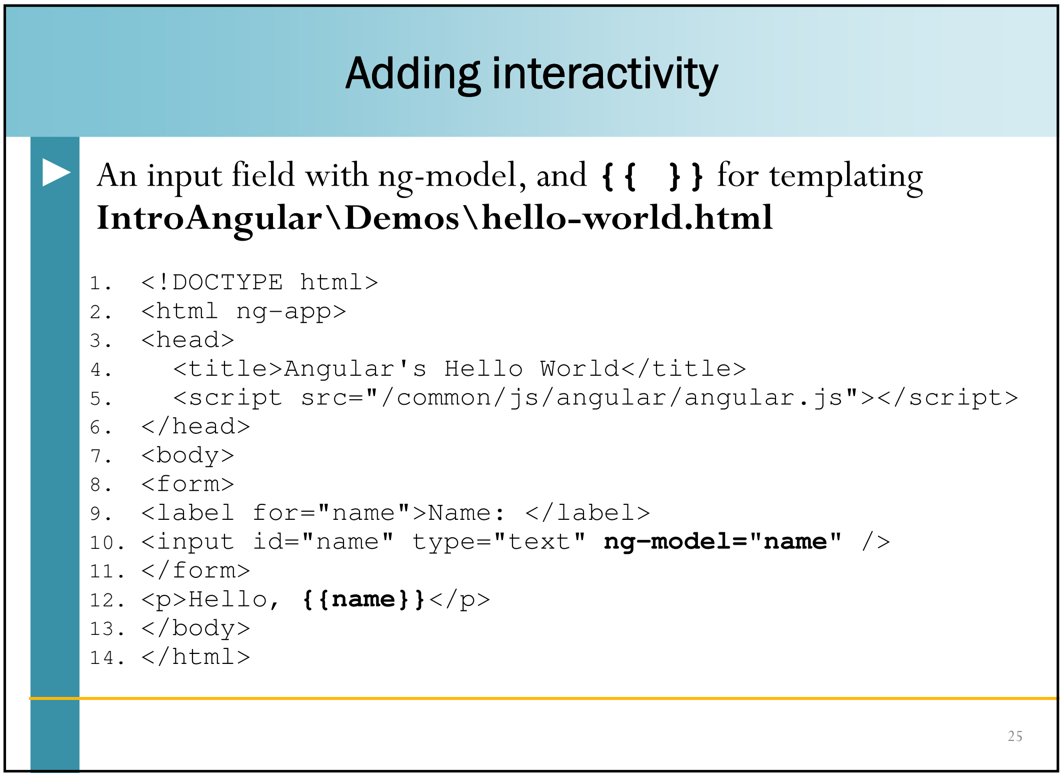
- A reusable template for Angular pages

```
<!DOCTYPE html>
<html ng-app>
<head>
  <title>Standard index.html file for Angular</title>
  <script src="/common/js/angular/angular.js"></script>
</head>
<body>
</body>
</html>
```



Doing something... anything!

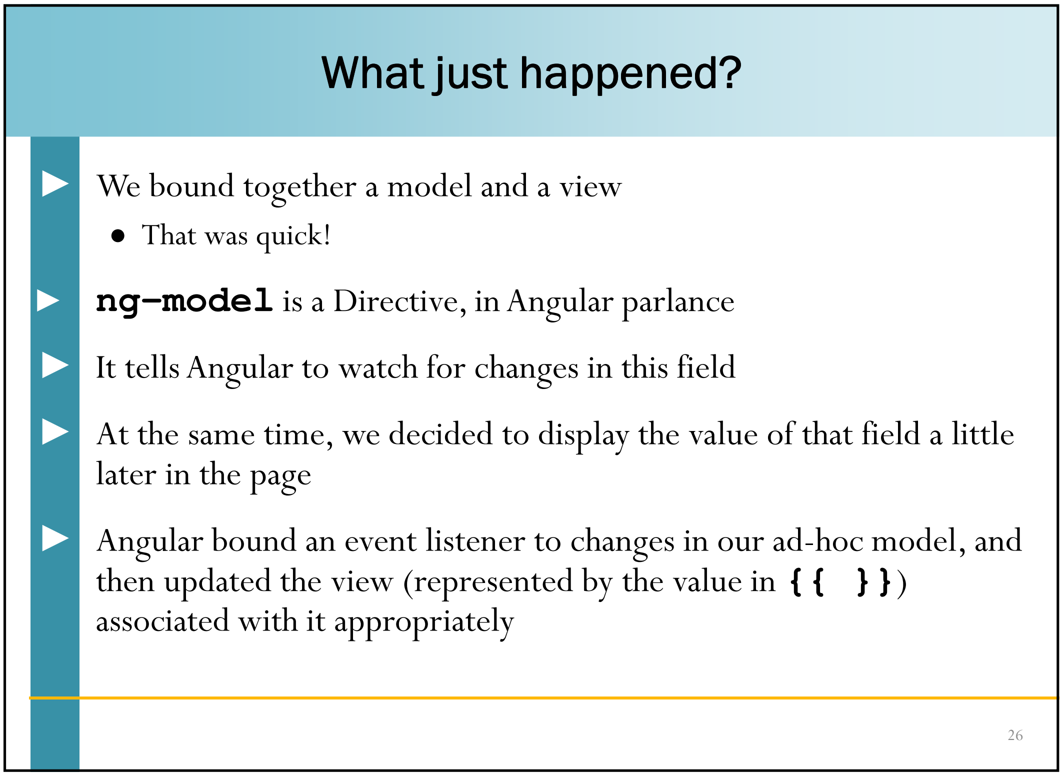
- Not very exciting at the moment, sadly
- Let's add something to get the page to do something
- We can add an **`<input>`** element with an **`ng-model`** attribute
  - For now, all we need to know is that the **`ng-model`** attribute will tell Angular to watch this element for changes
- We can then display those changes directly on the page by referring to the model value like so: **`{{ foo }}`**
  - Where foo is the value of the ng-model attribute



Adding interactivity

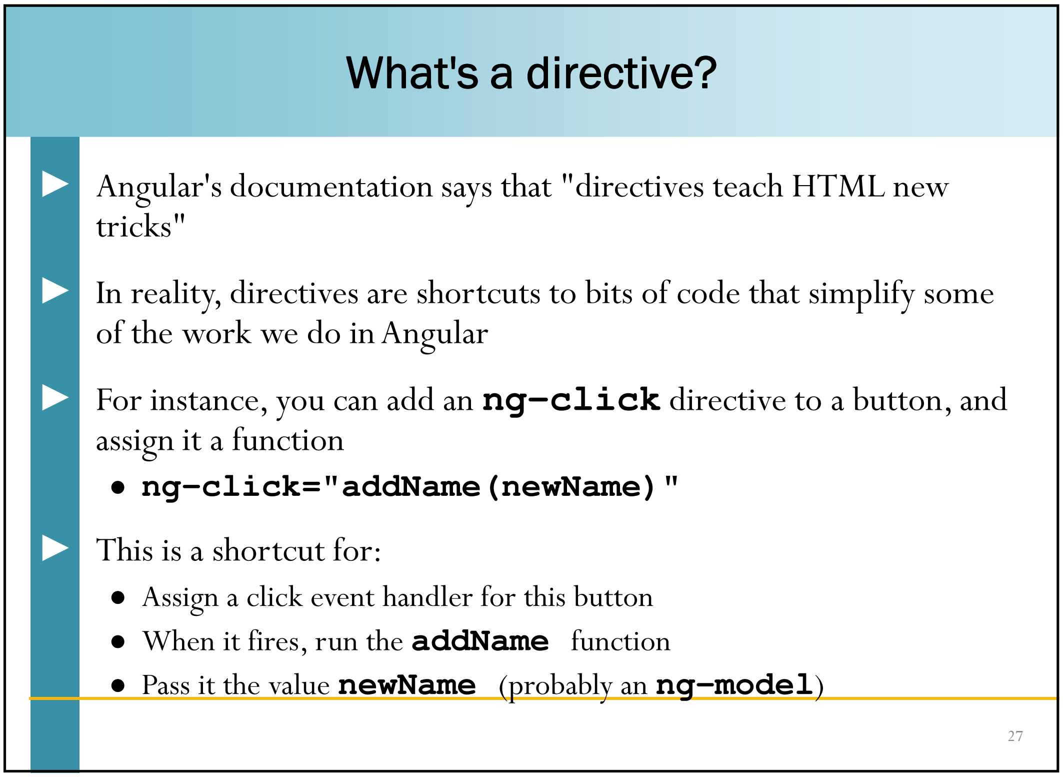
- An input field with ng-model, and **{{ }}** for templating **IntroAngular\Demos\hello-world.html**

```
<!DOCTYPE html>
<html ng-app>
<head>
  <title>Angular's Hello World</title>
  <script src="/common/js/angular/angular.js"></script>
</head>
<body>
<form>
<label for="name">Name: </label>
<input id="name" type="text" ng-model="name" />
</form>
<p>Hello, {{name}}</p>
</body>
</html>
```



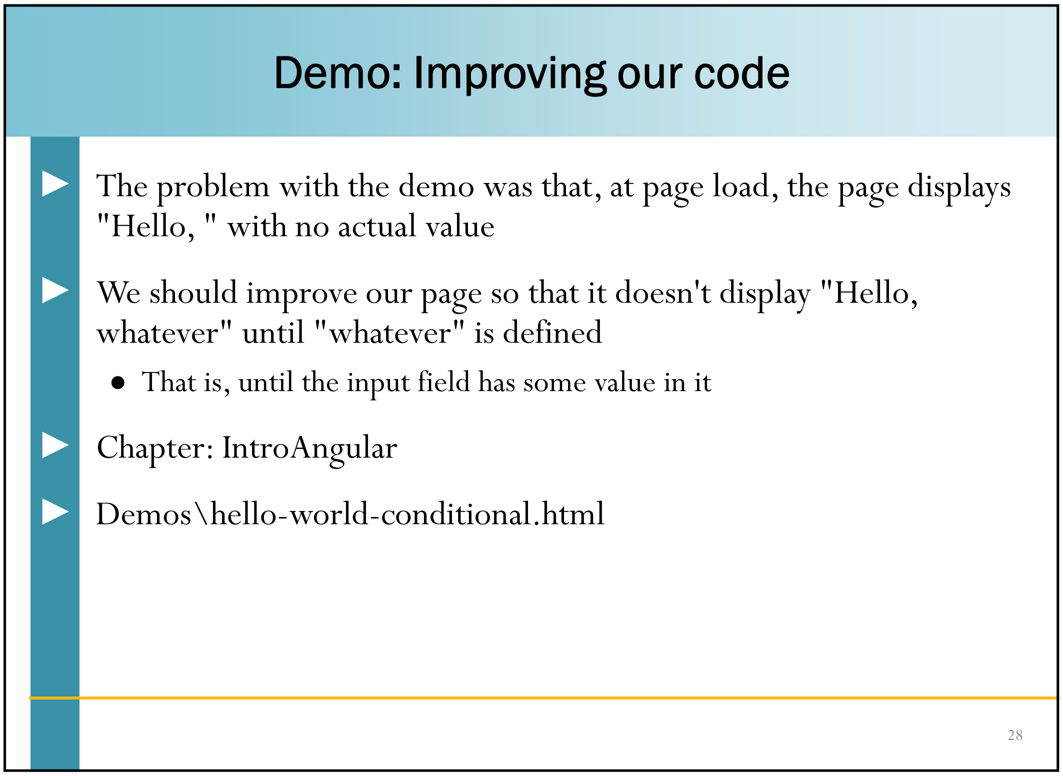
What just happened?

- We bound together a model and a view
  - That was quick!
- **`ng-model`** is a Directive, in Angular parlance
- It tells Angular to watch for changes in this field
- At the same time, we decided to display the value of that field a little later in the page
- Angular bound an event listener to changes in our ad-hoc model, and then updated the view (represented by the value in **{{ }}**) associated with it appropriately



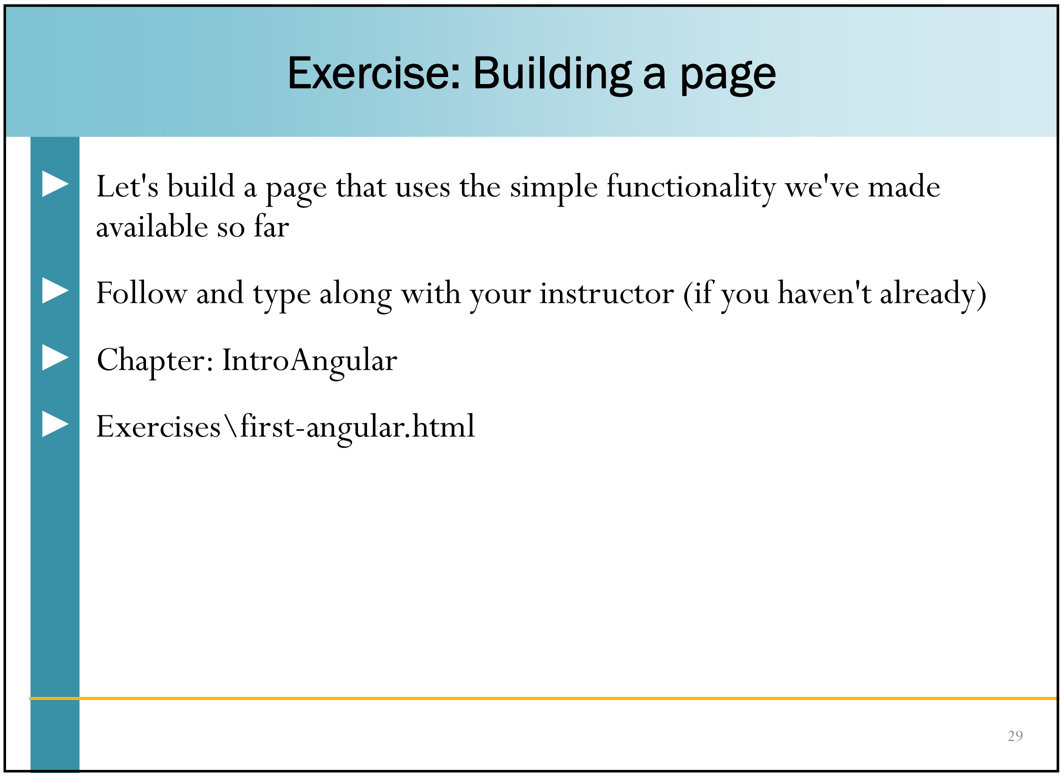
What's a directive?

- Angular's documentation says that "directives teach HTML new tricks"
- In reality, directives are shortcuts to bits of code that simplify some of the work we do in Angular
- For instance, you can add an **`ng-click`** directive to a button, and assign it a function
  - **`ng-click="addName(newName)"`**
- This is a shortcut for:
  - Assign a click event handler for this button
  - When it fires, run the **`addName`** function
  - Pass it the value **`newName`** (probably an **`ng-model`**)



Demo: Improving our code

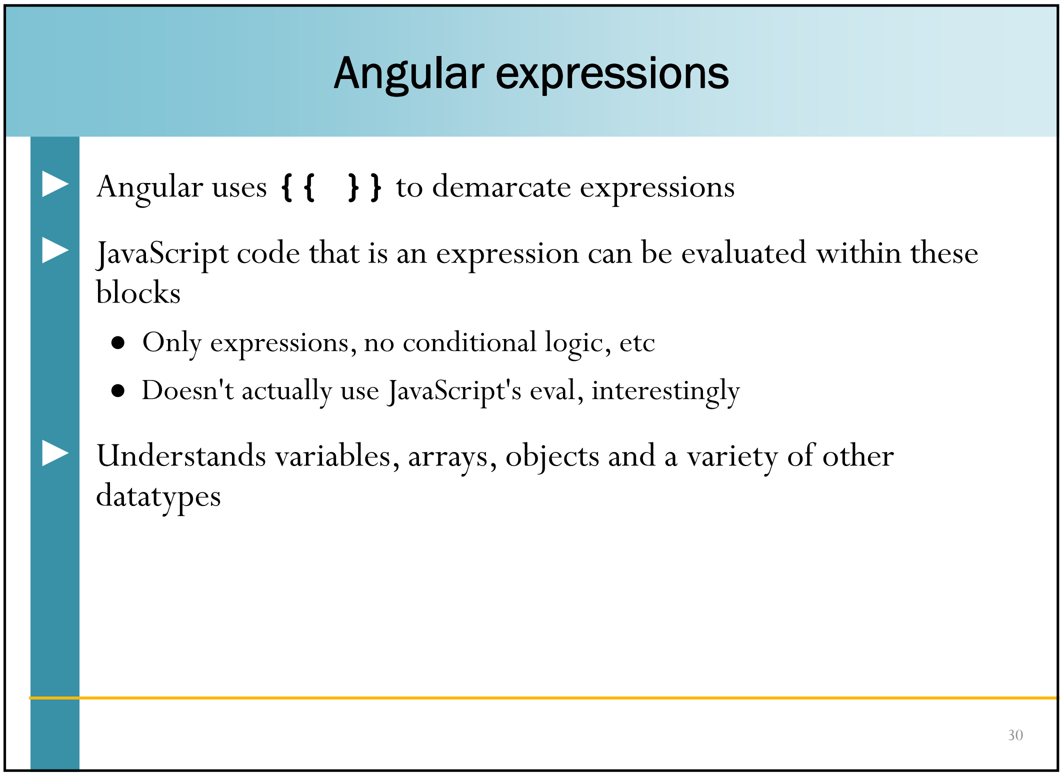
- The problem with the demo was that, at page load, the page displays "Hello, " with no actual value
- We should improve our page so that it doesn't display "Hello, whatever" until "whatever" is defined
  - That is, until the input field has some value in it
- Chapter: IntroAngular
- Demos\hello-world-conditional.html



Exercise: Building a page

- Let's build a page that uses the simple functionality we've made available so far
- Follow and type along with your instructor (if you haven't already)
- Chapter: IntroAngular
- Exercises\first-angular.html

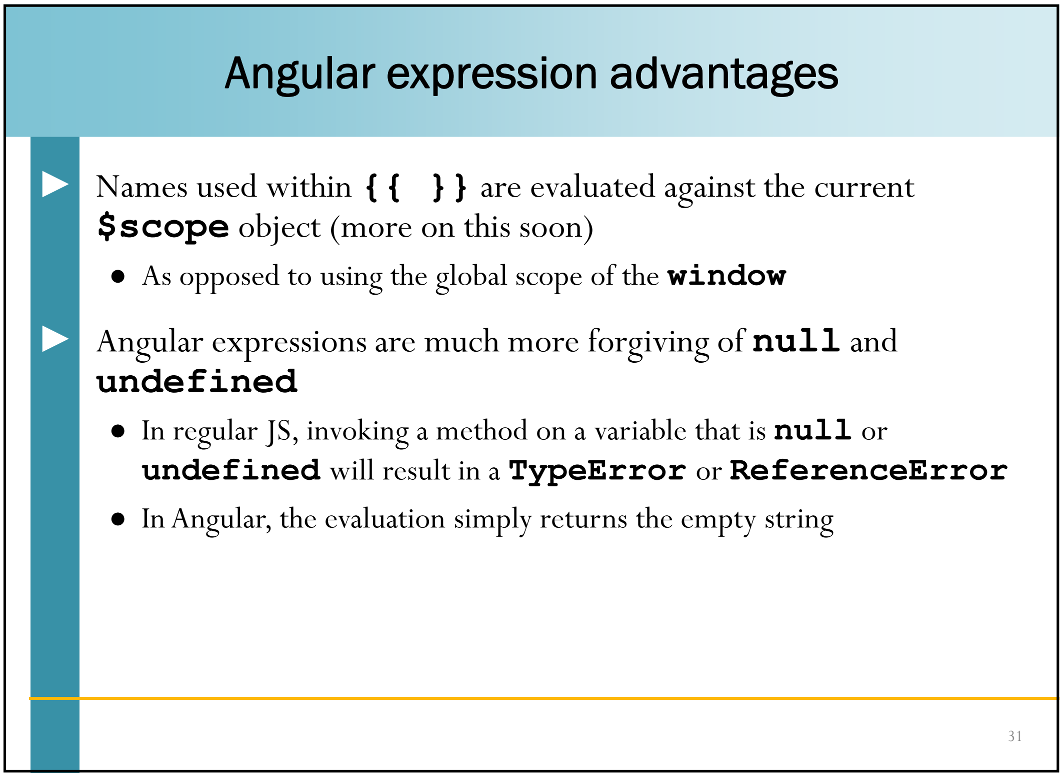
29



Angular expressions

- Angular uses **{{ }}** to demarcate expressions
- JavaScript code that is an expression can be evaluated within these blocks
  - Only expressions, no conditional logic, etc
  - Doesn't actually use JavaScript's eval, interestingly
- Understands variables, arrays, objects and a variety of other datatypes

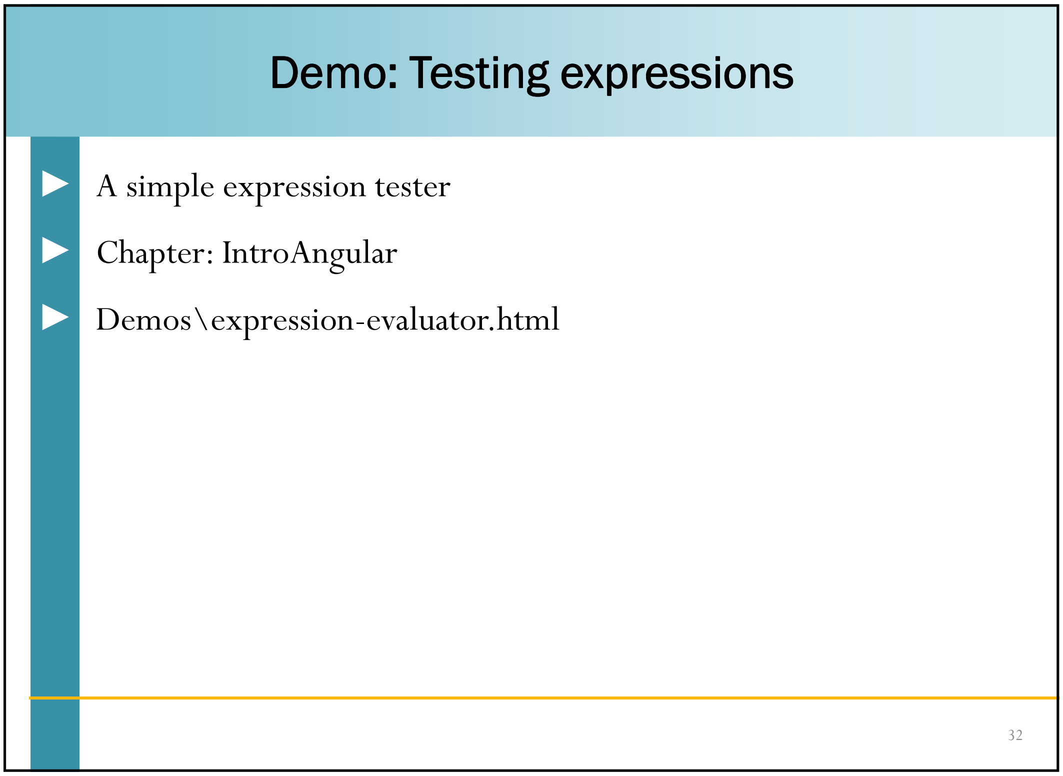
30



Angular expression advantages

- Names used within **`{{ }}`** are evaluated against the current **`$scope`** object (more on this soon)
  - As opposed to using the global scope of the **`window`**
- Angular expressions are much more forgiving of **`null`** and **`undefined`**
  - In regular JS, invoking a method on a variable that is **`null`** or **`undefined`** will result in a **`TypeError`** or **`ReferenceError`**
  - In Angular, the evaluation simply returns the empty string

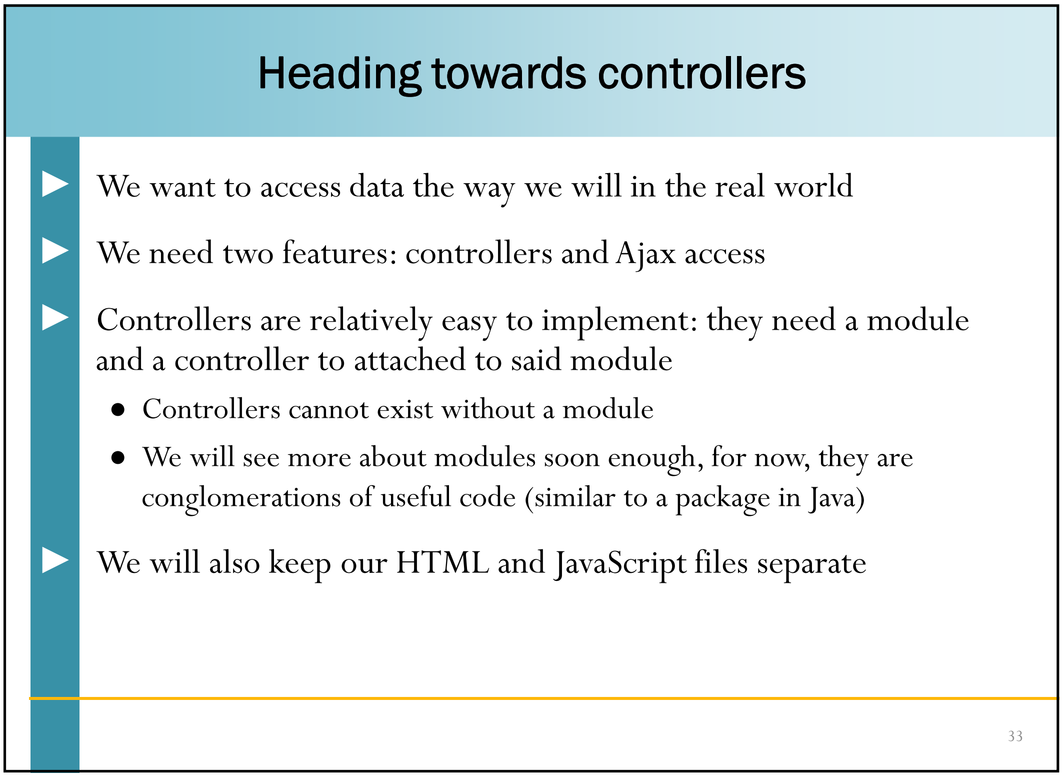
31



Demo: Testing expressions

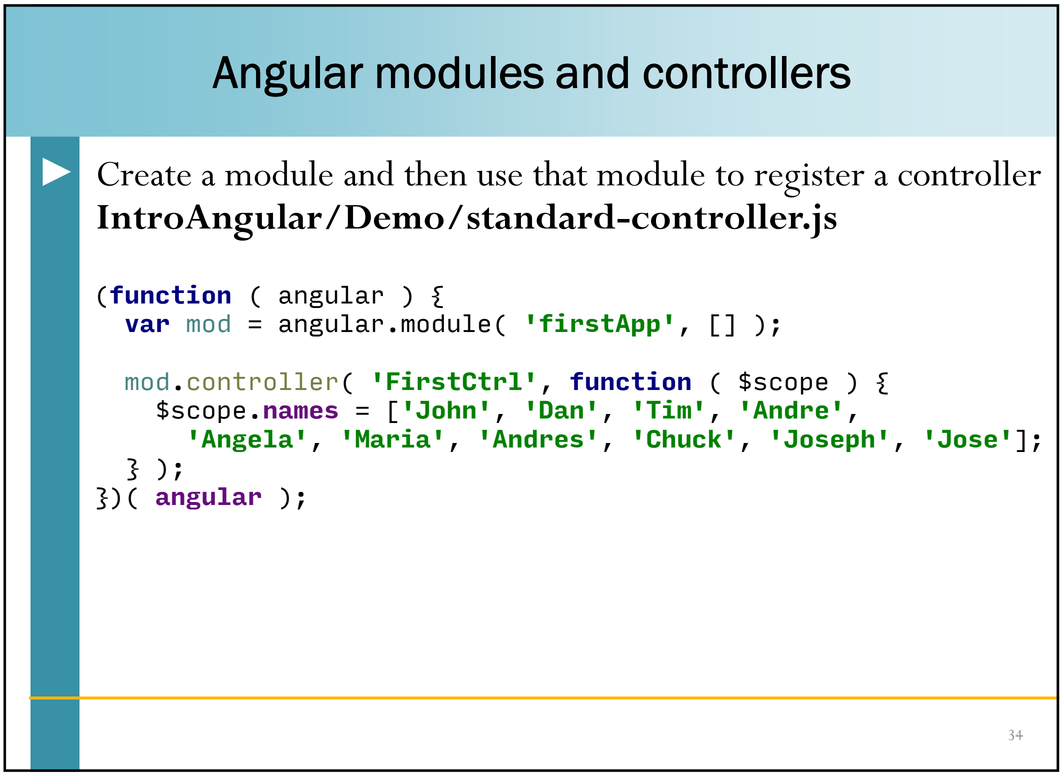
- A simple expression tester
- Chapter: IntroAngular
- Demos\expression-evaluator.html

32



Heading towards controllers

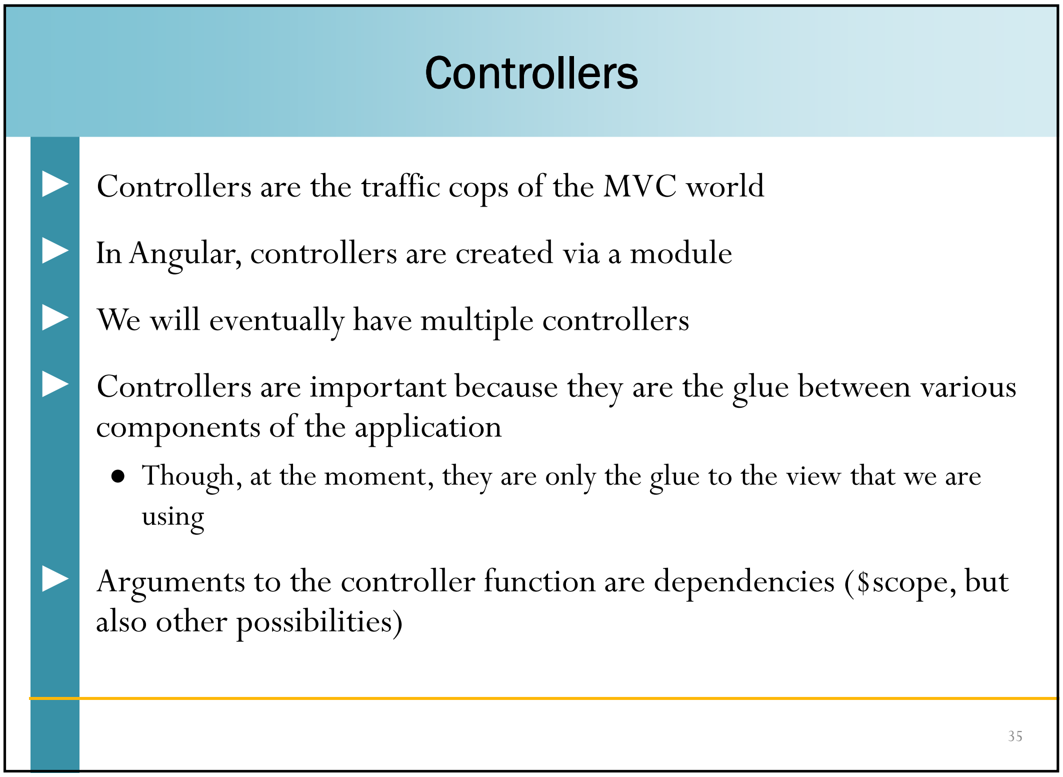
- We want to access data the way we will in the real world
- We need two features: controllers and Ajax access
- Controllers are relatively easy to implement: they need a module and a controller to attached to said module
  - Controllers cannot exist without a module
  - We will see more about modules soon enough, for now, they are conglomerations of useful code (similar to a package in Java)
- We will also keep our HTML and JavaScript files separate



Angular modules and controllers

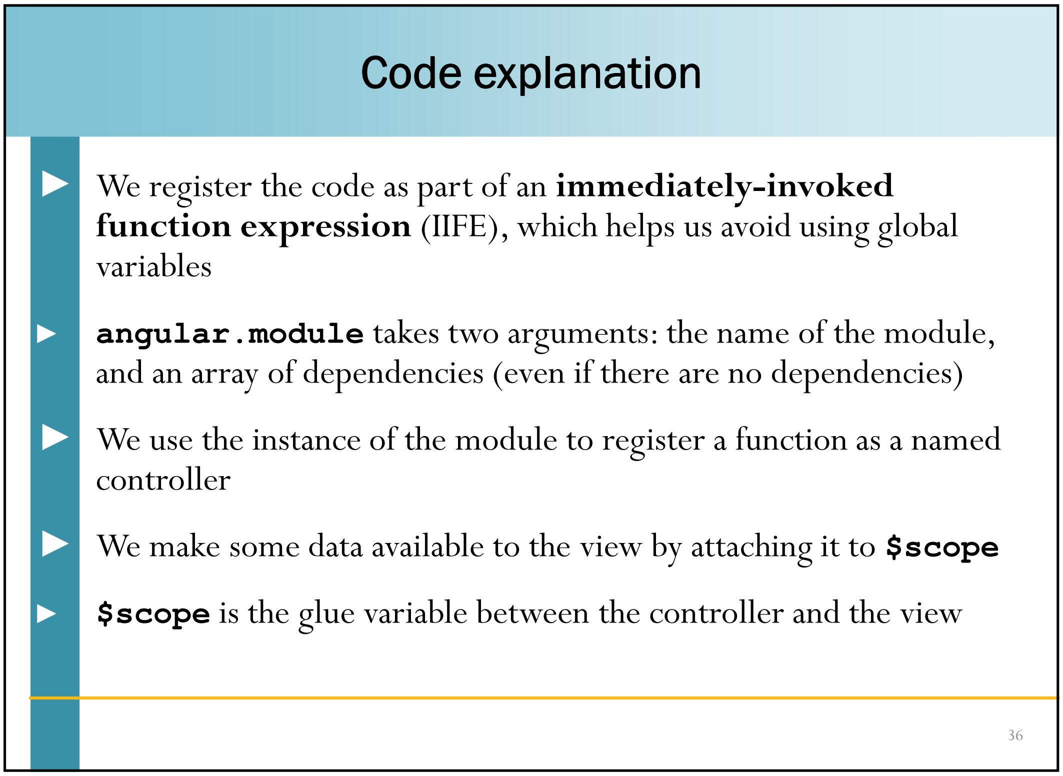
- Create a module and then use that module to register a controller **IntroAngular/Demo/standard-controller.js**

```
(function ( angular ) {
  var mod = angular.module( 'firstApp', [] );

  mod.controller( 'FirstCtrl', function ( $scope ) {
    $scope.names = ['John', 'Dan', 'Tim', 'Andre',
      'Angela', 'Maria', 'Andres', 'Chuck', 'Joseph', 'Jose'];
  } );
})( angular );
```



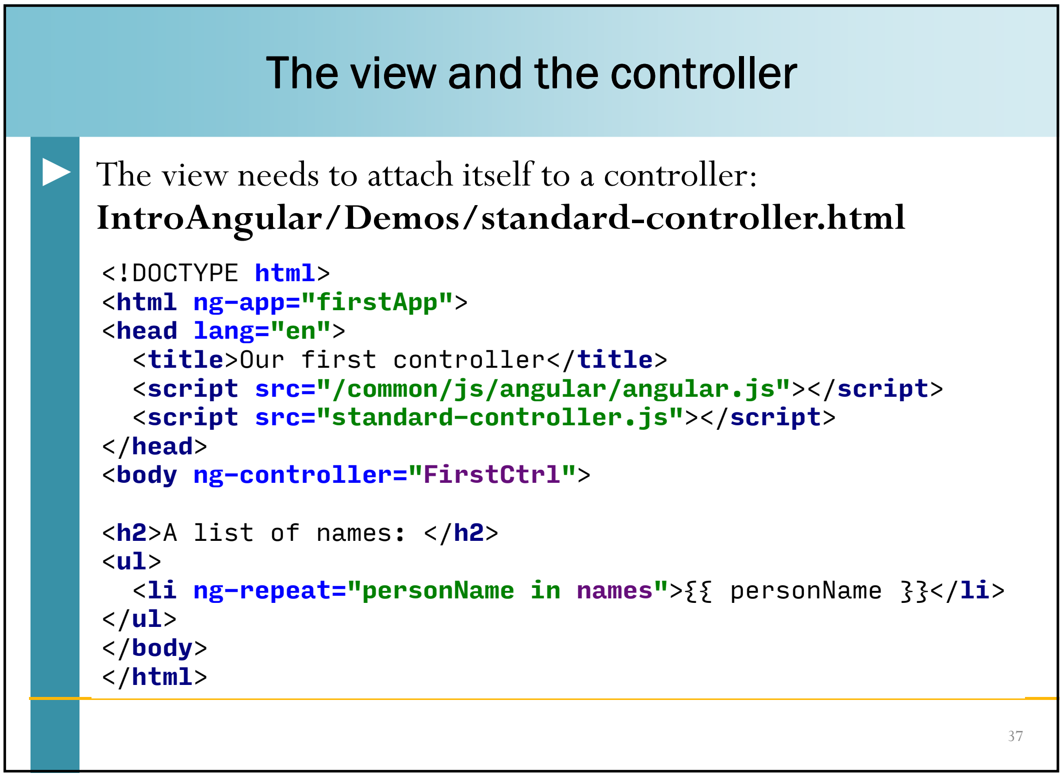
Controllers

- Controllers are the traffic cops of the MVC world
- In Angular, controllers are created via a module
- We will eventually have multiple controllers
- Controllers are important because they are the glue between various components of the application
  - Though, at the moment, they are only the glue to the view that we are using
- Arguments to the controller function are dependencies ($scope, but also other possibilities)



Code explanation

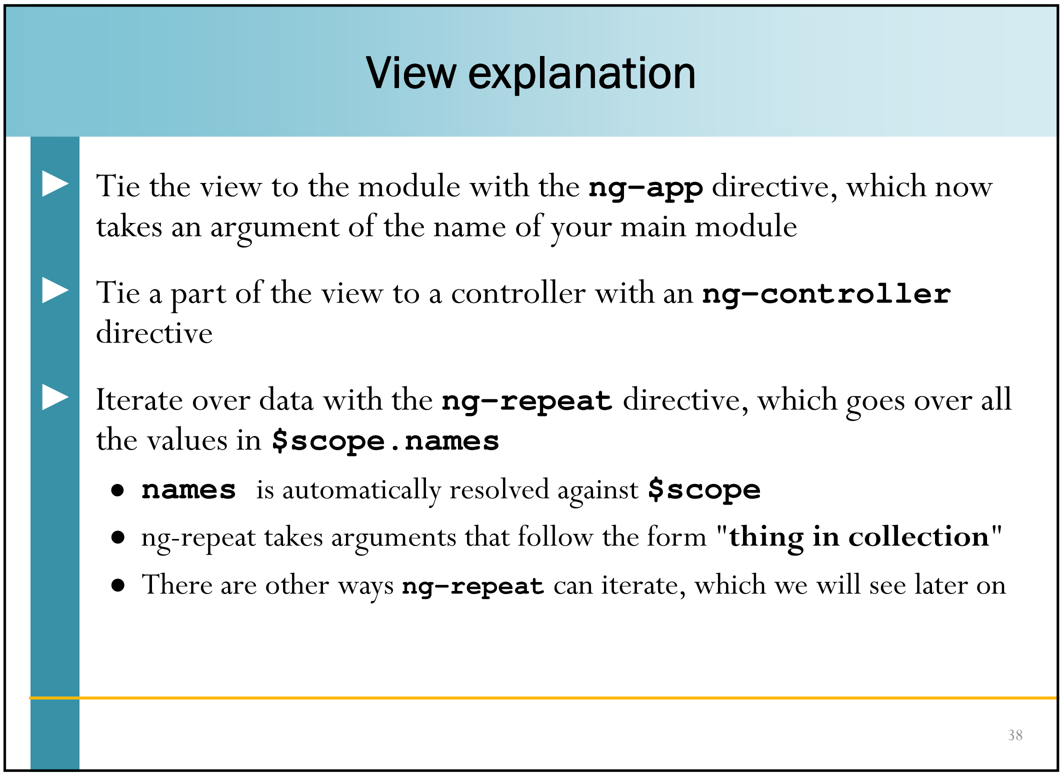
- We register the code as part of an **immediately-invoked function expression** (IIFE), which helps us avoid using global variables
- **`angular.module`** takes two arguments: the name of the module, and an array of dependencies (even if there are no dependencies)
- We use the instance of the module to register a function as a named controller
- We make some data available to the view by attaching it to **`$scope`**
- **`$scope`** is the glue variable between the controller and the view



The view and the controller

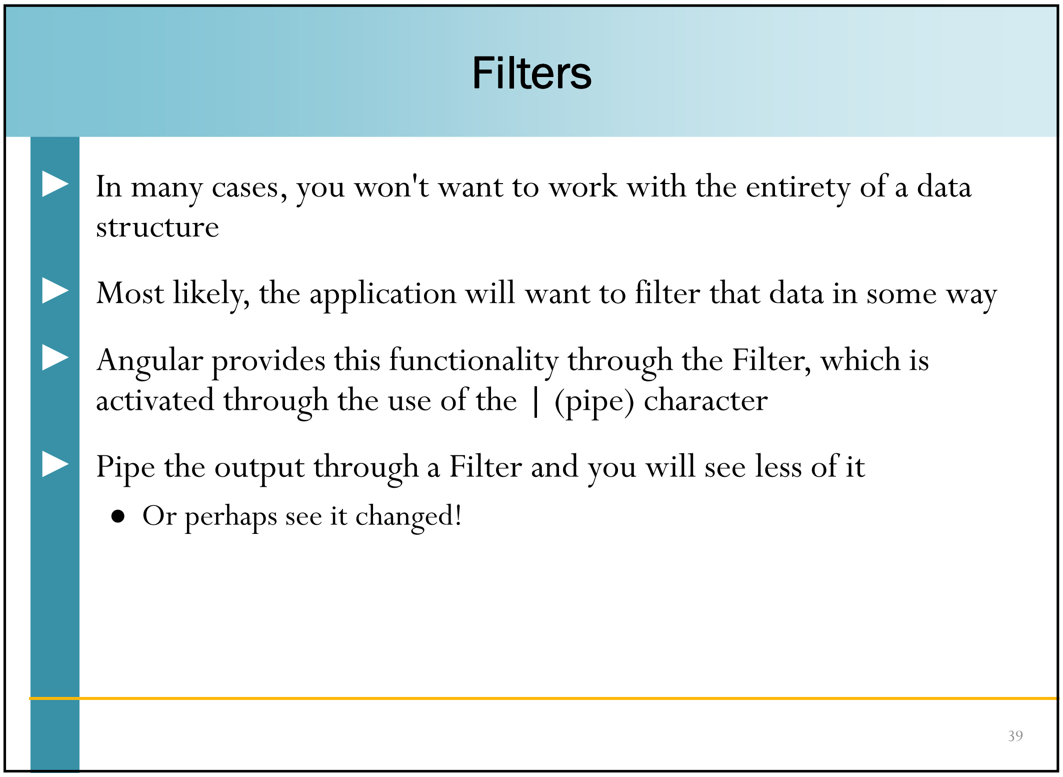
- The view needs to attach itself to a controller: **IntroAngular/Demos/standard-controller.html**

```
<!DOCTYPE html>
<html ng-app="firstApp">
<head lang="en">
  <title>Our first controller</title>
  <script src="/common/js/angular/angular.js"></script>
  <script src="standard-controller.js"></script>
</head>
<body ng-controller="FirstCtrl">

<h2>A list of names: </h2>
<ul>
  <li ng-repeat="personName in names">{{ personName }}</li>
</ul>
</body>
</html>
```



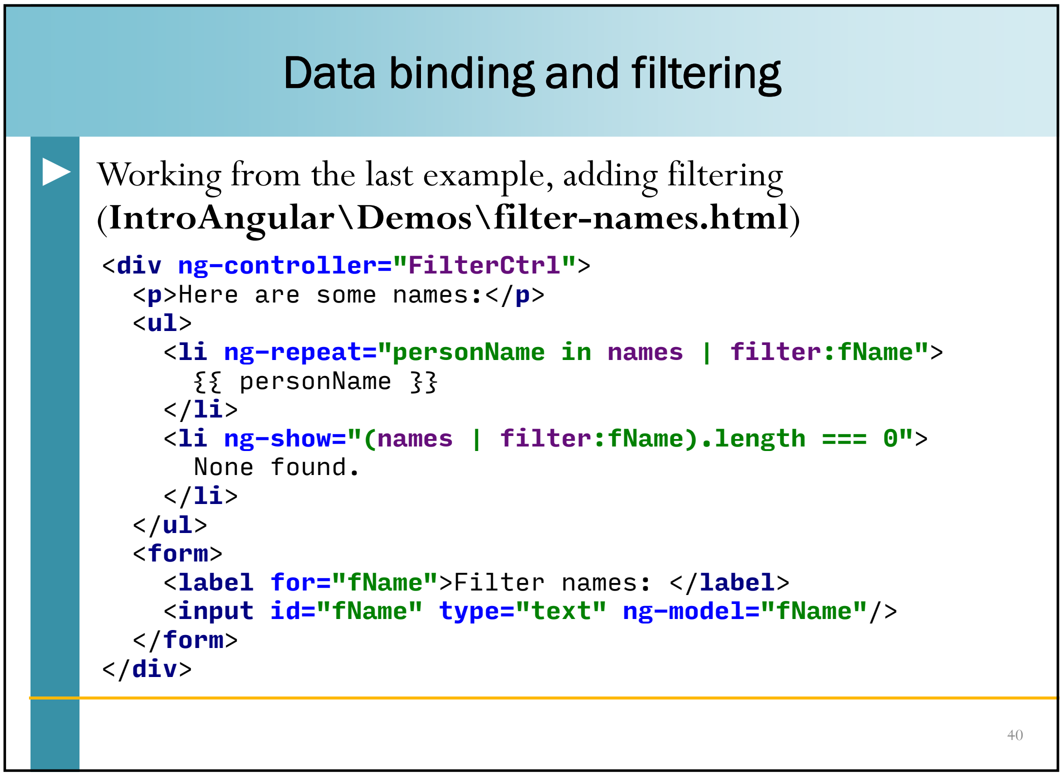
View explanation

- Tie the view to the module with the **`ng-app`** directive, which now takes an argument of the name of your main module
- Tie a part of the view to a controller with an **`ng-controller`** directive
- Iterate over data with the **`ng-repeat`** directive, which goes over all the values in **`$scope.names`**
  - **`names`** is automatically resolved against **`$scope`**
  - ng-repeat takes arguments that follow the form "**thing in collection**"
  - There are other ways **`ng-repeat`** can iterate, which we will see later on



Filters

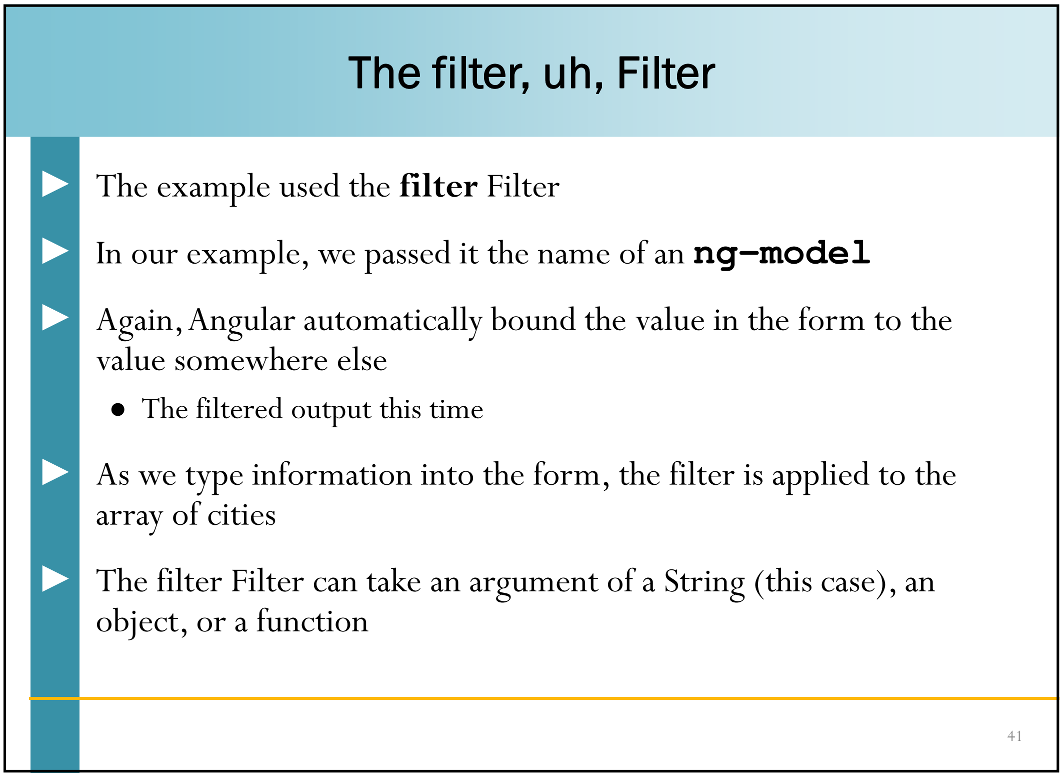
- In many cases, you won't want to work with the entirety of a data structure
- Most likely, the application will want to filter that data in some way
- Angular provides this functionality through the Filter, which is activated through the use of the | (pipe) character
- Pipe the output through a Filter and you will see less of it
  - Or perhaps see it changed!



Data binding and filtering

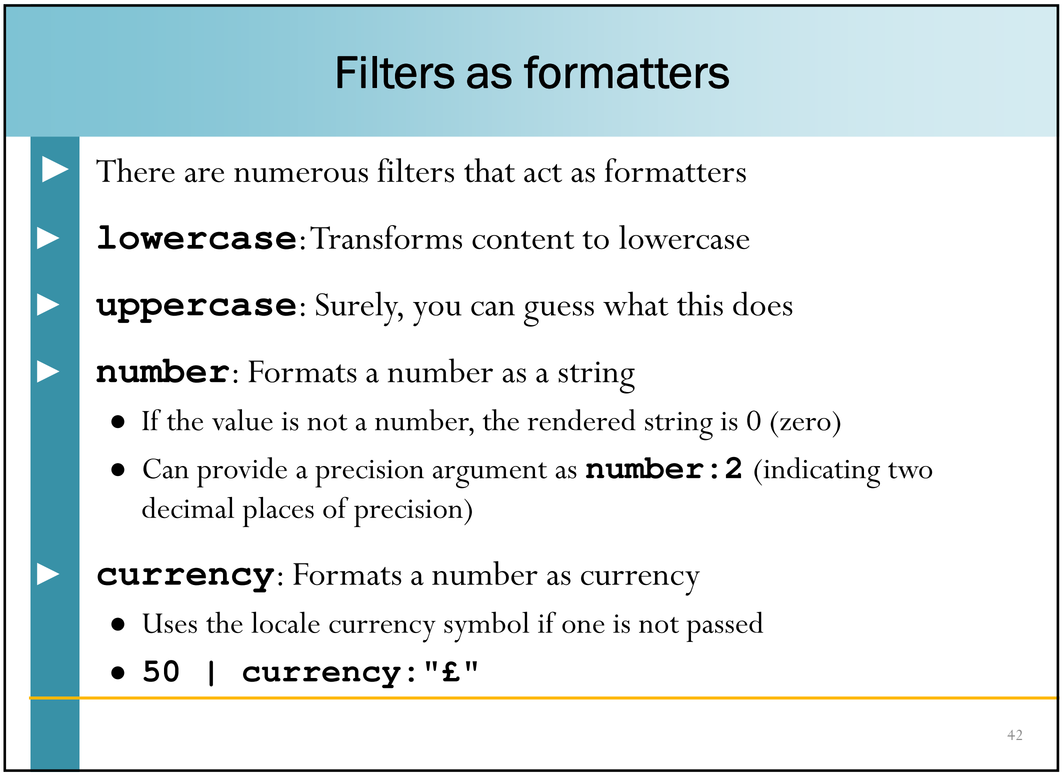
- Working from the last example, adding filtering (**IntroAngular\Demos\filter-names.html**)

```
<div ng-controller="FilterCtrl">
  <p>Here are some names:</p>
  <ul>
    <li ng-repeat="personName in names | filter:fName">
      {{ personName }}
    </li>
    <li ng-show="(names | filter:fName).length === 0">
      None found.
    </li>
  </ul>
  <form>
    <label for="fName">Filter names: </label>
    <input id="fName" type="text" ng-model="fName"/>
  </form>
</div>
```



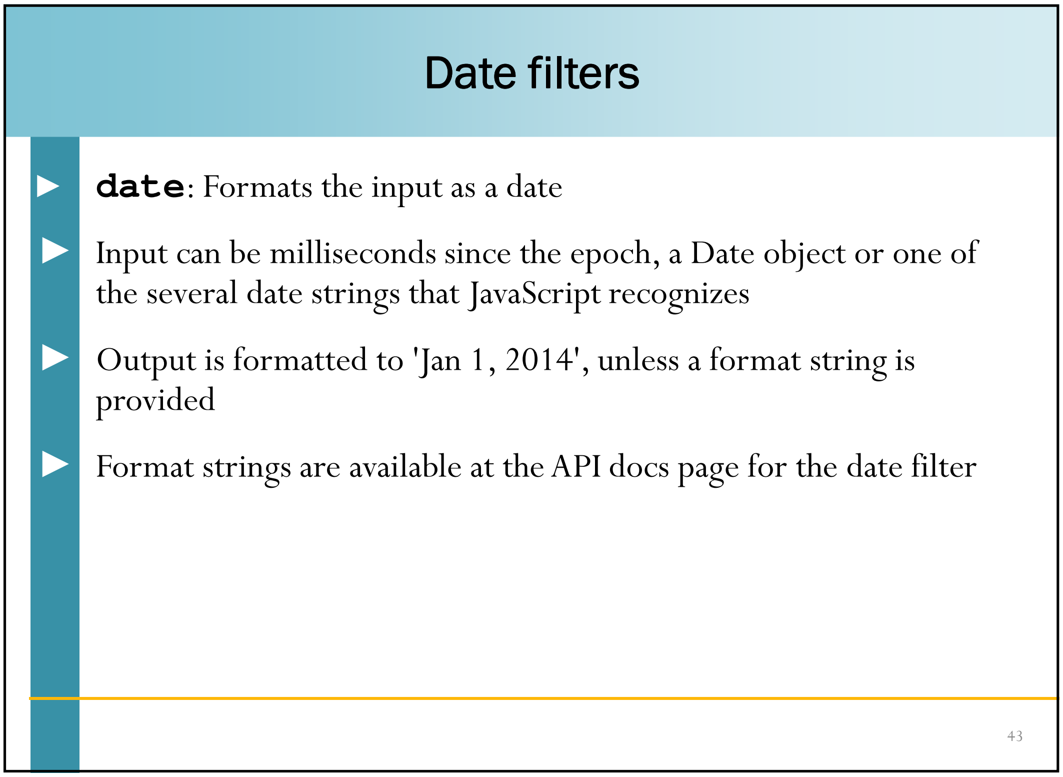
The filter, uh, Filter

- The example used the **filter** Filter
- In our example, we passed it the name of an **`ng-model`**
- Again, Angular automatically bound the value in the form to the value somewhere else
  - The filtered output this time
- As we type information into the form, the filter is applied to the array of cities
- The filter Filter can take an argument of a String (this case), an object, or a function



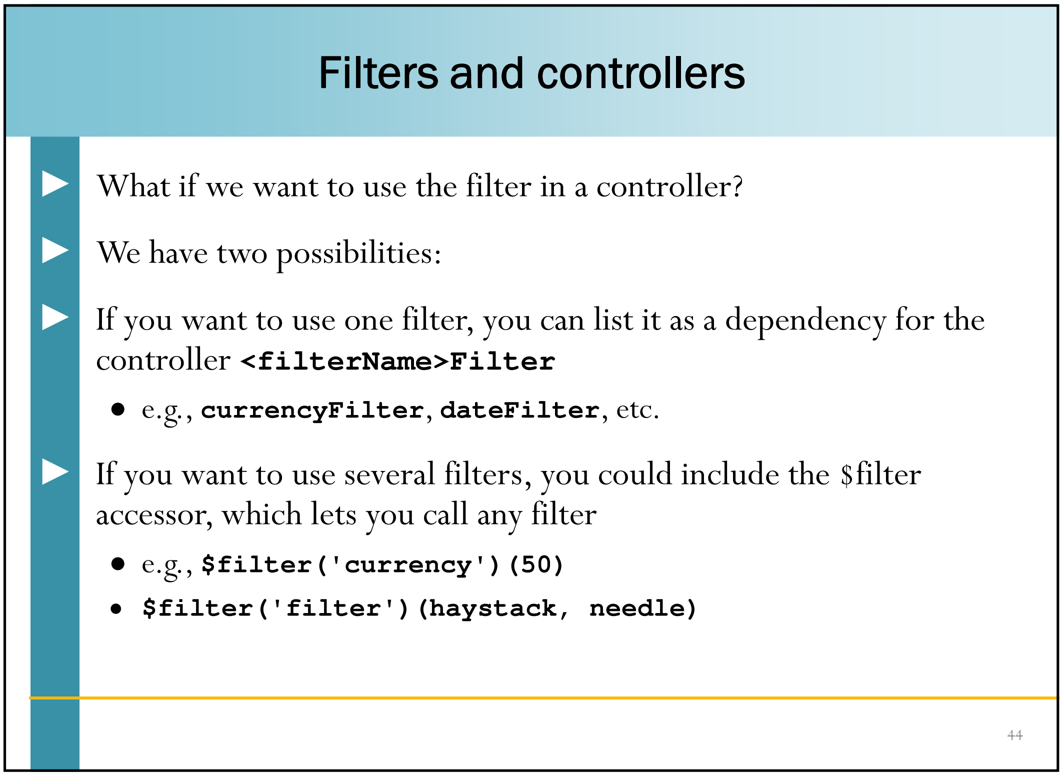
Filters as formatters

- There are numerous filters that act as formatters
- **`lowercase`**: Transforms content to lowercase
- **`uppercase`**: Surely, you can guess what this does
- **`number`**: Formats a number as a string
  - If the value is not a number, the rendered string is 0 (zero)
  - Can provide a precision argument as **`number:2`** (indicating two decimal places of precision)
- **`currency`**: Formats a number as currency
  - Uses the locale currency symbol if one is not passed
  - **`50 | currency:"£"`**



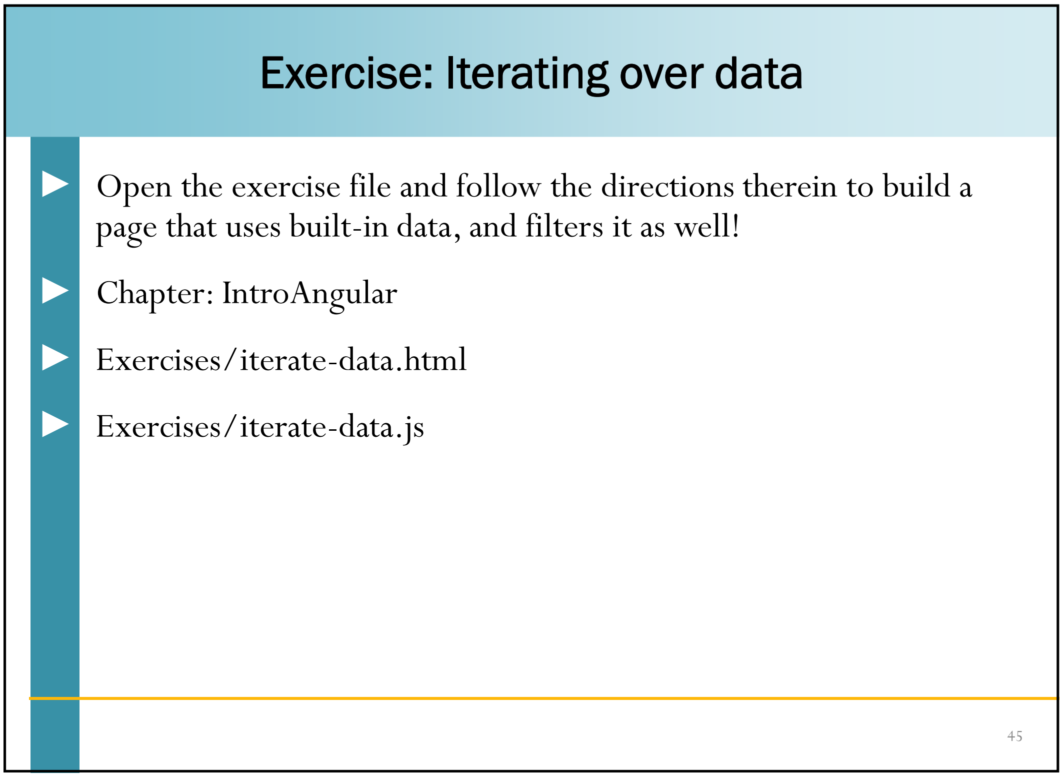
Date filters

- **`date`**: Formats the input as a date
- Input can be milliseconds since the epoch, a Date object or one of the several date strings that JavaScript recognizes
- Output is formatted to 'Jan 1, 2014', unless a format string is provided
- Format strings are available at the API docs page for the date filter



Filters and controllers

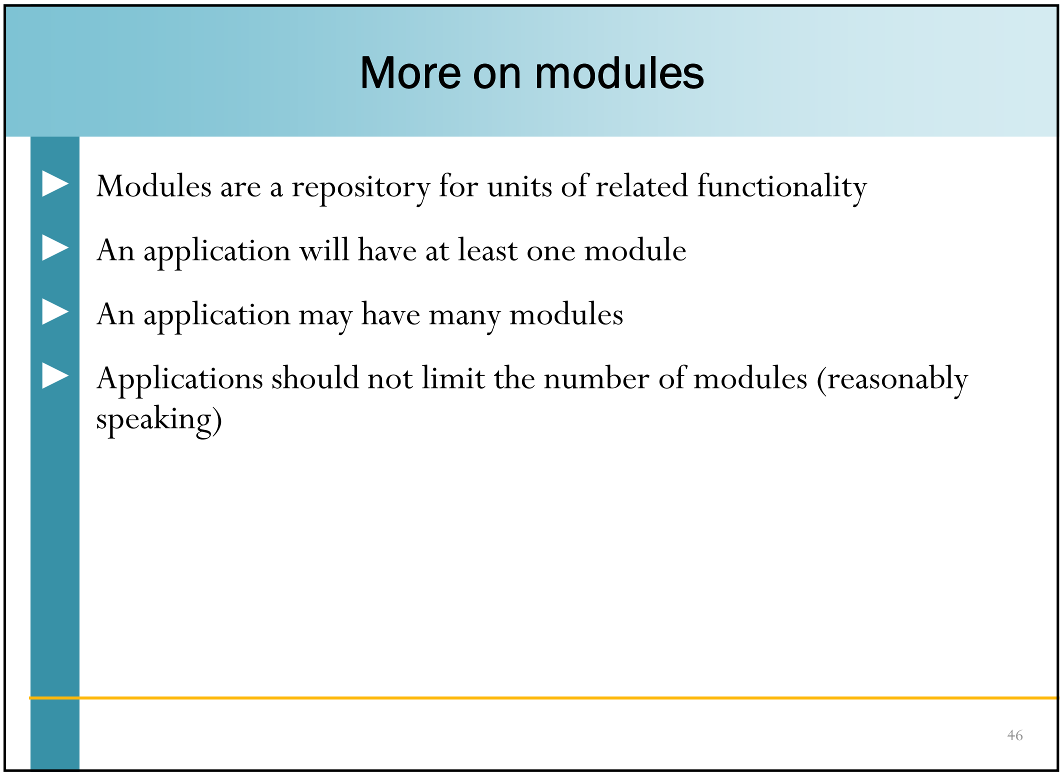
- What if we want to use the filter in a controller?
- We have two possibilities:
- If you want to use one filter, you can list it as a dependency for the controller **`<filterName>Filter`**
  - e.g., **`currencyFilter`**, **`dateFilter`**, etc.
- If you want to use several filters, you could include the $filter accessor, which lets you call any filter
  - e.g., **`$filter('currency')(50)`**
  - **`$filter('filter')(haystack, needle)`**



Exercise: Iterating over data

- Open the exercise file and follow the directions therein to build a page that uses built-in data, and filters it as well!
- Chapter: IntroAngular
- Exercises/iterate-data.html
- Exercises/iterate-data.js

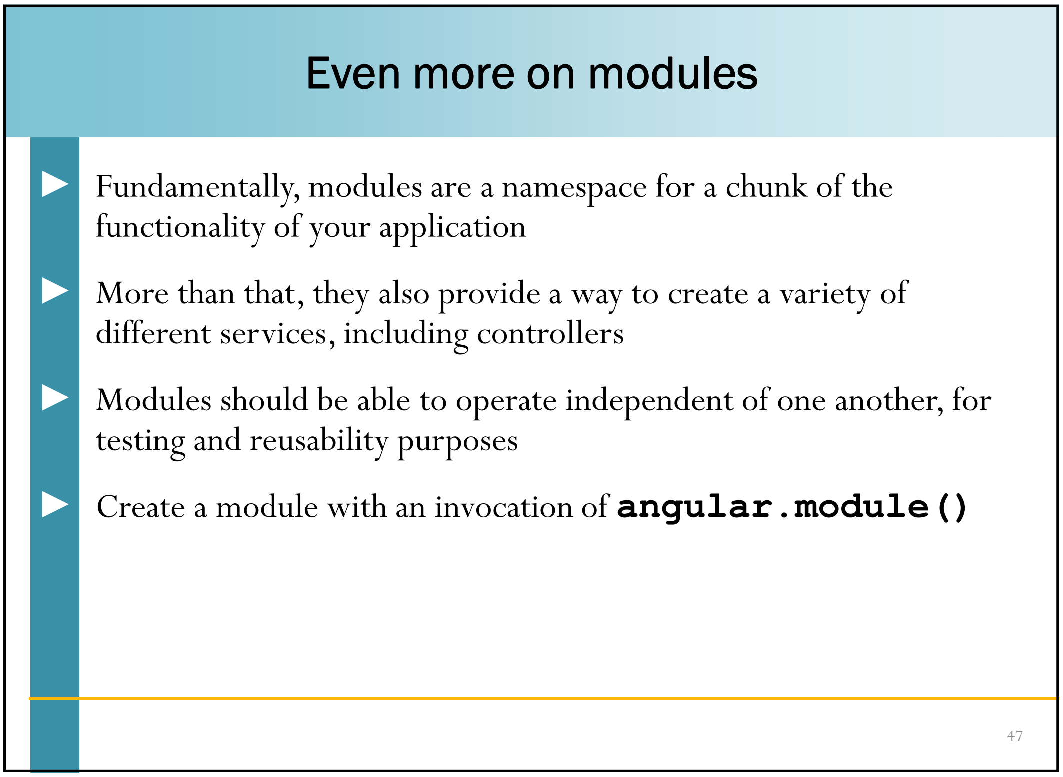
45



More on modules

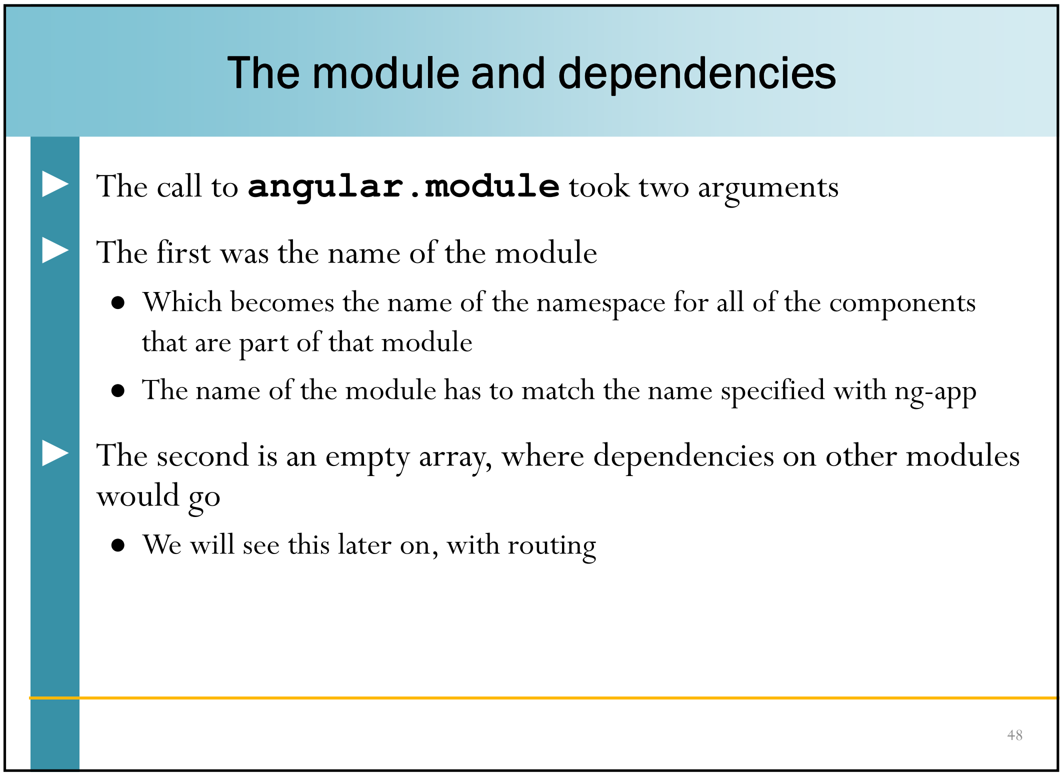
- Modules are a repository for units of related functionality
- An application will have at least one module
- An application may have many modules
- Applications should not limit the number of modules (reasonably speaking)

46



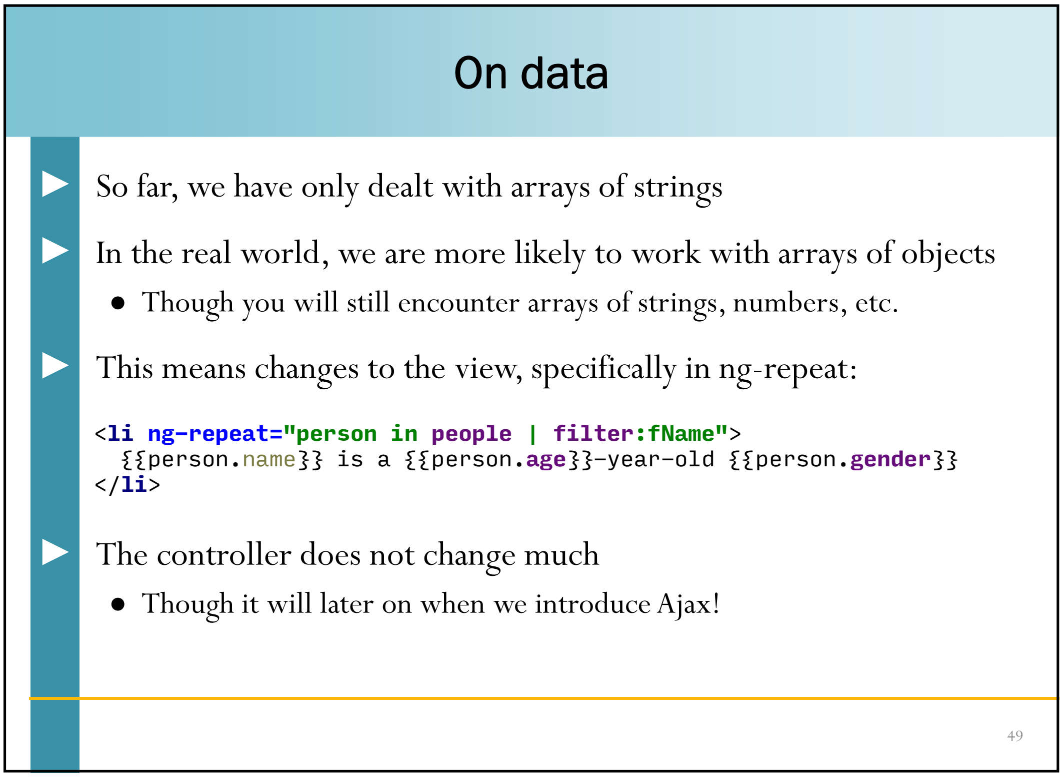
Even more on modules

- Fundamentally, modules are a namespace for a chunk of the functionality of your application
- More than that, they also provide a way to create a variety of different services, including controllers
- Modules should be able to operate independent of one another, for testing and reusability purposes
- Create a module with an invocation of **`angular.module()`**



The module and dependencies

- The call to **`angular.module`** took two arguments
- The first was the name of the module
  - Which becomes the name of the namespace for all of the components that are part of that module
  - The name of the module has to match the name specified with ng-app
- The second is an empty array, where dependencies on other modules would go
  - We will see this later on, with routing

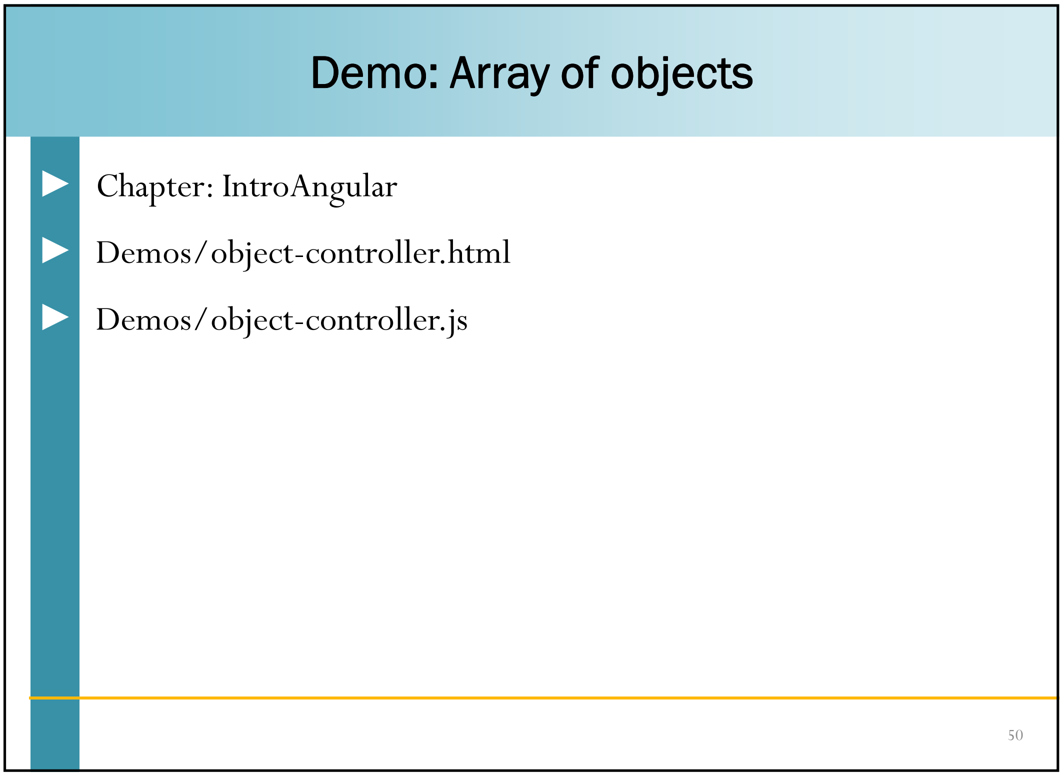


On data

- So far, we have only dealt with arrays of strings
- In the real world, we are more likely to work with arrays of objects
  - Though you will still encounter arrays of strings, numbers, etc.
- This means changes to the view, specifically in ng-repeat:

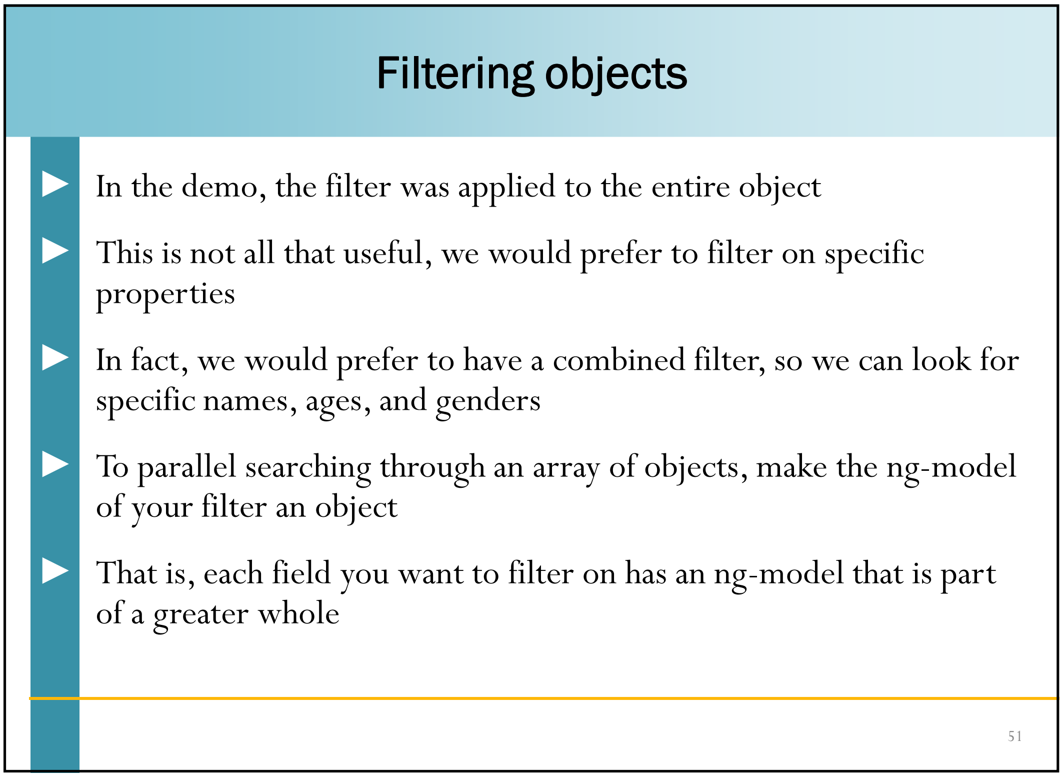
```
<li ng-repeat="person in people | filter:fName">
  {{person.name}} is a {{person.age}}-year-old {{person.gender}}
</li>
```

- The controller does not change much
  - Though it will later on when we introduce Ajax!



Demo: Array of objects

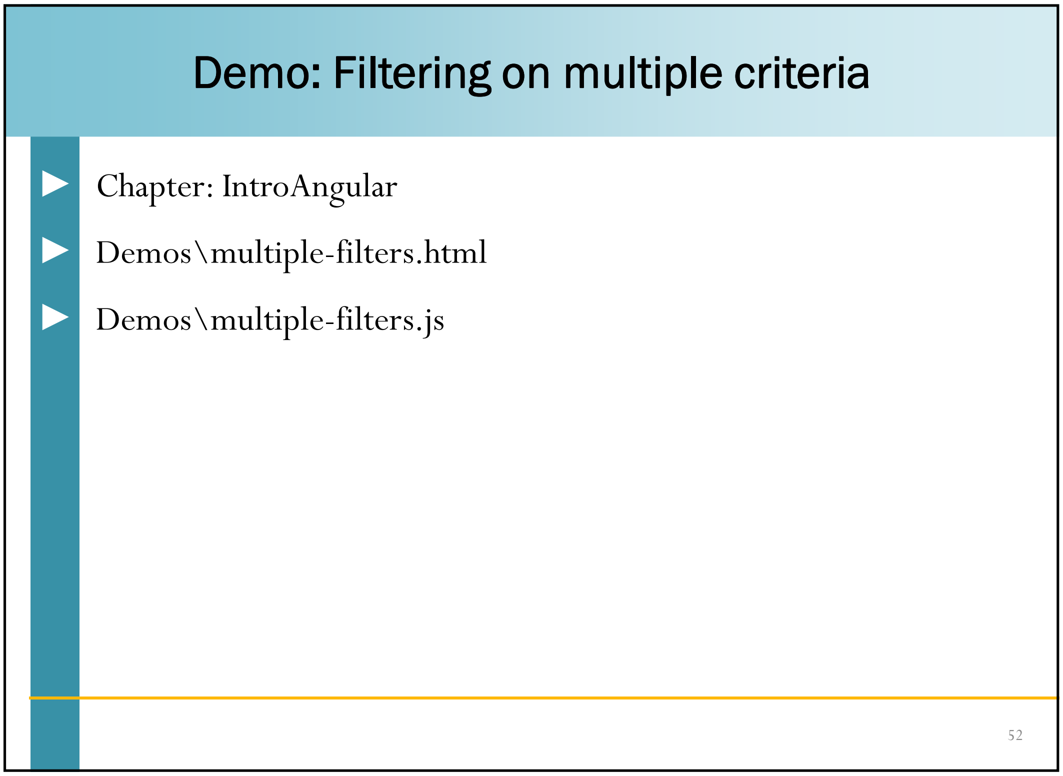
- Chapter: IntroAngular
- Demos/object-controller.html
- Demos/object-controller.js



Filtering objects

- In the demo, the filter was applied to the entire object
- This is not all that useful, we would prefer to filter on specific properties
- In fact, we would prefer to have a combined filter, so we can look for specific names, ages, and genders
- To parallel searching through an array of objects, make the ng-model of your filter an object
- That is, each field you want to filter on has an ng-model that is part of a greater whole

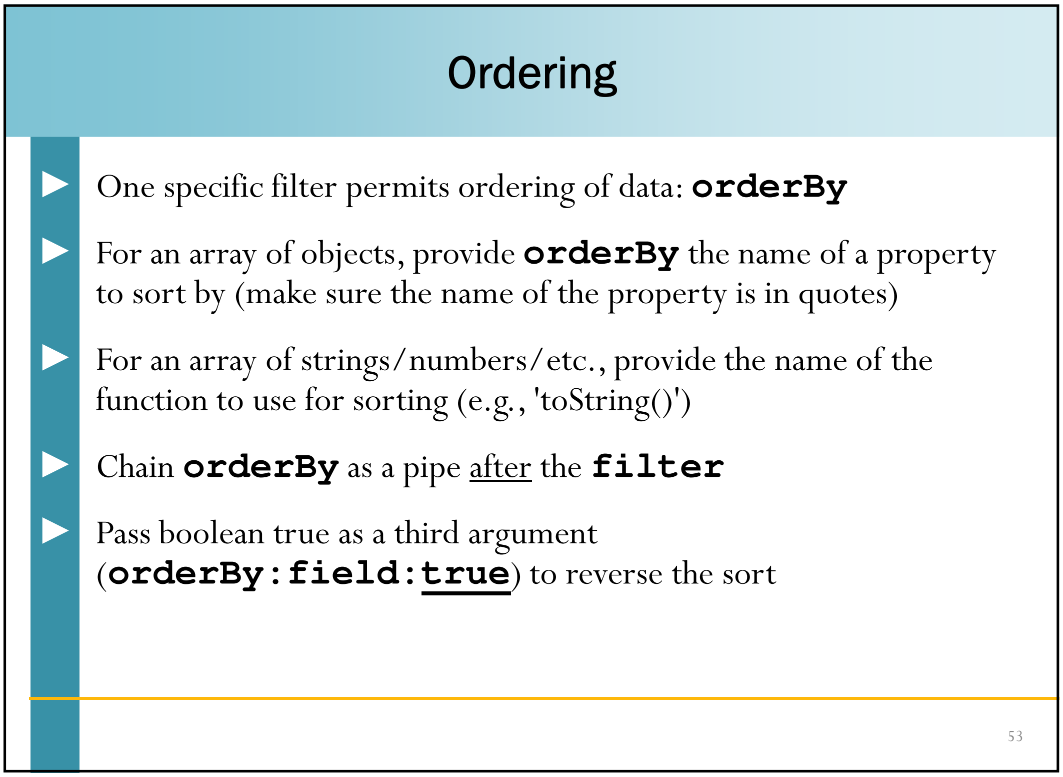
51



Demo: Filtering on multiple criteria

- Chapter: IntroAngular
- Demos\multiple-filters.html
- Demos\multiple-filters.js

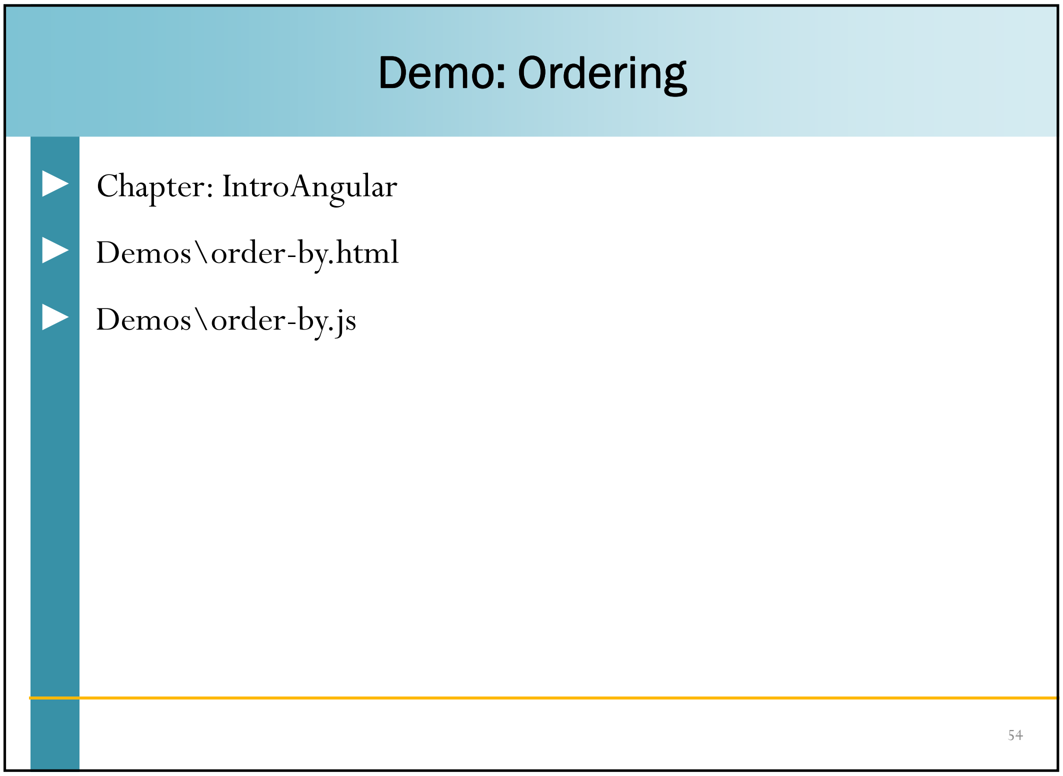
52



Ordering

- One specific filter permits ordering of data: **`orderBy`**
- For an array of objects, provide **`orderBy`** the name of a property to sort by (make sure the name of the property is in quotes)
- For an array of strings/numbers/etc., provide the name of the function to use for sorting (e.g., 'toString()')
- Chain **`orderBy`** as a pipe after the **`filter`**
- Pass boolean true as a third argument (**`orderBy:field:true`**) to reverse the sort

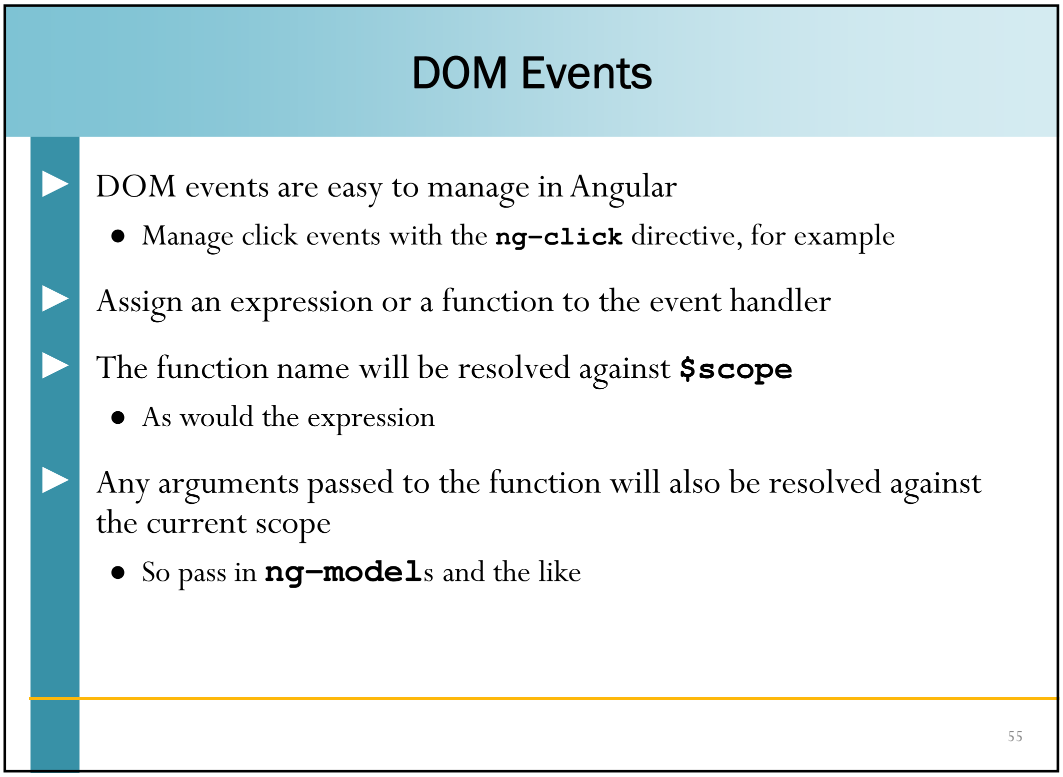
53



Demo: Ordering

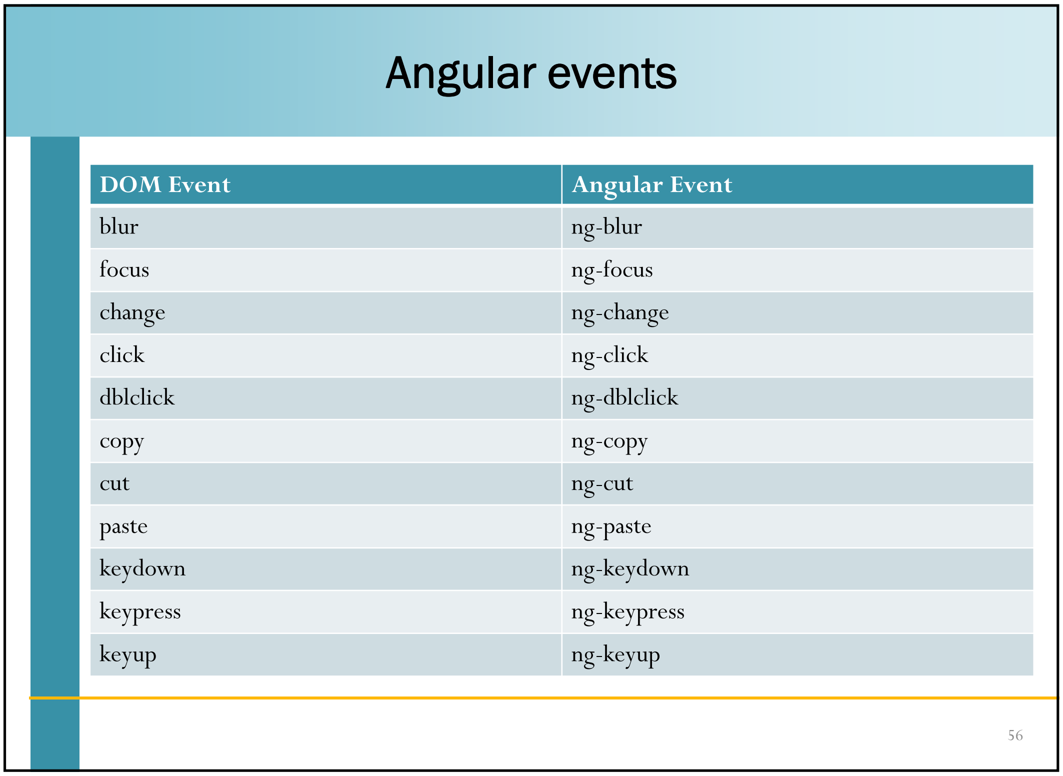
- Chapter: IntroAngular
- Demos\order-by.html
- Demos\order-by.js

54



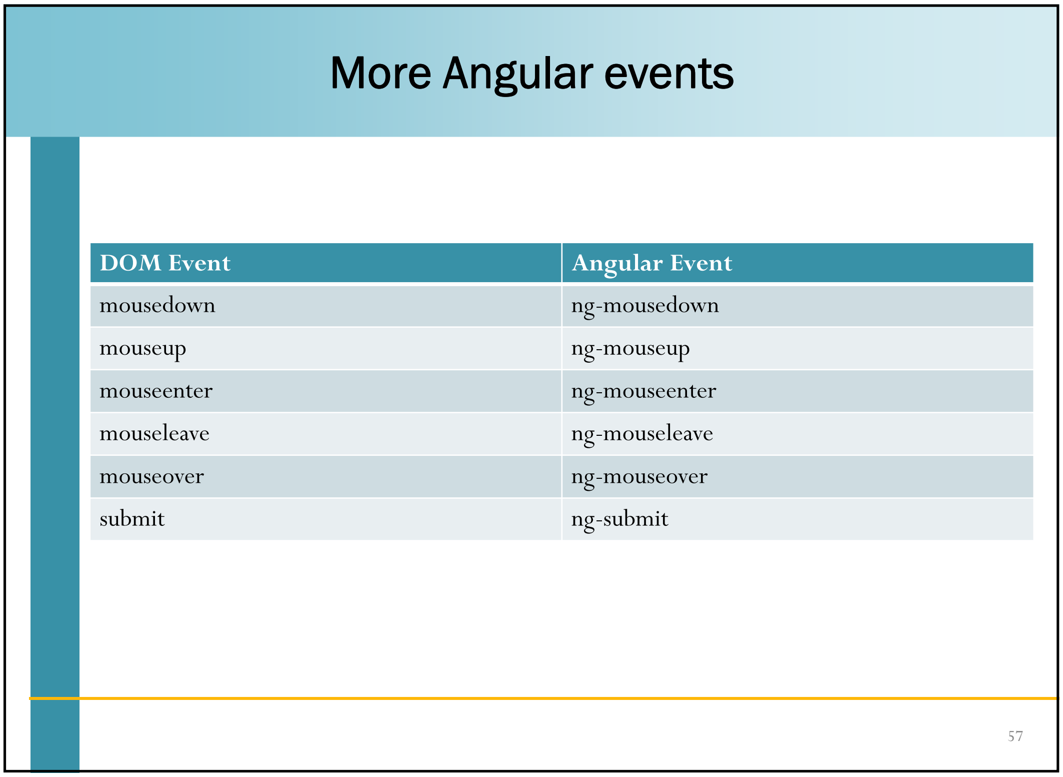
DOM Events

- DOM events are easy to manage in Angular
  - Manage click events with the **`ng-click`** directive, for example
- Assign an expression or a function to the event handler
- The function name will be resolved against **`$scope`**
  - As would the expression
- Any arguments passed to the function will also be resolved against the current scope
  - So pass in **`ng-model`**s and the like



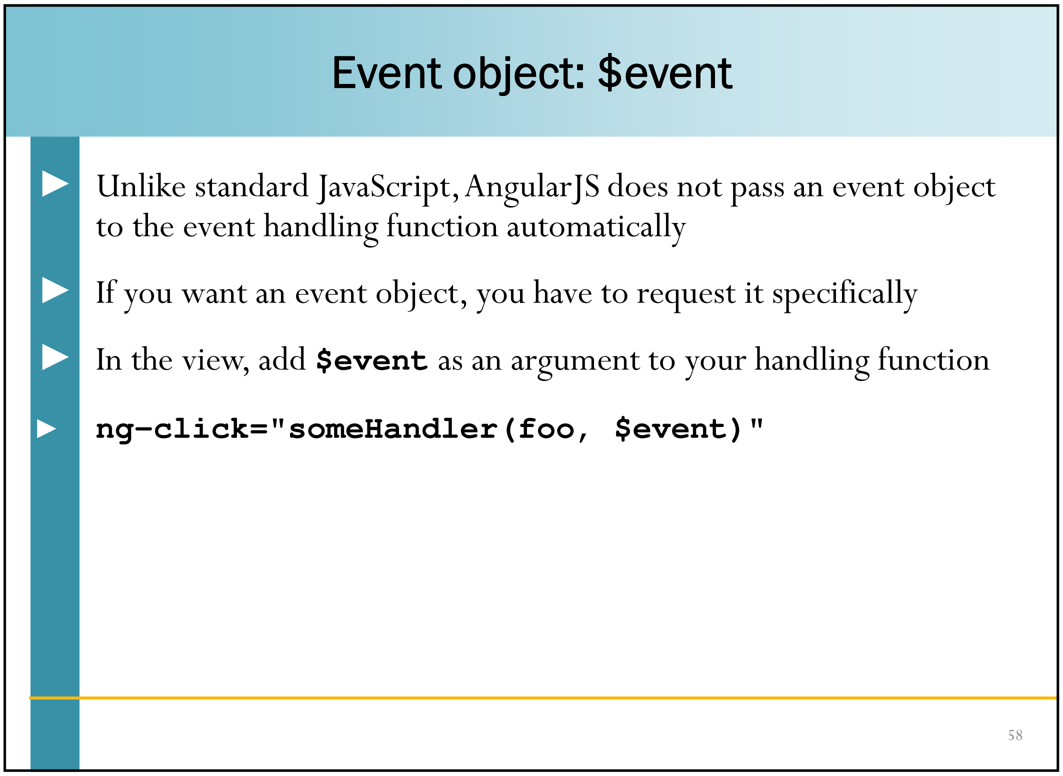
Angular events

DOM Event	Angular Event
blur	ng-blur
focus	ng-focus
change	ng-change
click	ng-click
dblclick	ng-dblclick
copy	ng-copy
cut	ng-cut
paste	ng-paste
keydown	ng-keydown
keypress	ng-keypress
keyup	ng-keyup



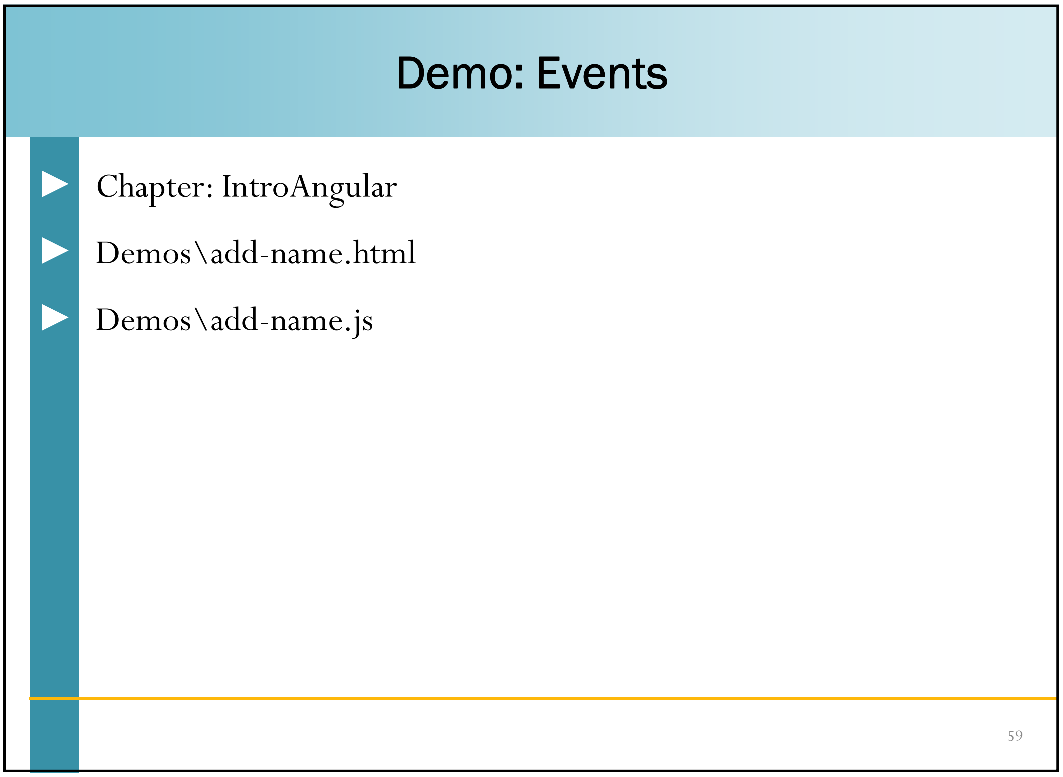
More Angular events

DOM Event	Angular Event
mousedown	ng-mousedown
mouseup	ng-mouseup
mouseenter	ng-mouseenter
mouseleave	ng-mouseleave
mouseover	ng-mouseover
submit	ng-submit



Event object: $event

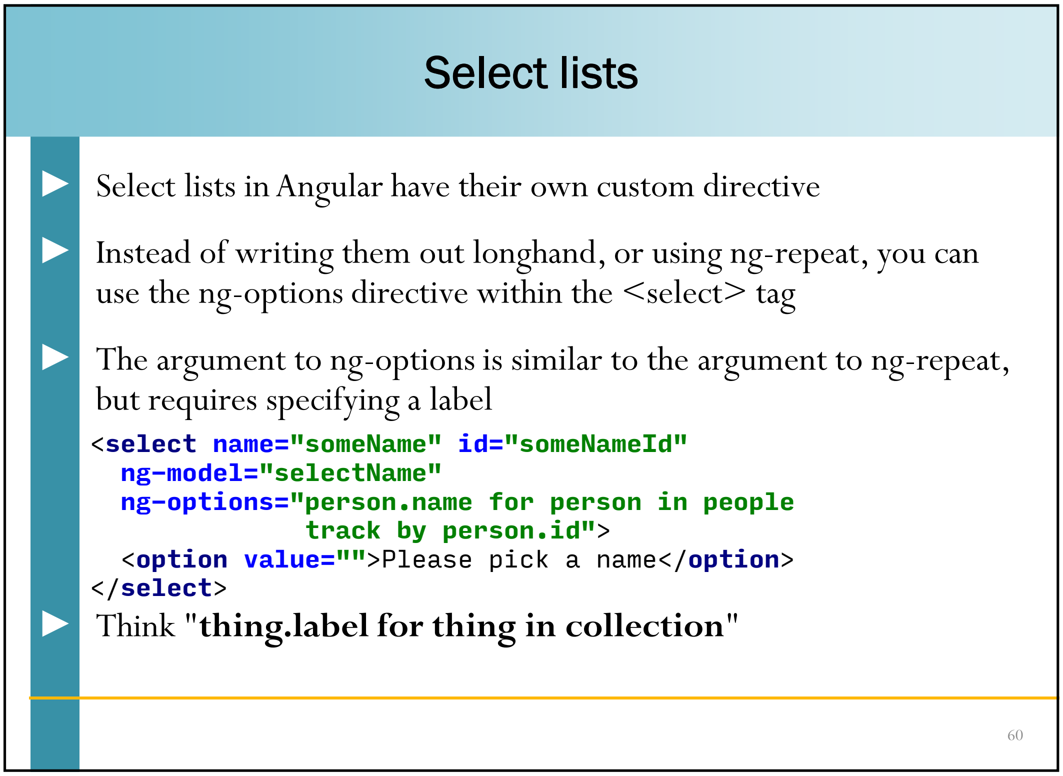
- Unlike standard JavaScript, AngularJS does not pass an event object to the event handling function automatically
- If you want an event object, you have to request it specifically
- In the view, add **`$event`** as an argument to your handling function
- **`ng-click="someHandler(foo, $event)"`**



Demo: Events

- Chapter: IntroAngular
- Demos\add-name.html
- Demos\add-name.js

59



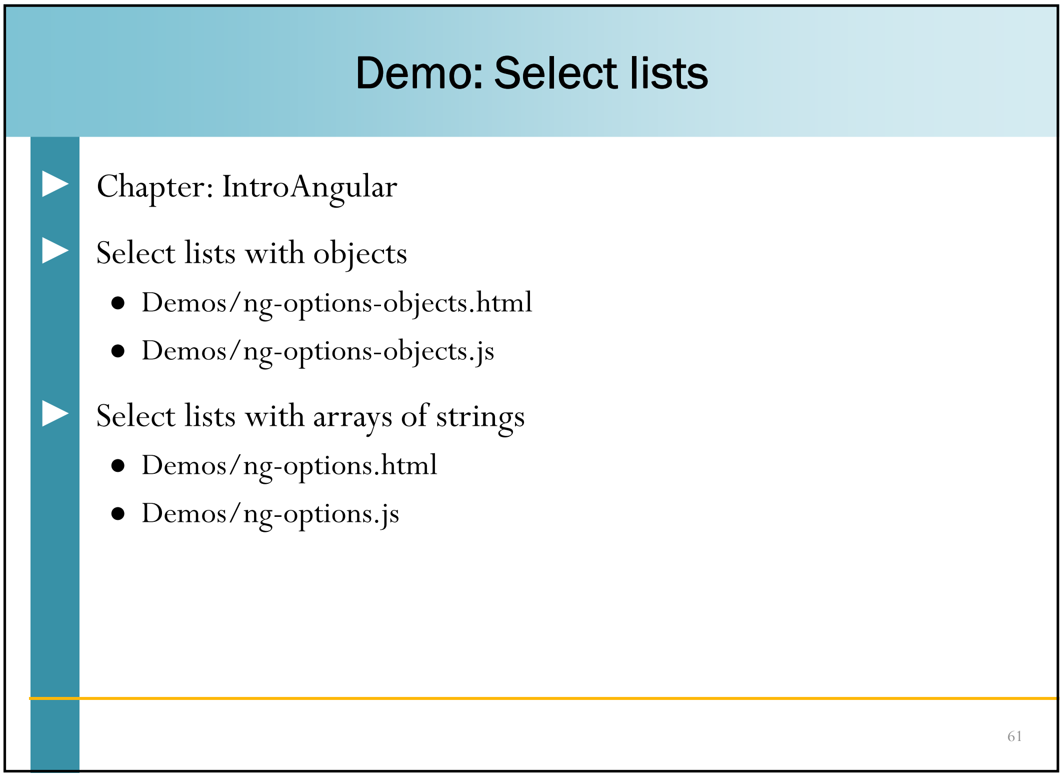
Select lists

- Select lists in Angular have their own custom directive
- Instead of writing them out longhand, or using ng-repeat, you can use the ng-options directive within the <select> tag
- The argument to ng-options is similar to the argument to ng-repeat, but requires specifying a label

```
<select name="someName" id="someNameId"
  ng-model="selectName"
  ng-options="person.name for person in people
              track by person.id">
  <option value="">Please pick a name</option>
</select>
```

- Think "**thing.label for thing in collection**"

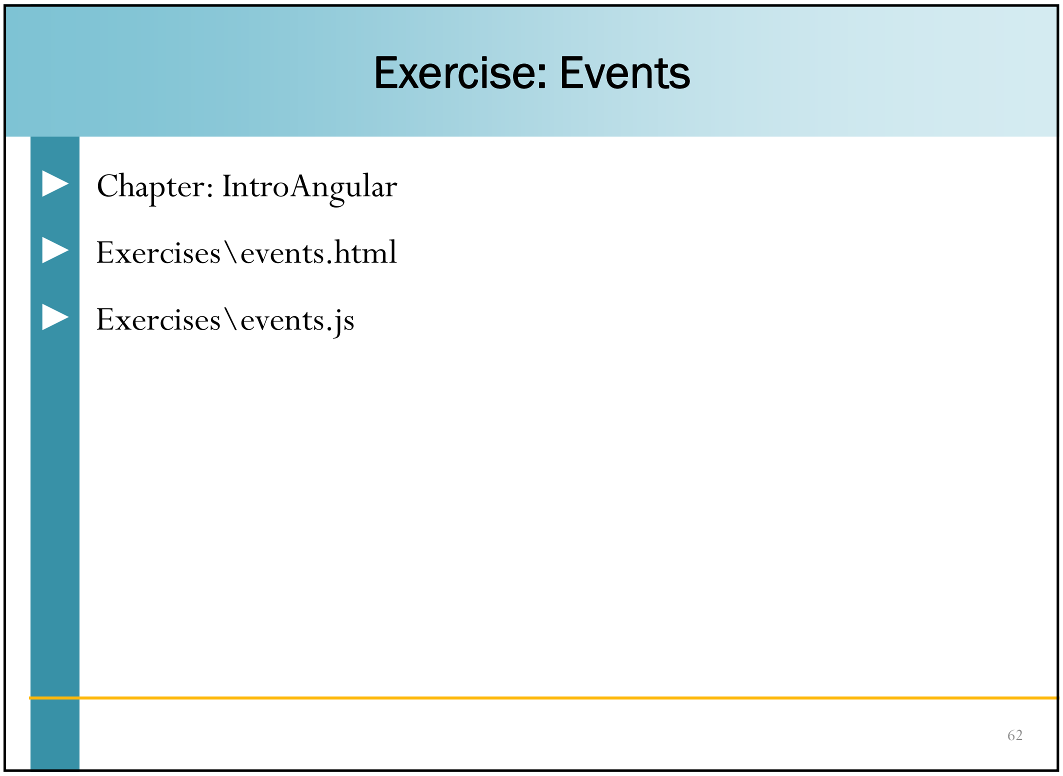
60



Demo: Select lists

- Chapter: IntroAngular
- Select lists with objects
  - Demos/ng-options-objects.html
  - Demos/ng-options-objects.js
- Select lists with arrays of strings
  - Demos/ng-options.html
  - Demos/ng-options.js

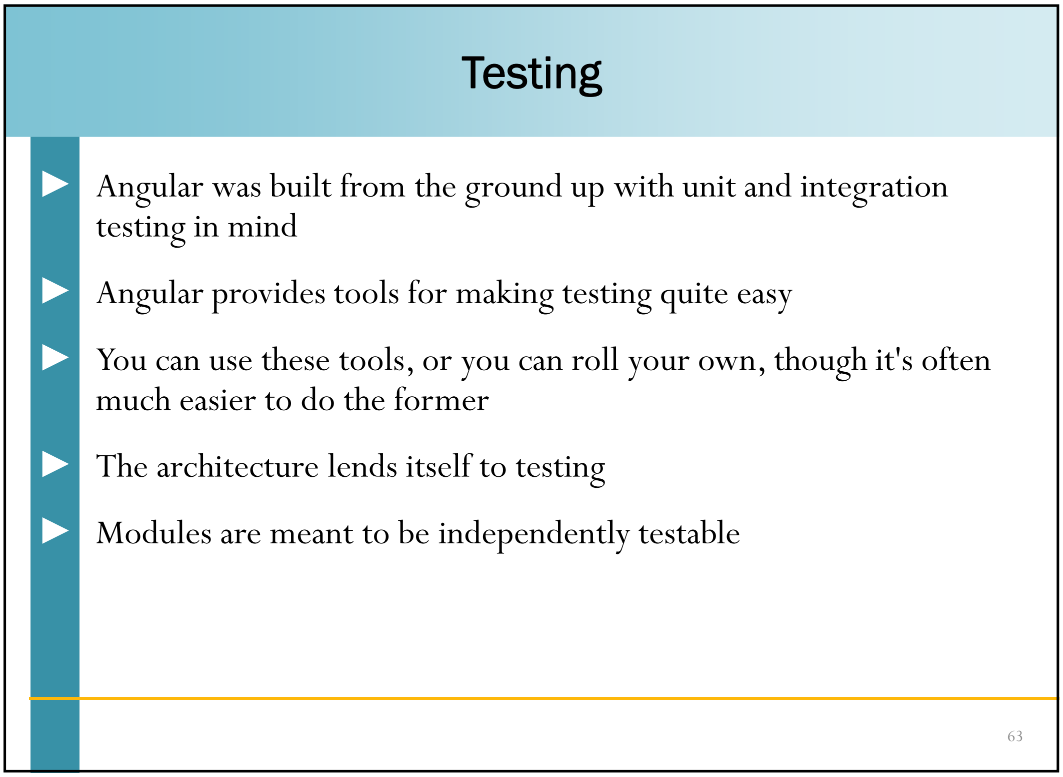
61



Exercise: Events

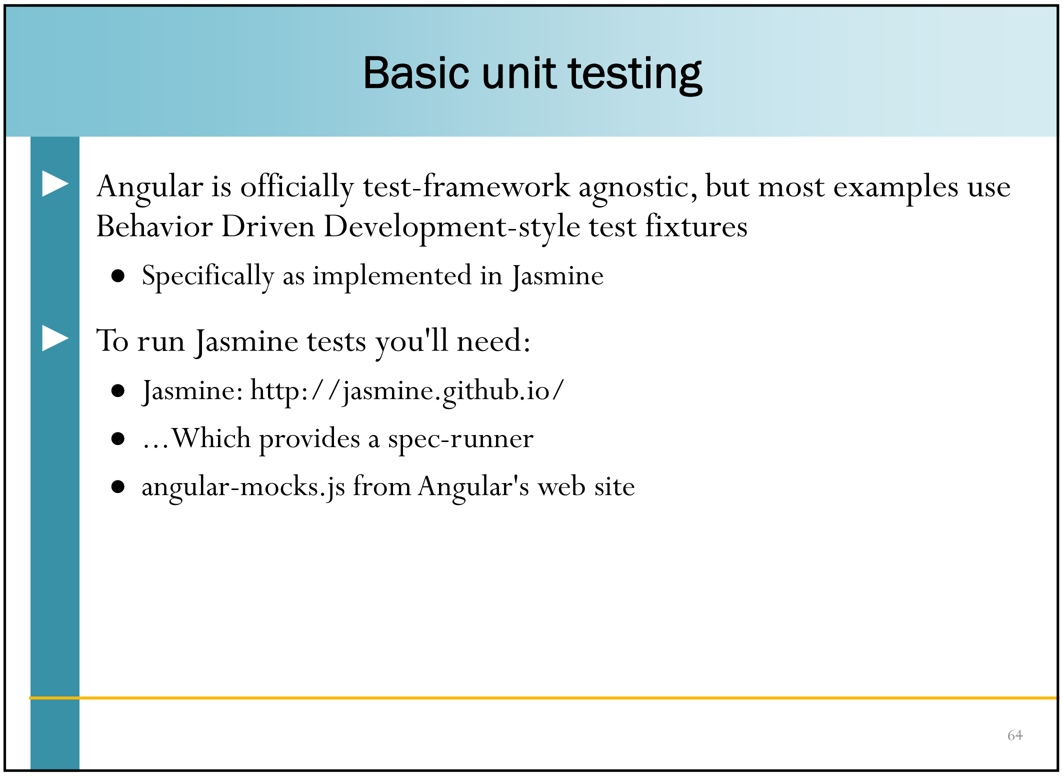
- Chapter: IntroAngular
- Exercises\events.html
- Exercises\events.js

62



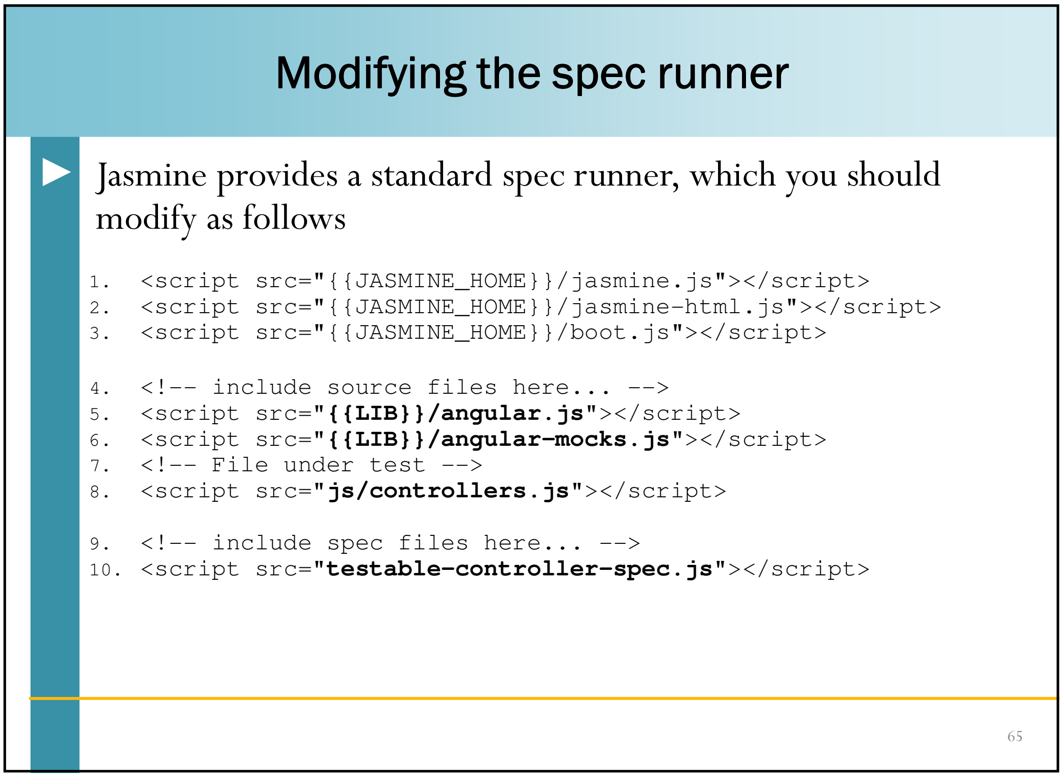
Testing

- Angular was built from the ground up with unit and integration testing in mind
- Angular provides tools for making testing quite easy
- You can use these tools, or you can roll your own, though it's often much easier to do the former
- The architecture lends itself to testing
- Modules are meant to be independently testable



Basic unit testing

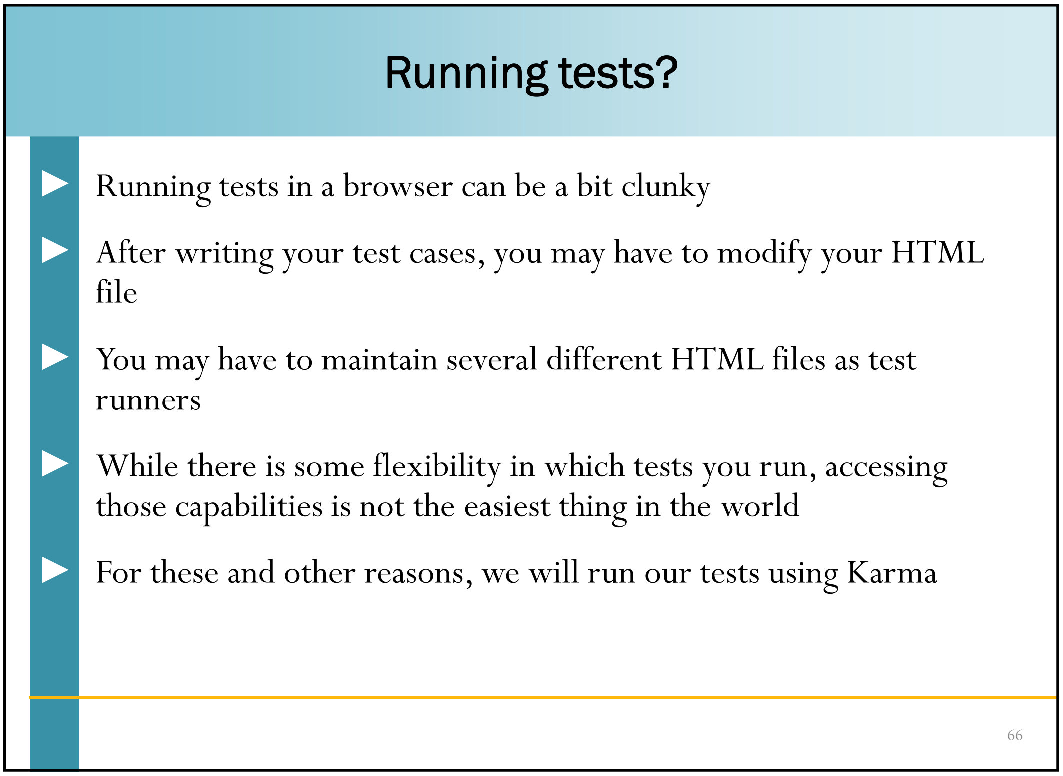
- Angular is officially test-framework agnostic, but most examples use Behavior Driven Development-style test fixtures
  - Specifically as implemented in Jasmine
- To run Jasmine tests you'll need:
  - Jasmine: http://jasmine.github.io/
  - …Which provides a spec-runner
  - angular-mocks.js from Angular's web site



Modifying the spec runner

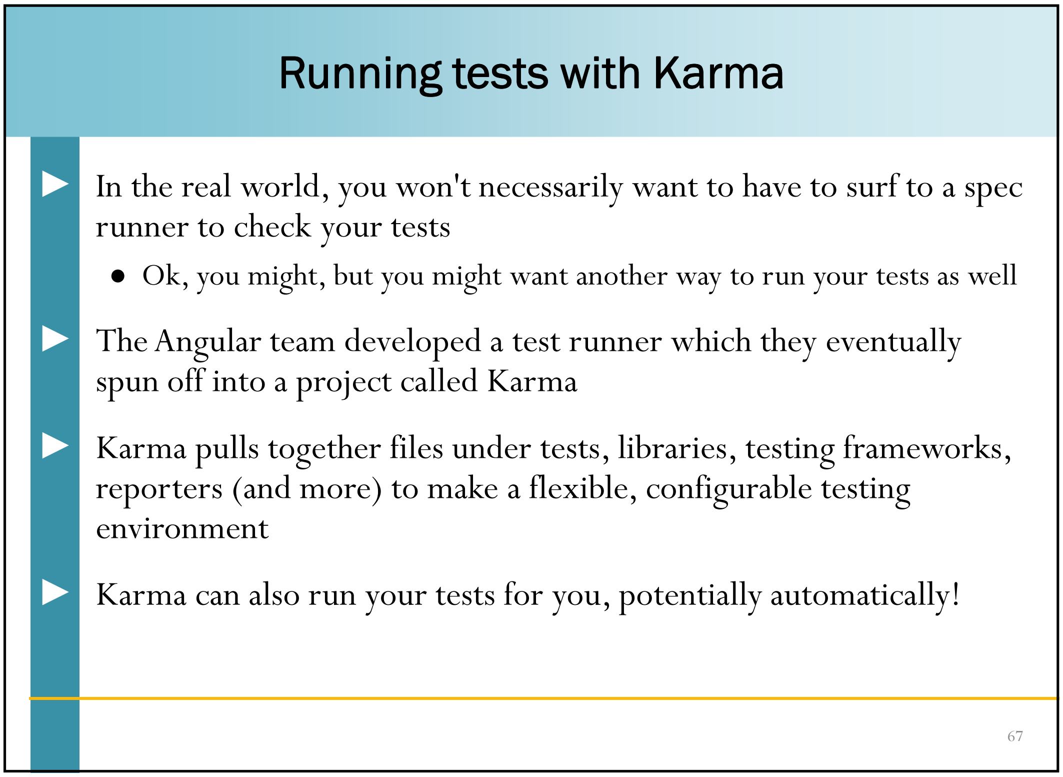
- Jasmine provides a standard spec runner, which you should modify as follows

```
<script src="{{JASMINE_HOME}}/jasmine.js"></script>
<script src="{{JASMINE_HOME}}/jasmine-html.js"></script>
<script src="{{JASMINE_HOME}}/boot.js"></script>

<!-- include source files here... -->
<script src="{{LIB}}/angular.js"></script>
<script src="{{LIB}}/angular-mocks.js"></script>
<!-- File under test -->
<script src="js/controllers.js"></script>

<!-- include spec files here... -->
<script src="testable-controller-spec.js"></script>
```



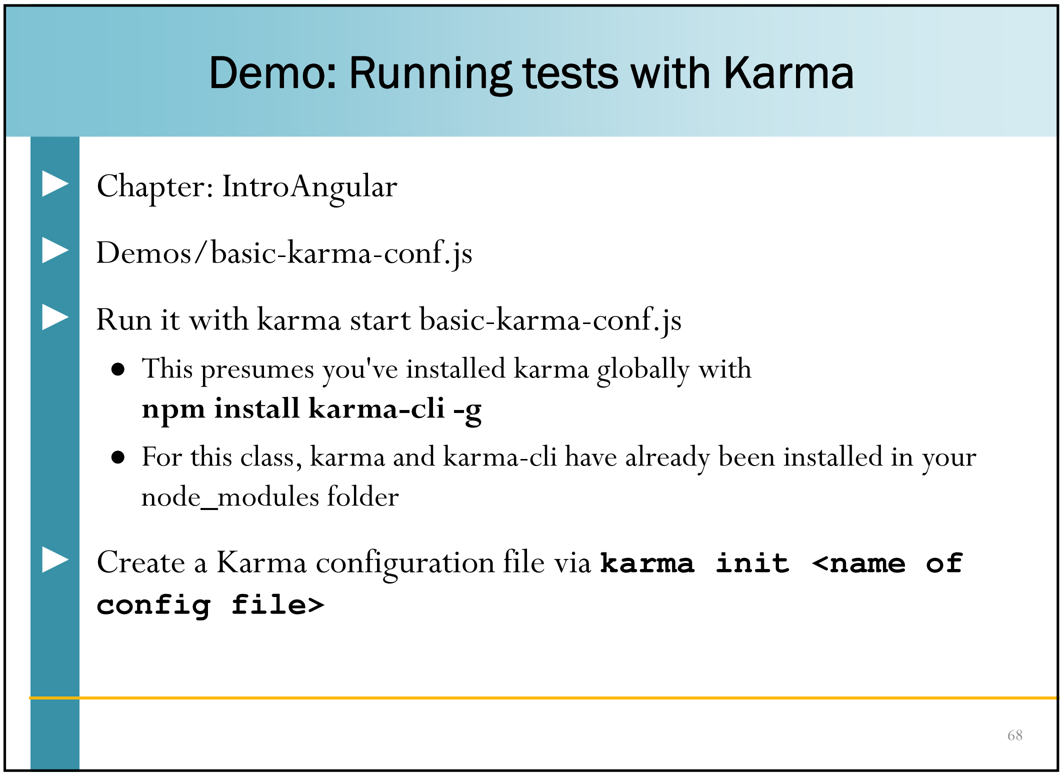
Running tests?

- Running tests in a browser can be a bit clunky
- After writing your test cases, you may have to modify your HTML file
- You may have to maintain several different HTML files as test runners
- While there is some flexibility in which tests you run, accessing those capabilities is not the easiest thing in the world
- For these and other reasons, we will run our tests using Karma



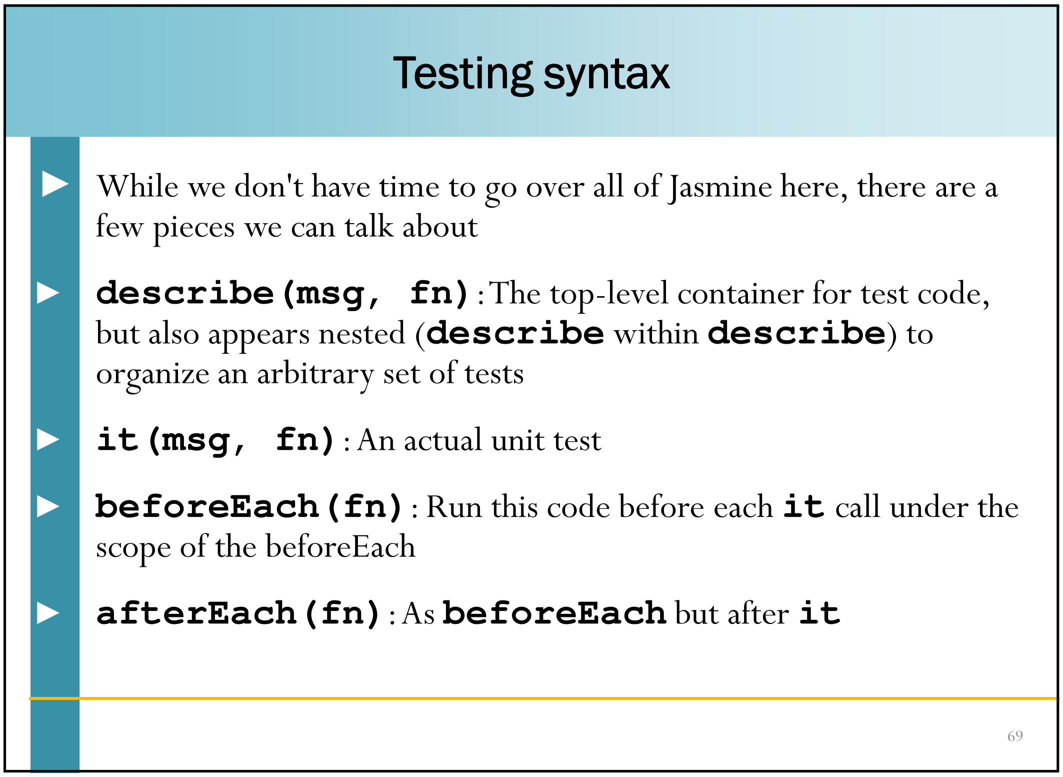
Running tests with Karma

- In the real world, you won't necessarily want to have to surf to a spec runner to check your tests
  - Ok, you might, but you might want another way to run your tests as well
- The Angular team developed a test runner which they eventually spun off into a project called Karma
- Karma pulls together files under tests, libraries, testing frameworks, reporters (and more) to make a flexible, configurable testing environment
- Karma can also run your tests for you, potentially automatically!



Demo: Running tests with Karma

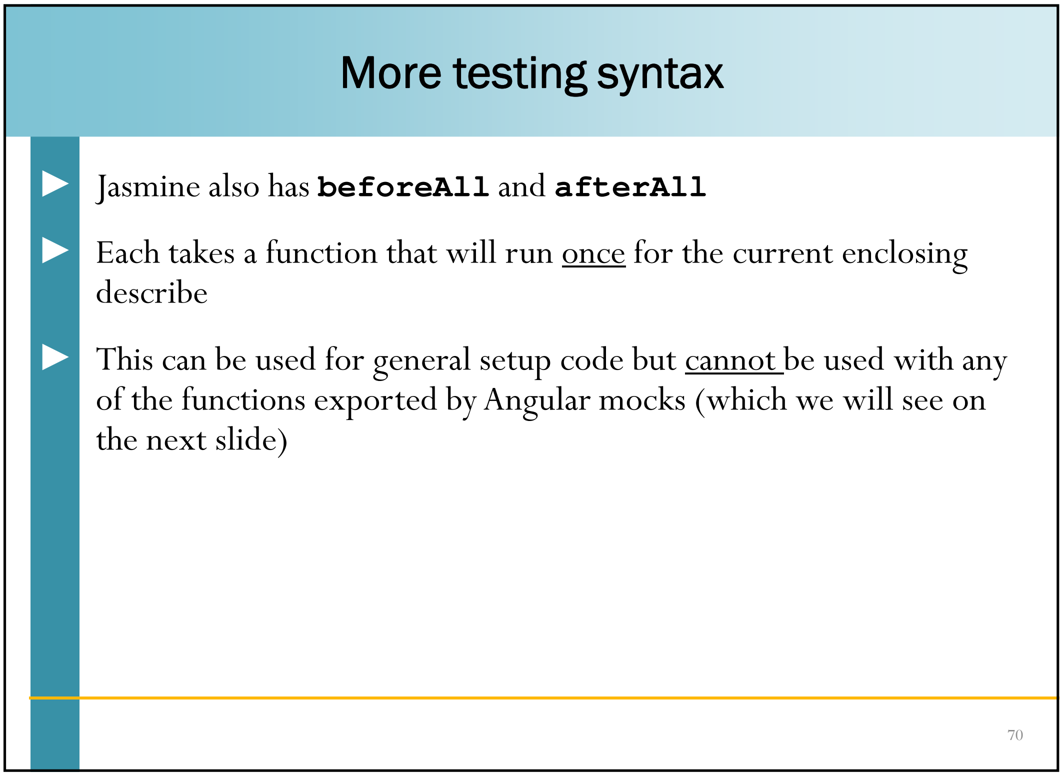
- Chapter: IntroAngular
- Demos/basic-karma-conf.js
- Run it with karma start basic-karma-conf.js
  - This presumes you've installed karma globally with **npm install karma-cli -g**
  - For this class, karma and karma-cli have already been installed in your node_modules folder
- Create a Karma configuration file via **`karma init <name of config file>`**



Testing syntax

- While we don't have time to go over all of Jasmine here, there are a few pieces we can talk about
- **`describe(msg, fn)`** : The top-level container for test code, but also appears nested (**`describe`** within **`describe`**) to organize an arbitrary set of tests
- **`it(msg, fn)`** : An actual unit test
- **`beforeEach(fn)`** : Run this code before each **`it`** call under the scope of the beforeEach
- **`afterEach(fn)`** : As **`beforeEach`** but after **`it`**

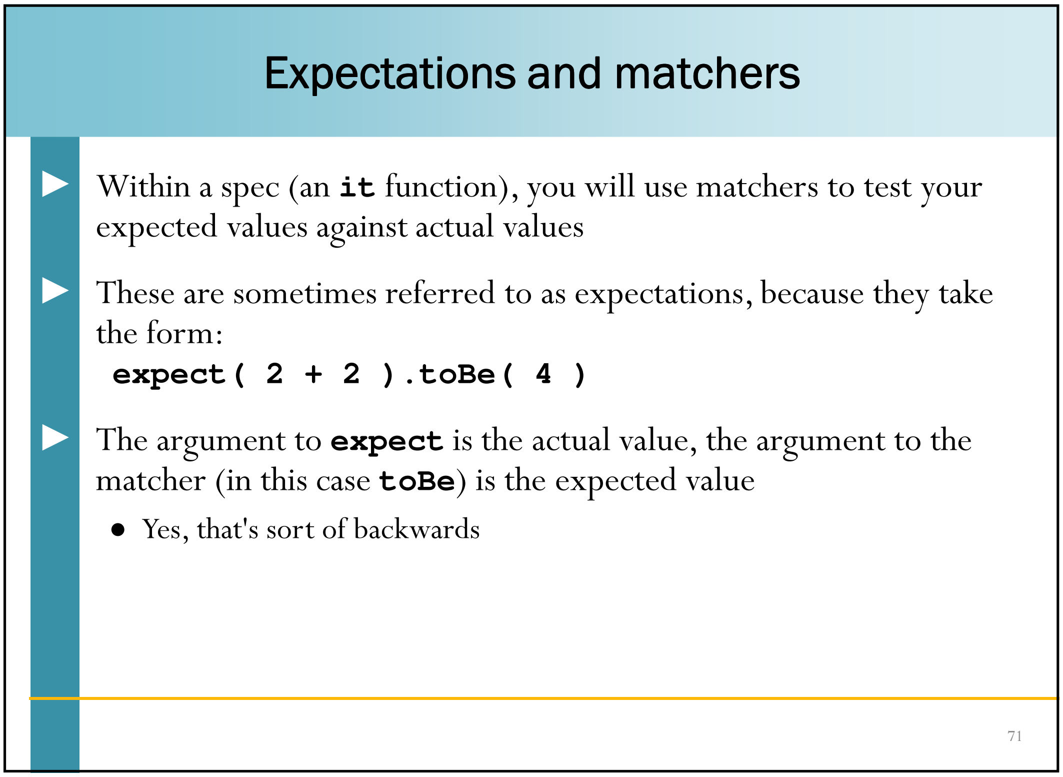
69



More testing syntax

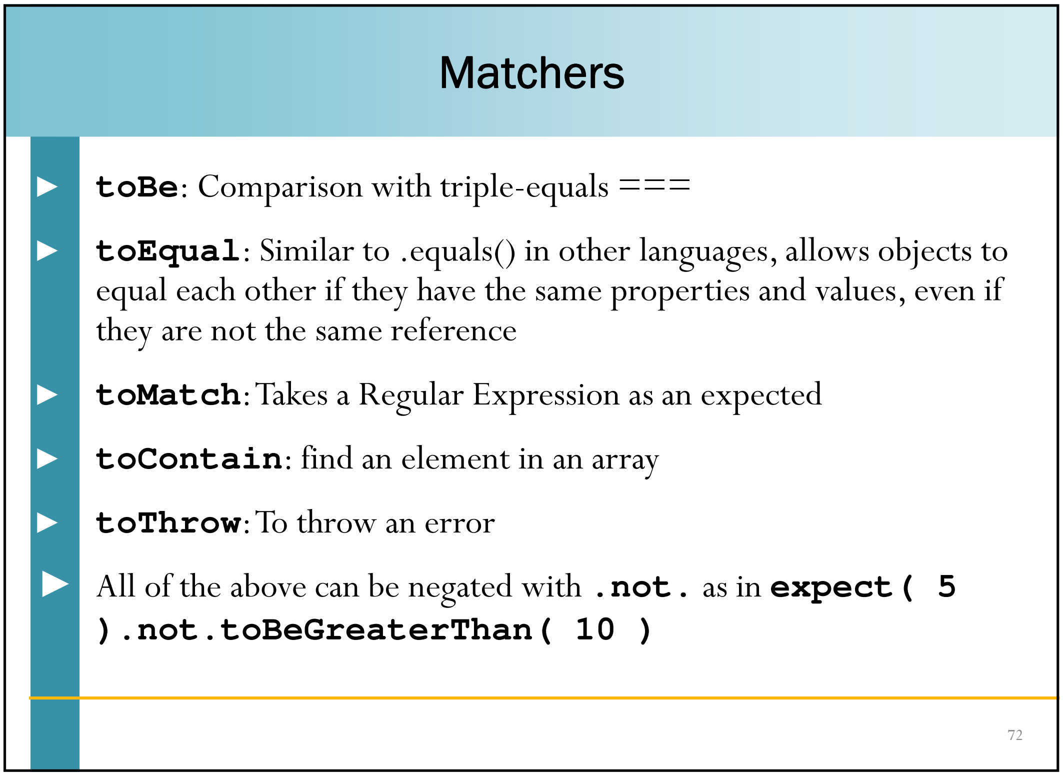
- Jasmine also has **`beforeAll`** and **`afterAll`**
- Each takes a function that will run once for the current enclosing describe
- This can be used for general setup code but cannot be used with any of the functions exported by Angular mocks (which we will see on the next slide)

70



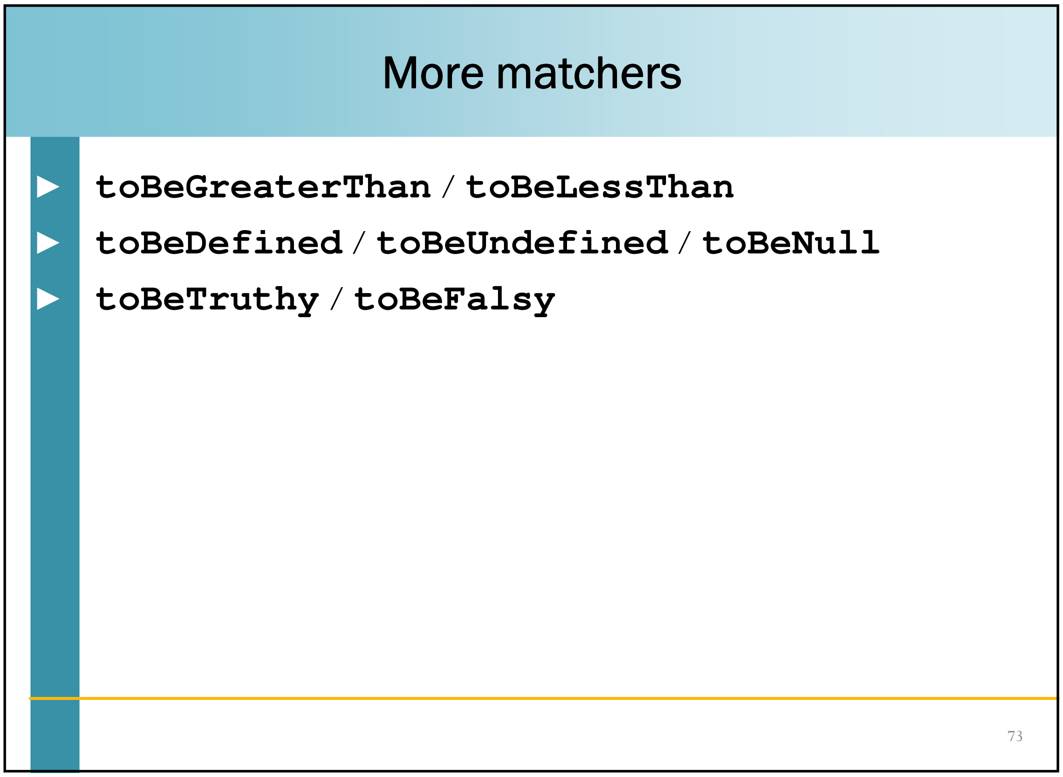
Expectations and matchers

- Within a spec (an **`it`** function), you will use matchers to test your expected values against actual values
- These are sometimes referred to as expectations, because they take the form:
  `expect( 2 + 2 ).toBe( 4 )`
- The argument to **`expect`** is the actual value, the argument to the matcher (in this case **`toBe`**) is the expected value
  - Yes, that's sort of backwards



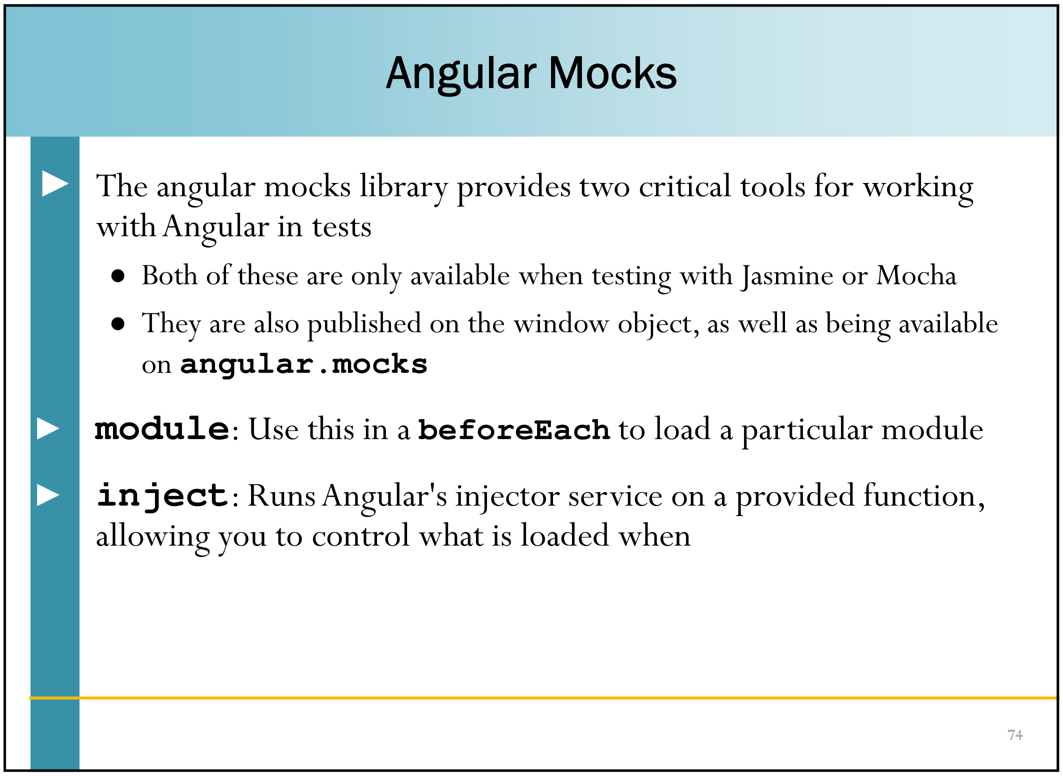
Matchers

- **`toBe`**: Comparison with triple-equals ===
- **`toEqual`**: Similar to .equals() in other languages, allows objects to equal each other if they have the same properties and values, even if they are not the same reference
- **`toMatch`**: Takes a Regular Expression as an expected
- **`toContain`**: find an element in an array
- **`toThrow`**: To throw an error
- All of the above can be negated with **`.not.`** as in **`expect( 5 ).not.toBeGreaterThan( 10 )`**



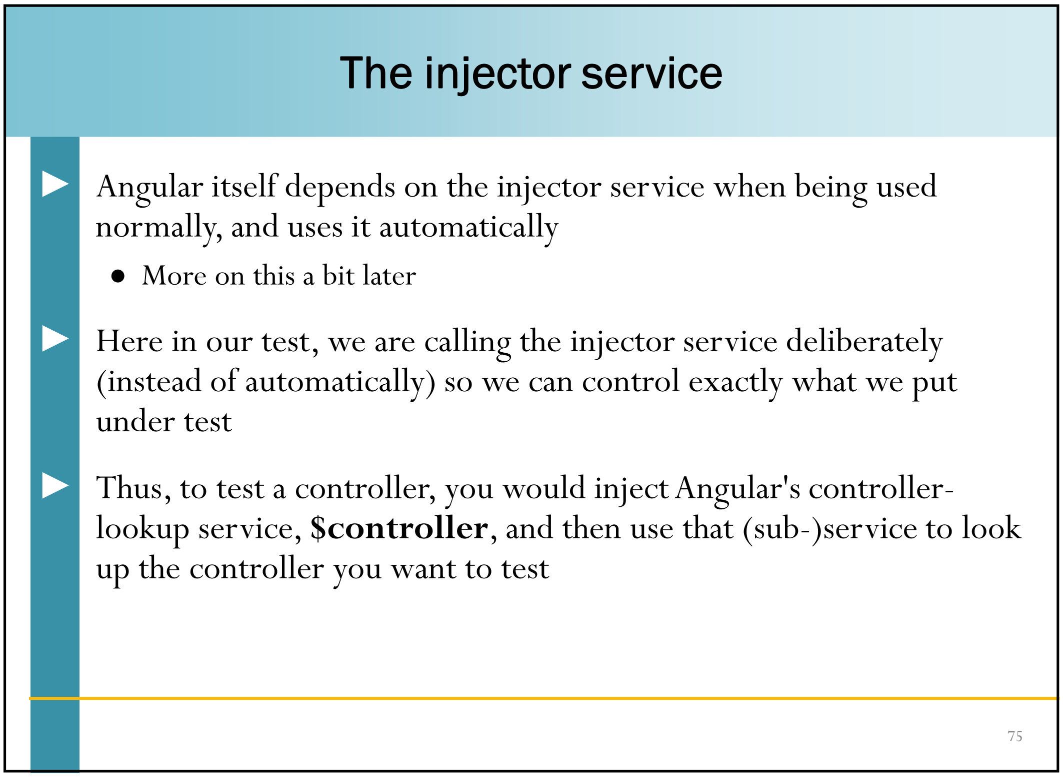
More matchers

- **`toBeGreaterThan`** / **`toBeLessThan`**
- **`toBeDefined`** / **`toBeUndefined`** / **`toBeNull`**
- **`toBeTruthy`** / **`toBeFalsy`**



Angular Mocks

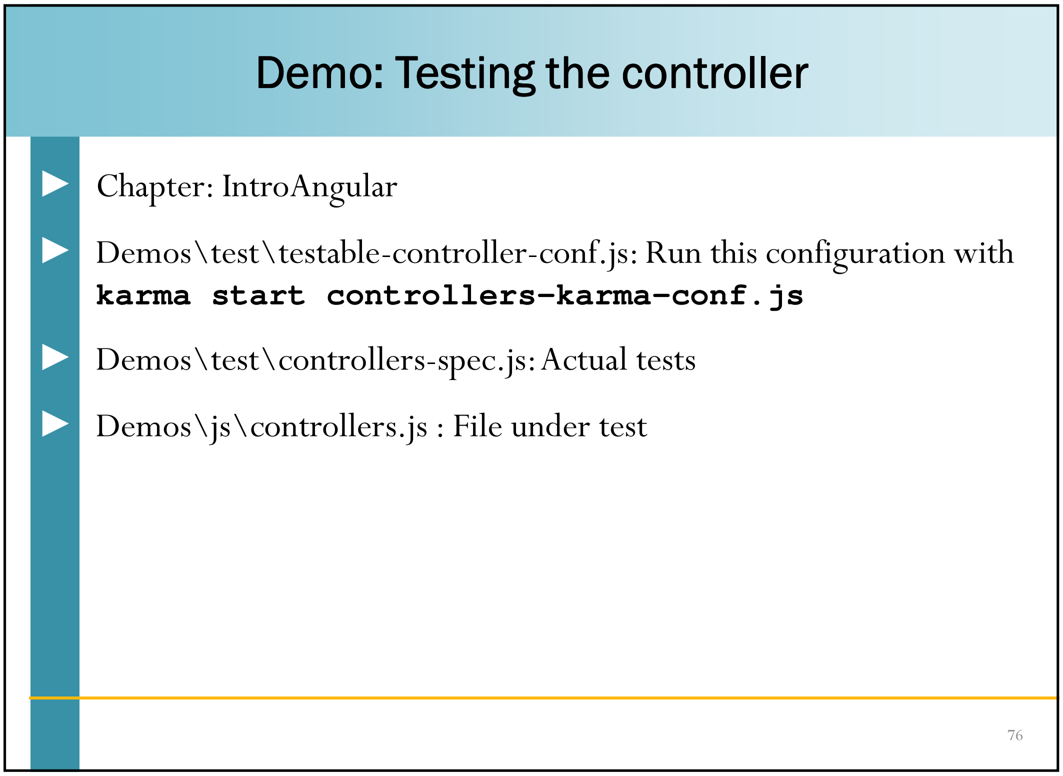
- The angular mocks library provides two critical tools for working with Angular in tests
  - Both of these are only available when testing with Jasmine or Mocha
  - They are also published on the window object, as well as being available on **`angular.mocks`**
- **`module`**: Use this in a **`beforeEach`** to load a particular module
- **`inject`**: Runs Angular's injector service on a provided function, allowing you to control what is loaded when



The injector service

- Angular itself depends on the injector service when being used normally, and uses it automatically
  - More on this a bit later
- Here in our test, we are calling the injector service deliberately (instead of automatically) so we can control exactly what we put under test
- Thus, to test a controller, you would inject Angular's controller-lookup service, **$controller**, and then use that (sub-)service to look up the controller you want to test

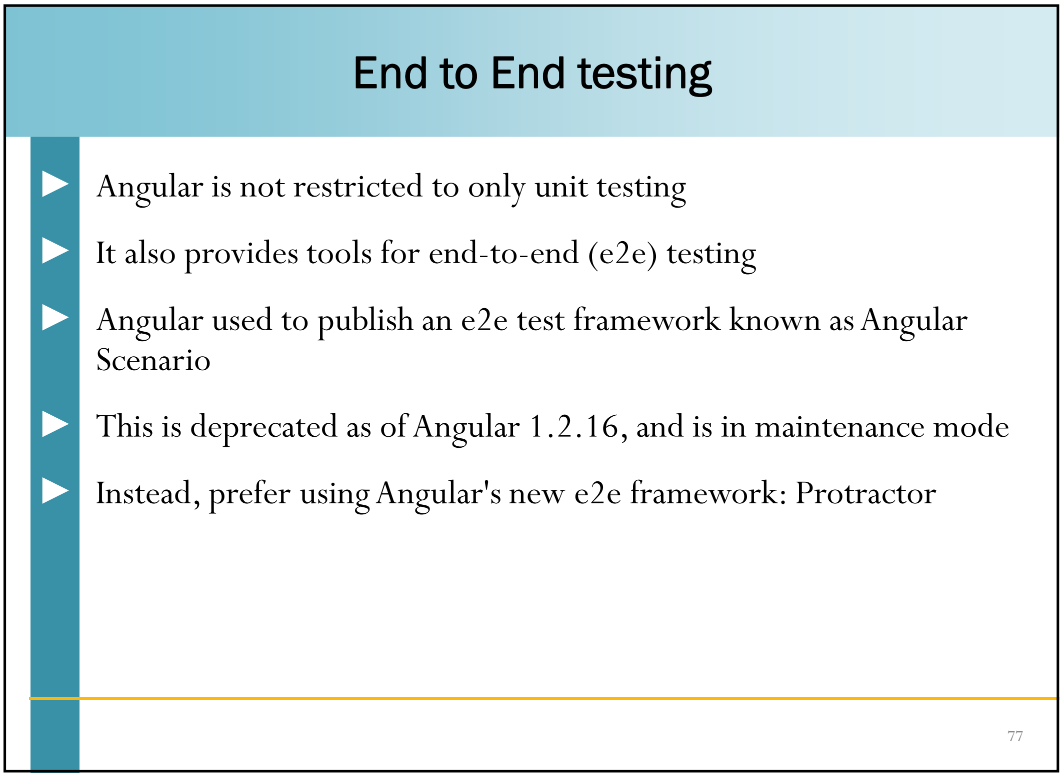
75



Demo: Testing the controller

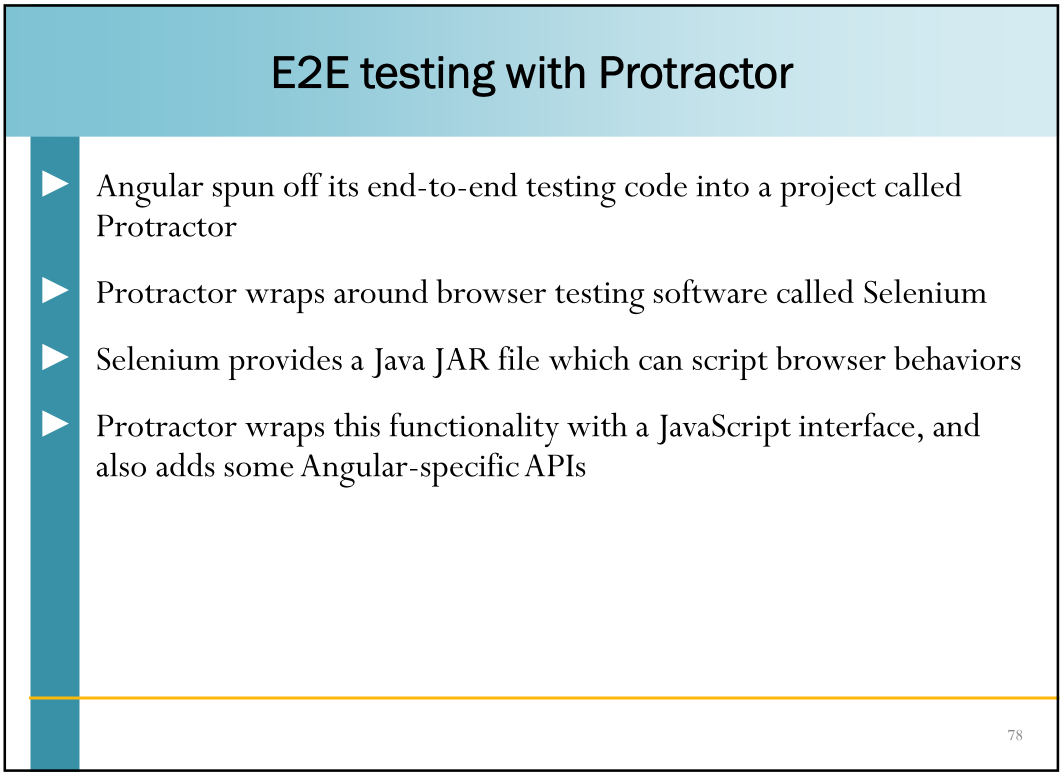
- Chapter: IntroAngular
- Demos\test\testable-controller-conf.js: Run this configuration with **`karma start controllers-karma-conf.js`**
- Demos\test\controllers-spec.js: Actual tests
- Demos\js\controllers.js : File under test

76



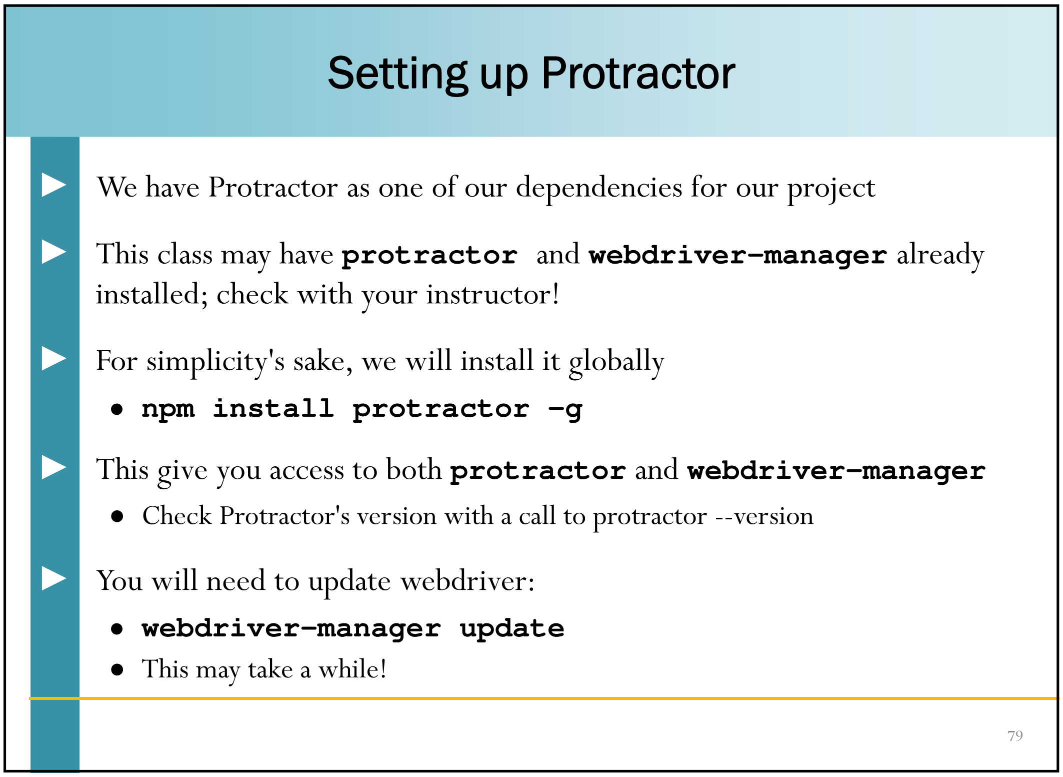
End to End testing

- Angular is not restricted to only unit testing
- It also provides tools for end-to-end (e2e) testing
- Angular used to publish an e2e test framework known as Angular Scenario
- This is deprecated as of Angular 1.2.16, and is in maintenance mode
- Instead, prefer using Angular's new e2e framework: Protractor



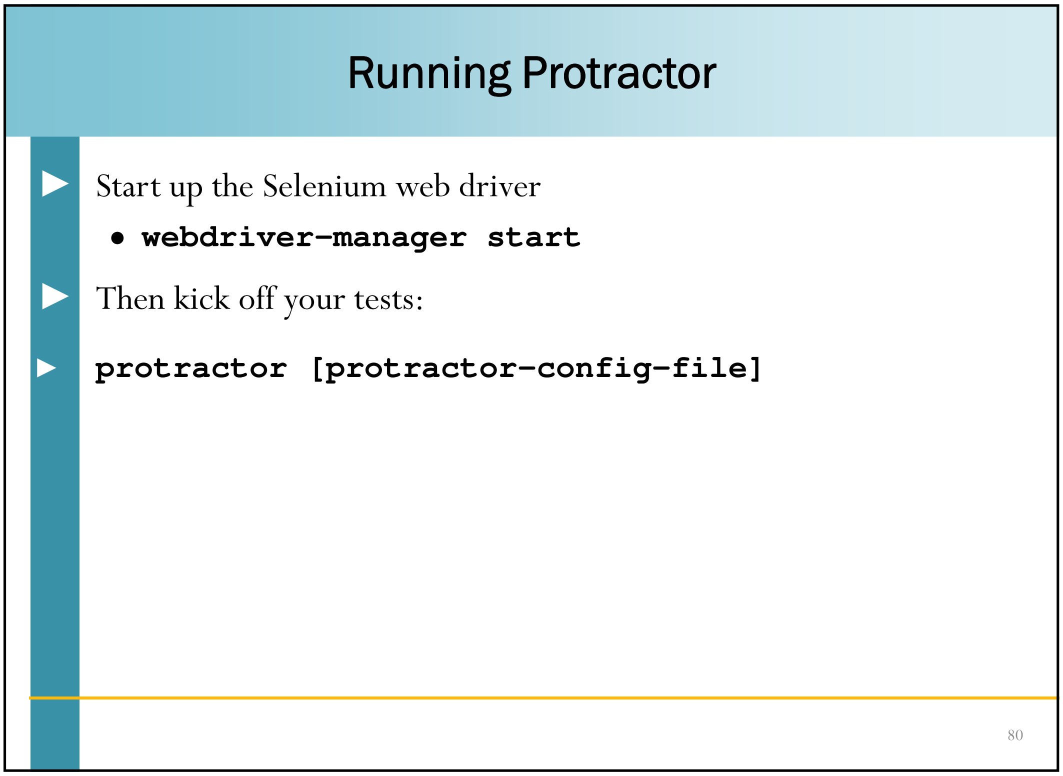
E2E testing with Protractor

- Angular spun off its end-to-end testing code into a project called Protractor
- Protractor wraps around browser testing software called Selenium
- Selenium provides a Java JAR file which can script browser behaviors
- Protractor wraps this functionality with a JavaScript interface, and also adds some Angular-specific APIs



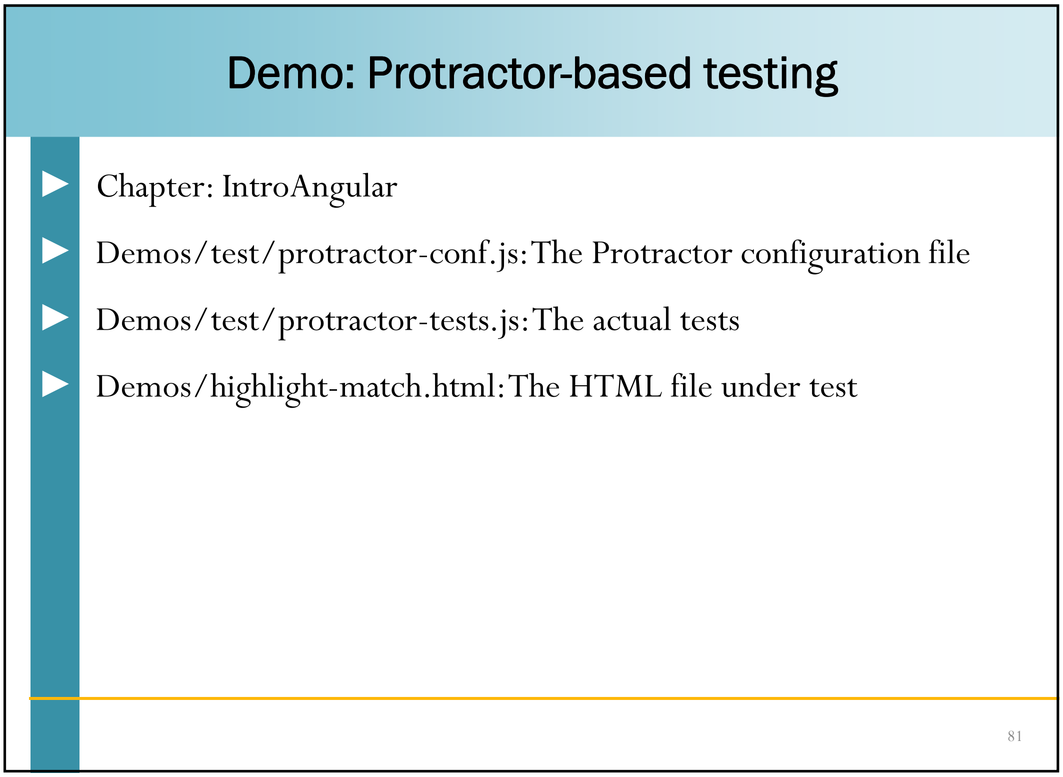
Setting up Protractor

- We have Protractor as one of our dependencies for our project
- This class may have **`protractor`** and **`webdriver-manager`** already installed; check with your instructor!
- For simplicity's sake, we will install it globally
  - **`npm install protractor -g`**
- This give you access to both **`protractor`** and **`webdriver-manager`**
  - Check Protractor's version with a call to protractor --version
- You will need to update webdriver:
  - **`webdriver-manager update`**
  - This may take a while!



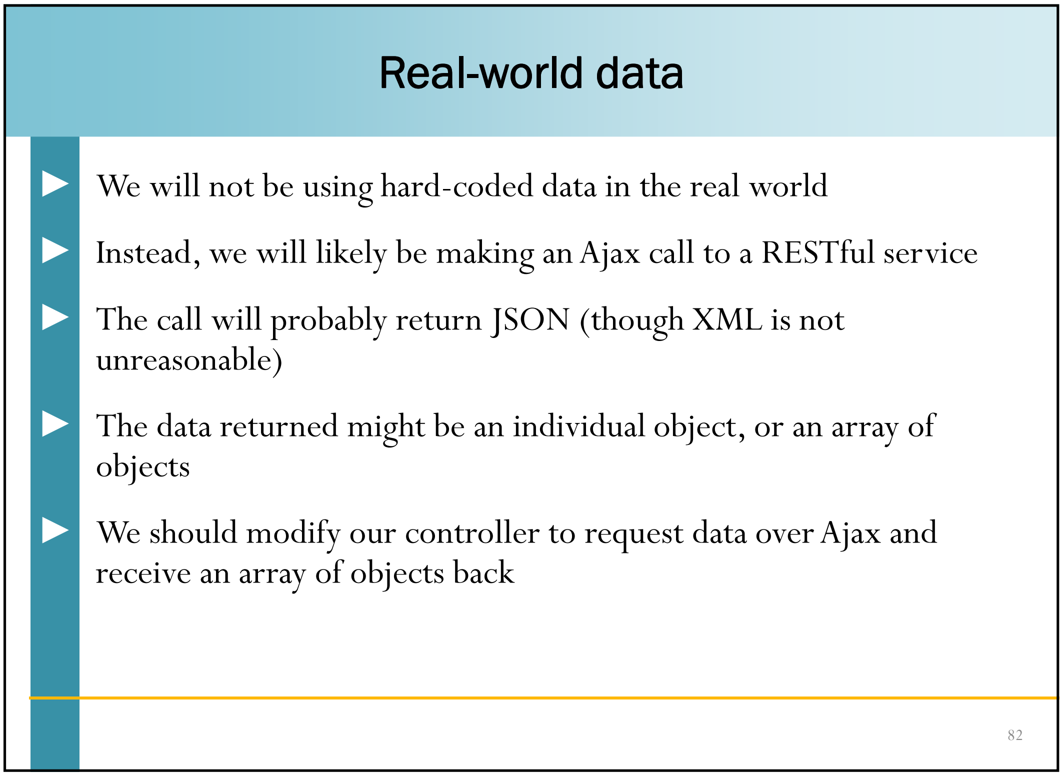
Running Protractor

- Start up the Selenium web driver
  - **`webdriver-manager start`**
- Then kick off your tests:
- **`protractor [protractor-config-file]`**



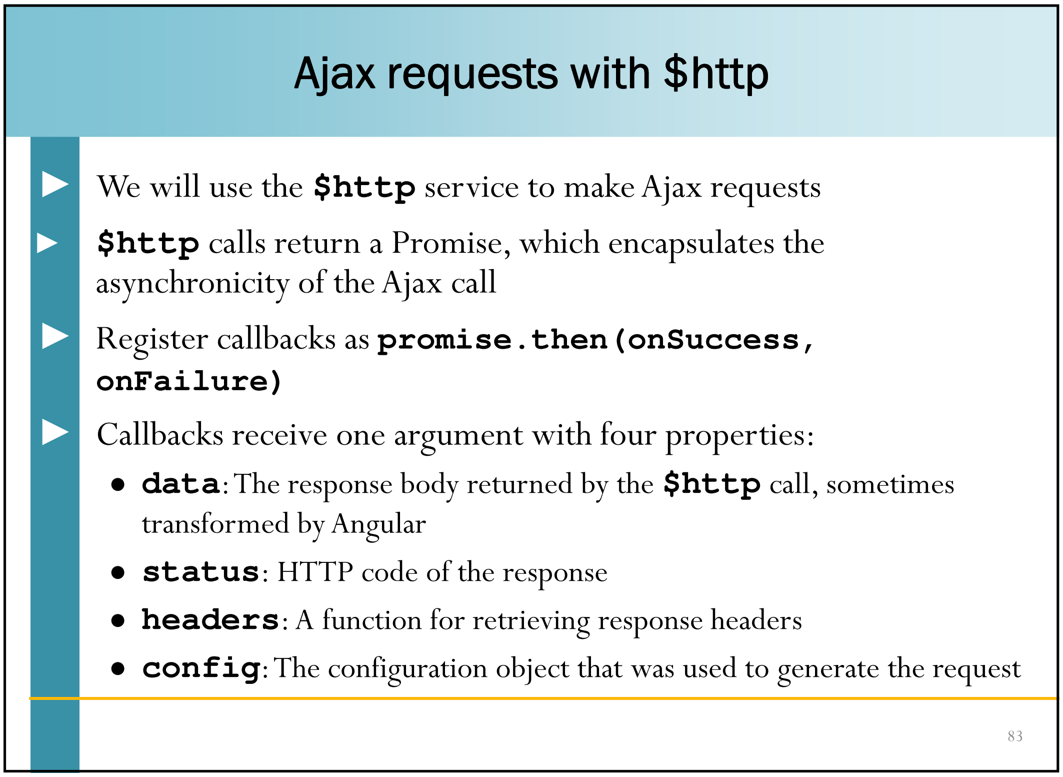
Demo: Protractor-based testing

- Chapter: IntroAngular
- Demos/test/protractor-conf.js: The Protractor configuration file
- Demos/test/protractor-tests.js: The actual tests
- Demos/highlight-match.html: The HTML file under test



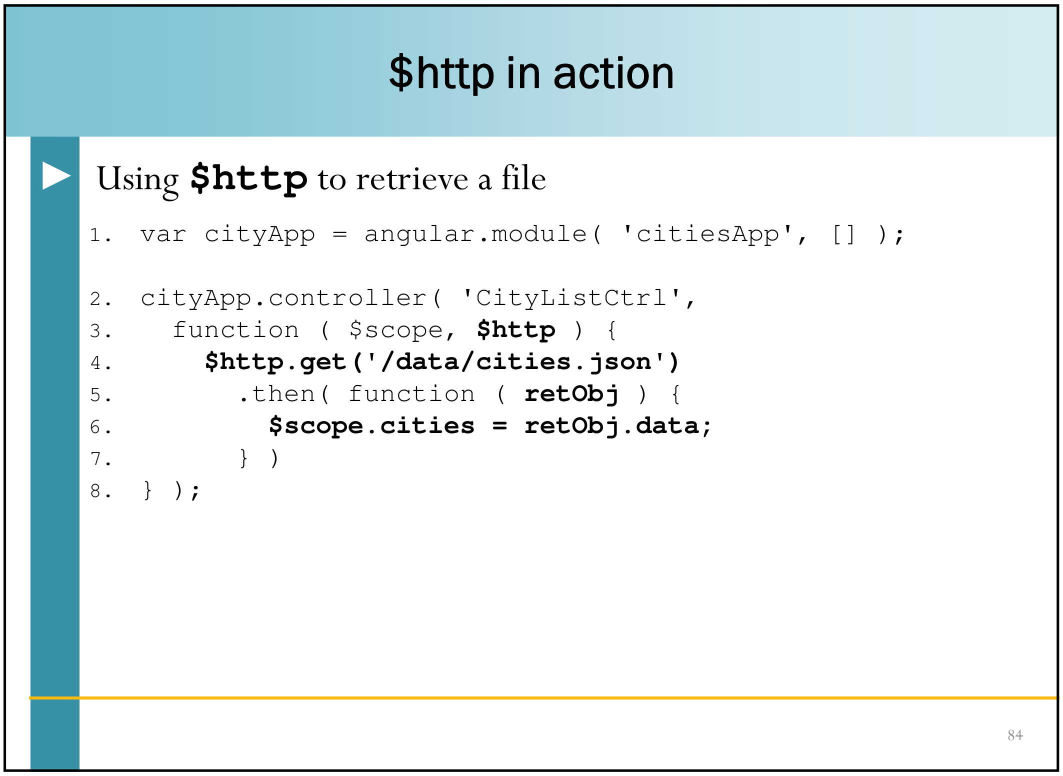
Real-world data

- We will not be using hard-coded data in the real world
- Instead, we will likely be making an Ajax call to a RESTful service
- The call will probably return JSON (though XML is not unreasonable)
- The data returned might be an individual object, or an array of objects
- We should modify our controller to request data over Ajax and receive an array of objects back



Ajax requests with $http

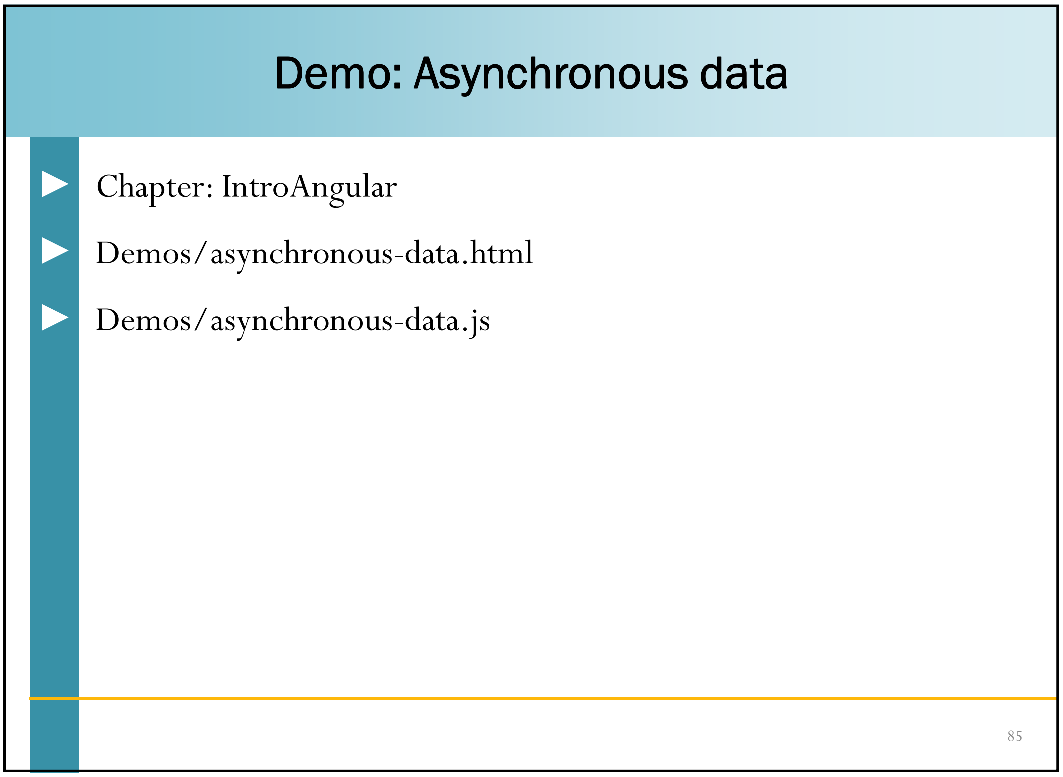
- We will use the **`$http`** service to make Ajax requests
- **`$http`** calls return a Promise, which encapsulates the asynchronicity of the Ajax call
- Register callbacks as **`promise.then(onSuccess, onFailure)`**
- Callbacks receive one argument with four properties:
  - **`data`**: The response body returned by the **`$http`** call, sometimes transformed by Angular
  - **`status`**: HTTP code of the response
  - **`headers`**: A function for retrieving response headers
  - **`config`**: The configuration object that was used to generate the request



$http in action

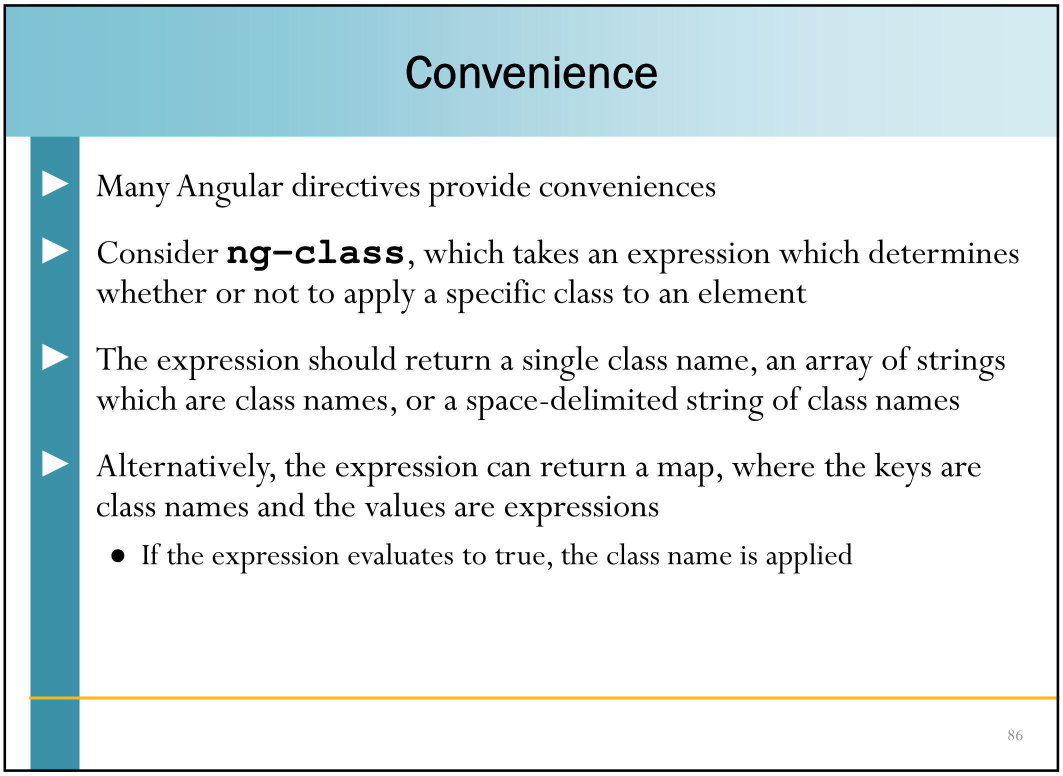
- Using **`$http`** to retrieve a file

```
var cityApp = angular.module( 'citiesApp', [] );

cityApp.controller( 'CityListCtrl',
  function ( $scope, $http ) {
    $http.get('/data/cities.json')
      .then( function ( retObj ) {
        $scope.cities = retObj.data;
      } )
} );
```



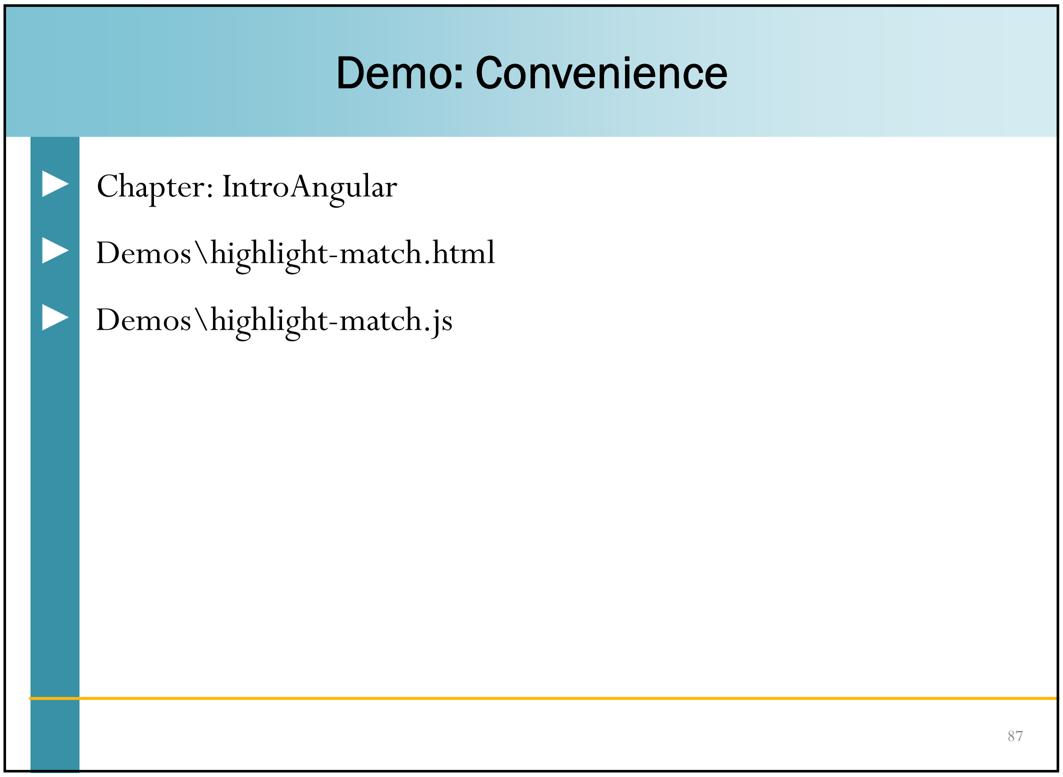
Demo: Asynchronous data

- Chapter: IntroAngular
- Demos/asynchronous-data.html
- Demos/asynchronous-data.js



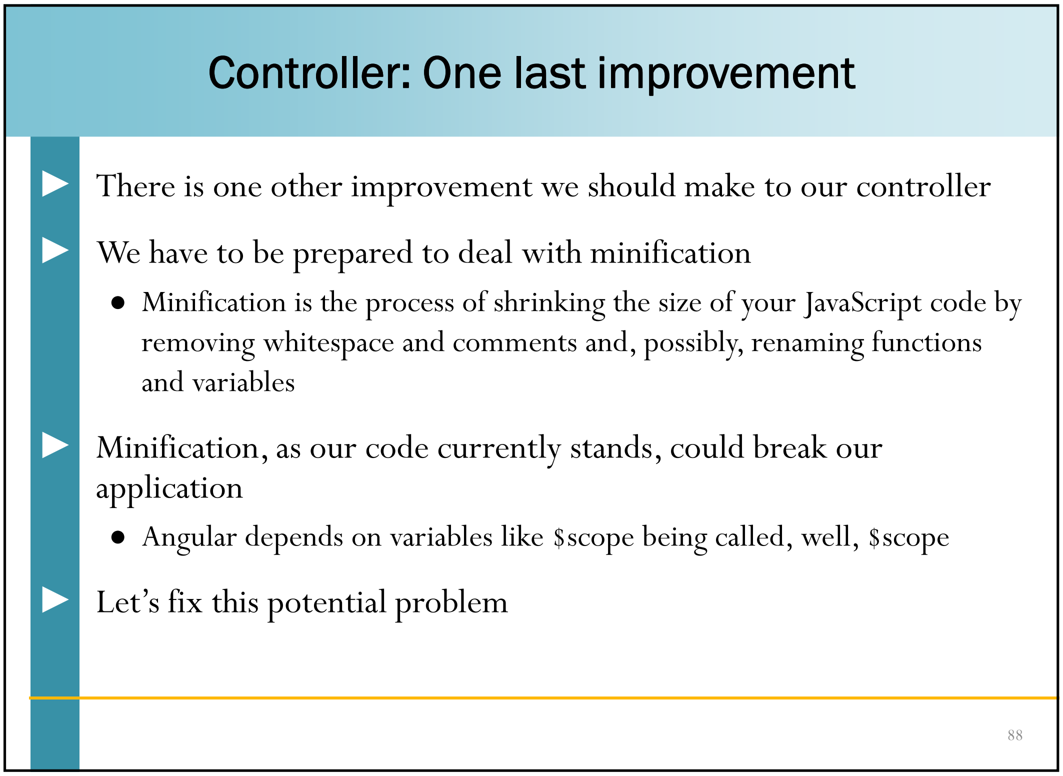
Convenience

- Many Angular directives provide conveniences
- Consider **`ng-class`**, which takes an expression which determines whether or not to apply a specific class to an element
- The expression should return a single class name, an array of strings which are class names, or a space-delimited string of class names
- Alternatively, the expression can return a map, where the keys are class names and the values are expressions
  - If the expression evaluates to true, the class name is applied



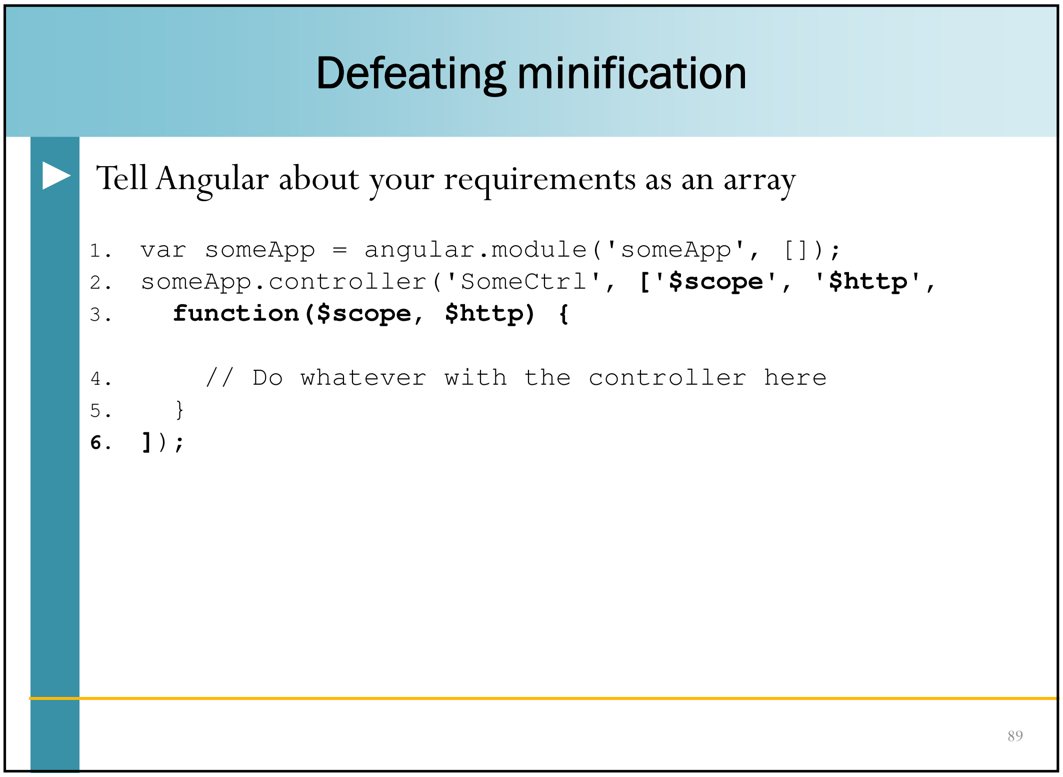
Demo: Convenience

- Chapter: IntroAngular
- Demos\highlight-match.html
- Demos\highlight-match.js



Controller: One last improvement

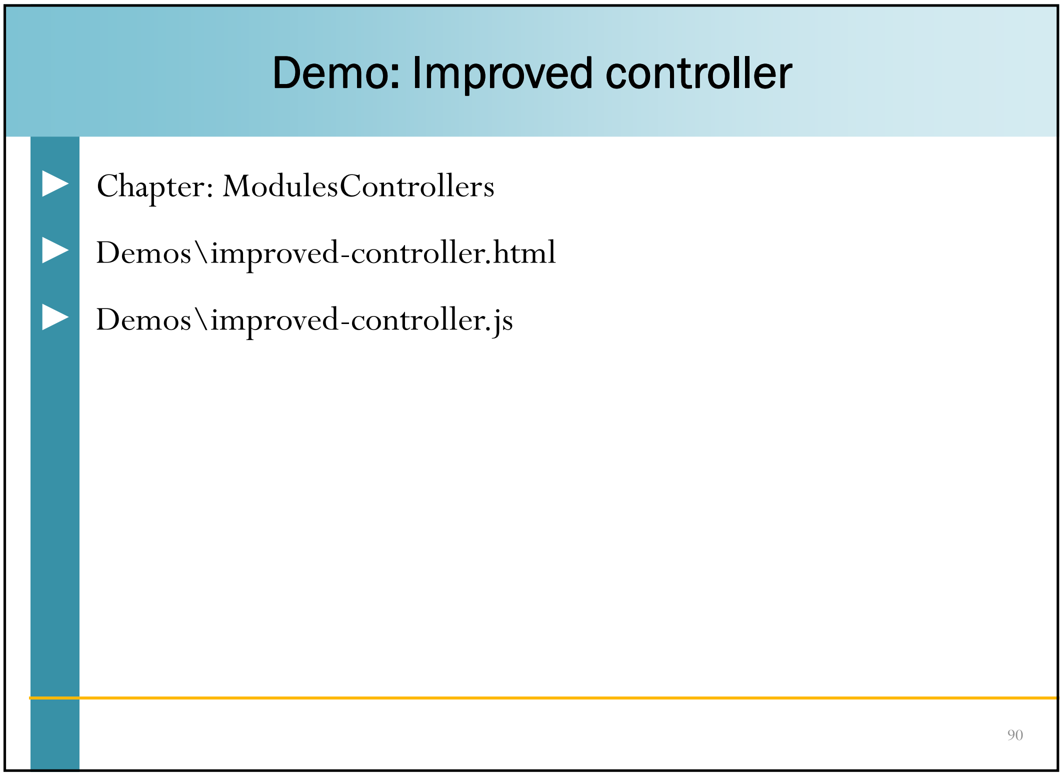
- There is one other improvement we should make to our controller
- We have to be prepared to deal with minification
  - Minification is the process of shrinking the size of your JavaScript code by removing whitespace and comments and, possibly, renaming functions and variables
- Minification, as our code currently stands, could break our application
  - Angular depends on variables like $scope being called, well, $scope
- Let’s fix this potential problem



Defeating minification

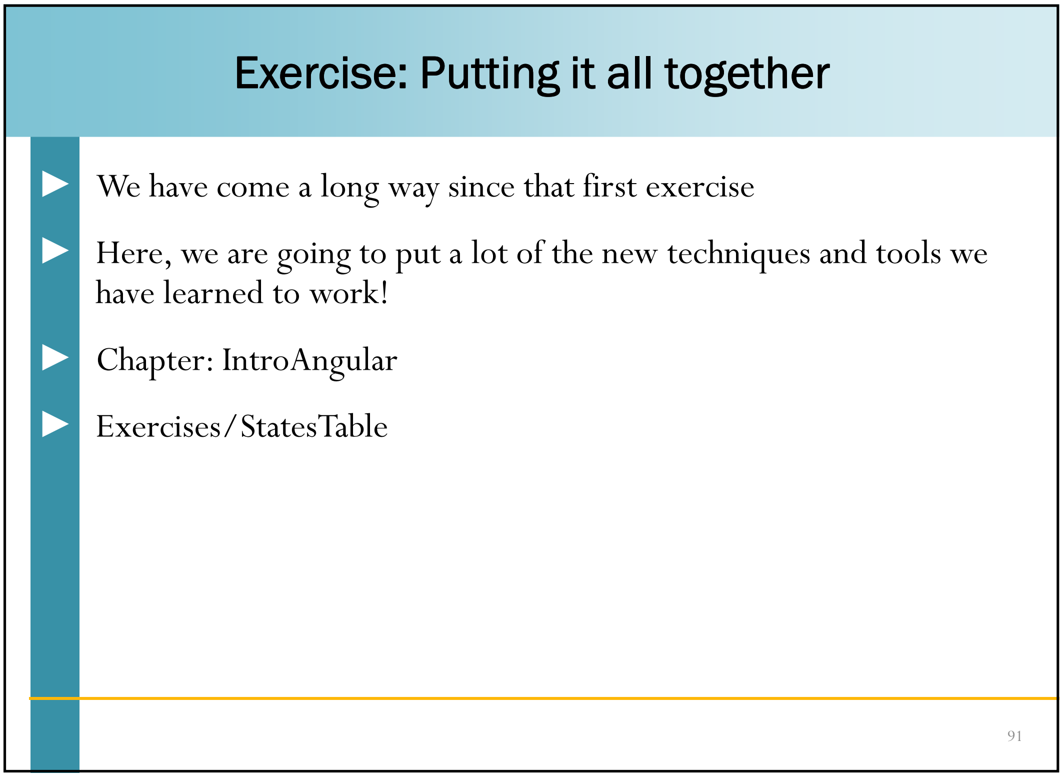
- Tell Angular about your requirements as an array

```
var someApp = angular.module('someApp', []);
someApp.controller('SomeCtrl', ['$scope', '$http',
  function($scope, $http) {

    // Do whatever with the controller here
  }
]);
```



Demo: Improved controller

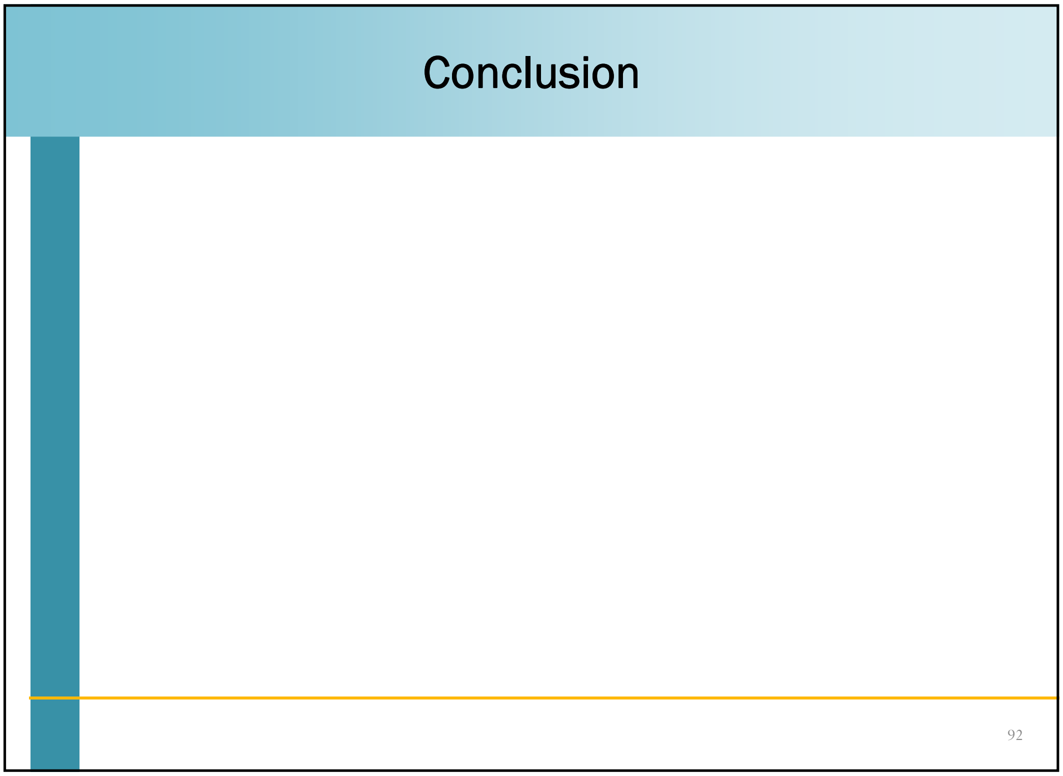
- Chapter: ModulesControllers
- Demos\improved-controller.html
- Demos\improved-controller.js



Exercise: Putting it all together

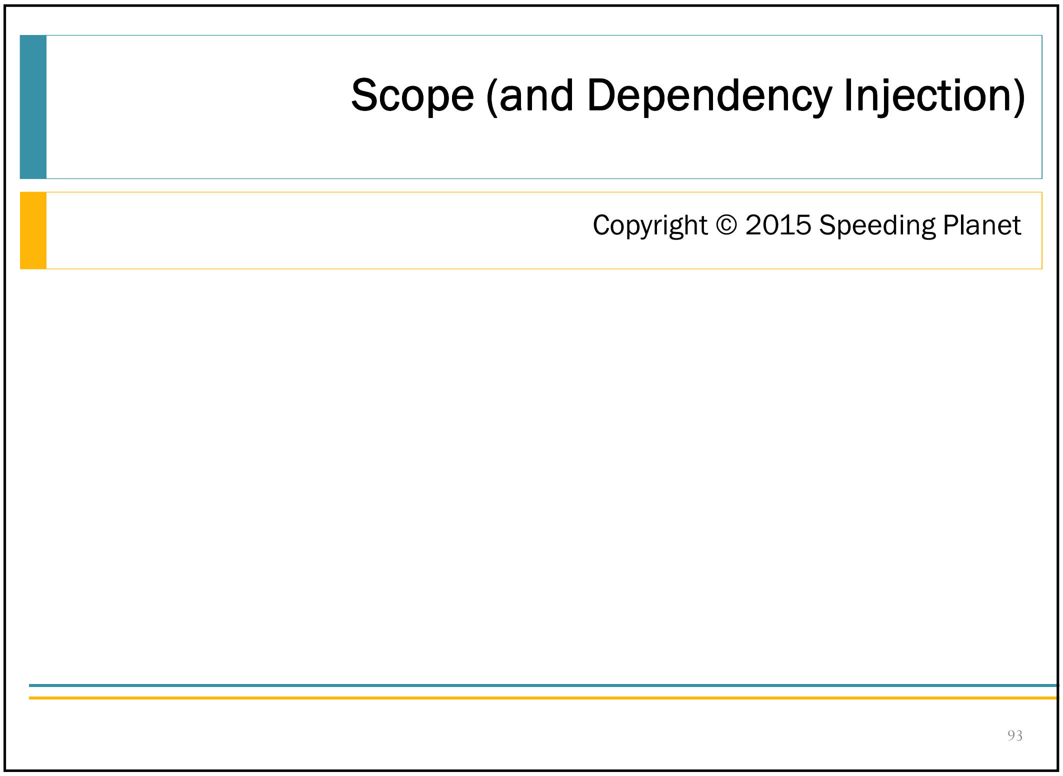
- We have come a long way since that first exercise
- Here, we are going to put a lot of the new techniques and tools we have learned to work!
- Chapter: IntroAngular
- Exercises/StatesTable

91



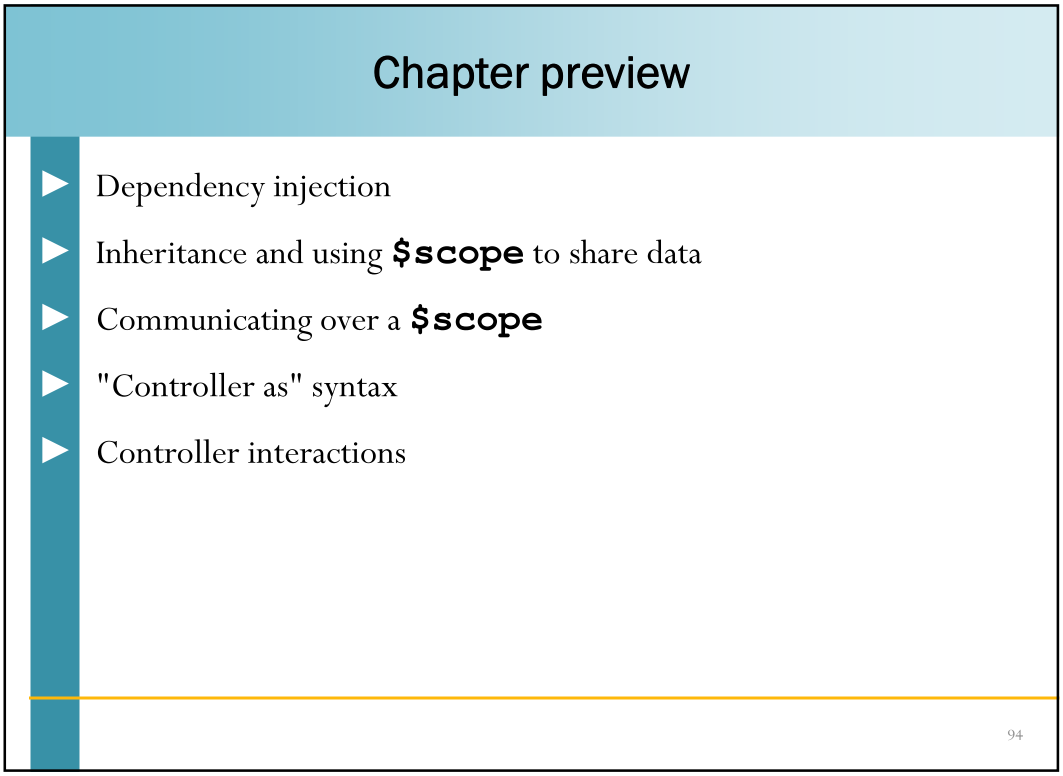
Conclusion

92



Scope (and Dependency Injection)

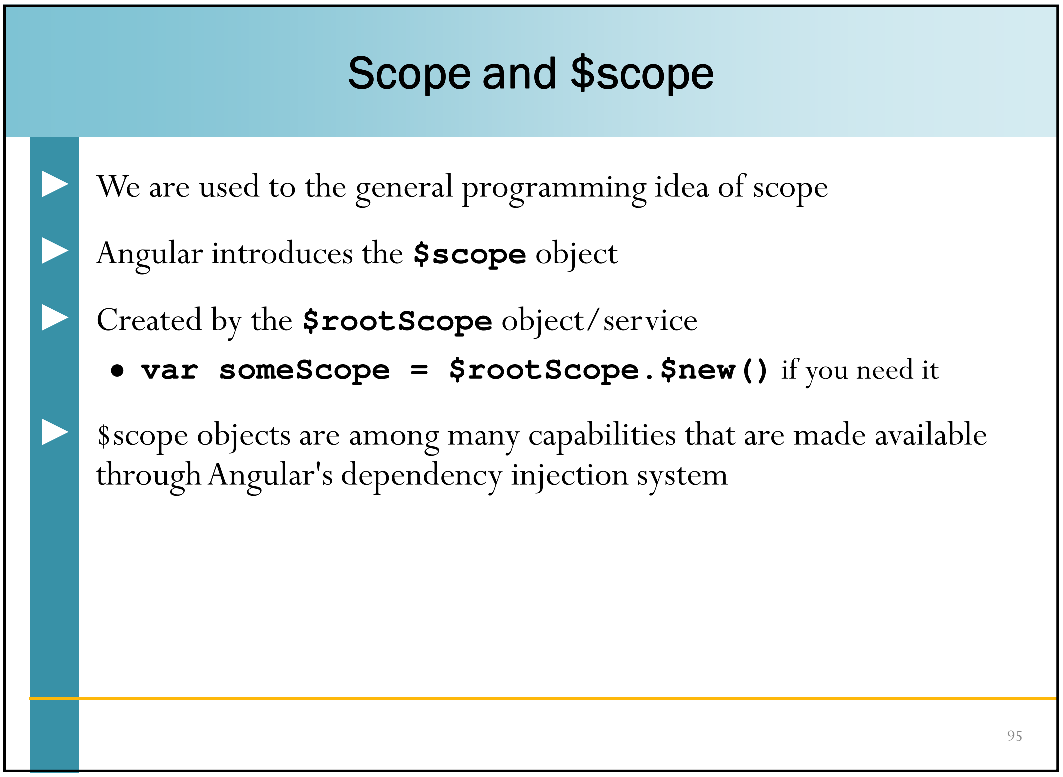
93



Chapter preview

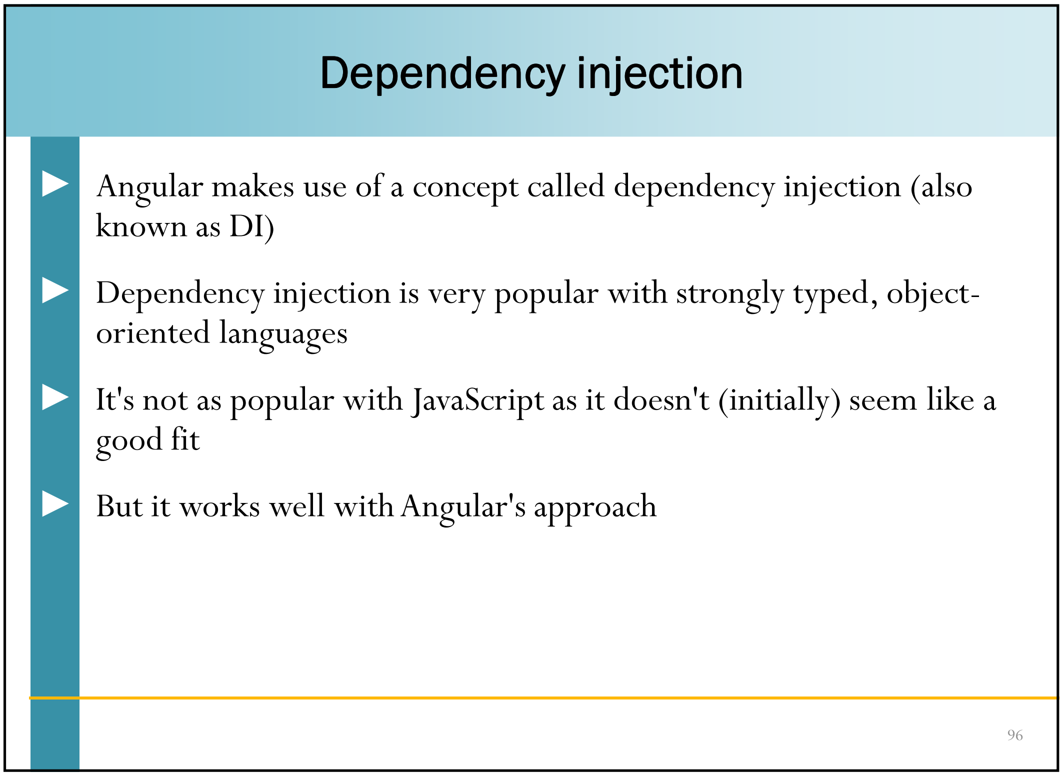
- Dependency injection
- Inheritance and using **`$scope`** to share data
- Communicating over a **`$scope`**
- "Controller as" syntax
- Controller interactions

94



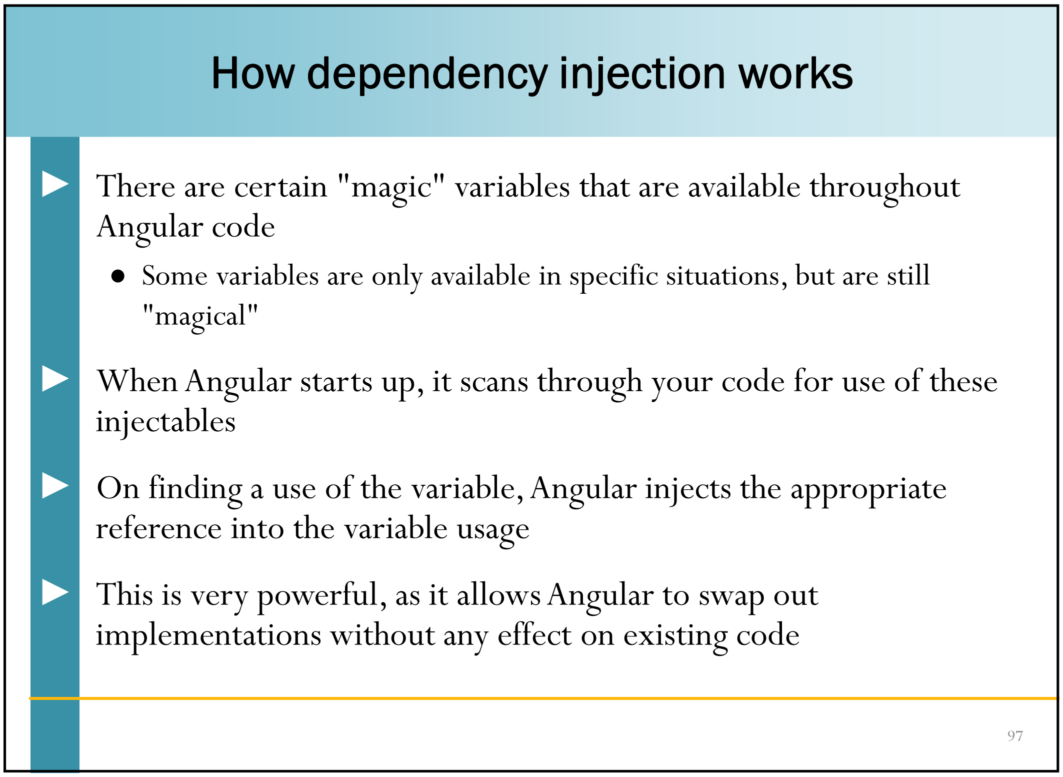
Scope and $scope

- We are used to the general programming idea of scope
- Angular introduces the **`$scope`** object
- Created by the **`$rootScope`** object/service
  - **`var someScope = $rootScope.$new()`** if you need it
- $scope objects are among many capabilities that are made available through Angular's dependency injection system



Dependency injection

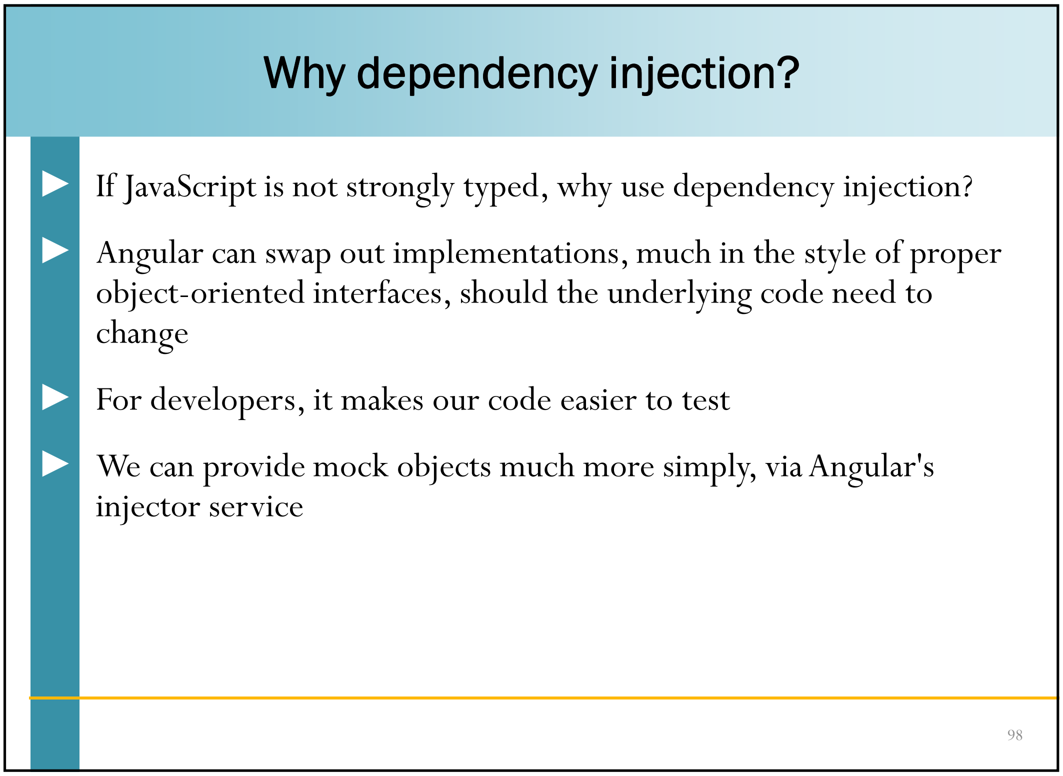
- Angular makes use of a concept called dependency injection (also known as DI)
- Dependency injection is very popular with strongly typed, object-oriented languages
- It's not as popular with JavaScript as it doesn't (initially) seem like a good fit
- But it works well with Angular's approach



How dependency injection works

- There are certain "magic" variables that are available throughout Angular code
  - Some variables are only available in specific situations, but are still "magical"
- When Angular starts up, it scans through your code for use of these injectables
- On finding a use of the variable, Angular injects the appropriate reference into the variable usage
- This is very powerful, as it allows Angular to swap out implementations without any effect on existing code

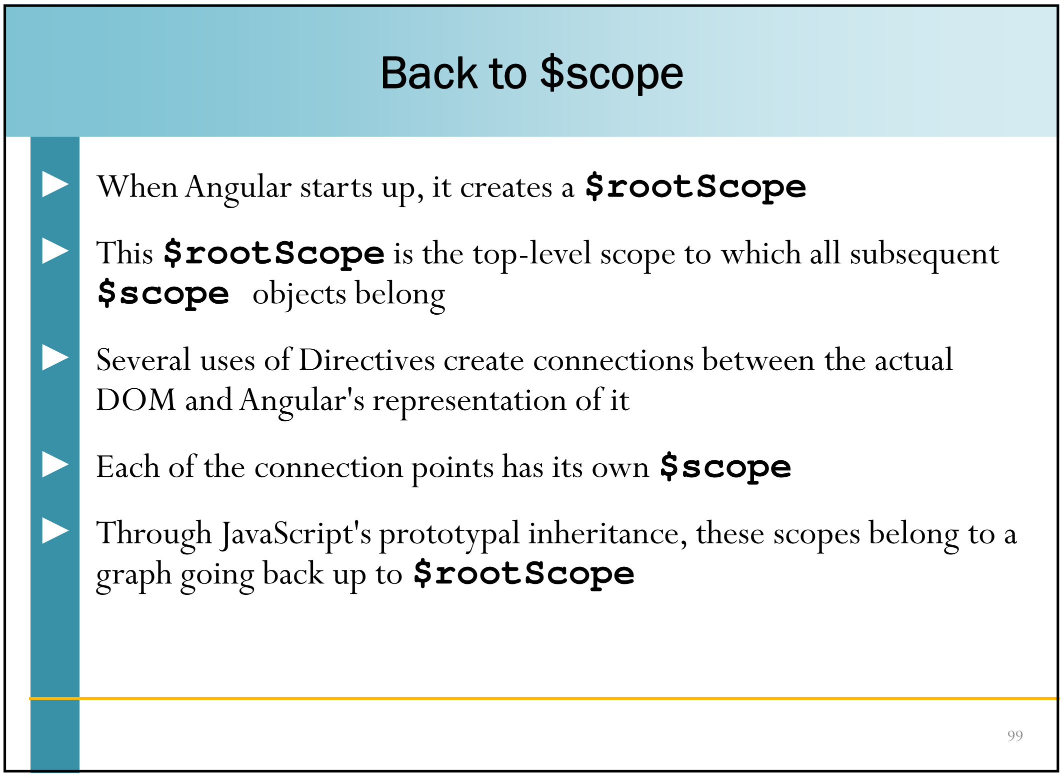
97



Why dependency injection?

- If JavaScript is not strongly typed, why use dependency injection?
- Angular can swap out implementations, much in the style of proper object-oriented interfaces, should the underlying code need to change
- For developers, it makes our code easier to test
- We can provide mock objects much more simply, via Angular's injector service

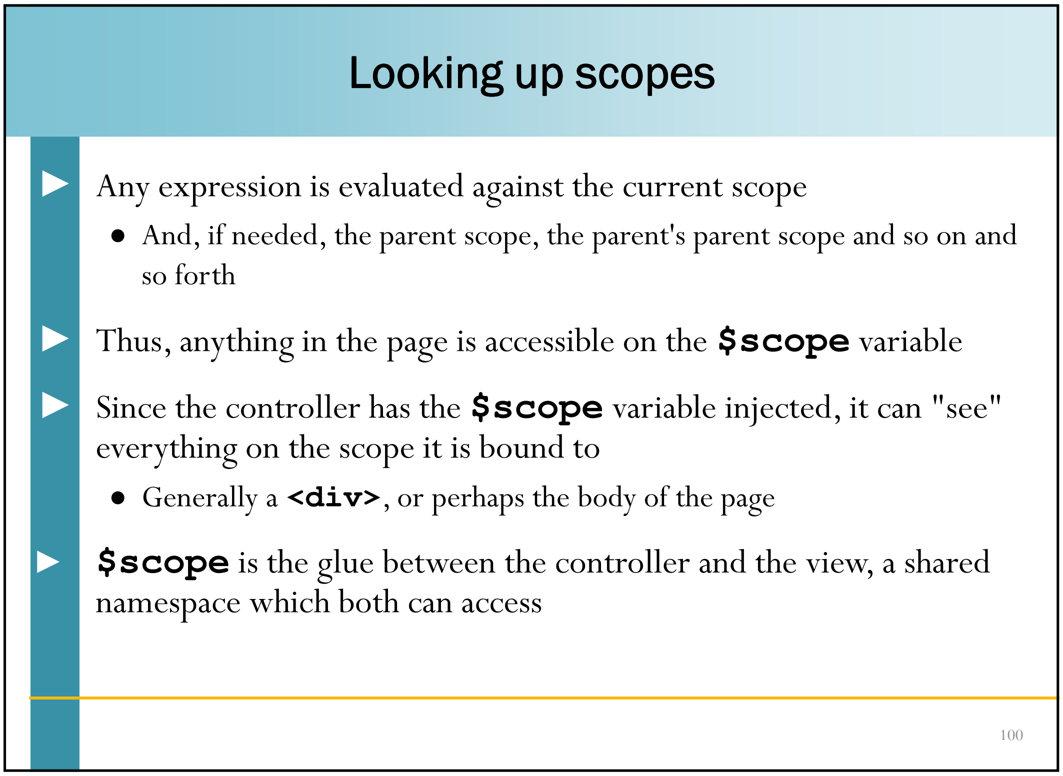
98



Back to $scope

- When Angular starts up, it creates a **`$rootScope`**
- This **`$rootScope`** is the top-level scope to which all subsequent **`$scope`** objects belong
- Several uses of Directives create connections between the actual DOM and Angular's representation of it
- Each of the connection points has its own **`$scope`**
- Through JavaScript's prototypal inheritance, these scopes belong to a graph going back up to **`$rootScope`**

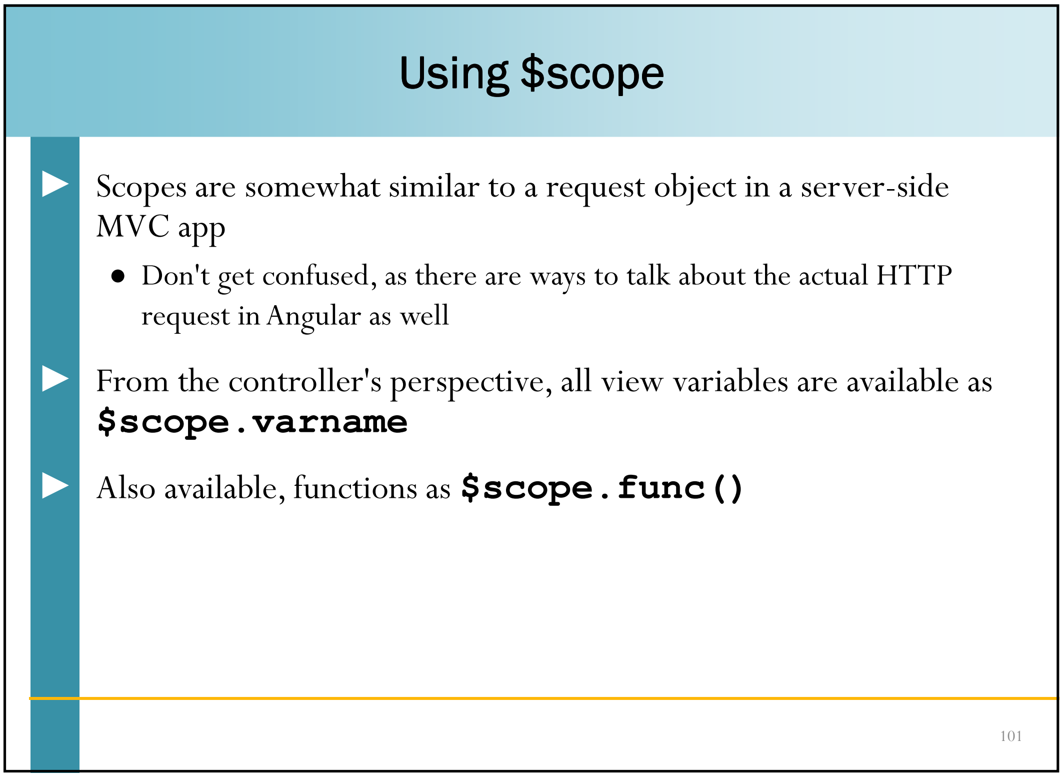
99



Looking up scopes

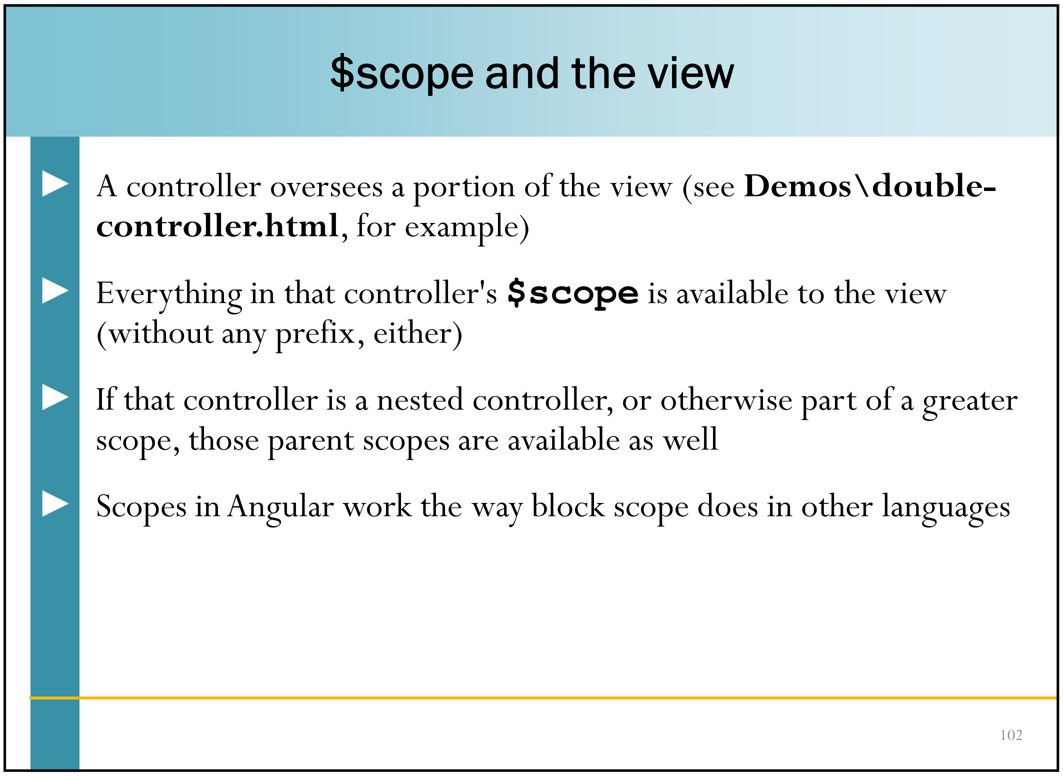
- Any expression is evaluated against the current scope
  - And, if needed, the parent scope, the parent's parent scope and so on and so forth
- Thus, anything in the page is accessible on the **`$scope`** variable
- Since the controller has the **`$scope`** variable injected, it can "see" everything on the scope it is bound to
  - Generally a **`<div>`**, or perhaps the body of the page
- **`$scope`** is the glue between the controller and the view, a shared namespace which both can access

100



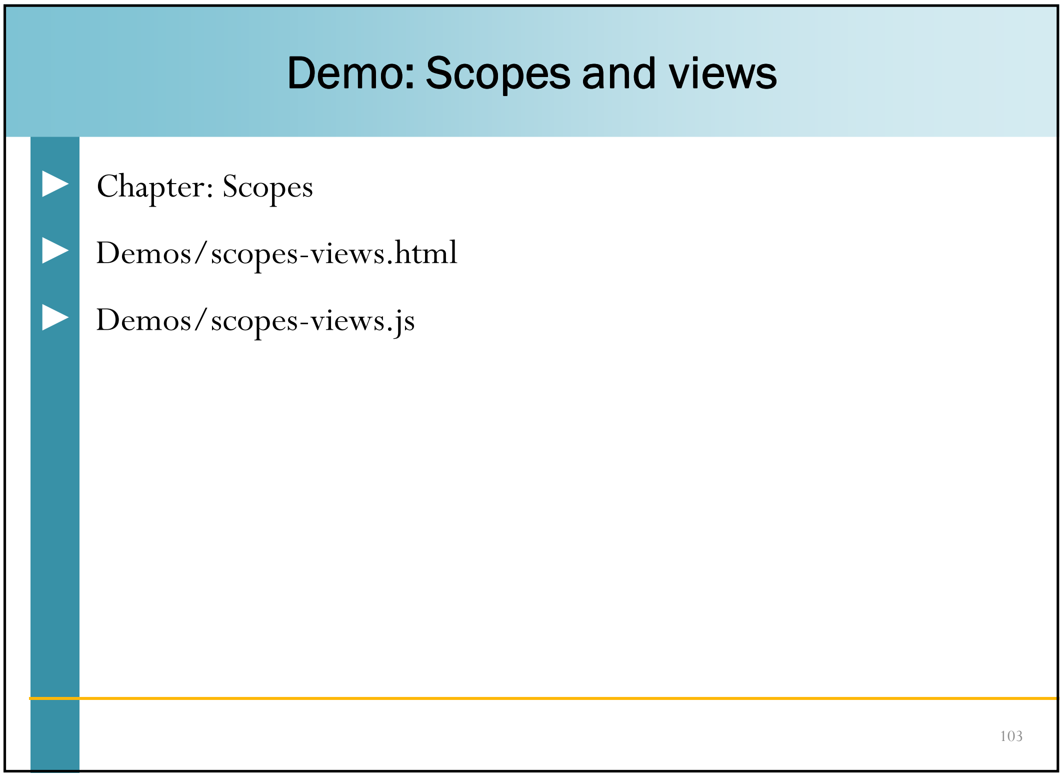
Using $scope

- Scopes are somewhat similar to a request object in a server-side MVC app
  - Don't get confused, as there are ways to talk about the actual HTTP request in Angular as well
- From the controller's perspective, all view variables are available as **`$scope.varname`**
- Also available, functions as **`$scope.func()`**



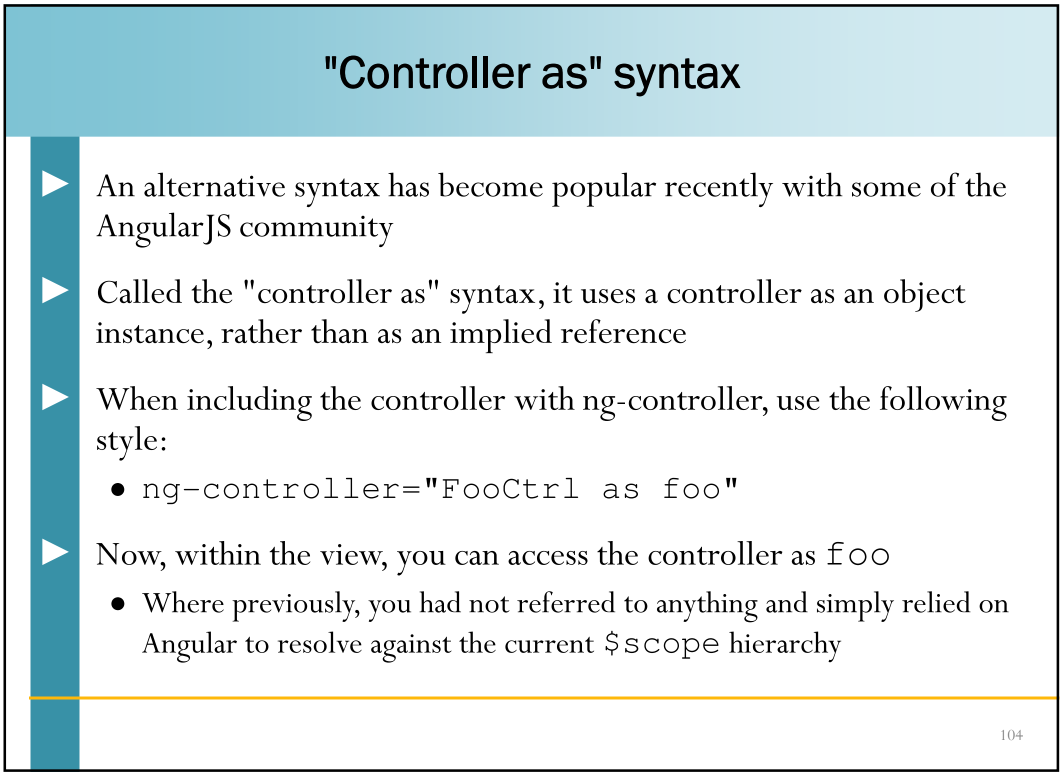
$scope and the view

- A controller oversees a portion of the view (see **Demos\double-controller.html**, for example)
- Everything in that controller's **`$scope`** is available to the view (without any prefix, either)
- If that controller is a nested controller, or otherwise part of a greater scope, those parent scopes are available as well
- Scopes in Angular work the way block scope does in other languages



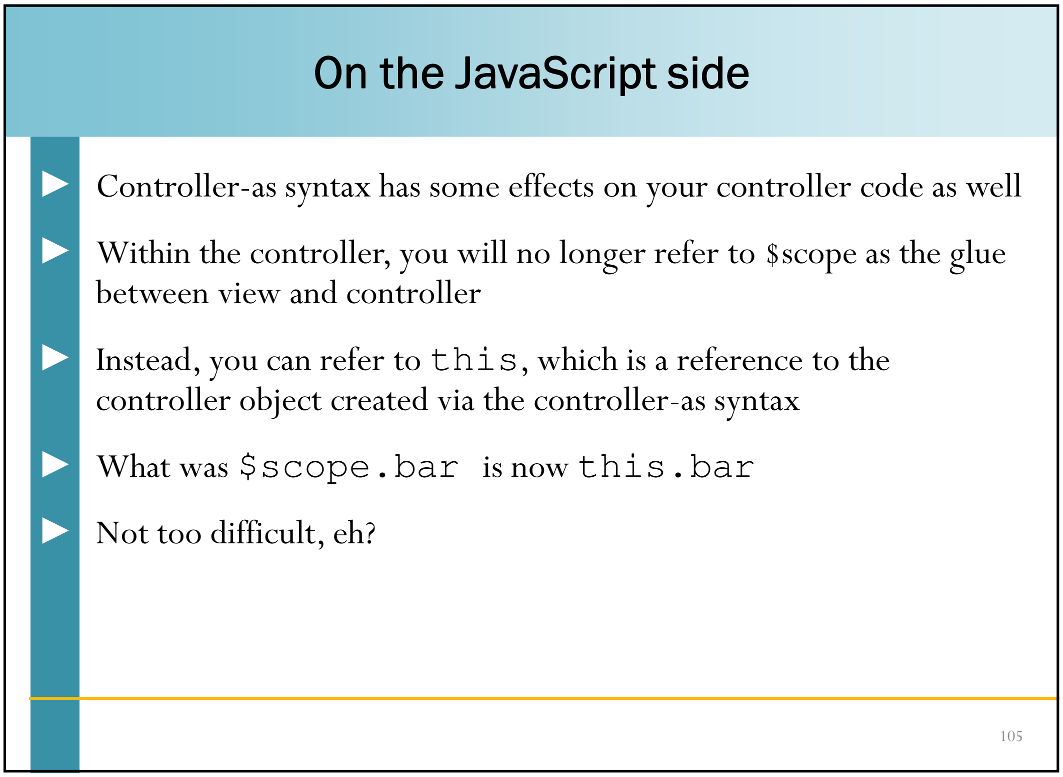
Demo: Scopes and views

- Chapter: Scopes
- Demos/scopes-views.html
- Demos/scopes-views.js



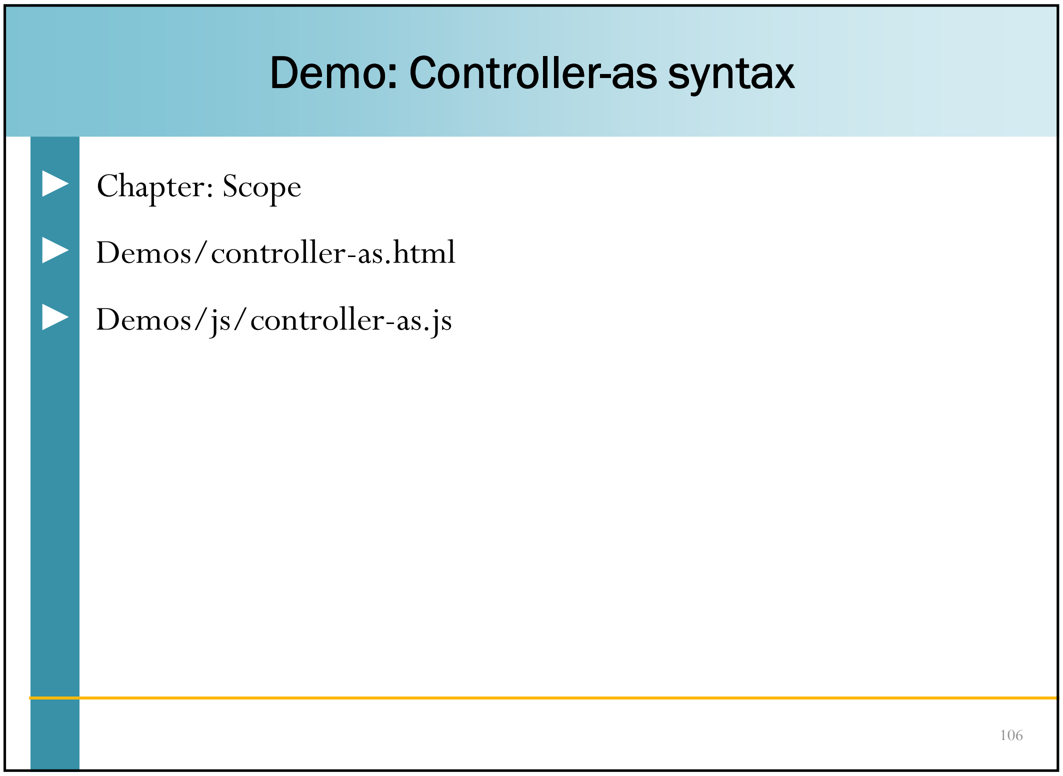
"Controller as" syntax

- An alternative syntax has become popular recently with some of the AngularJS community
- Called the "controller as" syntax, it uses a controller as an object instance, rather than as an implied reference
- When including the controller with ng-controller, use the following style:
  - `ng-controller="FooCtrl as foo"`
- Now, within the view, you can access the controller as `foo`
  - Where previously, you had not referred to anything and simply relied on Angular to resolve against the current `$scope` hierarchy



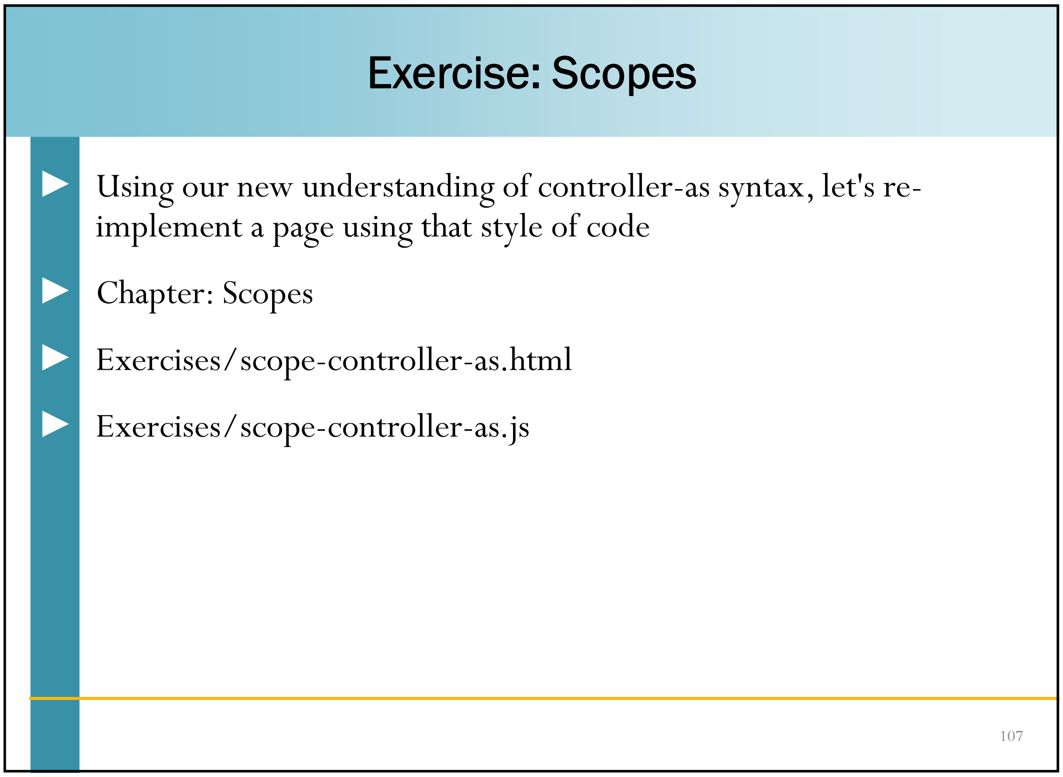
On the JavaScript side

- Controller-as syntax has some effects on your controller code as well
- Within the controller, you will no longer refer to $scope as the glue between view and controller
- Instead, you can refer to `this`, which is a reference to the controller object created via the controller-as syntax
- What was `$scope.bar` is now `this.bar`
- Not too difficult, eh?



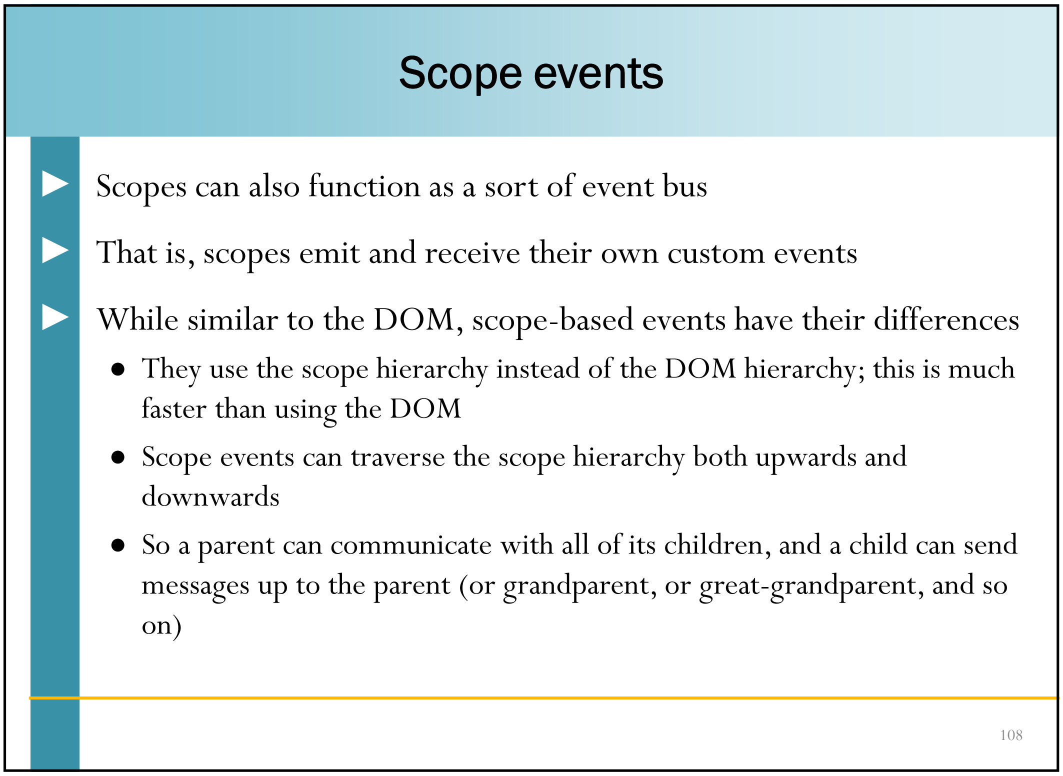
Demo: Controller-as syntax

- Chapter: Scope
- Demos/controller-as.html
- Demos/js/controller-as.js



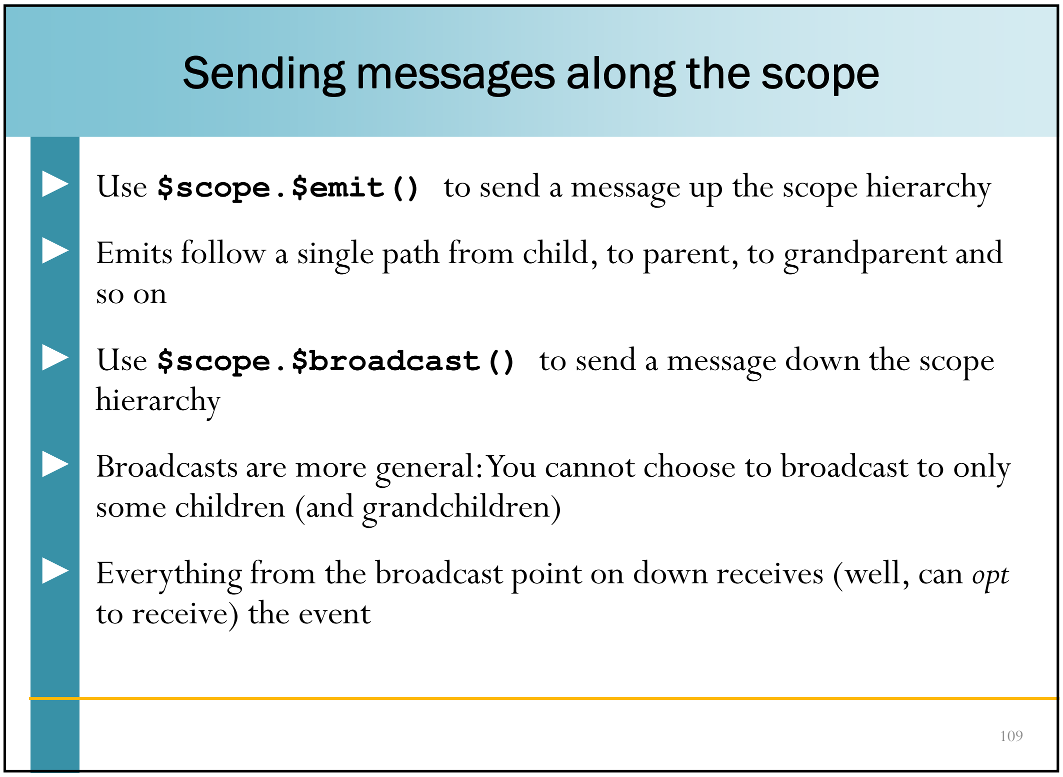
Exercise: Scopes

- Using our new understanding of controller-as syntax, let's re-implement a page using that style of code
- Chapter: Scopes
- Exercises/scope-controller-as.html
- Exercises/scope-controller-as.js



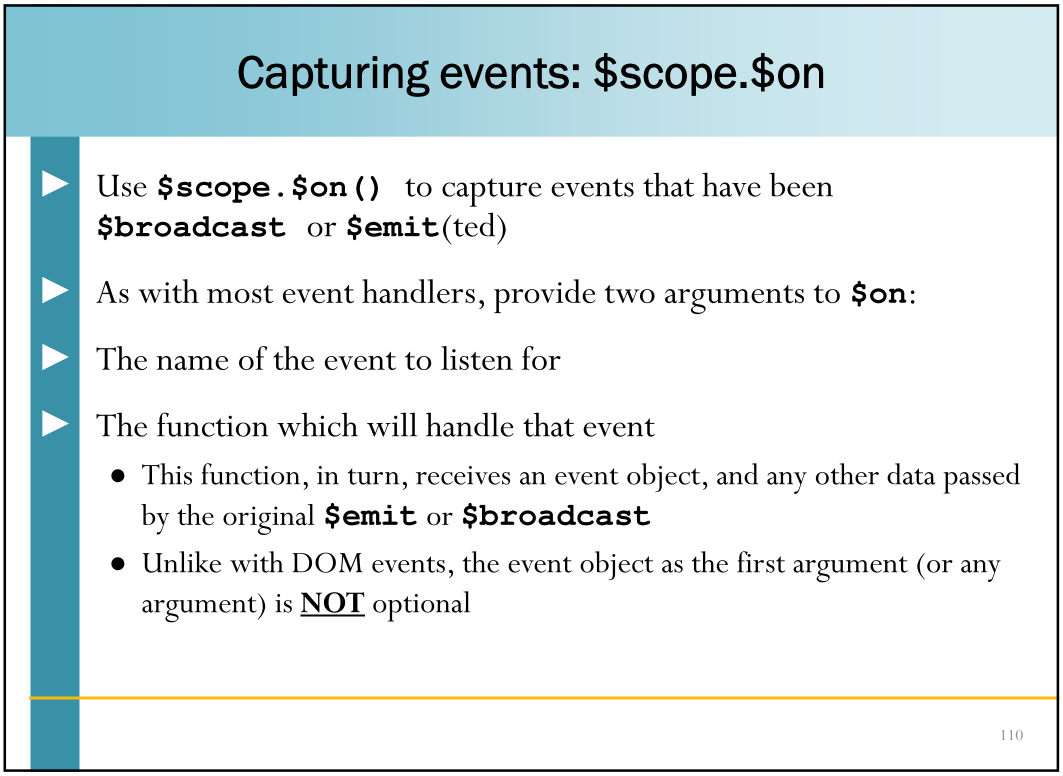
Scope events

- Scopes can also function as a sort of event bus
- That is, scopes emit and receive their own custom events
- While similar to the DOM, scope-based events have their differences
  - They use the scope hierarchy instead of the DOM hierarchy; this is much faster than using the DOM
  - Scope events can traverse the scope hierarchy both upwards and downwards
  - So a parent can communicate with all of its children, and a child can send messages up to the parent (or grandparent, or great-grandparent, and so on)



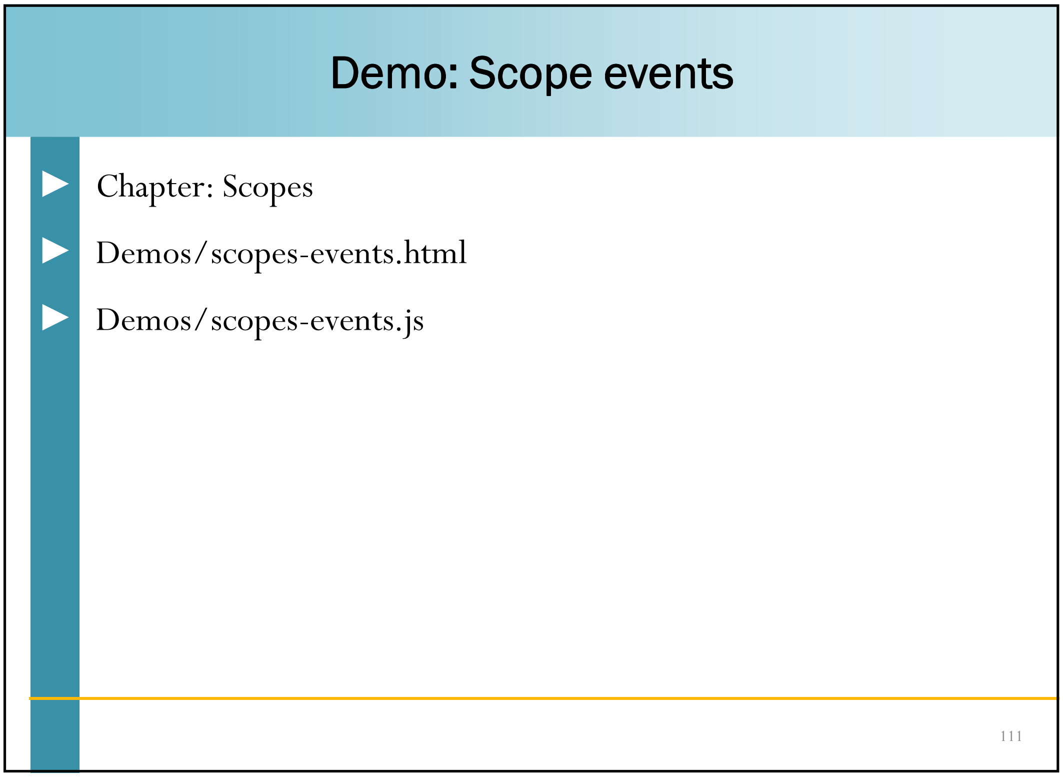
Sending messages along the scope

- Use **`$scope.$emit()`** to send a message up the scope hierarchy
- Emits follow a single path from child, to parent, to grandparent and so on
- Use **`$scope.$broadcast()`** to send a message down the scope hierarchy
- Broadcasts are more general: You cannot choose to broadcast to only some children (and grandchildren)
- Everything from the broadcast point on down receives (well, can *opt* to receive) the event



Capturing events: $scope.$on

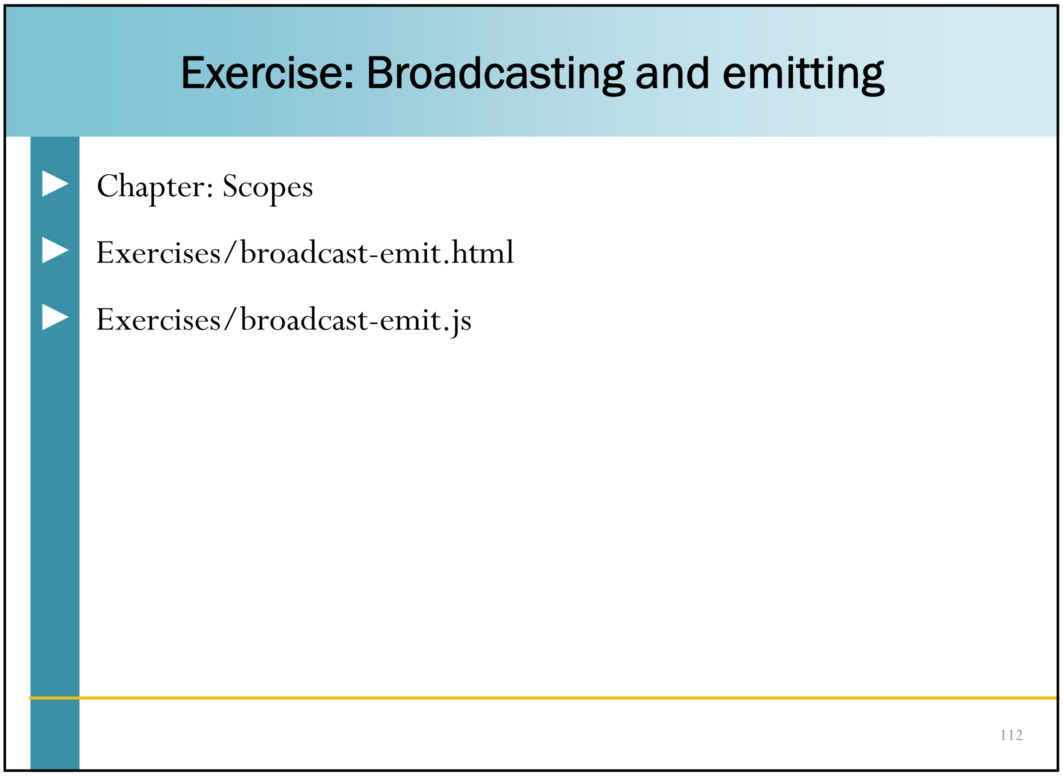
- Use **`$scope.$on()`** to capture events that have been **`$broadcast`** or **`$emit`**(ted)
- As with most event handlers, provide two arguments to **`$on`**:
- The name of the event to listen for
- The function which will handle that event
  - This function, in turn, receives an event object, and any other data passed by the original **`$emit`** or **`$broadcast`**
  - Unlike with DOM events, the event object as the first argument (or any argument) is **NOT** optional



Demo: Scope events

- Chapter: Scopes
- Demos/scopes-events.html
- Demos/scopes-events.js

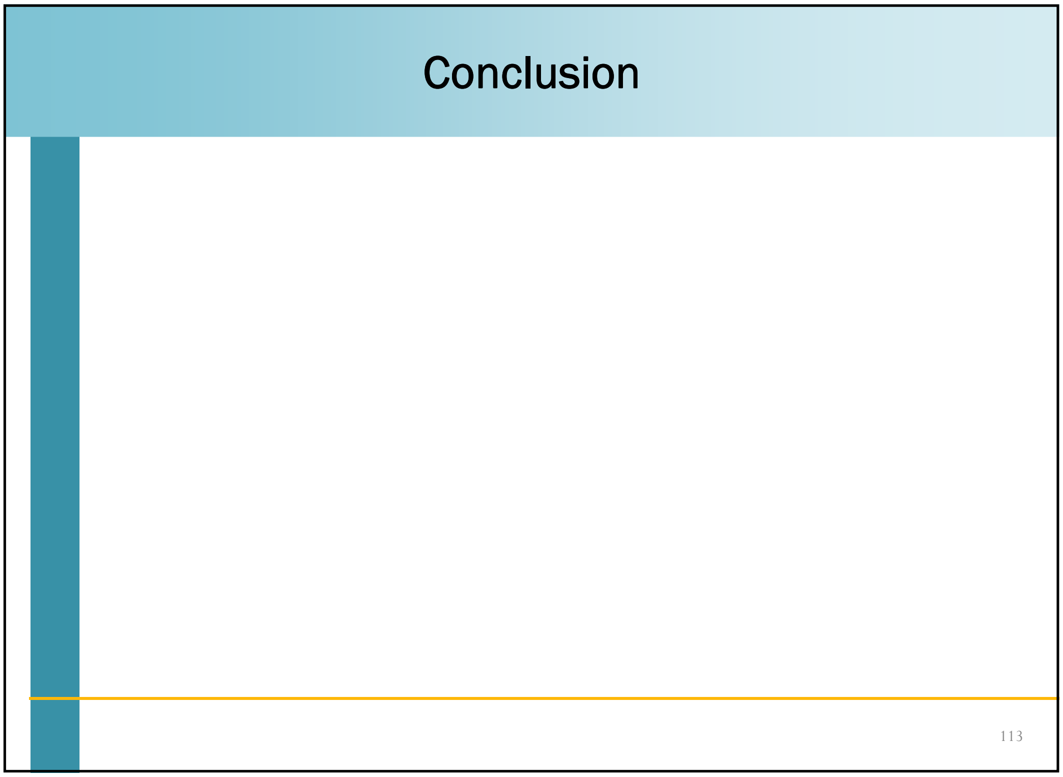
111



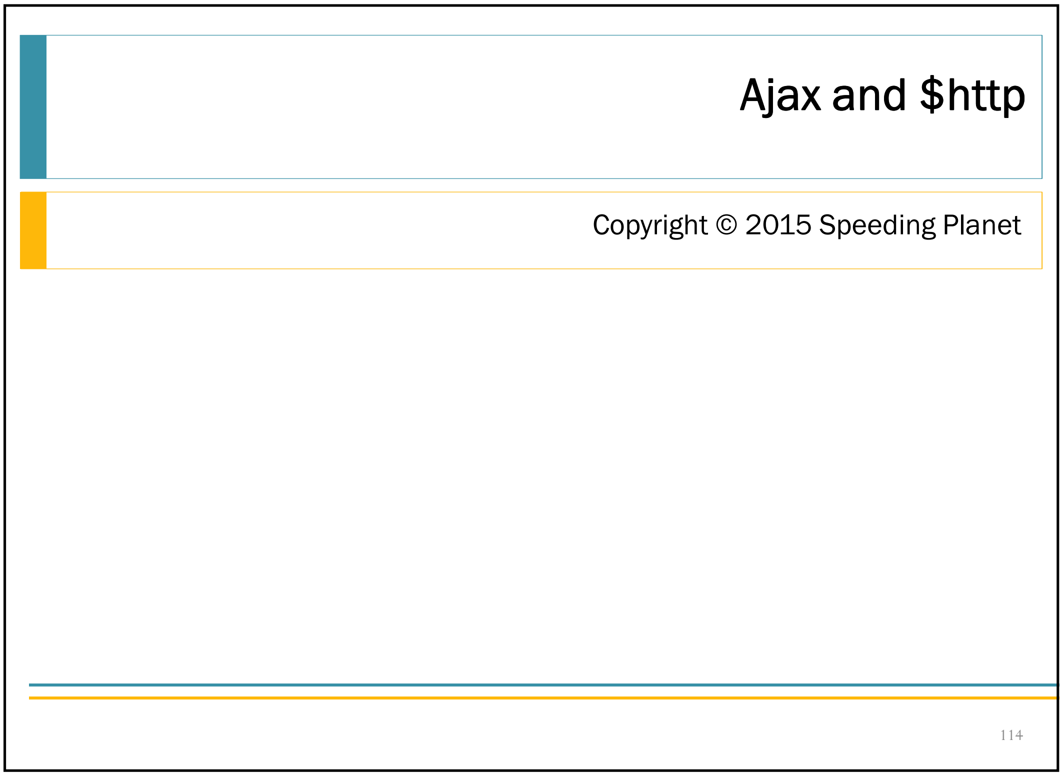
Exercise: Broadcasting and emitting

- Chapter: Scopes
- Exercises/broadcast-emit.html
- Exercises/broadcast-emit.js

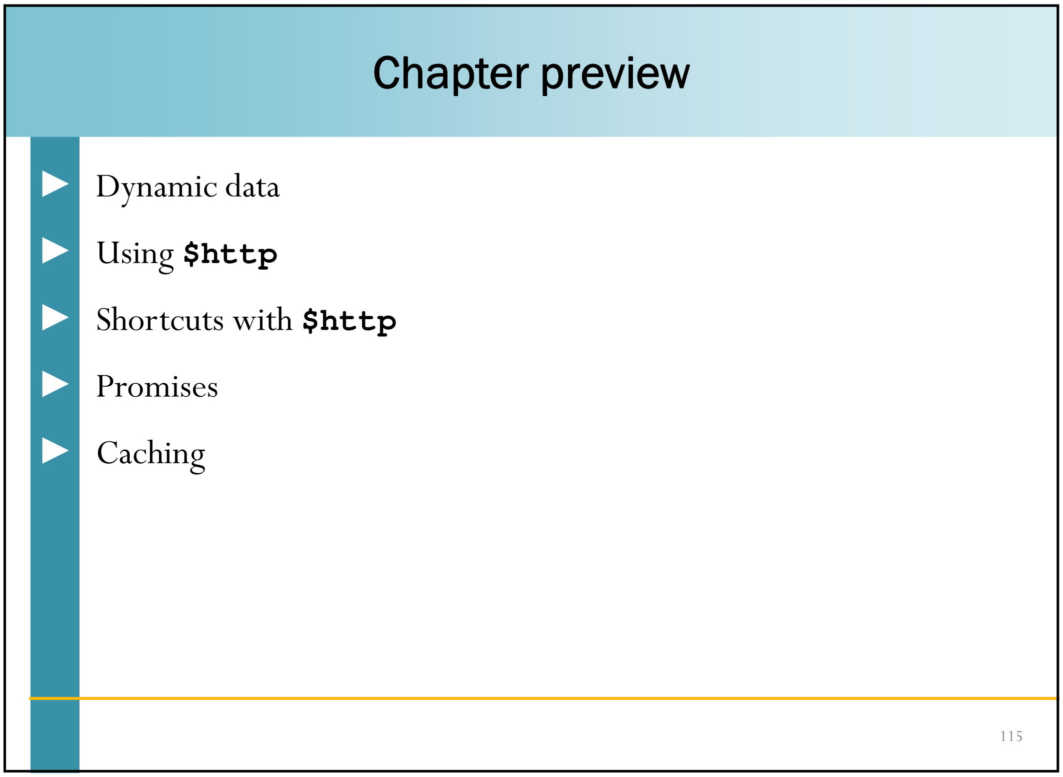
112



Conclusion

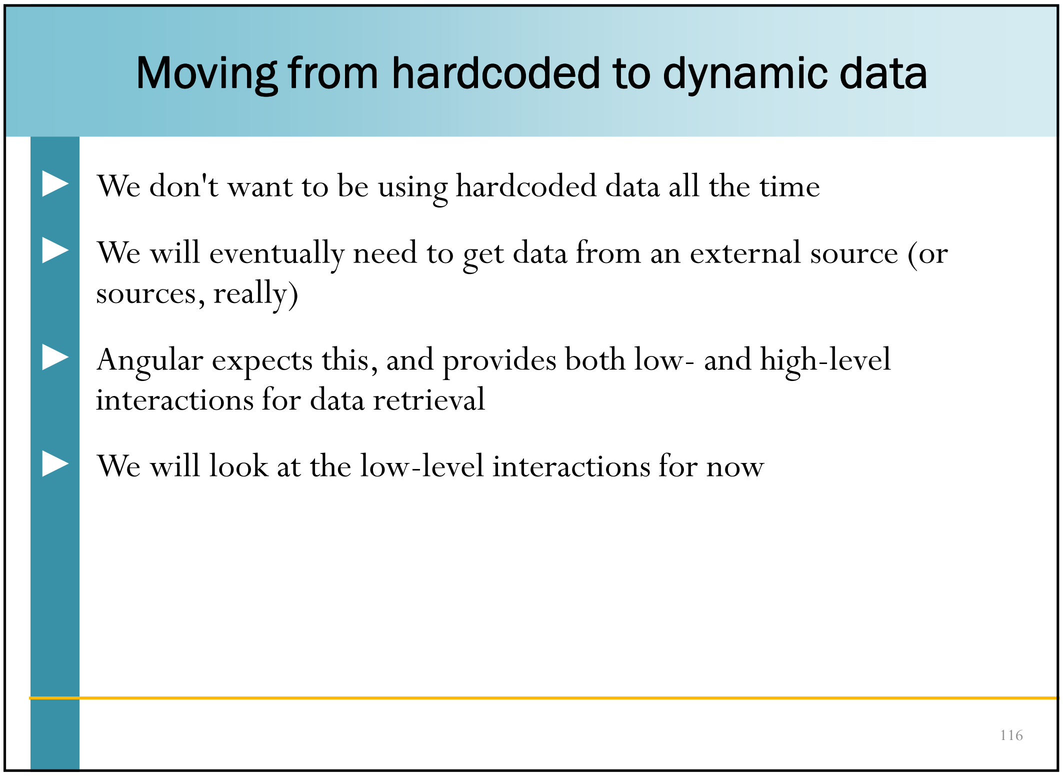


Ajax and $http



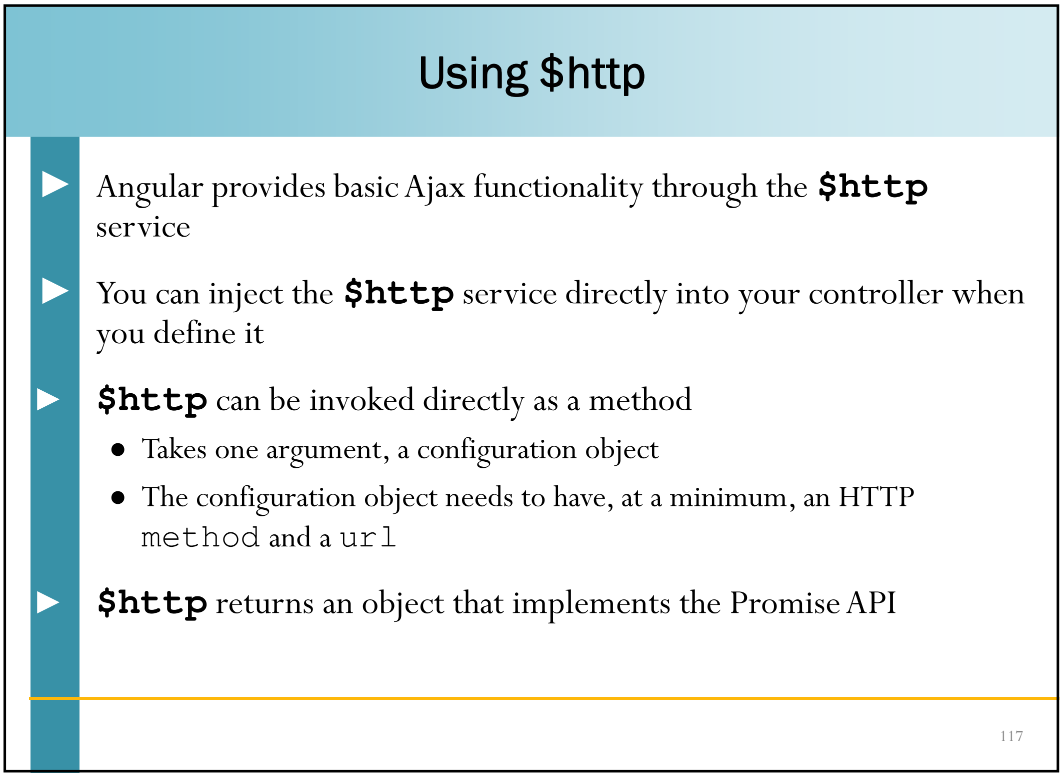
Chapter preview

- Dynamic data
- Using **`$http`**
- Shortcuts with **`$http`**
- Promises
- Caching



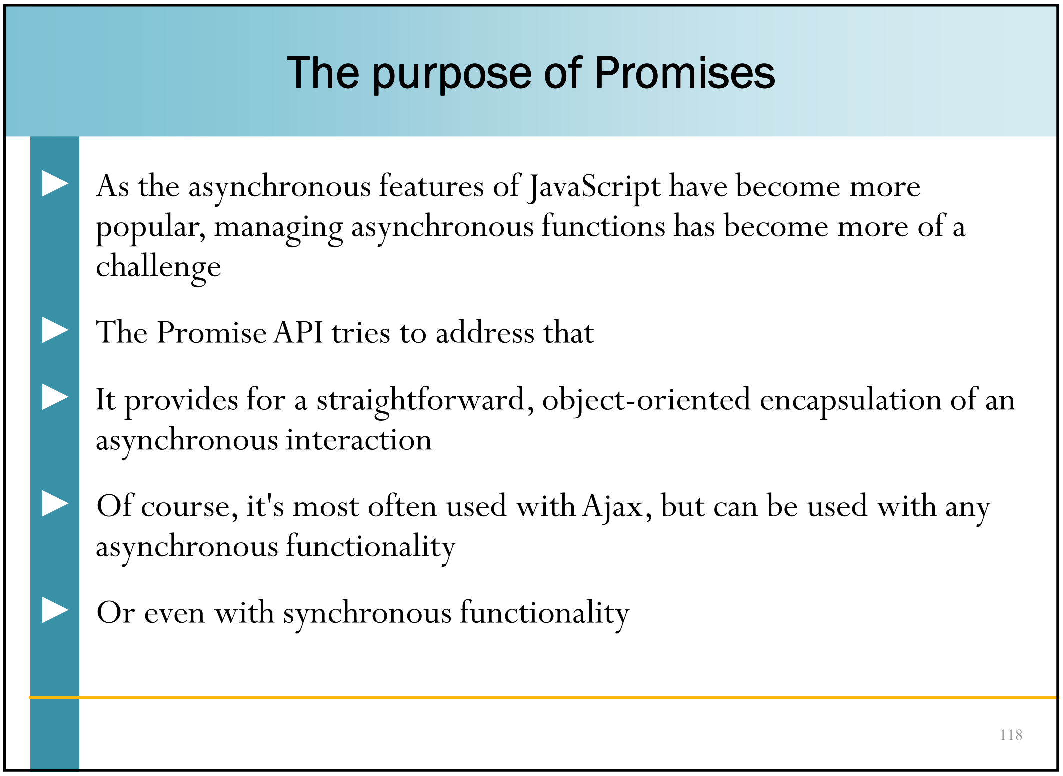
Moving from hardcoded to dynamic data

- We don't want to be using hardcoded data all the time
- We will eventually need to get data from an external source (or sources, really)
- Angular expects this, and provides both low- and high-level interactions for data retrieval
- We will look at the low-level interactions for now



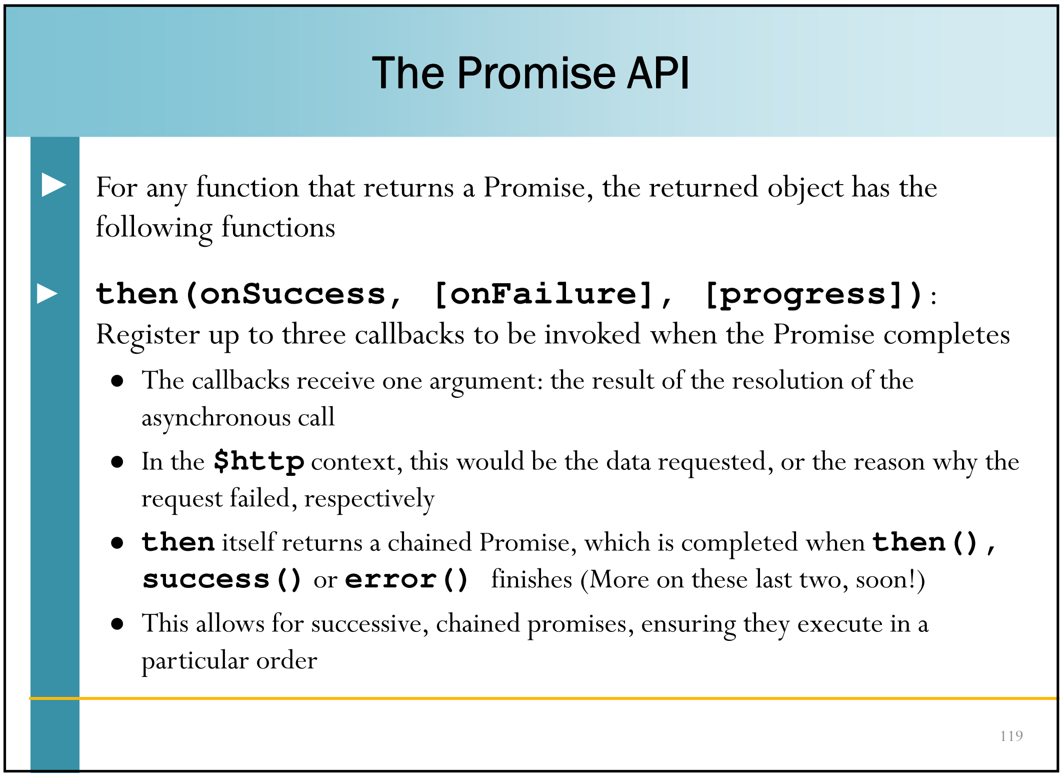
Using $http

- Angular provides basic Ajax functionality through the **`$http`** service
- You can inject the **`$http`** service directly into your controller when you define it
- **`$http`** can be invoked directly as a method
  - Takes one argument, a configuration object
  - The configuration object needs to have, at a minimum, an HTTP `method` and a `url`
- **`$http`** returns an object that implements the Promise API



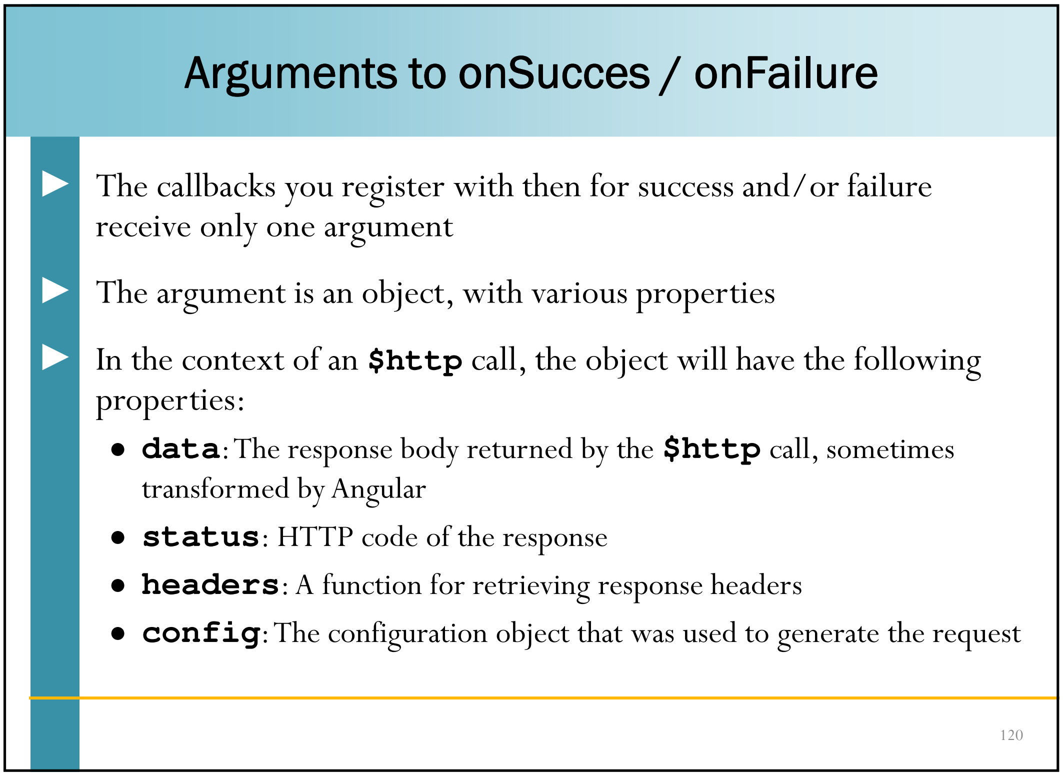
The purpose of Promises

- As the asynchronous features of JavaScript have become more popular, managing asynchronous functions has become more of a challenge
- The Promise API tries to address that
- It provides for a straightforward, object-oriented encapsulation of an asynchronous interaction
- Of course, it's most often used with Ajax, but can be used with any asynchronous functionality
- Or even with synchronous functionality



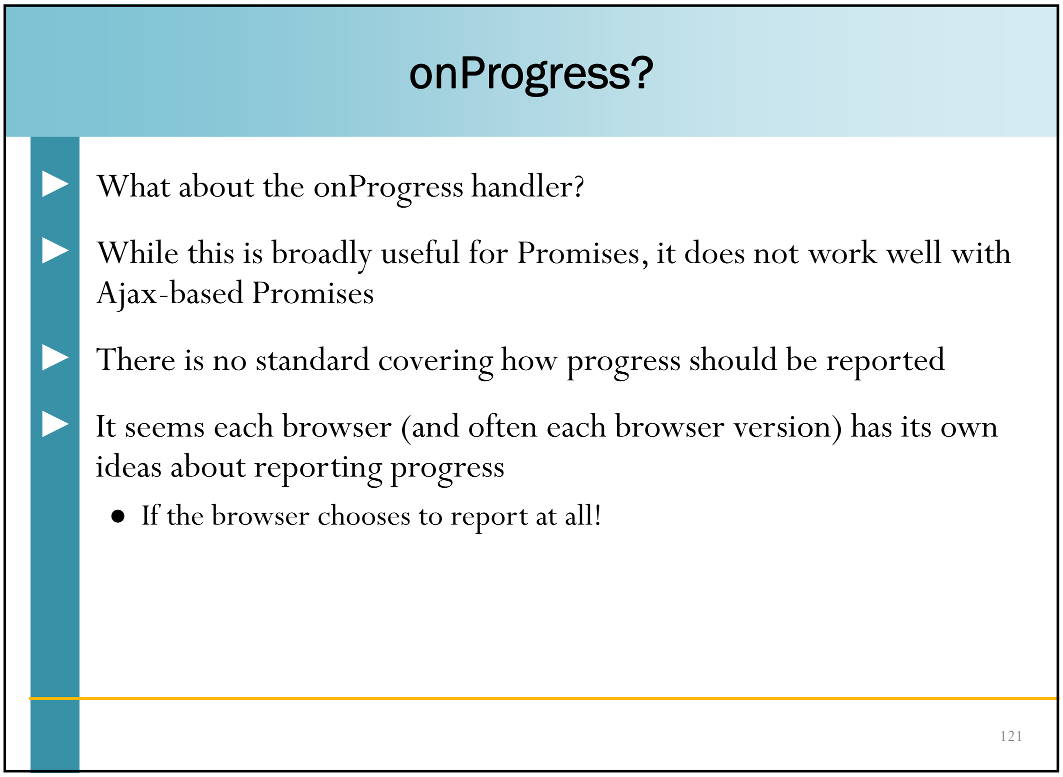
The Promise API

- For any function that returns a Promise, the returned object has the following functions
- **`then(onSuccess, [onFailure], [progress])`**: Register up to three callbacks to be invoked when the Promise completes
  - The callbacks receive one argument: the result of the resolution of the asynchronous call
  - In the **`$http`** context, this would be the data requested, or the reason why the request failed, respectively
  - **`then`** itself returns a chained Promise, which is completed when **`then()`**, **`success()`** or **`error()`** finishes (More on these last two, soon!)
  - This allows for successive, chained promises, ensuring they execute in a particular order



Arguments to onSucces / onFailure

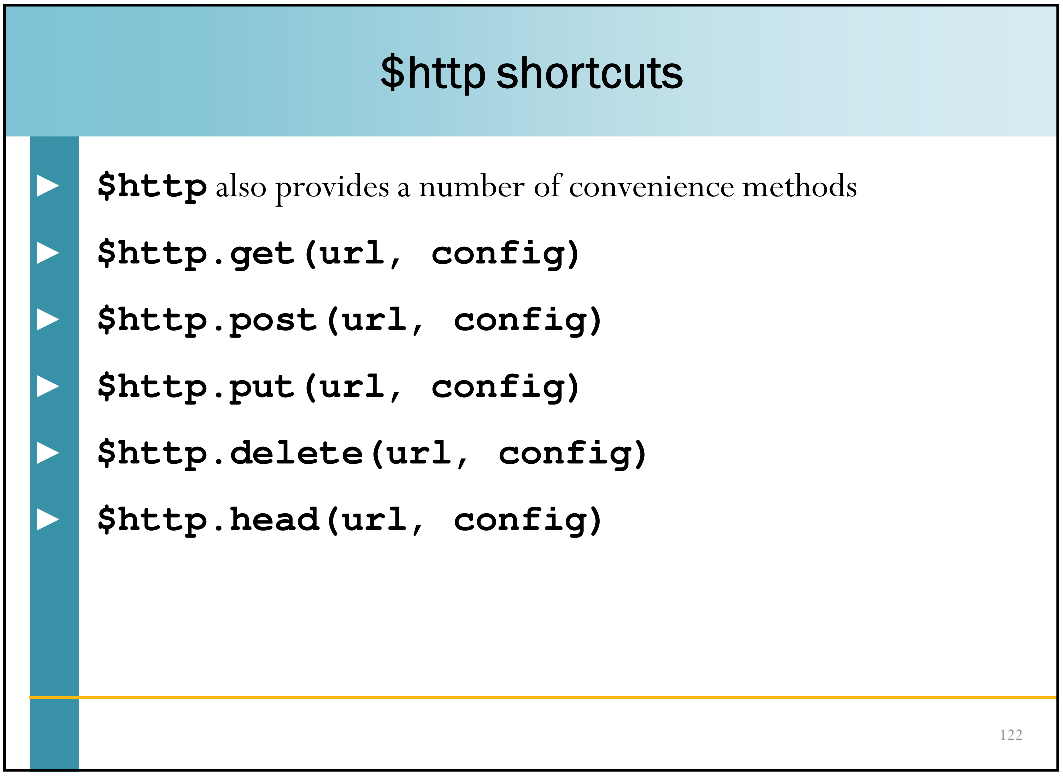
- The callbacks you register with then for success and/or failure receive only one argument
- The argument is an object, with various properties
- In the context of an **`$http`** call, the object will have the following properties:
  - **`data`**: The response body returned by the **`$http`** call, sometimes transformed by Angular
  - **`status`**: HTTP code of the response
  - **`headers`**: A function for retrieving response headers
  - **`config`**: The configuration object that was used to generate the request



onProgress?

- What about the onProgress handler?
- While this is broadly useful for Promises, it does not work well with Ajax-based Promises
- There is no standard covering how progress should be reported
- It seems each browser (and often each browser version) has its own ideas about reporting progress
  - If the browser chooses to report at all!

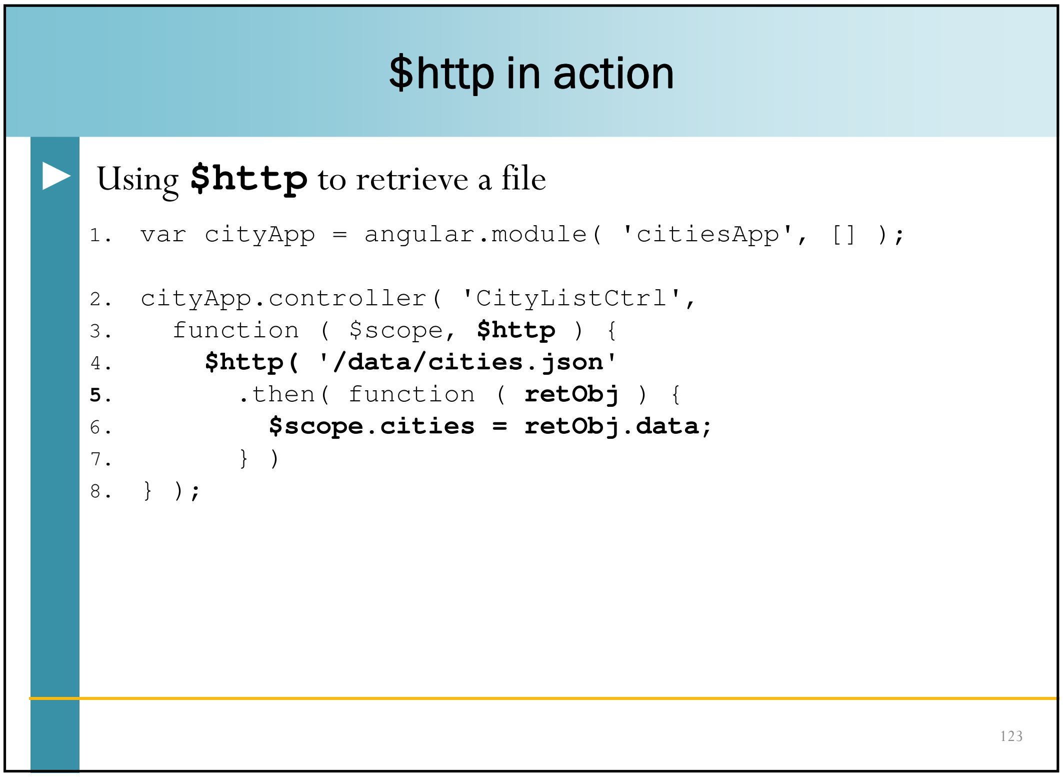
121



$http shortcuts

- **`$http`** also provides a number of convenience methods
- **`$http.get(url, config)`**
- **`$http.post(url, config)`**
- **`$http.put(url, config)`**
- **`$http.delete(url, config)`**
- **`$http.head(url, config)`**

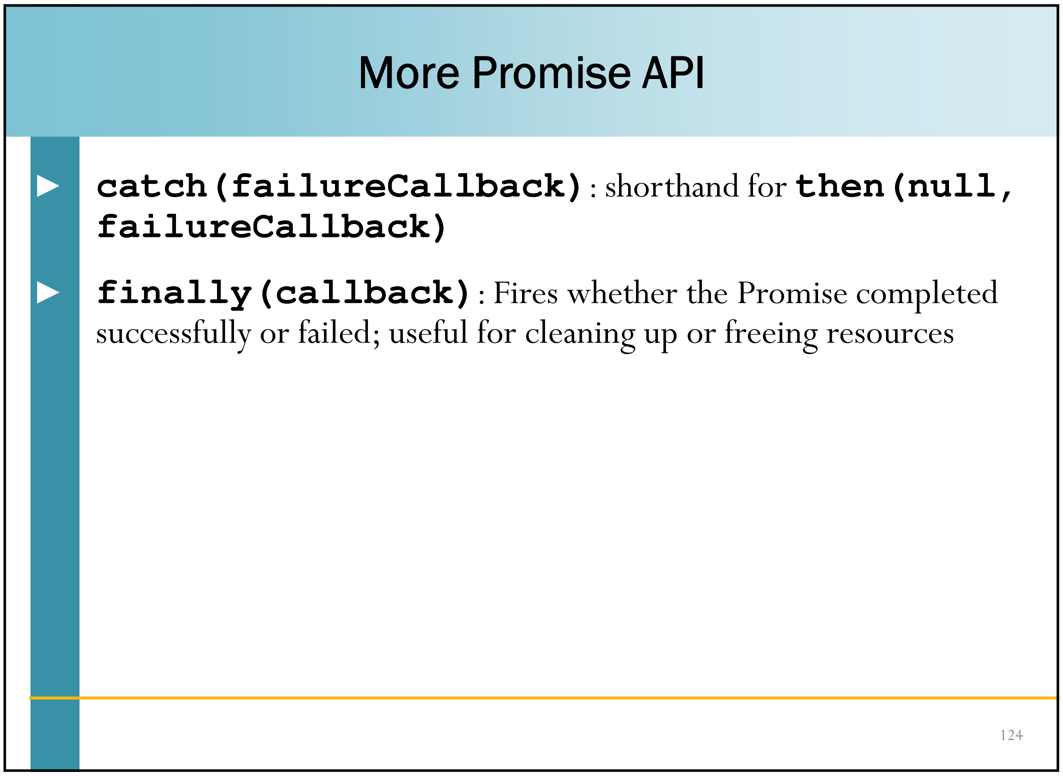
122



$http in action

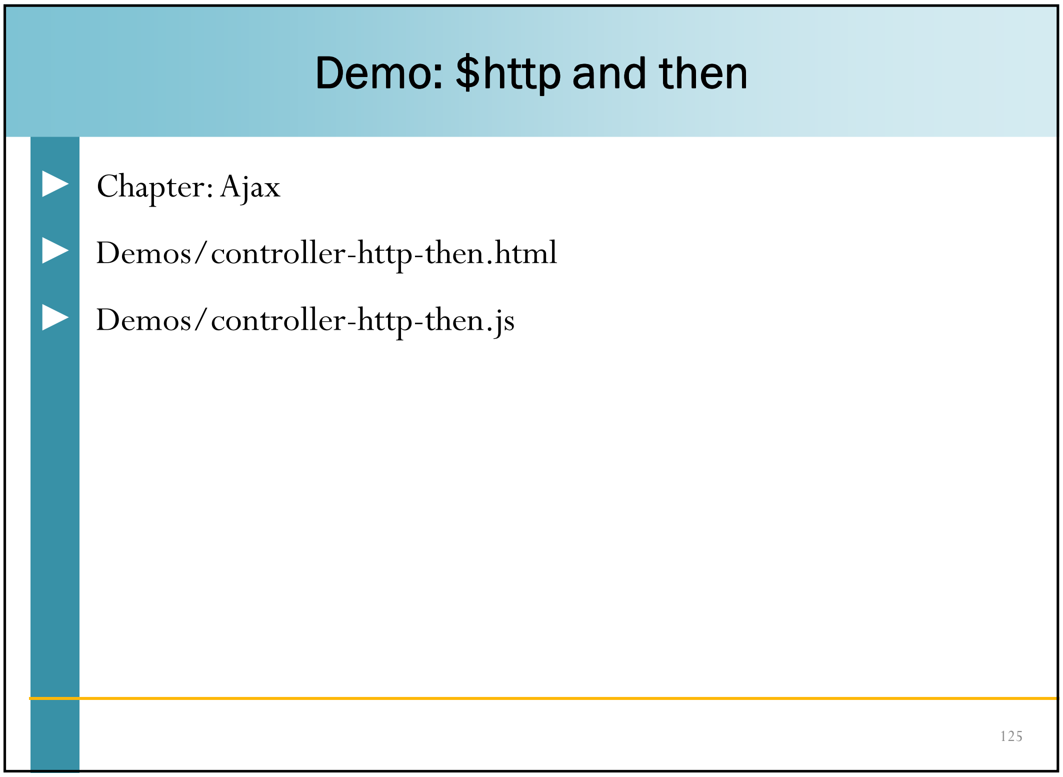
- Using **$http** to retrieve a file

```
var cityApp = angular.module( 'citiesApp', [] );

cityApp.controller( 'CityListCtrl',
  function ( $scope, $http ) {
    $http( '/data/cities.json'
      .then( function ( retObj ) {
        $scope.cities = retObj.data;
      } )
} );
```



More Promise API

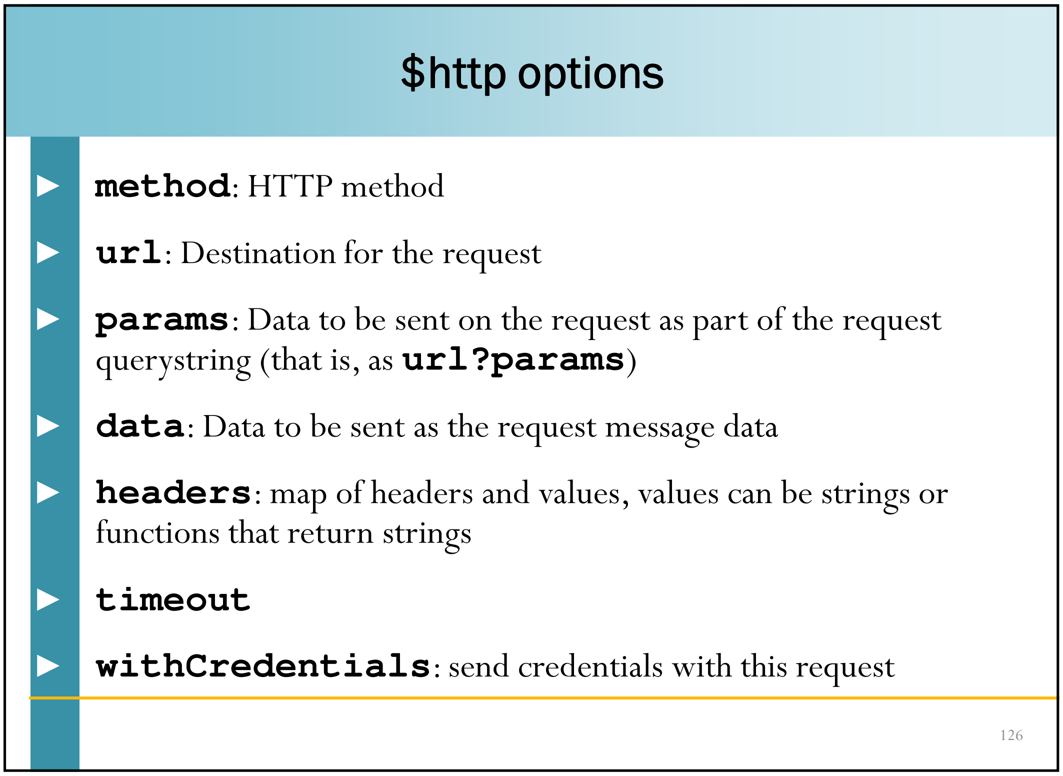
- **catch(failureCallback)** : shorthand for **then(null, failureCallback)**
- **finally(callback)** : Fires whether the Promise completed successfully or failed; useful for cleaning up or freeing resources



Demo: $http and then

- Chapter: Ajax
- Demos/controller-http-then.html
- Demos/controller-http-then.js

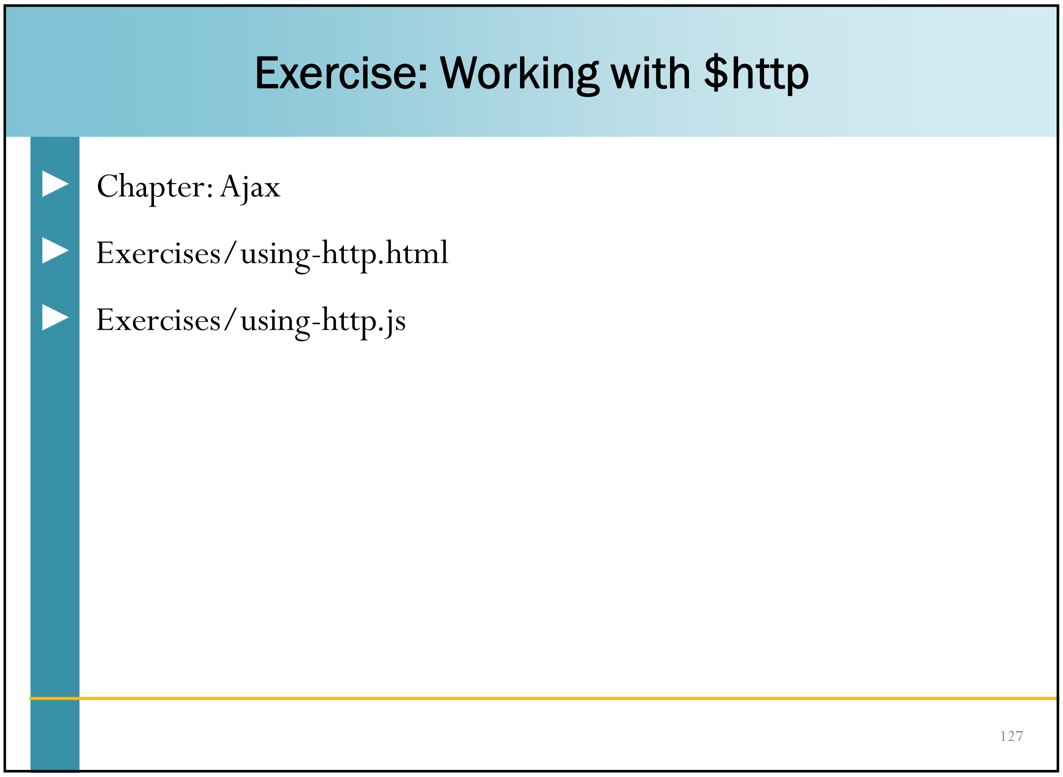
125



$http options

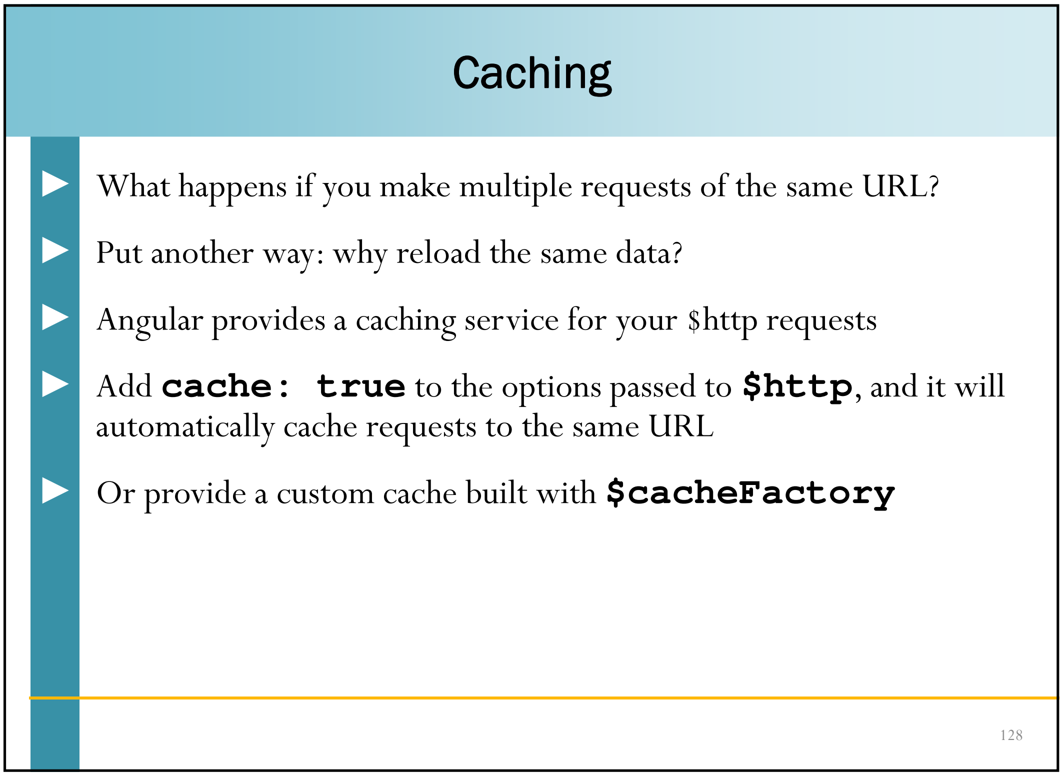
- **`method`**: HTTP method
- **`url`**: Destination for the request
- **`params`**: Data to be sent on the request as part of the request querystring (that is, as **`url?params`**)
- **`data`**: Data to be sent as the request message data
- **`headers`**: map of headers and values, values can be strings or functions that return strings
- **`timeout`**
- **`withCredentials`**: send credentials with this request

126



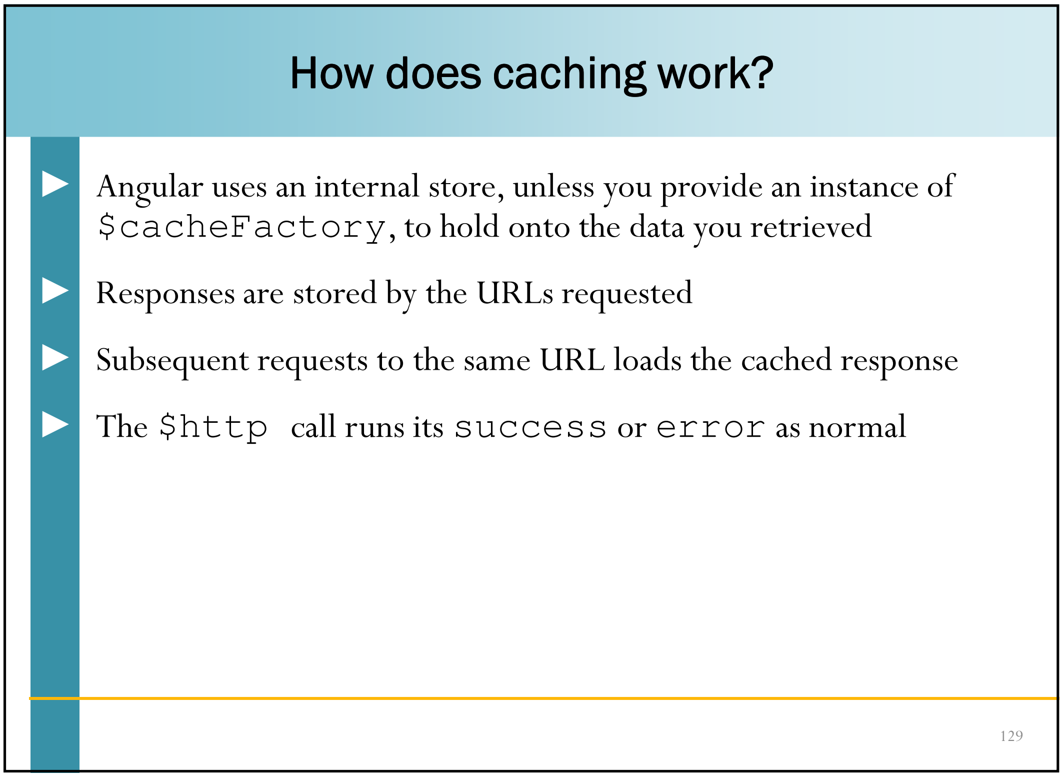
Exercise: Working with $http

- Chapter: Ajax
- Exercises/using-http.html
- Exercises/using-http.js



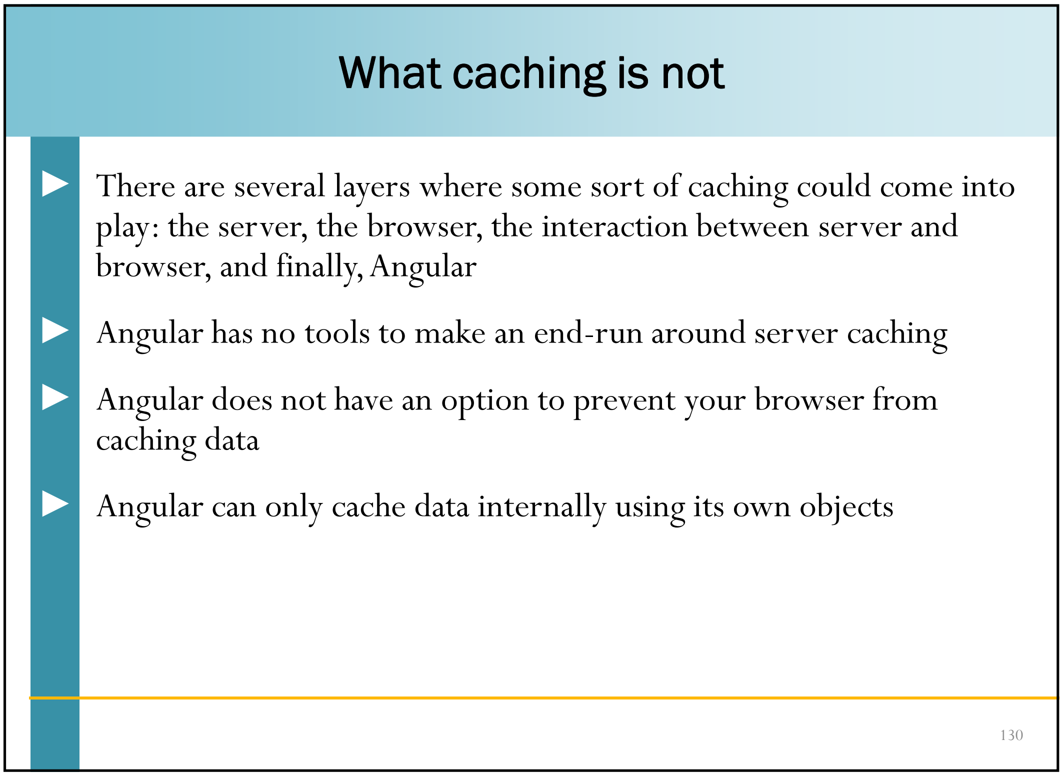
Caching

- What happens if you make multiple requests of the same URL?
- Put another way: why reload the same data?
- Angular provides a caching service for your $http requests
- Add **`cache: true`** to the options passed to **`$http`**, and it will automatically cache requests to the same URL
- Or provide a custom cache built with **`$cacheFactory`**



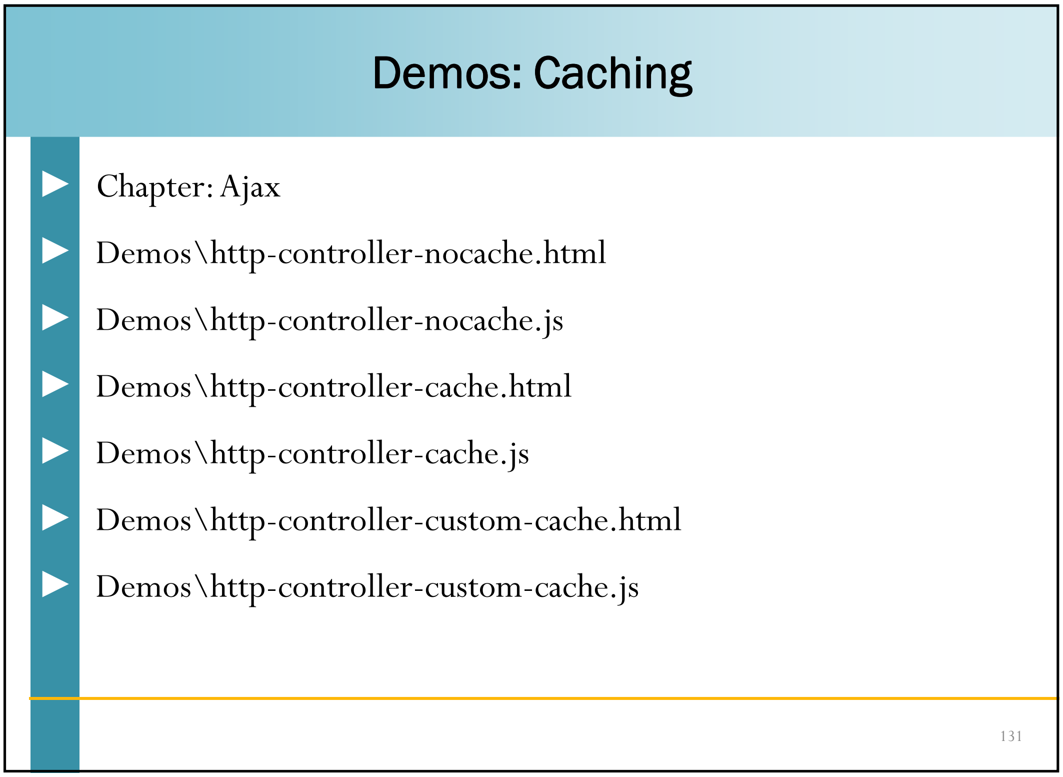
How does caching work?

- Angular uses an internal store, unless you provide an instance of `$cacheFactory`, to hold onto the data you retrieved
- Responses are stored by the URLs requested
- Subsequent requests to the same URL loads the cached response
- The `$http` call runs its `success` or `error` as normal



What caching is not

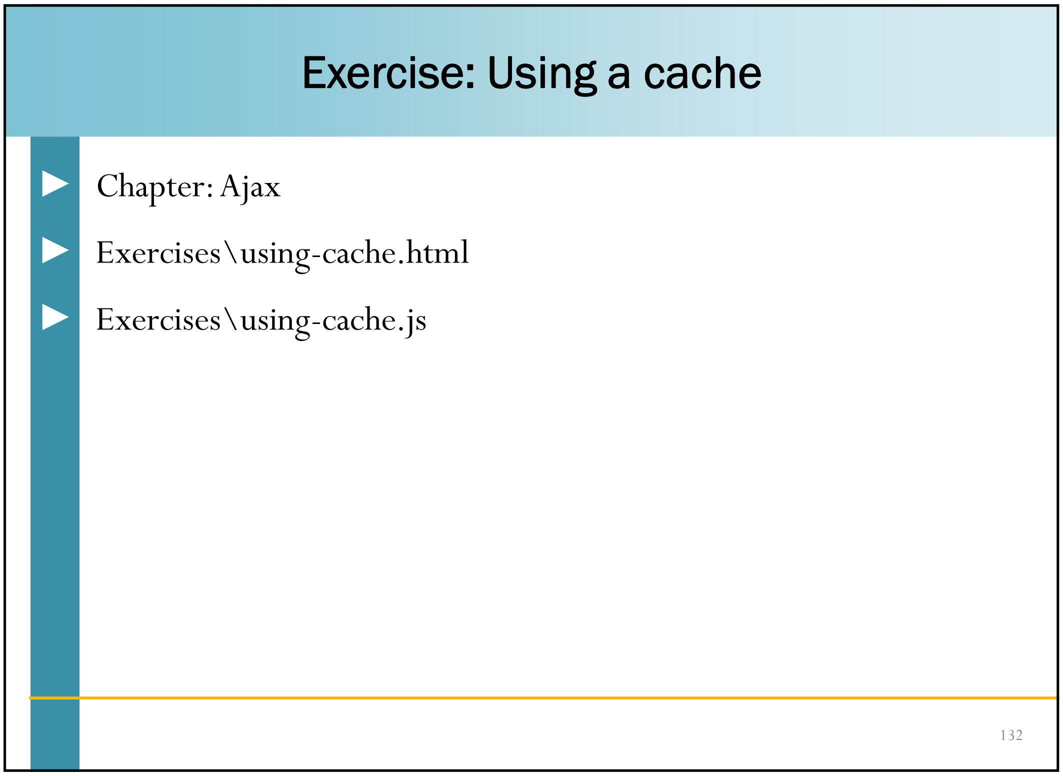
- There are several layers where some sort of caching could come into play: the server, the browser, the interaction between server and browser, and finally, Angular
- Angular has no tools to make an end-run around server caching
- Angular does not have an option to prevent your browser from caching data
- Angular can only cache data internally using its own objects



Demos: Caching

- Chapter: Ajax
- Demos\http-controller-nocache.html
- Demos\http-controller-nocache.js
- Demos\http-controller-cache.html
- Demos\http-controller-cache.js
- Demos\http-controller-custom-cache.html
- Demos\http-controller-custom-cache.js

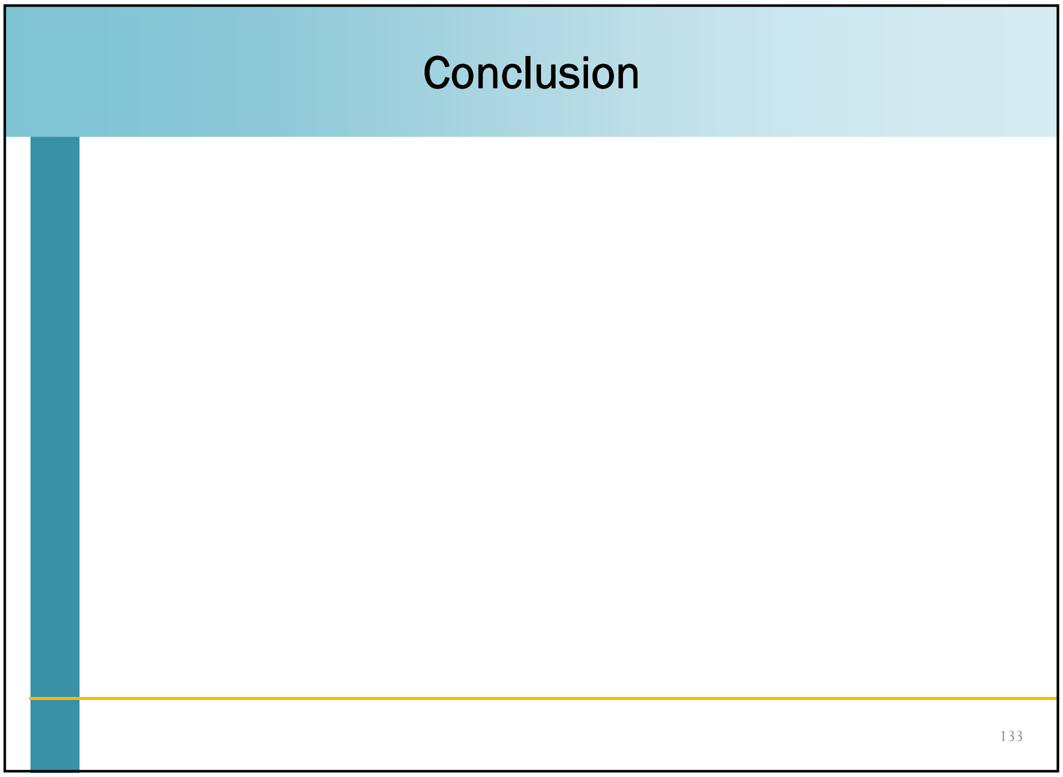
131



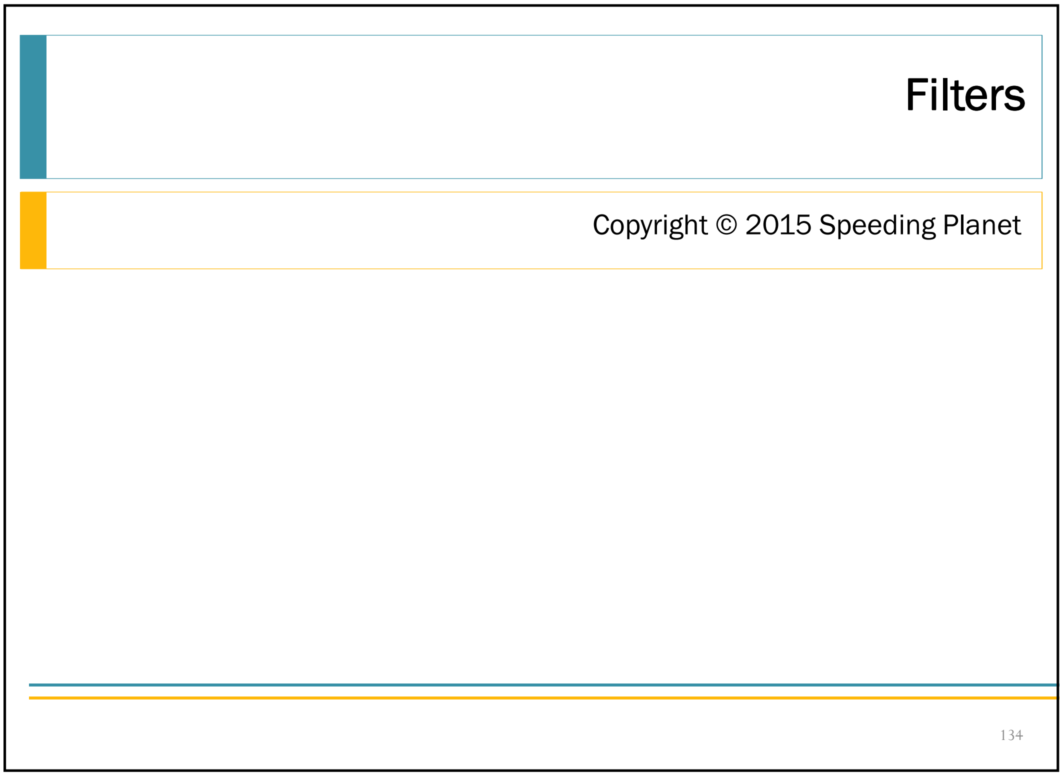
Exercise: Using a cache

- Chapter: Ajax
- Exercises\using-cache.html
- Exercises\using-cache.js

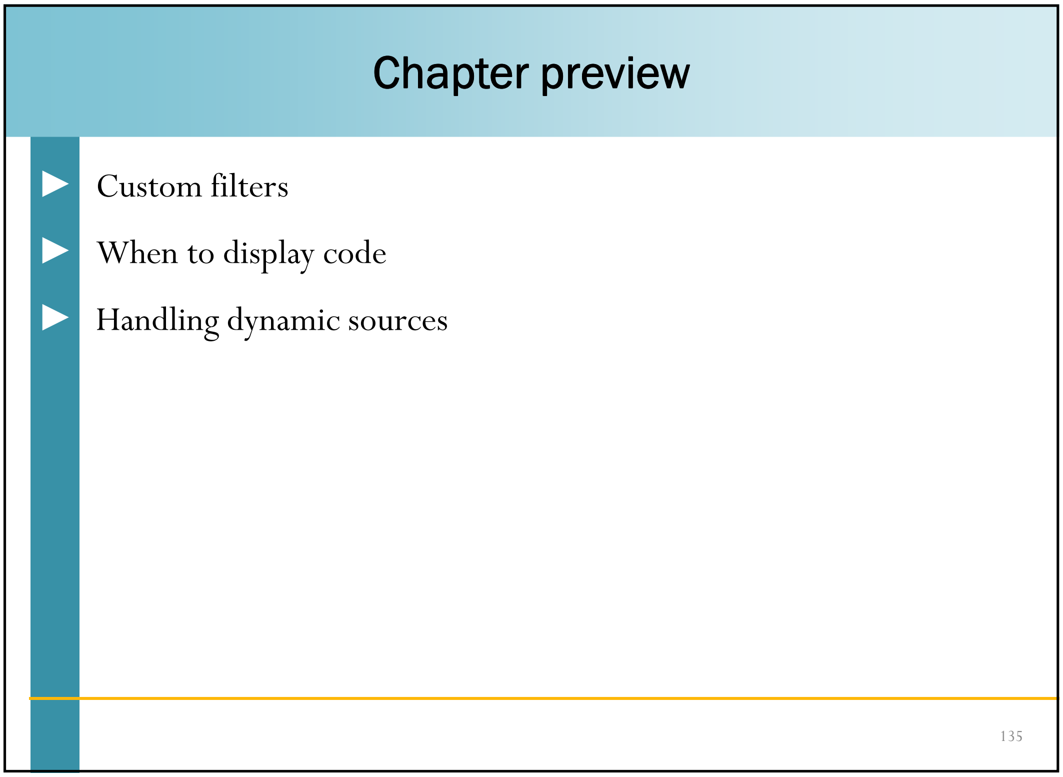
132



Conclusion

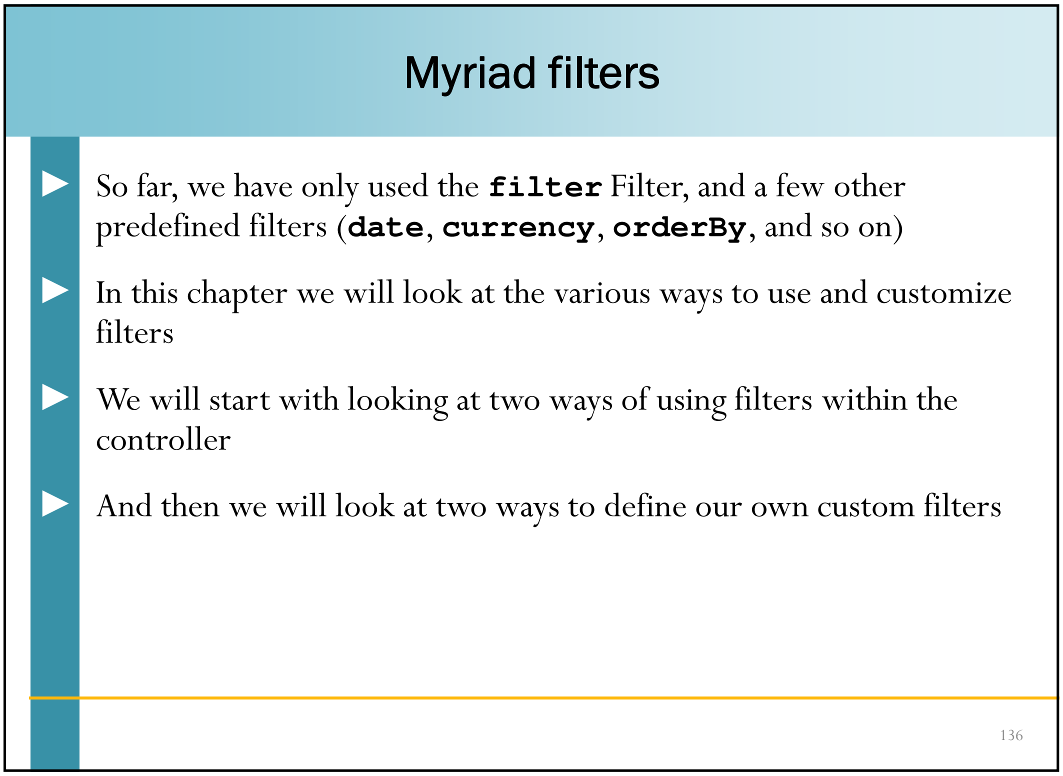


Filters



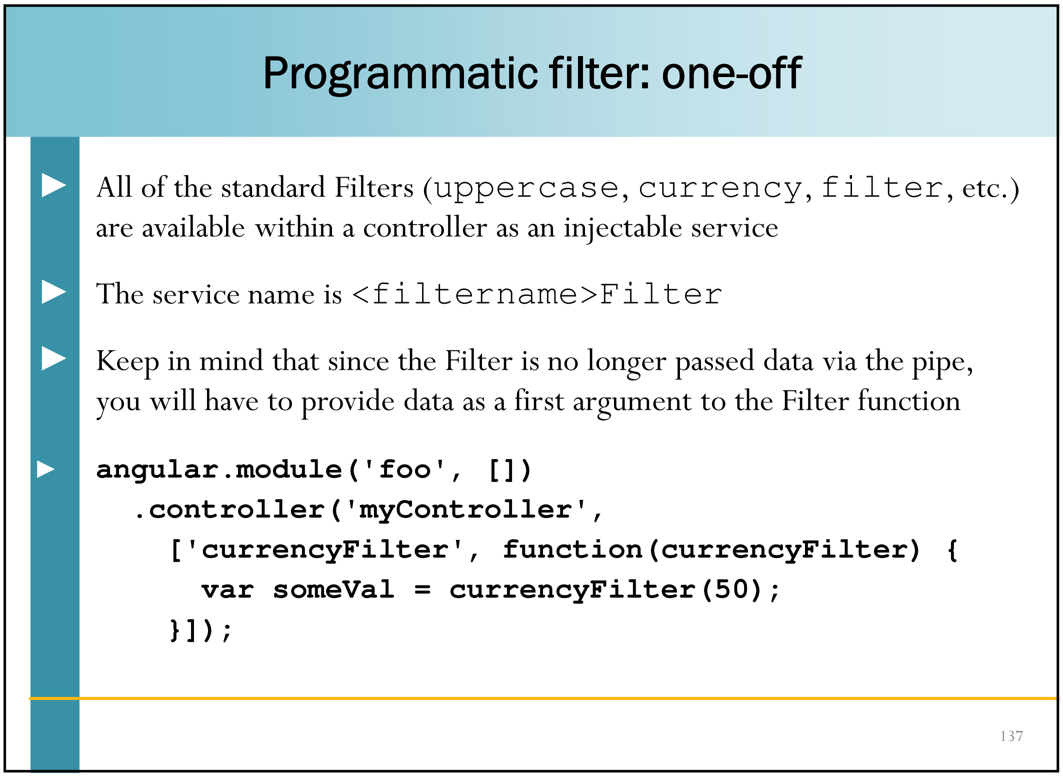
Chapter preview

- Custom filters
- When to display code
- Handling dynamic sources



Myriad filters

- So far, we have only used the **`filter`** Filter, and a few other predefined filters (**`date`**, **`currency`**, **`orderBy`**, and so on)
- In this chapter we will look at the various ways to use and customize filters
- We will start with looking at two ways of using filters within the controller
- And then we will look at two ways to define our own custom filters

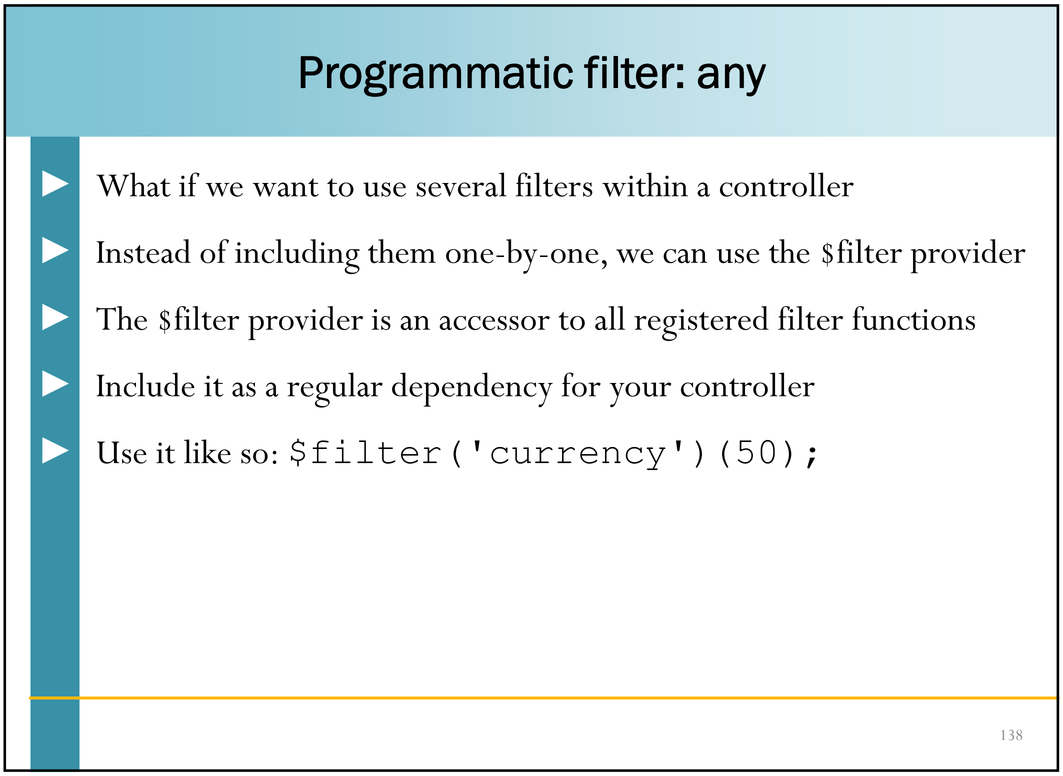


Programmatic filter: one-off

- All of the standard Filters (`uppercase`, `currency`, `filter`, etc.) are available within a controller as an injectable service
- The service name is `<filtername>Filter`
- Keep in mind that since the Filter is no longer passed data via the pipe, you will have to provide data as a first argument to the Filter function

-
  ```
  angular.module('foo', [])
    .controller('myController',
      ['currencyFilter', function(currencyFilter) {
        var someVal = currencyFilter(50);
      }]);
  ```

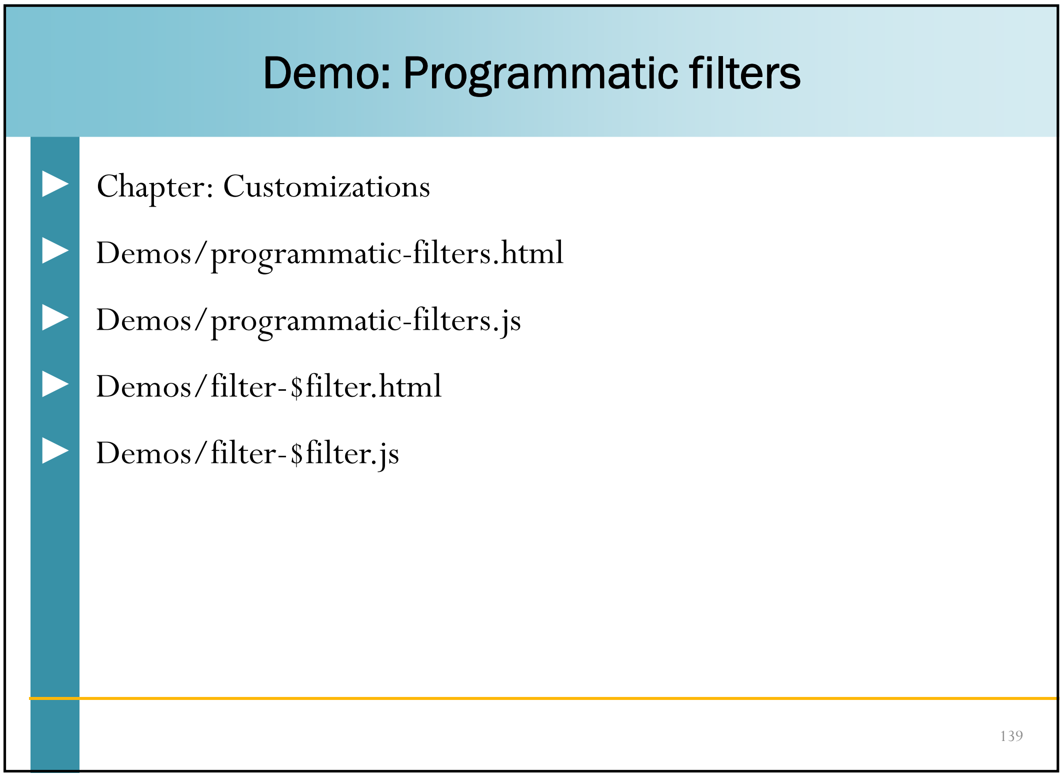
137



Programmatic filter: any

- What if we want to use several filters within a controller
- Instead of including them one-by-one, we can use the $filter provider
- The $filter provider is an accessor to all registered filter functions
- Include it as a regular dependency for your controller
- Use it like so: `$filter('currency')(50);`

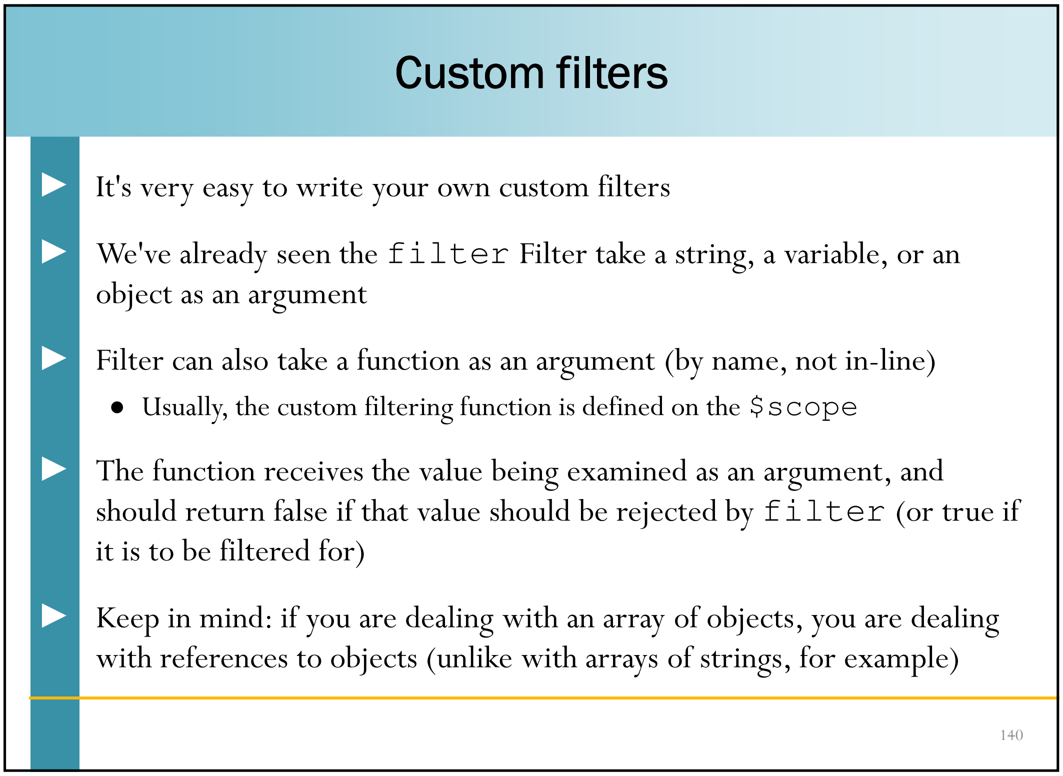
138



Demo: Programmatic filters

- Chapter: Customizations
- Demos/programmatic-filters.html
- Demos/programmatic-filters.js
- Demos/filter-$filter.html
- Demos/filter-$filter.js

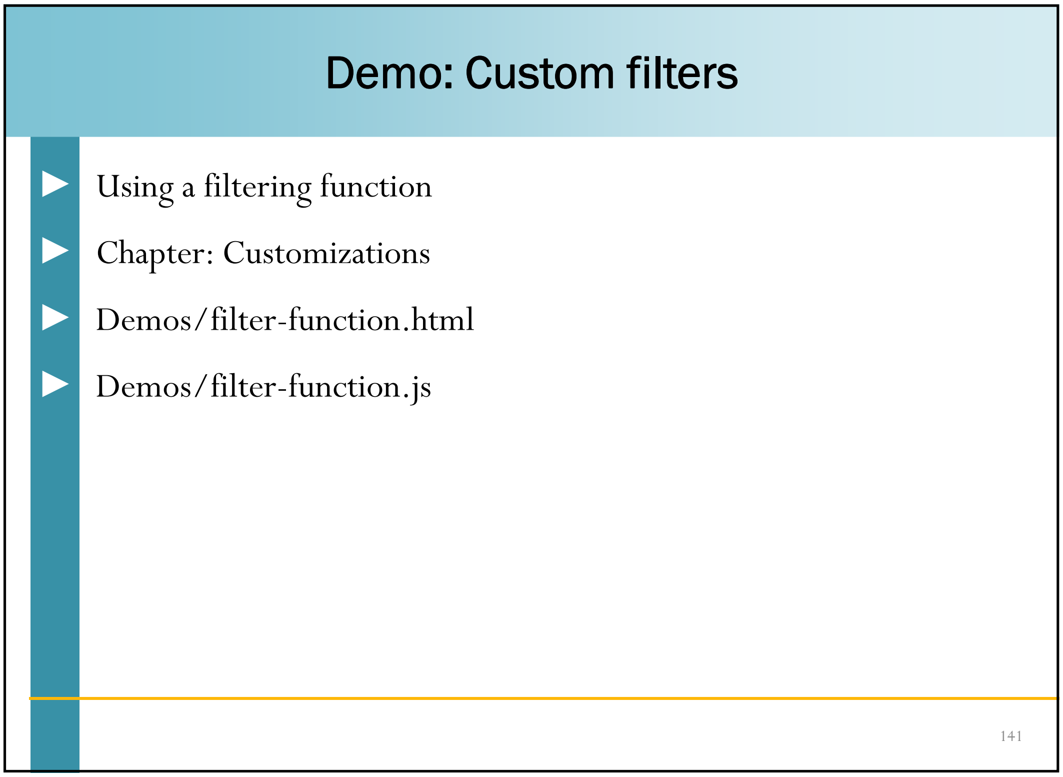
139



Custom filters

- It's very easy to write your own custom filters
- We've already seen the `filter` Filter take a string, a variable, or an object as an argument
- Filter can also take a function as an argument (by name, not in-line)
  - Usually, the custom filtering function is defined on the `$scope`
- The function receives the value being examined as an argument, and should return false if that value should be rejected by `filter` (or true if it is to be filtered for)
- Keep in mind: if you are dealing with an array of objects, you are dealing with references to objects (unlike with arrays of strings, for example)

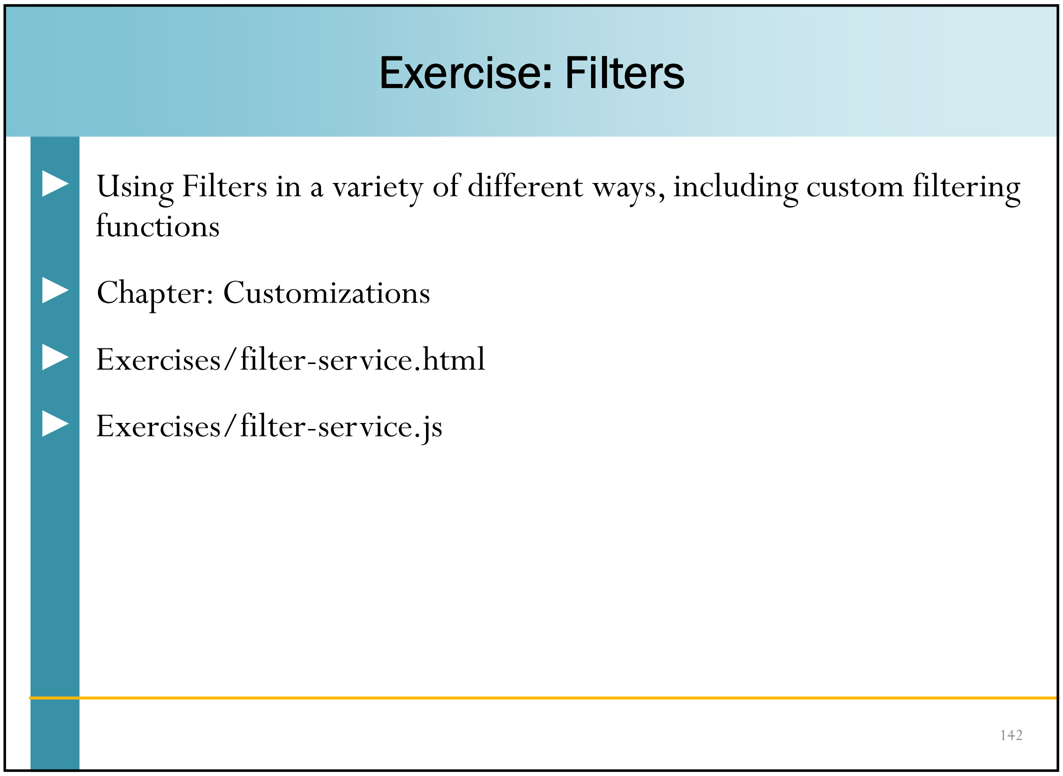
140



Demo: Custom filters

- Using a filtering function
- Chapter: Customizations
- Demos/filter-function.html
- Demos/filter-function.js

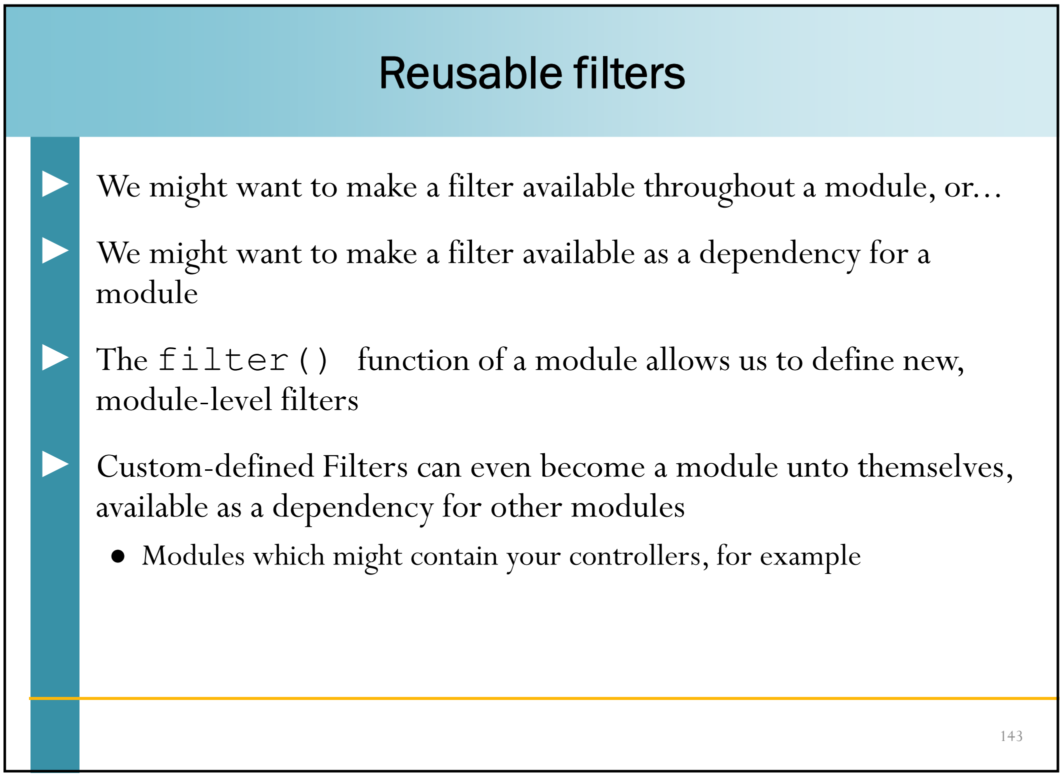
141



Exercise: Filters

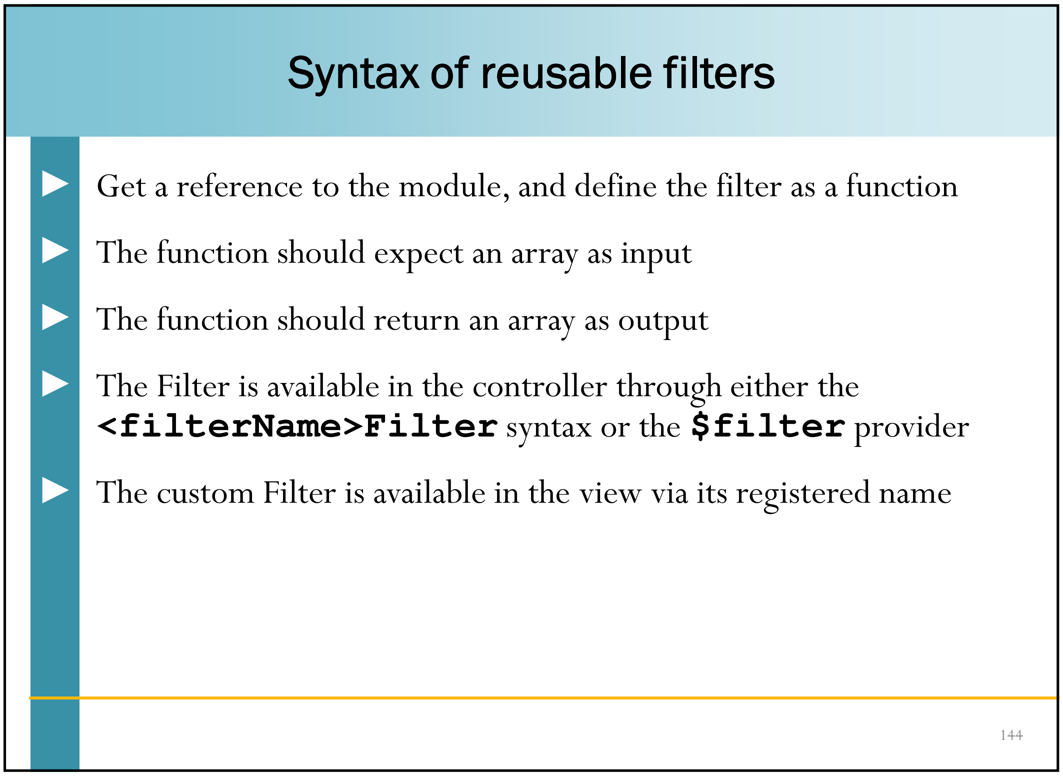
- Using Filters in a variety of different ways, including custom filtering functions
- Chapter: Customizations
- Exercises/filter-service.html
- Exercises/filter-service.js

142



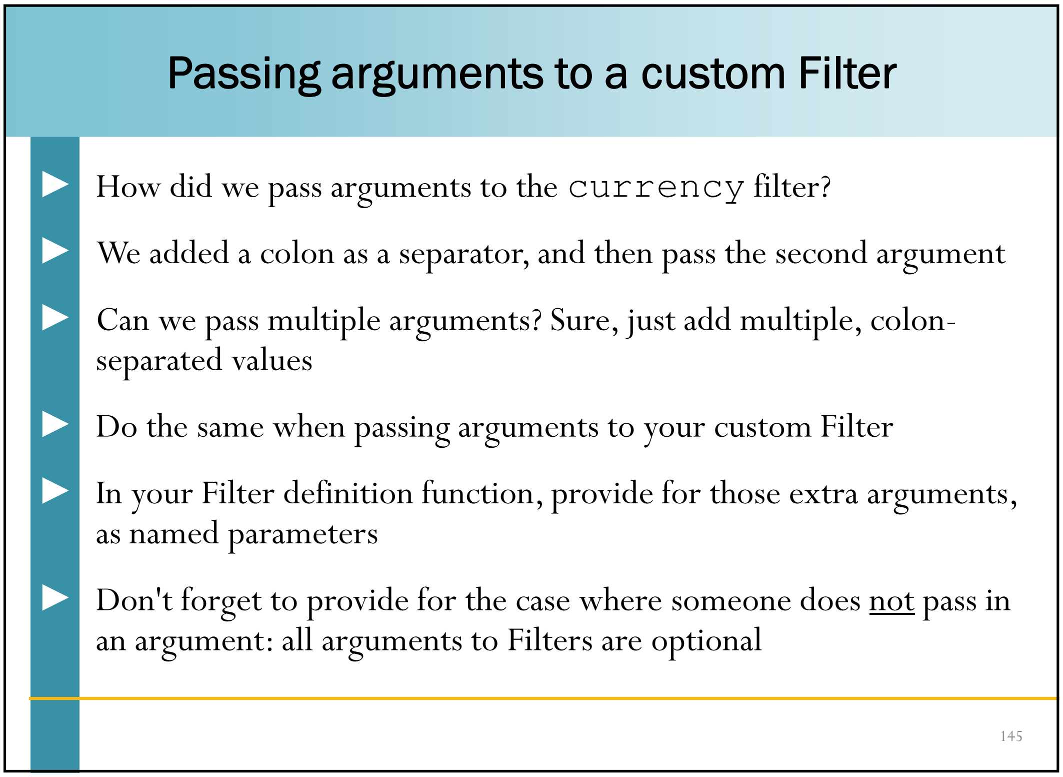
Reusable filters

- We might want to make a filter available throughout a module, or…
- We might want to make a filter available as a dependency for a module
- The `filter()` function of a module allows us to define new, module-level filters
- Custom-defined Filters can even become a module unto themselves, available as a dependency for other modules
  - Modules which might contain your controllers, for example



Syntax of reusable filters

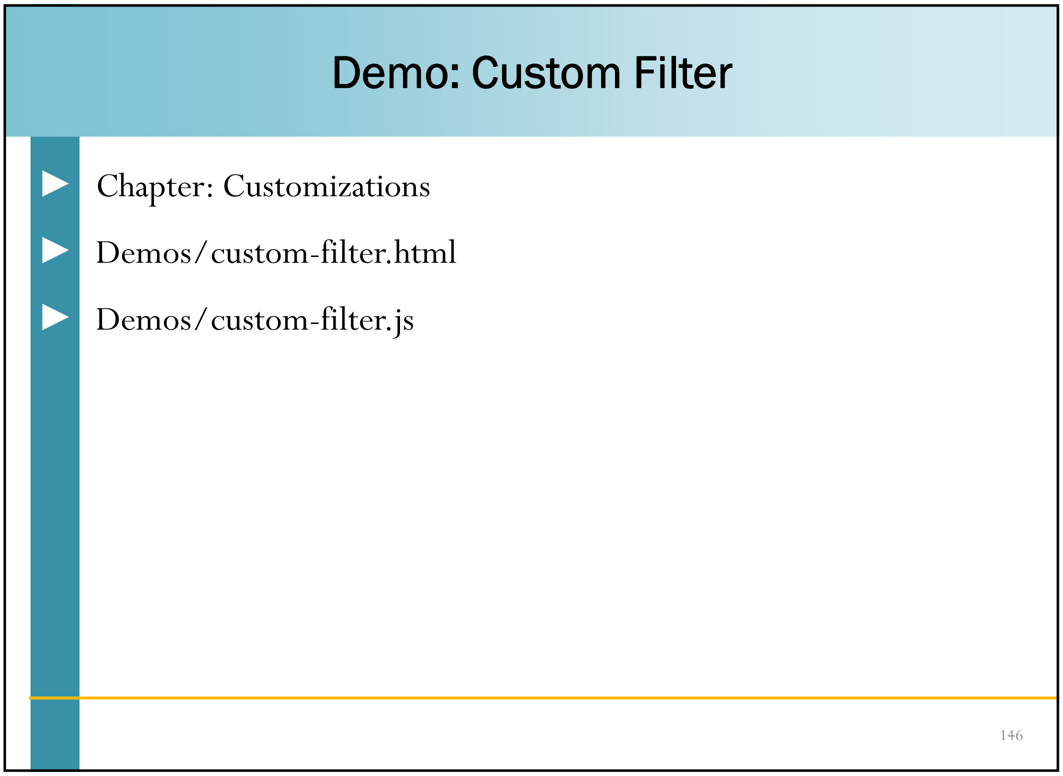
- Get a reference to the module, and define the filter as a function
- The function should expect an array as input
- The function should return an array as output
- The Filter is available in the controller through either the **`<filterName>Filter`** syntax or the **`$filter`** provider
- The custom Filter is available in the view via its registered name



Passing arguments to a custom Filter

- How did we pass arguments to the `currency` filter?
- We added a colon as a separator, and then pass the second argument
- Can we pass multiple arguments? Sure, just add multiple, colon-separated values
- Do the same when passing arguments to your custom Filter
- In your Filter definition function, provide for those extra arguments, as named parameters
- Don't forget to provide for the case where someone does not pass in an argument: all arguments to Filters are optional

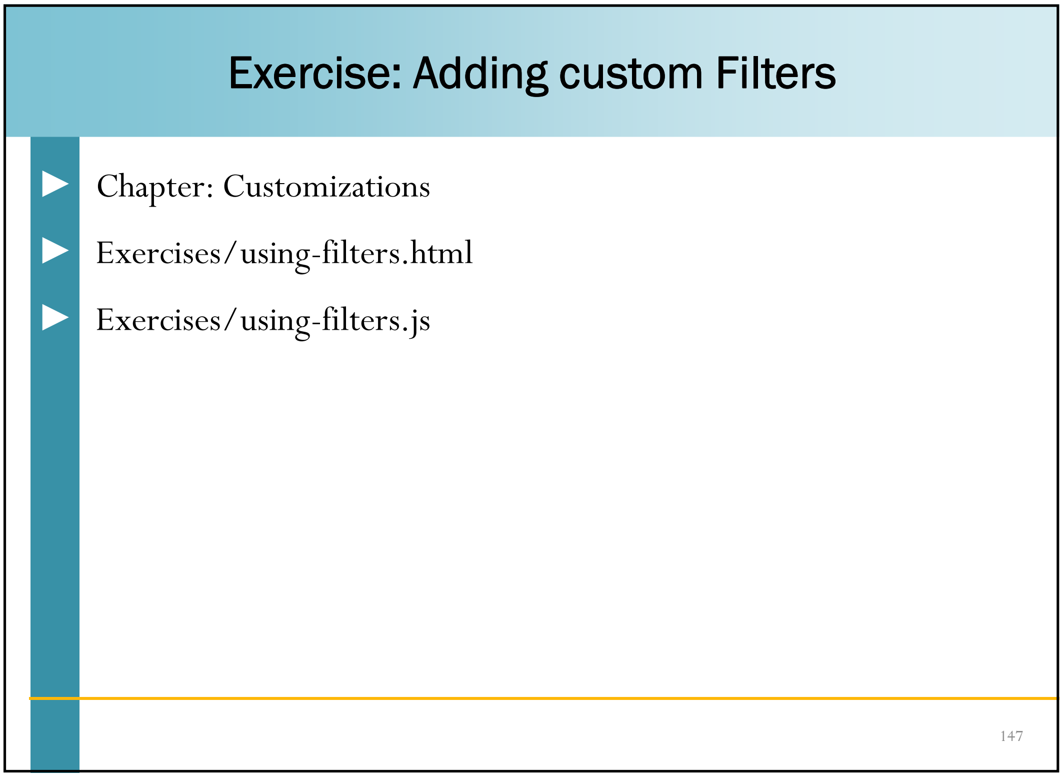
145



Demo: Custom Filter

- Chapter: Customizations
- Demos/custom-filter.html
- Demos/custom-filter.js

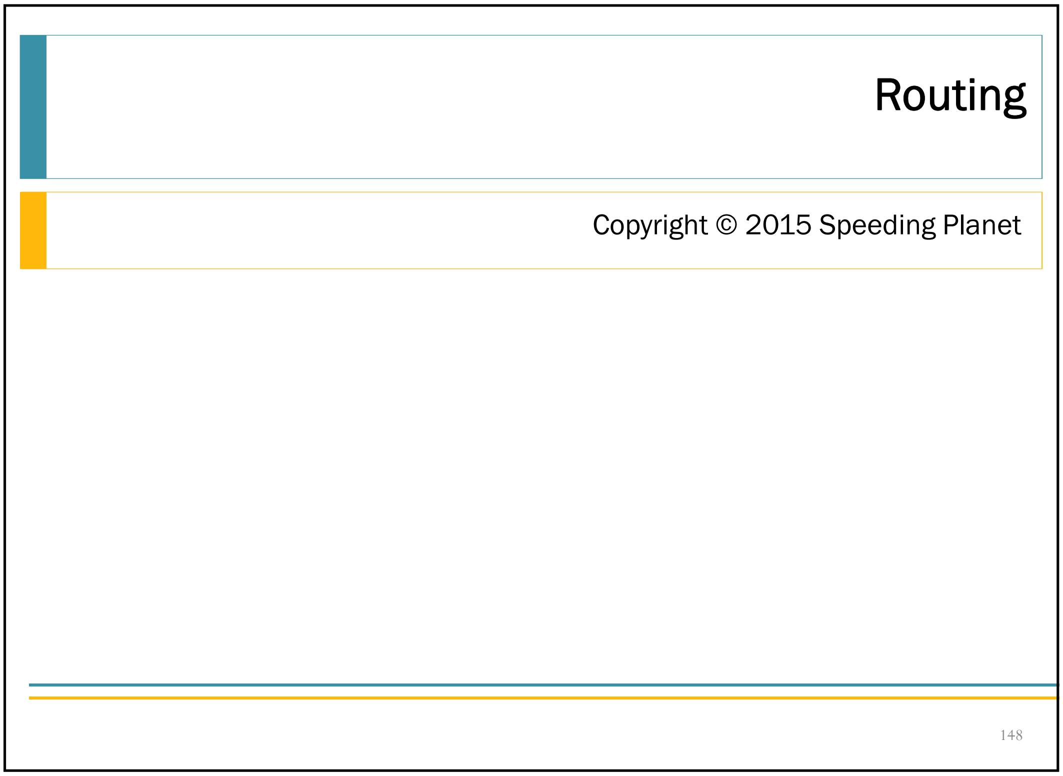
146



Exercise: Adding custom Filters

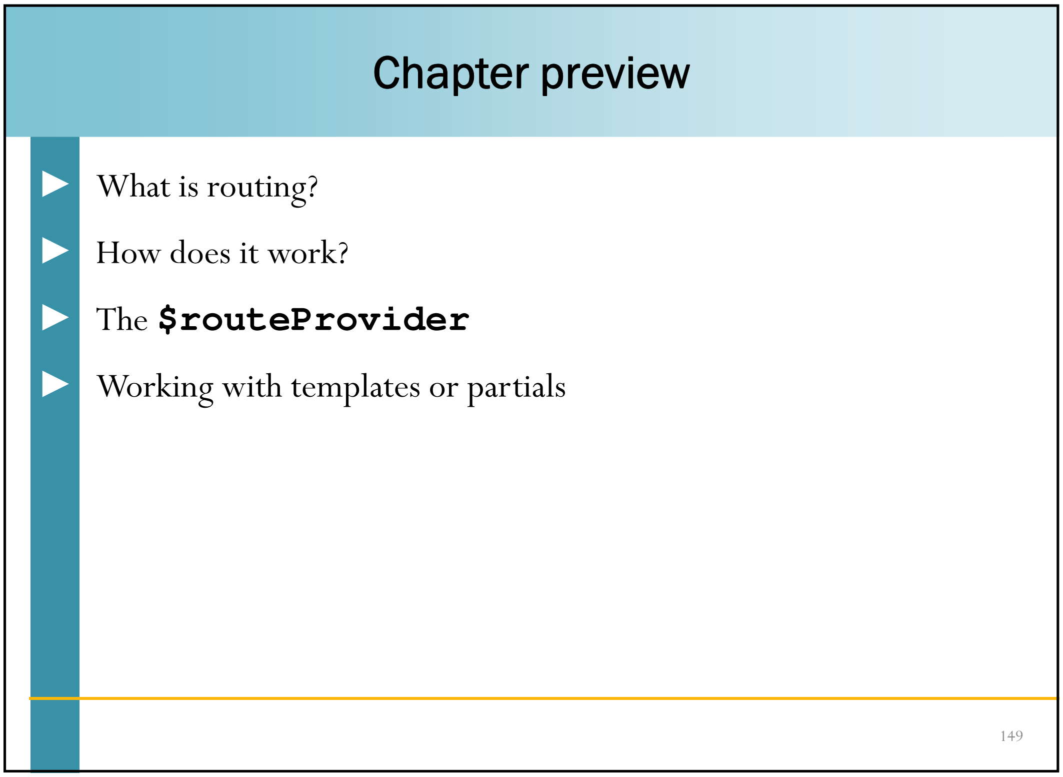
- Chapter: Customizations
- Exercises/using-filters.html
- Exercises/using-filters.js

147



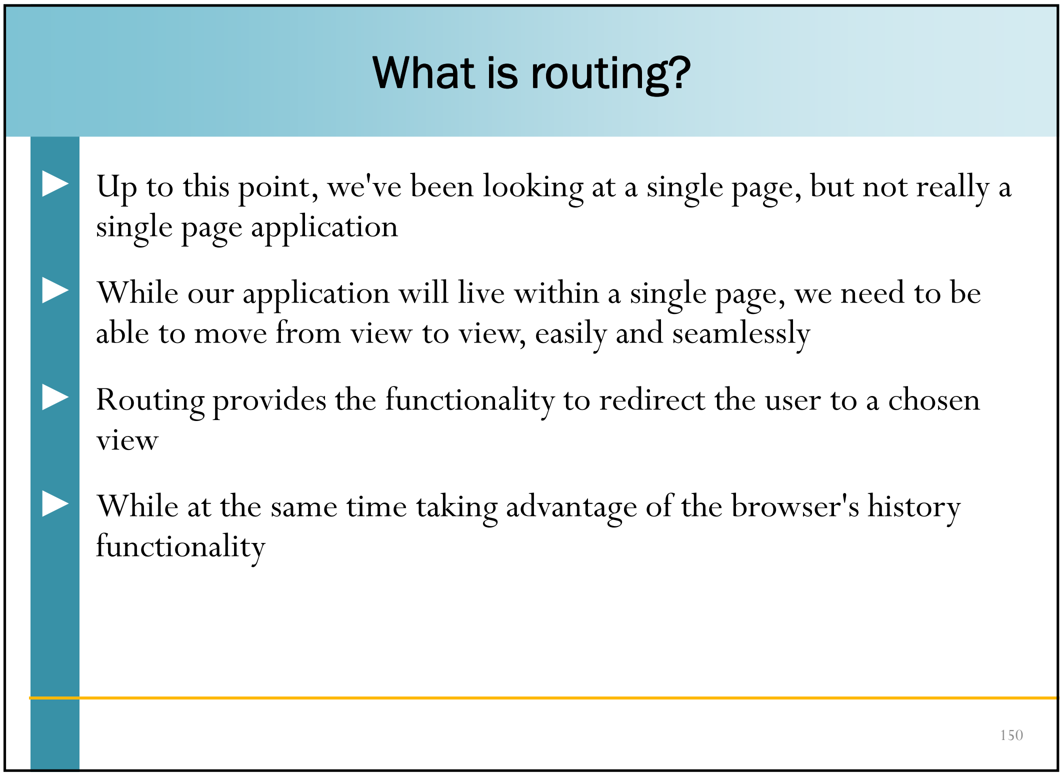
Routing

148



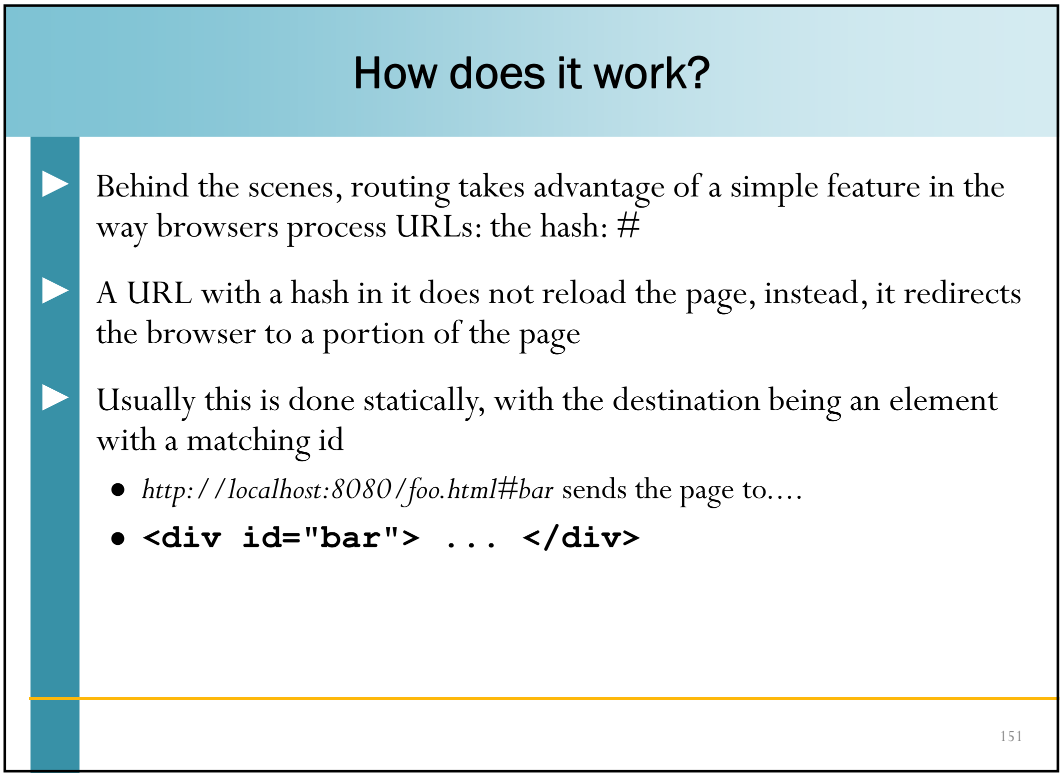
Chapter preview

- What is routing?
- How does it work?
- The **`$routeProvider`**
- Working with templates or partials



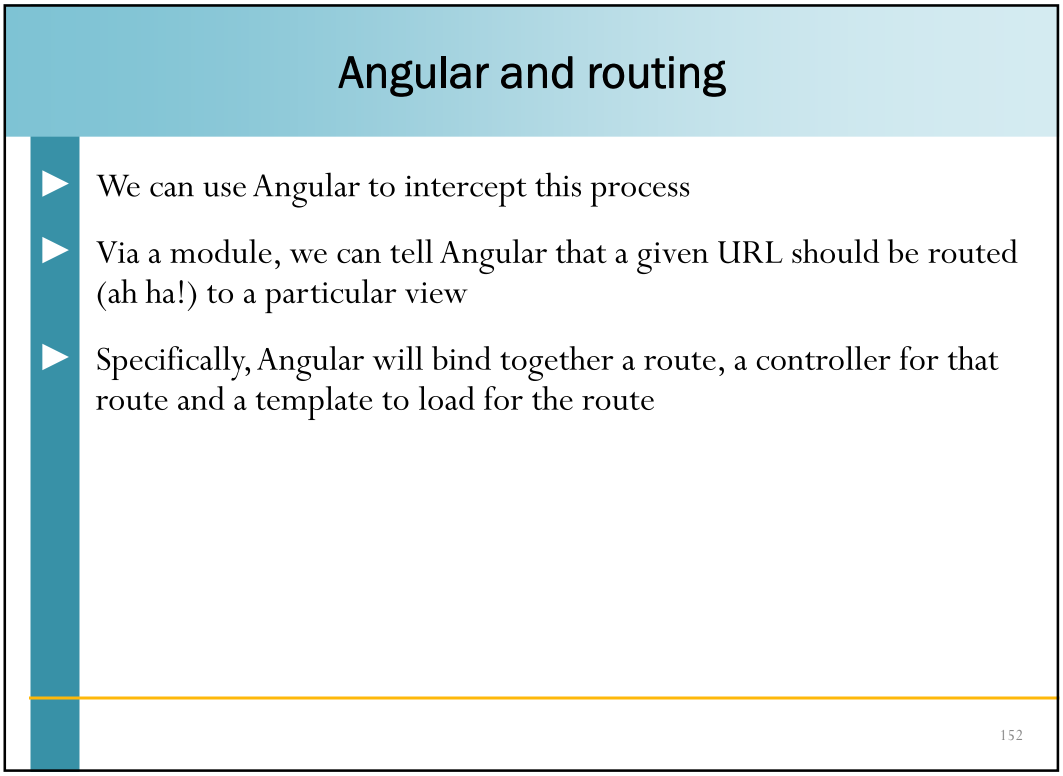
What is routing?

- Up to this point, we've been looking at a single page, but not really a single page application
- While our application will live within a single page, we need to be able to move from view to view, easily and seamlessly
- Routing provides the functionality to redirect the user to a chosen view
- While at the same time taking advantage of the browser's history functionality



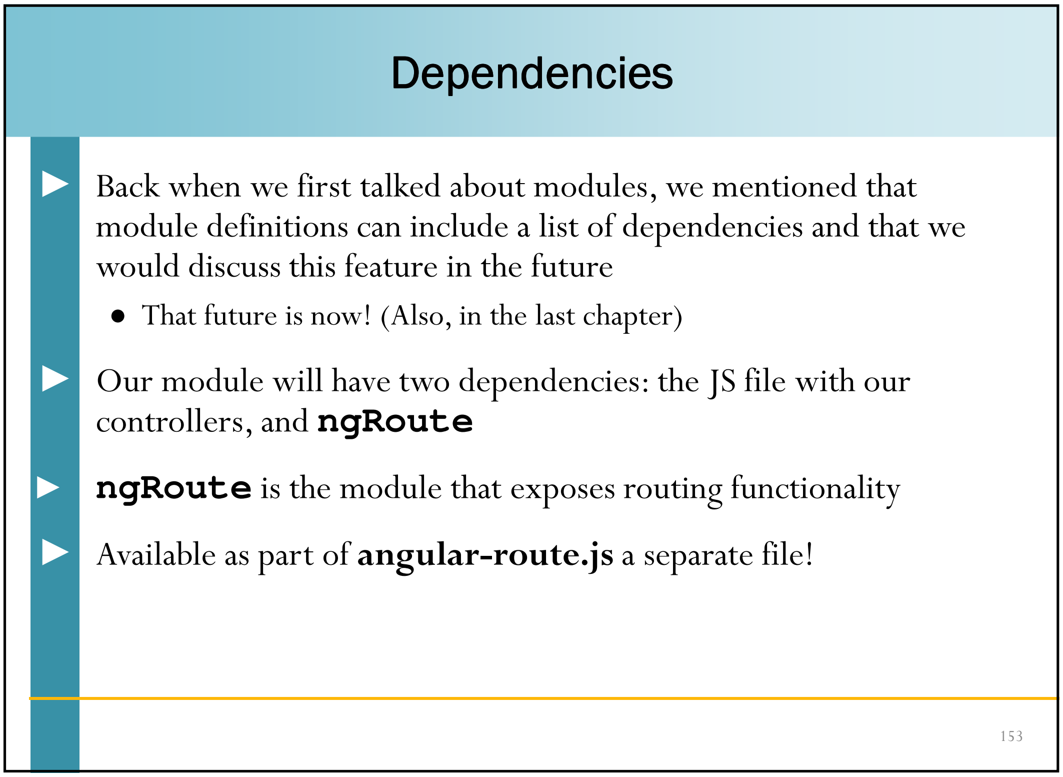
How does it work?

- Behind the scenes, routing takes advantage of a simple feature in the way browsers process URLs: the hash: #
- A URL with a hash in it does not reload the page, instead, it redirects the browser to a portion of the page
- Usually this is done statically, with the destination being an element with a matching id
  - *http://localhost:8080/foo.html#bar* sends the page to….
  - **`<div id="bar"> ... </div>`**



Angular and routing

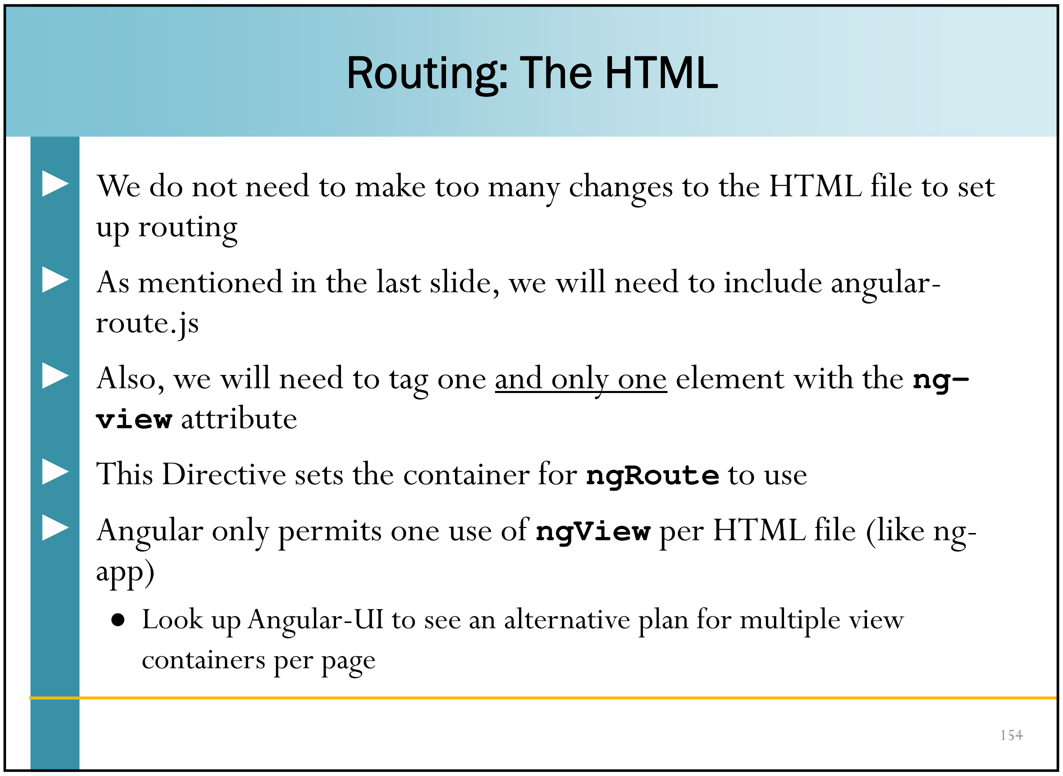
- We can use Angular to intercept this process
- Via a module, we can tell Angular that a given URL should be routed (ah ha!) to a particular view
- Specifically, Angular will bind together a route, a controller for that route and a template to load for the route



Dependencies

- Back when we first talked about modules, we mentioned that module definitions can include a list of dependencies and that we would discuss this feature in the future
  - That future is now! (Also, in the last chapter)
- Our module will have two dependencies: the JS file with our controllers, and **`ngRoute`**
- **`ngRoute`** is the module that exposes routing functionality
- Available as part of **angular-route.js** a separate file!

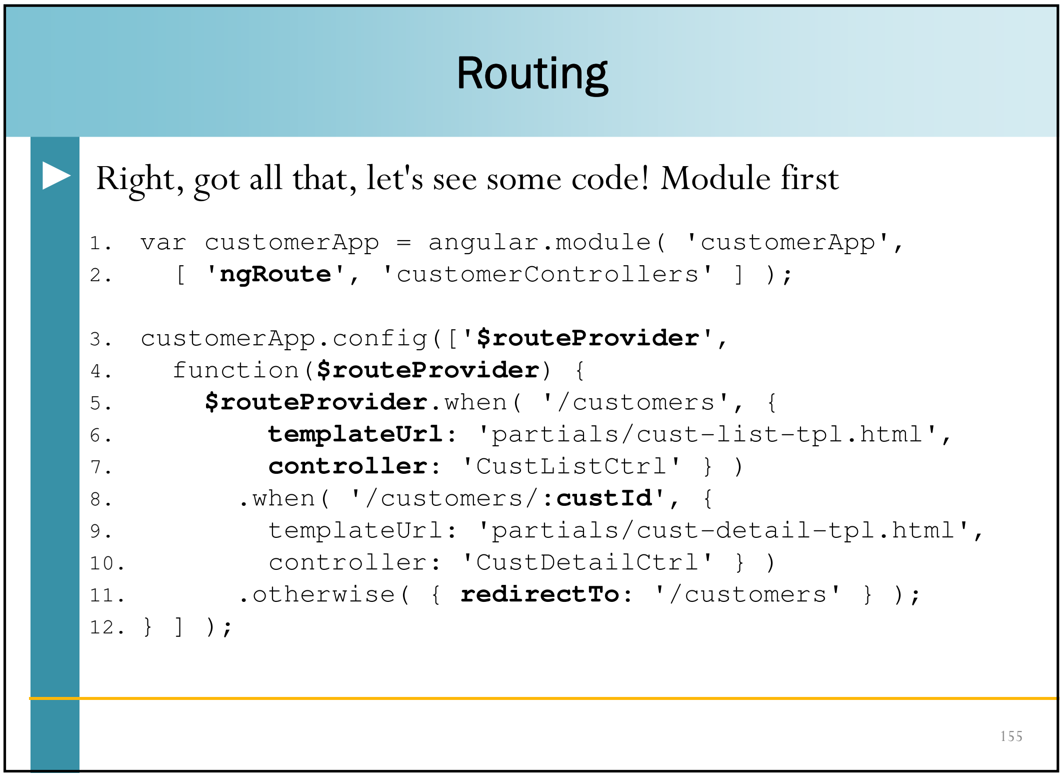
153



Routing: The HTML

- We do not need to make too many changes to the HTML file to set up routing
- As mentioned in the last slide, we will need to include angular-route.js
- Also, we will need to tag one <u>and only one</u> element with the **`ng-view`** attribute
- This Directive sets the container for **`ngRoute`** to use
- Angular only permits one use of **`ngView`** per HTML file (like ng-app)
  - Look up Angular-UI to see an alternative plan for multiple view containers per page

154

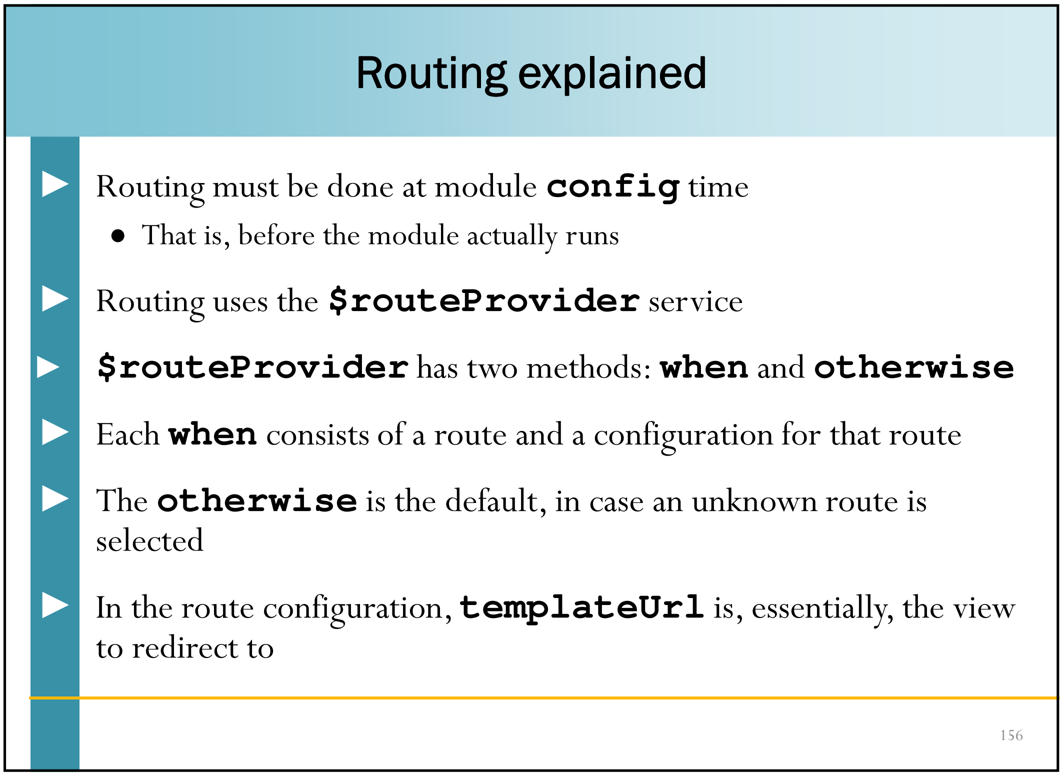


Routing

- Right, got all that, let's see some code! Module first

```
var customerApp = angular.module( 'customerApp',
  [ 'ngRoute', 'customerControllers' ] );

customerApp.config(['$routeProvider',
  function($routeProvider) {
    $routeProvider.when( '/customers', {
        templateUrl: 'partials/cust-list-tpl.html',
        controller: 'CustListCtrl' } )
      .when( '/customers/:custId', {
        templateUrl: 'partials/cust-detail-tpl.html',
        controller: 'CustDetailCtrl' } )
      .otherwise( { redirectTo: '/customers' } );
} ] );
```

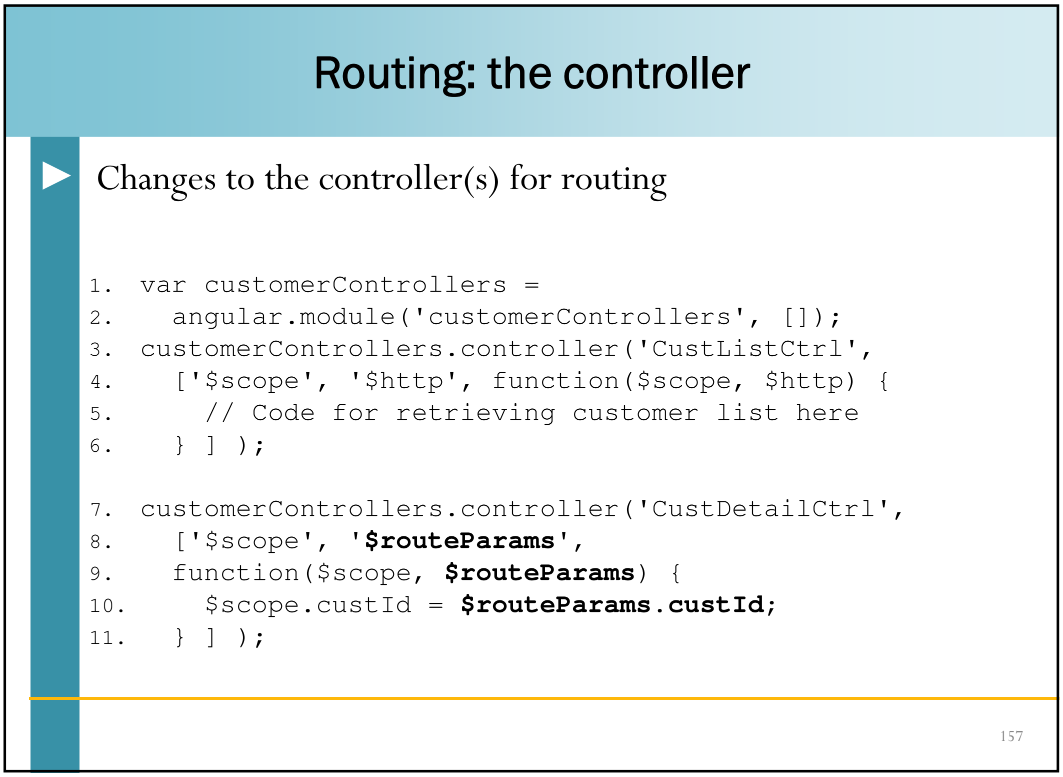
155



Routing explained

- Routing must be done at module **`config`** time
  - That is, before the module actually runs
- Routing uses the **`$routeProvider`** service
- **`$routeProvider`** has two methods: **`when`** and **`otherwise`**
- Each **`when`** consists of a route and a configuration for that route
- The **`otherwise`** is the default, in case an unknown route is selected
- In the route configuration, **`templateUrl`** is, essentially, the view to redirect to

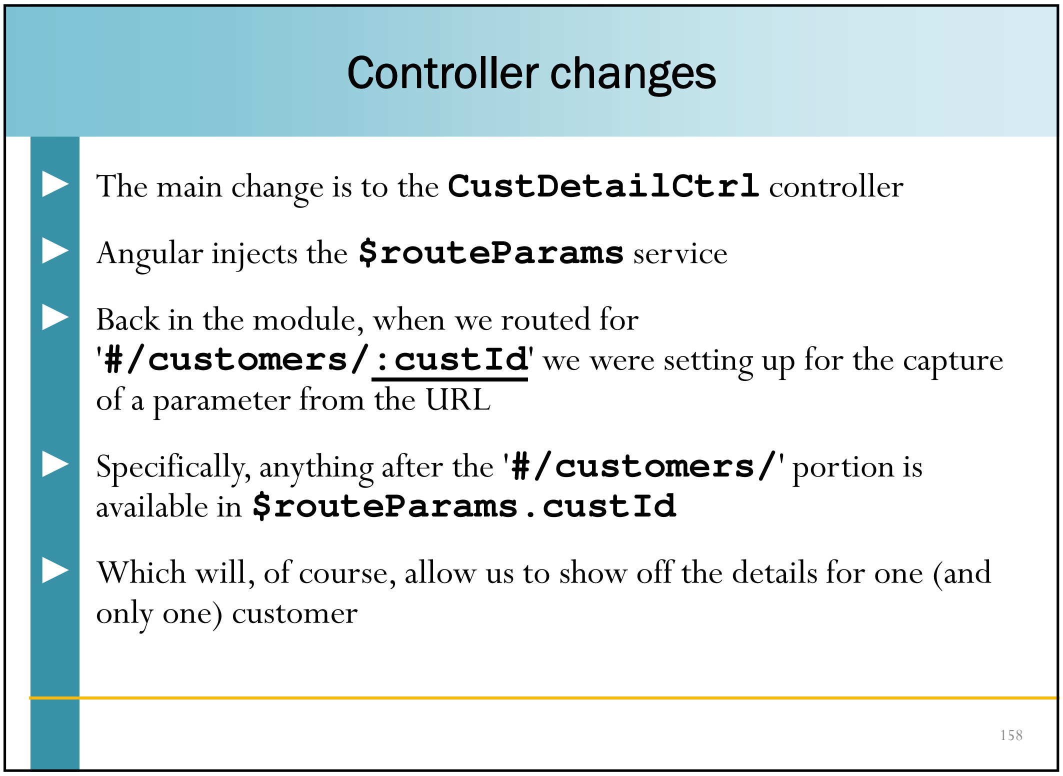
156



Routing: the controller

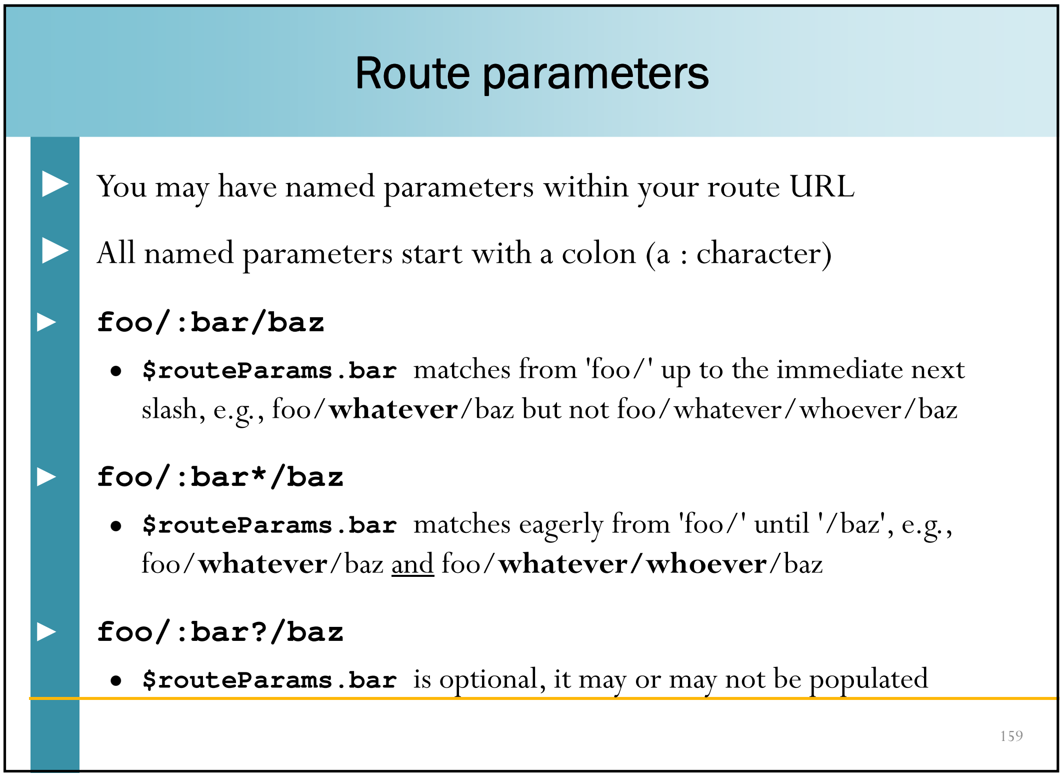
- Changes to the controller(s) for routing

```
var customerControllers =
  angular.module('customerControllers', []);
customerControllers.controller('CustListCtrl',
  ['$scope', '$http', function($scope, $http) {
    // Code for retrieving customer list here
  } ] );

customerControllers.controller('CustDetailCtrl',
  ['$scope', '$routeParams',
  function($scope, $routeParams) {
    $scope.custId = $routeParams.custId;
  } ] );
```



Controller changes

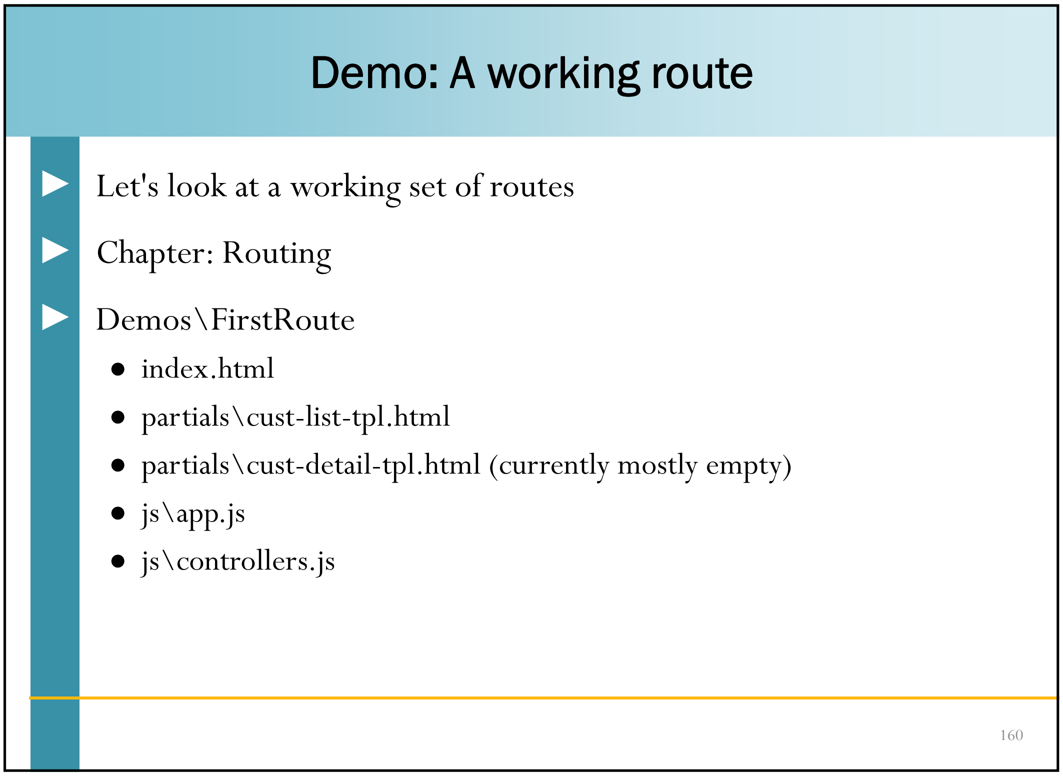
- The main change is to the **`CustDetailCtrl`** controller
- Angular injects the **`$routeParams`** service
- Back in the module, when we routed for '**`#/customers/:custId`**' we were setting up for the capture of a parameter from the URL
- Specifically, anything after the '**`#/customers/`**' portion is available in **`$routeParams.custId`**
- Which will, of course, allow us to show off the details for one (and only one) customer



Route parameters

- You may have named parameters within your route URL
- All named parameters start with a colon (a : character)
- **`foo/:bar/baz`**
  - **`$routeParams.bar`** matches from 'foo/' up to the immediate next slash, e.g., foo/**whatever**/baz but not foo/whatever/whoever/baz
- **`foo/:bar*/baz`**
  - **`$routeParams.bar`** matches eagerly from 'foo/' until '/baz', e.g., foo/**whatever**/baz and foo/**whatever/whoever**/baz
- **`foo/:bar?/baz`**
  - **`$routeParams.bar`** is optional, it may or may not be populated

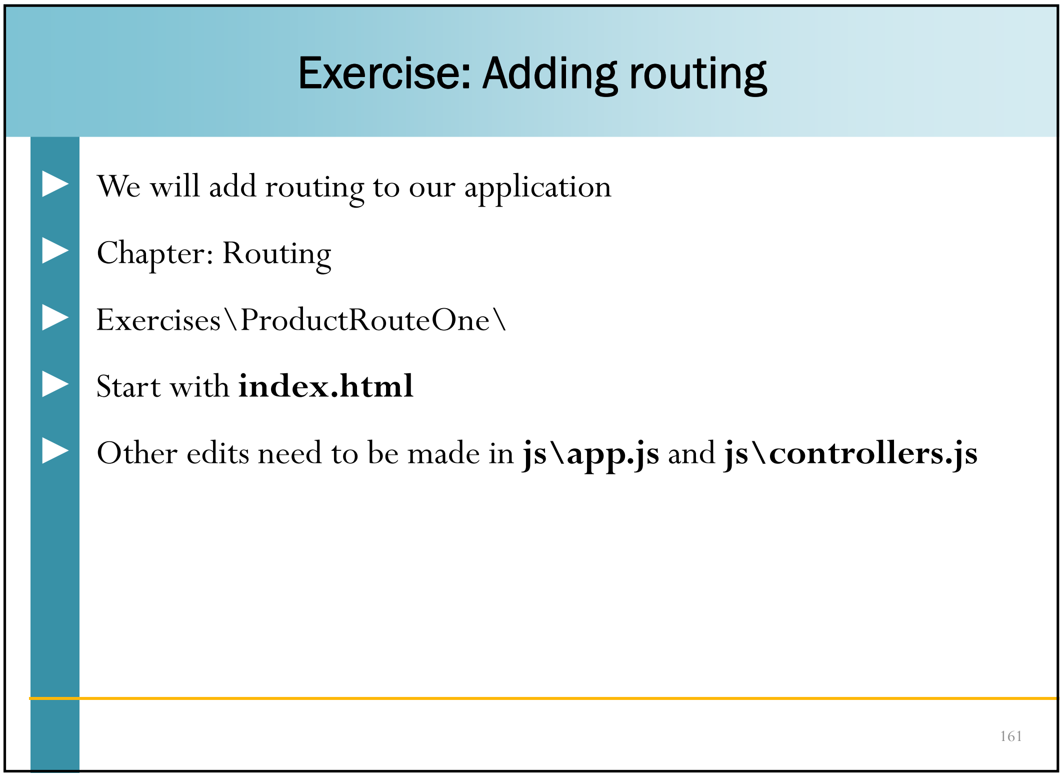
159



Demo: A working route

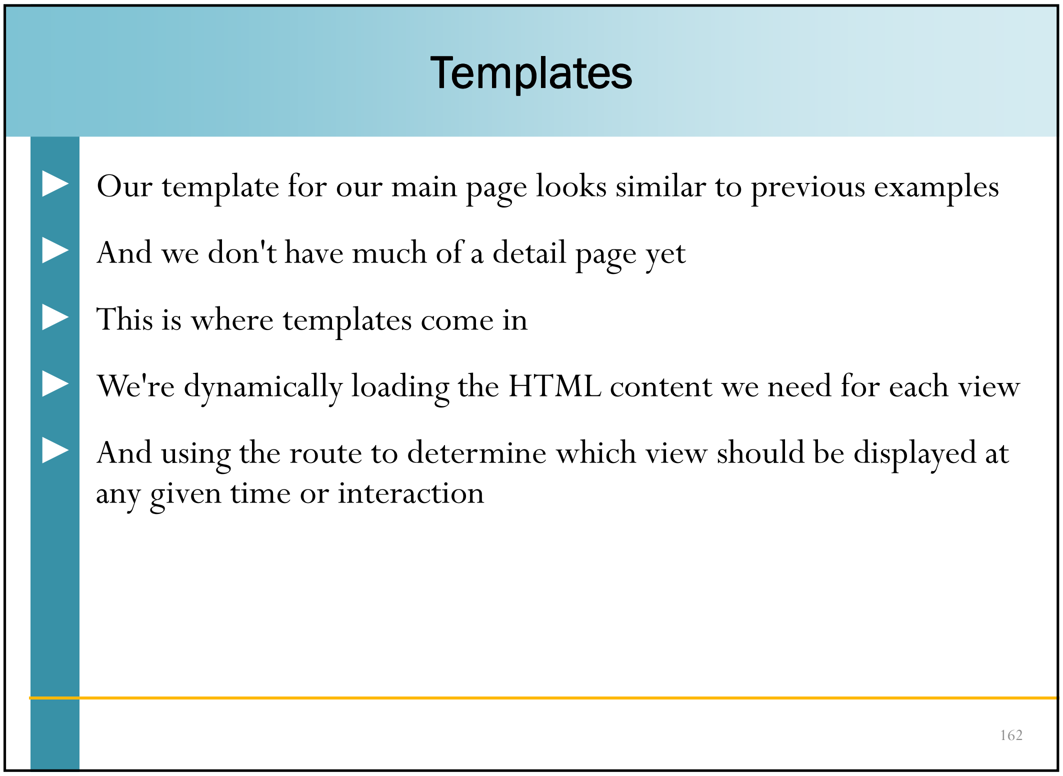
- Let's look at a working set of routes
- Chapter: Routing
- Demos\FirstRoute
  - index.html
  - partials\cust-list-tpl.html
  - partials\cust-detail-tpl.html (currently mostly empty)
  - js\app.js
  - js\controllers.js

160



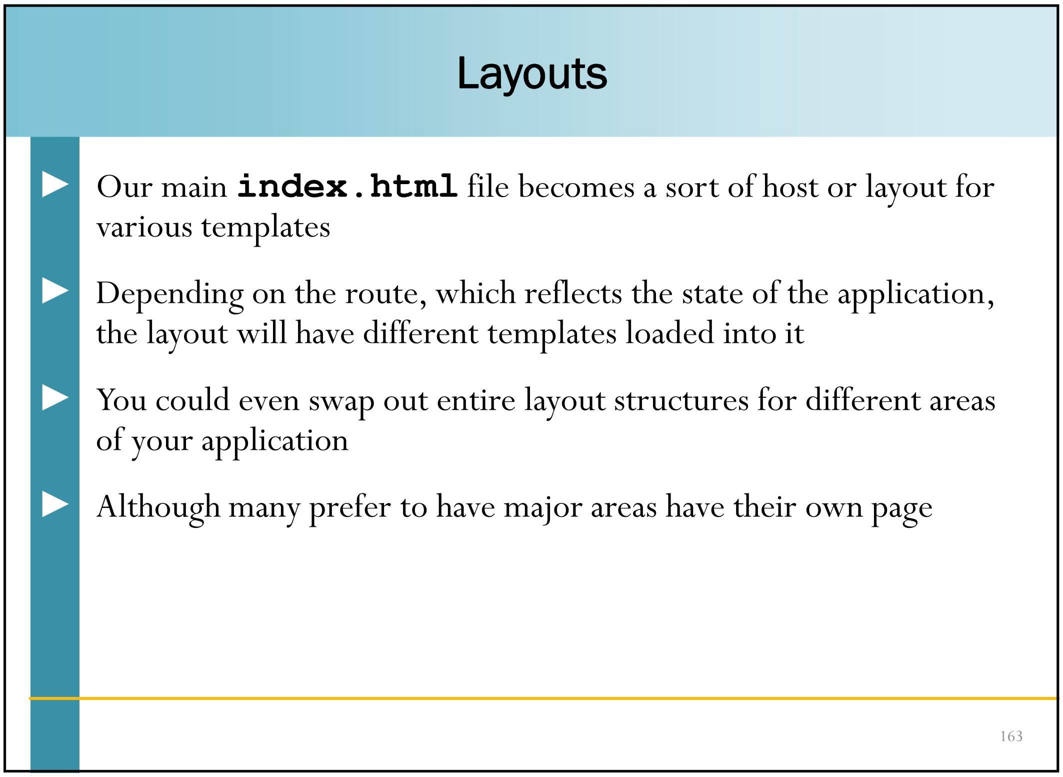
Exercise: Adding routing

- We will add routing to our application
- Chapter: Routing
- Exercises\ProductRouteOne\
- Start with **index.html**
- Other edits need to be made in **js\app.js** and **js\controllers.js**



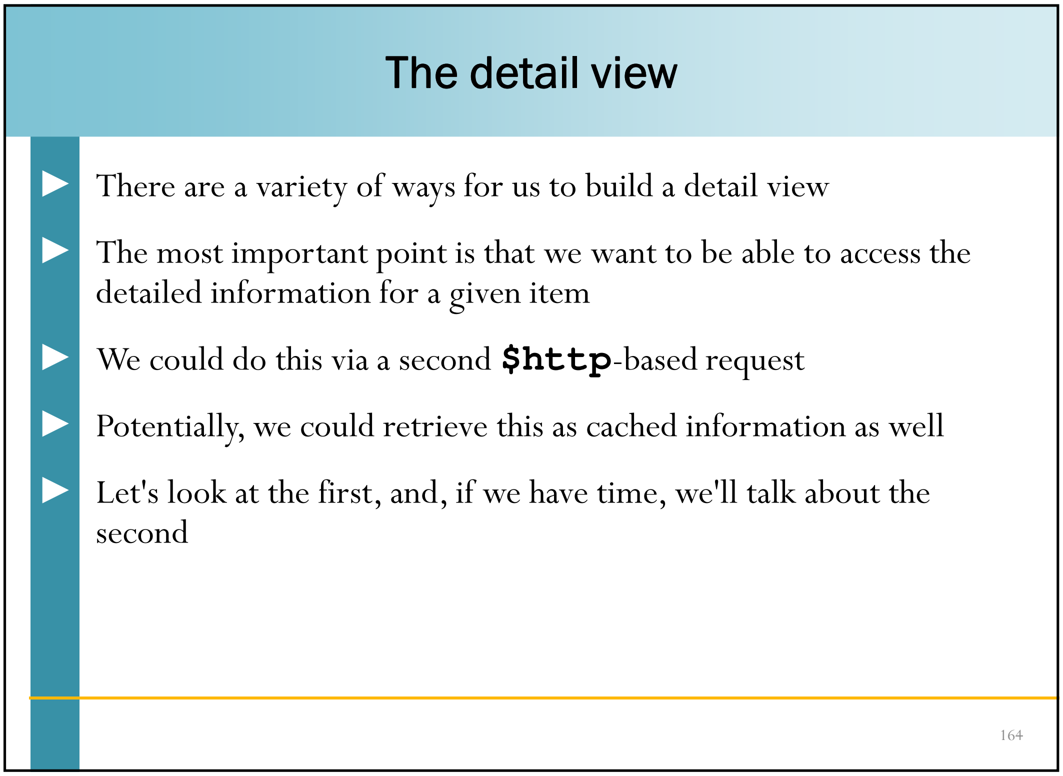
Templates

- Our template for our main page looks similar to previous examples
- And we don't have much of a detail page yet
- This is where templates come in
- We're dynamically loading the HTML content we need for each view
- And using the route to determine which view should be displayed at any given time or interaction



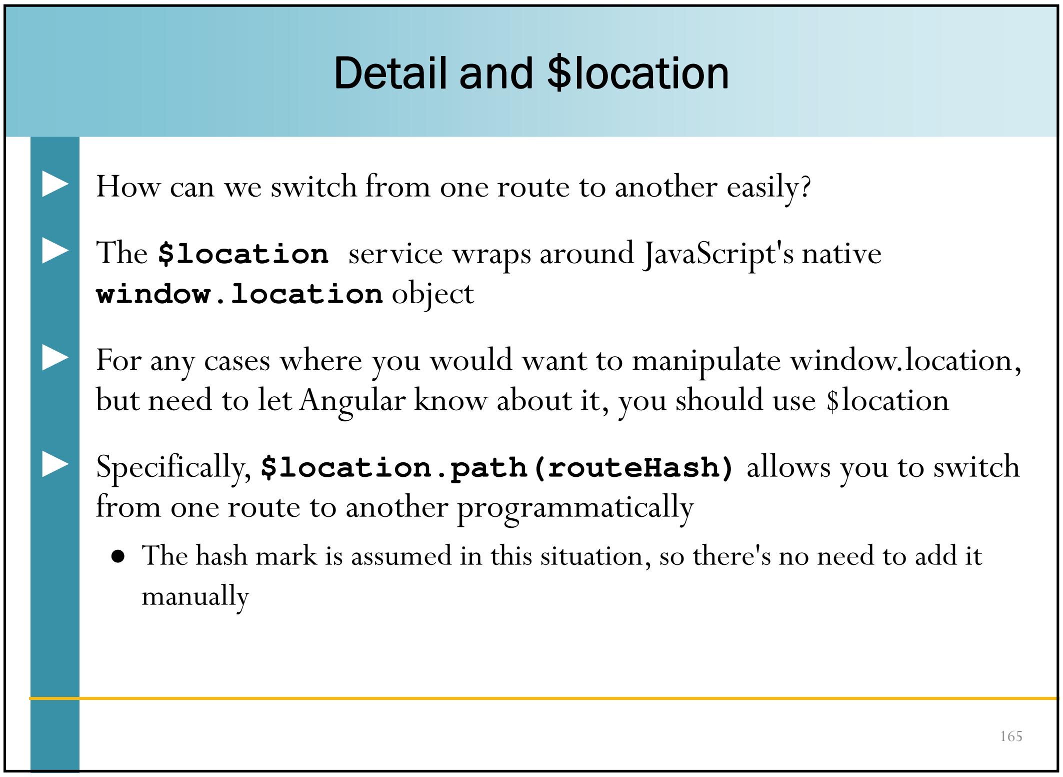
Layouts

- Our main **`index.html`** file becomes a sort of host or layout for various templates
- Depending on the route, which reflects the state of the application, the layout will have different templates loaded into it
- You could even swap out entire layout structures for different areas of your application
- Although many prefer to have major areas have their own page



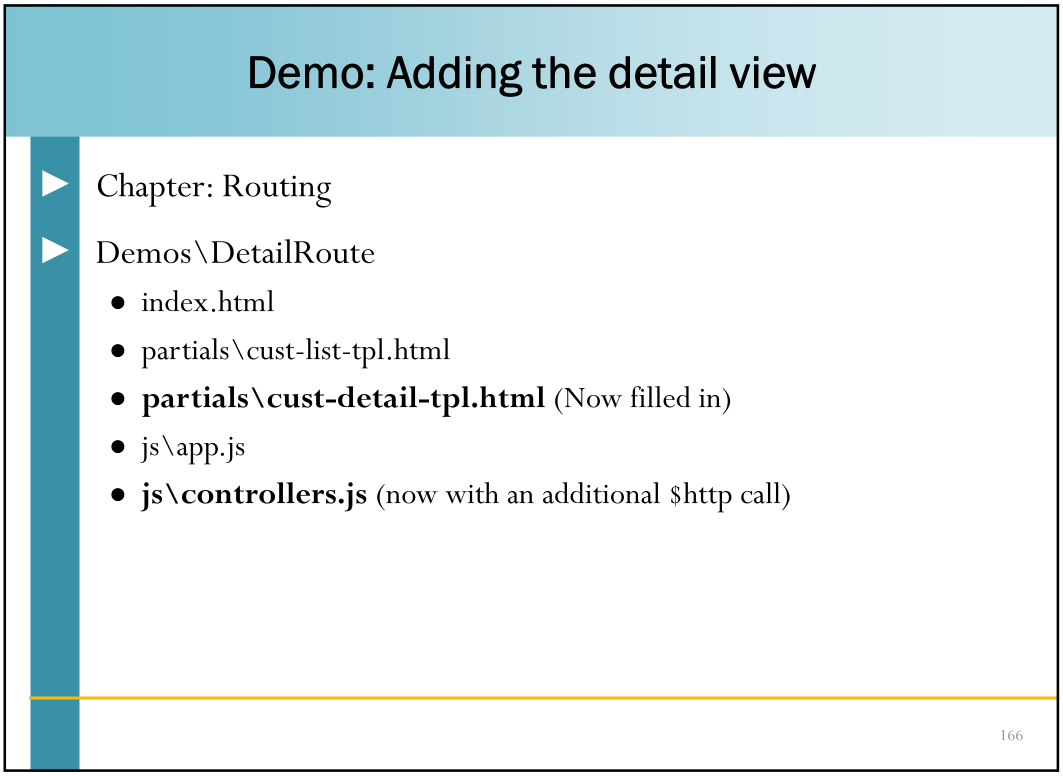
The detail view

- There are a variety of ways for us to build a detail view
- The most important point is that we want to be able to access the detailed information for a given item
- We could do this via a second **`$http`**-based request
- Potentially, we could retrieve this as cached information as well
- Let's look at the first, and, if we have time, we'll talk about the second



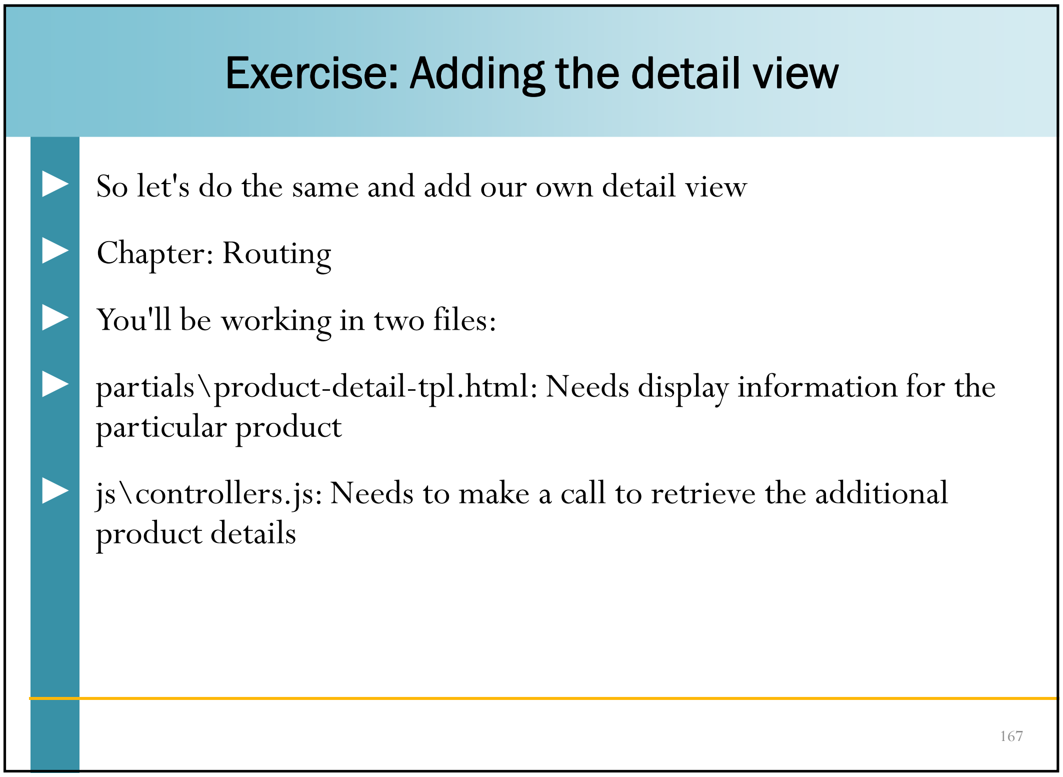
Detail and $location

- How can we switch from one route to another easily?
- The **`$location`** service wraps around JavaScript's native **`window.location`** object
- For any cases where you would want to manipulate window.location, but need to let Angular know about it, you should use $location
- Specifically, **`$location.path(routeHash)`** allows you to switch from one route to another programmatically
  - The hash mark is assumed in this situation, so there's no need to add it manually



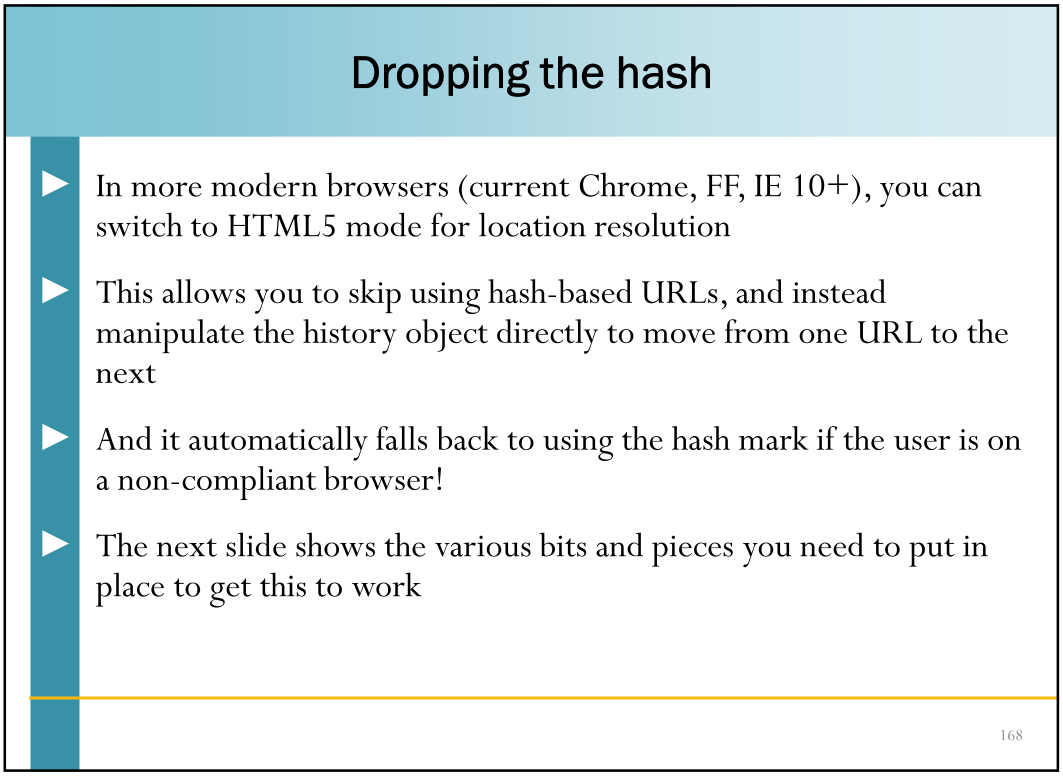
Demo: Adding the detail view

- Chapter: Routing
- Demos\DetailRoute
  - index.html
  - partials\cust-list-tpl.html
  - **partials\cust-detail-tpl.html** (Now filled in)
  - js\app.js
  - **js\controllers.js** (now with an additional $http call)



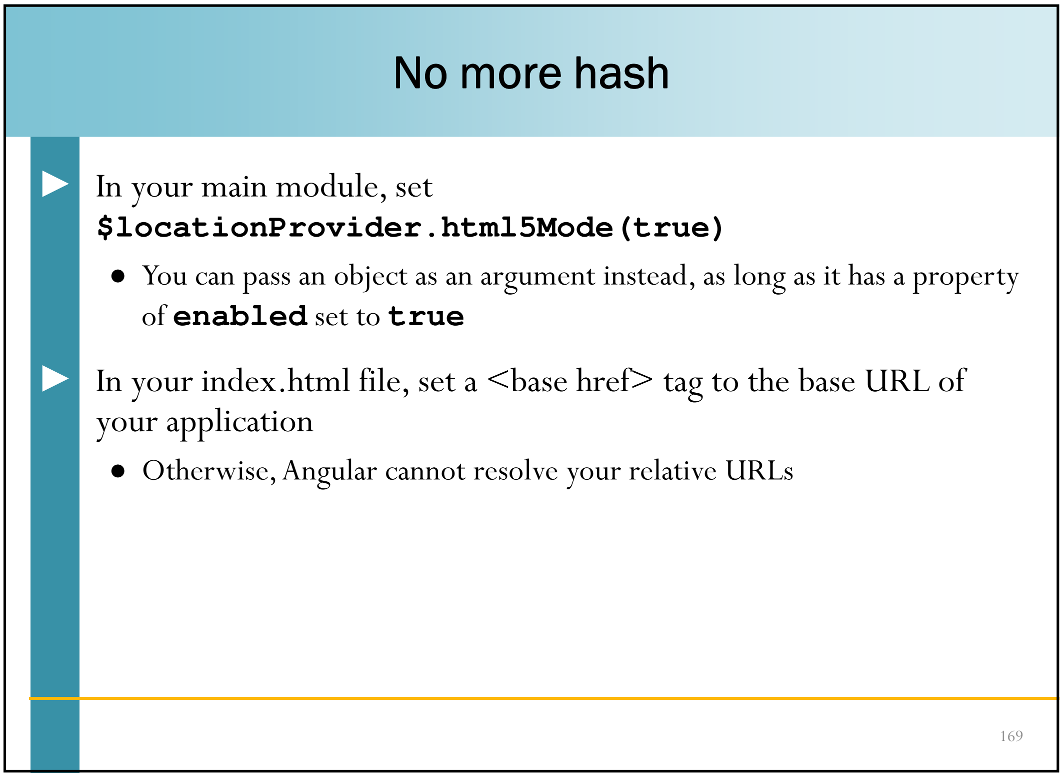
Exercise: Adding the detail view

- So let's do the same and add our own detail view
- Chapter: Routing
- You'll be working in two files:
- partials\product-detail-tpl.html: Needs display information for the particular product
- js\controllers.js: Needs to make a call to retrieve the additional product details



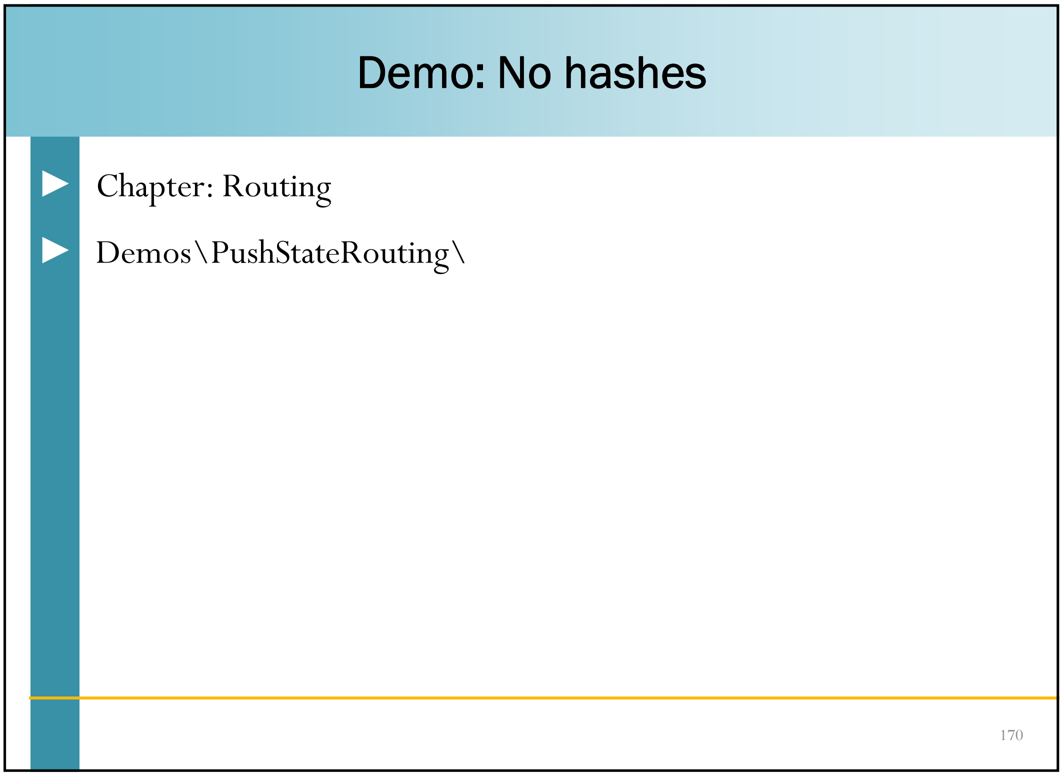
Dropping the hash

- In more modern browsers (current Chrome, FF, IE 10+), you can switch to HTML5 mode for location resolution
- This allows you to skip using hash-based URLs, and instead manipulate the history object directly to move from one URL to the next
- And it automatically falls back to using the hash mark if the user is on a non-compliant browser!
- The next slide shows the various bits and pieces you need to put in place to get this to work



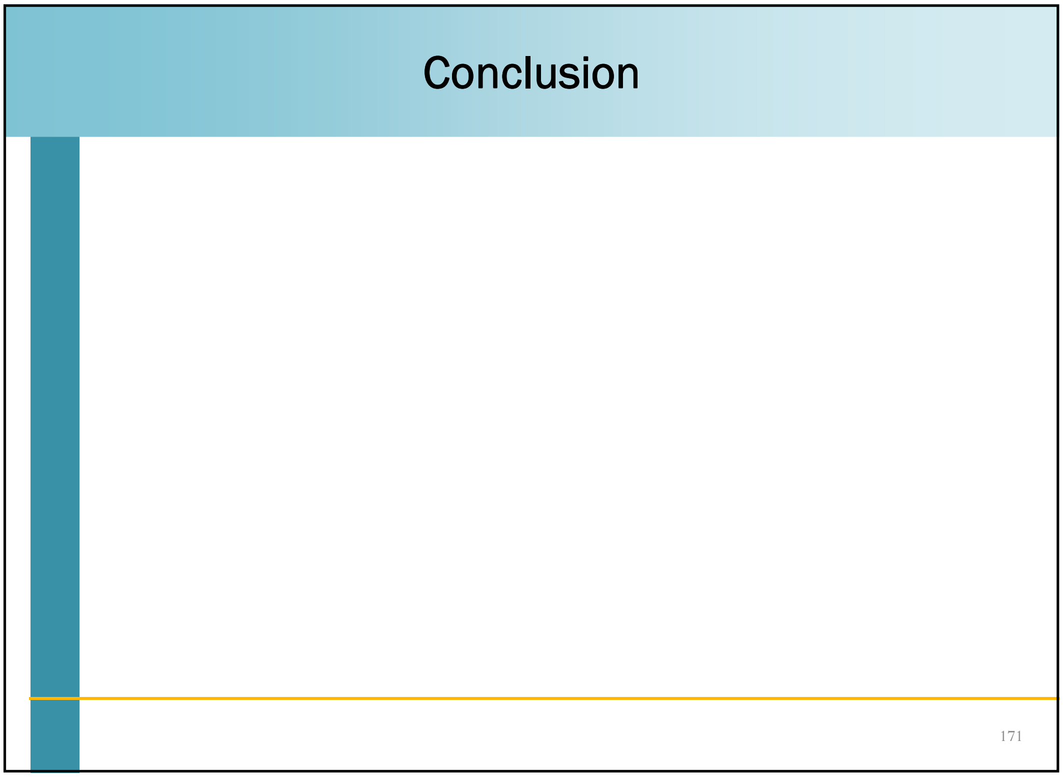
No more hash

- In your main module, set **`$locationProvider.html5Mode(true)`**
  - You can pass an object as an argument instead, as long as it has a property of **`enabled`** set to **`true`**
- In your index.html file, set a <base href> tag to the base URL of your application
  - Otherwise, Angular cannot resolve your relative URLs

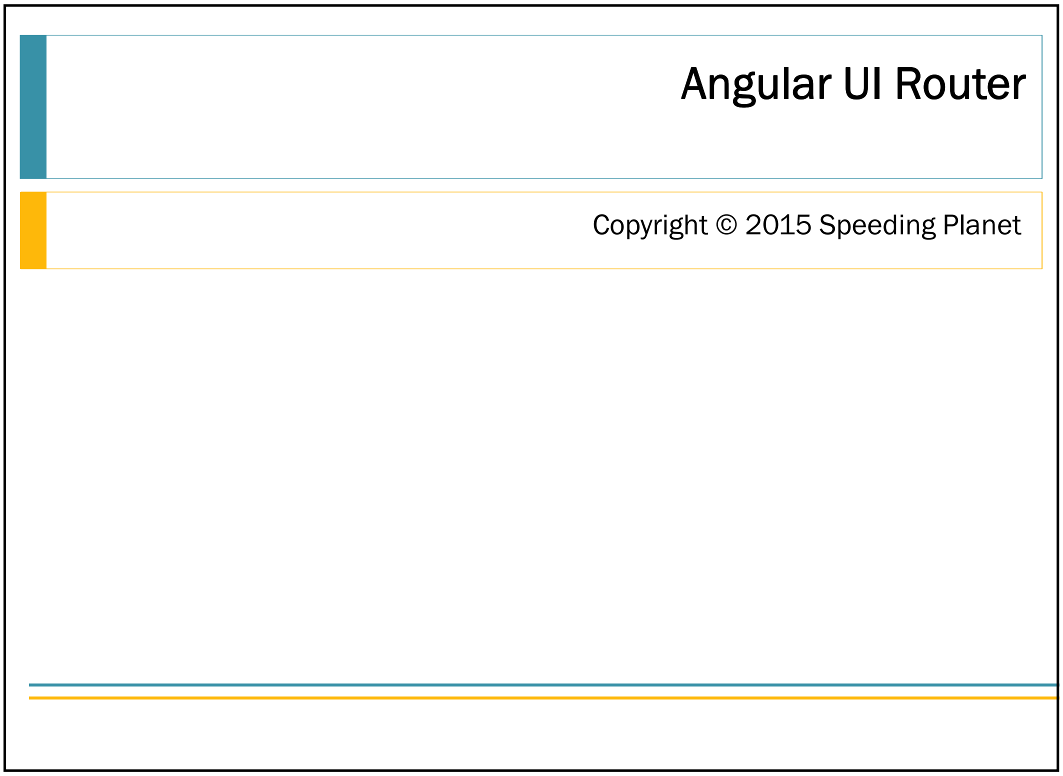


Demo: No hashes

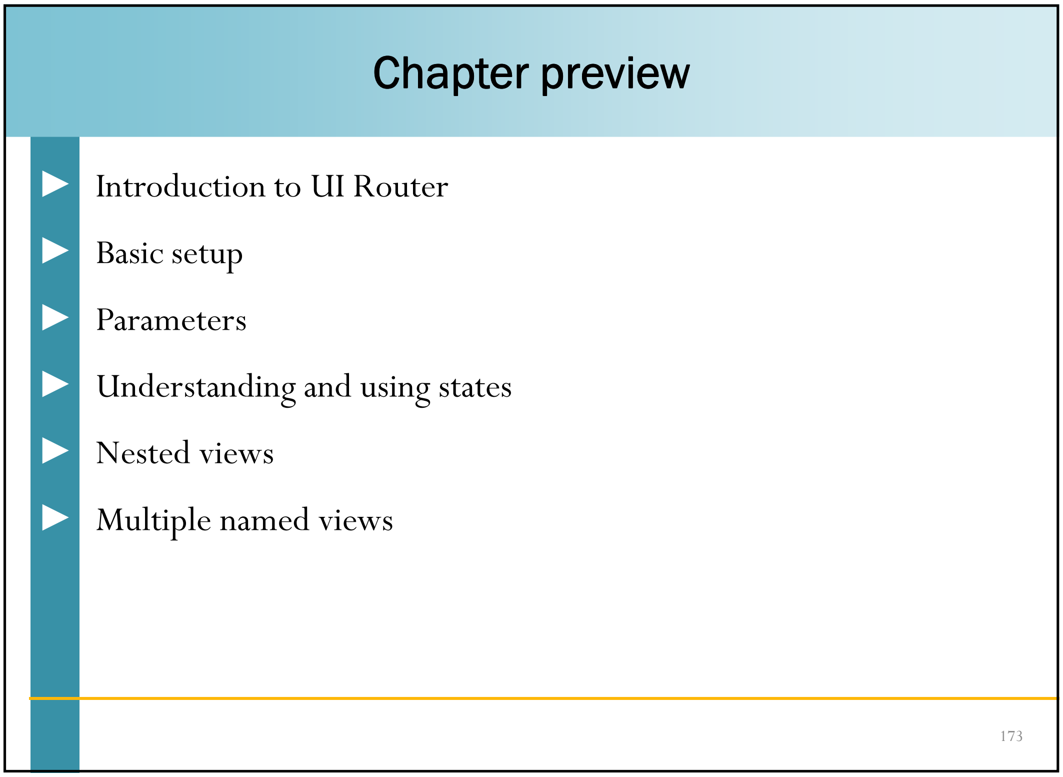
- Chapter: Routing
- Demos\PushStateRouting\



Conclusion

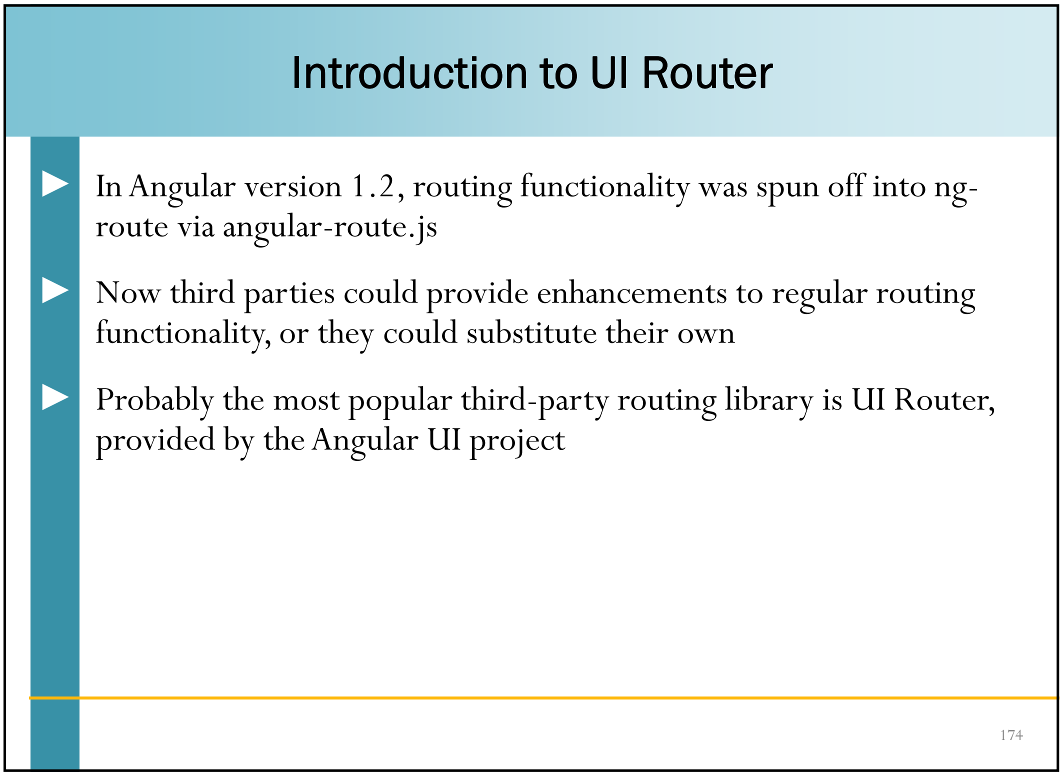


Angular UI Router



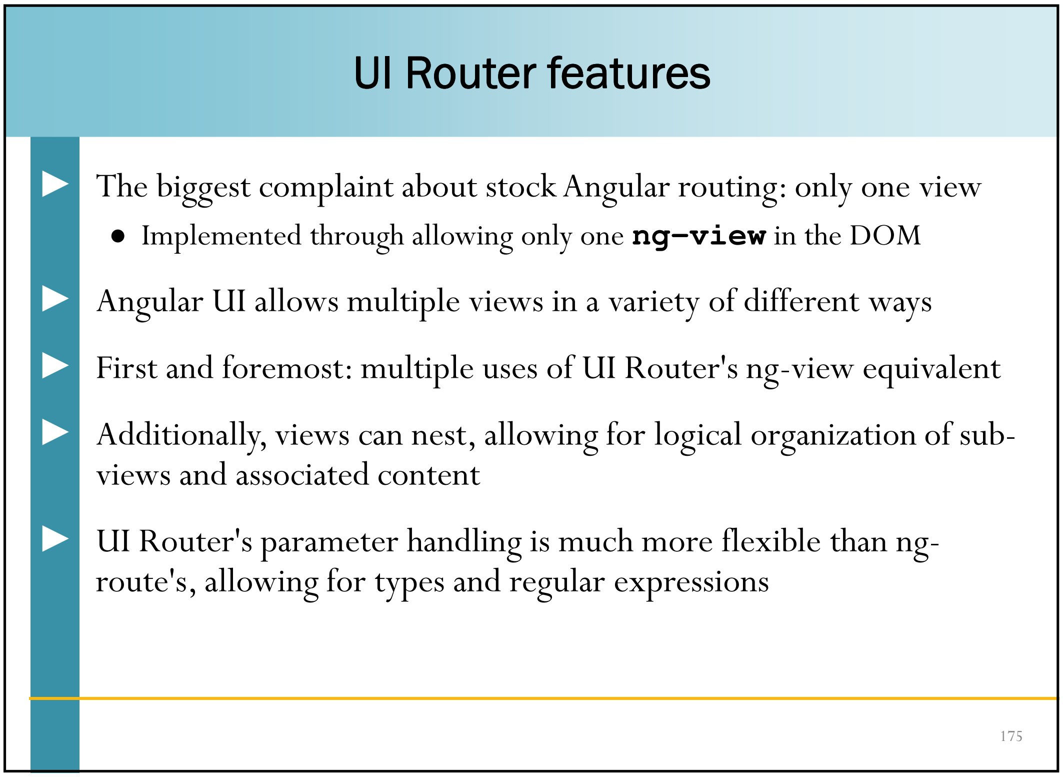
Chapter preview

- Introduction to UI Router
- Basic setup
- Parameters
- Understanding and using states
- Nested views
- Multiple named views



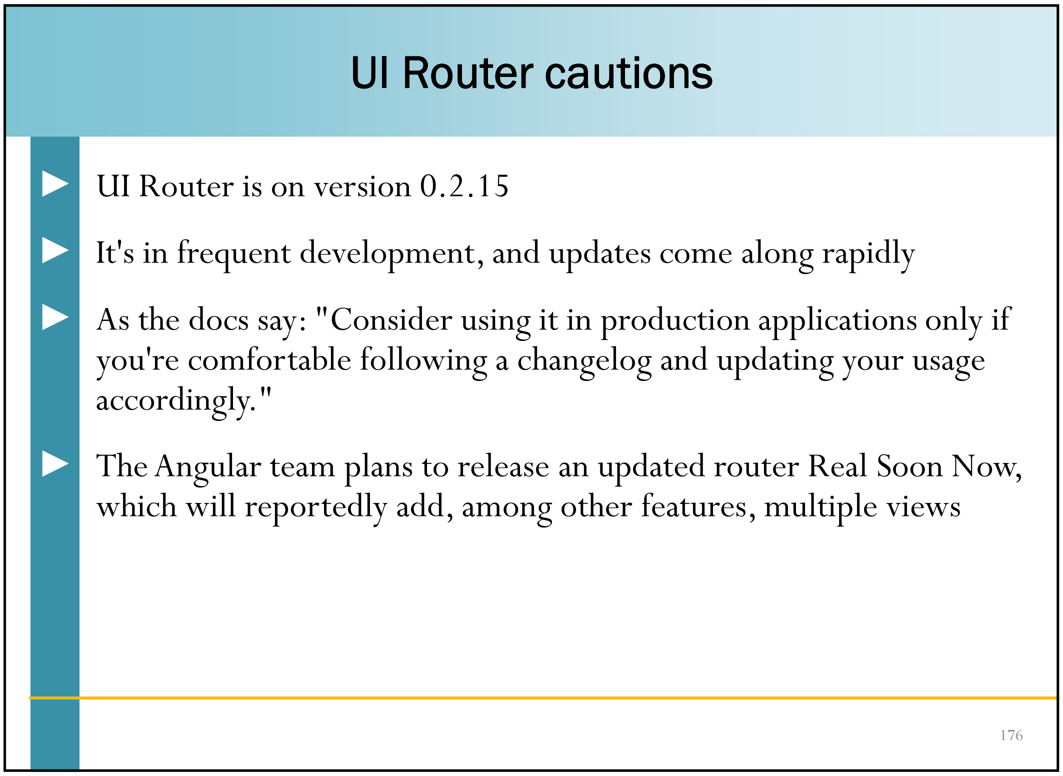
Introduction to UI Router

- In Angular version 1.2, routing functionality was spun off into ng-route via angular-route.js
- Now third parties could provide enhancements to regular routing functionality, or they could substitute their own
- Probably the most popular third-party routing library is UI Router, provided by the Angular UI project



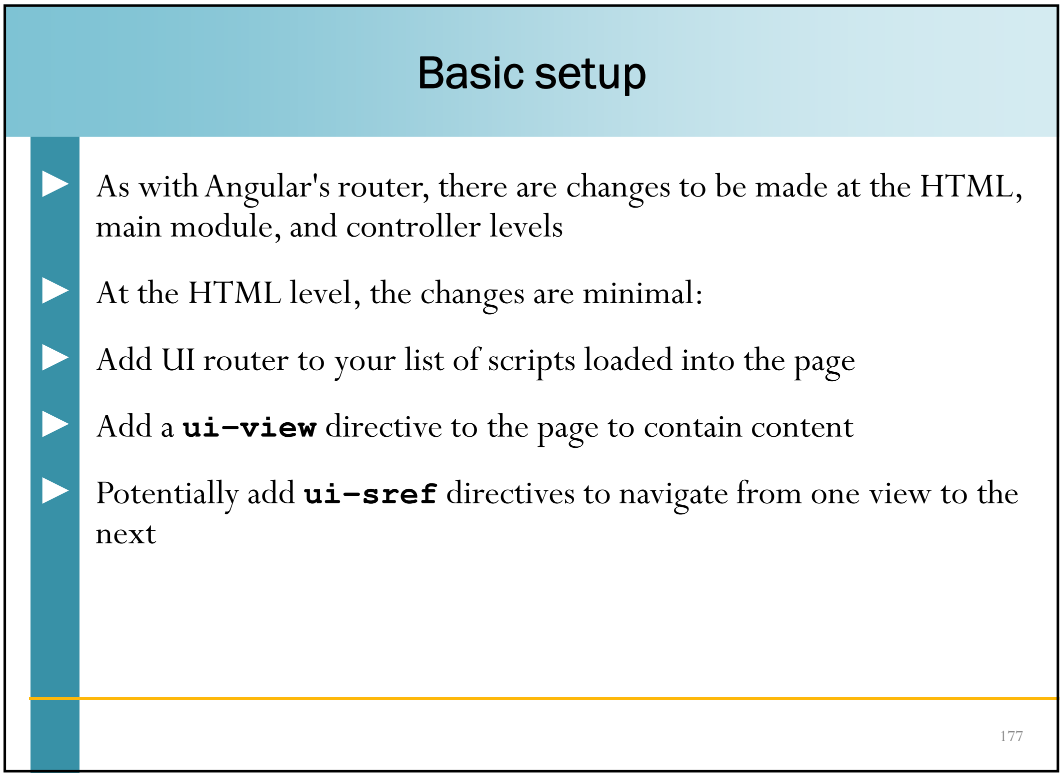
UI Router features

- The biggest complaint about stock Angular routing: only one view
  - Implemented through allowing only one **`ng-view`** in the DOM
- Angular UI allows multiple views in a variety of different ways
- First and foremost: multiple uses of UI Router's ng-view equivalent
- Additionally, views can nest, allowing for logical organization of sub-views and associated content
- UI Router's parameter handling is much more flexible than ng-route's, allowing for types and regular expressions



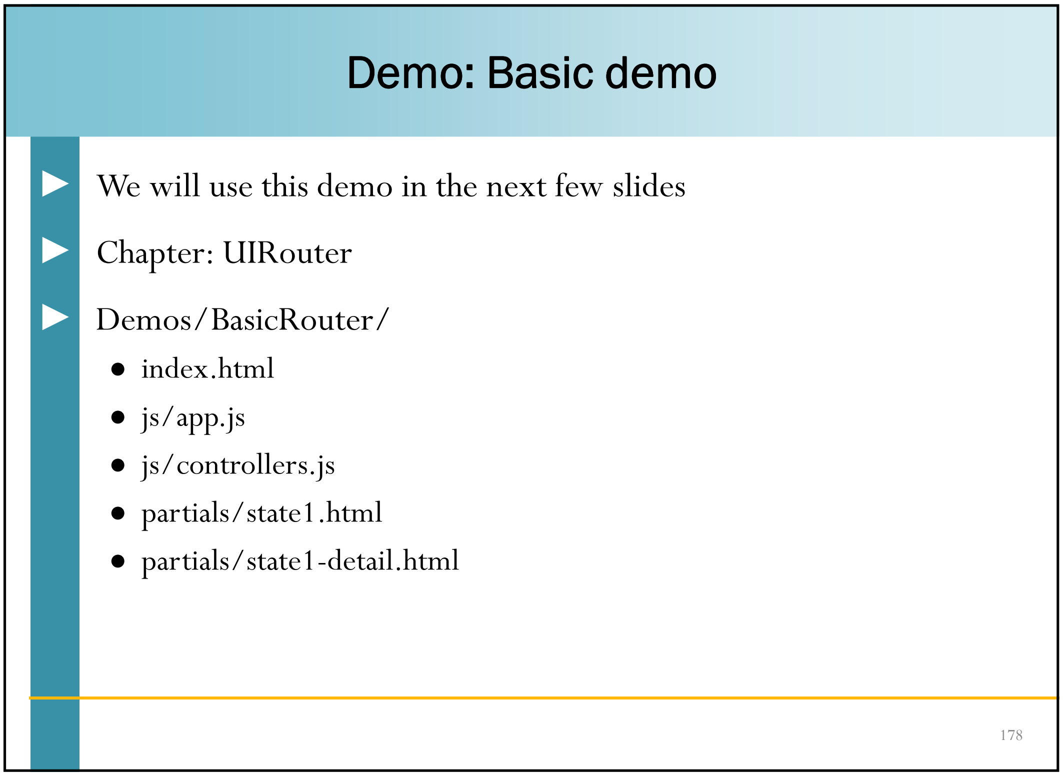
UI Router cautions

- UI Router is on version 0.2.15
- It's in frequent development, and updates come along rapidly
- As the docs say: "Consider using it in production applications only if you're comfortable following a changelog and updating your usage accordingly."
- The Angular team plans to release an updated router Real Soon Now, which will reportedly add, among other features, multiple views



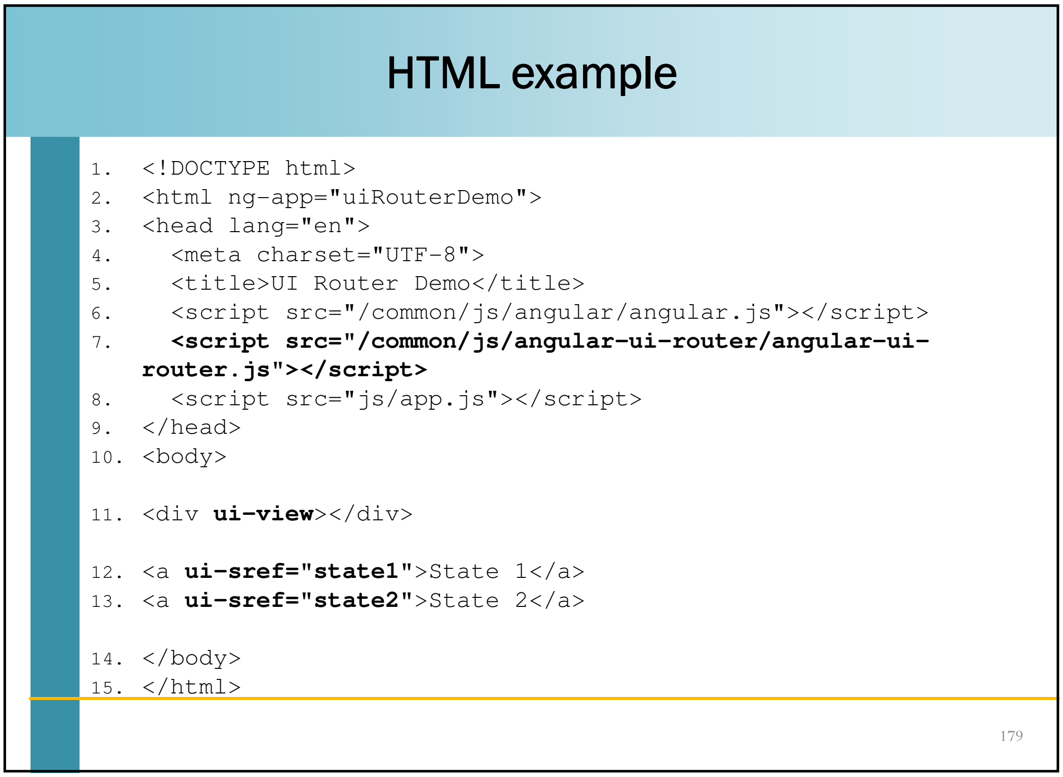
Basic setup

- As with Angular's router, there are changes to be made at the HTML, main module, and controller levels
- At the HTML level, the changes are minimal:
- Add UI router to your list of scripts loaded into the page
- Add a **`ui-view`** directive to the page to contain content
- Potentially add **`ui-sref`** directives to navigate from one view to the next



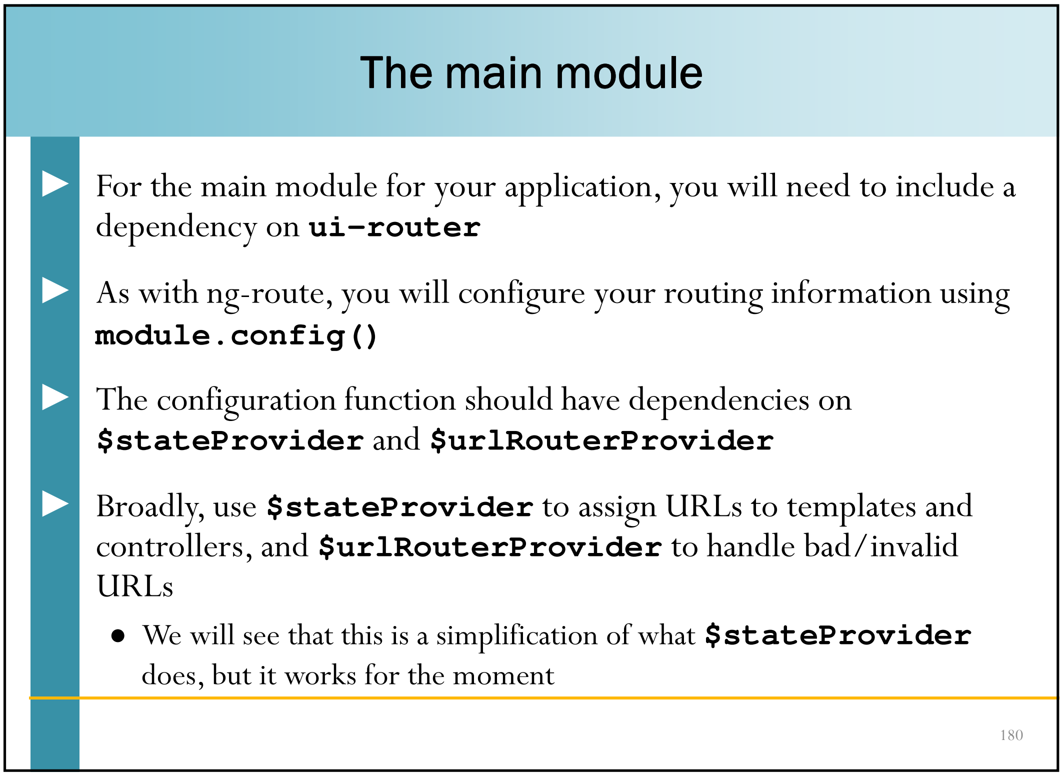
Demo: Basic demo

- We will use this demo in the next few slides
- Chapter: UIRouter
- Demos/BasicRouter/
  - index.html
  - js/app.js
  - js/controllers.js
  - partials/state1.html
  - partials/state1-detail.html



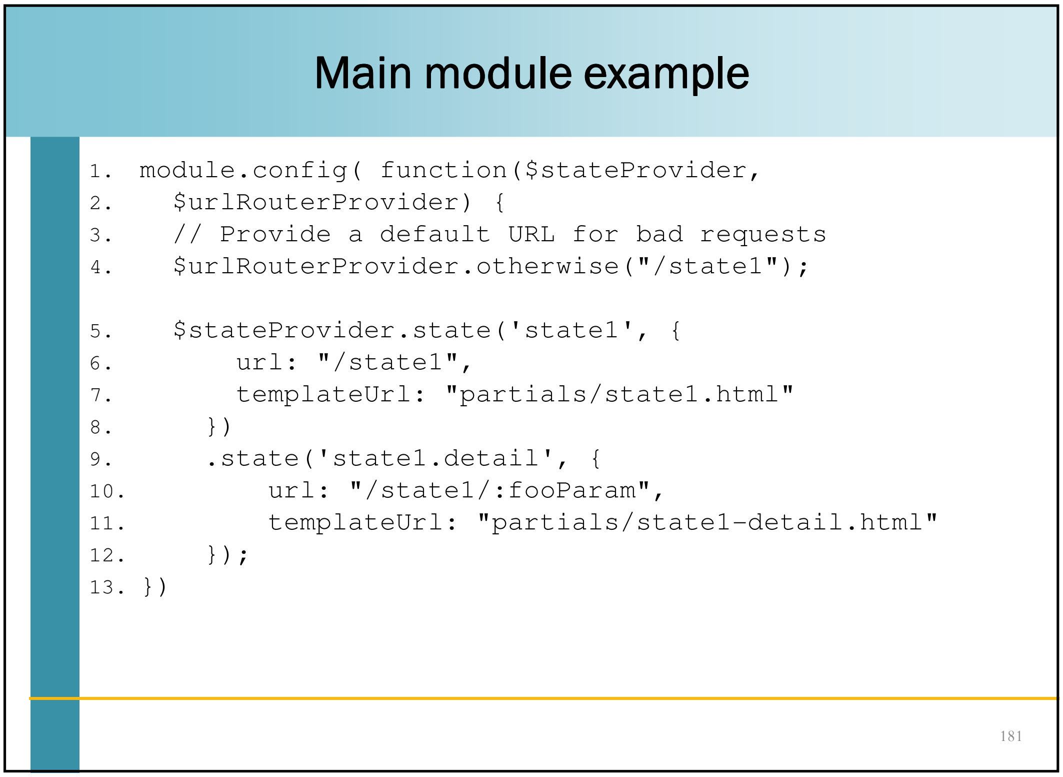
HTML example

```
<!DOCTYPE html>
<html ng-app="uiRouterDemo">
<head lang="en">
  <meta charset="UTF-8">
  <title>UI Router Demo</title>
  <script src="/common/js/angular/angular.js"></script>
  <script src="/common/js/angular-ui-router/angular-ui-router.js"></script>
  <script src="js/app.js"></script>
</head>
<body>

<div ui-view></div>

<a ui-sref="state1">State 1</a>
<a ui-sref="state2">State 2</a>

</body>
</html>
```



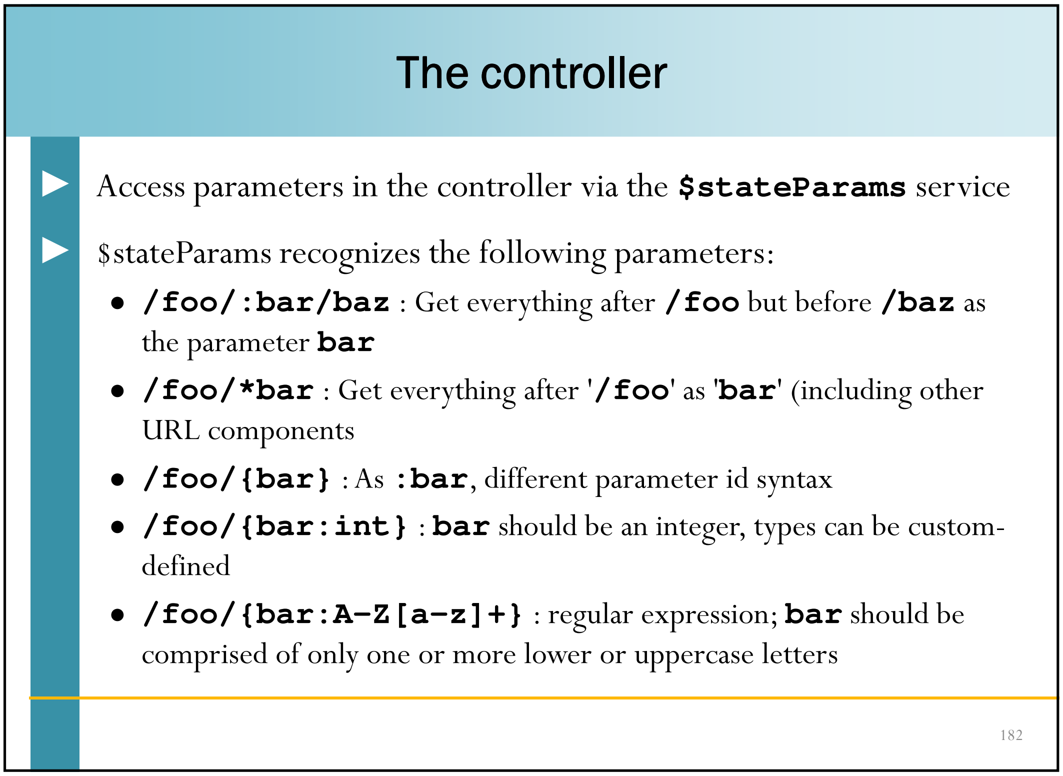
The main module

- For the main module for your application, you will need to include a dependency on **`ui-router`**
- As with ng-route, you will configure your routing information using **`module.config()`**
- The configuration function should have dependencies on **`$stateProvider`** and **`$urlRouterProvider`**
- Broadly, use **`$stateProvider`** to assign URLs to templates and controllers, and **`$urlRouterProvider`** to handle bad/invalid URLs
  - We will see that this is a simplification of what **`$stateProvider`** does, but it works for the moment



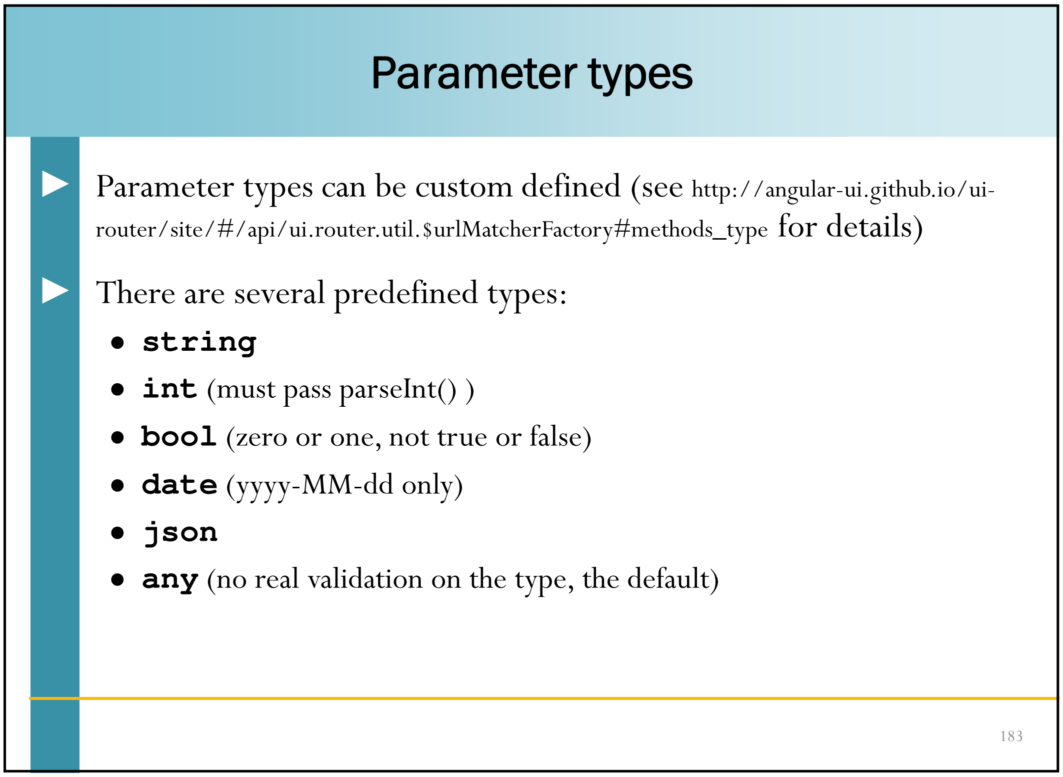
Main module example

```
module.config( function($stateProvider,
  $urlRouterProvider) {
  // Provide a default URL for bad requests
  $urlRouterProvider.otherwise("/state1");

  $stateProvider.state('state1', {
      url: "/state1",
      templateUrl: "partials/state1.html"
    })
    .state('state1.detail', {
        url: "/state1/:fooParam",
        templateUrl: "partials/state1-detail.html"
    });
})
```



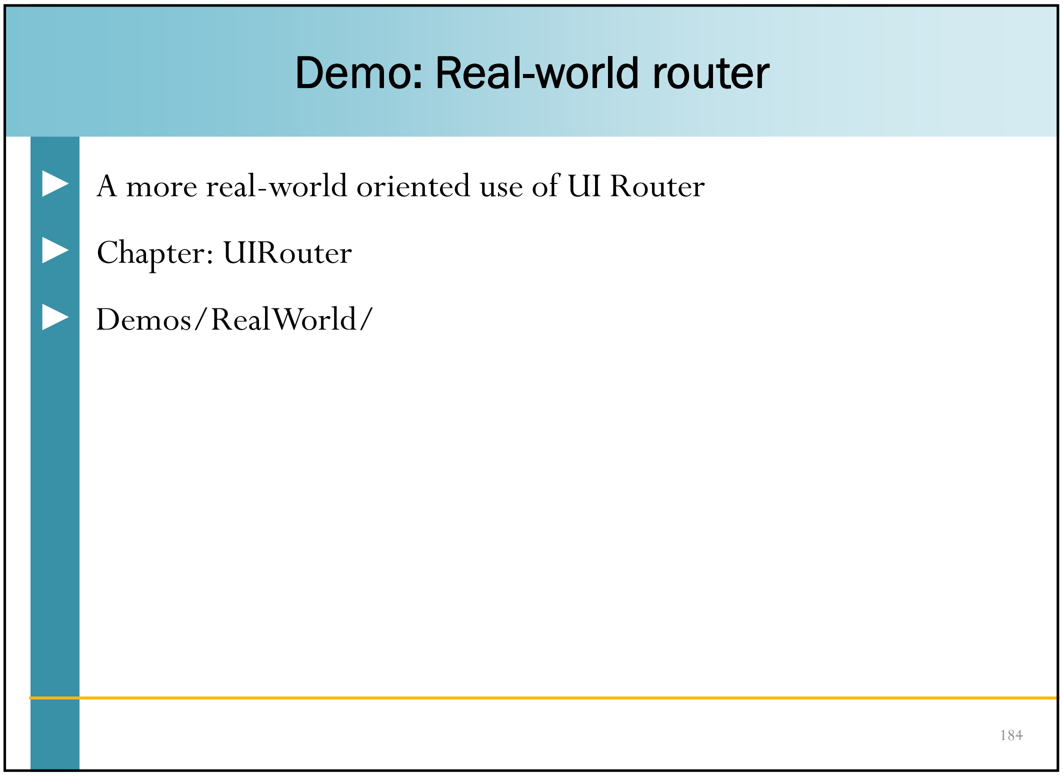
The controller

- Access parameters in the controller via the **`$stateParams`** service
- $stateParams recognizes the following parameters:
  - **`/foo/:bar/baz`** : Get everything after **`/foo`** but before **`/baz`** as the parameter **`bar`**
  - **`/foo/*bar`** : Get everything after '**`/foo`**' as '**`bar`**' (including other URL components
  - **`/foo/{bar}`** : As **`:bar`**, different parameter id syntax
  - **`/foo/{bar:int}`** : **`bar`** should be an integer, types can be custom-defined
  - **`/foo/{bar:A-Z[a-z]+}`** : regular expression; **`bar`** should be comprised of only one or more lower or uppercase letters



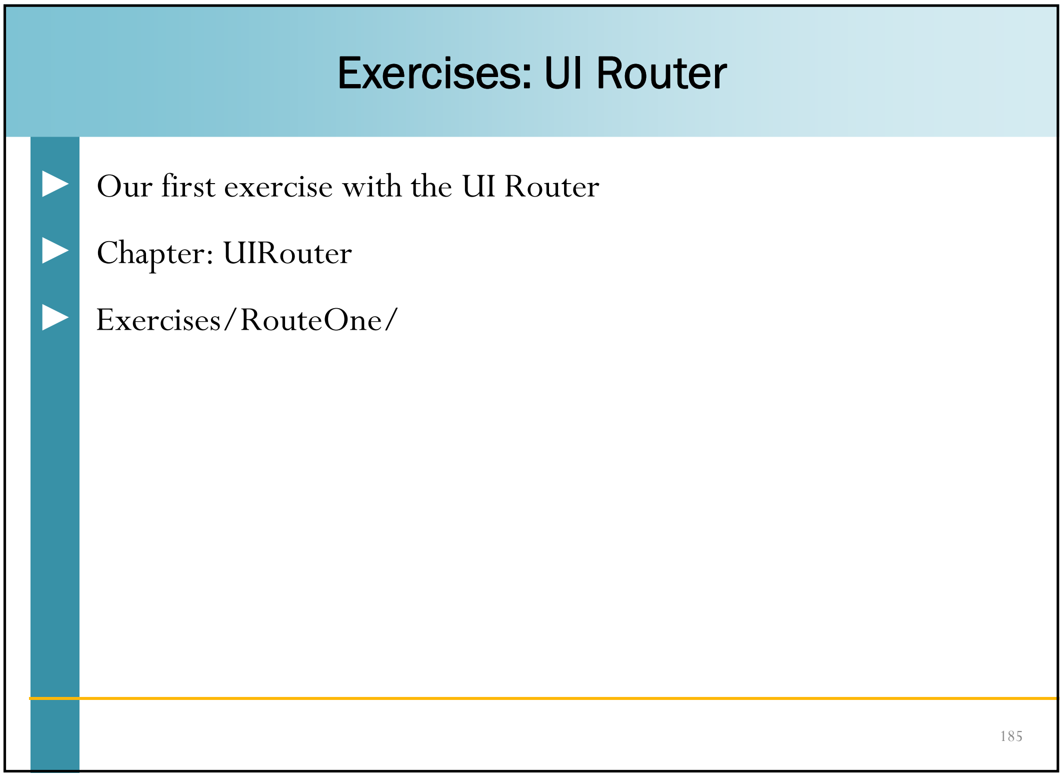
Parameter types

- Parameter types can be custom defined (see http://angular-ui.github.io/ui-router/site/#/api/ui.router.util.$urlMatcherFactory#methods_type for details)
- There are several predefined types:
  - **`string`**
  - **`int`** (must pass parseInt())
  - **`bool`** (zero or one, not true or false)
  - **`date`** (yyyy-MM-dd only)
  - **`json`**
  - **`any`** (no real validation on the type, the default)



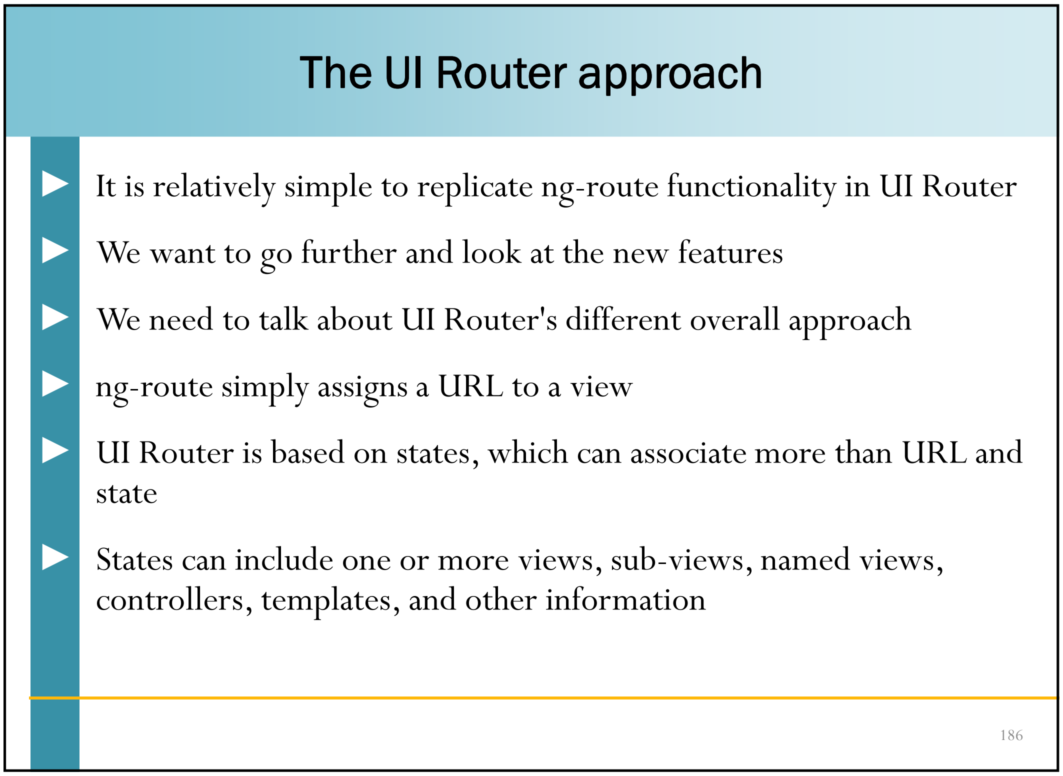
Demo: Real-world router

- A more real-world oriented use of UI Router
- Chapter: UIRouter
- Demos/RealWorld/



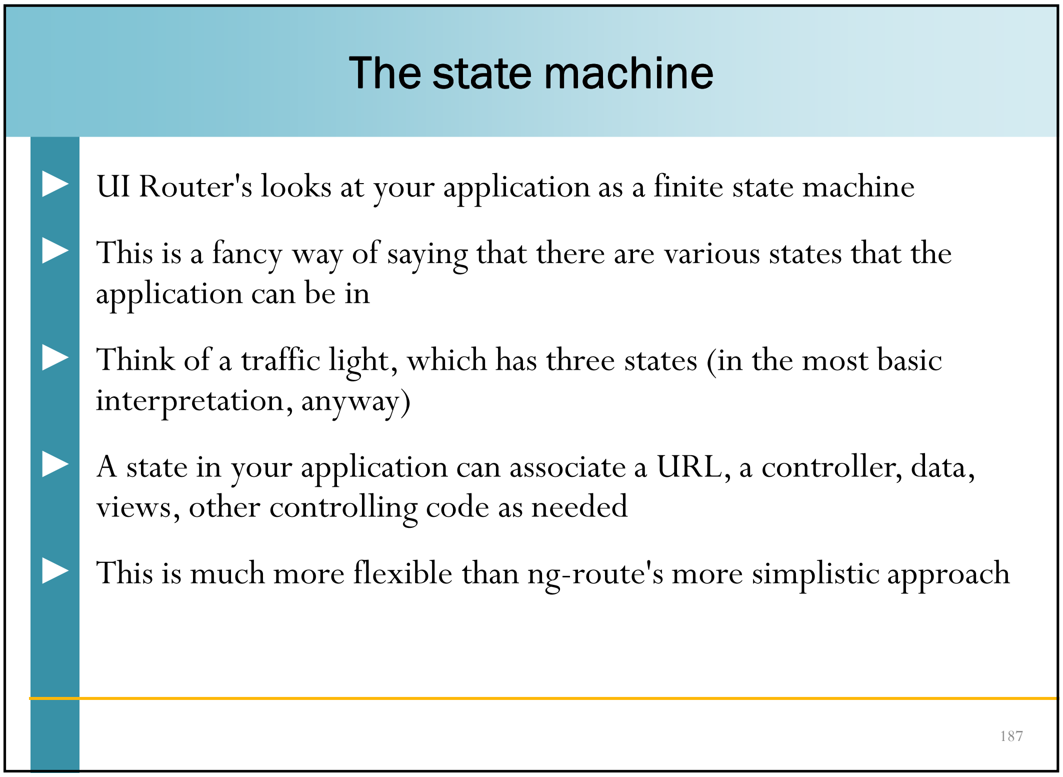
Exercises: UI Router

- Our first exercise with the UI Router
- Chapter: UIRouter
- Exercises/RouteOne/



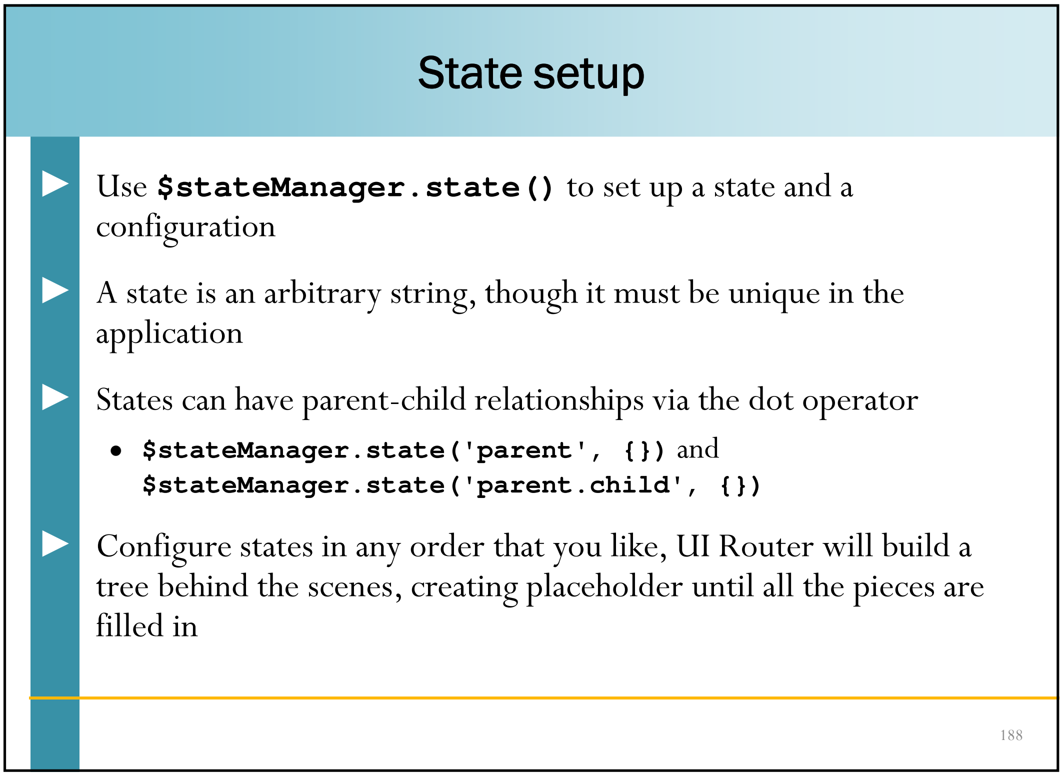
The UI Router approach

- It is relatively simple to replicate ng-route functionality in UI Router
- We want to go further and look at the new features
- We need to talk about UI Router's different overall approach
- ng-route simply assigns a URL to a view
- UI Router is based on states, which can associate more than URL and state
- States can include one or more views, sub-views, named views, controllers, templates, and other information



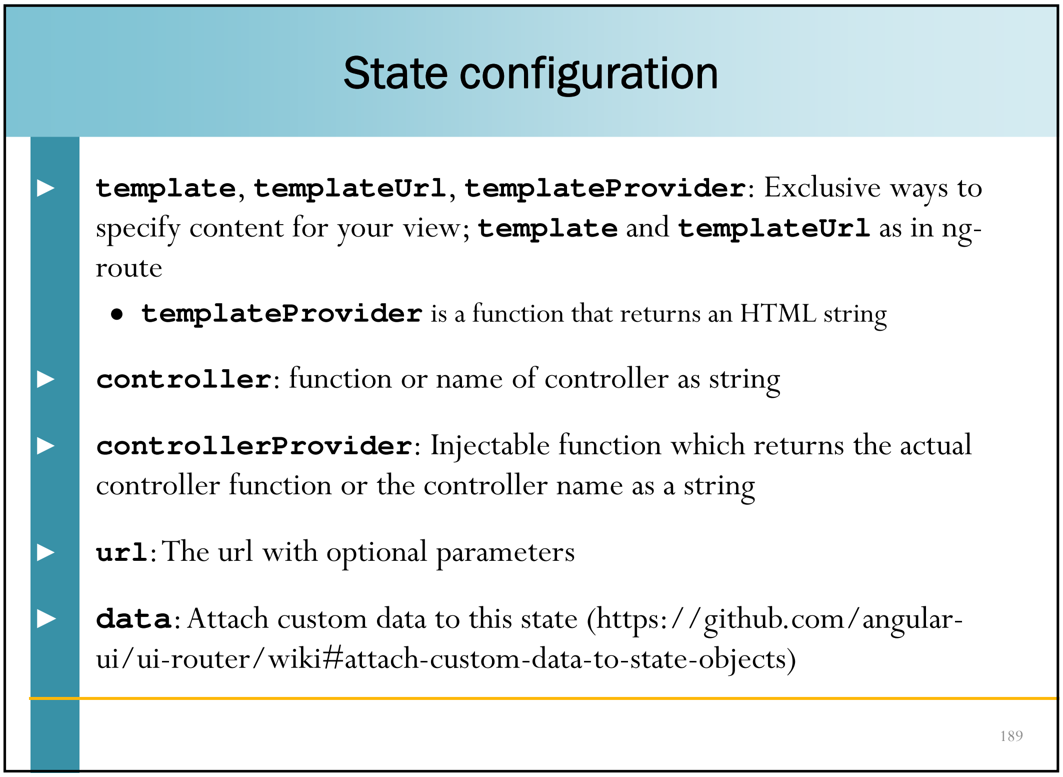
The state machine

- UI Router's looks at your application as a finite state machine
- This is a fancy way of saying that there are various states that the application can be in
- Think of a traffic light, which has three states (in the most basic interpretation, anyway)
- A state in your application can associate a URL, a controller, data, views, other controlling code as needed
- This is much more flexible than ng-route's more simplistic approach



State setup

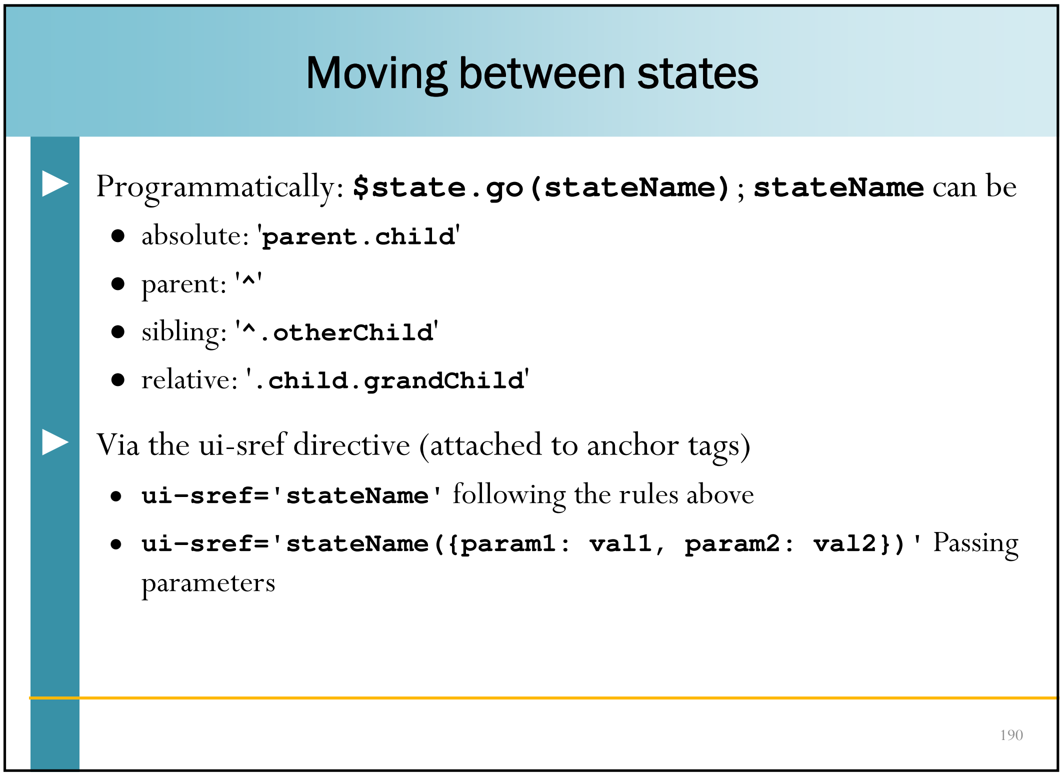
- Use **`$stateManager.state()`** to set up a state and a configuration
- A state is an arbitrary string, though it must be unique in the application
- States can have parent-child relationships via the dot operator
  - **`$stateManager.state('parent', {})`** and **`$stateManager.state('parent.child', {})`**
- Configure states in any order that you like, UI Router will build a tree behind the scenes, creating placeholder until all the pieces are filled in



State configuration

- **`template`**, **`templateUrl`**, **`templateProvider`**: Exclusive ways to specify content for your view; **`template`** and **`templateUrl`** as in ng-route
  - **`templateProvider`** is a function that returns an HTML string
- **`controller`**: function or name of controller as string
- **`controllerProvider`**: Injectable function which returns the actual controller function or the controller name as a string
- **`url`**: The url with optional parameters
- **`data`**: Attach custom data to this state (https://github.com/angular-ui/ui-router/wiki#attach-custom-data-to-state-objects)

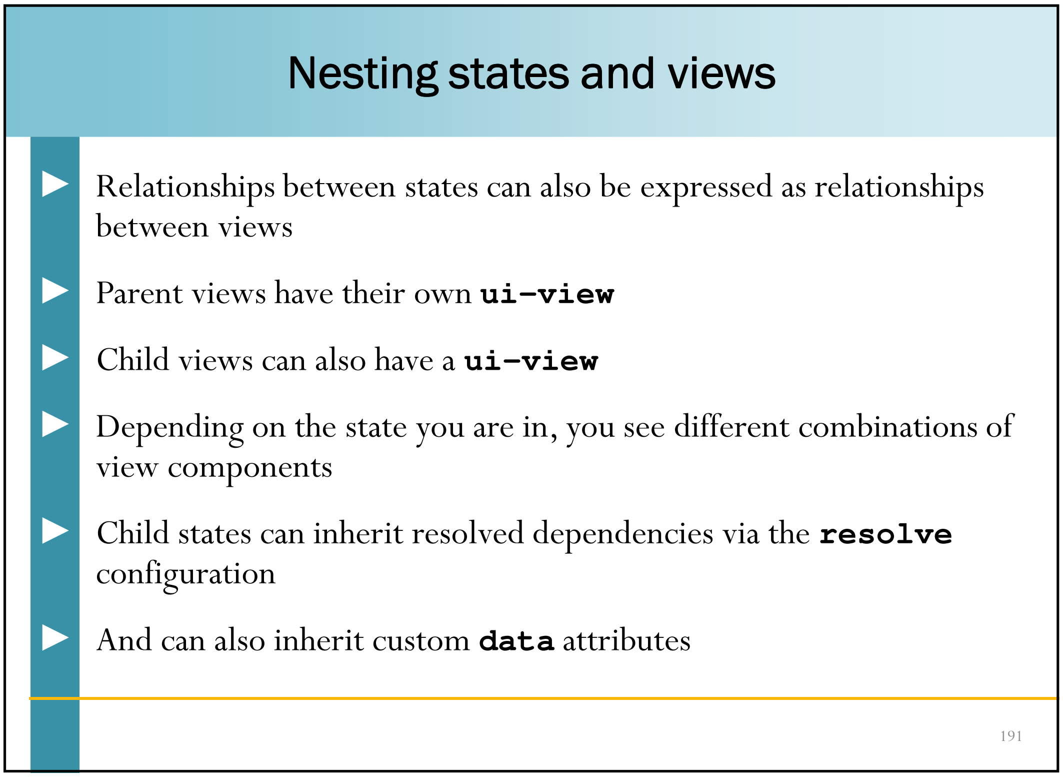
189



Moving between states

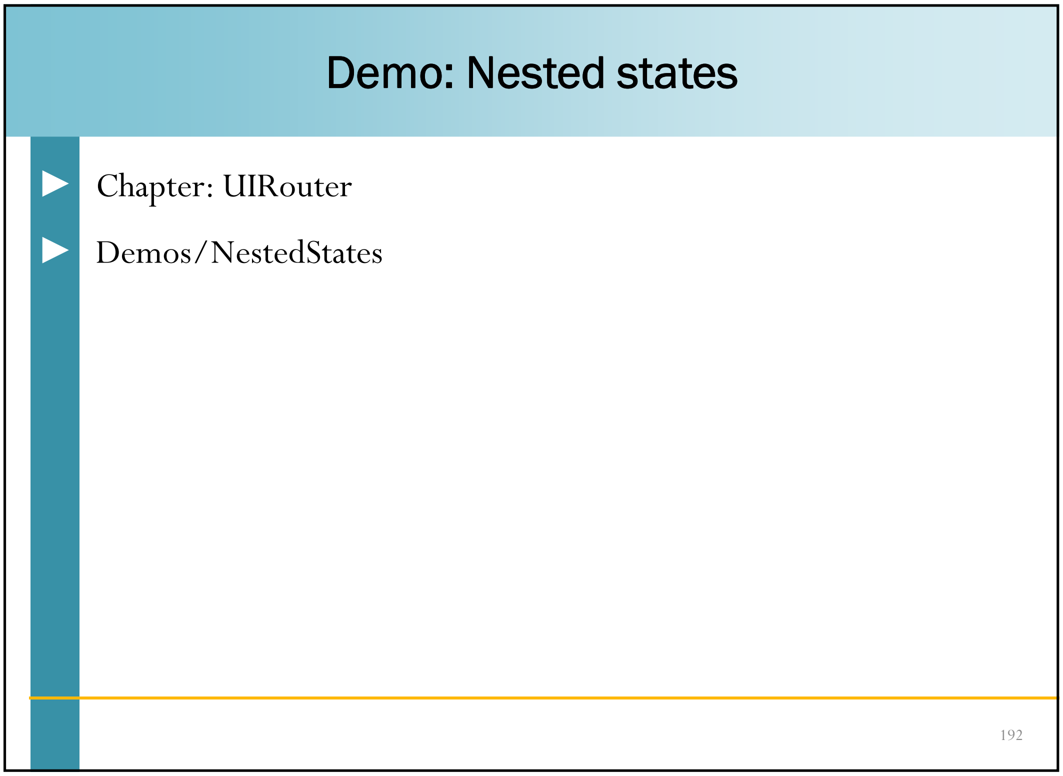
- Programmatically: **`$state.go(stateName)`**; **`stateName`** can be
  - absolute: '**`parent.child`**'
  - parent: '**`^`**'
  - sibling: '**`^.otherChild`**'
  - relative: '**`.child.grandChild`**'
- Via the ui-sref directive (attached to anchor tags)
  - **`ui-sref='stateName'`** following the rules above
  - **`ui-sref='stateName({param1: val1, param2: val2})'`** Passing parameters

190



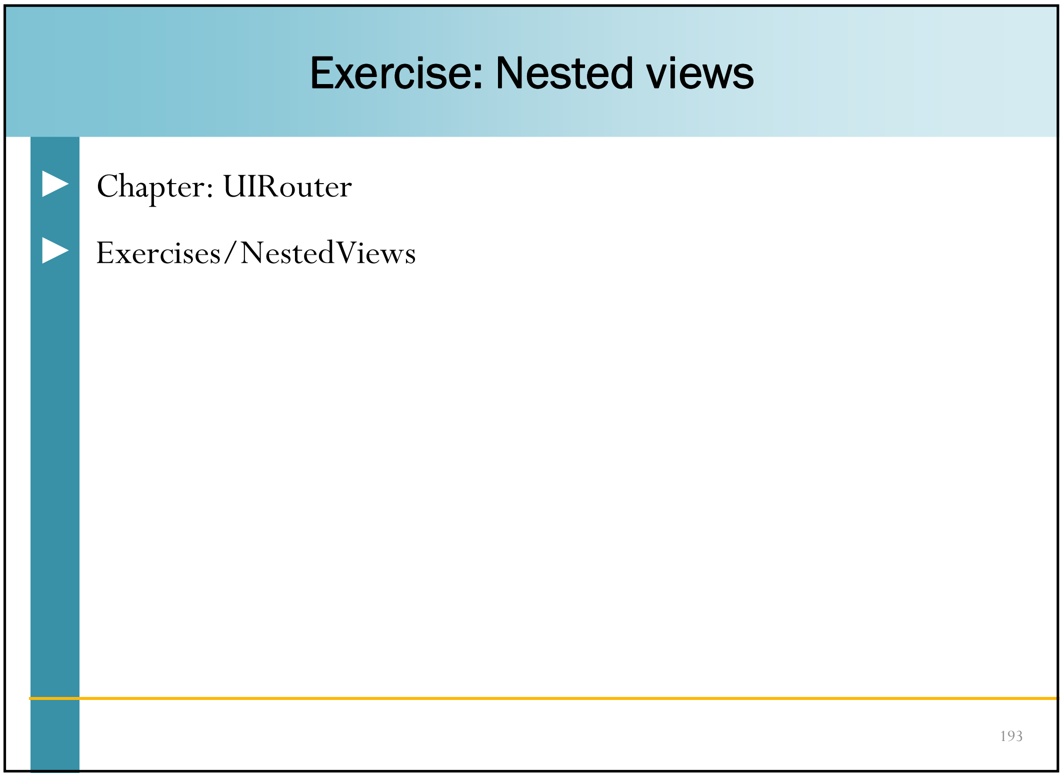
Nesting states and views

- Relationships between states can also be expressed as relationships between views
- Parent views have their own **`ui-view`**
- Child views can also have a **`ui-view`**
- Depending on the state you are in, you see different combinations of view components
- Child states can inherit resolved dependencies via the **`resolve`** configuration
- And can also inherit custom **`data`** attributes



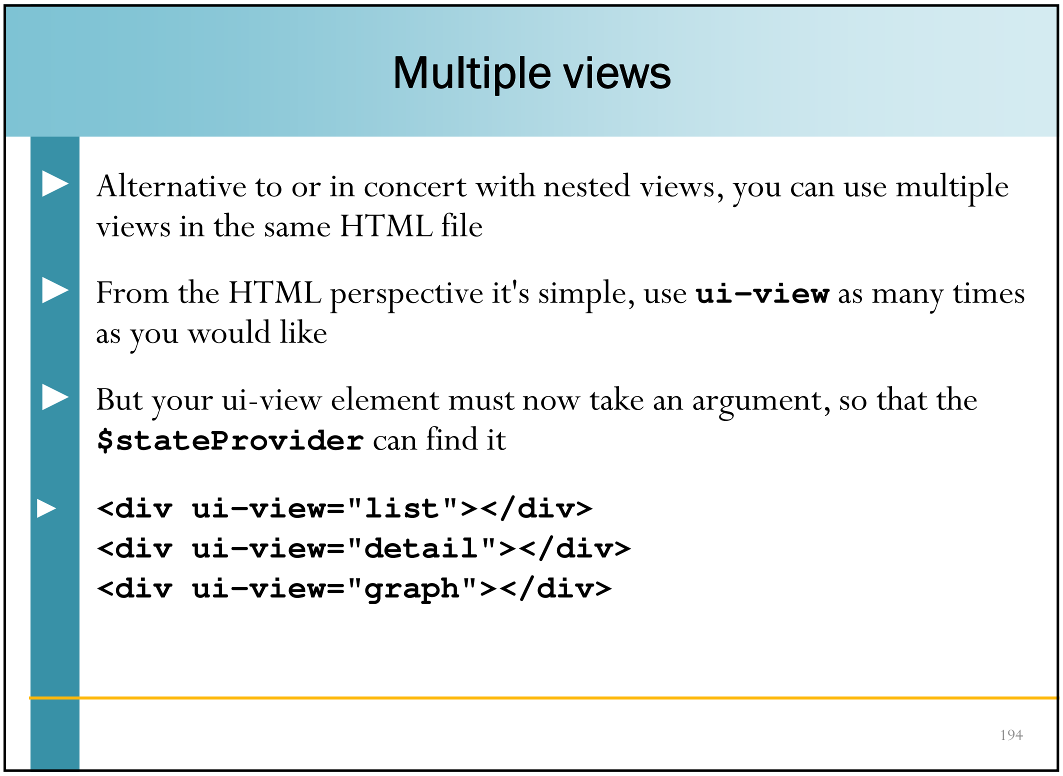
Demo: Nested states

- Chapter: UIRouter
- Demos/NestedStates



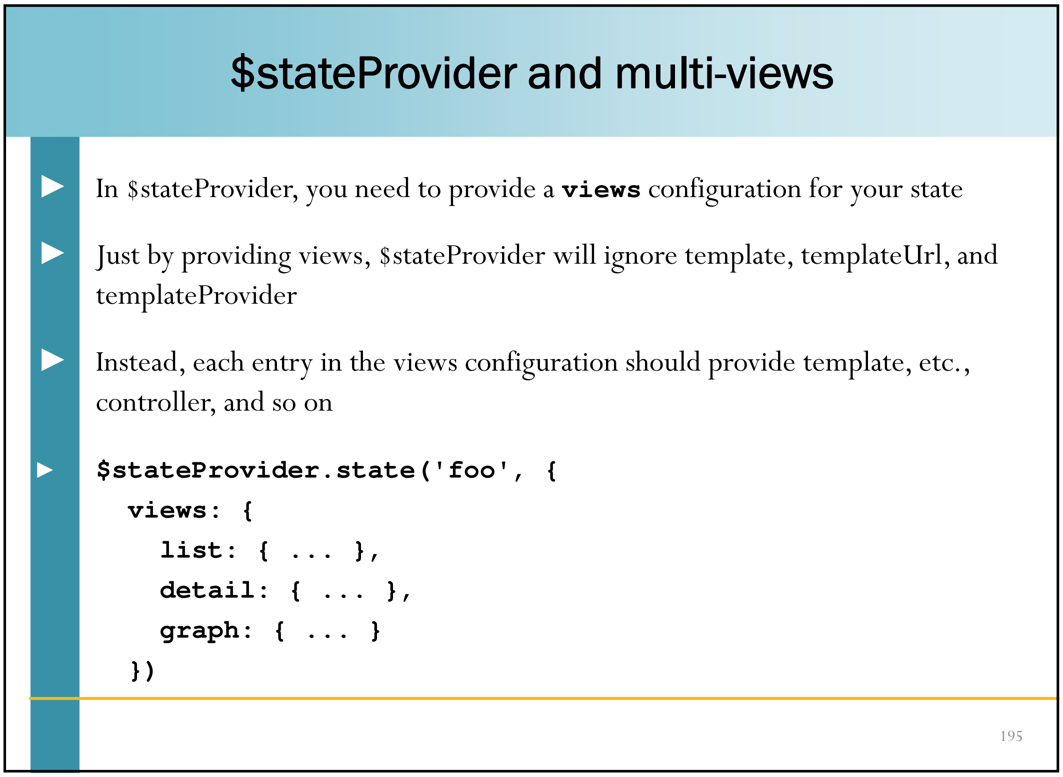
Exercise: Nested views

- Chapter: UIRouter
- Exercises/NestedViews



Multiple views

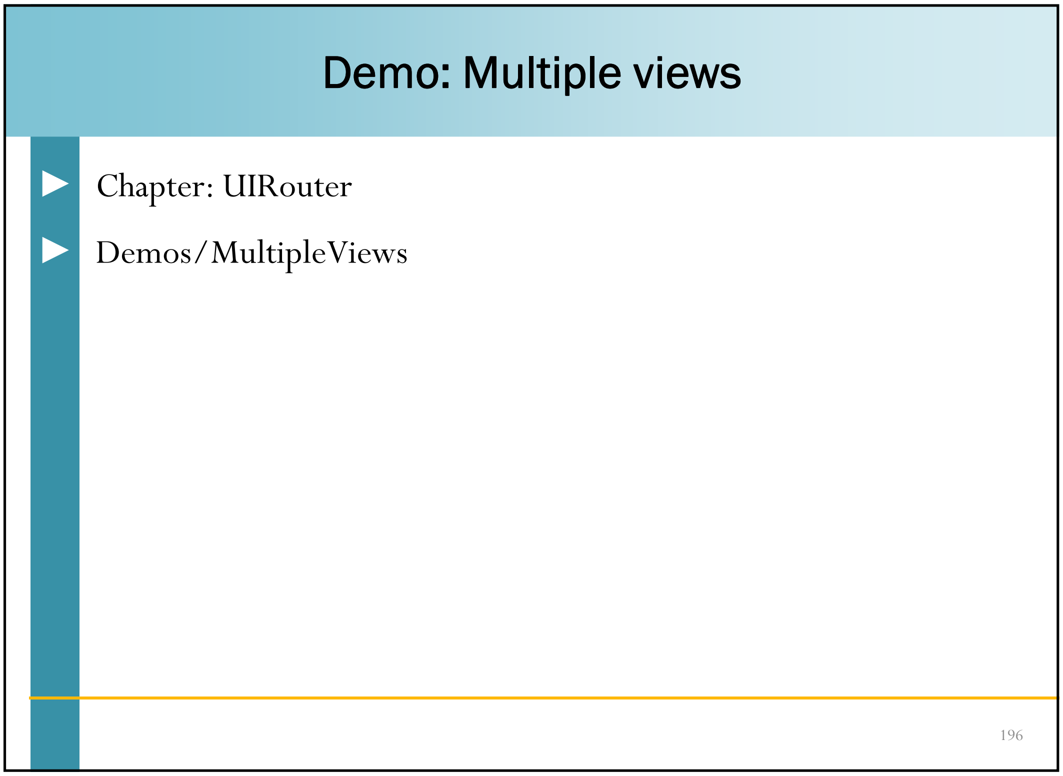
- Alternative to or in concert with nested views, you can use multiple views in the same HTML file
- From the HTML perspective it's simple, use **`ui-view`** as many times as you would like
- But your ui-view element must now take an argument, so that the **`$stateProvider`** can find it
-
```
<div ui-view="list"></div>
<div ui-view="detail"></div>
<div ui-view="graph"></div>
```



$stateProvider and multi-views

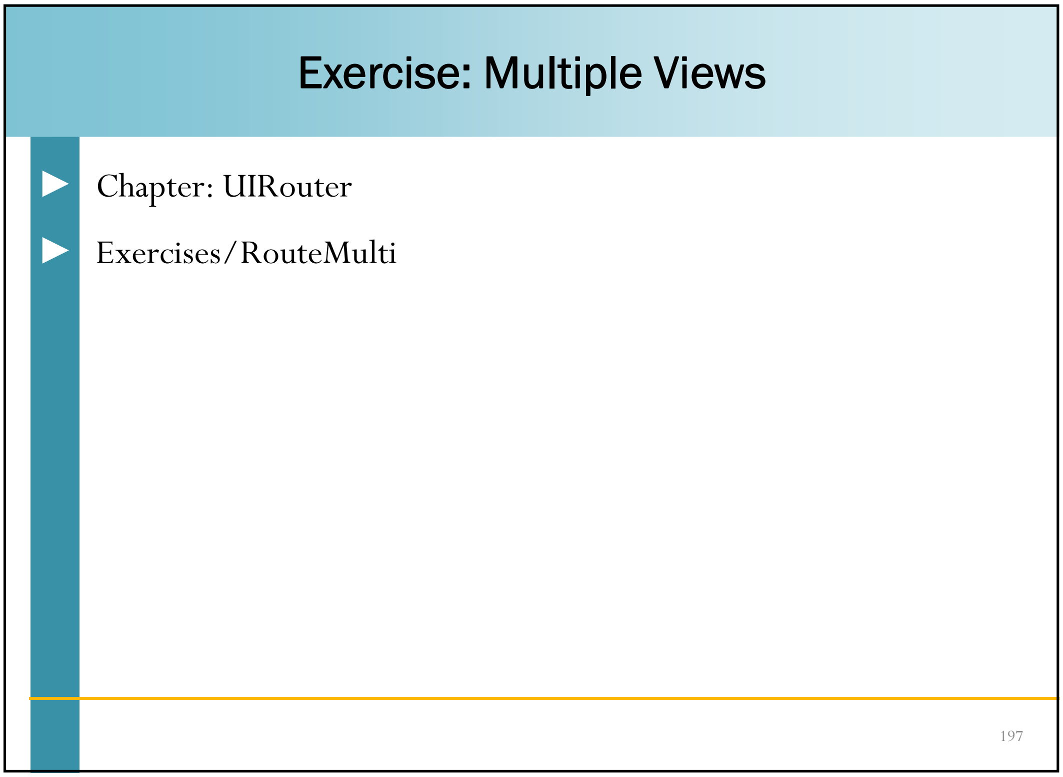
- In $stateProvider, you need to provide a **`views`** configuration for your state
- Just by providing views, $stateProvider will ignore template, templateUrl, and templateProvider
- Instead, each entry in the views configuration should provide template, etc., controller, and so on
-

```
$stateProvider.state('foo', {
  views: {
    list: { ... },
    detail: { ... },
    graph: { ... }
  })
```



Demo: Multiple views

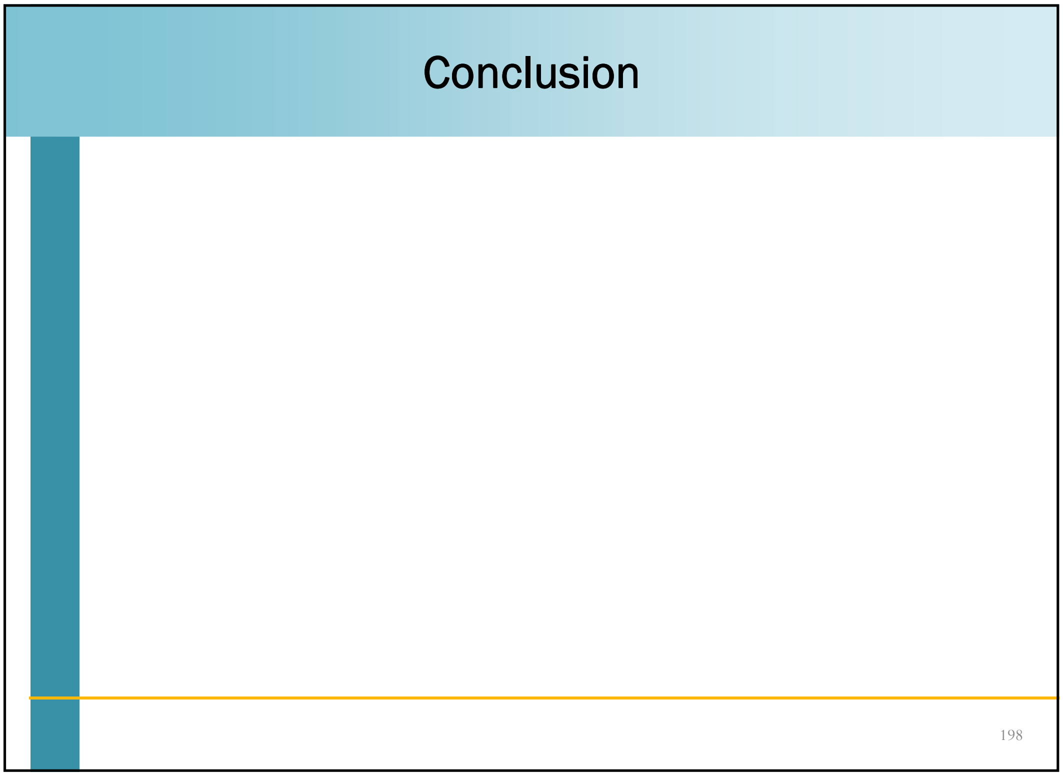
- Chapter: UIRouter
- Demos/MultipleViews



Exercise: Multiple Views

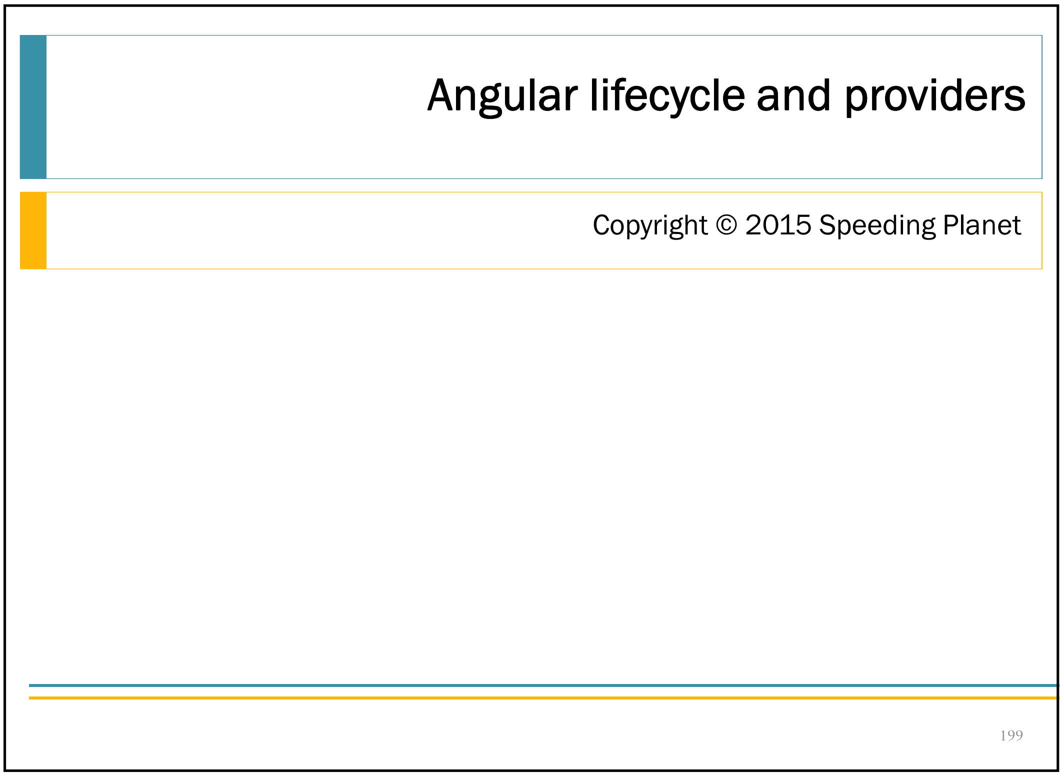
- Chapter: UIRouter
- Exercises/RouteMulti

197



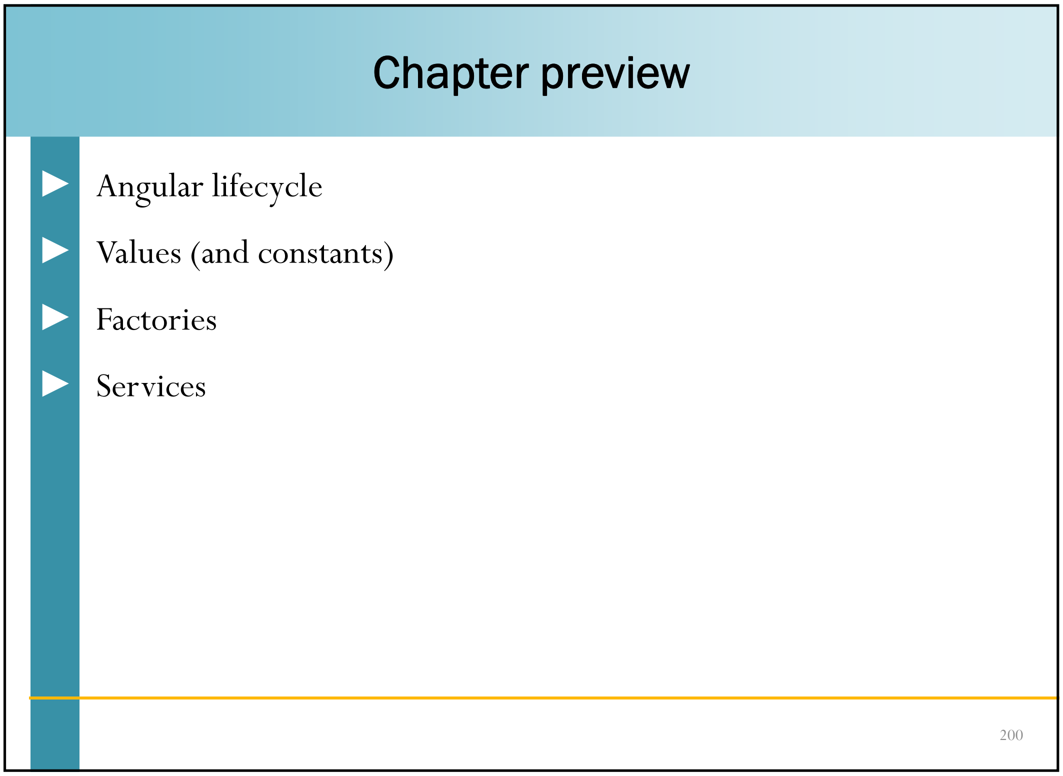
Conclusion

198



Angular lifecycle and providers

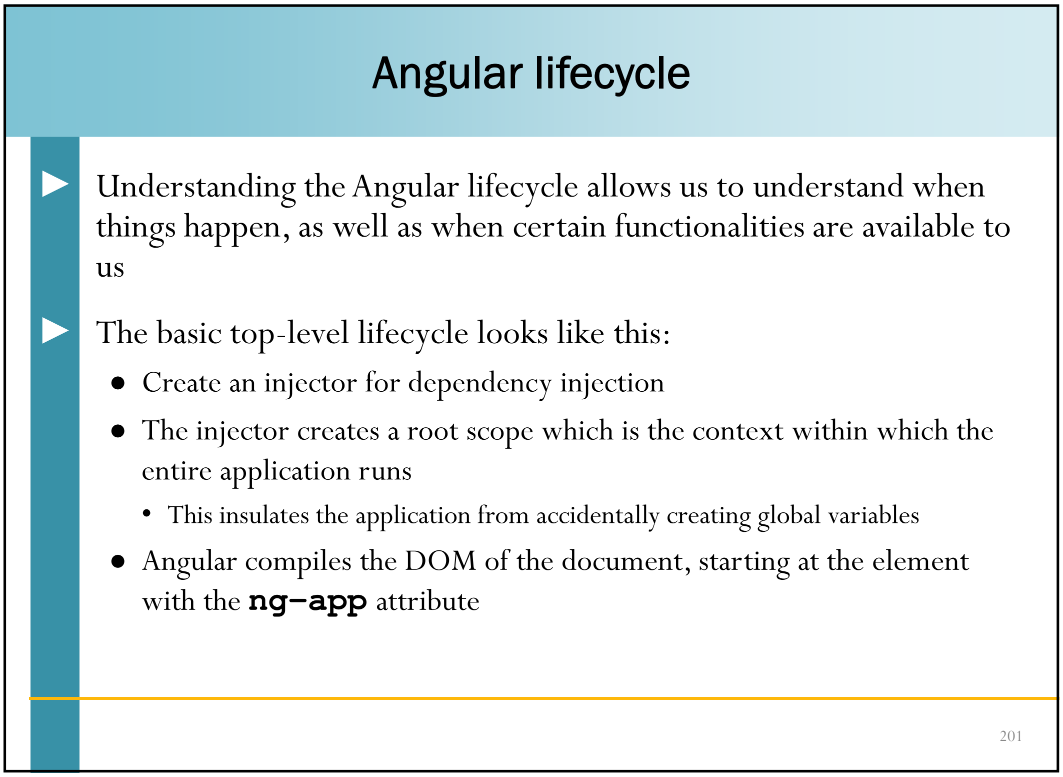
199



Chapter preview

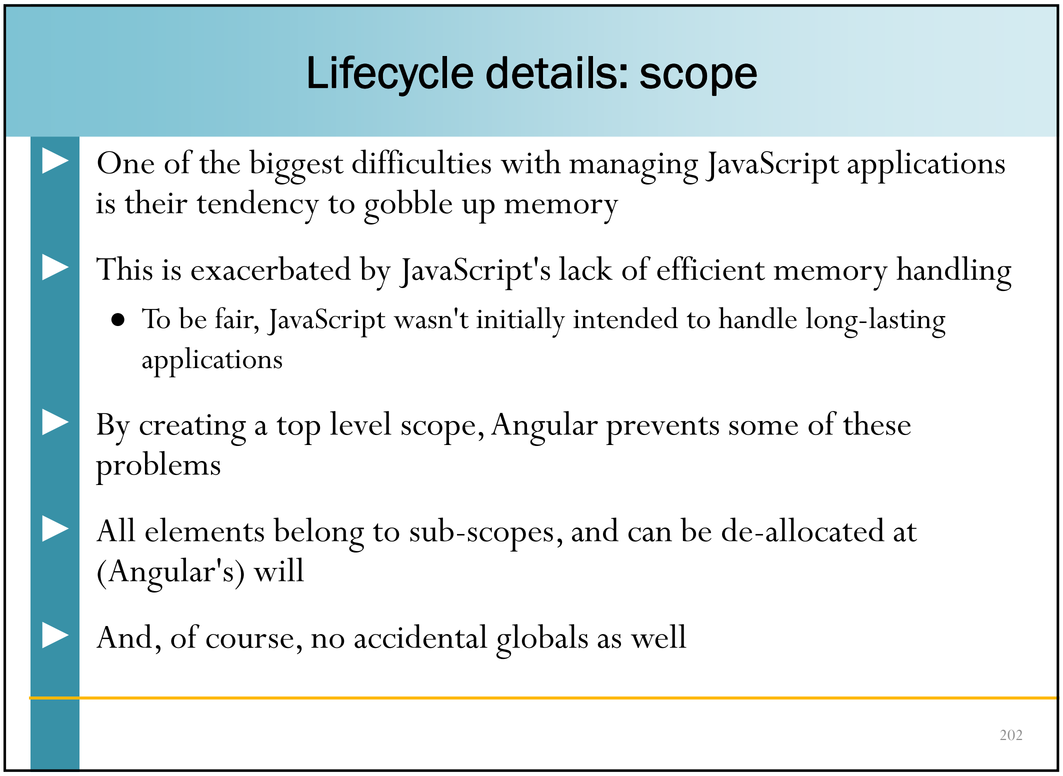
- Angular lifecycle
- Values (and constants)
- Factories
- Services

200



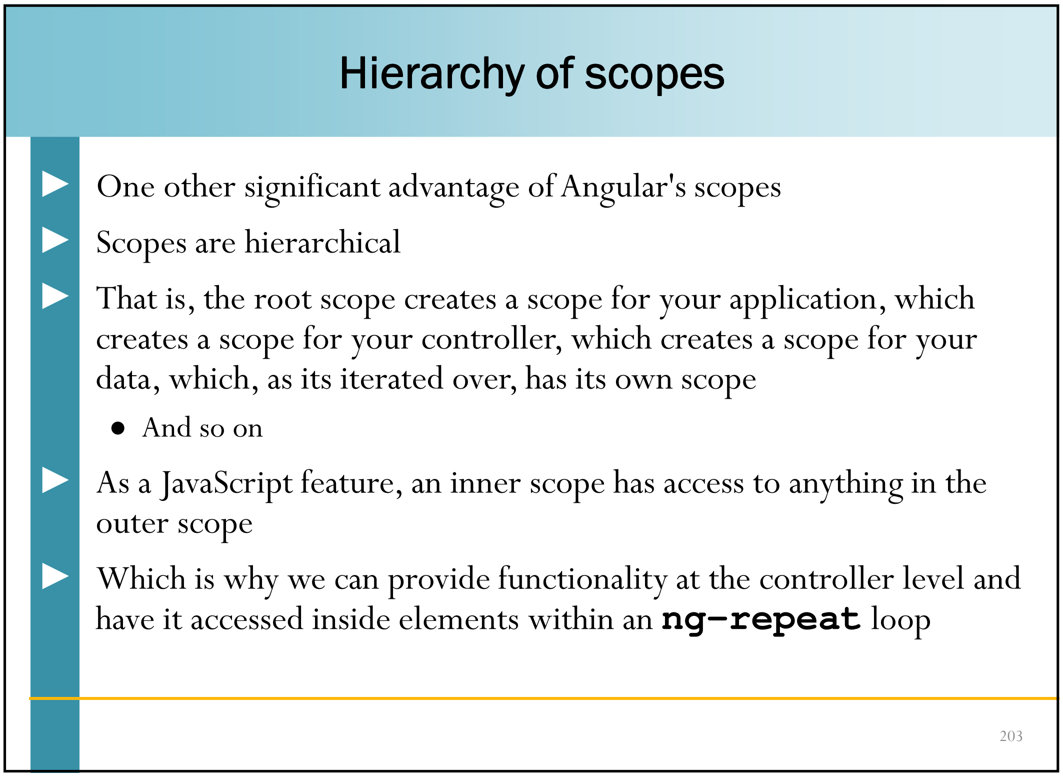
Angular lifecycle

- Understanding the Angular lifecycle allows us to understand when things happen, as well as when certain functionalities are available to us
- The basic top-level lifecycle looks like this:
  - Create an injector for dependency injection
  - The injector creates a root scope which is the context within which the entire application runs
    - This insulates the application from accidentally creating global variables
  - Angular compiles the DOM of the document, starting at the element with the **`ng-app`** attribute



Lifecycle details: scope

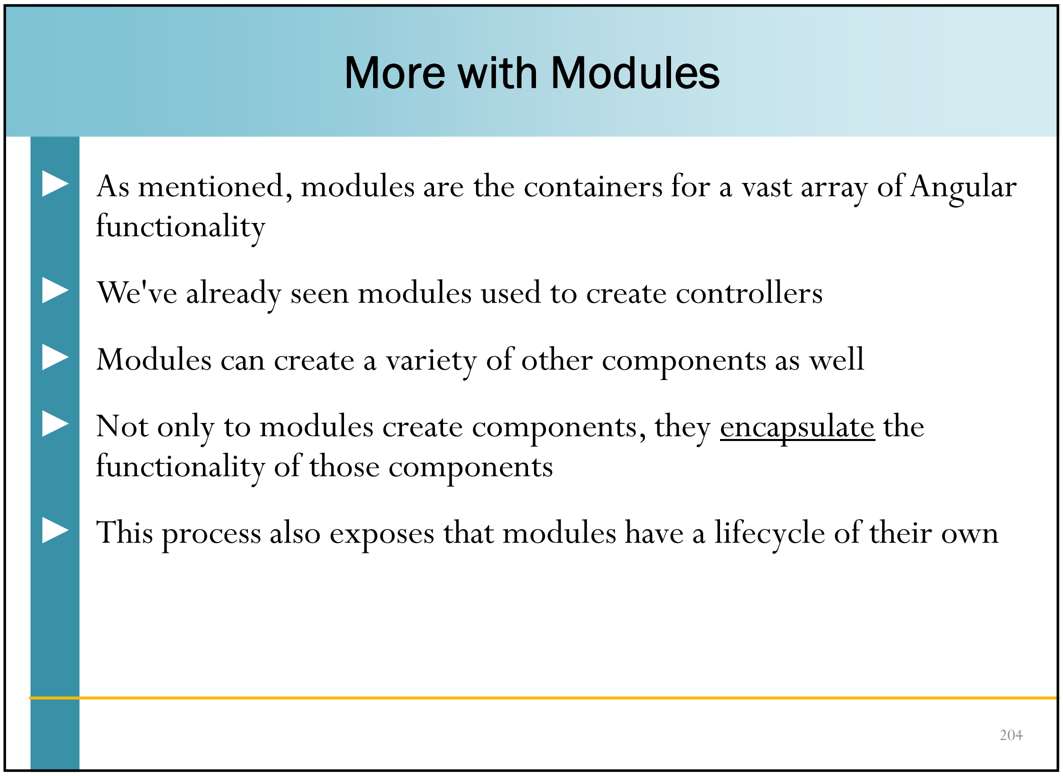
- One of the biggest difficulties with managing JavaScript applications is their tendency to gobble up memory
- This is exacerbated by JavaScript's lack of efficient memory handling
  - To be fair, JavaScript wasn't initially intended to handle long-lasting applications
- By creating a top level scope, Angular prevents some of these problems
- All elements belong to sub-scopes, and can be de-allocated at (Angular's) will
- And, of course, no accidental globals as well



Hierarchy of scopes

- One other significant advantage of Angular's scopes
- Scopes are hierarchical
- That is, the root scope creates a scope for your application, which creates a scope for your controller, which creates a scope for your data, which, as its iterated over, has its own scope
  - And so on
- As a JavaScript feature, an inner scope has access to anything in the outer scope
- Which is why we can provide functionality at the controller level and have it accessed inside elements within an **`ng-repeat`** loop

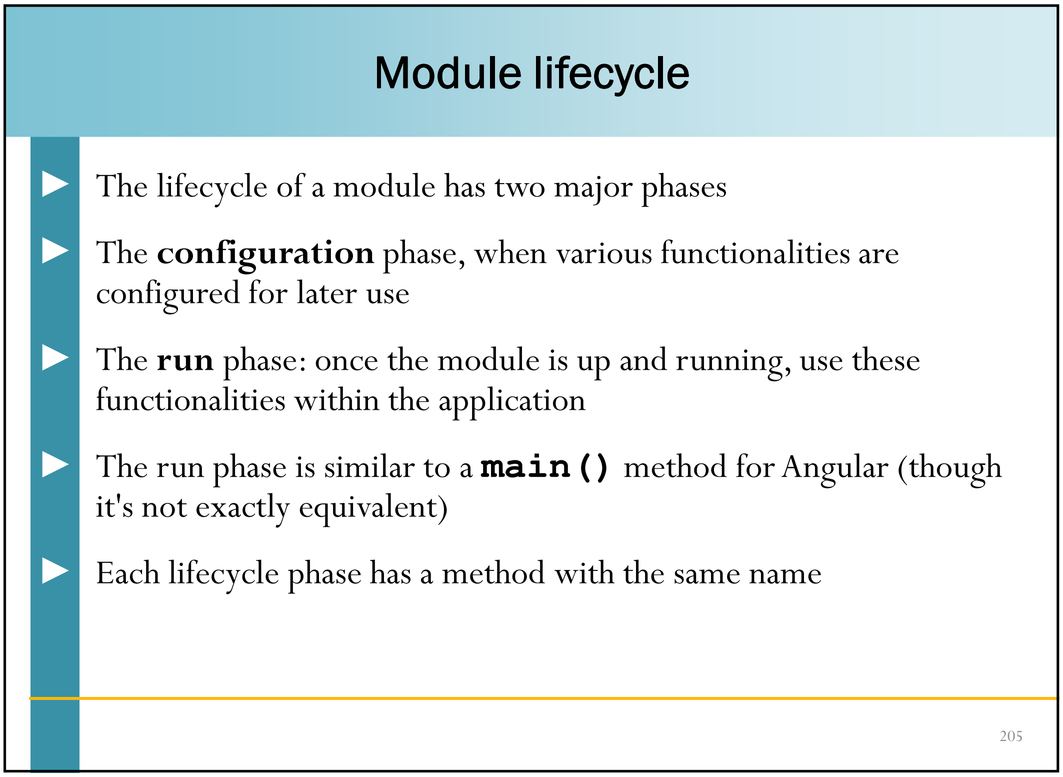
203



More with Modules

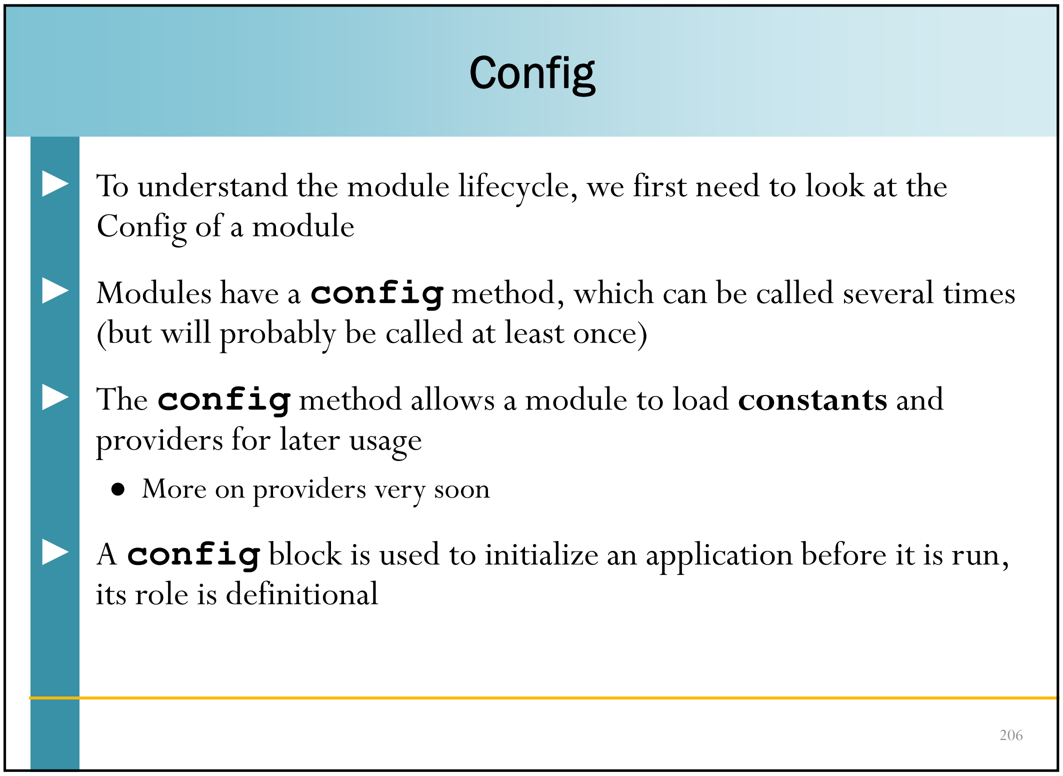
- As mentioned, modules are the containers for a vast array of Angular functionality
- We've already seen modules used to create controllers
- Modules can create a variety of other components as well
- Not only to modules create components, they encapsulate the functionality of those components
- This process also exposes that modules have a lifecycle of their own

204



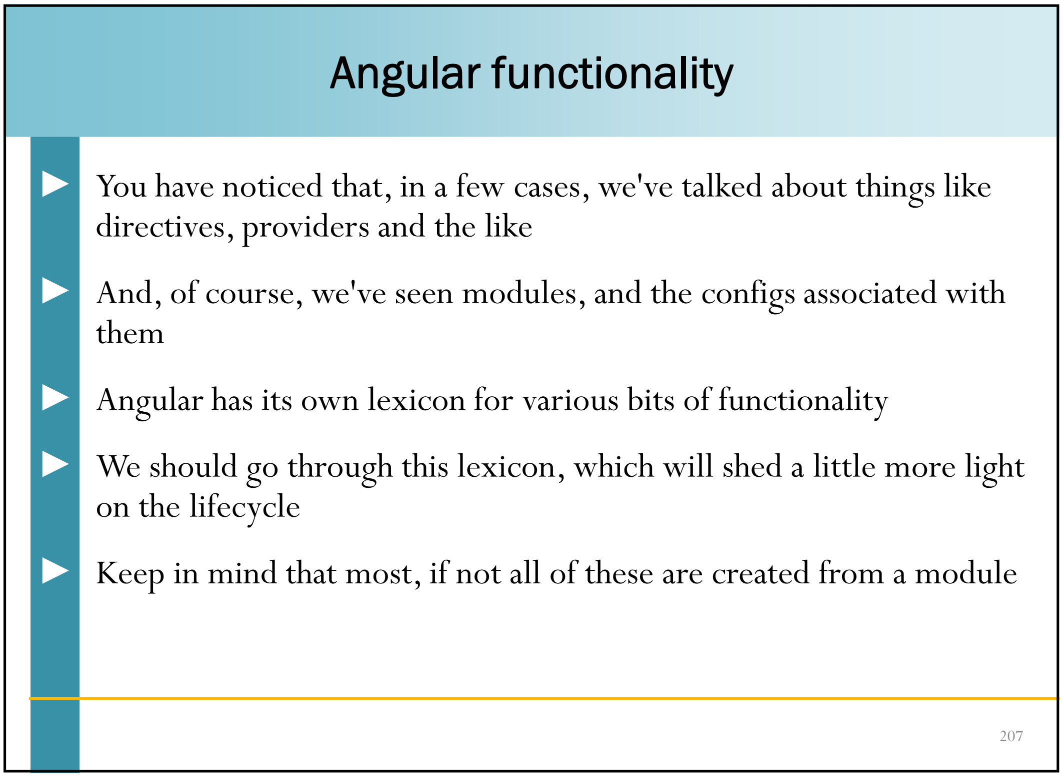
Module lifecycle

- The lifecycle of a module has two major phases
- The **configuration** phase, when various functionalities are configured for later use
- The **run** phase: once the module is up and running, use these functionalities within the application
- The run phase is similar to a **`main()`** method for Angular (though it's not exactly equivalent)
- Each lifecycle phase has a method with the same name



Config

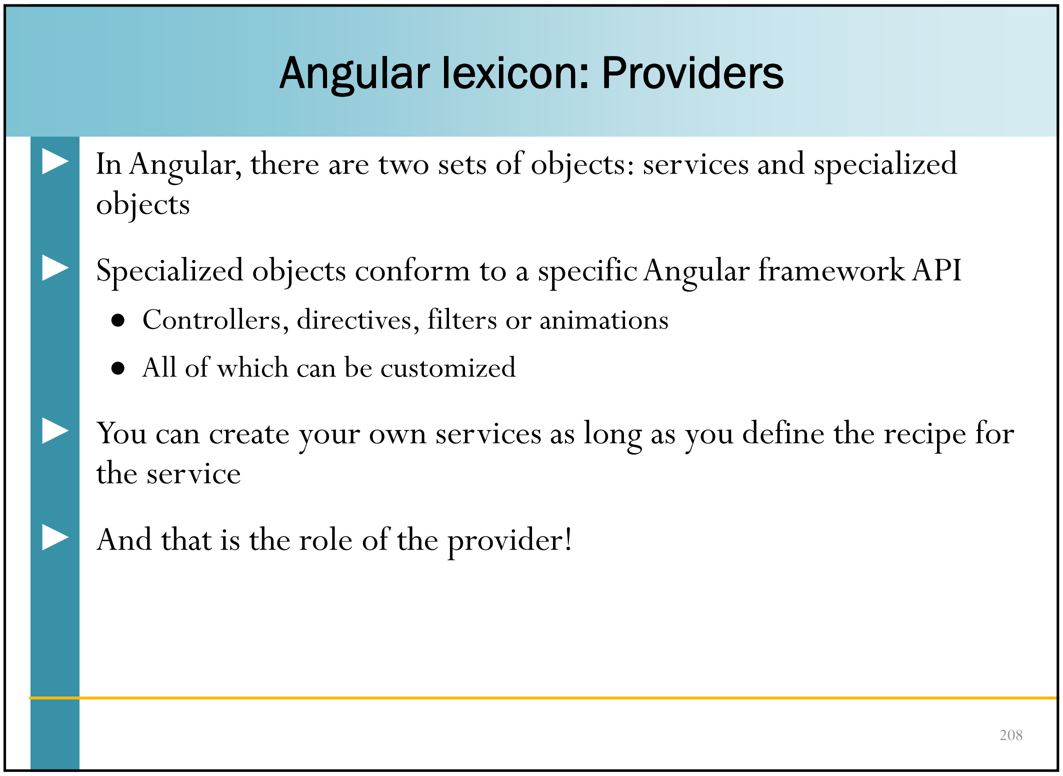
- To understand the module lifecycle, we first need to look at the Config of a module
- Modules have a **`config`** method, which can be called several times (but will probably be called at least once)
- The **`config`** method allows a module to load **constants** and providers for later usage
  - More on providers very soon
- A **`config`** block is used to initialize an application before it is run, its role is definitional



Angular functionality

- You have noticed that, in a few cases, we've talked about things like directives, providers and the like
- And, of course, we've seen modules, and the configs associated with them
- Angular has its own lexicon for various bits of functionality
- We should go through this lexicon, which will shed a little more light on the lifecycle
- Keep in mind that most, if not all of these are created from a module

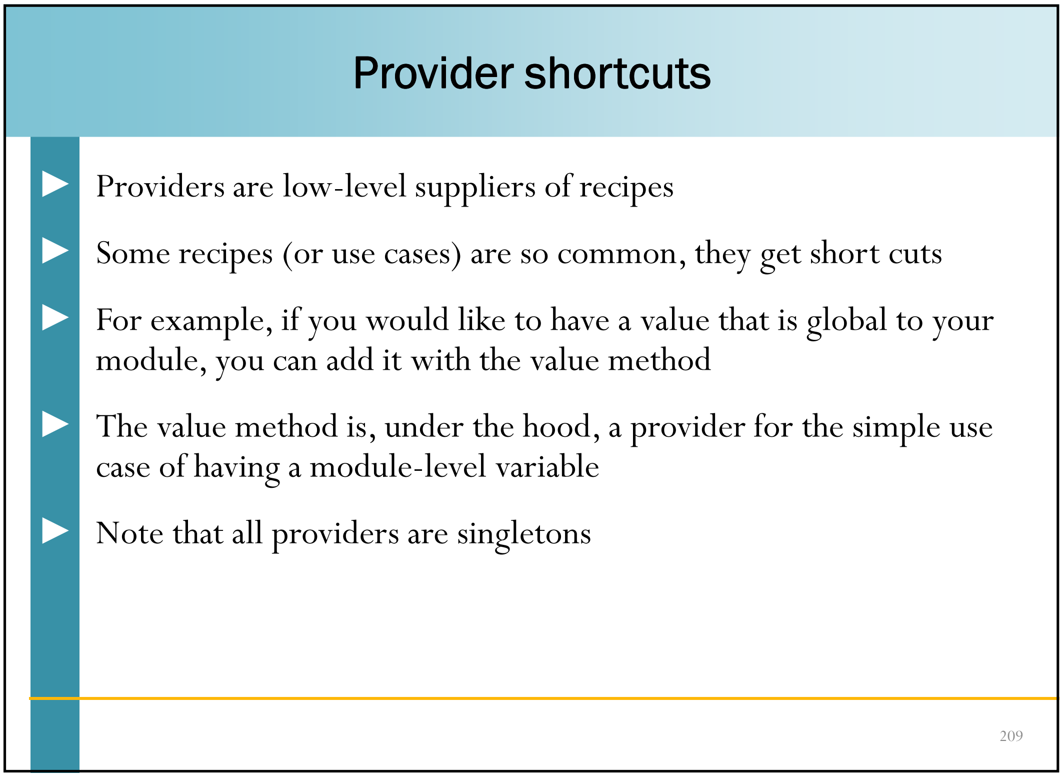
207



Angular lexicon: Providers

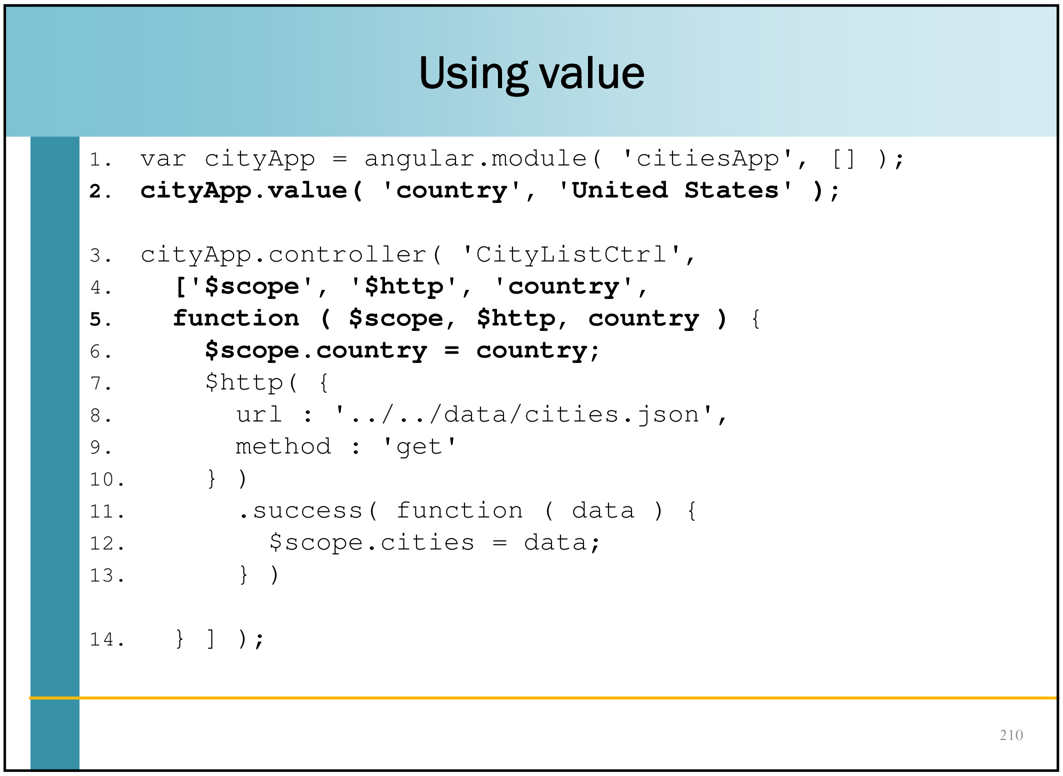
- In Angular, there are two sets of objects: services and specialized objects
- Specialized objects conform to a specific Angular framework API
  - Controllers, directives, filters or animations
  - All of which can be customized
- You can create your own services as long as you define the recipe for the service
- And that is the role of the provider!

208



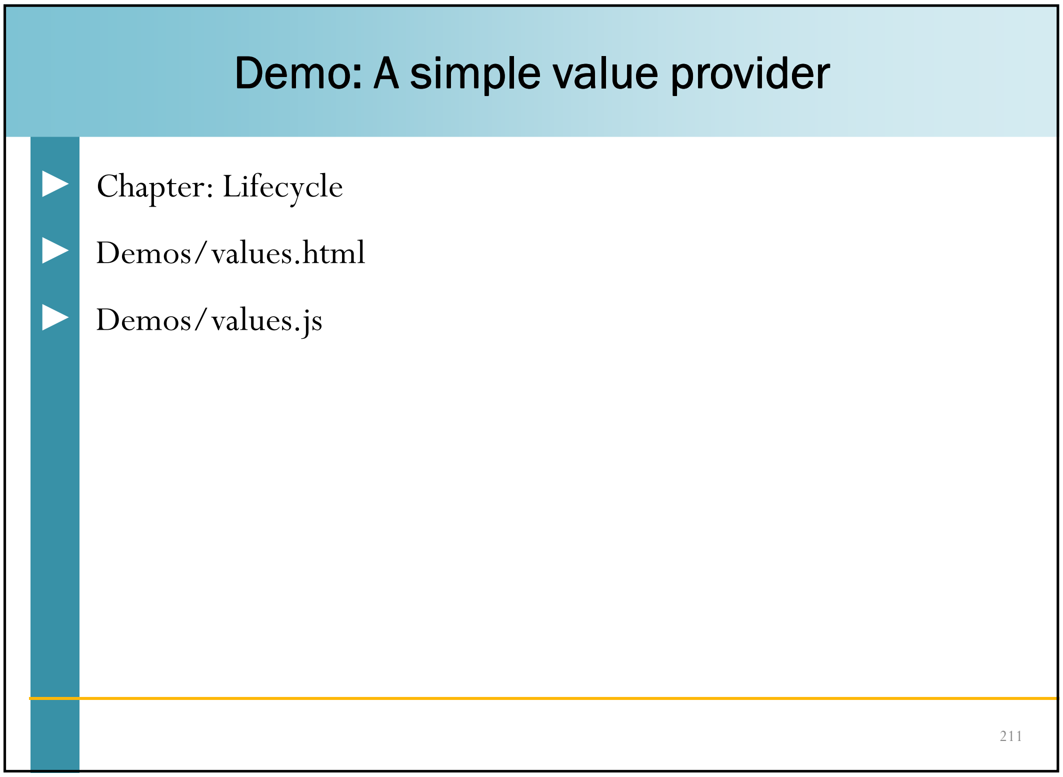
Provider shortcuts

- Providers are low-level suppliers of recipes
- Some recipes (or use cases) are so common, they get short cuts
- For example, if you would like to have a value that is global to your module, you can add it with the value method
- The value method is, under the hood, a provider for the simple use case of having a module-level variable
- Note that all providers are singletons



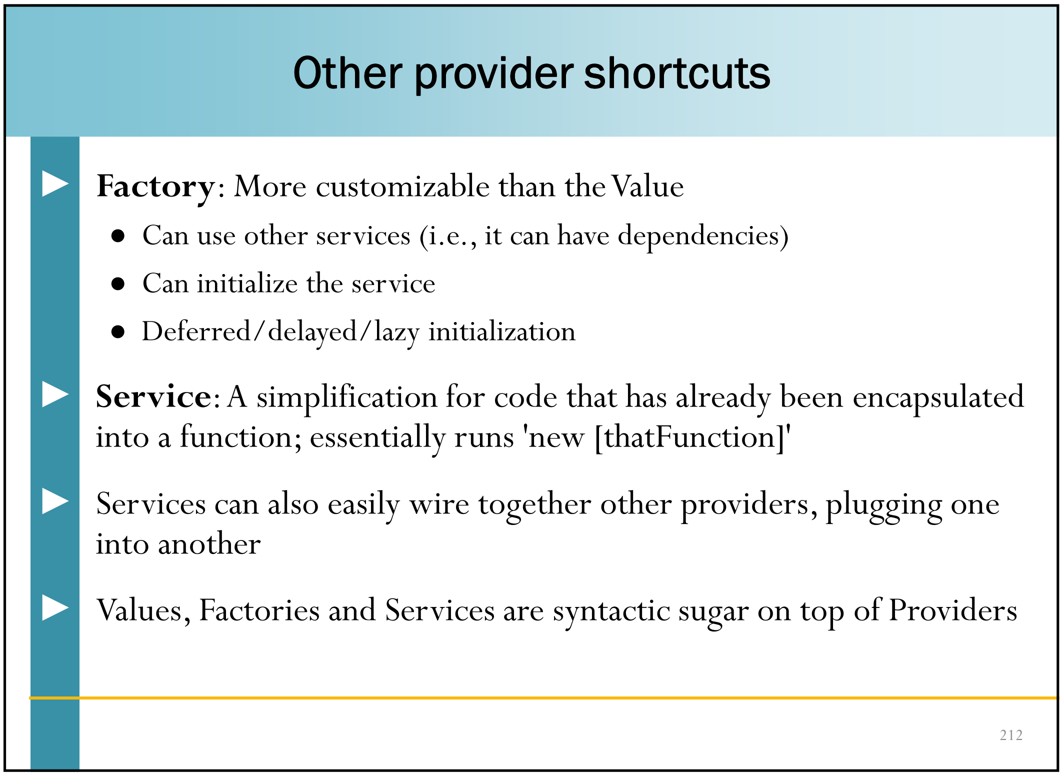
Using value

```
var cityApp = angular.module( 'citiesApp', [] );
cityApp.value( 'country', 'United States' );

cityApp.controller( 'CityListCtrl',
  ['$scope', '$http', 'country',
  function ( $scope, $http, country ) {
    $scope.country = country;
    $http( {
      url : '../../data/cities.json',
      method : 'get'
    } )
      .success( function ( data ) {
        $scope.cities = data;
      } )

  } ] );
```



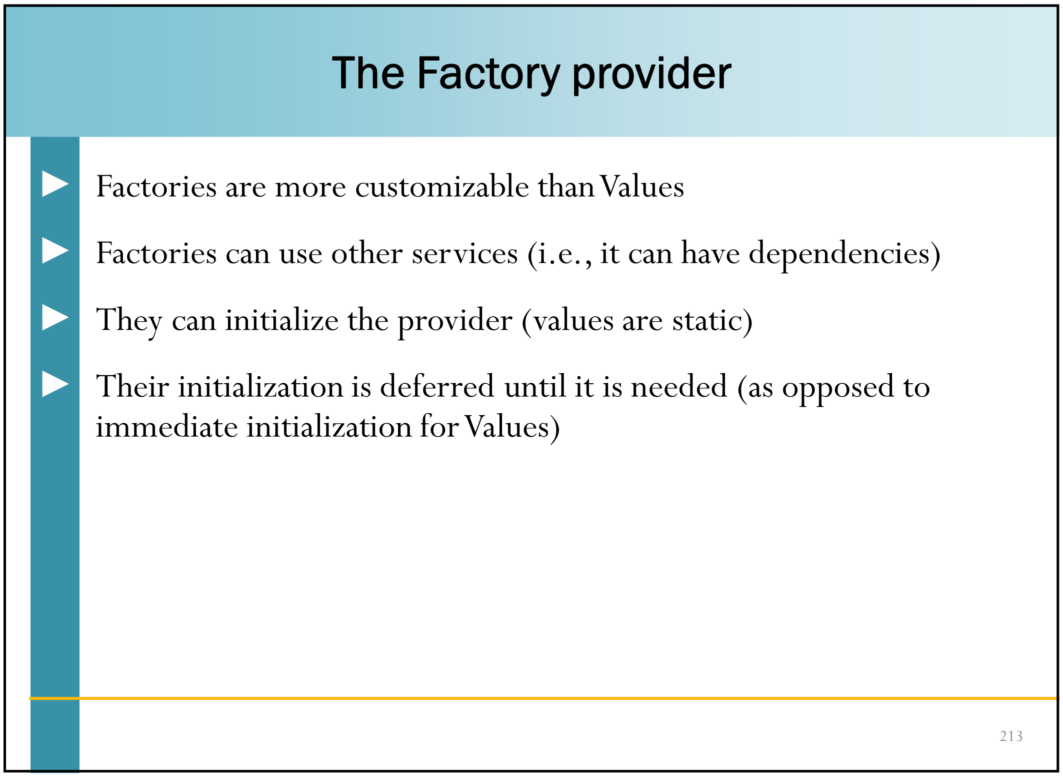
Demo: A simple value provider

- Chapter: Lifecycle
- Demos/values.html
- Demos/values.js



Other provider shortcuts

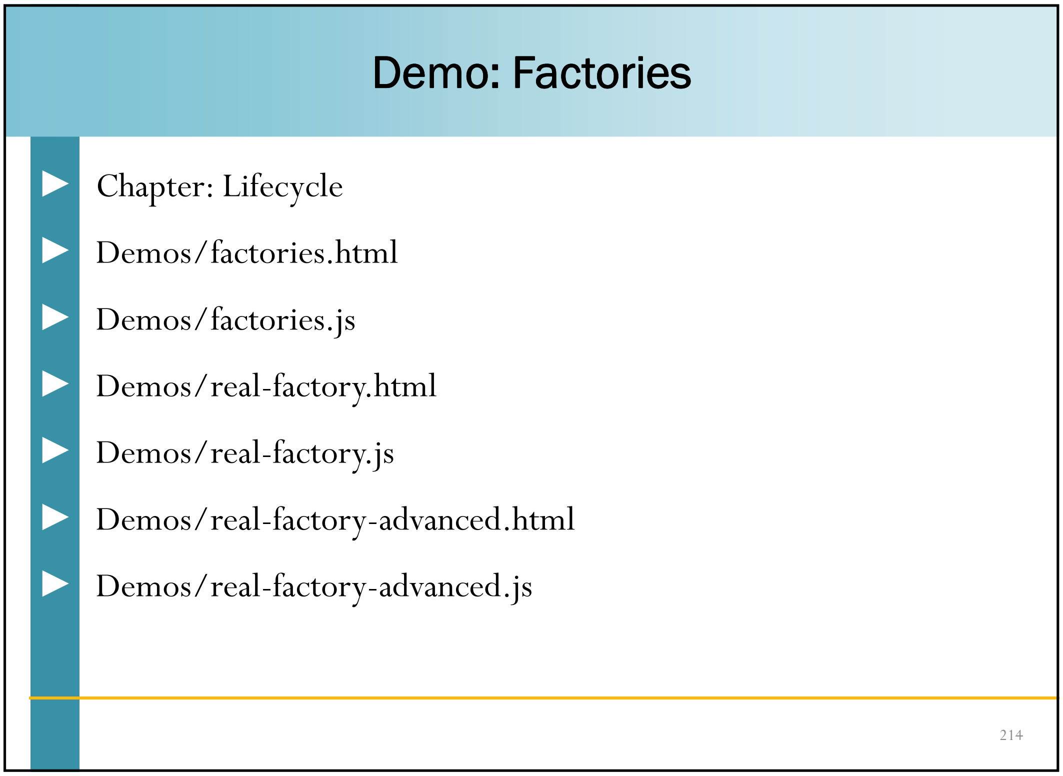
- **Factory**: More customizable than the Value
  - Can use other services (i.e., it can have dependencies)
  - Can initialize the service
  - Deferred/delayed/lazy initialization
- **Service**: A simplification for code that has already been encapsulated into a function; essentially runs 'new [thatFunction]'
- Services can also easily wire together other providers, plugging one into another
- Values, Factories and Services are syntactic sugar on top of Providers



The Factory provider

- Factories are more customizable than Values
- Factories can use other services (i.e., it can have dependencies)
- They can initialize the provider (values are static)
- Their initialization is deferred until it is needed (as opposed to immediate initialization for Values)

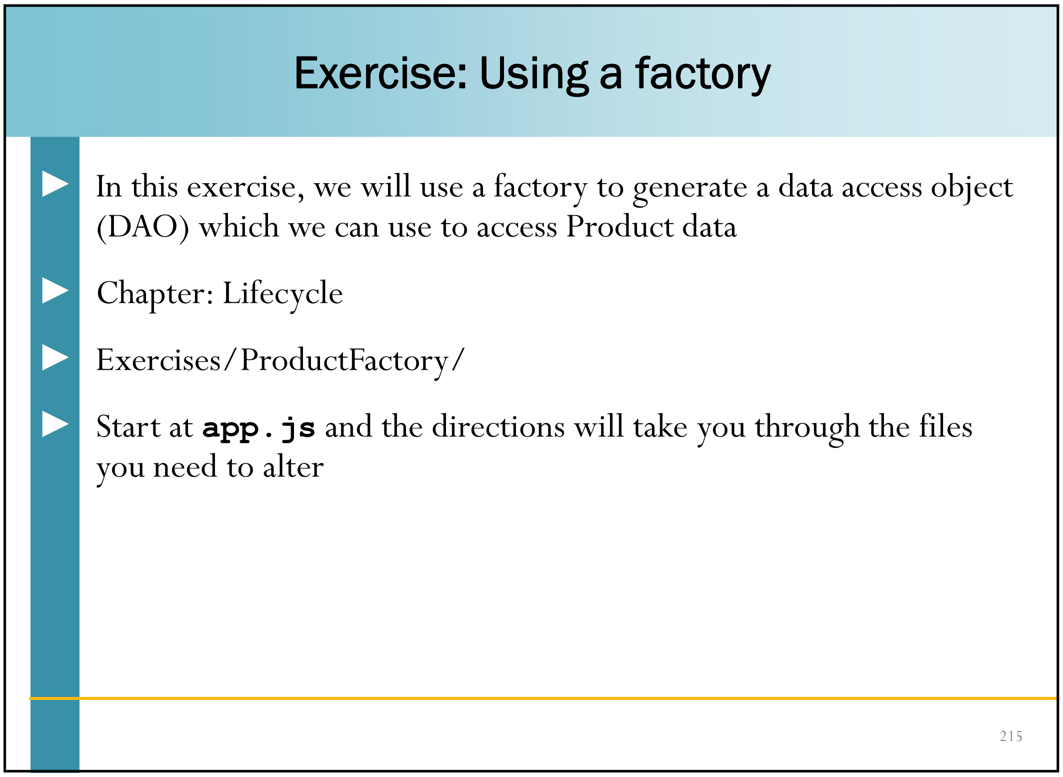
213



Demo: Factories

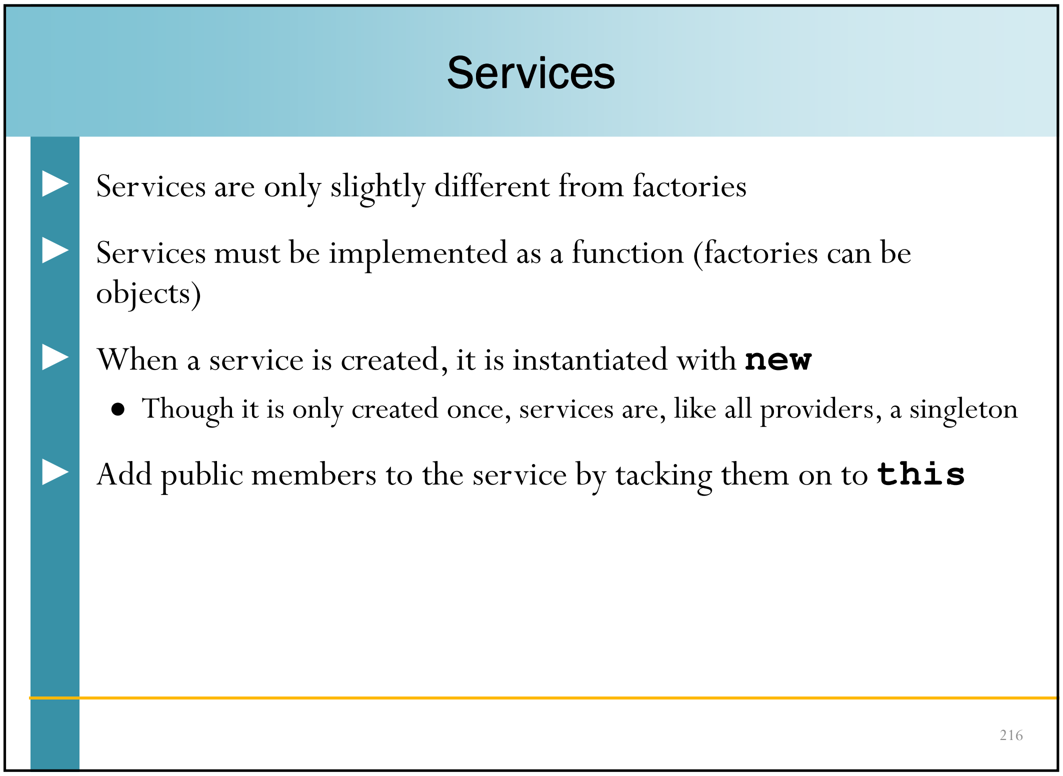
- Chapter: Lifecycle
- Demos/factories.html
- Demos/factories.js
- Demos/real-factory.html
- Demos/real-factory.js
- Demos/real-factory-advanced.html
- Demos/real-factory-advanced.js

214



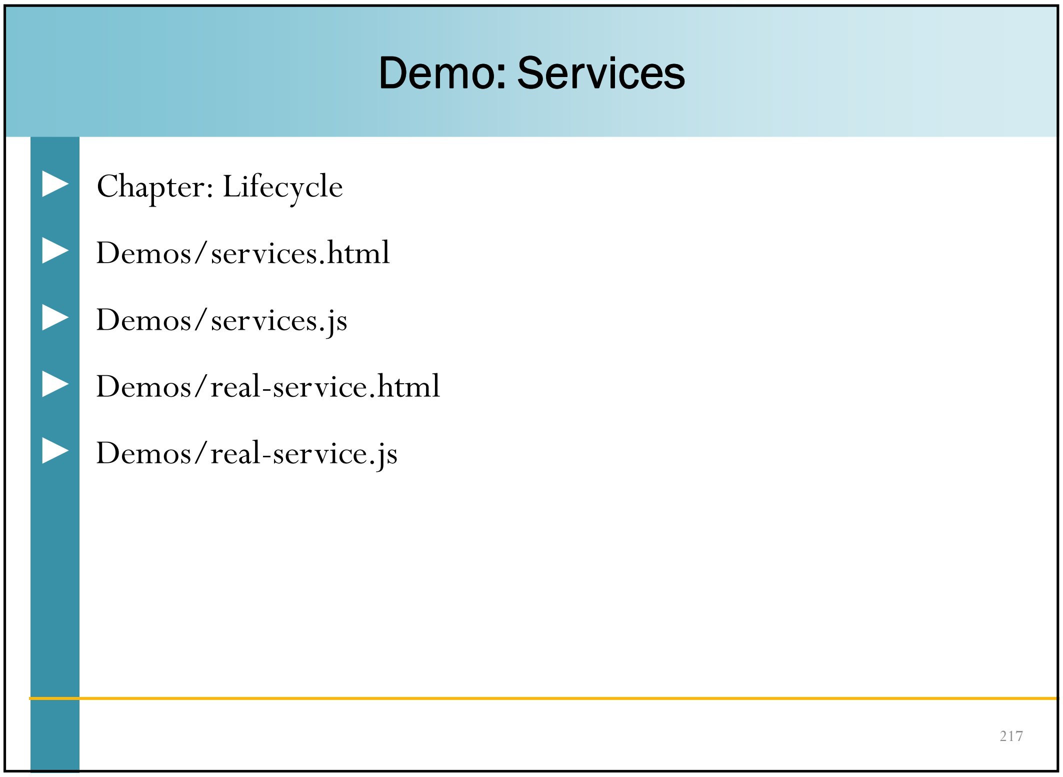
Exercise: Using a factory

- In this exercise, we will use a factory to generate a data access object (DAO) which we can use to access Product data
- Chapter: Lifecycle
- Exercises/ProductFactory/
- Start at **`app.js`** and the directions will take you through the files you need to alter



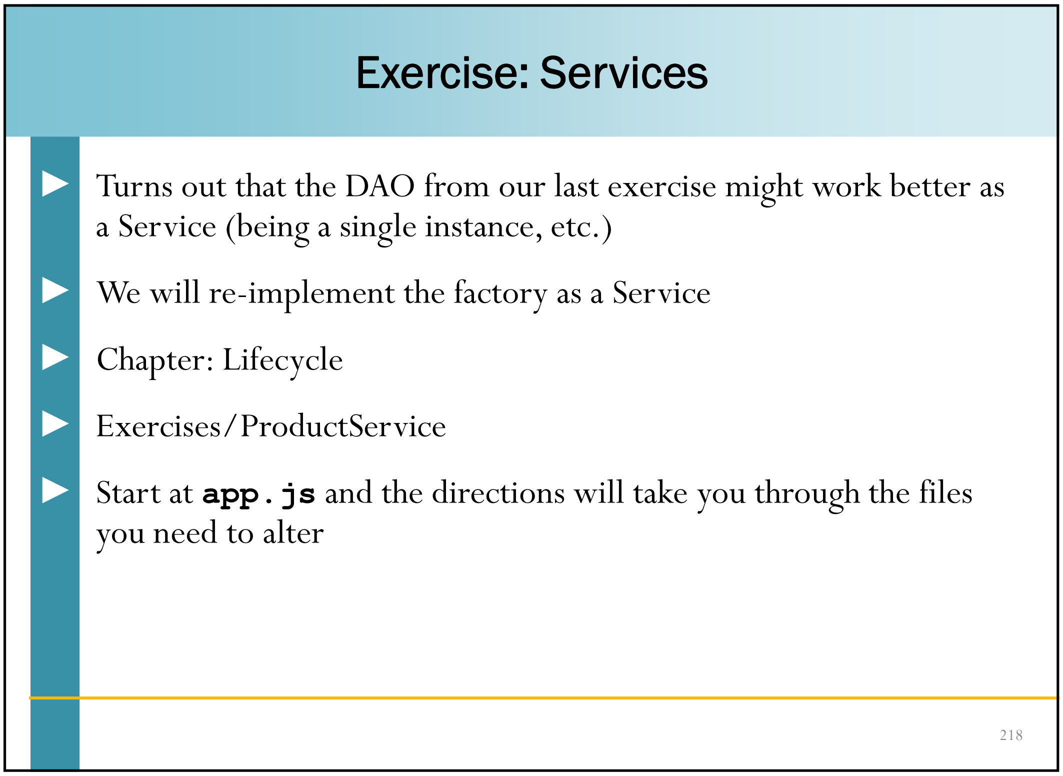
Services

- Services are only slightly different from factories
- Services must be implemented as a function (factories can be objects)
- When a service is created, it is instantiated with **`new`**
  - Though it is only created once, services are, like all providers, a singleton
- Add public members to the service by tacking them on to **`this`**



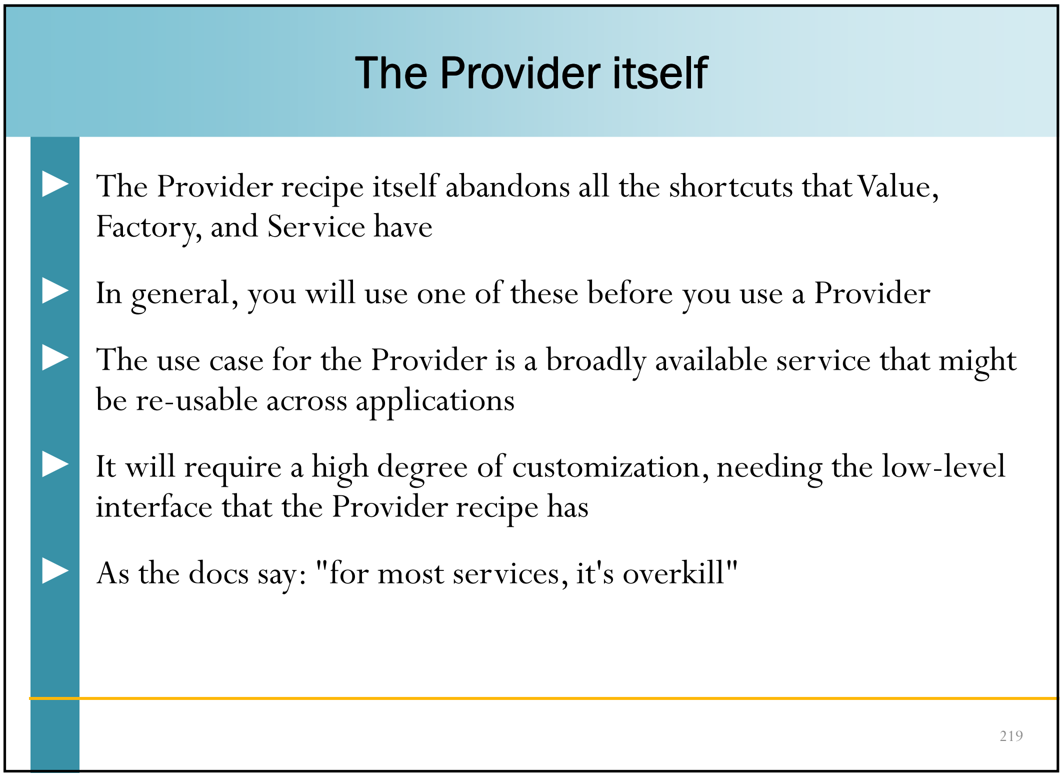
Demo: Services

- Chapter: Lifecycle
- Demos/services.html
- Demos/services.js
- Demos/real-service.html
- Demos/real-service.js



Exercise: Services

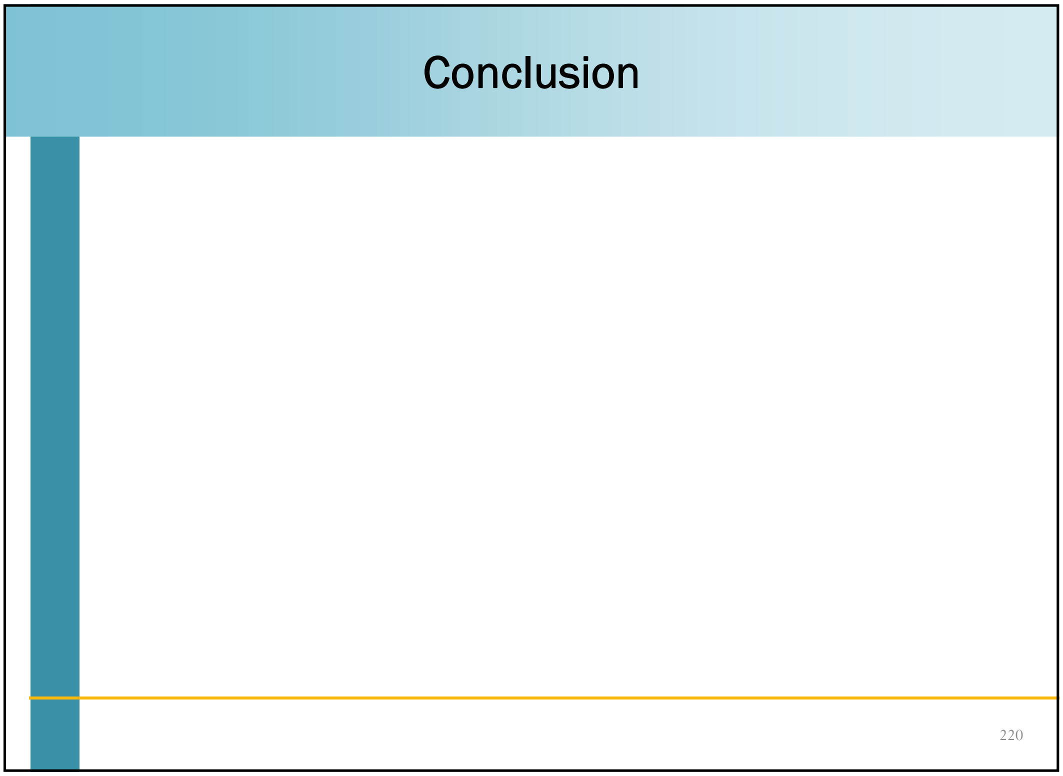
- Turns out that the DAO from our last exercise might work better as a Service (being a single instance, etc.)
- We will re-implement the factory as a Service
- Chapter: Lifecycle
- Exercises/ProductService
- Start at **`app.js`** and the directions will take you through the files you need to alter



The Provider itself

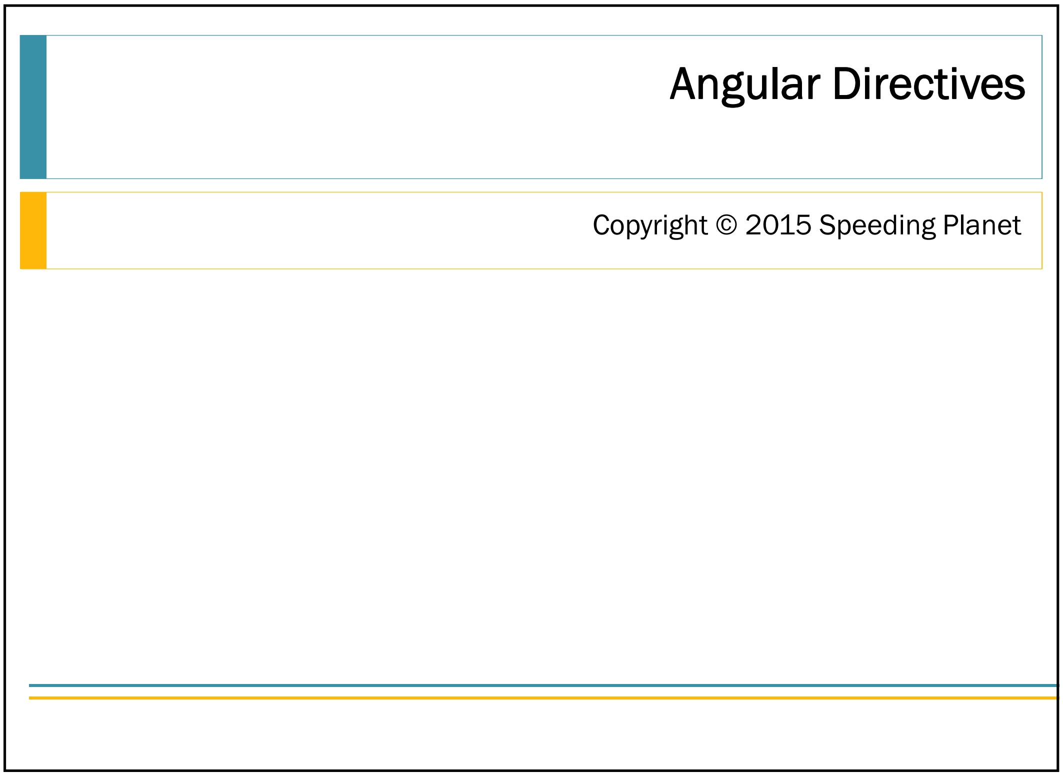
- The Provider recipe itself abandons all the shortcuts that Value, Factory, and Service have
- In general, you will use one of these before you use a Provider
- The use case for the Provider is a broadly available service that might be re-usable across applications
- It will require a high degree of customization, needing the low-level interface that the Provider recipe has
- As the docs say: "for most services, it's overkill"

219

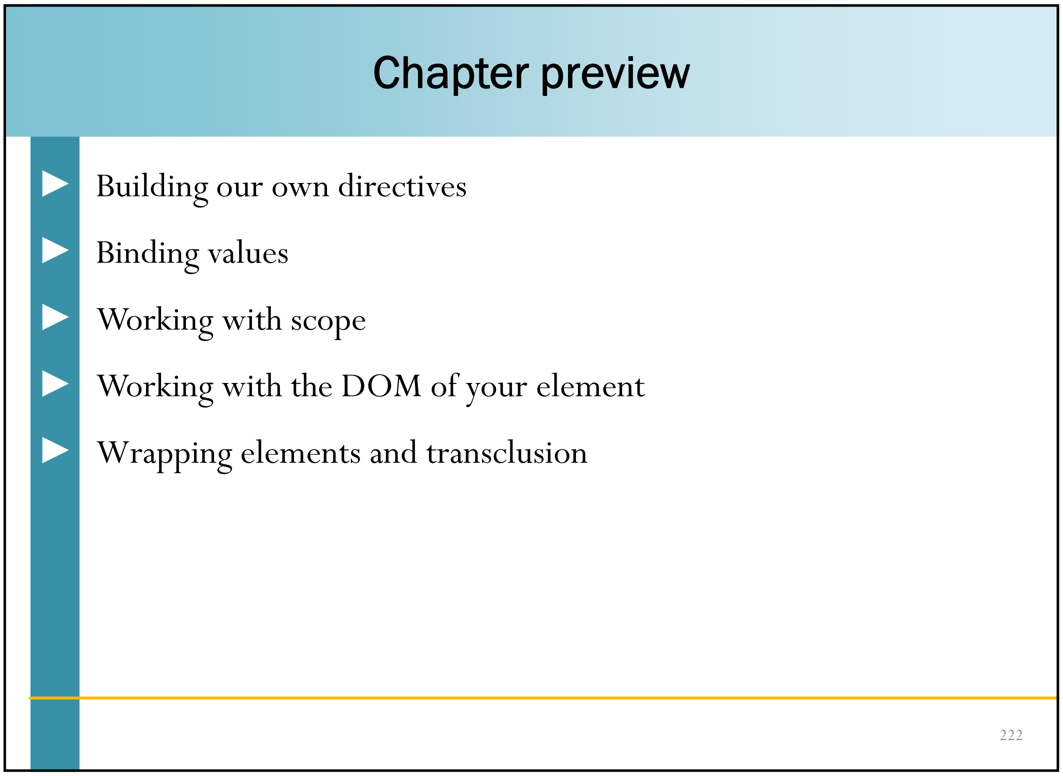


Conclusion

220



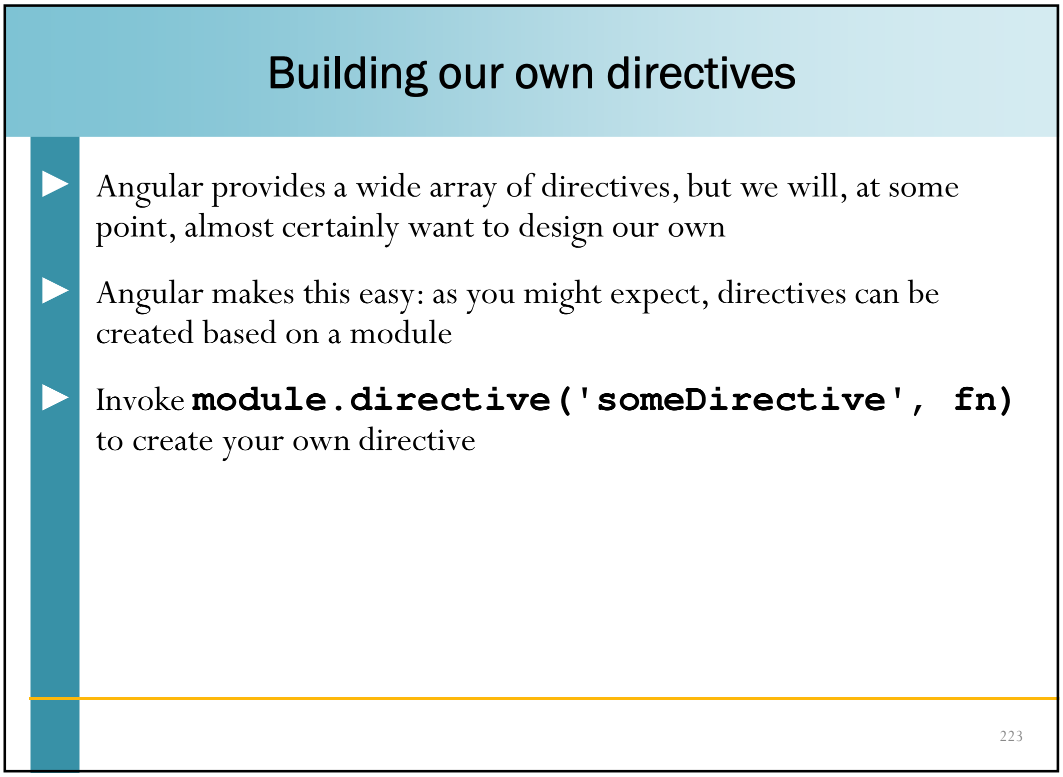
Angular Directives



Chapter preview

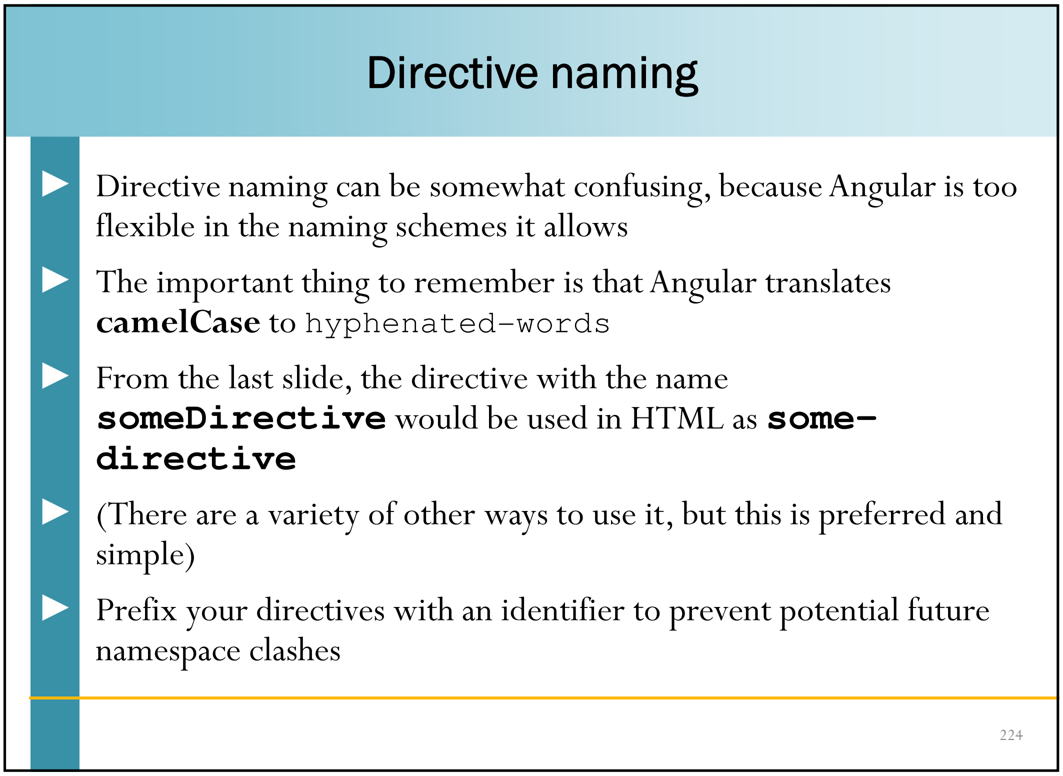
- Building our own directives
- Binding values
- Working with scope
- Working with the DOM of your element
- Wrapping elements and transclusion

222



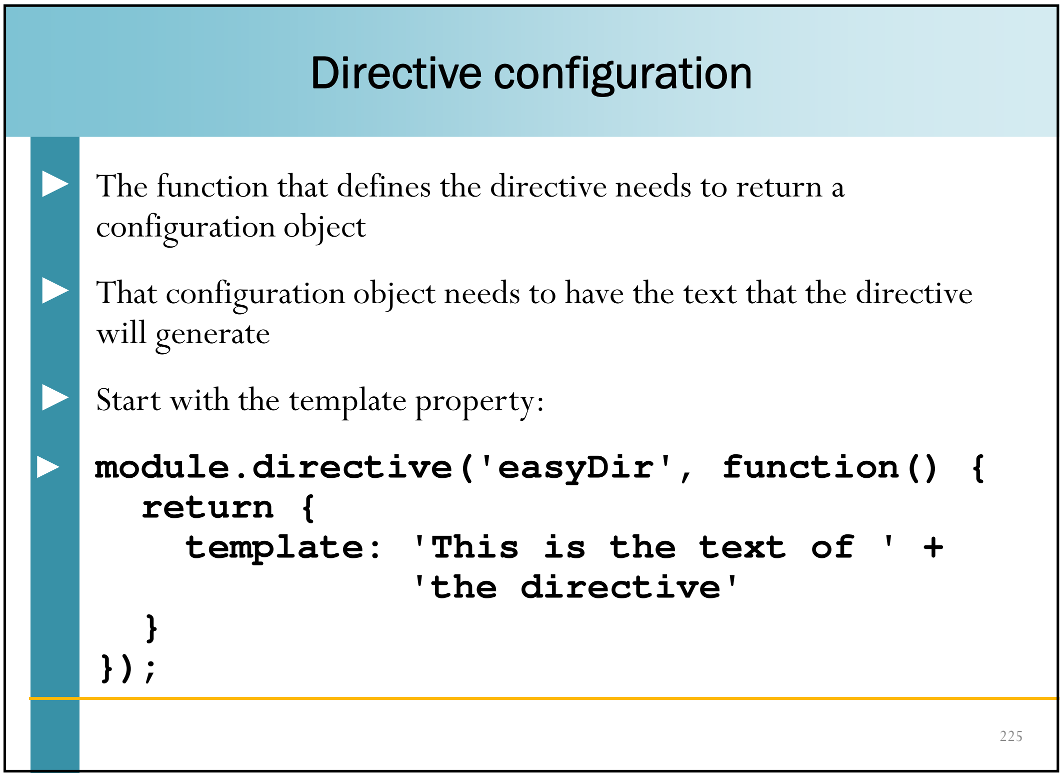
Building our own directives

- Angular provides a wide array of directives, but we will, at some point, almost certainly want to design our own
- Angular makes this easy: as you might expect, directives can be created based on a module
- Invoke **`module.directive('someDirective', fn)`** to create your own directive



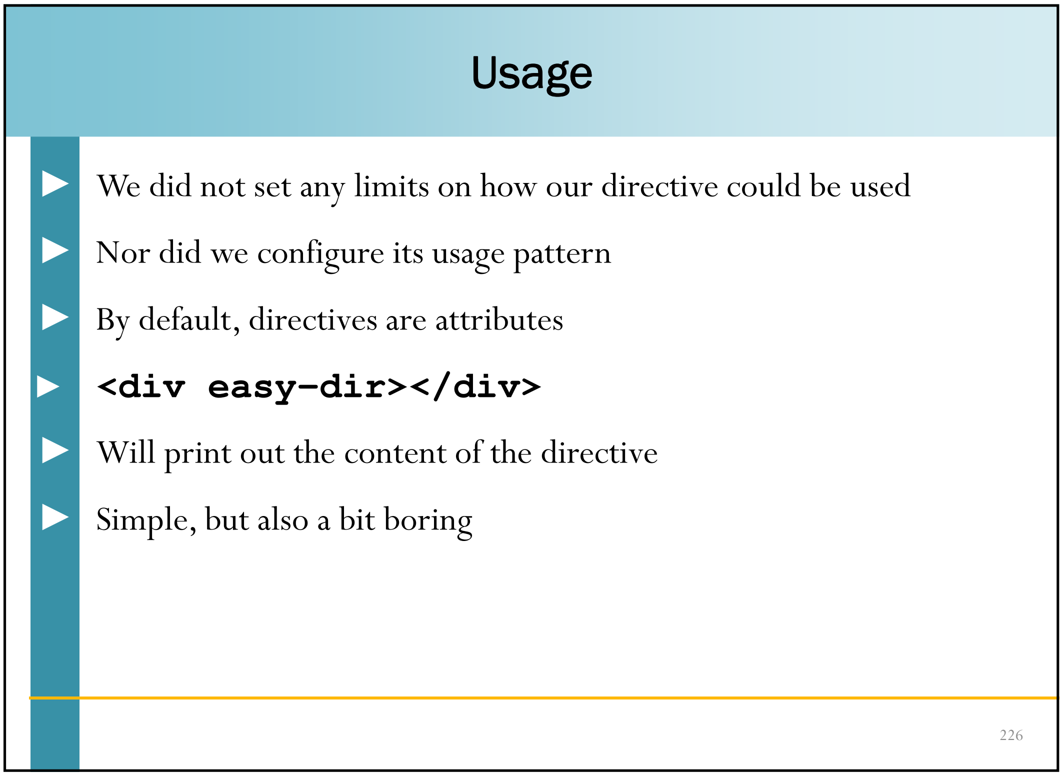
Directive naming

- Directive naming can be somewhat confusing, because Angular is too flexible in the naming schemes it allows
- The important thing to remember is that Angular translates **camelCase** to `hyphenated-words`
- From the last slide, the directive with the name **`someDirective`** would be used in HTML as **`some-directive`**
- (There are a variety of other ways to use it, but this is preferred and simple)
- Prefix your directives with an identifier to prevent potential future namespace clashes



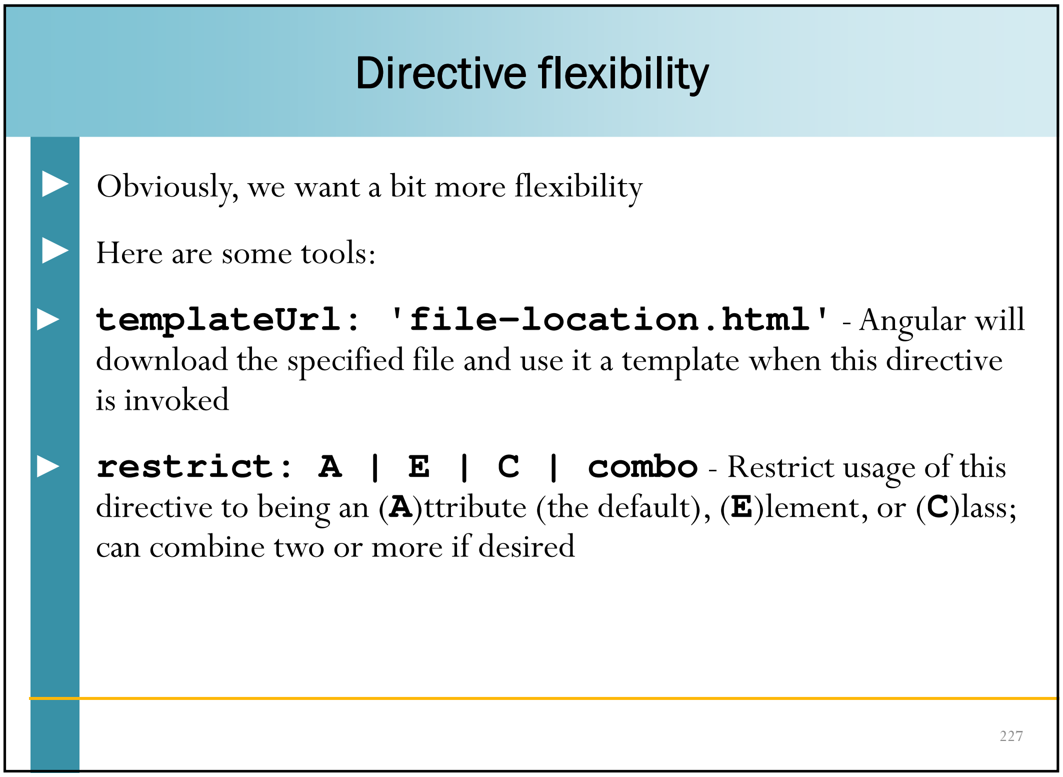
Directive configuration

- The function that defines the directive needs to return a configuration object
- That configuration object needs to have the text that the directive will generate
- Start with the template property:
-
```
module.directive('easyDir', function() {
  return {
    template: 'This is the text of ' +
              'the directive'
  }
});
```



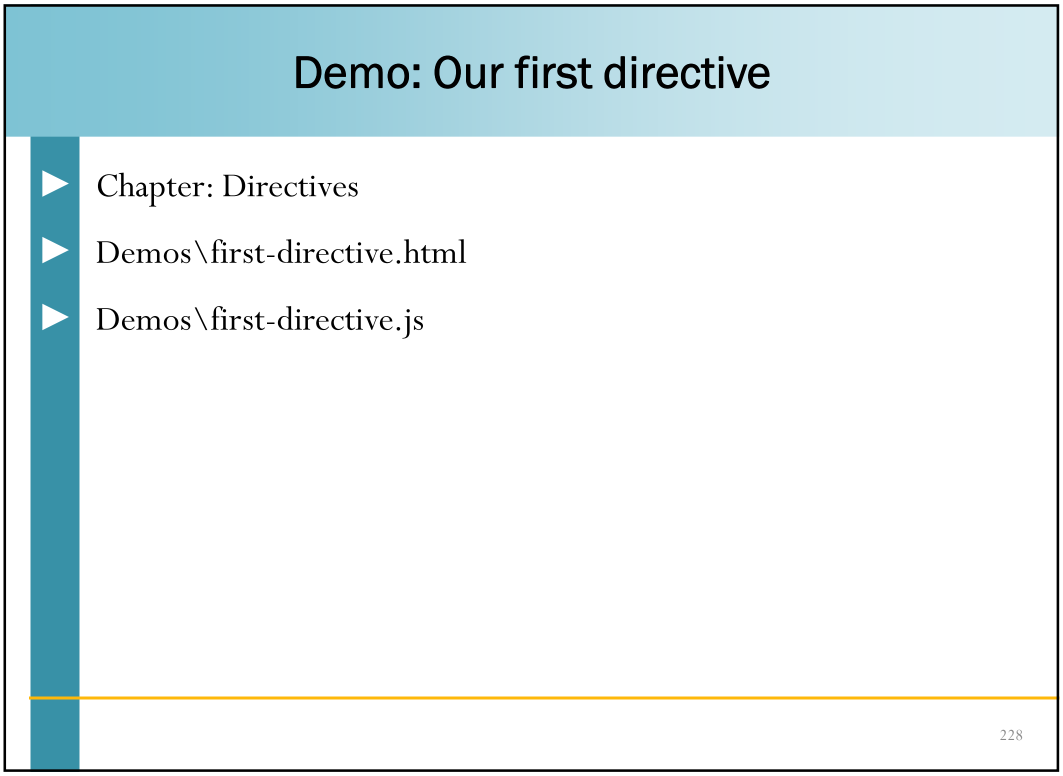
Usage

- We did not set any limits on how our directive could be used
- Nor did we configure its usage pattern
- By default, directives are attributes
- `<div easy-dir></div>`
- Will print out the content of the directive
- Simple, but also a bit boring



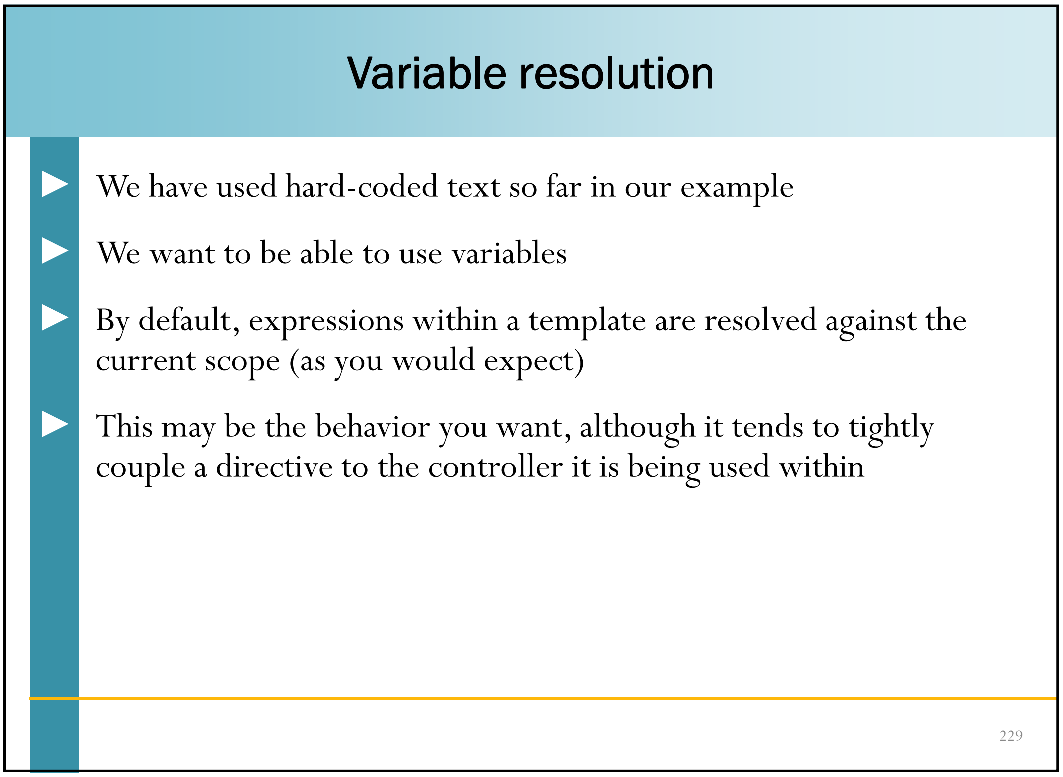
Directive flexibility

- Obviously, we want a bit more flexibility
- Here are some tools:
- **`templateUrl: 'file-location.html'`** - Angular will download the specified file and use it a template when this directive is invoked
- **`restrict: A | E | C | combo`** - Restrict usage of this directive to being an (**A**)ttribute (the default), (**E**)lement, or (**C**)lass; can combine two or more if desired



Demo: Our first directive

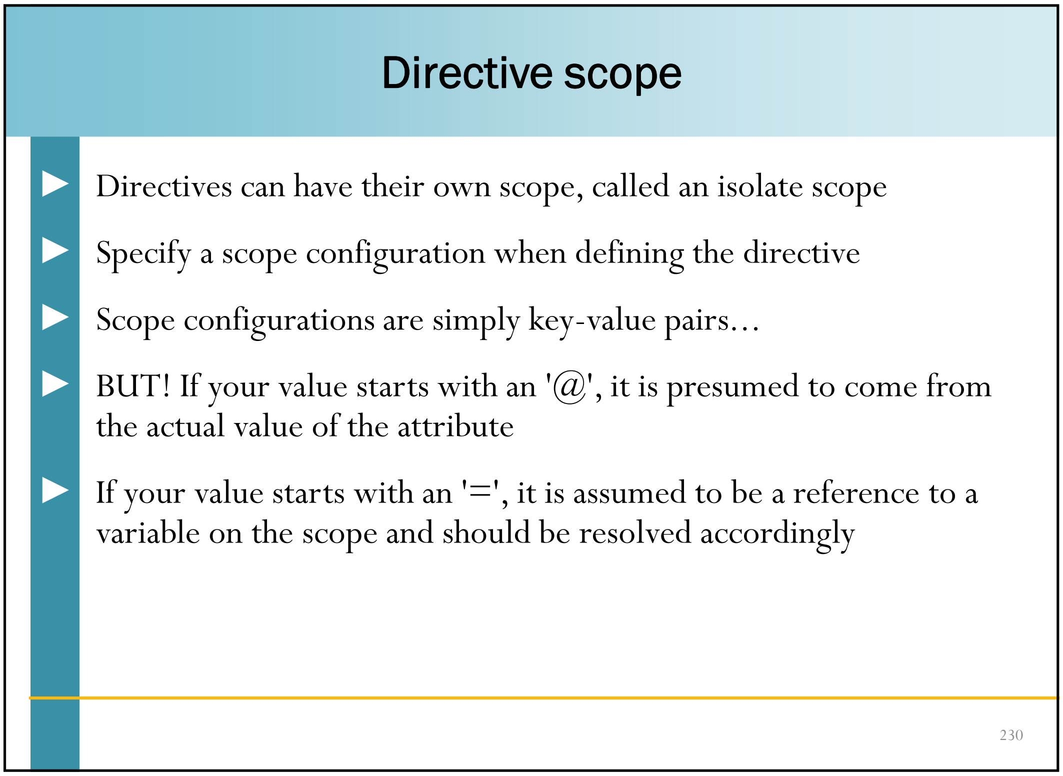
- Chapter: Directives
- Demos\first-directive.html
- Demos\first-directive.js



Variable resolution

- We have used hard-coded text so far in our example
- We want to be able to use variables
- By default, expressions within a template are resolved against the current scope (as you would expect)
- This may be the behavior you want, although it tends to tightly couple a directive to the controller it is being used within

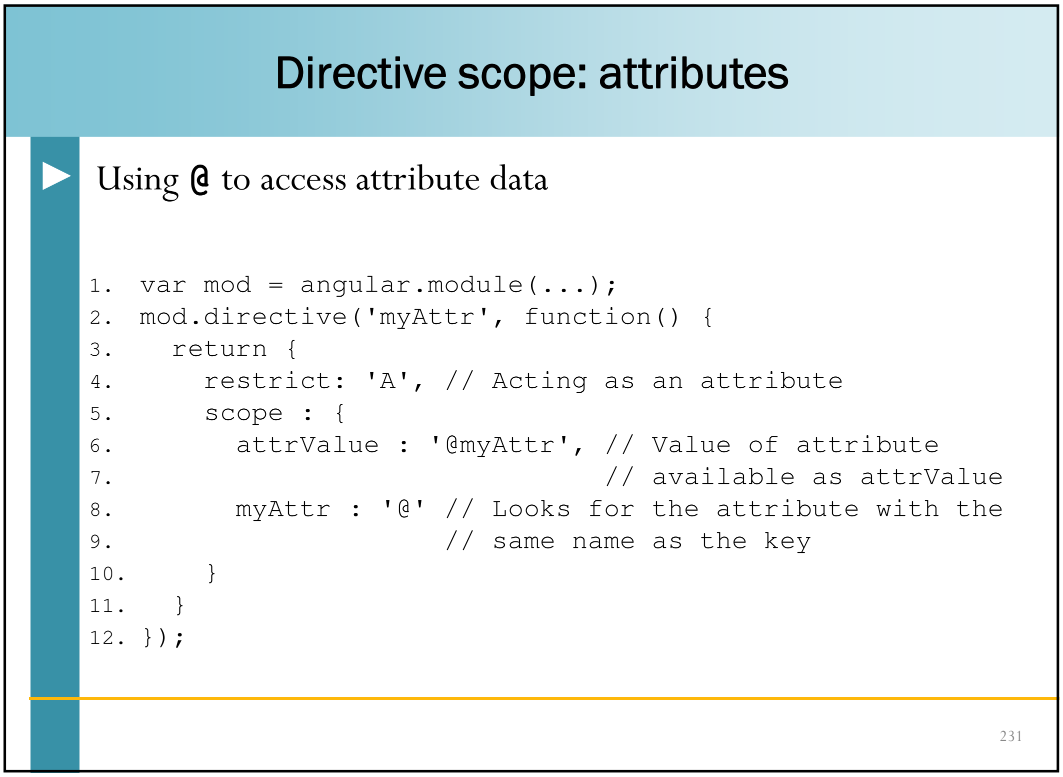
229



Directive scope

- Directives can have their own scope, called an isolate scope
- Specify a scope configuration when defining the directive
- Scope configurations are simply key-value pairs…
- BUT! If your value starts with an '@', it is presumed to come from the actual value of the attribute
- If your value starts with an '=', it is assumed to be a reference to a variable on the scope and should be resolved accordingly

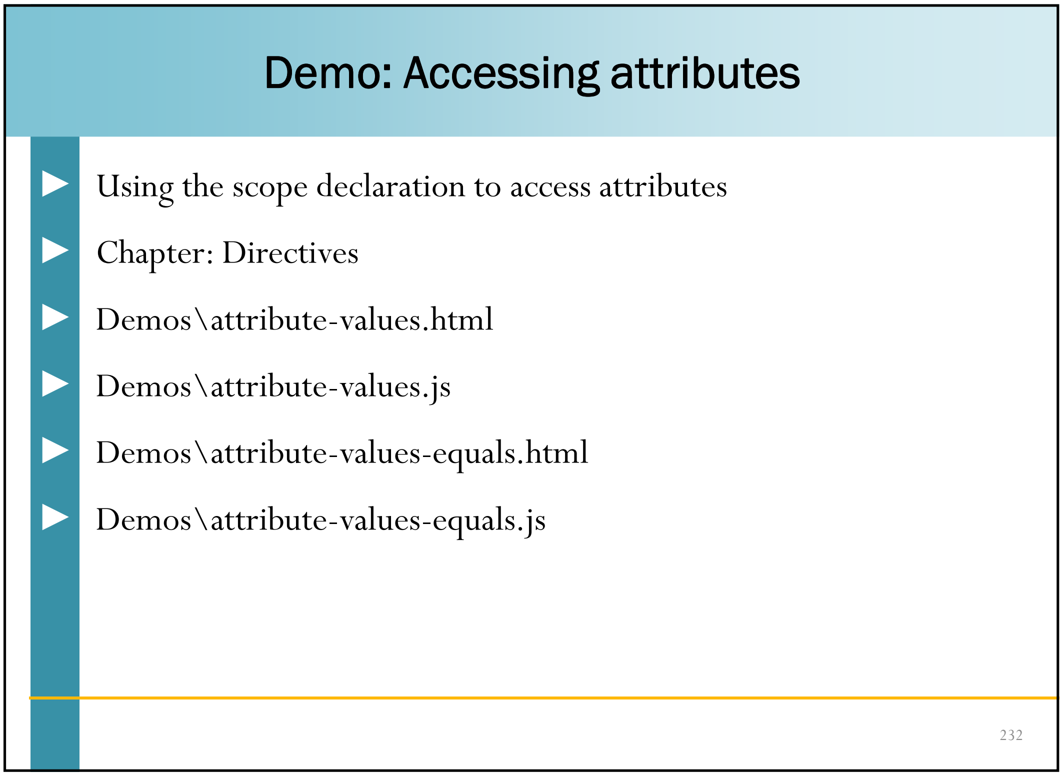
230



Directive scope: attributes

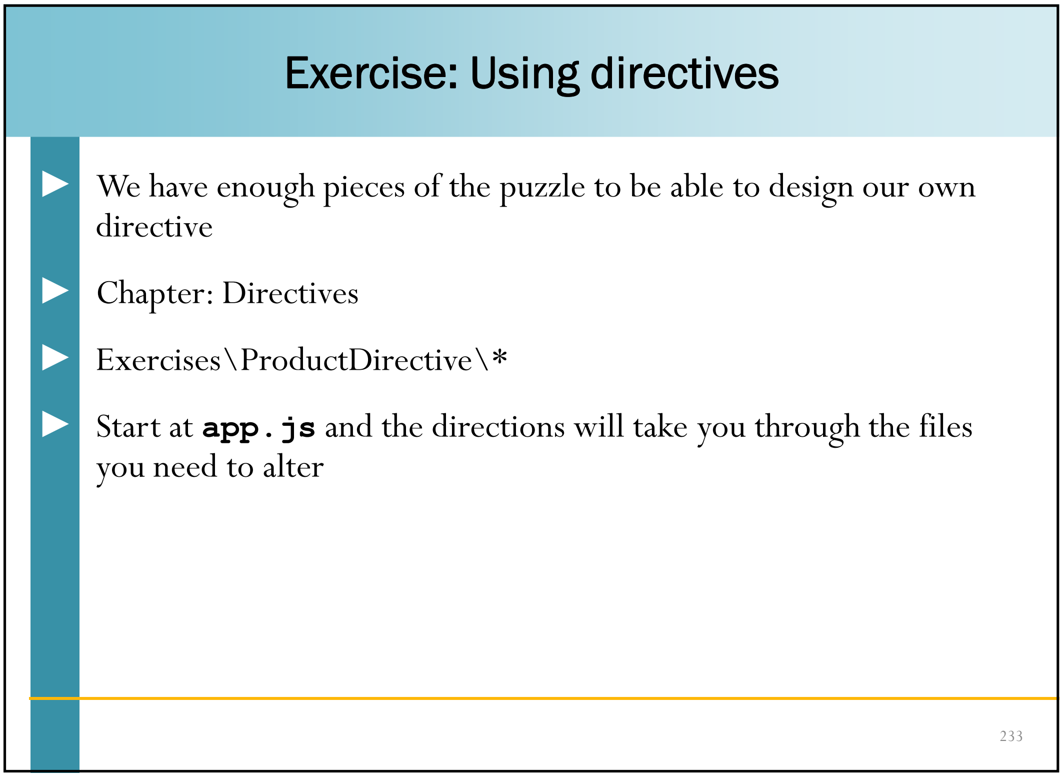
- Using @ to access attribute data

```
var mod = angular.module(...);
mod.directive('myAttr', function() {
  return {
    restrict: 'A', // Acting as an attribute
    scope : {
      attrValue : '@myAttr', // Value of attribute
                             // available as attrValue
      myAttr : '@' // Looks for the attribute with the
                   // same name as the key
    }
  }
});
```



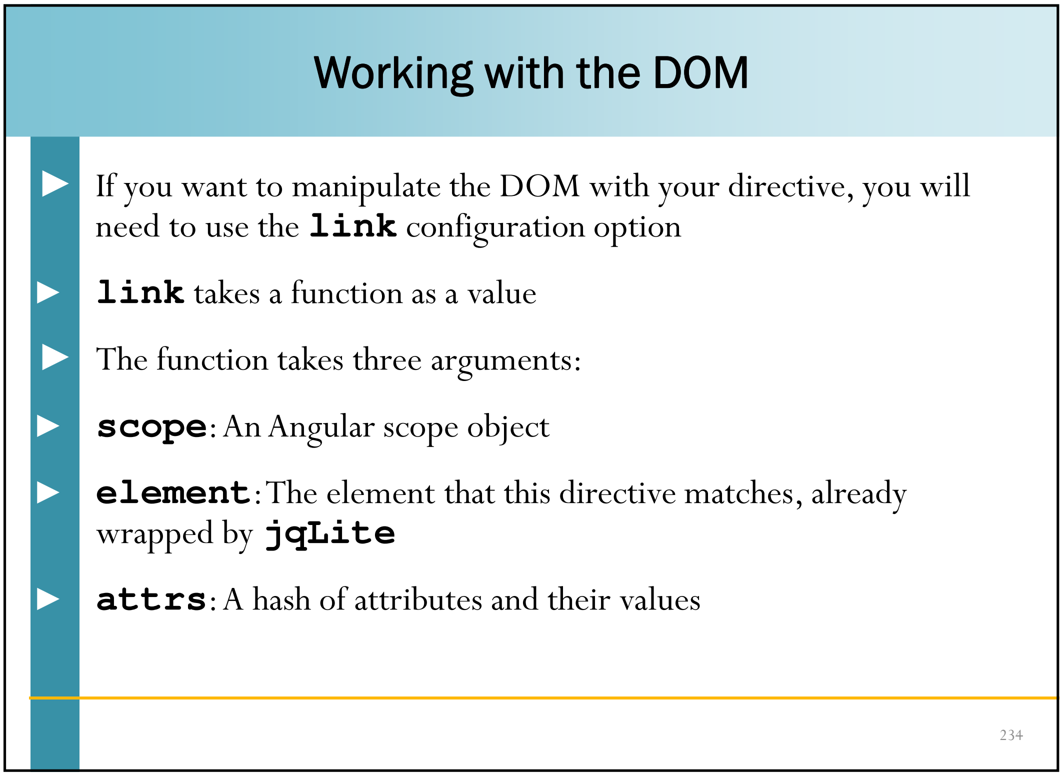
Demo: Accessing attributes

- Using the scope declaration to access attributes
- Chapter: Directives
- Demos\attribute-values.html
- Demos\attribute-values.js
- Demos\attribute-values-equals.html
- Demos\attribute-values-equals.js



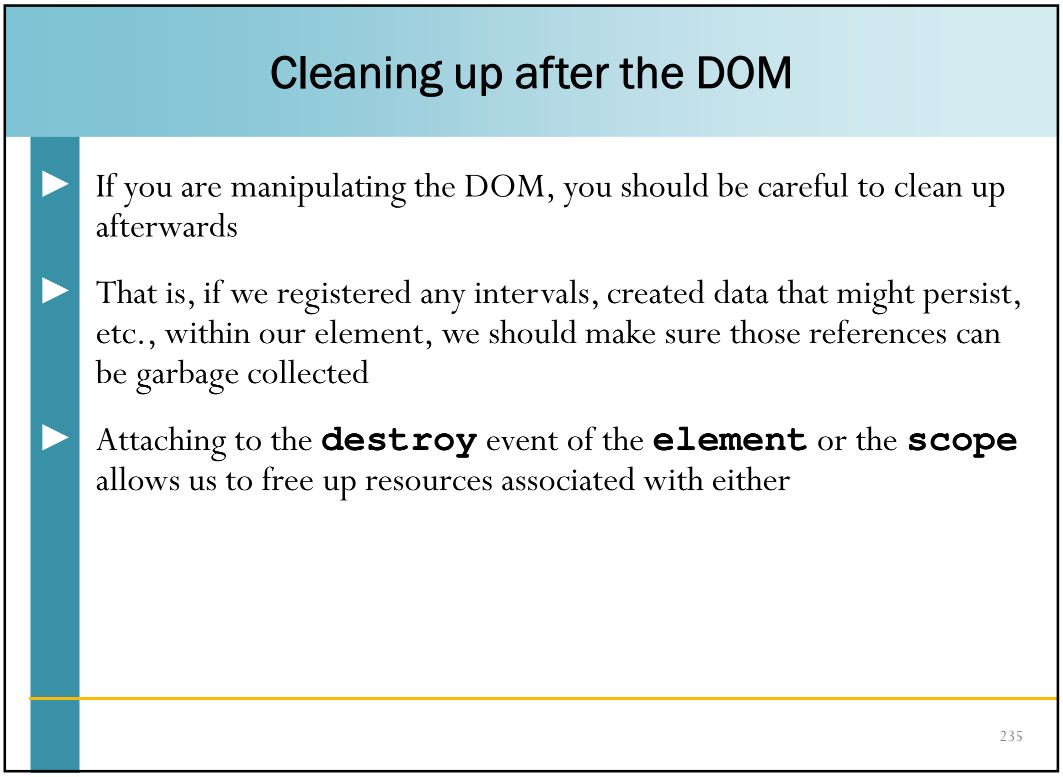
Exercise: Using directives

- We have enough pieces of the puzzle to be able to design our own directive
- Chapter: Directives
- Exercises\ProductDirective\*
- Start at **`app.js`** and the directions will take you through the files you need to alter



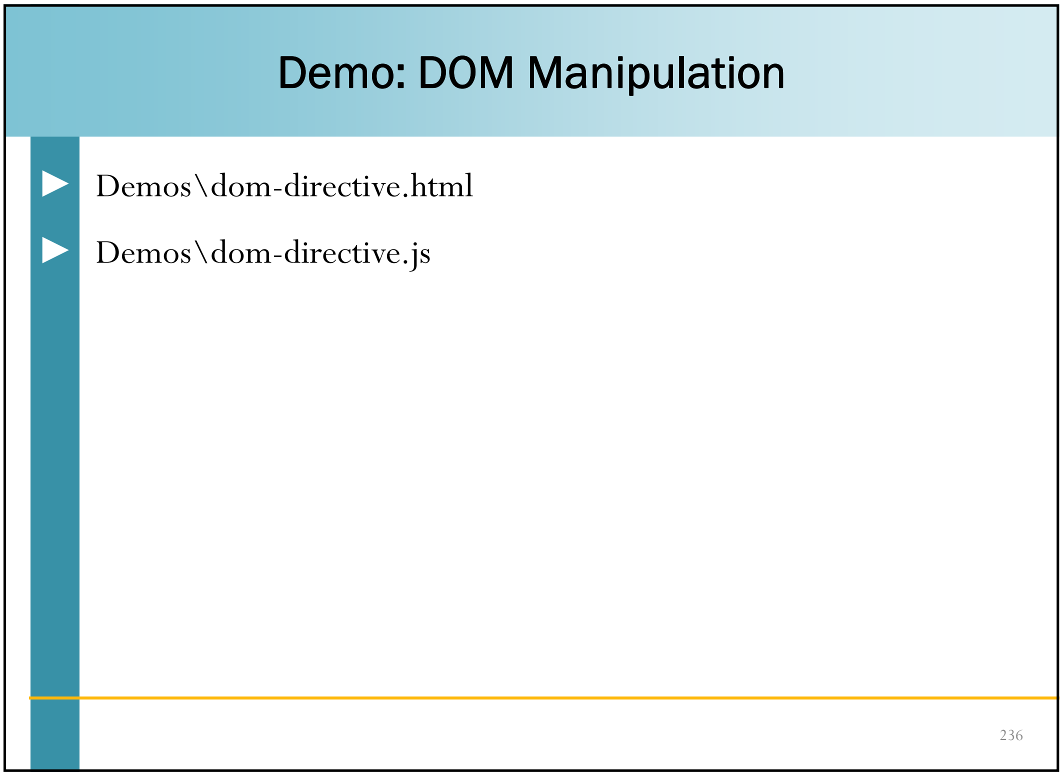
Working with the DOM

- If you want to manipulate the DOM with your directive, you will need to use the **`link`** configuration option
- **`link`** takes a function as a value
- The function takes three arguments:
- **`scope`**: An Angular scope object
- **`element`**: The element that this directive matches, already wrapped by **`jqLite`**
- **`attrs`**: A hash of attributes and their values



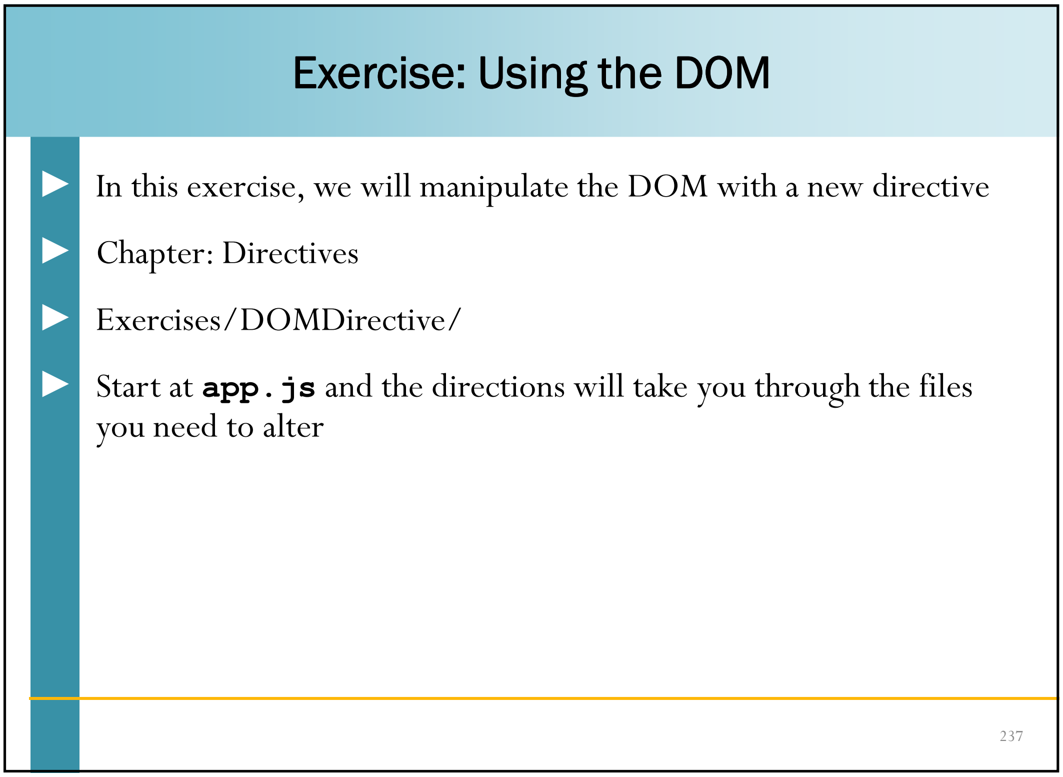
Cleaning up after the DOM

- If you are manipulating the DOM, you should be careful to clean up afterwards
- That is, if we registered any intervals, created data that might persist, etc., within our element, we should make sure those references can be garbage collected
- Attaching to the **`destroy`** event of the **`element`** or the **`scope`** allows us to free up resources associated with either



Demo: DOM Manipulation

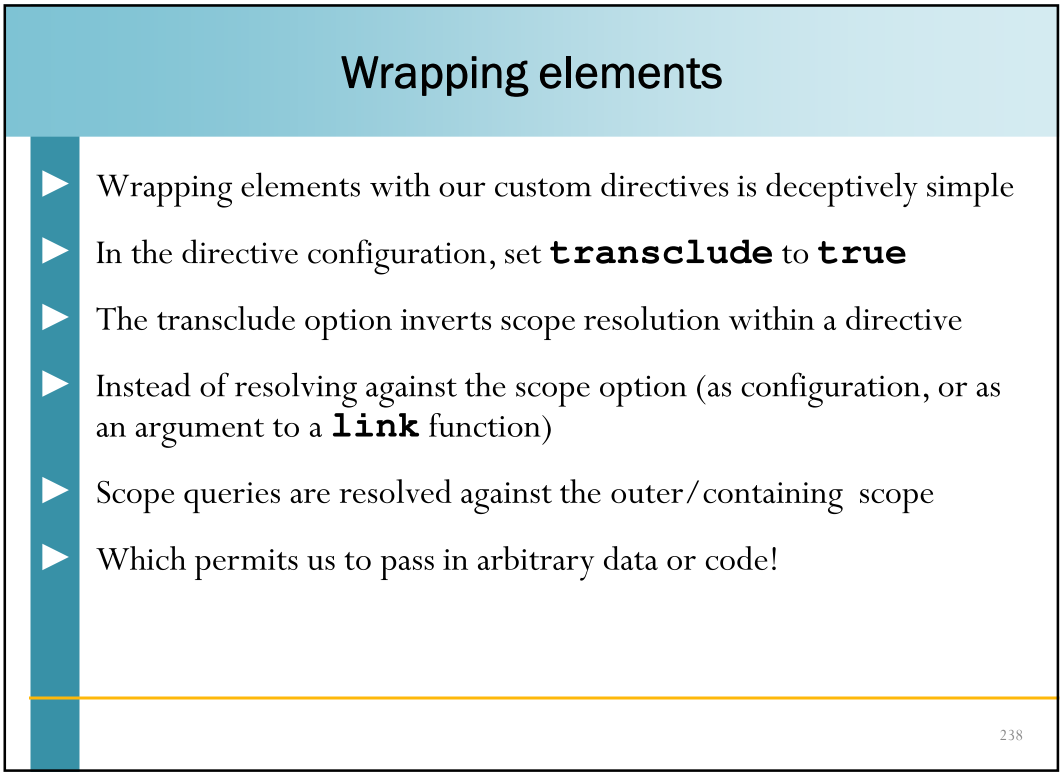
- Demos\dom-directive.html
- Demos\dom-directive.js



Exercise: Using the DOM

- In this exercise, we will manipulate the DOM with a new directive
- Chapter: Directives
- Exercises/DOMDirective/
- Start at **`app.js`** and the directions will take you through the files you need to alter

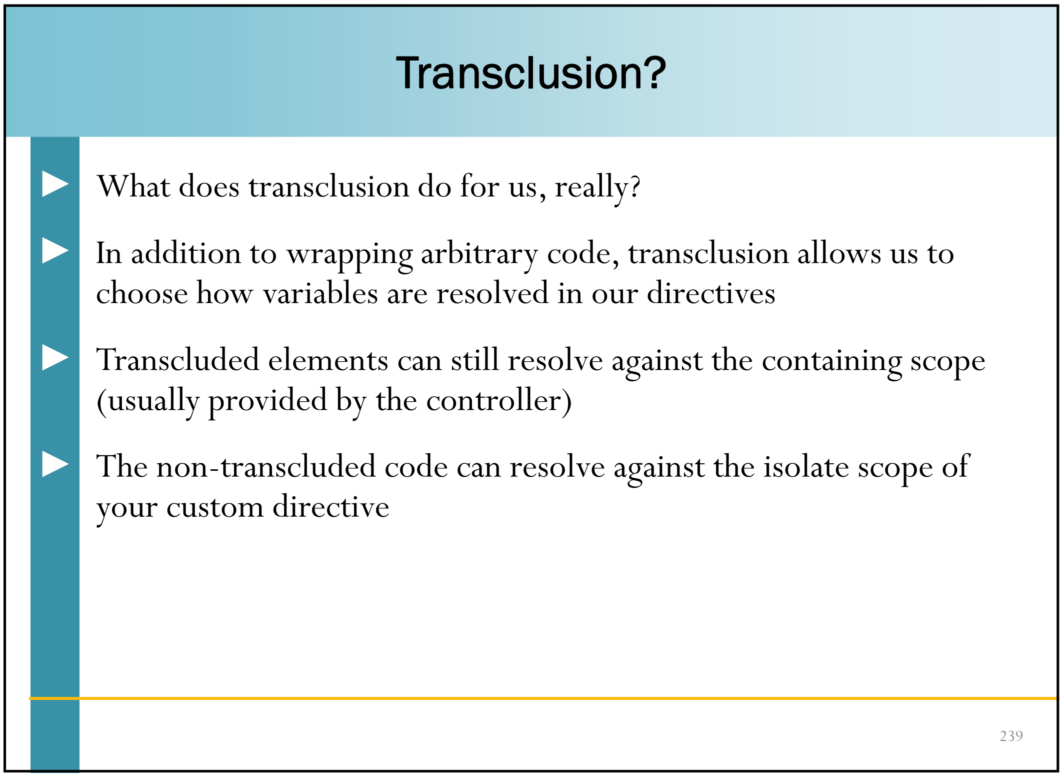
237



Wrapping elements

- Wrapping elements with our custom directives is deceptively simple
- In the directive configuration, set **`transclude`** to **`true`**
- The transclude option inverts scope resolution within a directive
- Instead of resolving against the scope option (as configuration, or as an argument to a **`link`** function)
- Scope queries are resolved against the outer/containing scope
- Which permits us to pass in arbitrary data or code!

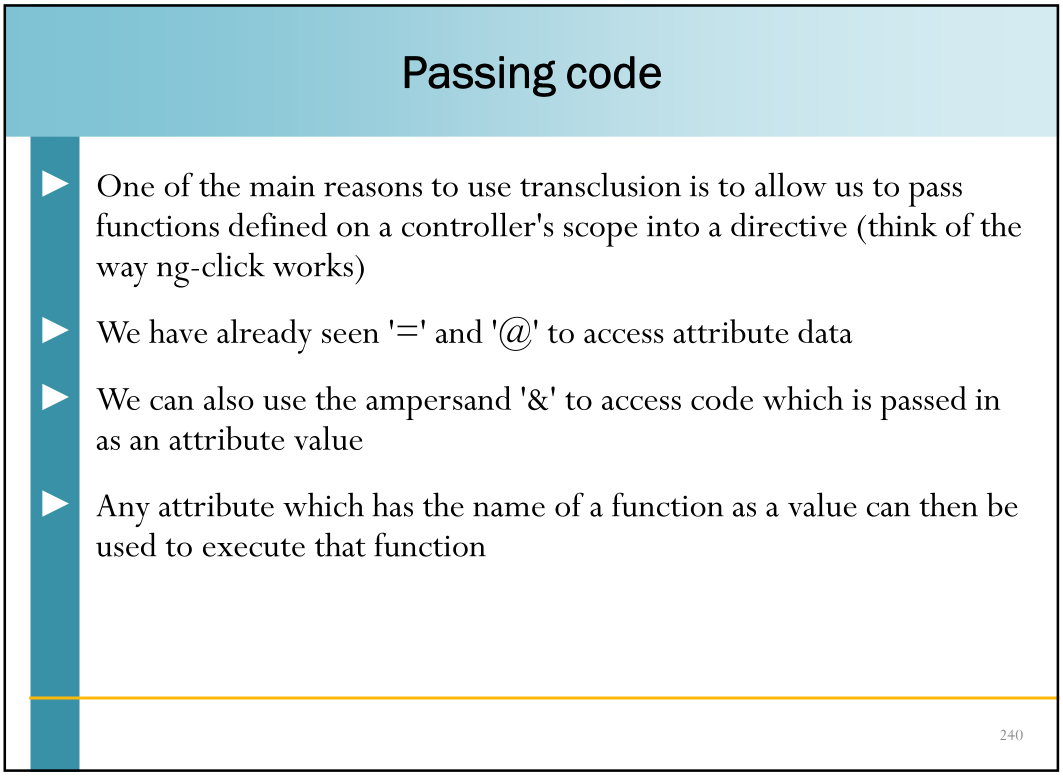
238



Transclusion?

- What does transclusion do for us, really?
- In addition to wrapping arbitrary code, transclusion allows us to choose how variables are resolved in our directives
- Transcluded elements can still resolve against the containing scope (usually provided by the controller)
- The non-transcluded code can resolve against the isolate scope of your custom directive

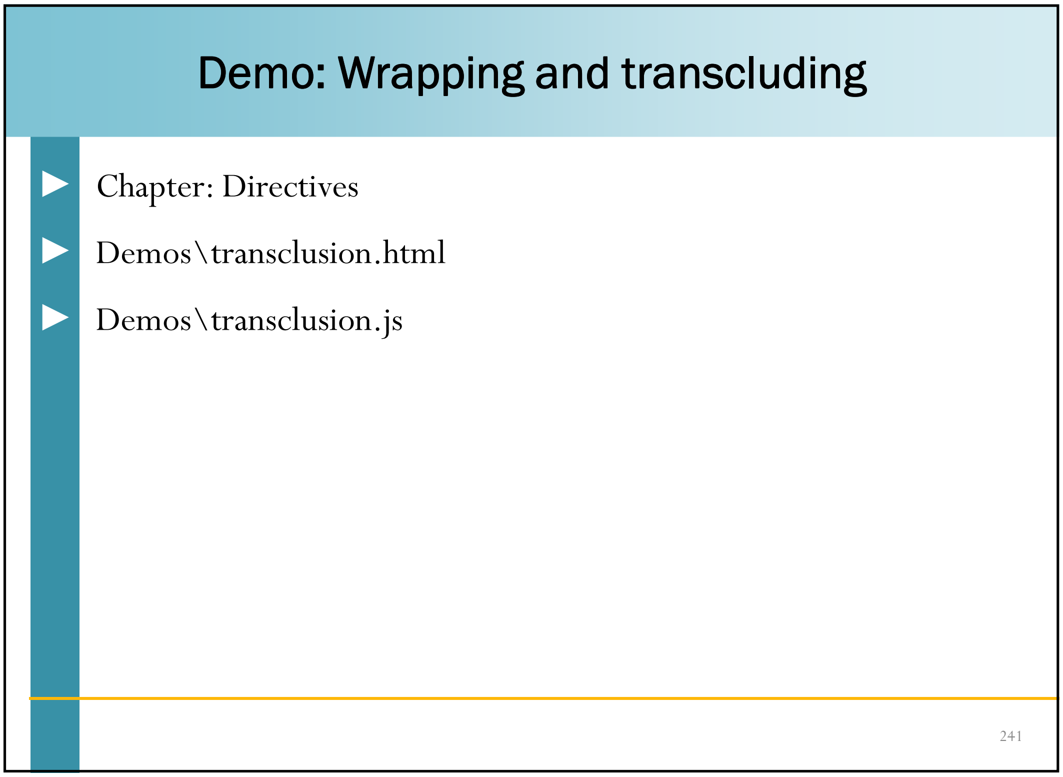
239



Passing code

- One of the main reasons to use transclusion is to allow us to pass functions defined on a controller's scope into a directive (think of the way ng-click works)
- We have already seen '=' and '@' to access attribute data
- We can also use the ampersand '&' to access code which is passed in as an attribute value
- Any attribute which has the name of a function as a value can then be used to execute that function

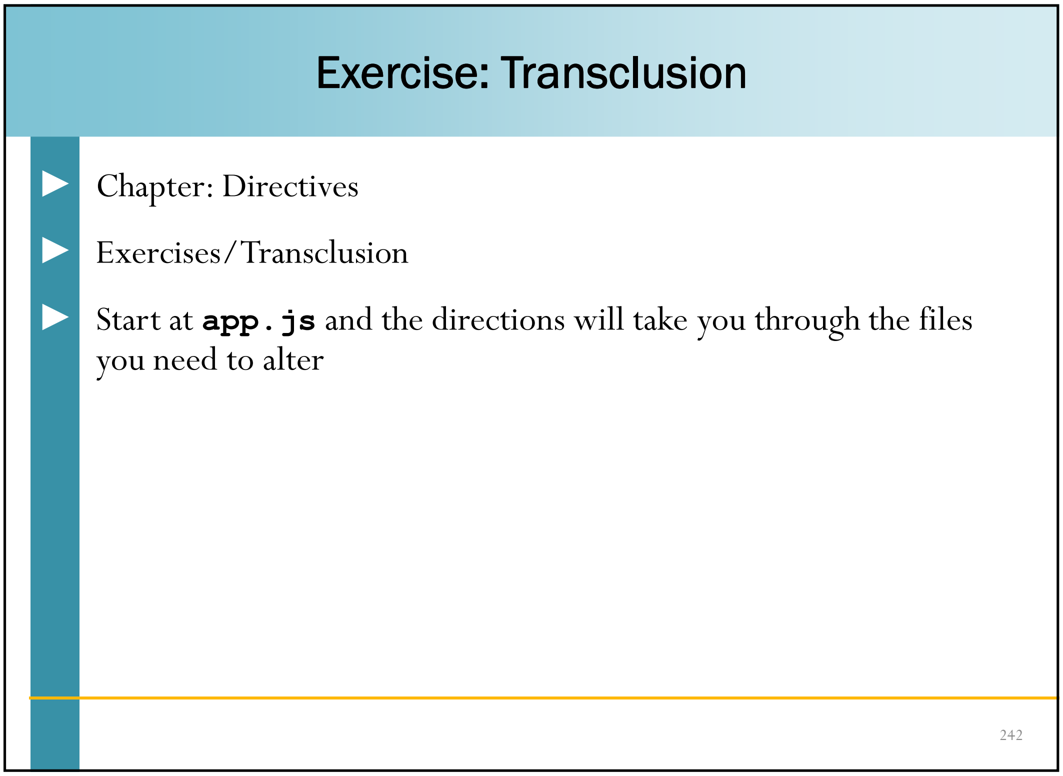
240



Demo: Wrapping and transcluding

- Chapter: Directives
- Demos\transclusion.html
- Demos\transclusion.js

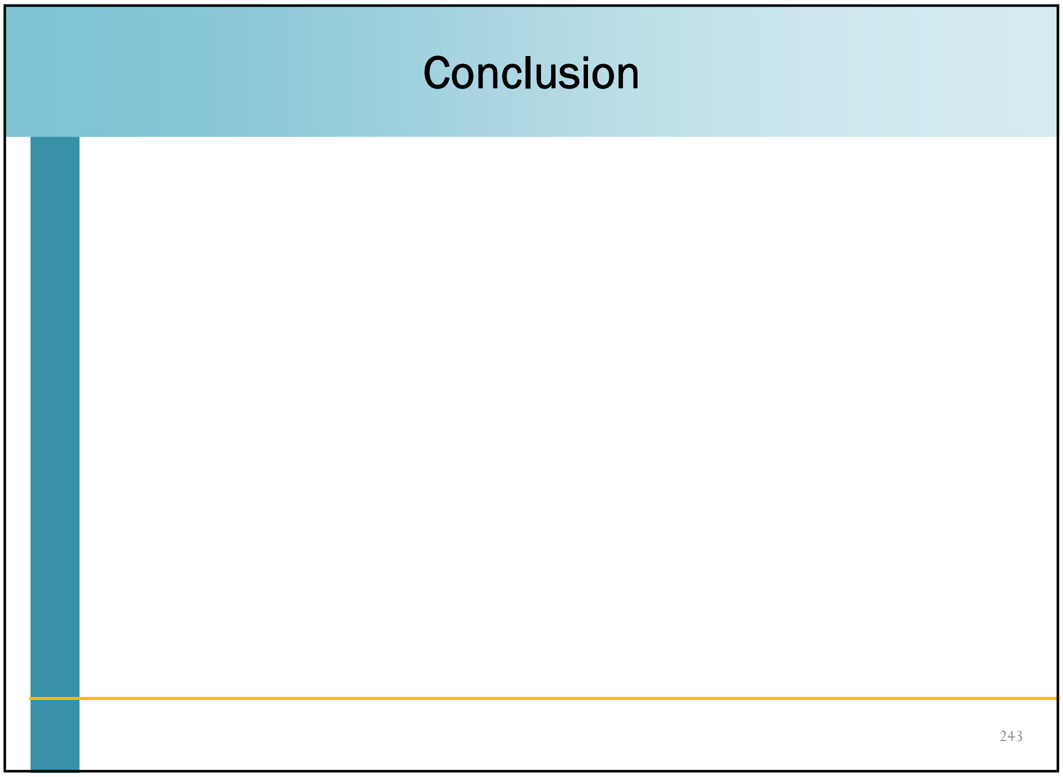
241



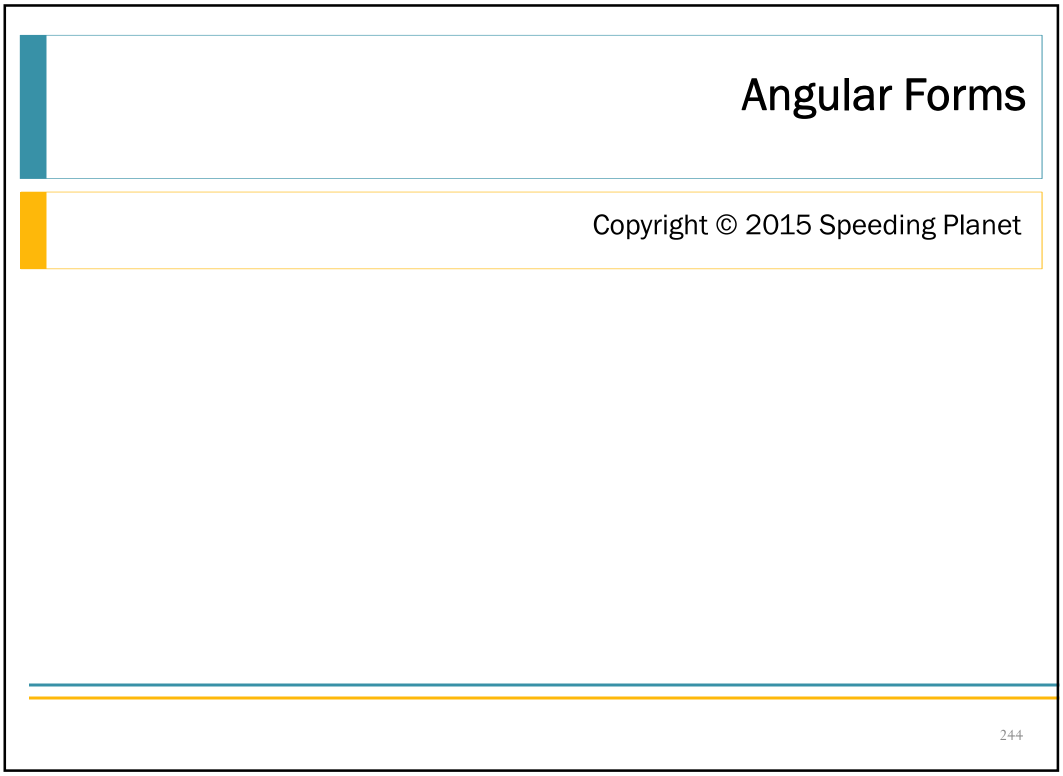
Exercise: Transclusion

- Chapter: Directives
- Exercises/Transclusion
- Start at **`app.js`** and the directions will take you through the files you need to alter

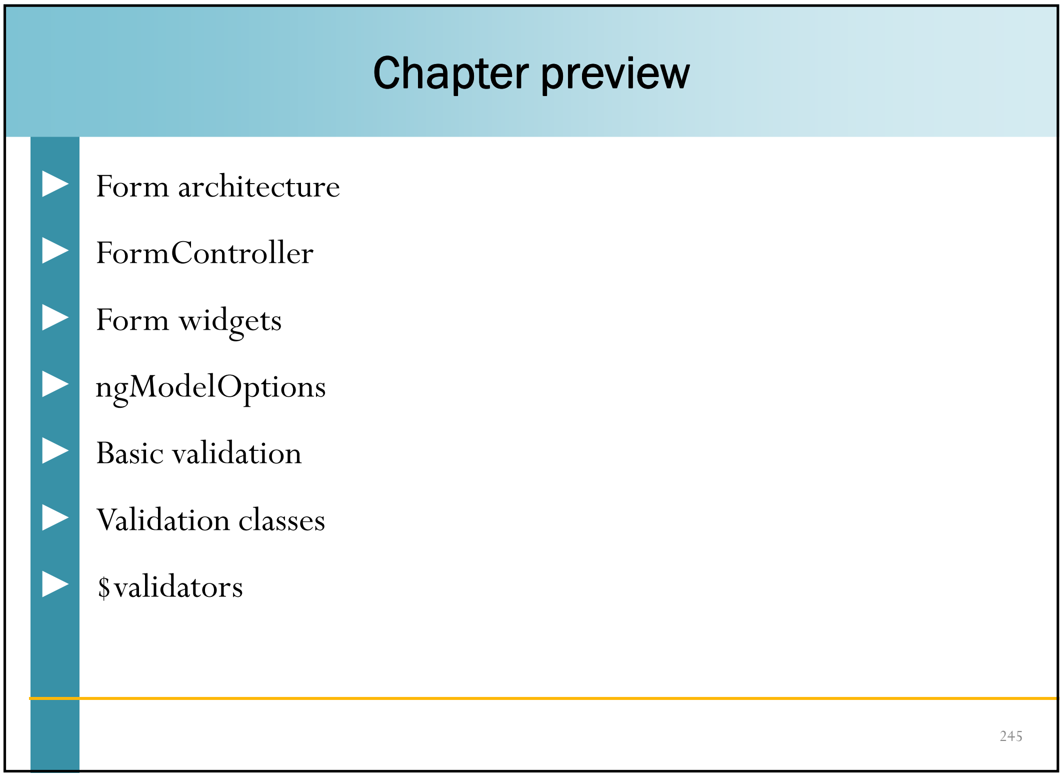
242



Conclusion

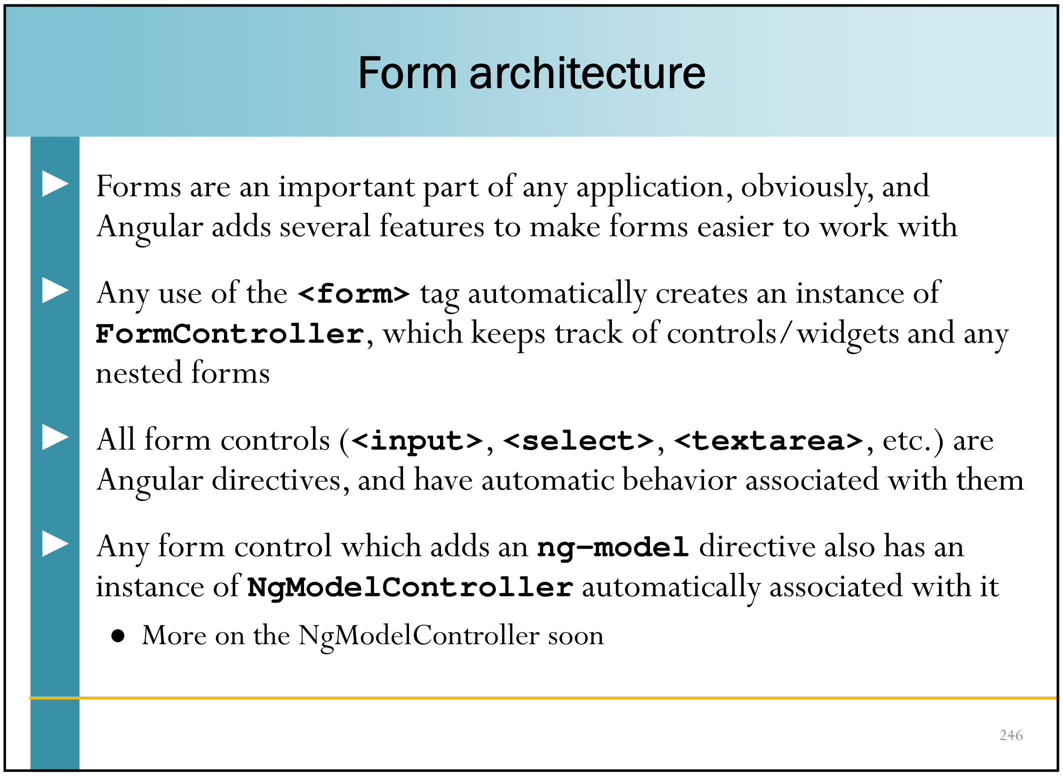


Angular Forms



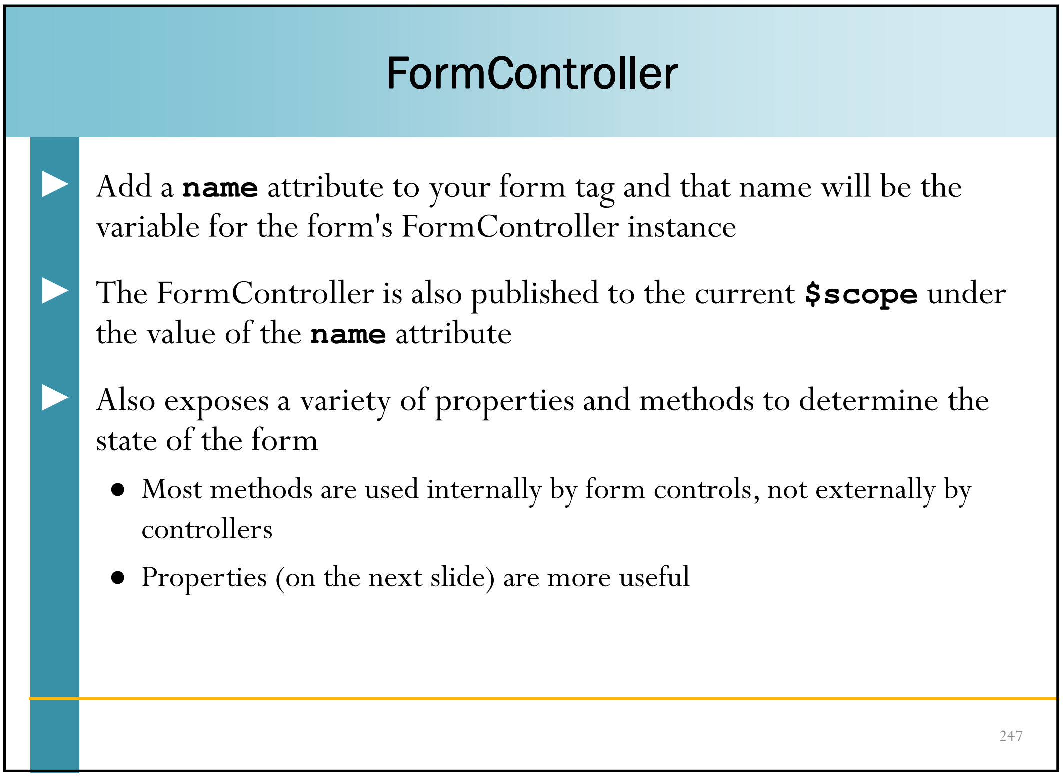
Chapter preview

- Form architecture
- FormController
- Form widgets
- ngModelOptions
- Basic validation
- Validation classes
- $validators



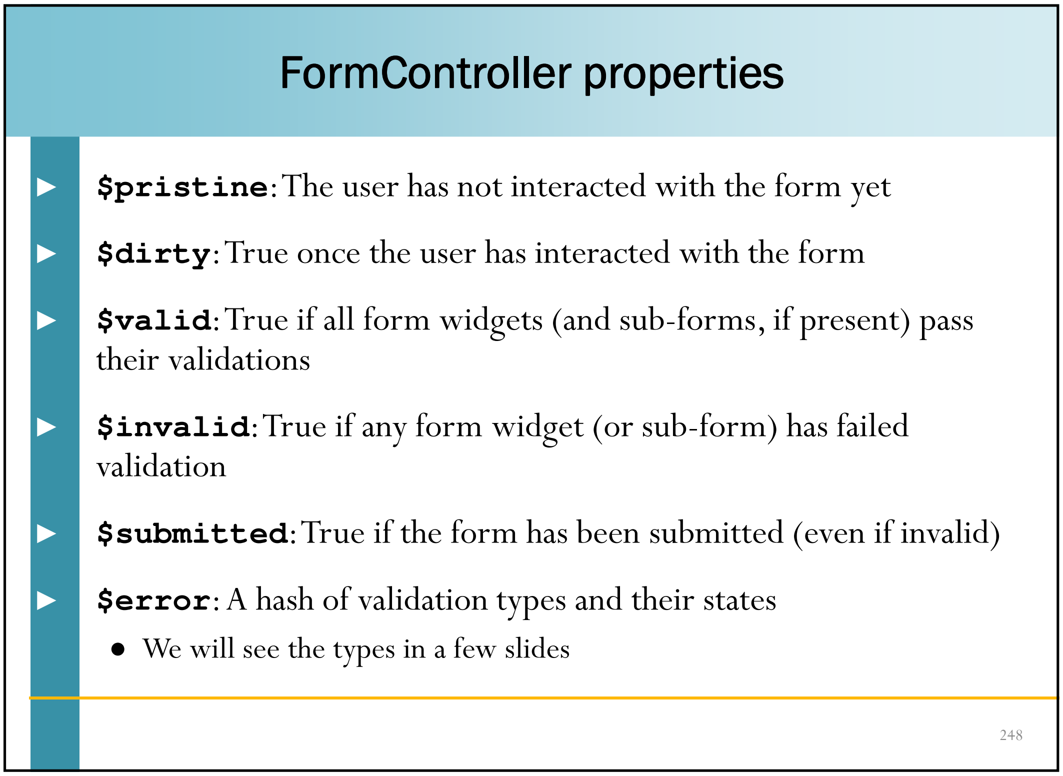
Form architecture

- Forms are an important part of any application, obviously, and Angular adds several features to make forms easier to work with
- Any use of the **`<form>`** tag automatically creates an instance of **`FormController`**, which keeps track of controls/widgets and any nested forms
- All form controls (**`<input>`**, **`<select>`**, **`<textarea>`**, etc.) are Angular directives, and have automatic behavior associated with them
- Any form control which adds an **`ng-model`** directive also has an instance of **`NgModelController`** automatically associated with it
  - More on the NgModelController soon



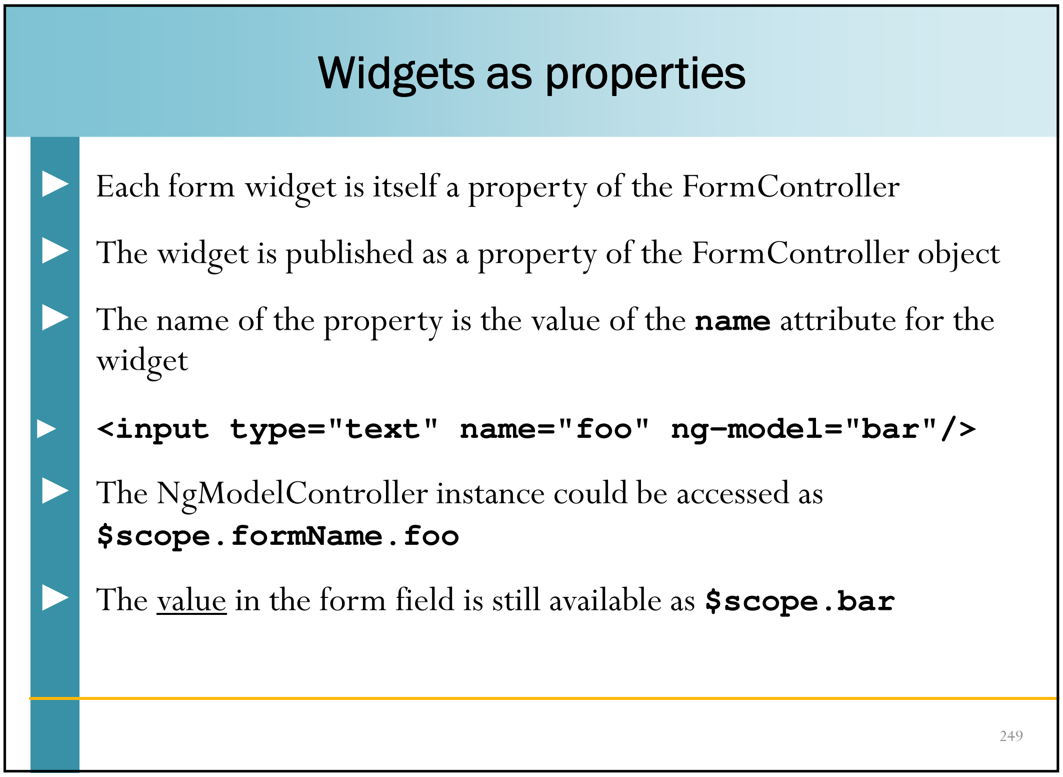
FormController

- Add a **`name`** attribute to your form tag and that name will be the variable for the form's FormController instance
- The FormController is also published to the current **`$scope`** under the value of the **`name`** attribute
- Also exposes a variety of properties and methods to determine the state of the form
  - Most methods are used internally by form controls, not externally by controllers
  - Properties (on the next slide) are more useful



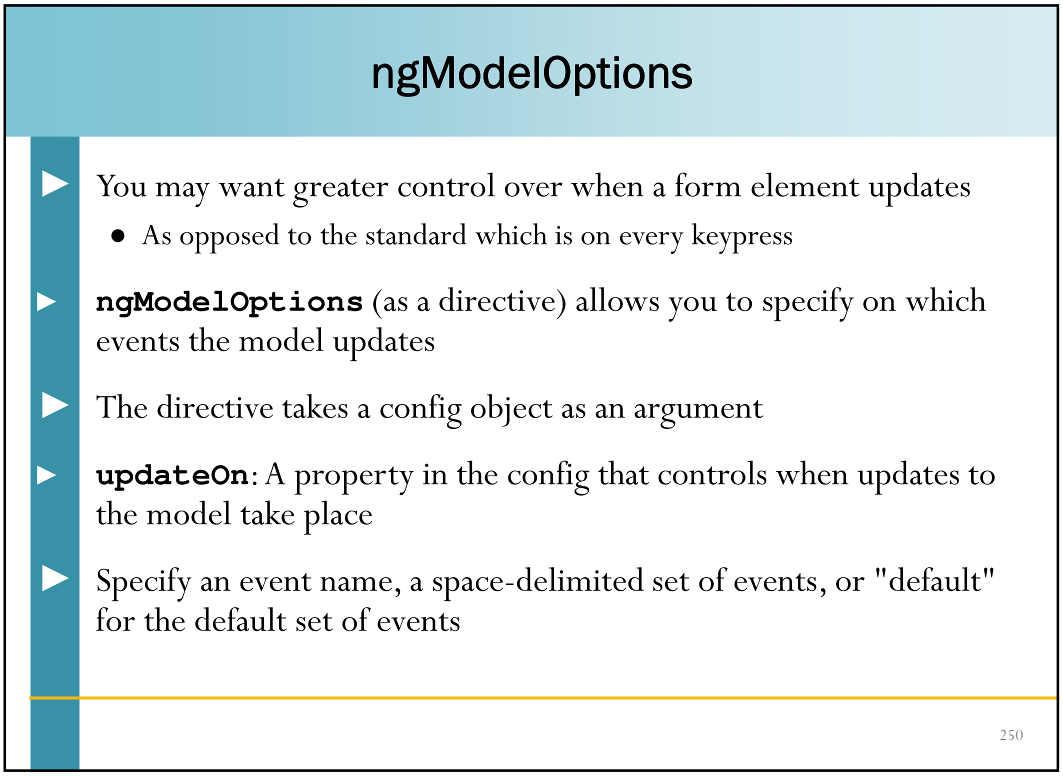
FormController properties

- **`$pristine`**: The user has not interacted with the form yet
- **`$dirty`**: True once the user has interacted with the form
- **`$valid`**: True if all form widgets (and sub-forms, if present) pass their validations
- **`$invalid`**: True if any form widget (or sub-form) has failed validation
- **`$submitted`**: True if the form has been submitted (even if invalid)
- **`$error`**: A hash of validation types and their states
  - We will see the types in a few slides



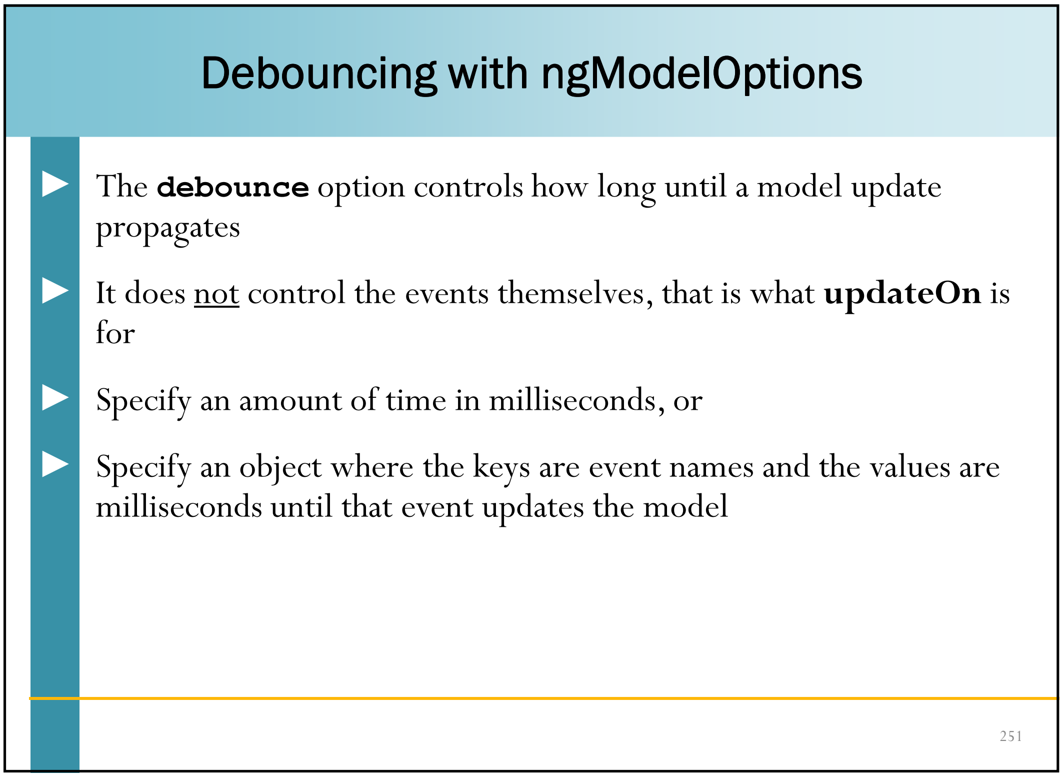
Widgets as properties

- Each form widget is itself a property of the FormController
- The widget is published as a property of the FormController object
- The name of the property is the value of the **`name`** attribute for the widget
- **`<input type="text" name="foo" ng-model="bar"/>`**
- The NgModelController instance could be accessed as **`$scope.formName.foo`**
- The value in the form field is still available as **`$scope.bar`**



ngModelOptions

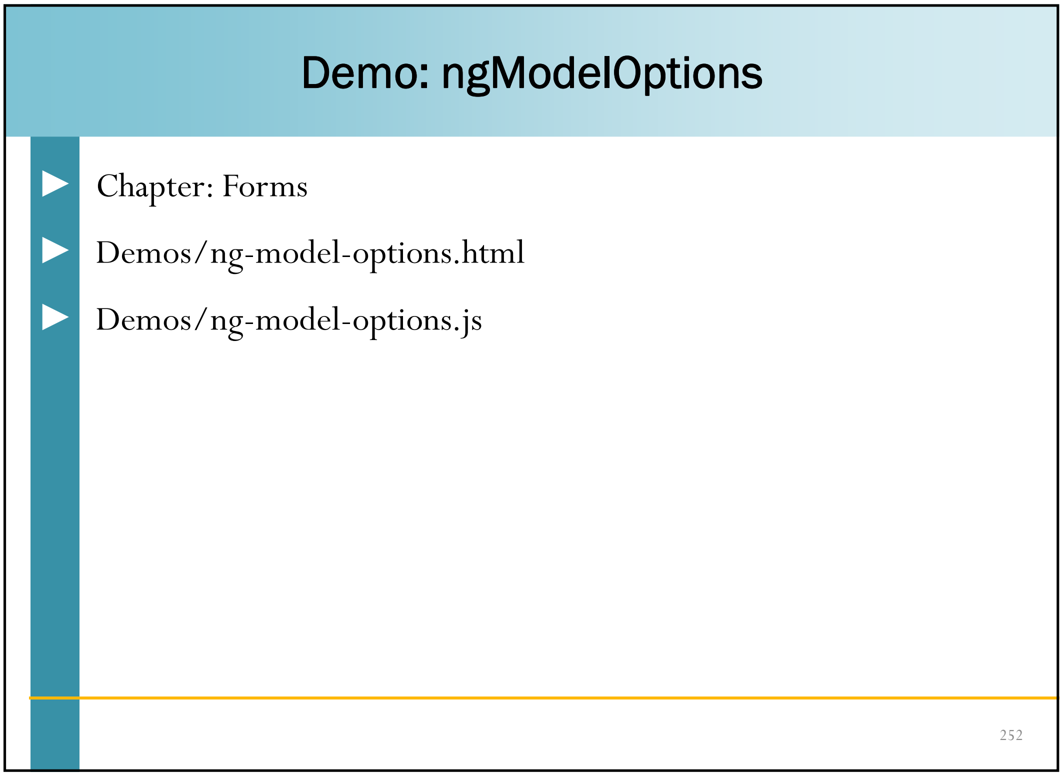
- You may want greater control over when a form element updates
  - As opposed to the standard which is on every keypress
- **`ngModelOptions`** (as a directive) allows you to specify on which events the model updates
- The directive takes a config object as an argument
- **`updateOn`**: A property in the config that controls when updates to the model take place
- Specify an event name, a space-delimited set of events, or "default" for the default set of events



Debouncing with ngModelOptions

- The **`debounce`** option controls how long until a model update propagates
- It does <u>not</u> control the events themselves, that is what **updateOn** is for
- Specify an amount of time in milliseconds, or
- Specify an object where the keys are event names and the values are milliseconds until that event updates the model

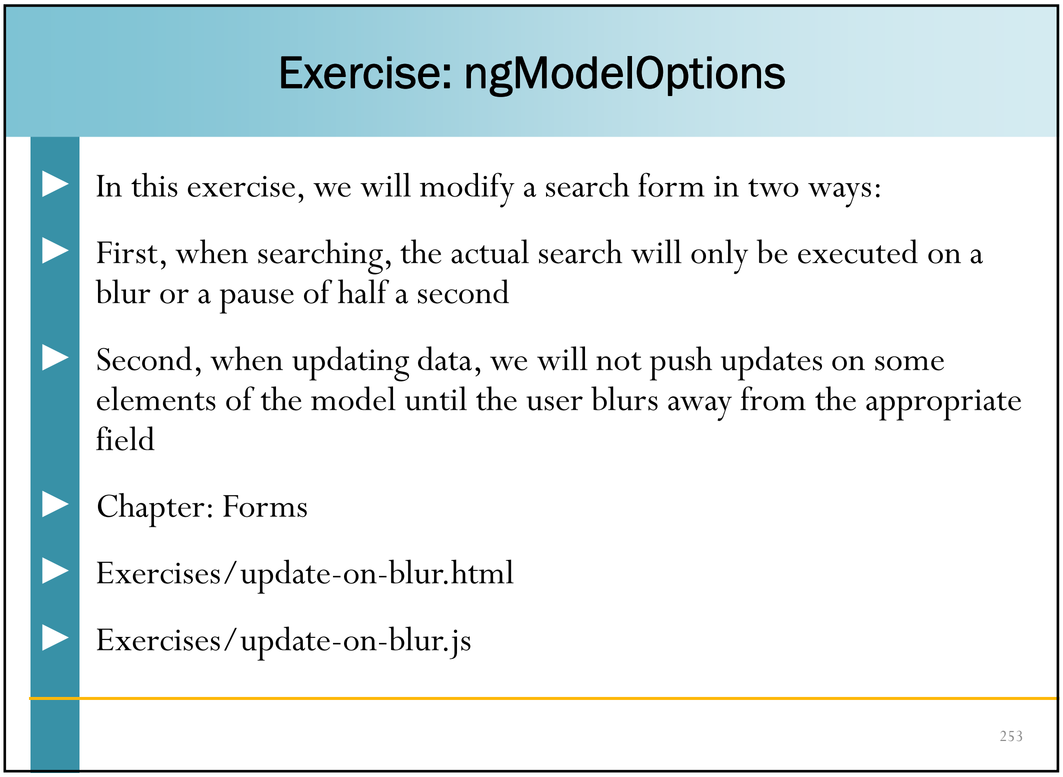
251



Demo: ngModelOptions

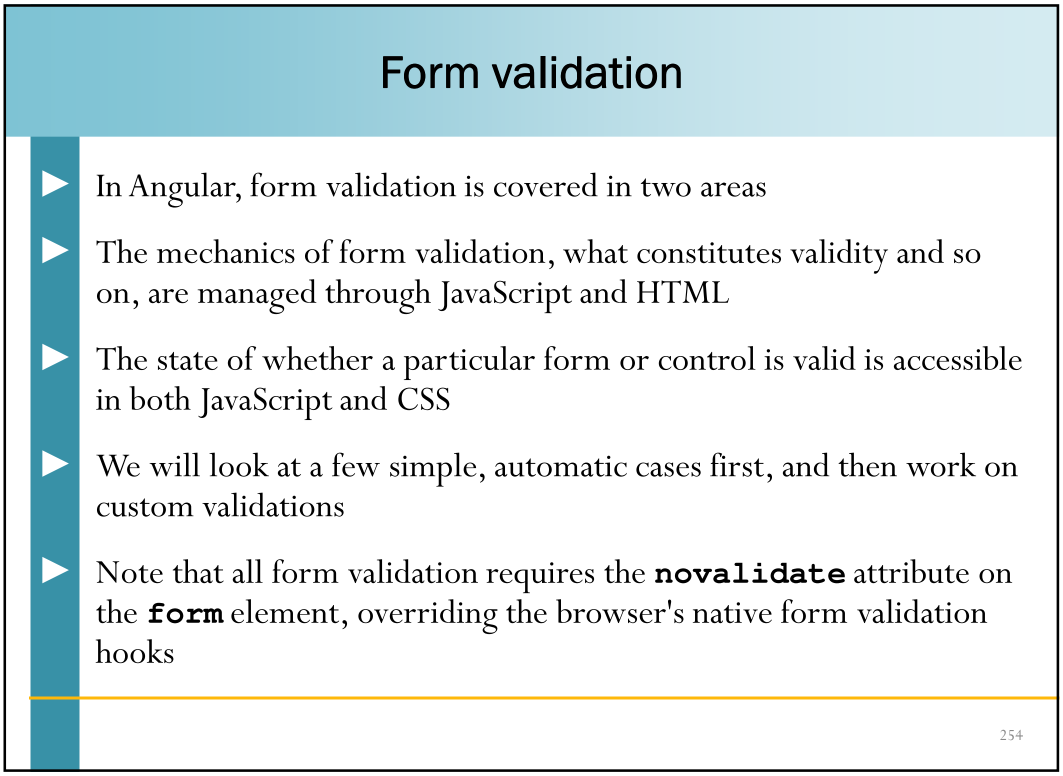
- Chapter: Forms
- Demos/ng-model-options.html
- Demos/ng-model-options.js

252



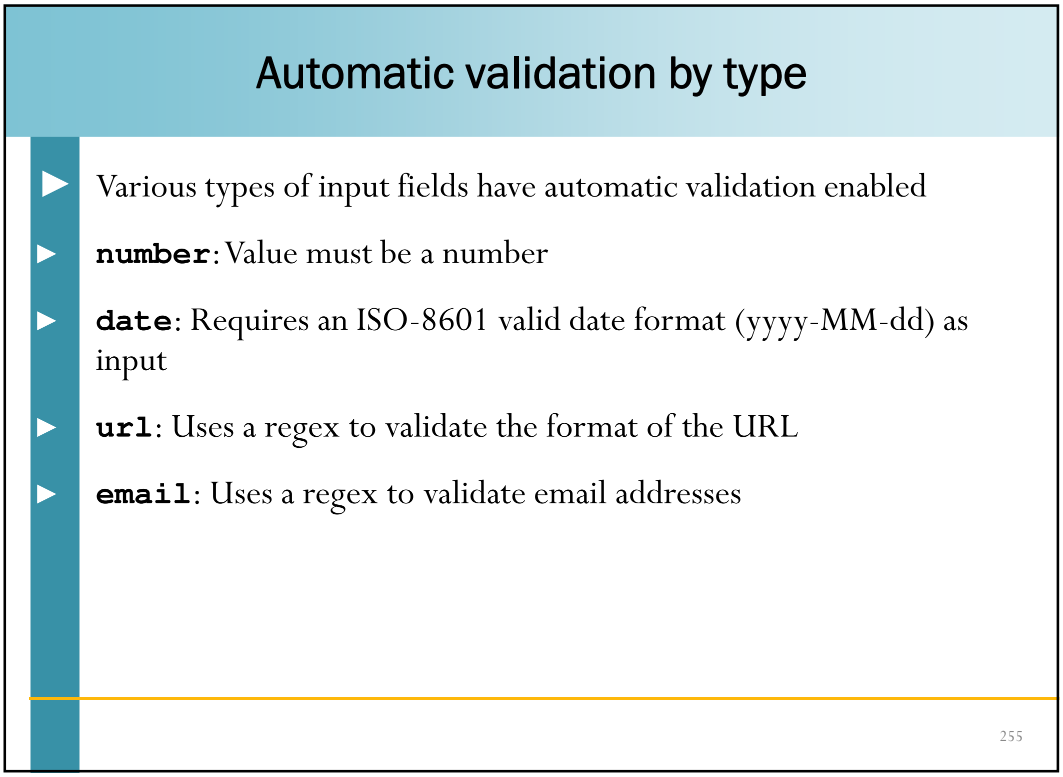
Exercise: ngModelOptions

- In this exercise, we will modify a search form in two ways:
- First, when searching, the actual search will only be executed on a blur or a pause of half a second
- Second, when updating data, we will not push updates on some elements of the model until the user blurs away from the appropriate field
- Chapter: Forms
- Exercises/update-on-blur.html
- Exercises/update-on-blur.js



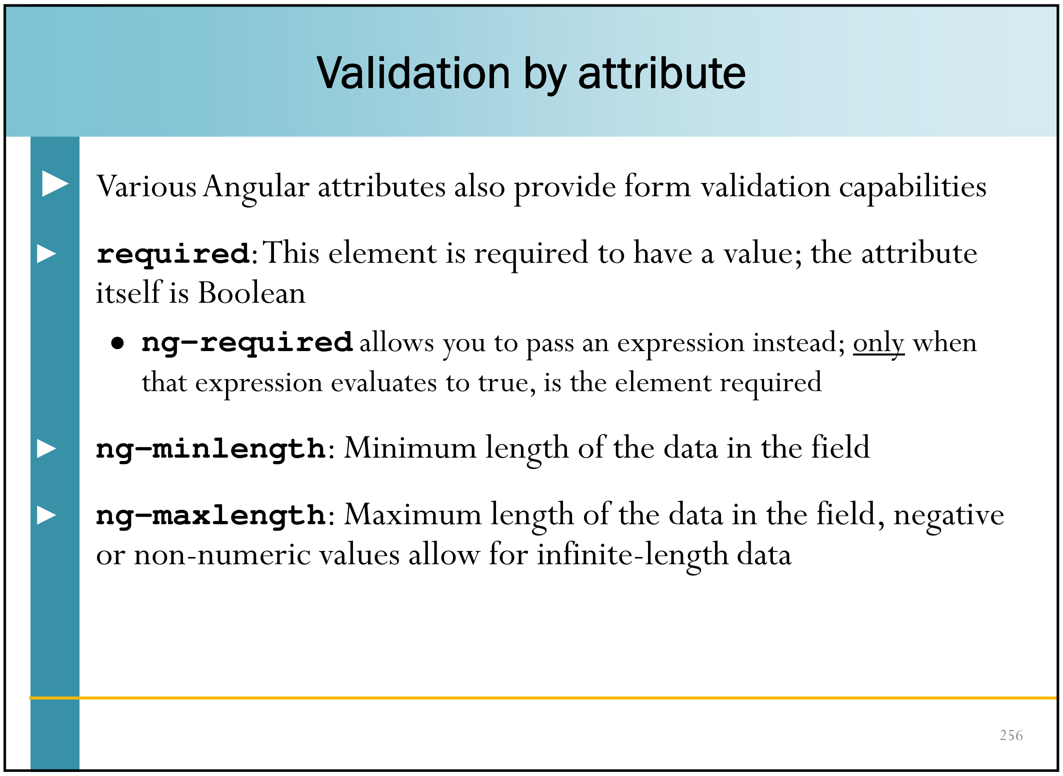
Form validation

- In Angular, form validation is covered in two areas
- The mechanics of form validation, what constitutes validity and so on, are managed through JavaScript and HTML
- The state of whether a particular form or control is valid is accessible in both JavaScript and CSS
- We will look at a few simple, automatic cases first, and then work on custom validations
- Note that all form validation requires the **`novalidate`** attribute on the **`form`** element, overriding the browser's native form validation hooks



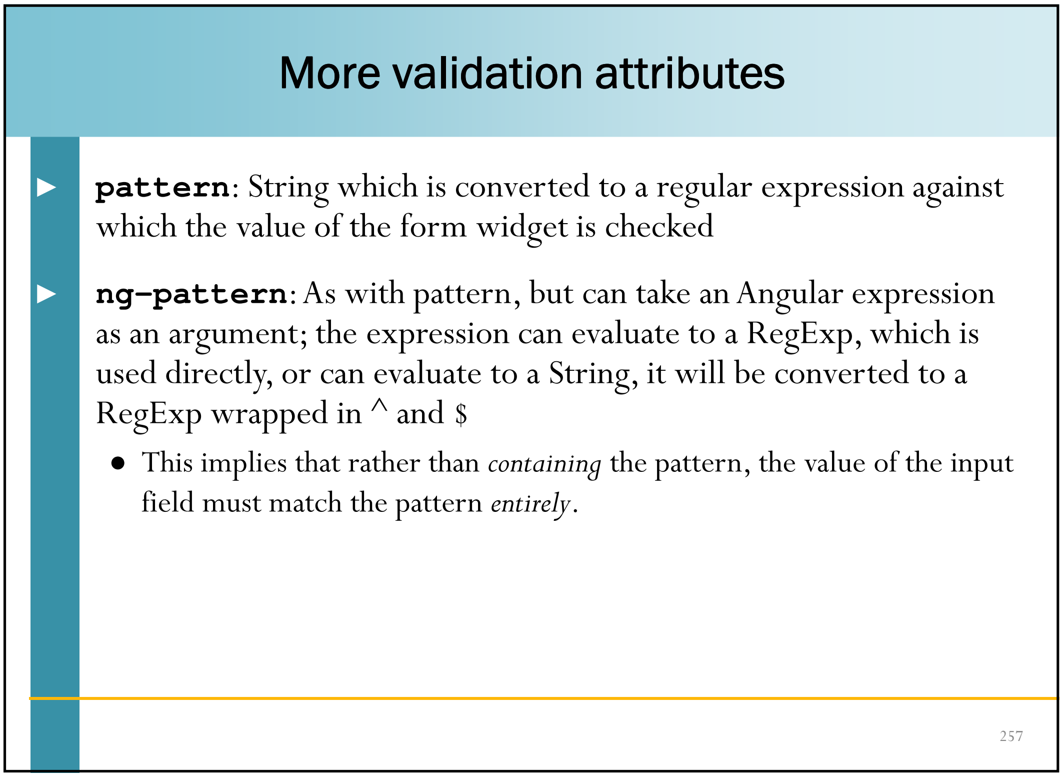
Automatic validation by type

- Various types of input fields have automatic validation enabled
- **`number`**: Value must be a number
- **`date`**: Requires an ISO-8601 valid date format (yyyy-MM-dd) as input
- **`url`**: Uses a regex to validate the format of the URL
- **`email`**: Uses a regex to validate email addresses



Validation by attribute

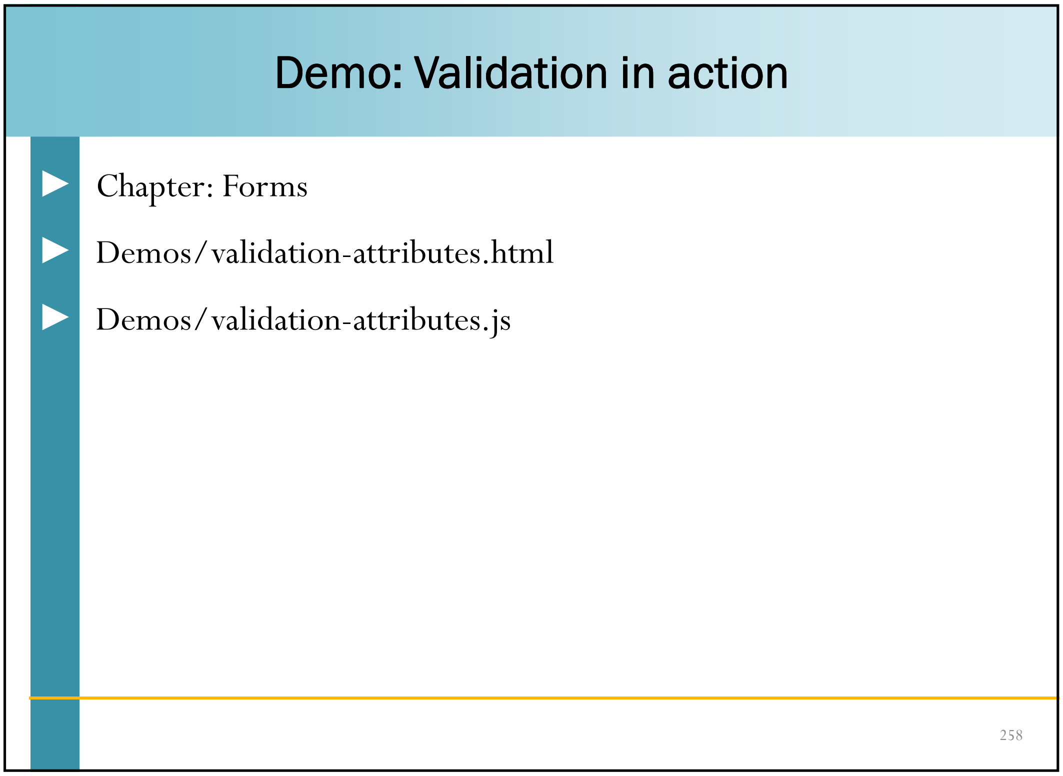
- Various Angular attributes also provide form validation capabilities
- **`required`**: This element is required to have a value; the attribute itself is Boolean
  - **`ng-required`** allows you to pass an expression instead; <u>only</u> when that expression evaluates to true, is the element required
- **`ng-minlength`**: Minimum length of the data in the field
- **`ng-maxlength`**: Maximum length of the data in the field, negative or non-numeric values allow for infinite-length data



More validation attributes

- **`pattern`**: String which is converted to a regular expression against which the value of the form widget is checked
- **`ng-pattern`**: As with pattern, but can take an Angular expression as an argument; the expression can evaluate to a RegExp, which is used directly, or can evaluate to a String, it will be converted to a RegExp wrapped in ^ and $
  - This implies that rather than *containing* the pattern, the value of the input field must match the pattern *entirely*.

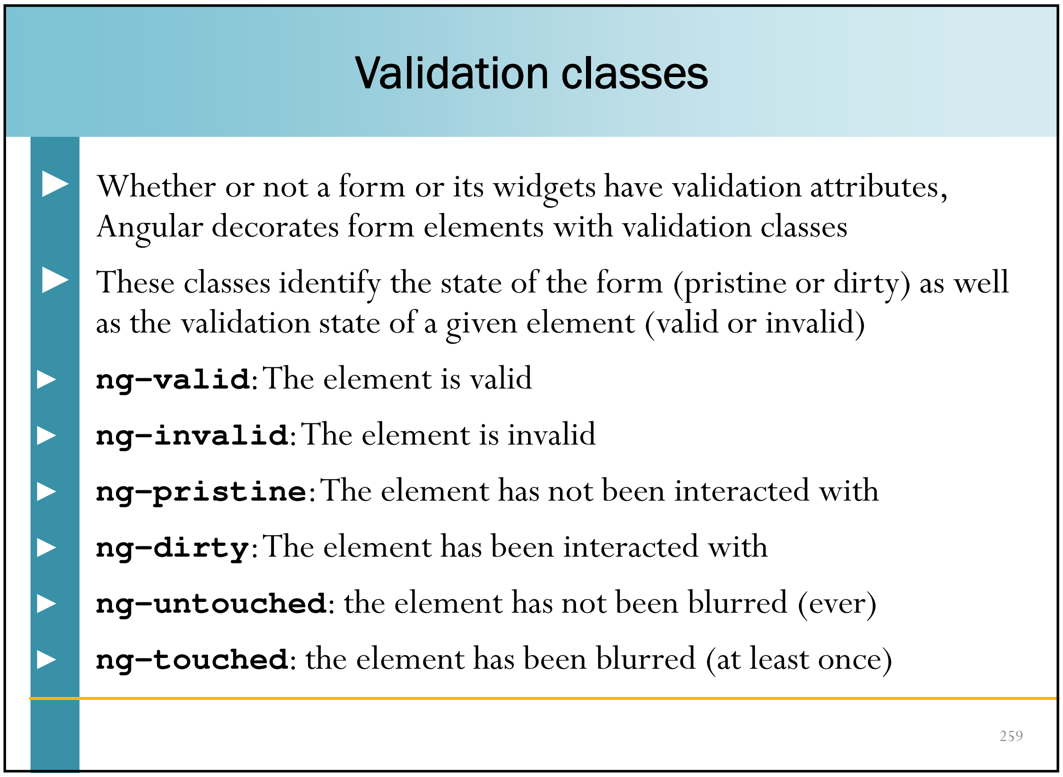
257



Demo: Validation in action

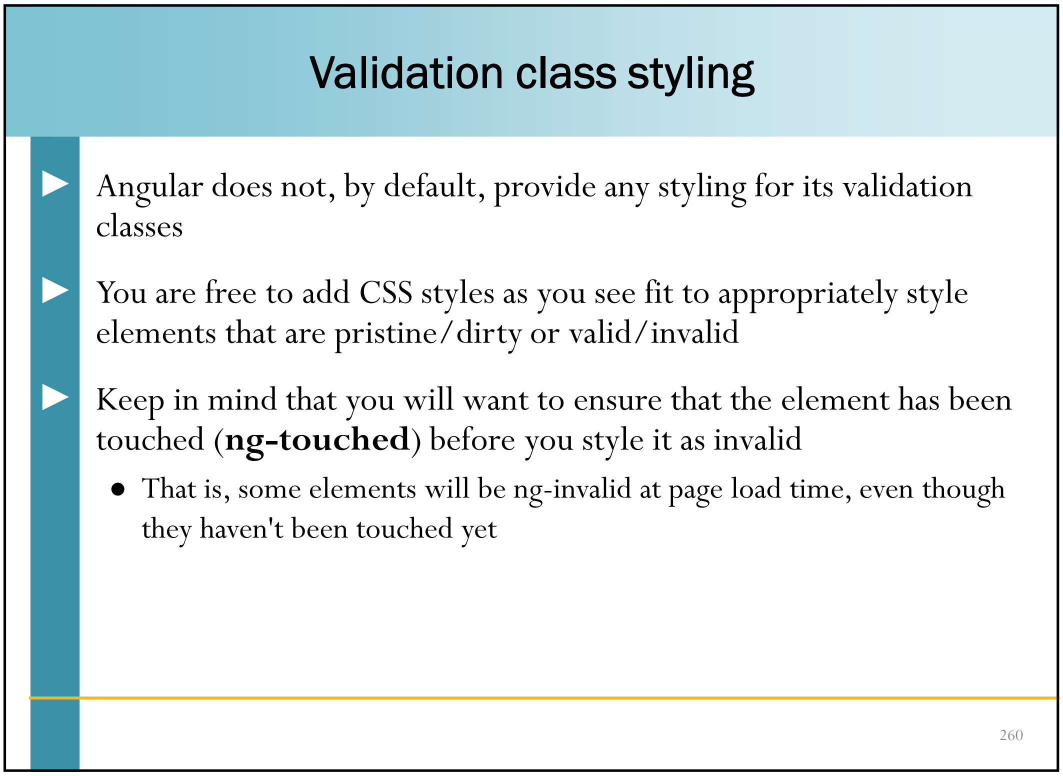
- Chapter: Forms
- Demos/validation-attributes.html
- Demos/validation-attributes.js

258



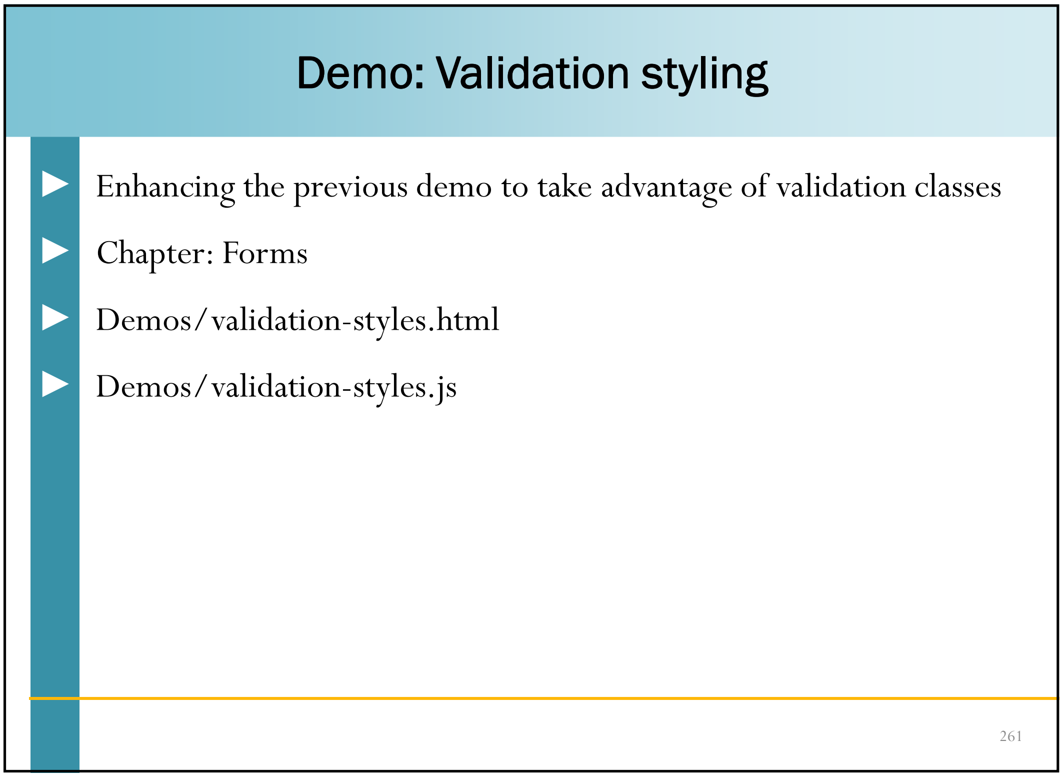
Validation classes

- Whether or not a form or its widgets have validation attributes, Angular decorates form elements with validation classes
- These classes identify the state of the form (pristine or dirty) as well as the validation state of a given element (valid or invalid)
- **ng-valid**:The element is valid
- **ng-invalid**:The element is invalid
- **ng-pristine**:The element has not been interacted with
- **ng-dirty**:The element has been interacted with
- **ng-untouched**: the element has not been blurred (ever)
- **ng-touched**: the element has been blurred (at least once)



Validation class styling

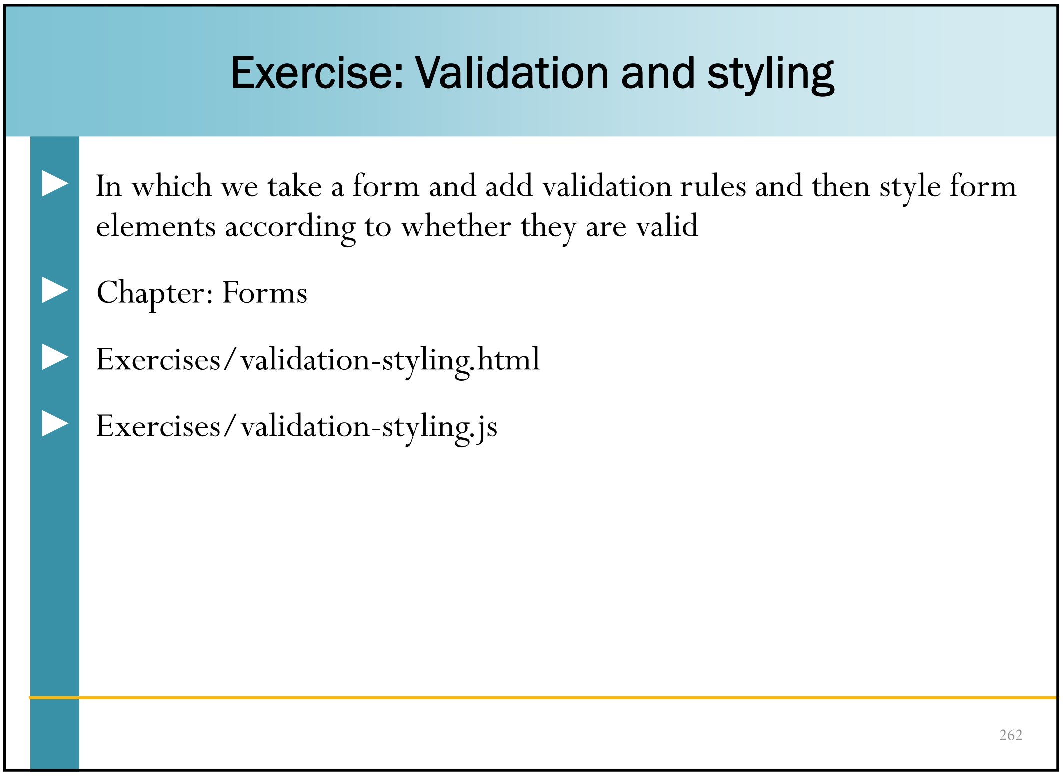
- Angular does not, by default, provide any styling for its validation classes
- You are free to add CSS styles as you see fit to appropriately style elements that are pristine/dirty or valid/invalid
- Keep in mind that you will want to ensure that the element has been touched (**ng-touched**) before you style it as invalid
  - That is, some elements will be ng-invalid at page load time, even though they haven't been touched yet



Demo: Validation styling

- Enhancing the previous demo to take advantage of validation classes
- Chapter: Forms
- Demos/validation-styles.html
- Demos/validation-styles.js

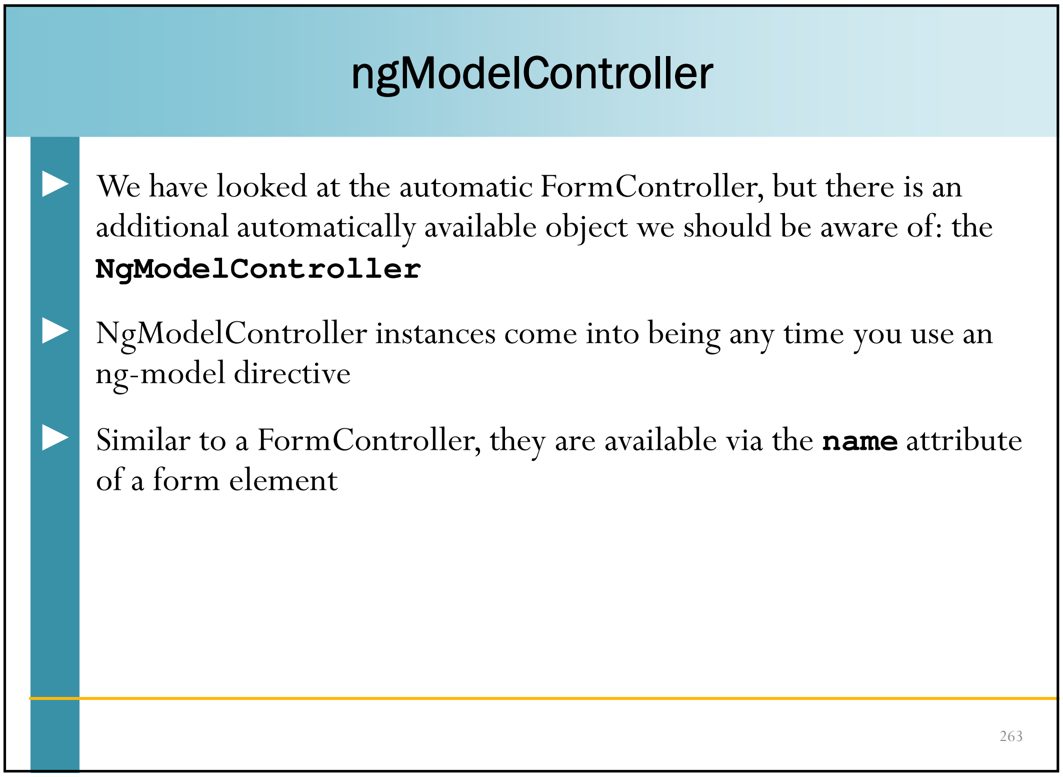
261



Exercise: Validation and styling

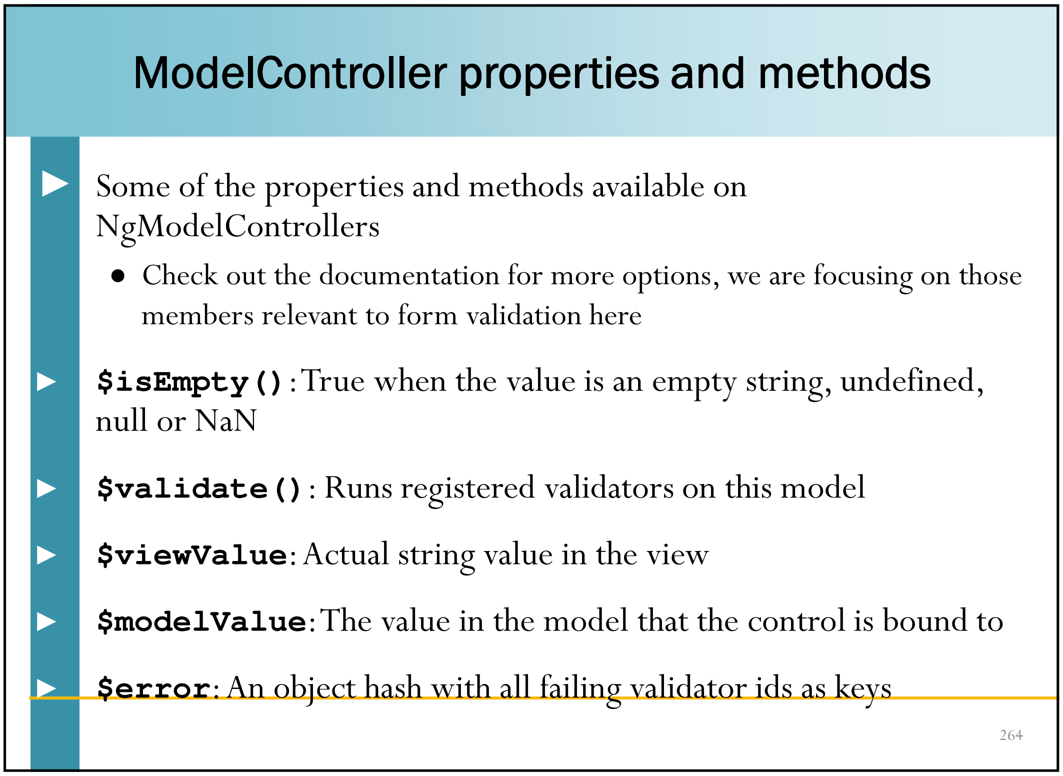
- In which we take a form and add validation rules and then style form elements according to whether they are valid
- Chapter: Forms
- Exercises/validation-styling.html
- Exercises/validation-styling.js

262



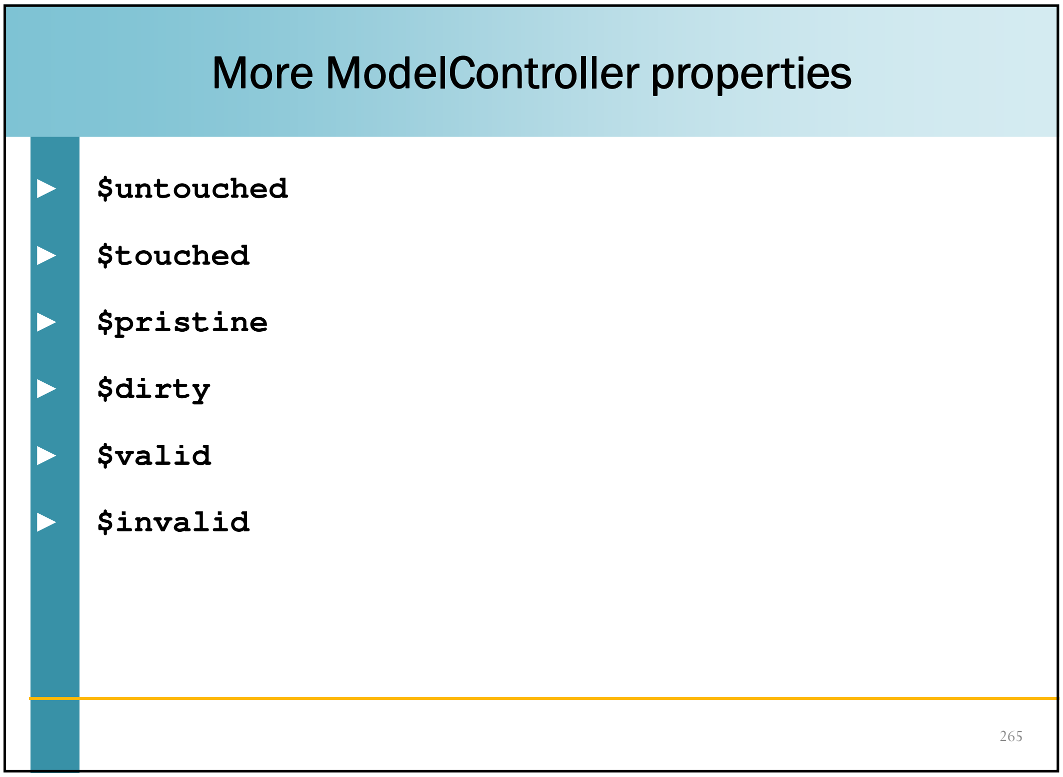
ngModelController

- We have looked at the automatic FormController, but there is an additional automatically available object we should be aware of: the **`NgModelController`**
- NgModelController instances come into being any time you use an ng-model directive
- Similar to a FormController, they are available via the **`name`** attribute of a form element



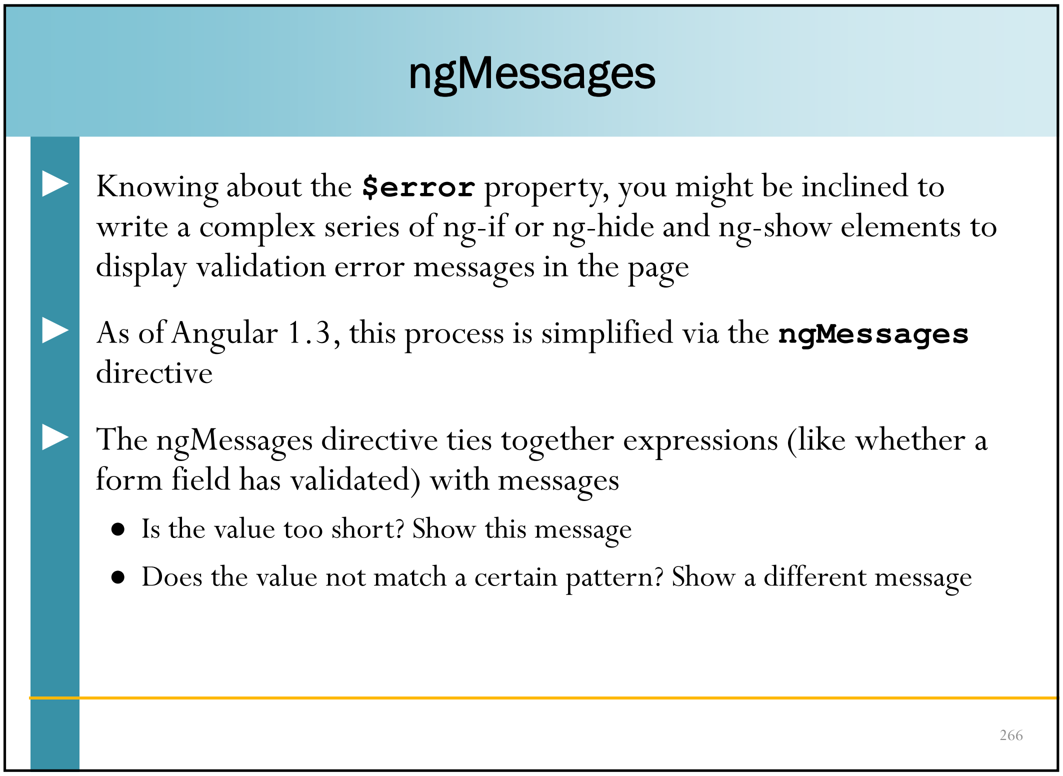
ModelController properties and methods

- Some of the properties and methods available on NgModelControllers
  - Check out the documentation for more options, we are focusing on those members relevant to form validation here
- **`$isEmpty()`**: True when the value is an empty string, undefined, null or NaN
- **`$validate()`**: Runs registered validators on this model
- **`$viewValue`**: Actual string value in the view
- **`$modelValue`**: The value in the model that the control is bound to
- **`$error`**: An object hash with all failing validator ids as keys



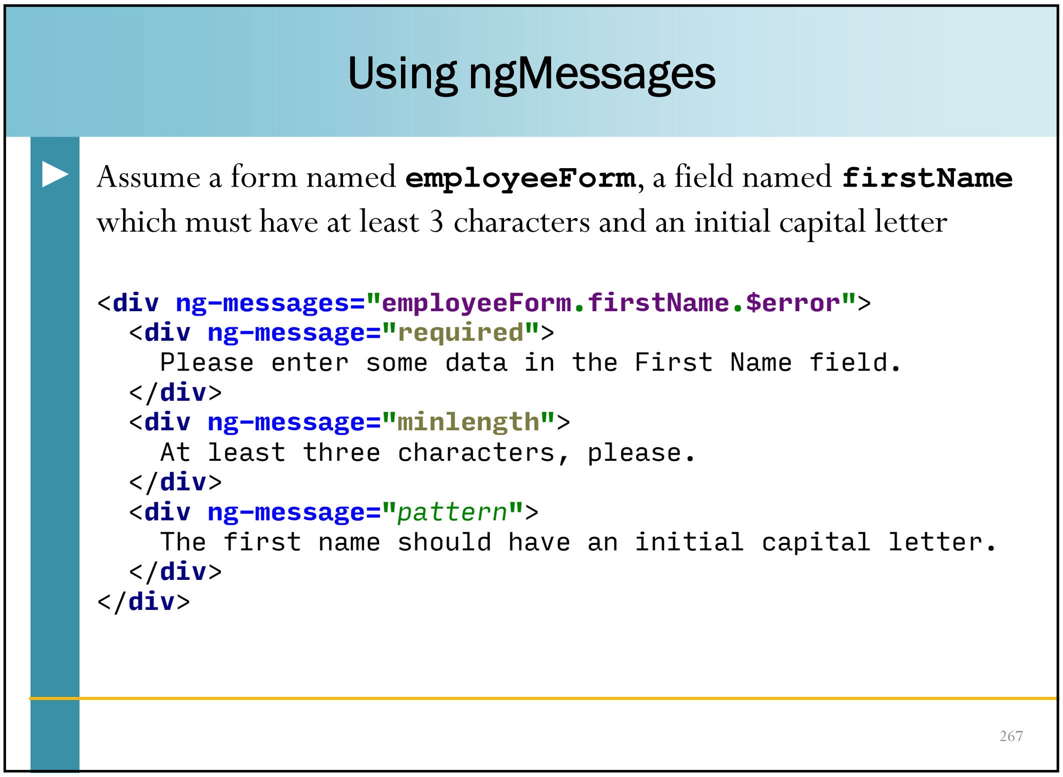
More ModelController properties

- `$untouched`
- `$touched`
- `$pristine`
- `$dirty`
- `$valid`
- `$invalid`



ngMessages

- Knowing about the **`$error`** property, you might be inclined to write a complex series of ng-if or ng-hide and ng-show elements to display validation error messages in the page
- As of Angular 1.3, this process is simplified via the **`ngMessages`** directive
- The ngMessages directive ties together expressions (like whether a form field has validated) with messages
  - Is the value too short? Show this message
  - Does the value not match a certain pattern? Show a different message

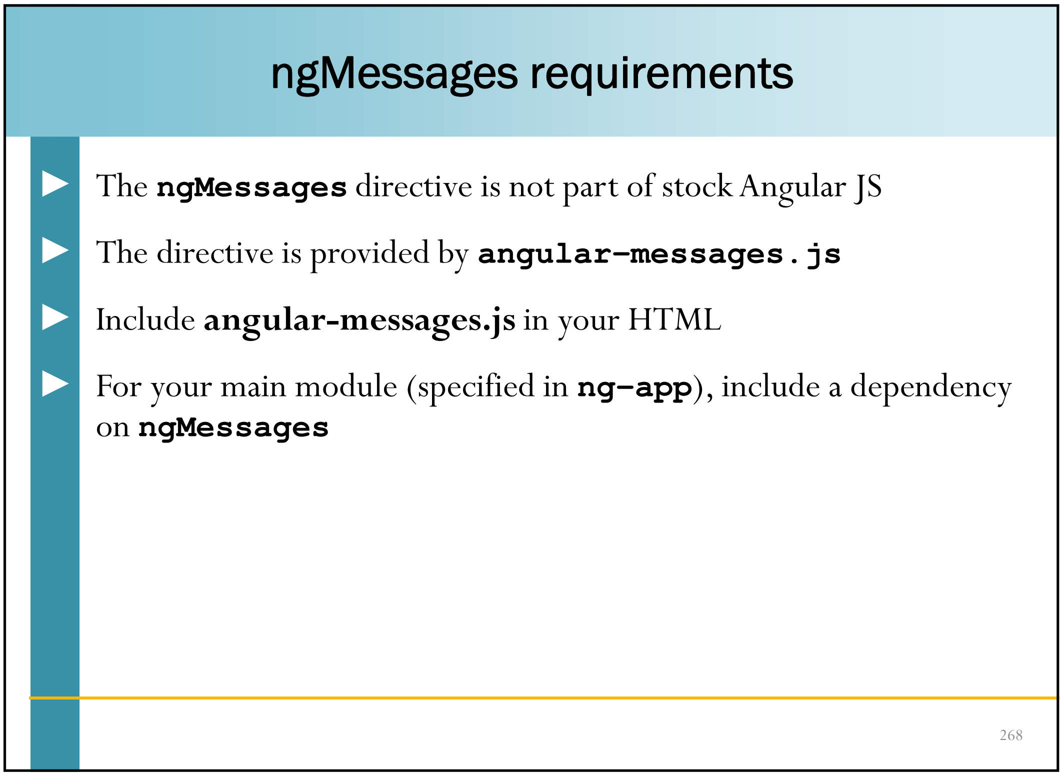


Using ngMessages

- Assume a form named **`employeeForm`**, a field named **`firstName`** which must have at least 3 characters and an initial capital letter

```
<div ng-messages="employeeForm.firstName.$error">
  <div ng-message="required">
    Please enter some data in the First Name field.
  </div>
  <div ng-message="minlength">
    At least three characters, please.
  </div>
  <div ng-message="pattern">
    The first name should have an initial capital letter.
  </div>
</div>
```

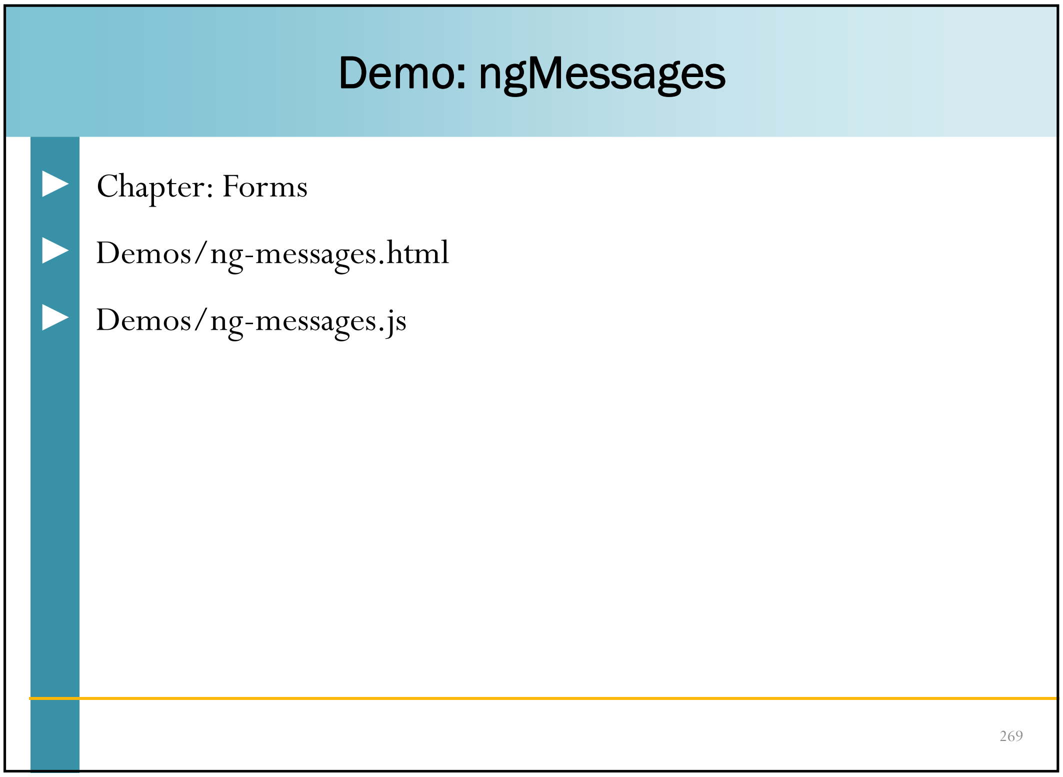
267



ngMessages requirements

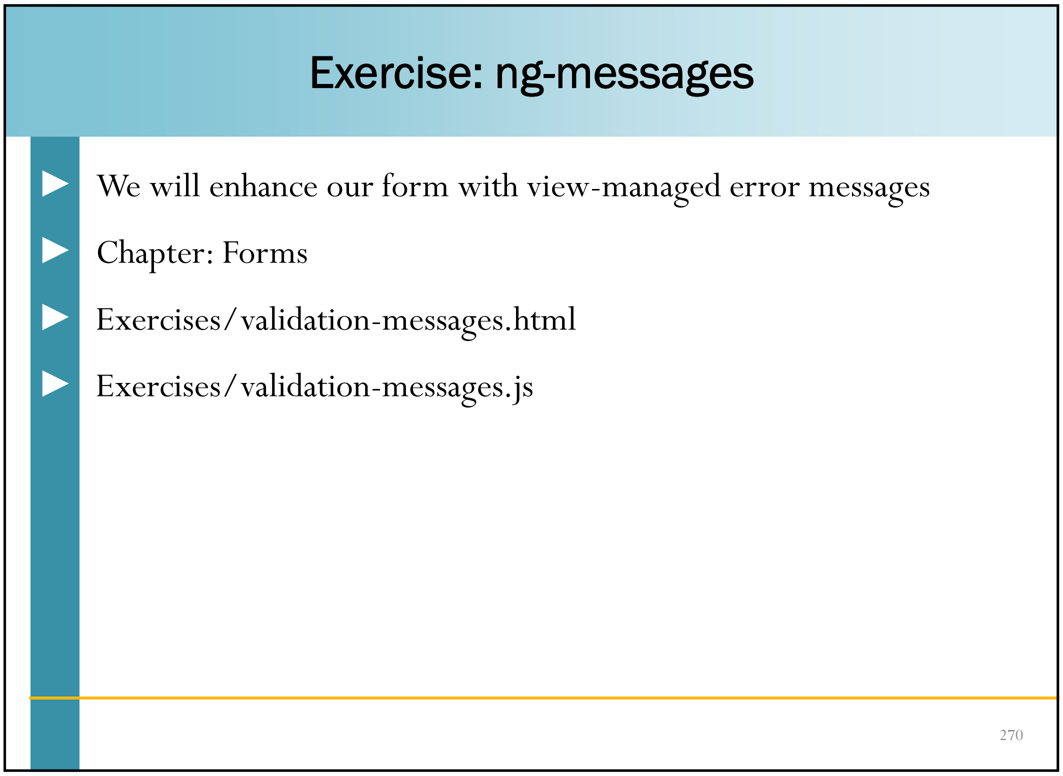
- The **`ngMessages`** directive is not part of stock Angular JS
- The directive is provided by **`angular-messages.js`**
- Include **angular-messages.js** in your HTML
- For your main module (specified in **`ng-app`**), include a dependency on **`ngMessages`**

268



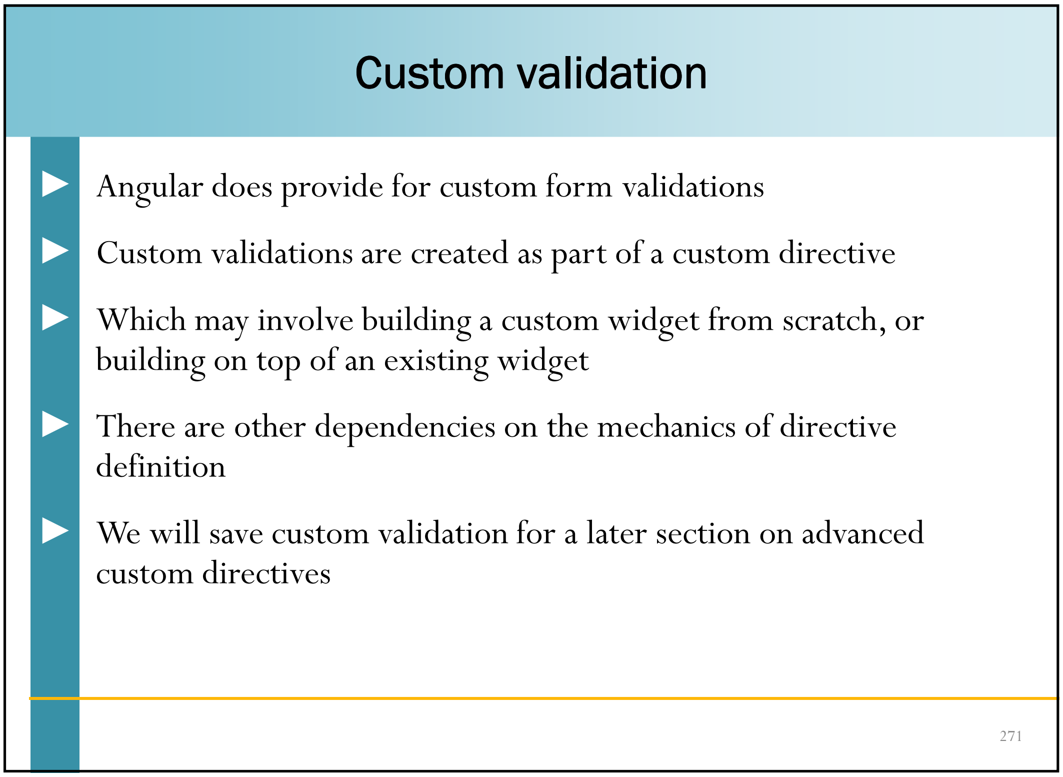
Demo: ngMessages

- Chapter: Forms
- Demos/ng-messages.html
- Demos/ng-messages.js



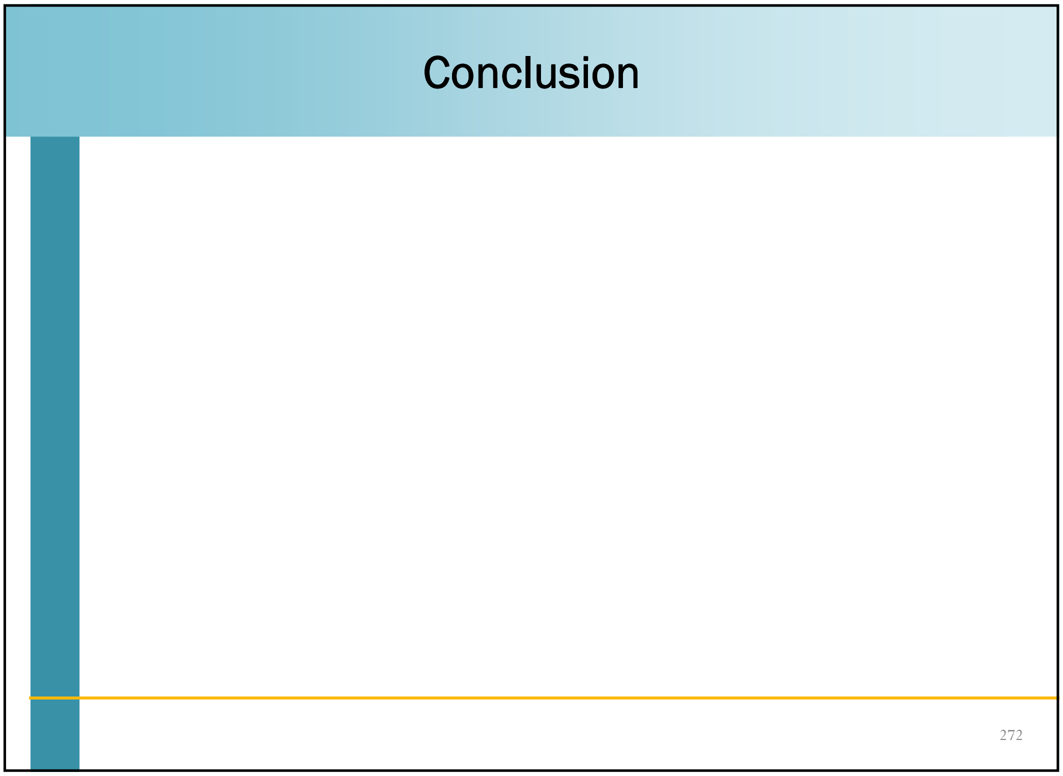
Exercise: ng-messages

- We will enhance our form with view-managed error messages
- Chapter: Forms
- Exercises/validation-messages.html
- Exercises/validation-messages.js

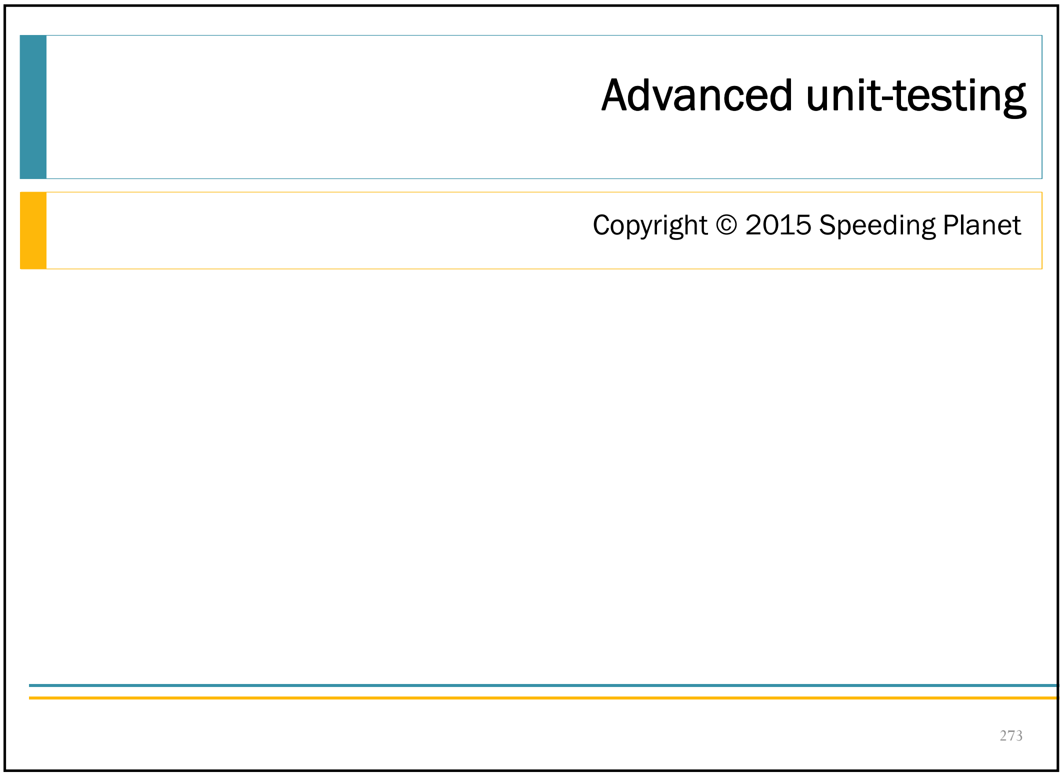


Custom validation

- Angular does provide for custom form validations
- Custom validations are created as part of a custom directive
- Which may involve building a custom widget from scratch, or building on top of an existing widget
- There are other dependencies on the mechanics of directive definition
- We will save custom validation for a later section on advanced custom directives

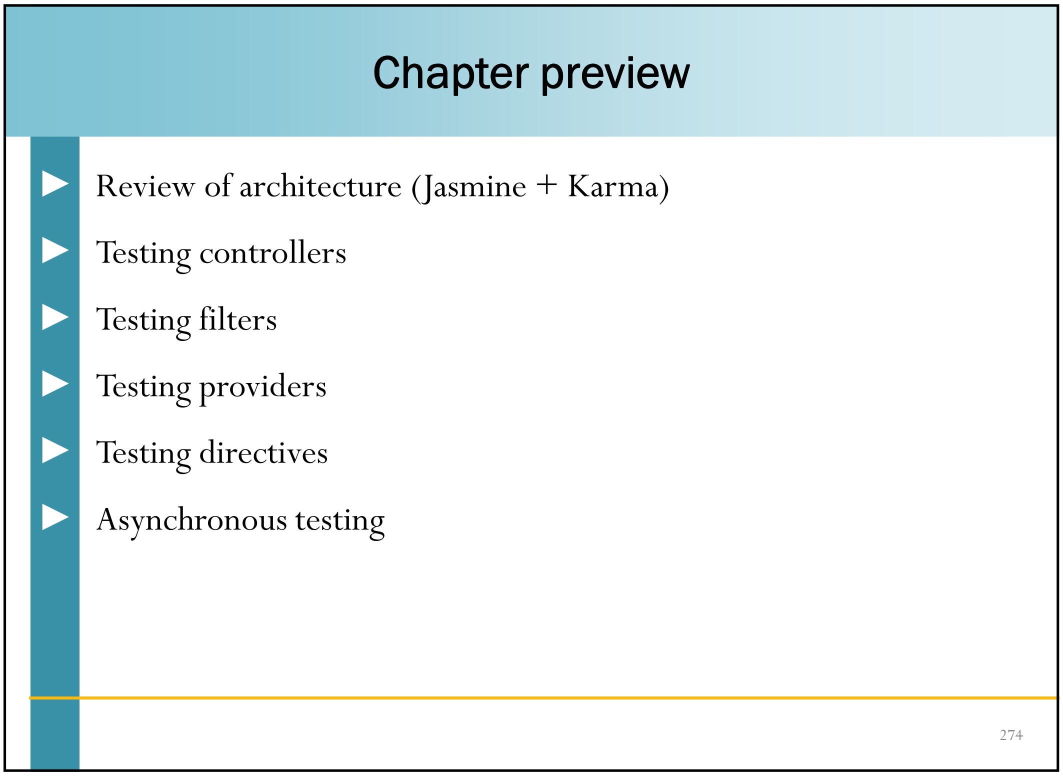


Conclusion



Advanced unit-testing

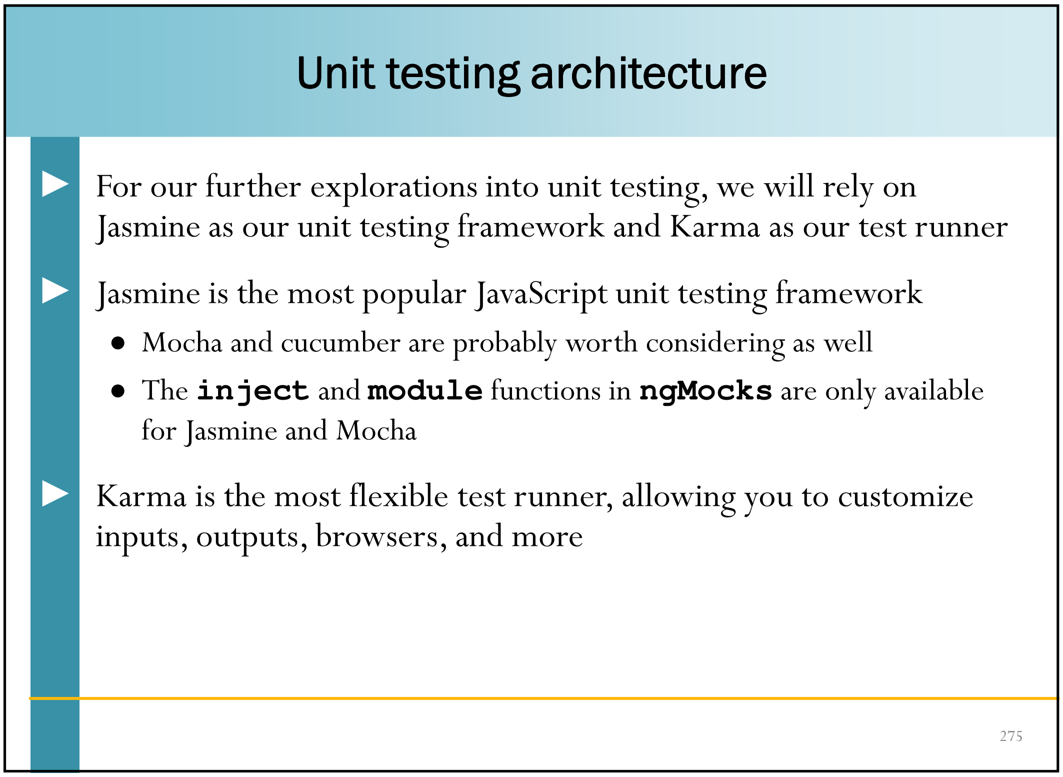
273



Chapter preview

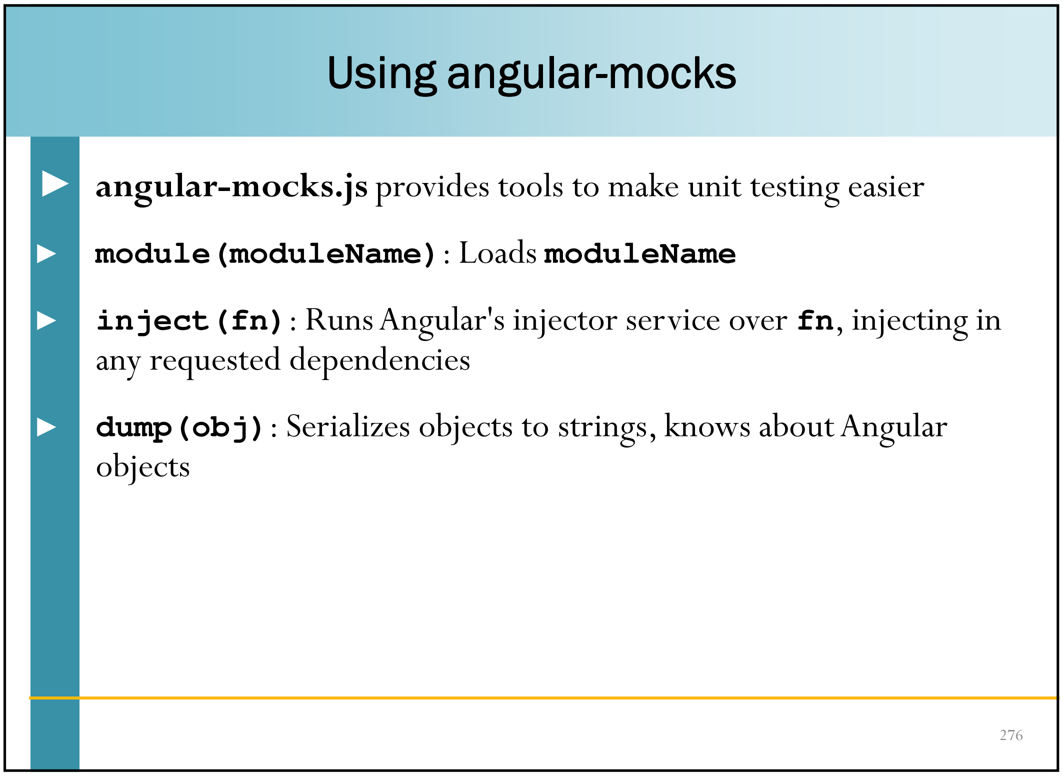
- Review of architecture (Jasmine + Karma)
- Testing controllers
- Testing filters
- Testing providers
- Testing directives
- Asynchronous testing

274



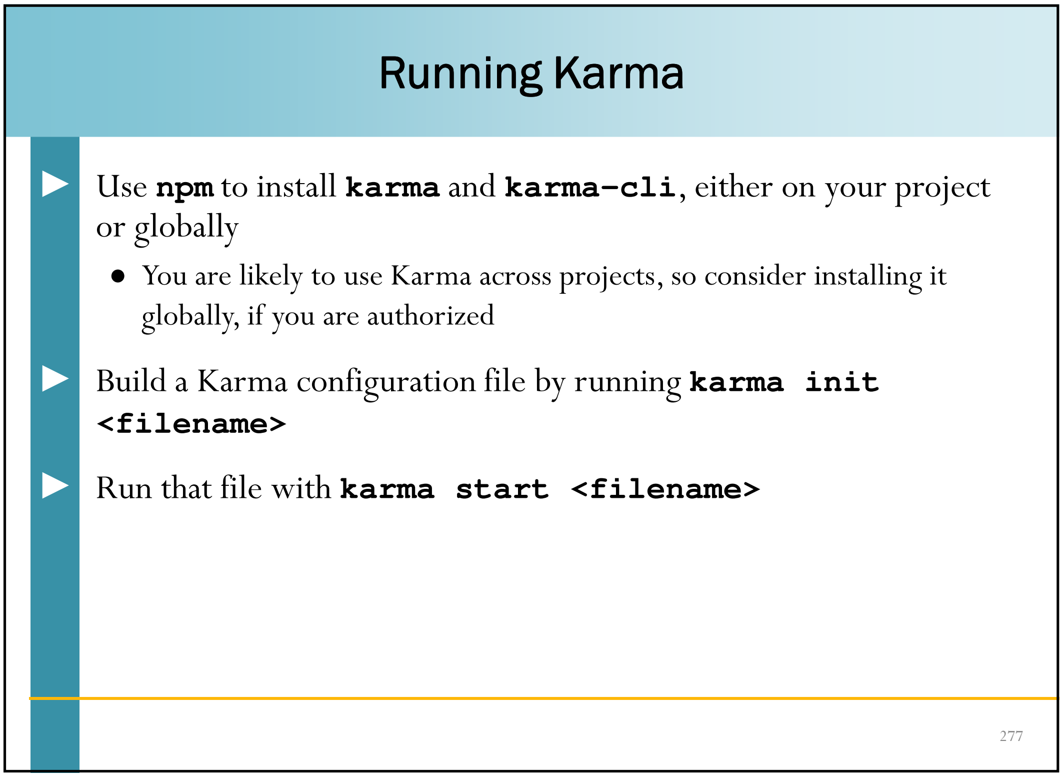
Unit testing architecture

- For our further explorations into unit testing, we will rely on Jasmine as our unit testing framework and Karma as our test runner
- Jasmine is the most popular JavaScript unit testing framework
  - Mocha and cucumber are probably worth considering as well
  - The **`inject`** and **`module`** functions in **`ngMocks`** are only available for Jasmine and Mocha
- Karma is the most flexible test runner, allowing you to customize inputs, outputs, browsers, and more



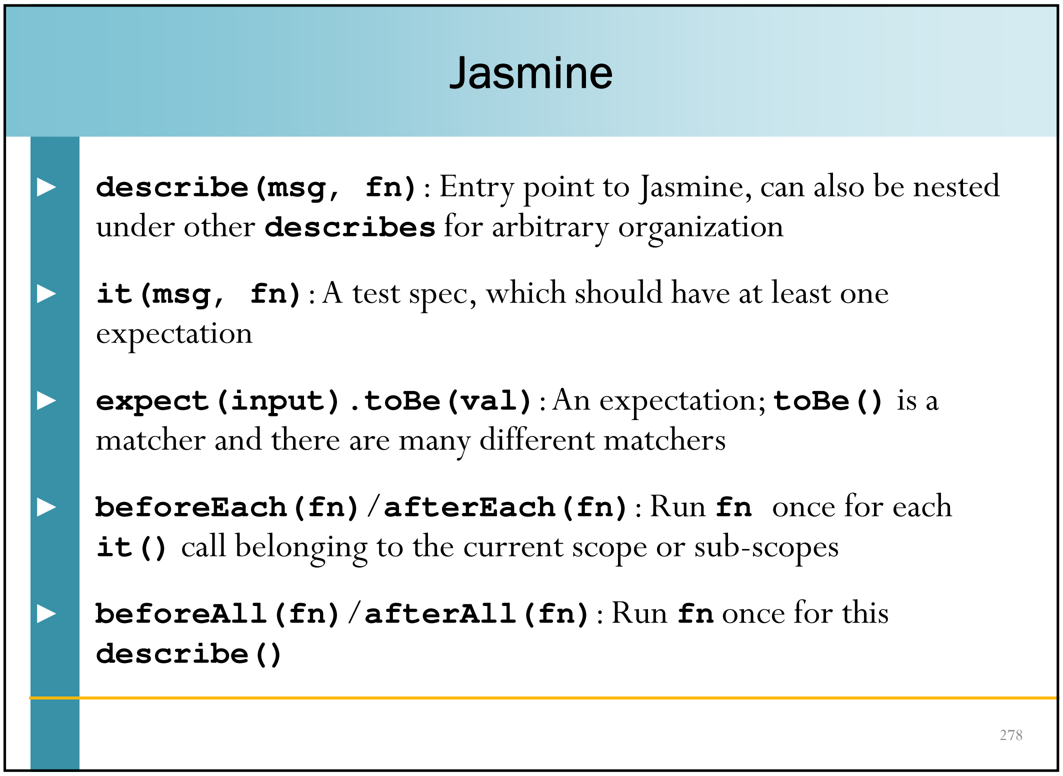
Using angular-mocks

- **angular-mocks.js** provides tools to make unit testing easier
- **`module(moduleName)`** : Loads **`moduleName`**
- **`inject(fn)`** : Runs Angular's injector service over **`fn`**, injecting in any requested dependencies
- **`dump(obj)`** : Serializes objects to strings, knows about Angular objects



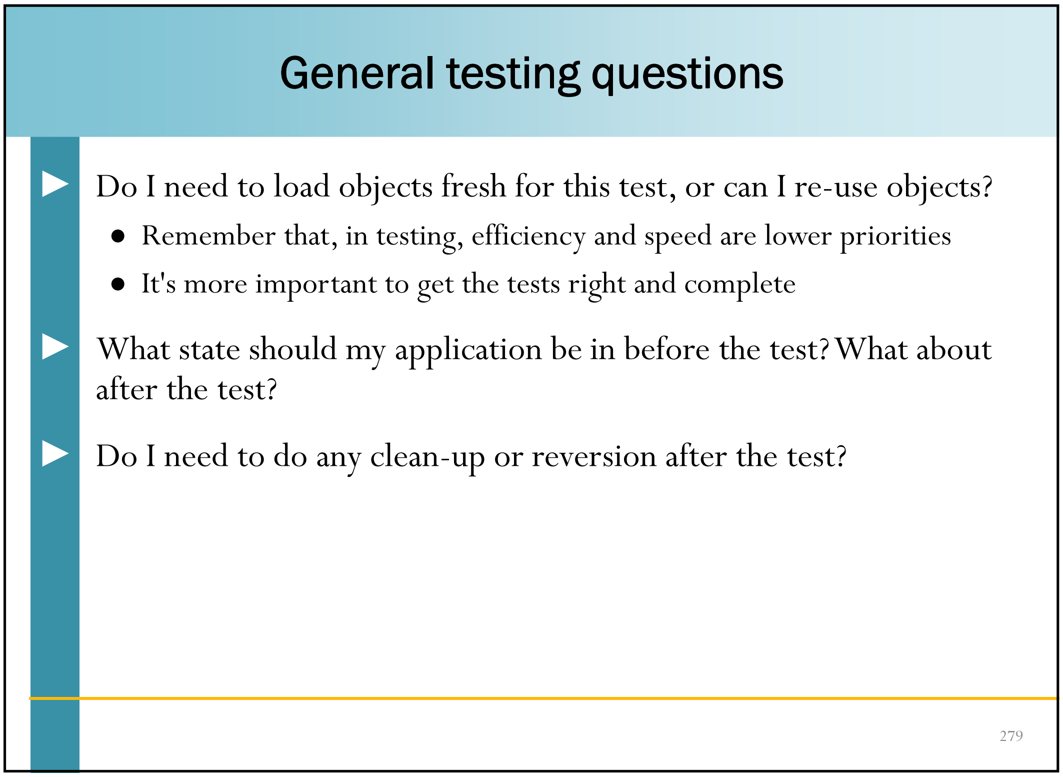
Running Karma

- Use **`npm`** to install **`karma`** and **`karma-cli`**, either on your project or globally
  - You are likely to use Karma across projects, so consider installing it globally, if you are authorized
- Build a Karma configuration file by running **`karma init <filename>`**
- Run that file with **`karma start <filename>`**



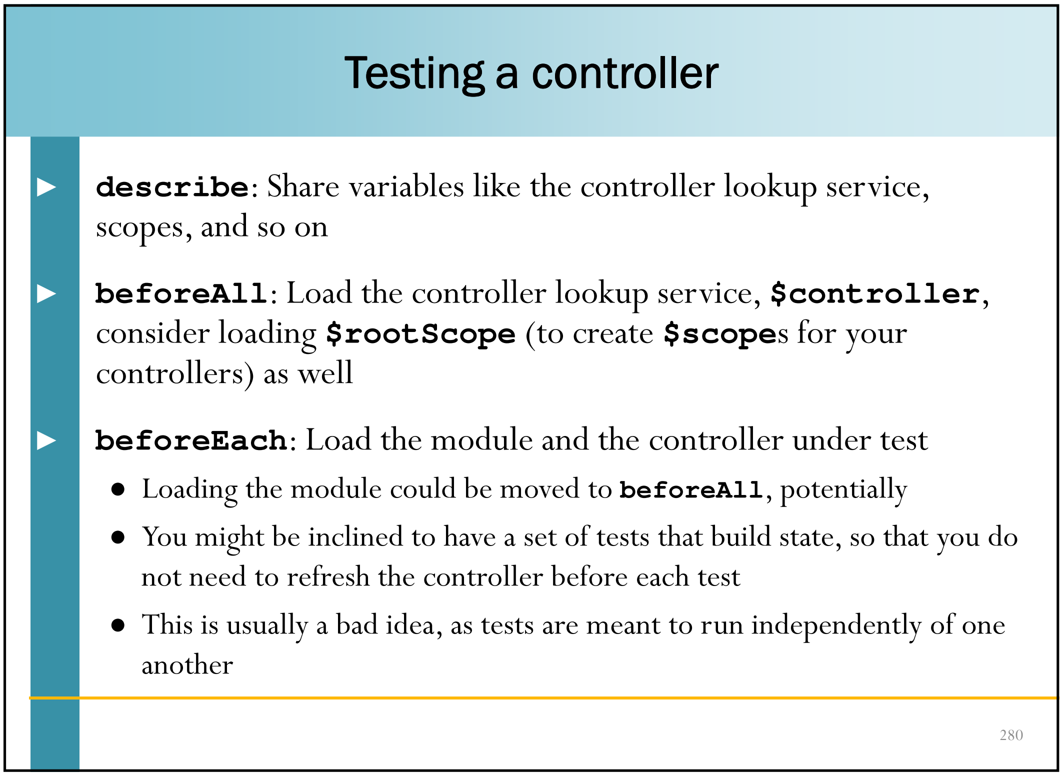
Jasmine

- **`describe(msg, fn)`** : Entry point to Jasmine, can also be nested under other **`describes`** for arbitrary organization
- **`it(msg, fn)`** : A test spec, which should have at least one expectation
- **`expect(input).toBe(val)`** : An expectation; **`toBe()`** is a matcher and there are many different matchers
- **`beforeEach(fn)`**/**`afterEach(fn)`** : Run **`fn`** once for each **`it()`** call belonging to the current scope or sub-scopes
- **`beforeAll(fn)`**/**`afterAll(fn)`** : Run **`fn`** once for this **`describe()`**



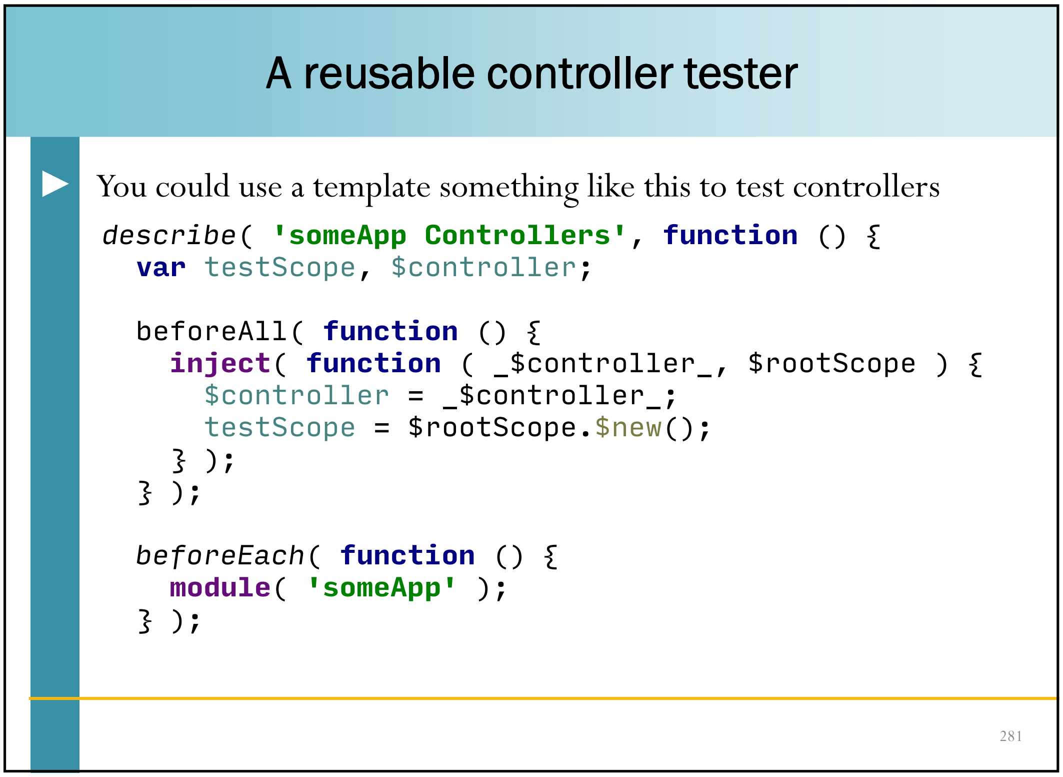
General testing questions

- Do I need to load objects fresh for this test, or can I re-use objects?
  - Remember that, in testing, efficiency and speed are lower priorities
  - It's more important to get the tests right and complete
- What state should my application be in before the test? What about after the test?
- Do I need to do any clean-up or reversion after the test?



Testing a controller

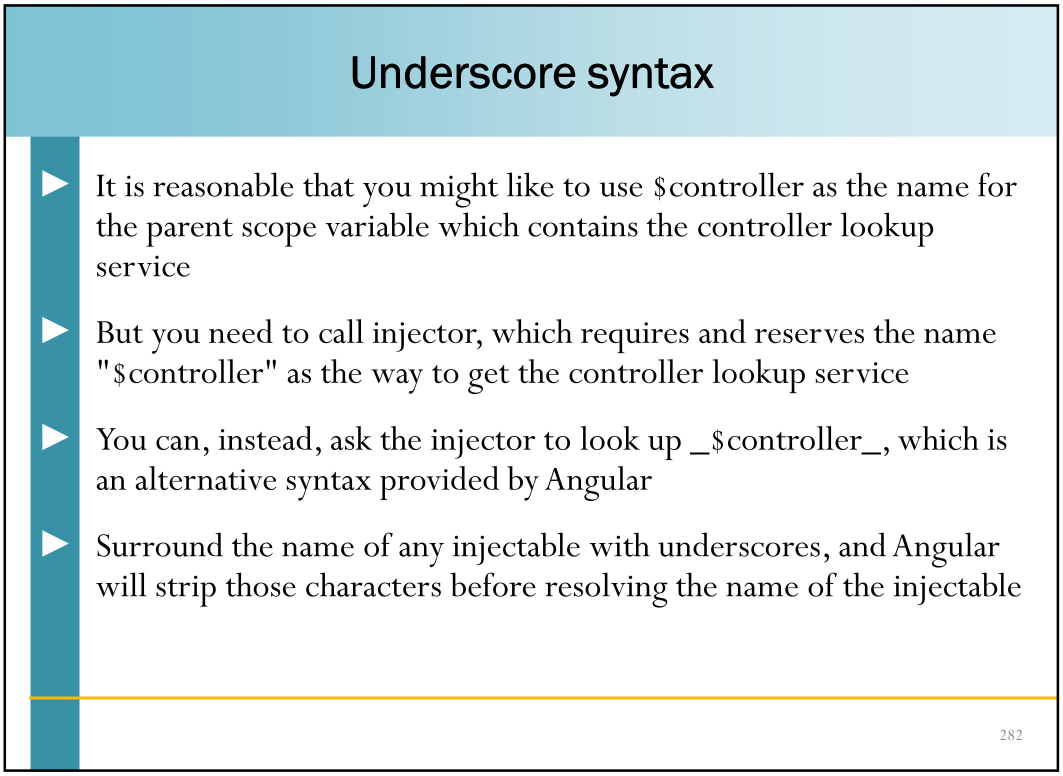
- **`describe`**: Share variables like the controller lookup service, scopes, and so on
- **`beforeAll`**: Load the controller lookup service, **`$controller`**, consider loading **`$rootScope`** (to create **`$scope`**s for your controllers) as well
- **`beforeEach`**: Load the module and the controller under test
  - Loading the module could be moved to **`beforeAll`**, potentially
  - You might be inclined to have a set of tests that build state, so that you do not need to refresh the controller before each test
  - This is usually a bad idea, as tests are meant to run independently of one another



A reusable controller tester

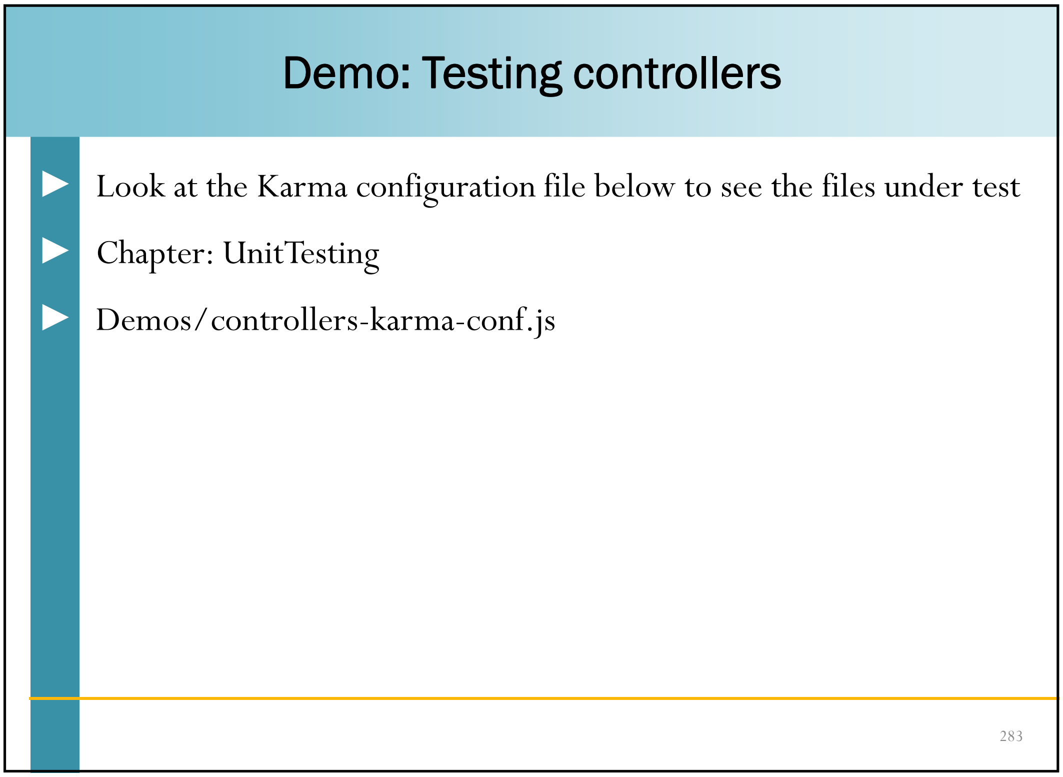
- You could use a template something like this to test controllers

```
describe( 'someApp Controllers', function () {
  var testScope, $controller;

  beforeAll( function () {
    inject( function ( _$controller_, $rootScope ) {
      $controller = _$controller_;
      testScope = $rootScope.$new();
    } );
  } );

  beforeEach( function () {
    module( 'someApp' );
  } );
```



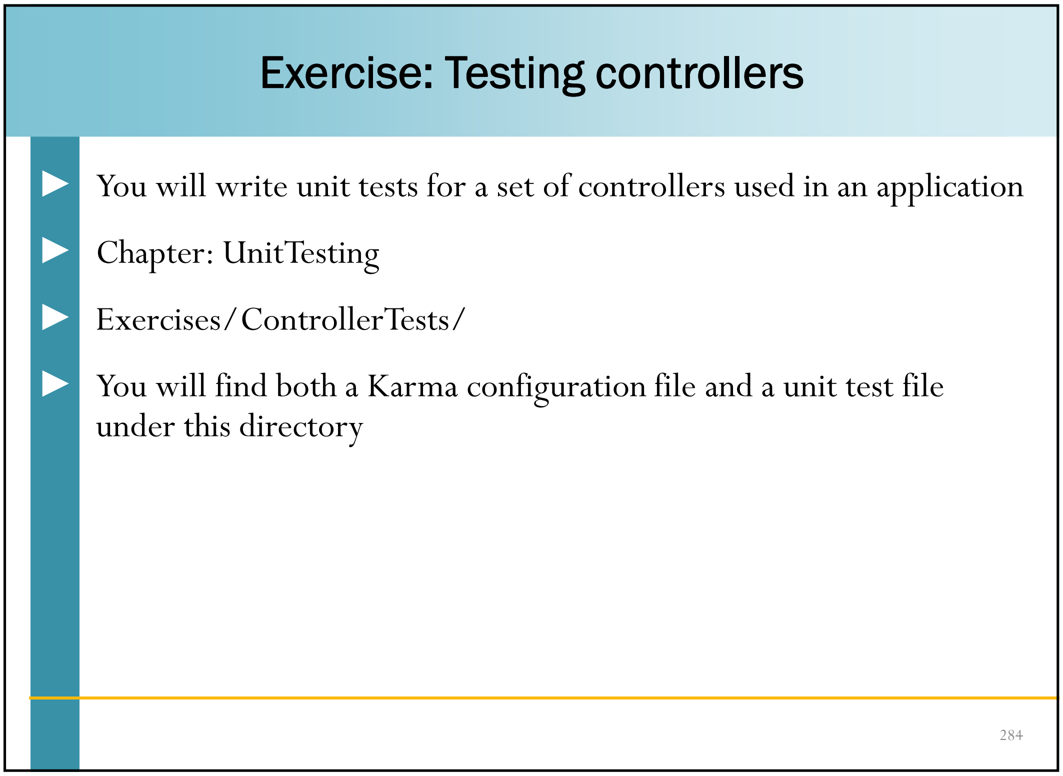
Underscore syntax

- It is reasonable that you might like to use $controller as the name for the parent scope variable which contains the controller lookup service
- But you need to call injector, which requires and reserves the name "$controller" as the way to get the controller lookup service
- You can, instead, ask the injector to look up _$controller_, which is an alternative syntax provided by Angular
- Surround the name of any injectable with underscores, and Angular will strip those characters before resolving the name of the injectable



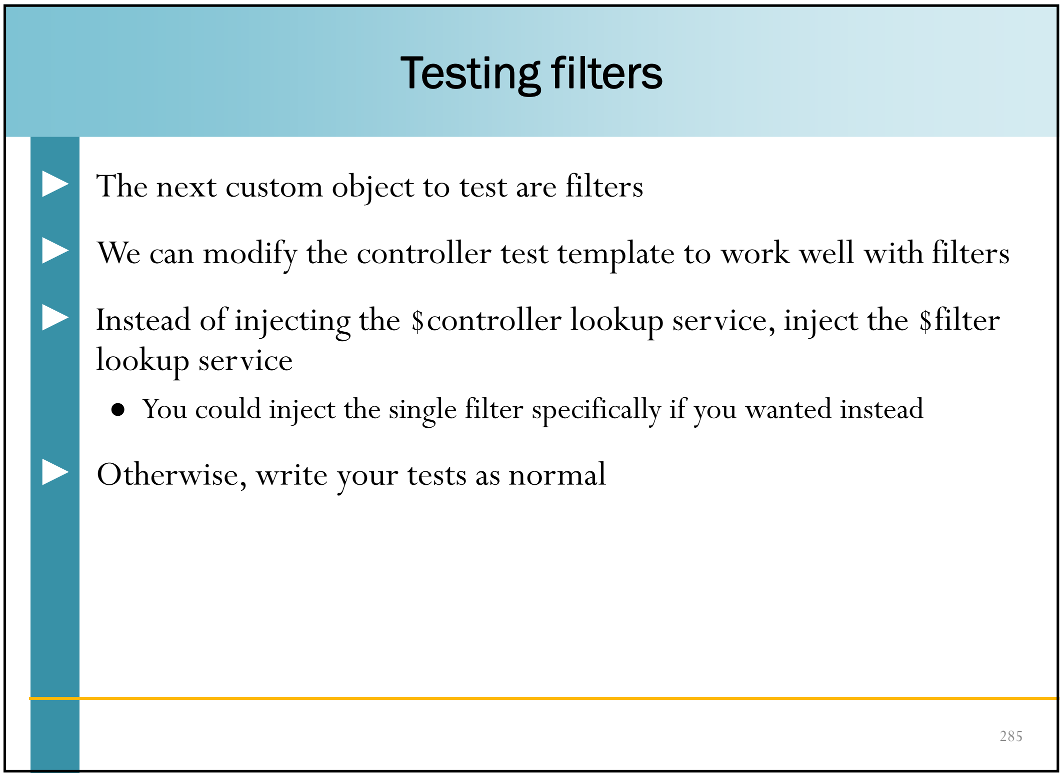
Demo: Testing controllers

- Look at the Karma configuration file below to see the files under test
- Chapter: UnitTesting
- Demos/controllers-karma-conf.js



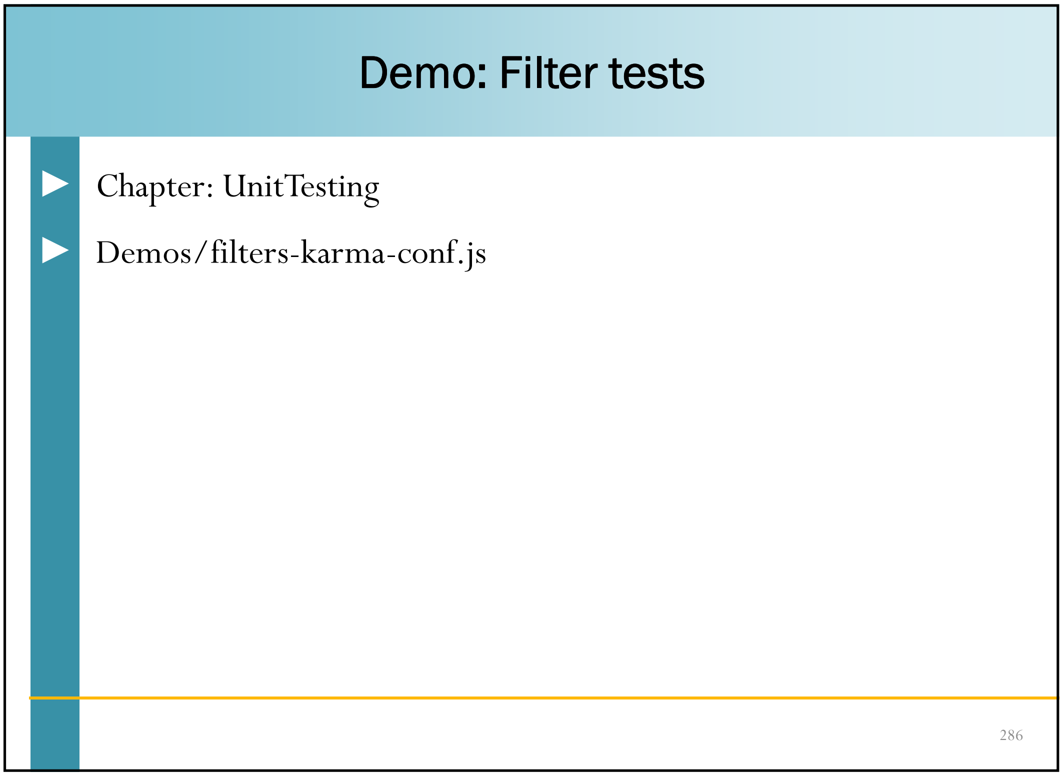
Exercise: Testing controllers

- You will write unit tests for a set of controllers used in an application
- Chapter: UnitTesting
- Exercises/ControllerTests/
- You will find both a Karma configuration file and a unit test file under this directory



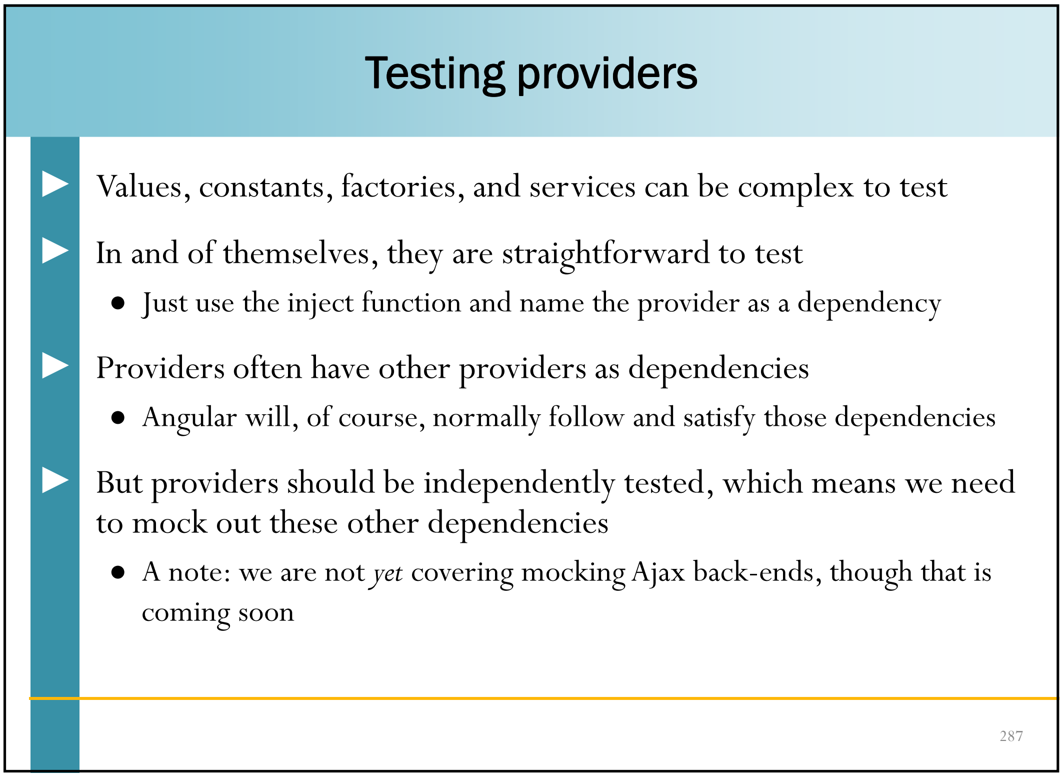
Testing filters

- The next custom object to test are filters
- We can modify the controller test template to work well with filters
- Instead of injecting the $controller lookup service, inject the $filter lookup service
  - You could inject the single filter specifically if you wanted instead
- Otherwise, write your tests as normal



Demo: Filter tests

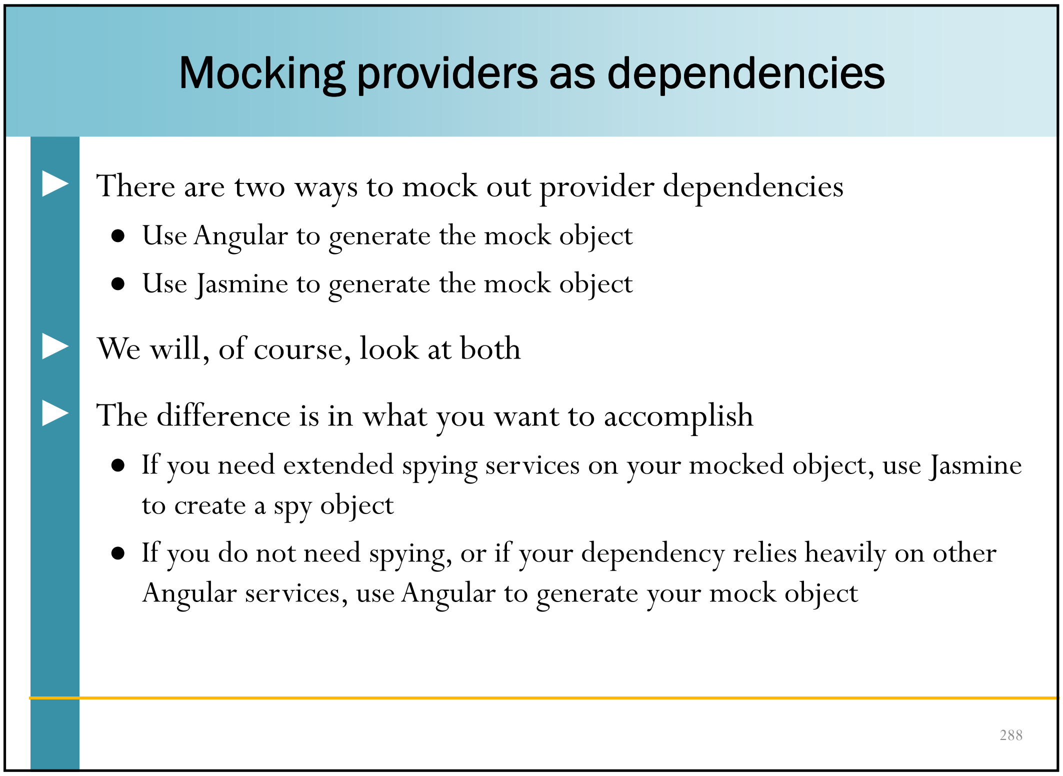
- Chapter: UnitTesting
- Demos/filters-karma-conf.js



Testing providers

- Values, constants, factories, and services can be complex to test
- In and of themselves, they are straightforward to test
  - Just use the inject function and name the provider as a dependency
- Providers often have other providers as dependencies
  - Angular will, of course, normally follow and satisfy those dependencies
- But providers should be independently tested, which means we need to mock out these other dependencies
  - A note: we are not *yet* covering mocking Ajax back-ends, though that is coming soon

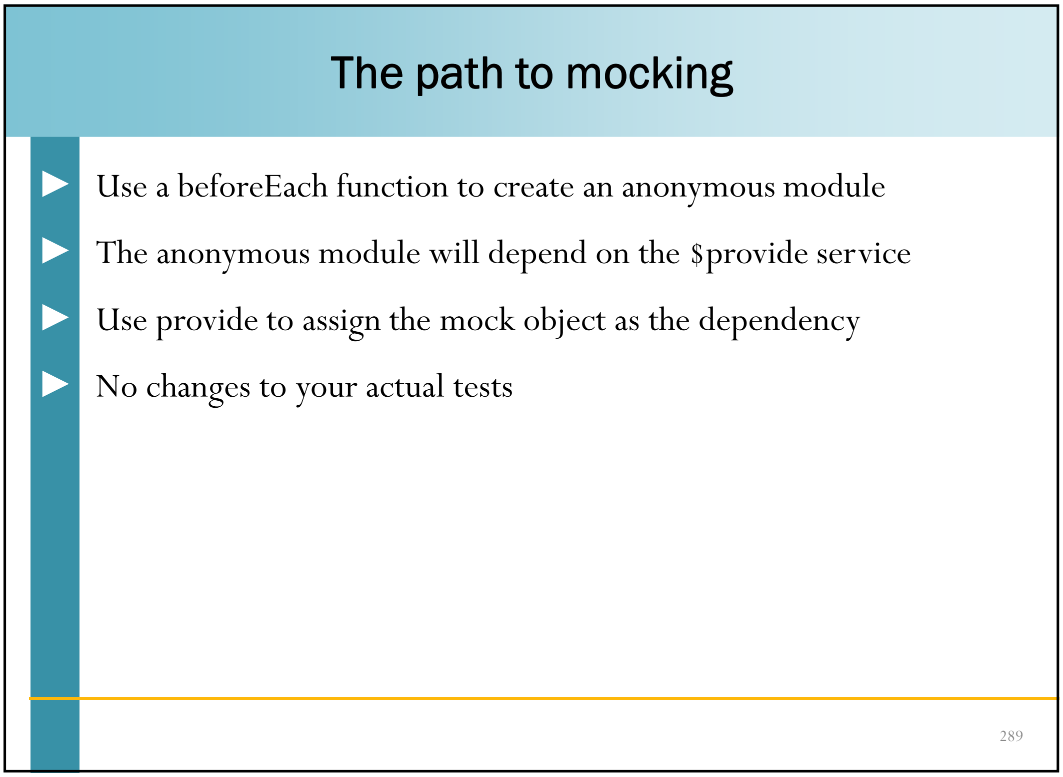
287



Mocking providers as dependencies

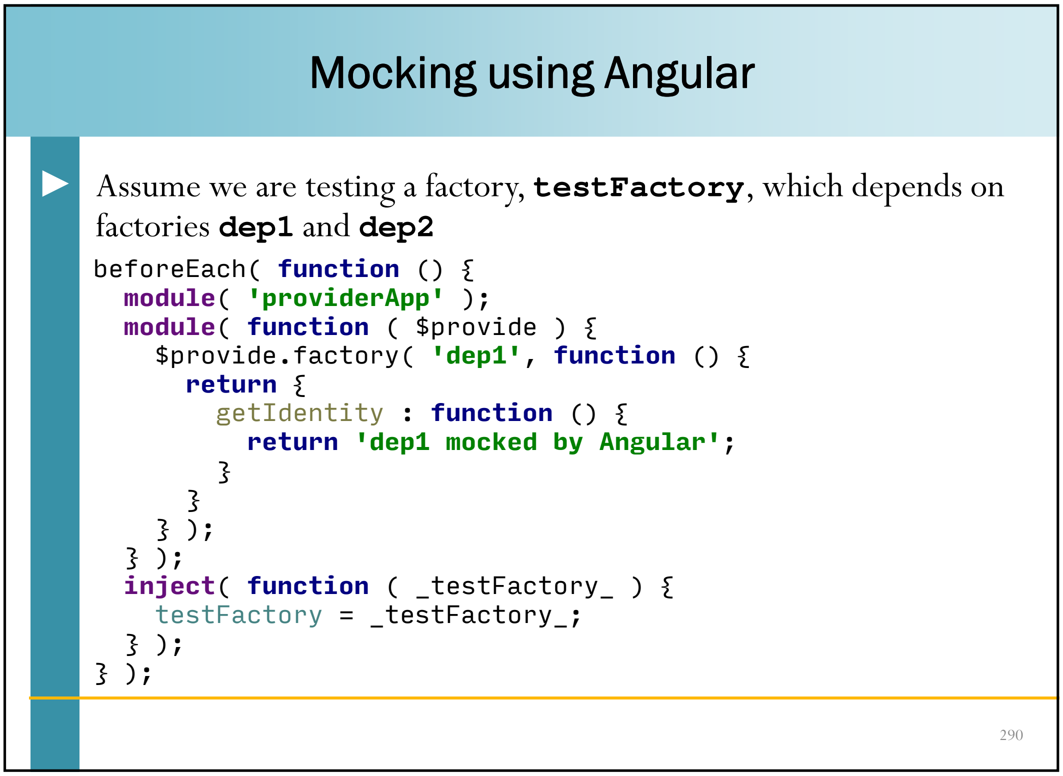
- There are two ways to mock out provider dependencies
  - Use Angular to generate the mock object
  - Use Jasmine to generate the mock object
- We will, of course, look at both
- The difference is in what you want to accomplish
  - If you need extended spying services on your mocked object, use Jasmine to create a spy object
  - If you do not need spying, or if your dependency relies heavily on other Angular services, use Angular to generate your mock object

288



The path to mocking

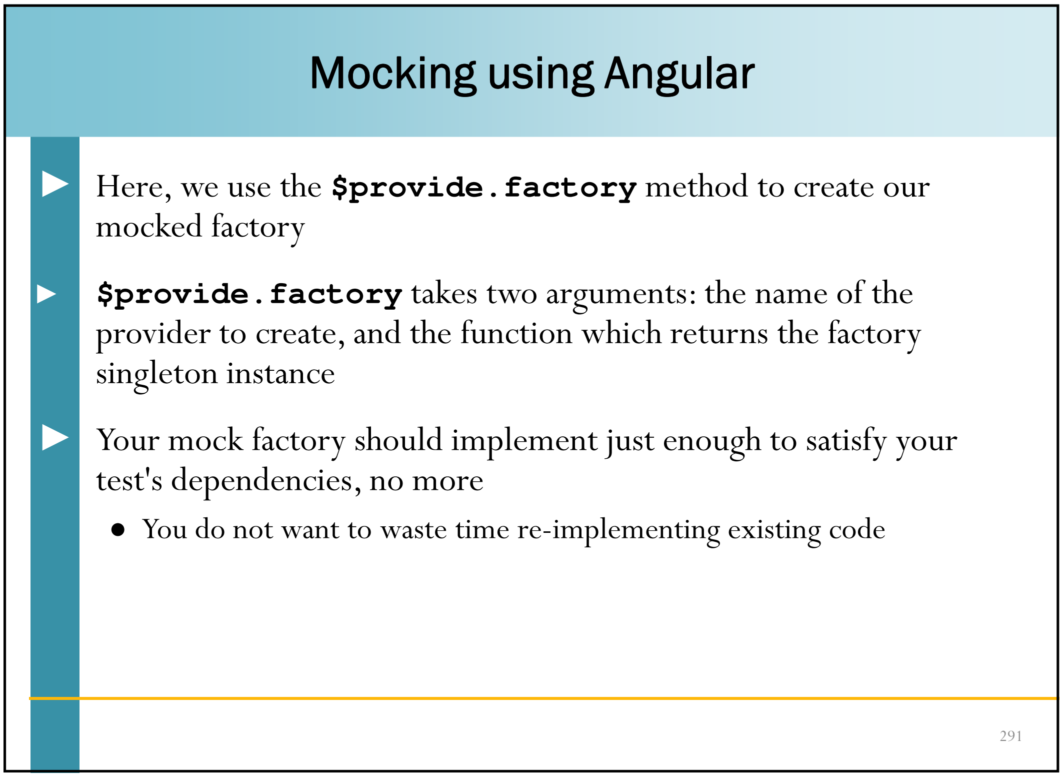
- Use a beforeEach function to create an anonymous module
- The anonymous module will depend on the $provide service
- Use provide to assign the mock object as the dependency
- No changes to your actual tests



Mocking using Angular

- Assume we are testing a factory, **testFactory**, which depends on factories **dep1** and **dep2**

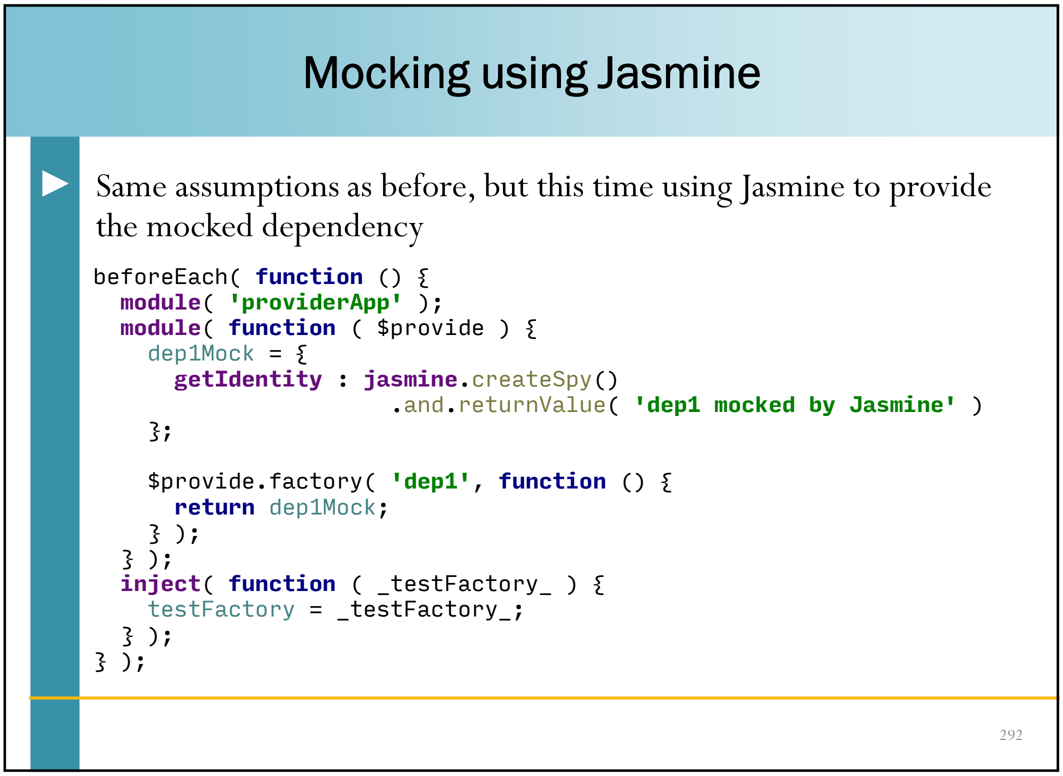
```
beforeEach( function () {
  module( 'providerApp' );
  module( function ( $provide ) {
    $provide.factory( 'dep1', function () {
      return {
        getIdentity : function () {
          return 'dep1 mocked by Angular';
        }
      }
    } );
  } );
  inject( function ( _testFactory_ ) {
    testFactory = _testFactory_;
  } );
} );
```



Mocking using Angular

- Here, we use the **`$provide.factory`** method to create our mocked factory
- **`$provide.factory`** takes two arguments: the name of the provider to create, and the function which returns the factory singleton instance
- Your mock factory should implement just enough to satisfy your test's dependencies, no more
  - You do not want to waste time re-implementing existing code

291

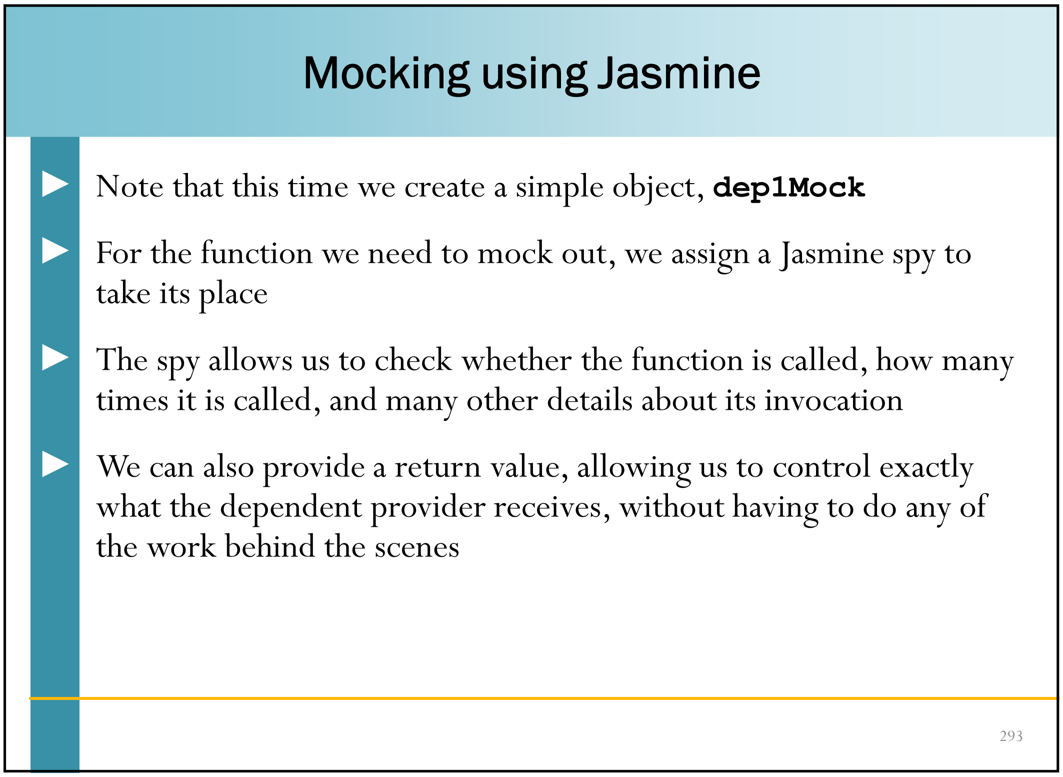


Mocking using Jasmine

- Same assumptions as before, but this time using Jasmine to provide the mocked dependency

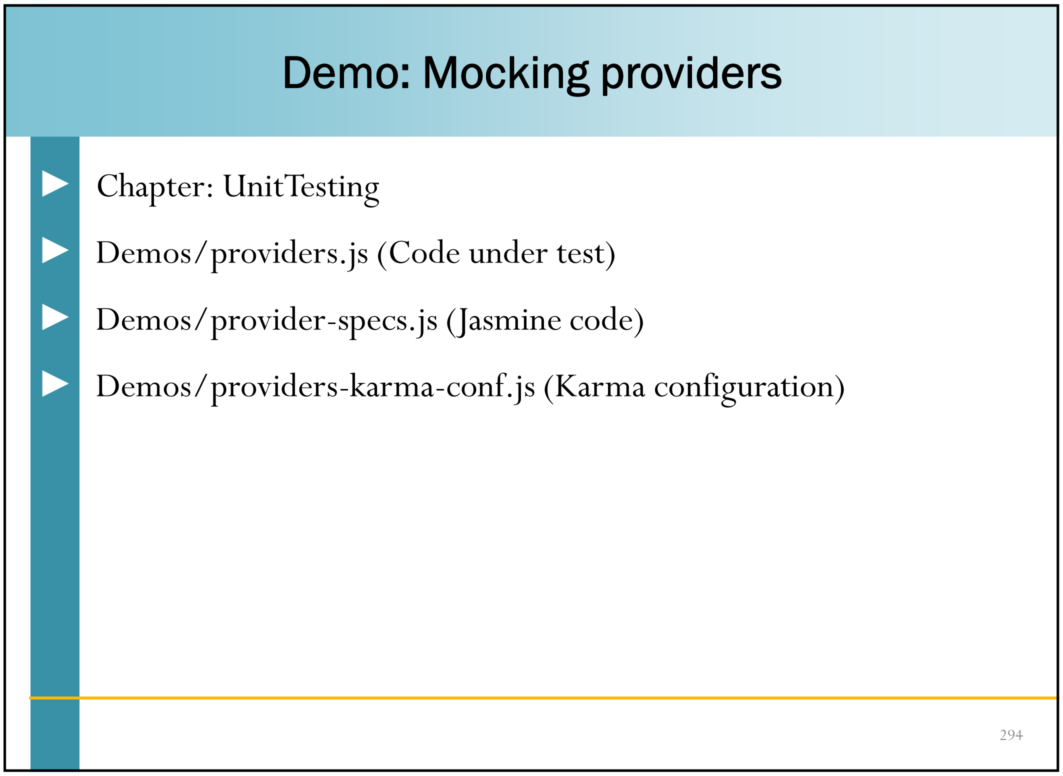
```
beforeEach( function () {
  module( 'providerApp' );
  module( function ( $provide ) {
    dep1Mock = {
      getIdentity : jasmine.createSpy()
                        .and.returnValue( 'dep1 mocked by Jasmine' )
    };

    $provide.factory( 'dep1', function () {
      return dep1Mock;
    } );
  } );
  inject( function ( _testFactory_ ) {
    testFactory = _testFactory_;
  } );
} );
```

292



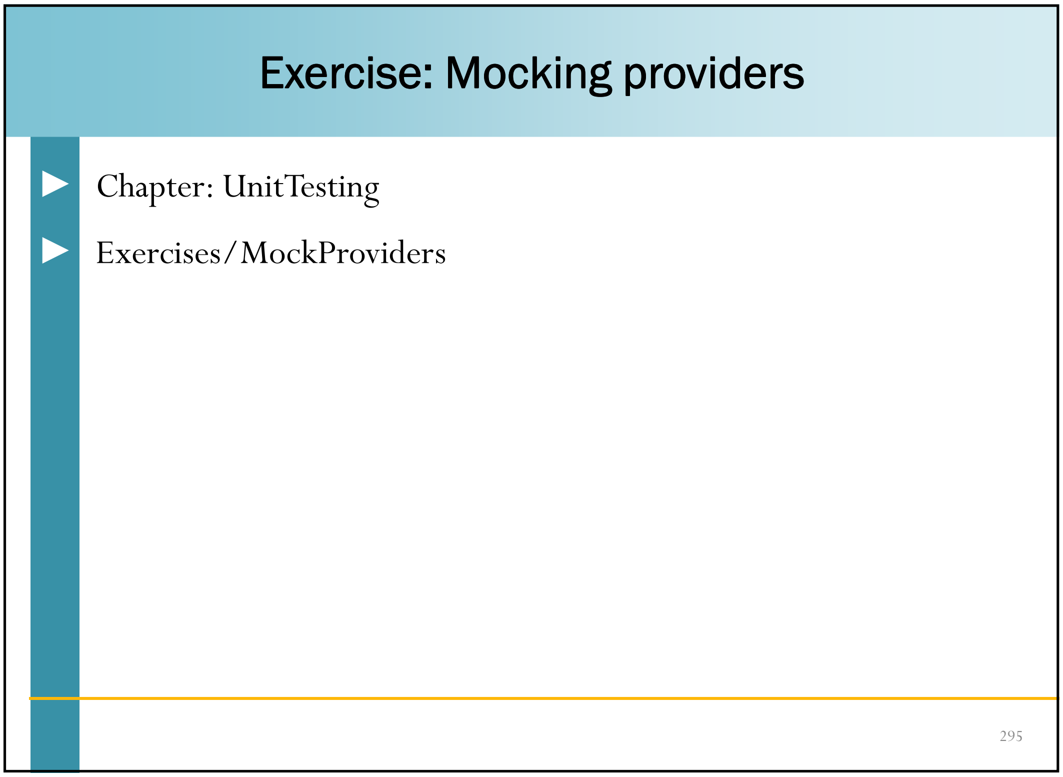
Mocking using Jasmine

- Note that this time we create a simple object, **`dep1Mock`**
- For the function we need to mock out, we assign a Jasmine spy to take its place
- The spy allows us to check whether the function is called, how many times it is called, and many other details about its invocation
- We can also provide a return value, allowing us to control exactly what the dependent provider receives, without having to do any of the work behind the scenes



Demo: Mocking providers

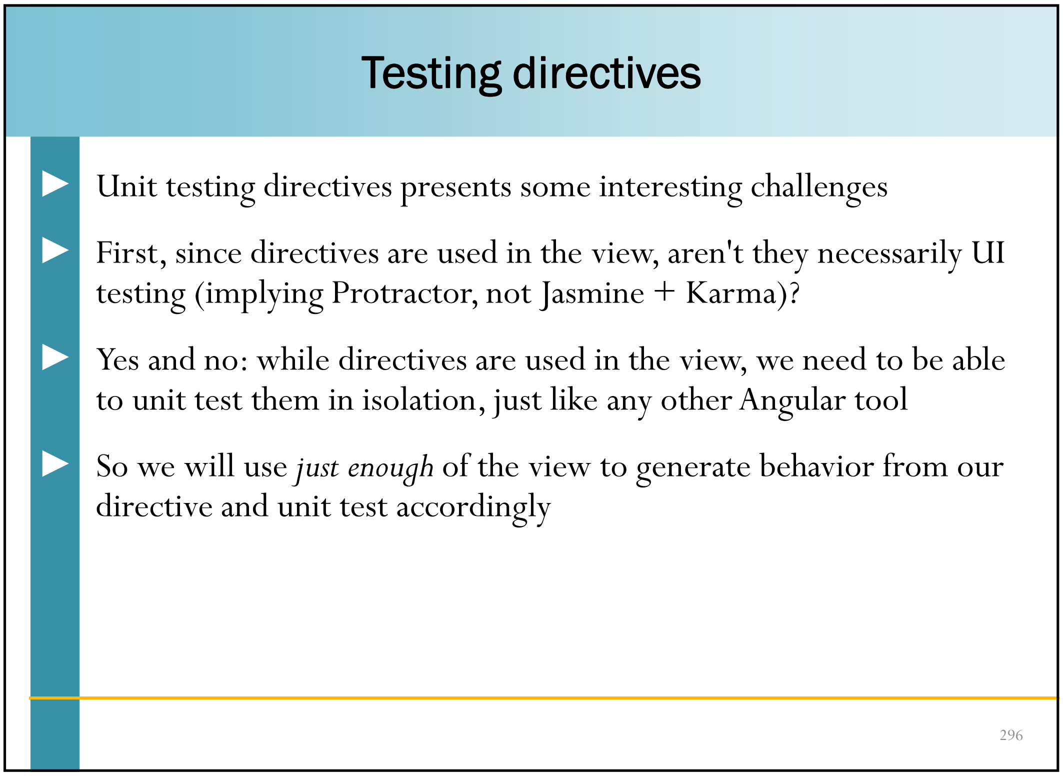
- Chapter: UnitTesting
- Demos/providers.js (Code under test)
- Demos/provider-specs.js (Jasmine code)
- Demos/providers-karma-conf.js (Karma configuration)



Exercise: Mocking providers

- Chapter: UnitTesting
- Exercises/MockProviders

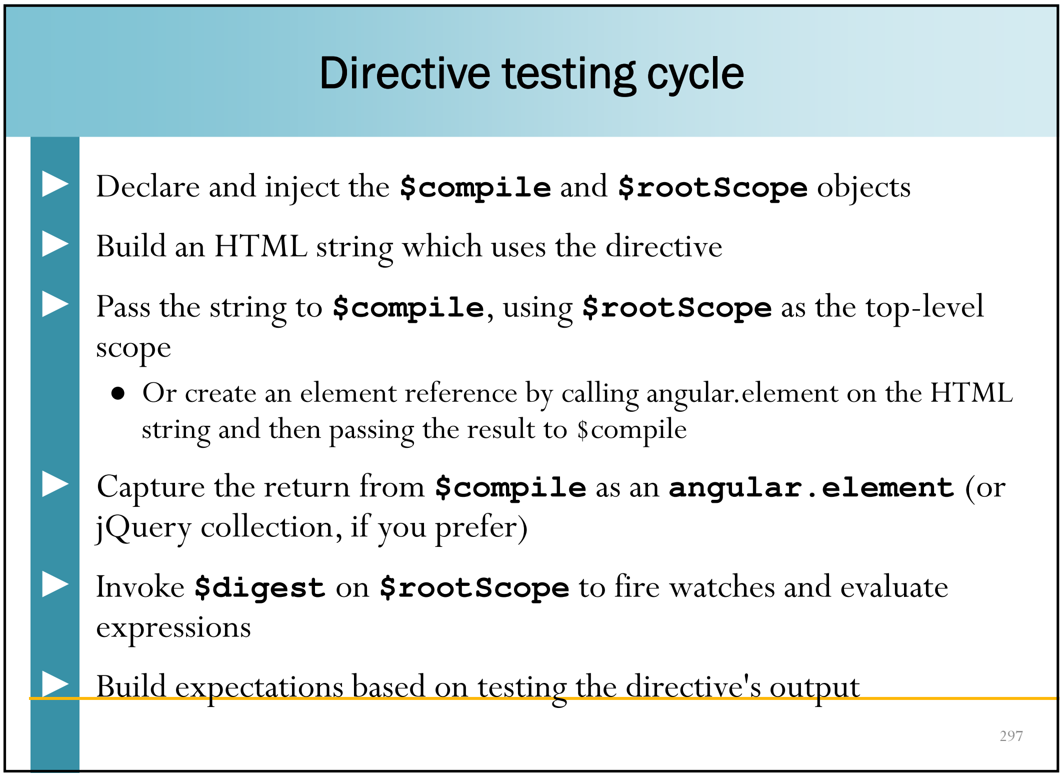
295



Testing directives

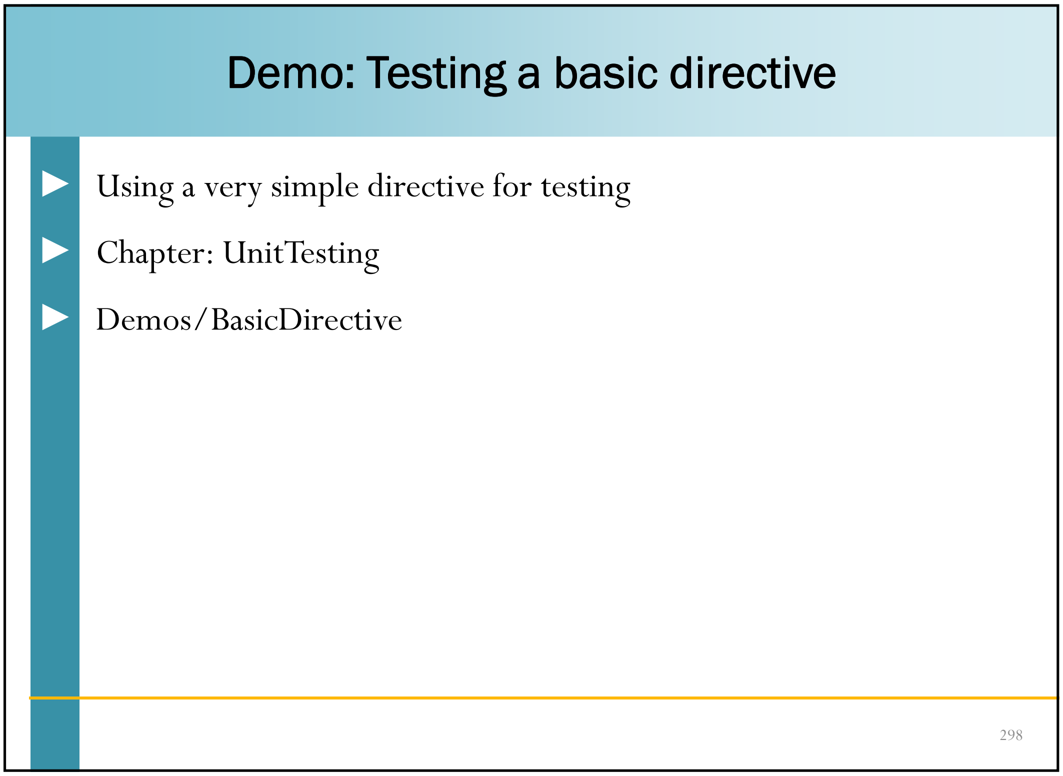
- Unit testing directives presents some interesting challenges
- First, since directives are used in the view, aren't they necessarily UI testing (implying Protractor, not Jasmine + Karma)?
- Yes and no: while directives are used in the view, we need to be able to unit test them in isolation, just like any other Angular tool
- So we will use *just enough* of the view to generate behavior from our directive and unit test accordingly

296



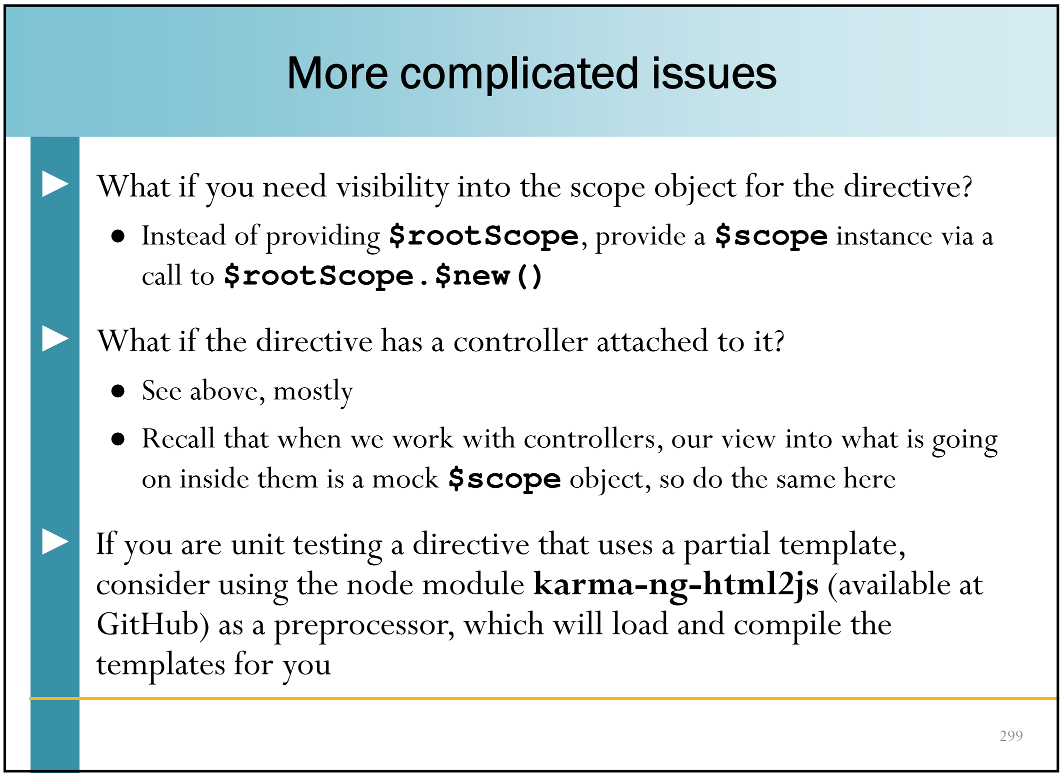
Directive testing cycle

- Declare and inject the `$compile` and `$rootScope` objects
- Build an HTML string which uses the directive
- Pass the string to `$compile`, using `$rootScope` as the top-level scope
  - Or create an element reference by calling angular.element on the HTML string and then passing the result to $compile
- Capture the return from `$compile` as an `angular.element` (or jQuery collection, if you prefer)
- Invoke `$digest` on `$rootScope` to fire watches and evaluate expressions
- Build expectations based on testing the directive's output



Demo: Testing a basic directive

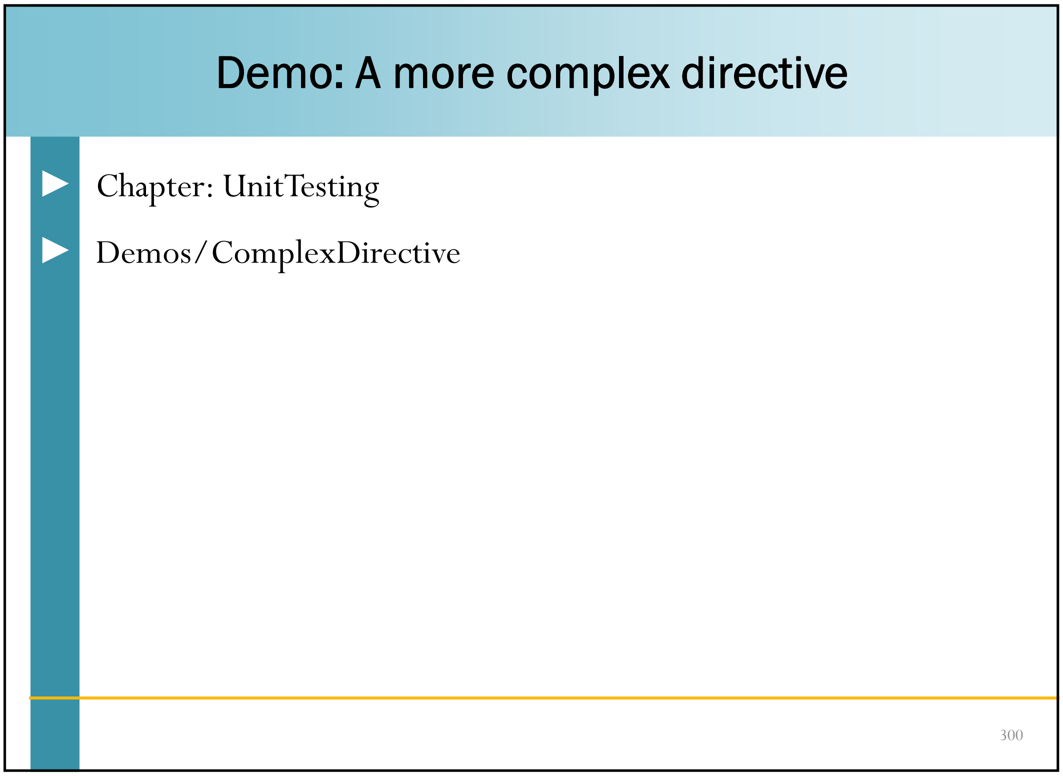
- Using a very simple directive for testing
- Chapter: UnitTesting
- Demos/BasicDirective



More complicated issues

- What if you need visibility into the scope object for the directive?
  - Instead of providing **`$rootScope`**, provide a **`$scope`** instance via a call to **`$rootScope.$new()`**
- What if the directive has a controller attached to it?
  - See above, mostly
  - Recall that when we work with controllers, our view into what is going on inside them is a mock **`$scope`** object, so do the same here
- If you are unit testing a directive that uses a partial template, consider using the node module **karma-ng-html2js** (available at GitHub) as a preprocessor, which will load and compile the templates for you

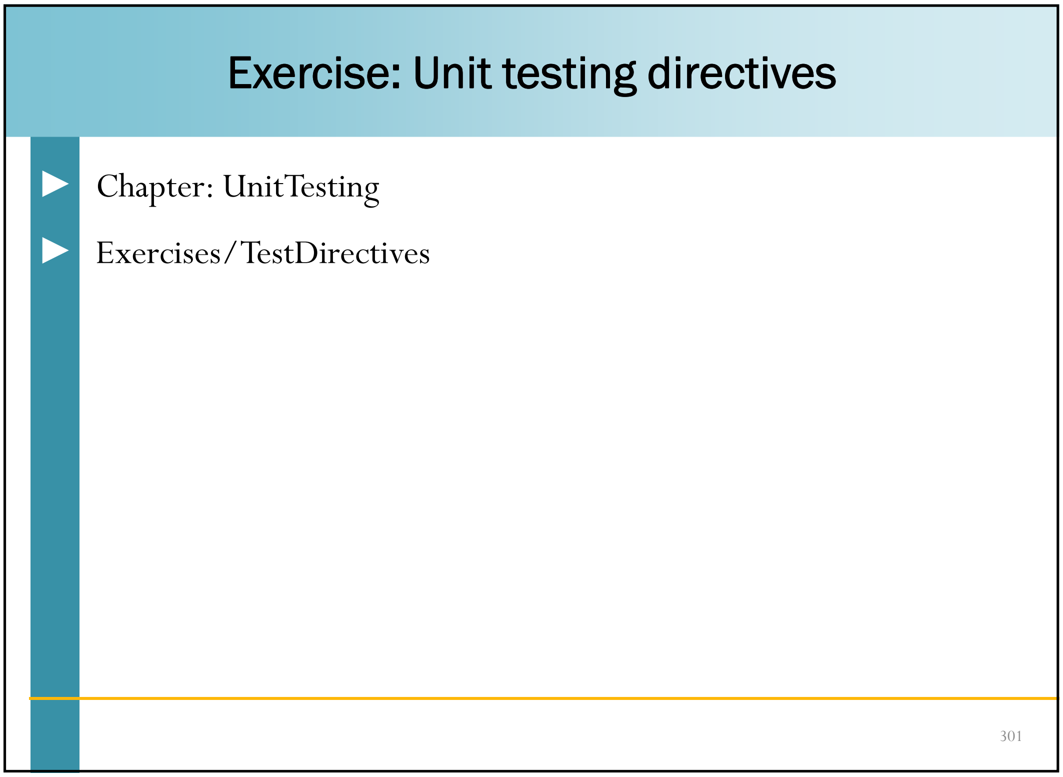
299



Demo: A more complex directive

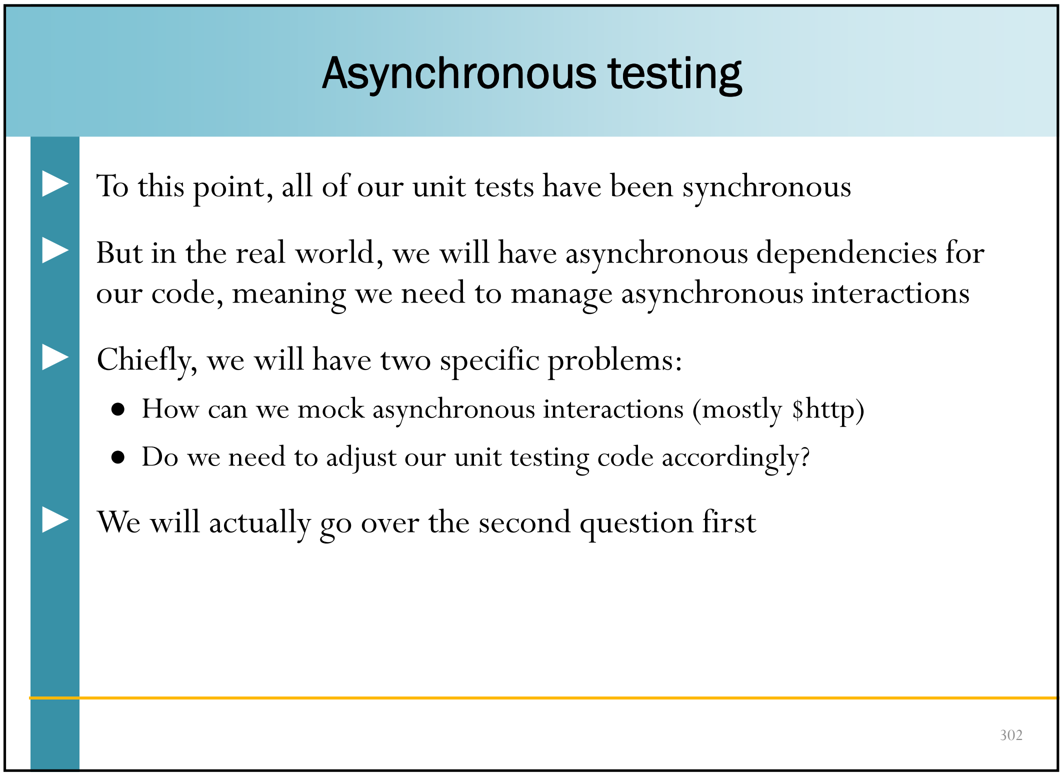
- Chapter: UnitTesting
- Demos/ComplexDirective

300



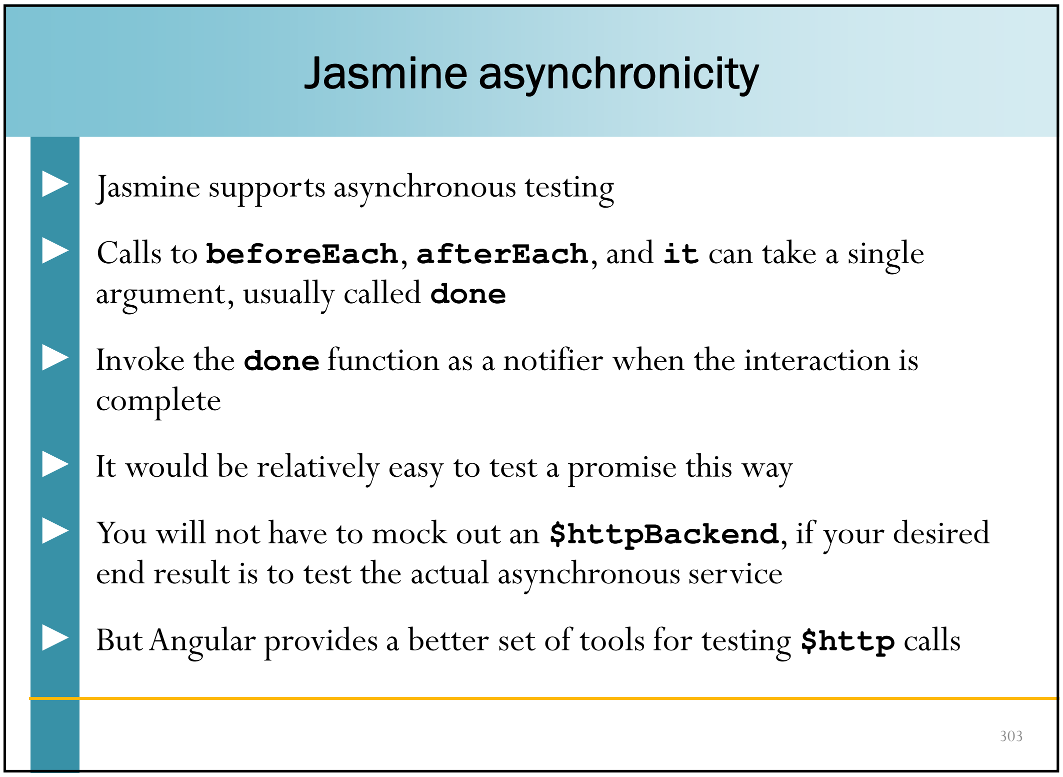
Exercise: Unit testing directives

- Chapter: UnitTesting
- Exercises/TestDirectives



Asynchronous testing

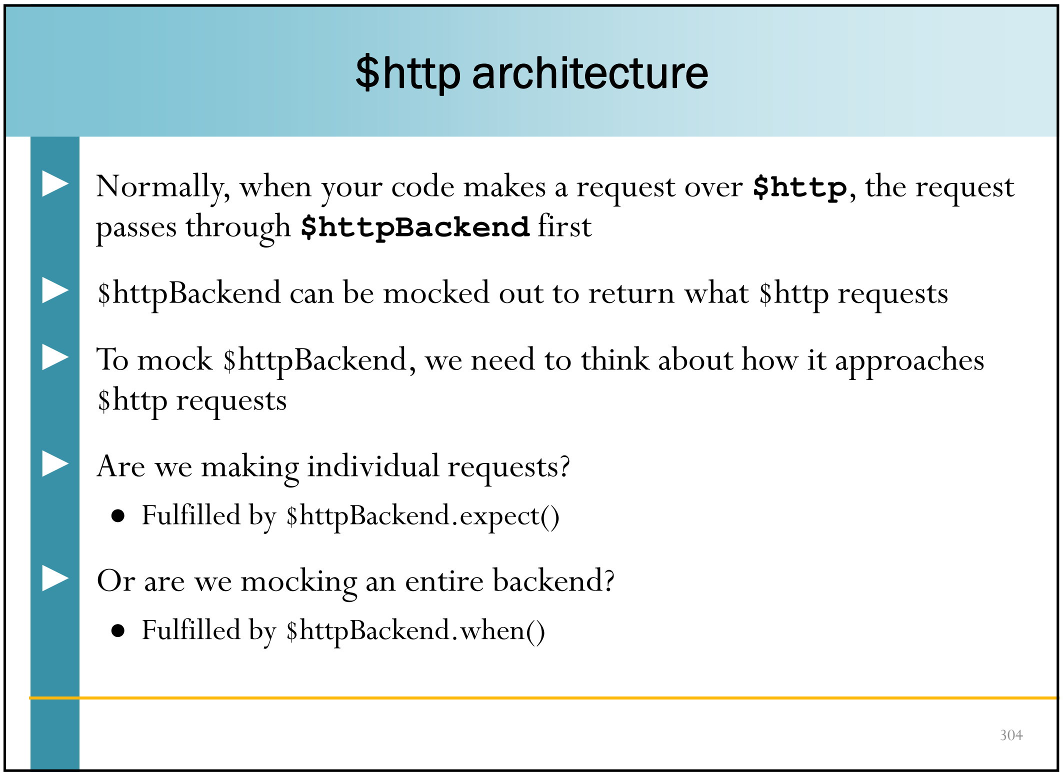
- To this point, all of our unit tests have been synchronous
- But in the real world, we will have asynchronous dependencies for our code, meaning we need to manage asynchronous interactions
- Chiefly, we will have two specific problems:
  - How can we mock asynchronous interactions (mostly $http)
  - Do we need to adjust our unit testing code accordingly?
- We will actually go over the second question first



Jasmine asynchronicity

- Jasmine supports asynchronous testing
- Calls to **`beforeEach`**, **`afterEach`**, and **`it`** can take a single argument, usually called **`done`**
- Invoke the **`done`** function as a notifier when the interaction is complete
- It would be relatively easy to test a promise this way
- You will not have to mock out an **`$httpBackend`**, if your desired end result is to test the actual asynchronous service
- But Angular provides a better set of tools for testing **`$http`** calls

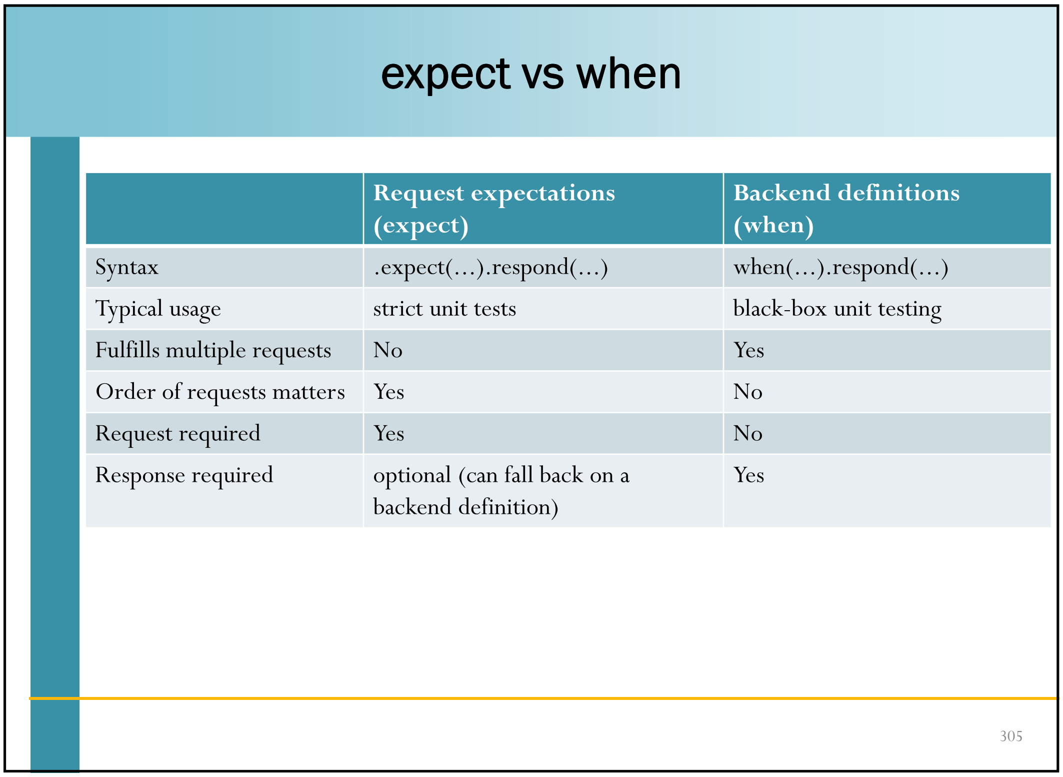
303



$http architecture

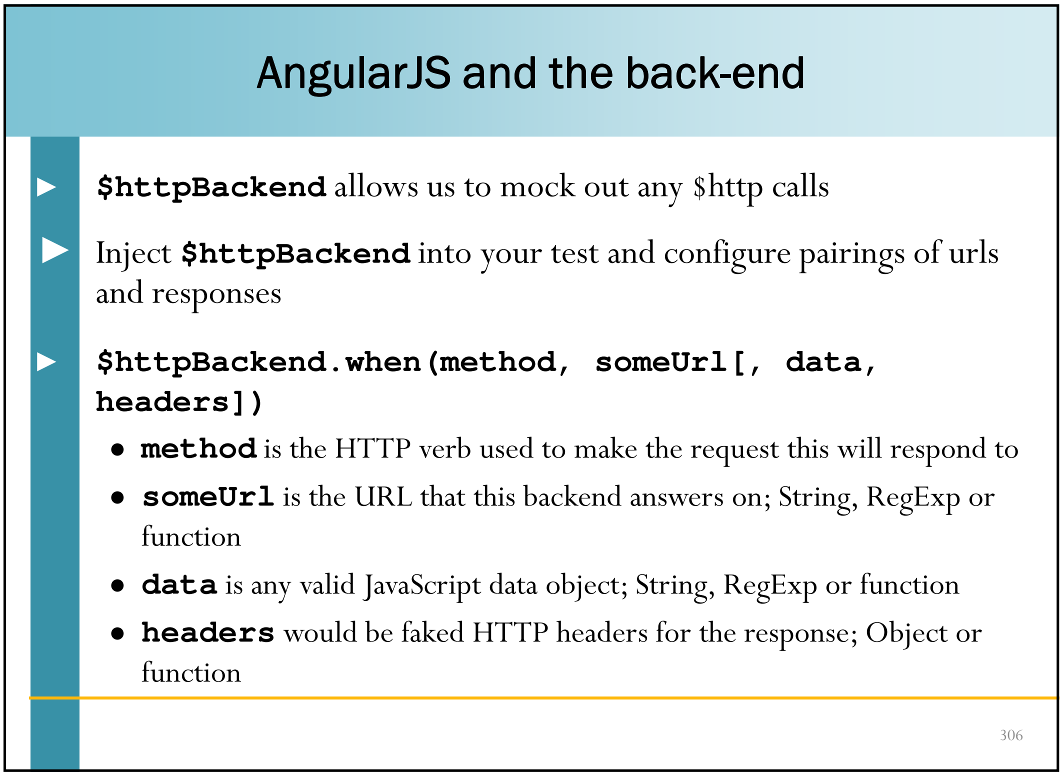
- Normally, when your code makes a request over **`$http`**, the request passes through **`$httpBackend`** first
- $httpBackend can be mocked out to return what $http requests
- To mock $httpBackend, we need to think about how it approaches $http requests
- Are we making individual requests?
  - Fulfilled by $httpBackend.expect()
- Or are we mocking an entire backend?
  - Fulfilled by $httpBackend.when()

304



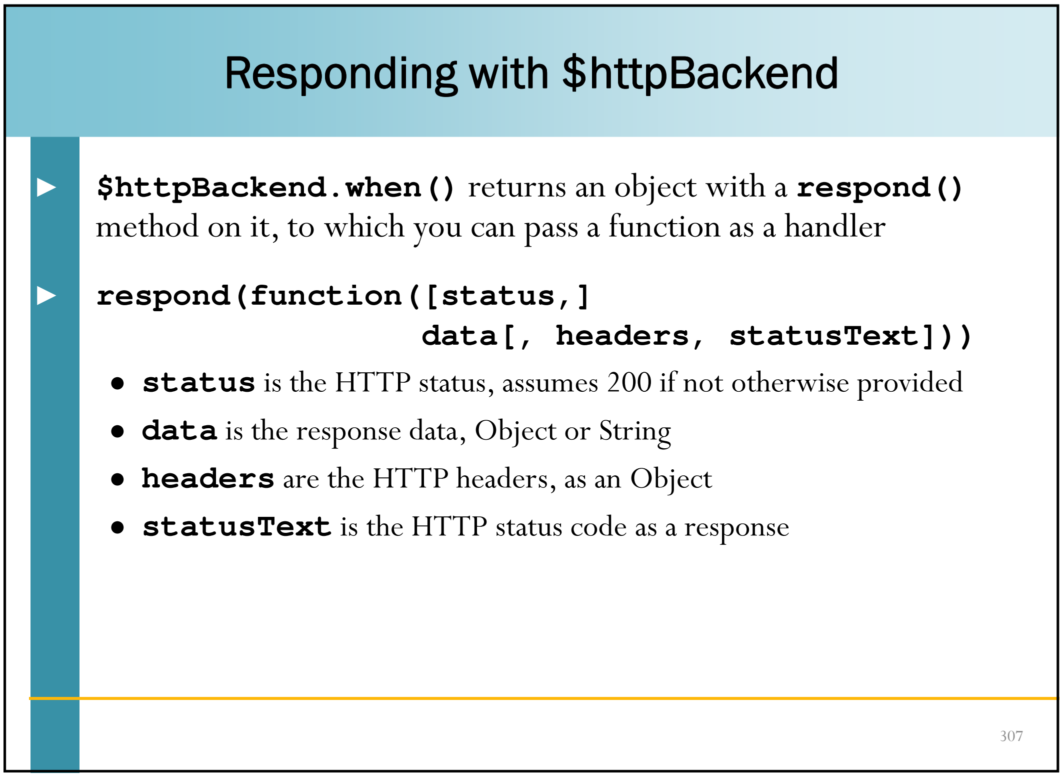
expect vs when

	Request expectations (expect)	Backend definitions (when)
Syntax	.expect(…).respond(…)	when(…).respond(…)
Typical usage	strict unit tests	black-box unit testing
Fulfills multiple requests	No	Yes
Order of requests matters	Yes	No
Request required	Yes	No
Response required	optional (can fall back on a backend definition)	Yes



AngularJS and the back-end

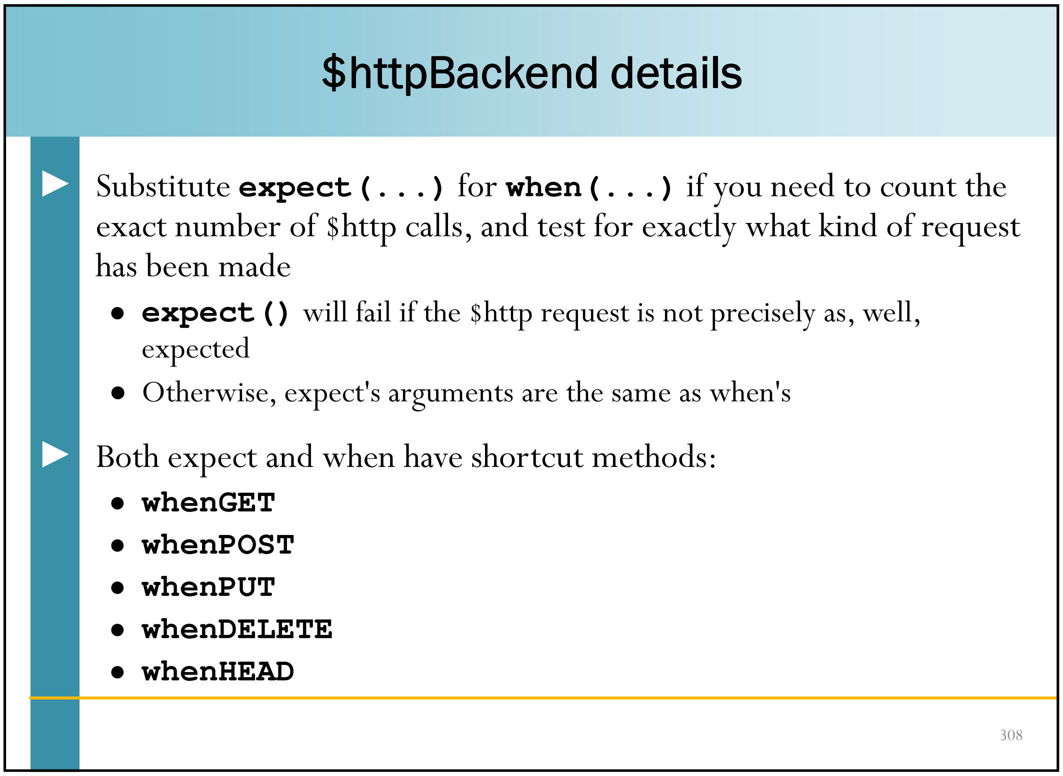
- **`$httpBackend`** allows us to mock out any $http calls
- Inject **`$httpBackend`** into your test and configure pairings of urls and responses
- **`$httpBackend.when(method, someUrl[, data, headers])`**
  - **`method`** is the HTTP verb used to make the request this will respond to
  - **`someUrl`** is the URL that this backend answers on; String, RegExp or function
  - **`data`** is any valid JavaScript data object; String, RegExp or function
  - **`headers`** would be faked HTTP headers for the response; Object or function



Responding with $httpBackend

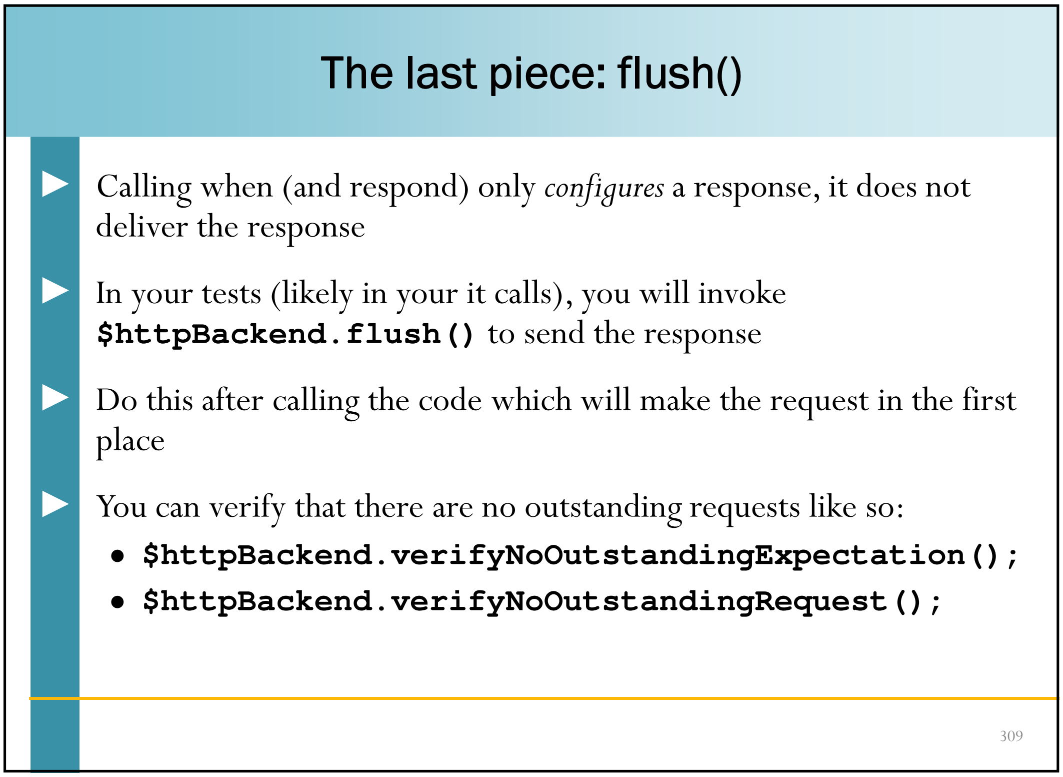
- **`$httpBackend.when()`** returns an object with a **`respond()`** method on it, to which you can pass a function as a handler

-
  ```
  respond(function([status,]
                   data[, headers, statusText]))
  ```
  - **`status`** is the HTTP status, assumes 200 if not otherwise provided
  - **`data`** is the response data, Object or String
  - **`headers`** are the HTTP headers, as an Object
  - **`statusText`** is the HTTP status code as a response



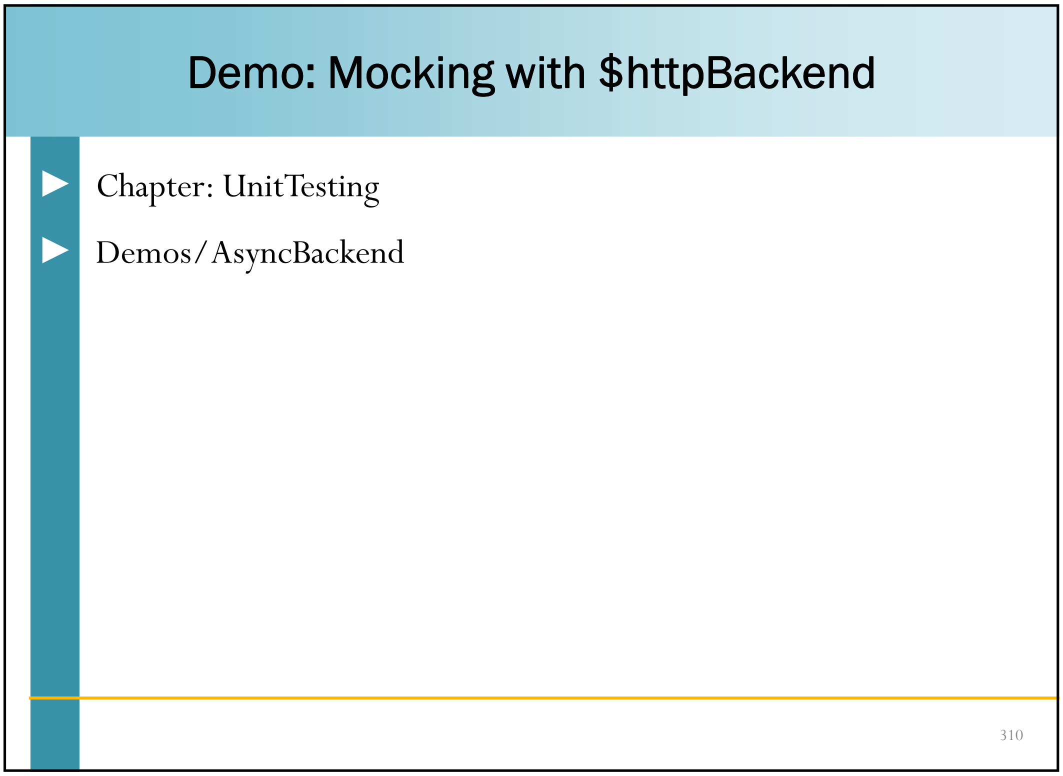
$httpBackend details

- Substitute **`expect(...)`** for **`when(...)`** if you need to count the exact number of $http calls, and test for exactly what kind of request has been made
  - **`expect()`** will fail if the $http request is not precisely as, well, expected
  - Otherwise, expect's arguments are the same as when's
- Both expect and when have shortcut methods:
  - **`whenGET`**
  - **`whenPOST`**
  - **`whenPUT`**
  - **`whenDELETE`**
  - **`whenHEAD`**



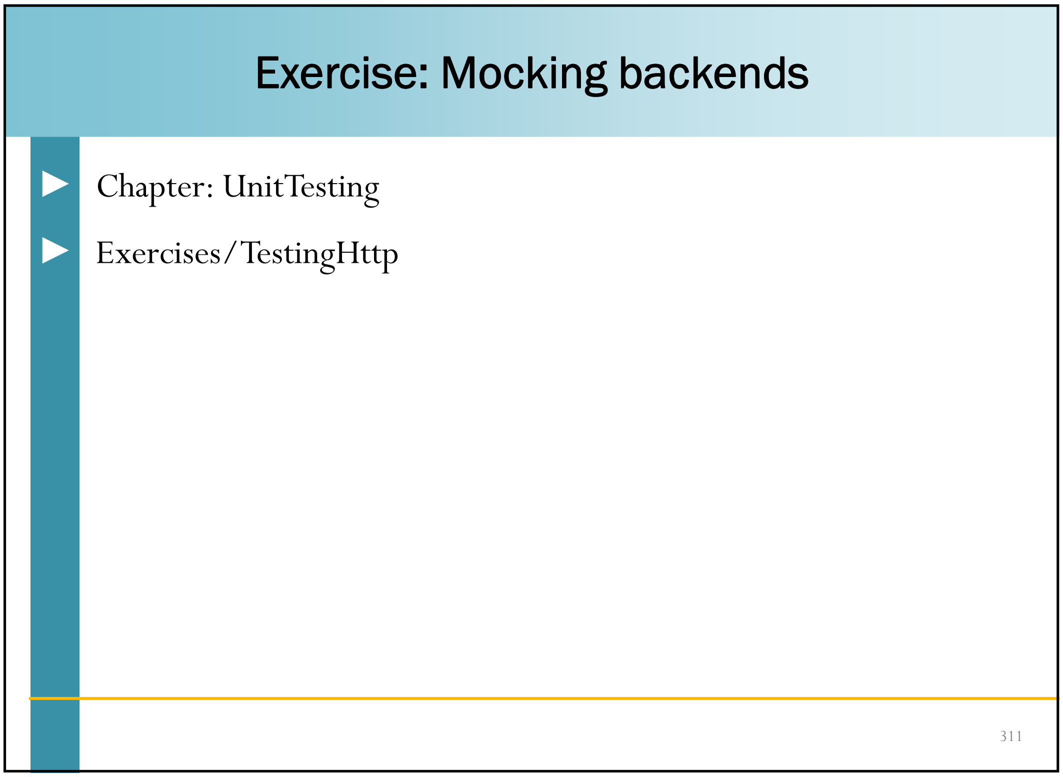
The last piece: flush()

- Calling when (and respond) only *configures* a response, it does not deliver the response
- In your tests (likely in your it calls), you will invoke **`$httpBackend.flush()`** to send the response
- Do this after calling the code which will make the request in the first place
- You can verify that there are no outstanding requests like so:
  - **`$httpBackend.verifyNoOutstandingExpectation();`**
  - **`$httpBackend.verifyNoOutstandingRequest();`**



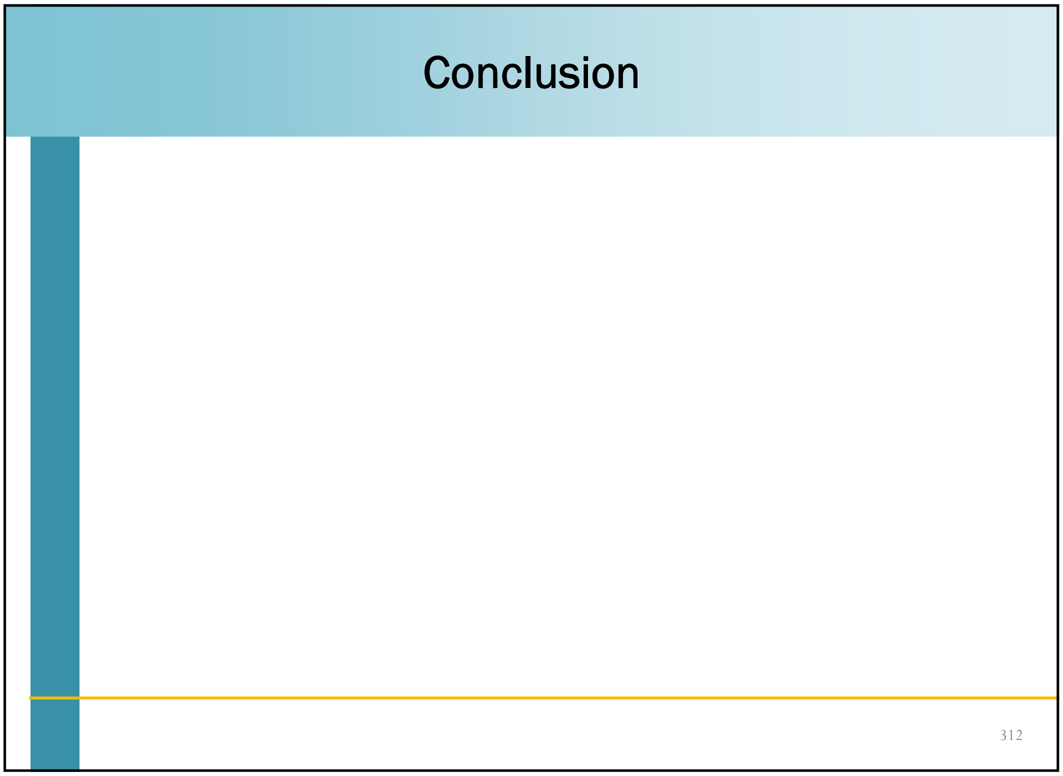
Demo: Mocking with $httpBackend

- Chapter: UnitTesting
- Demos/AsyncBackend

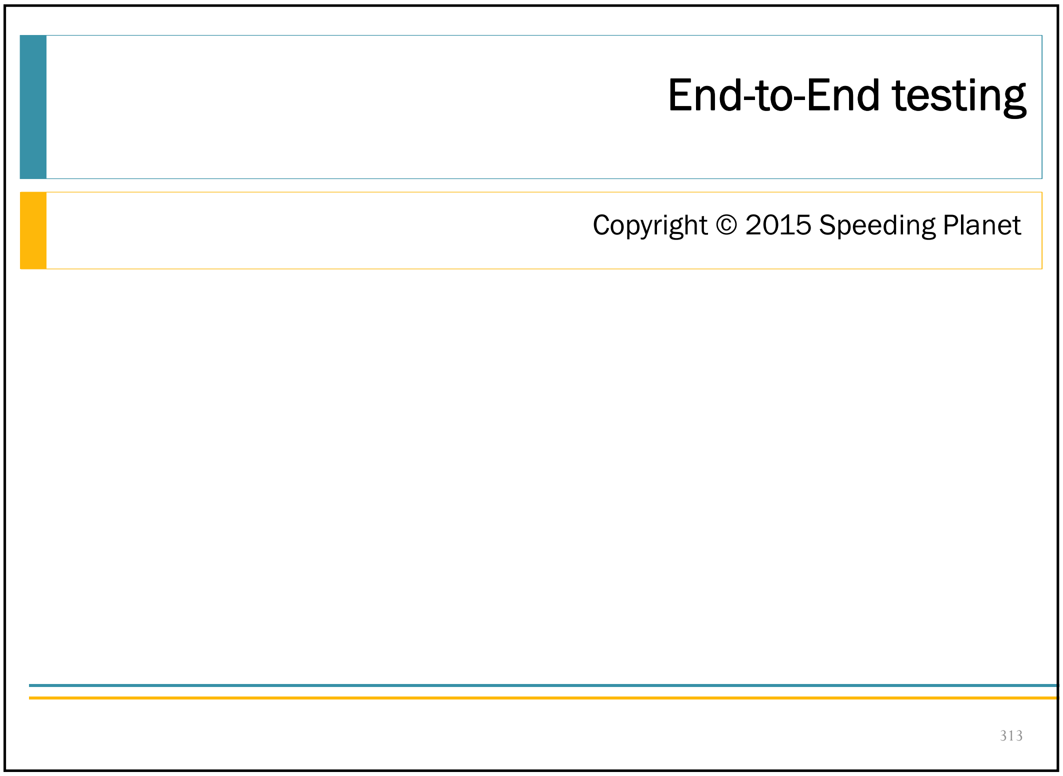


Exercise: Mocking backends

- Chapter: UnitTesting
- Exercises/TestingHttp

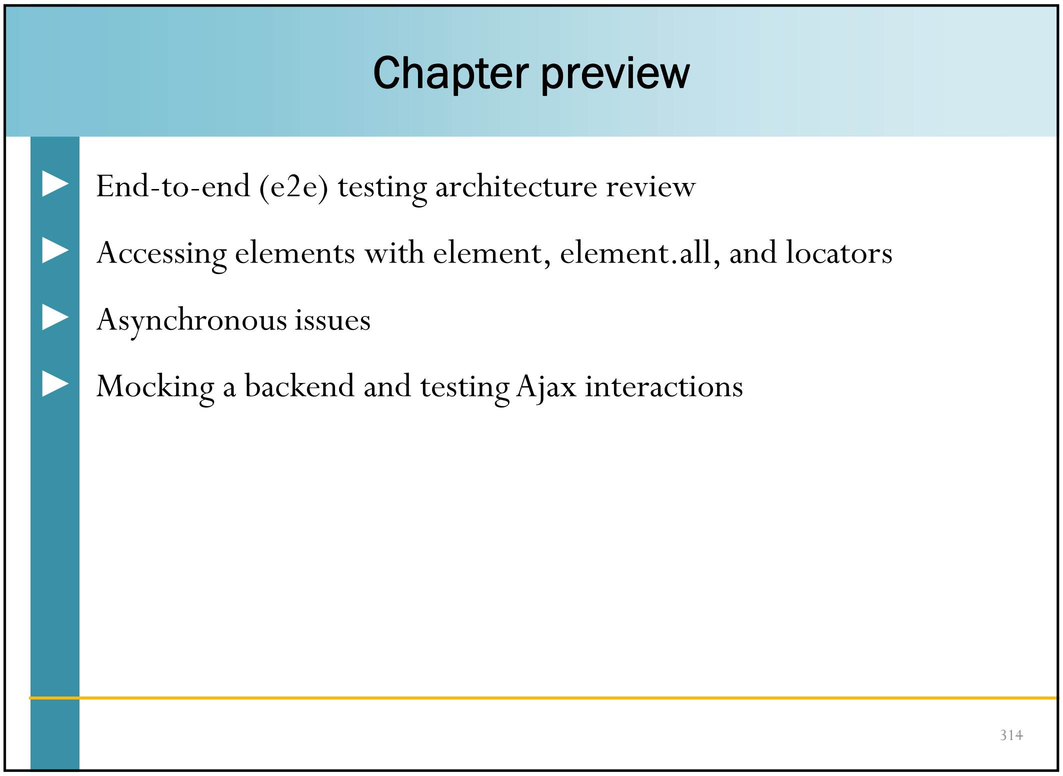


Conclusion



End-to-End testing

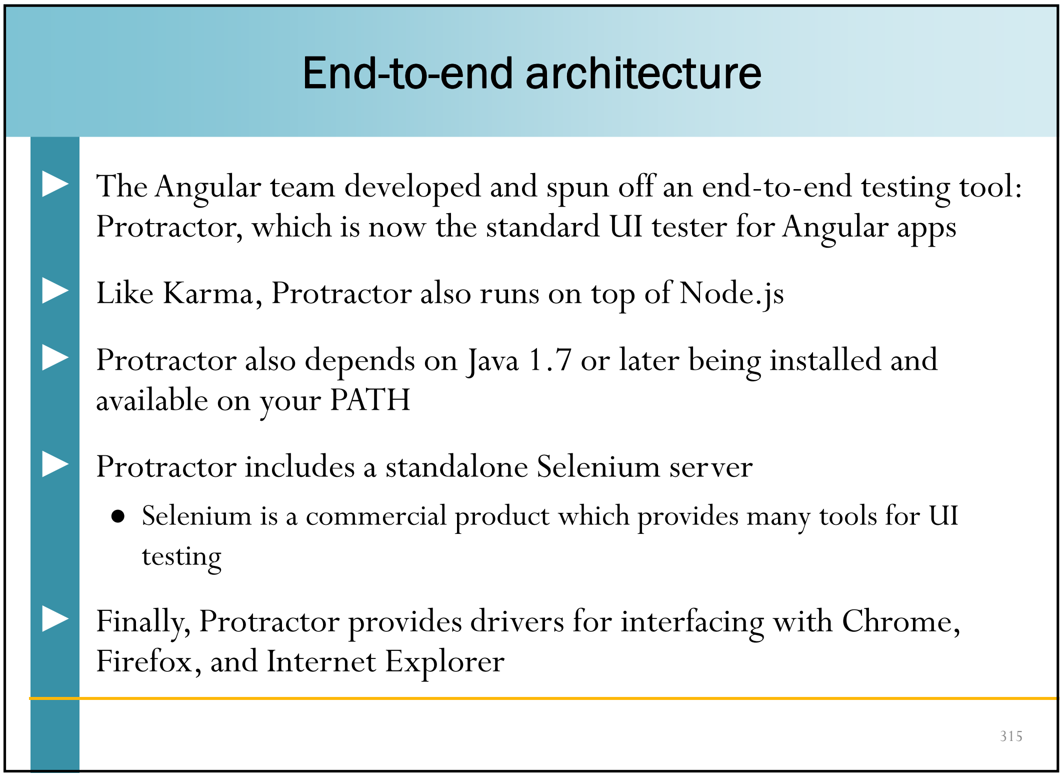
313



Chapter preview

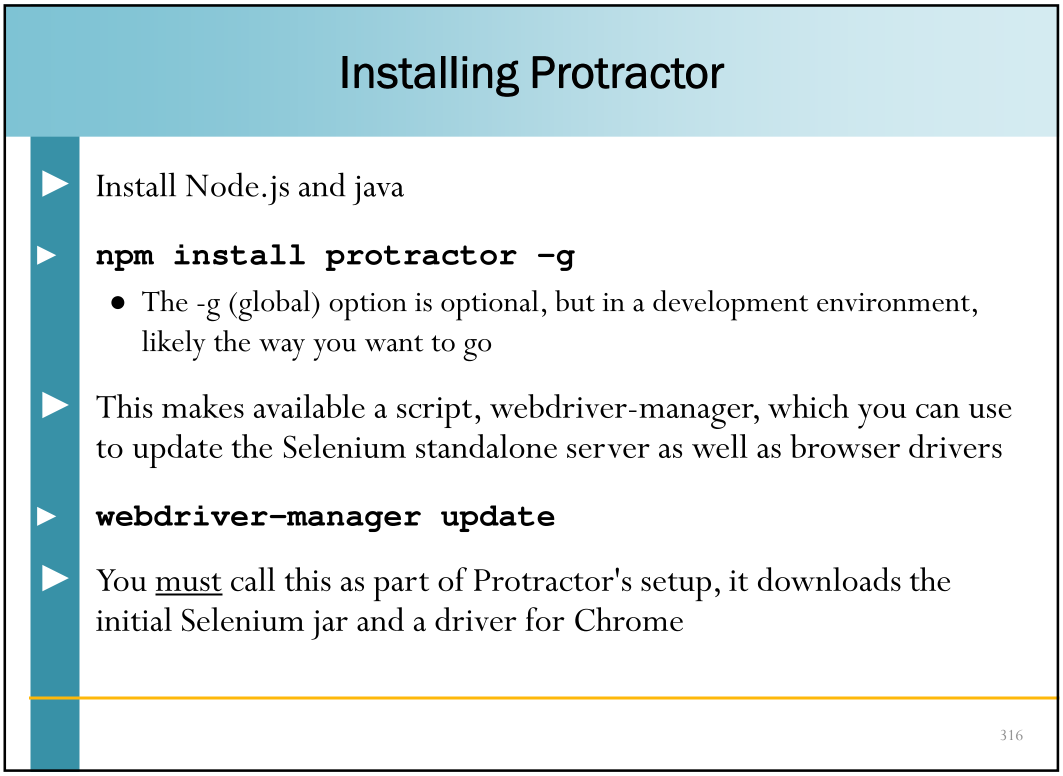
- End-to-end (e2e) testing architecture review
- Accessing elements with element, element.all, and locators
- Asynchronous issues
- Mocking a backend and testing Ajax interactions

314



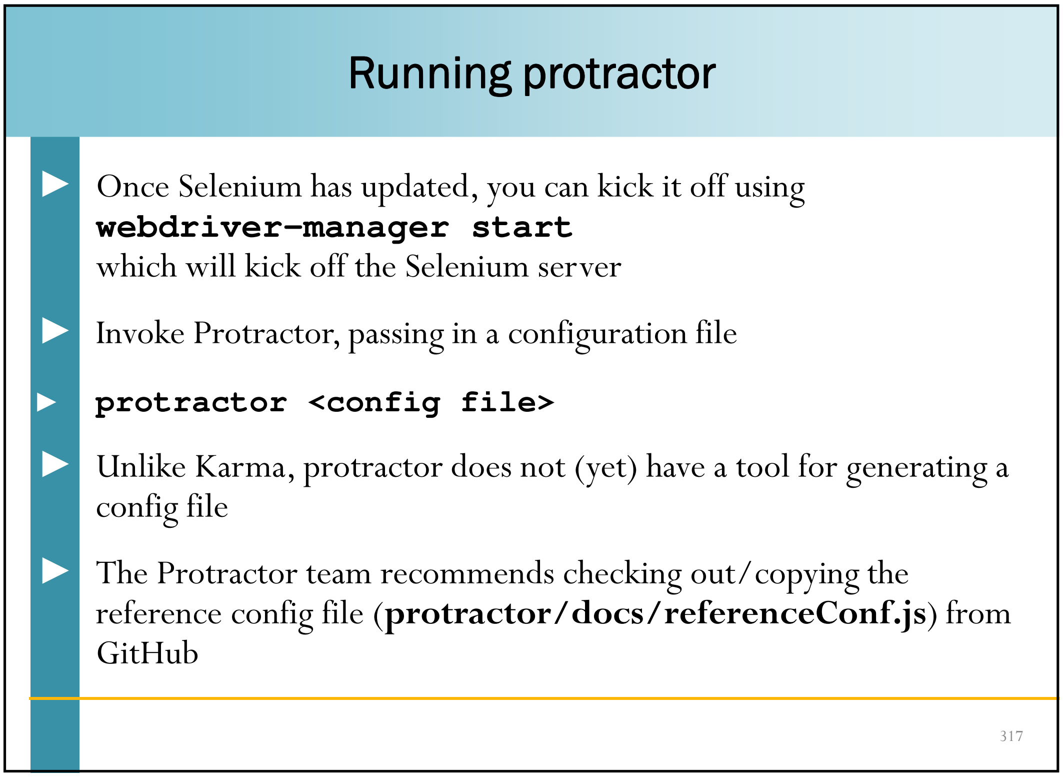
End-to-end architecture

- The Angular team developed and spun off an end-to-end testing tool: Protractor, which is now the standard UI tester for Angular apps
- Like Karma, Protractor also runs on top of Node.js
- Protractor also depends on Java 1.7 or later being installed and available on your PATH
- Protractor includes a standalone Selenium server
  - Selenium is a commercial product which provides many tools for UI testing
- Finally, Protractor provides drivers for interfacing with Chrome, Firefox, and Internet Explorer



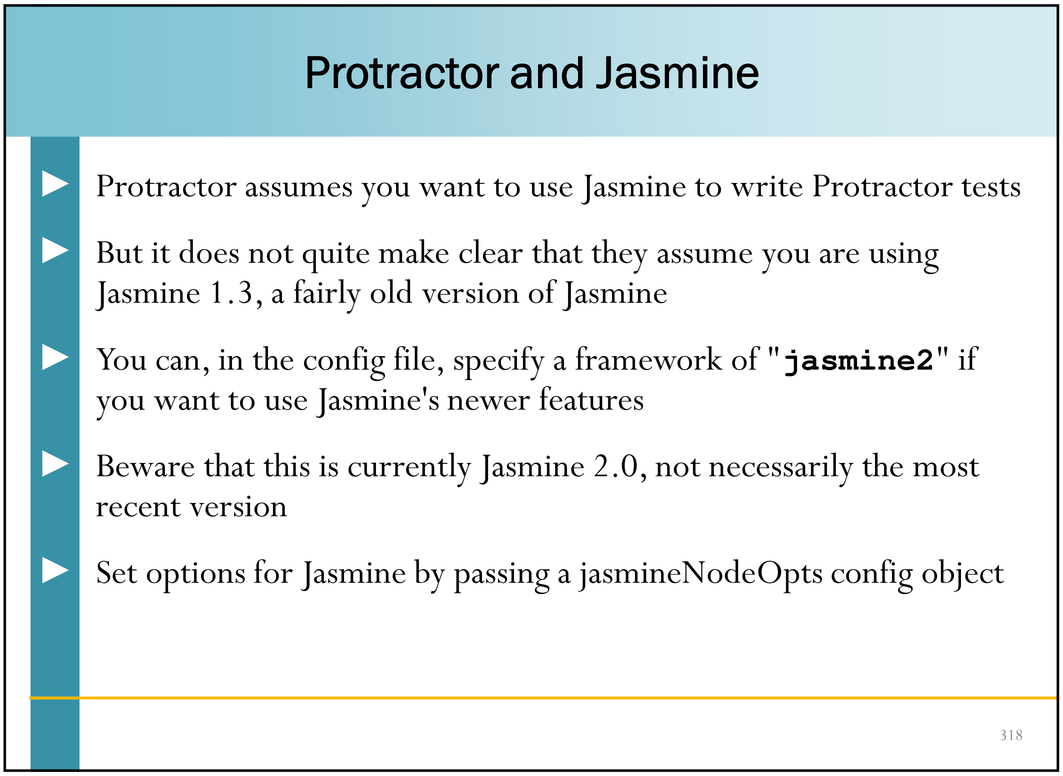
Installing Protractor

- Install Node.js and java
- **`npm install protractor -g`**
  - The -g (global) option is optional, but in a development environment, likely the way you want to go
- This makes available a script, webdriver-manager, which you can use to update the Selenium standalone server as well as browser drivers
- **`webdriver-manager update`**
- You must call this as part of Protractor's setup, it downloads the initial Selenium jar and a driver for Chrome



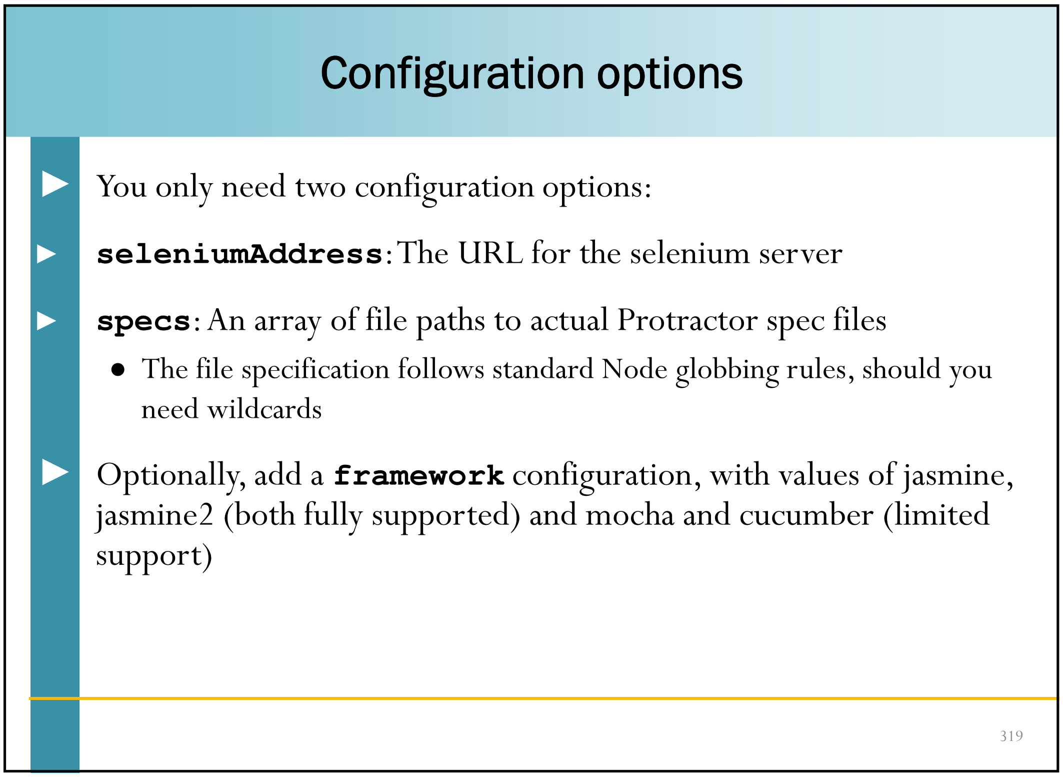
Running protractor

- Once Selenium has updated, you can kick it off using
  `webdriver-manager start`
  which will kick off the Selenium server
- Invoke Protractor, passing in a configuration file
- **`protractor <config file>`**
- Unlike Karma, protractor does not (yet) have a tool for generating a config file
- The Protractor team recommends checking out/copying the reference config file (**protractor/docs/referenceConf.js**) from GitHub



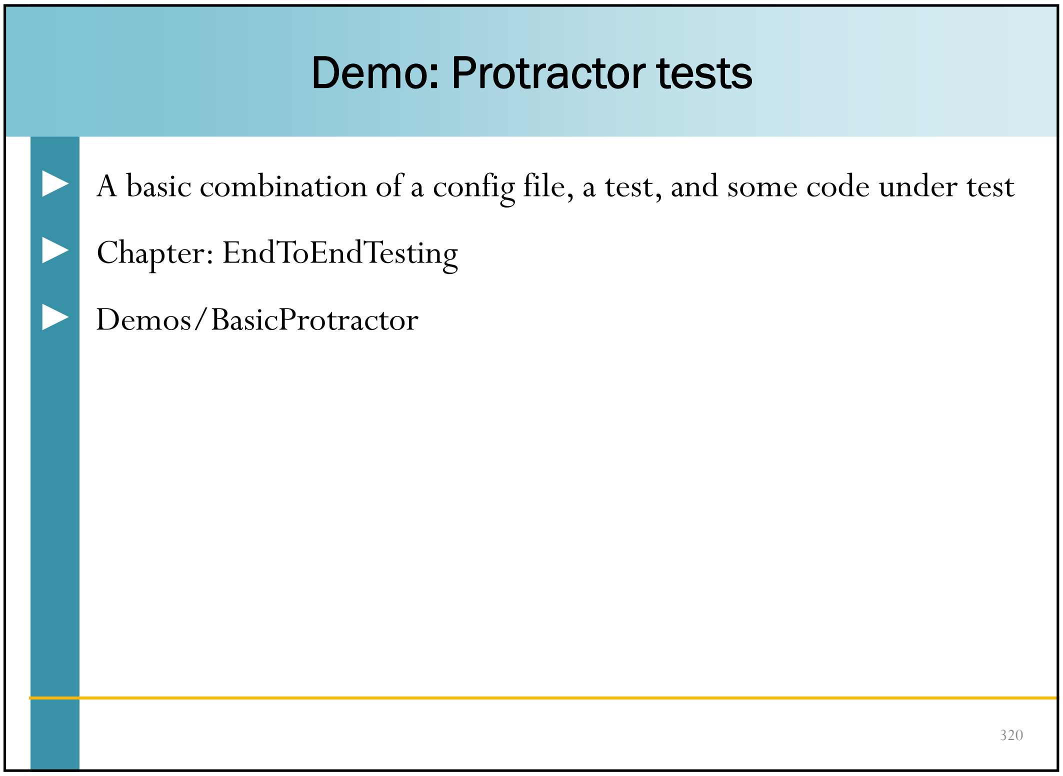
Protractor and Jasmine

- Protractor assumes you want to use Jasmine to write Protractor tests
- But it does not quite make clear that they assume you are using Jasmine 1.3, a fairly old version of Jasmine
- You can, in the config file, specify a framework of "**`jasmine2`**" if you want to use Jasmine's newer features
- Beware that this is currently Jasmine 2.0, not necessarily the most recent version
- Set options for Jasmine by passing a jasmineNodeOpts config object



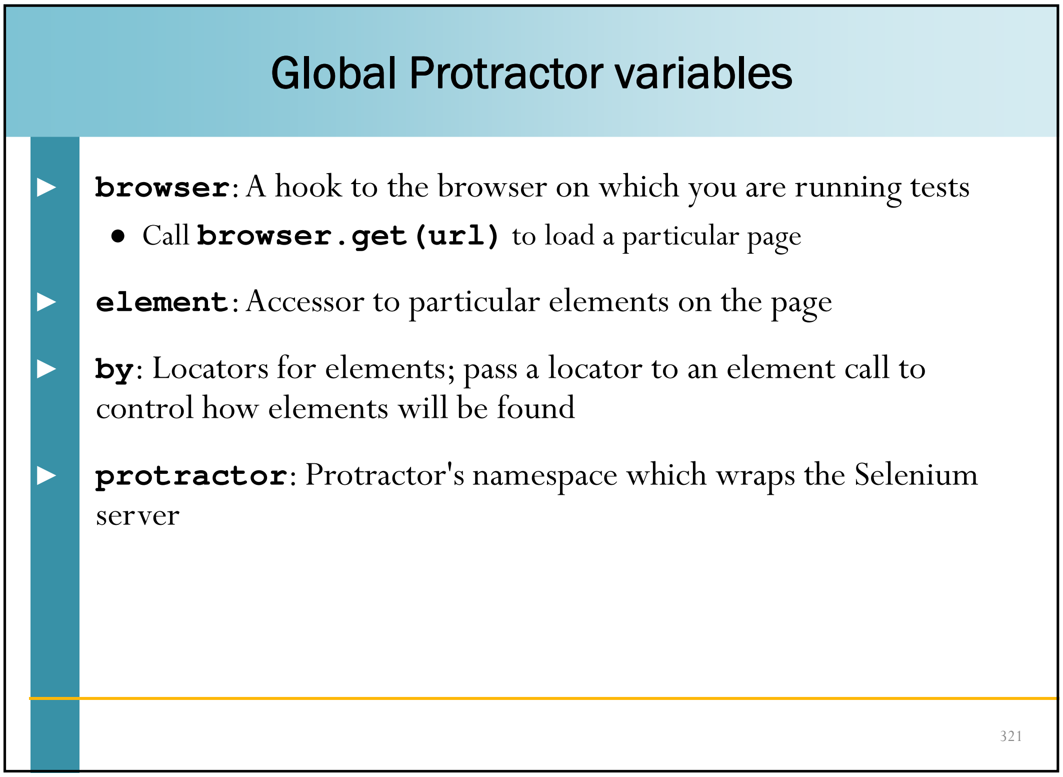
Configuration options

- You only need two configuration options:
- `seleniumAddress`: The URL for the selenium server
- `specs`: An array of file paths to actual Protractor spec files
  - The file specification follows standard Node globbing rules, should you need wildcards
- Optionally, add a `framework` configuration, with values of jasmine, jasmine2 (both fully supported) and mocha and cucumber (limited support)



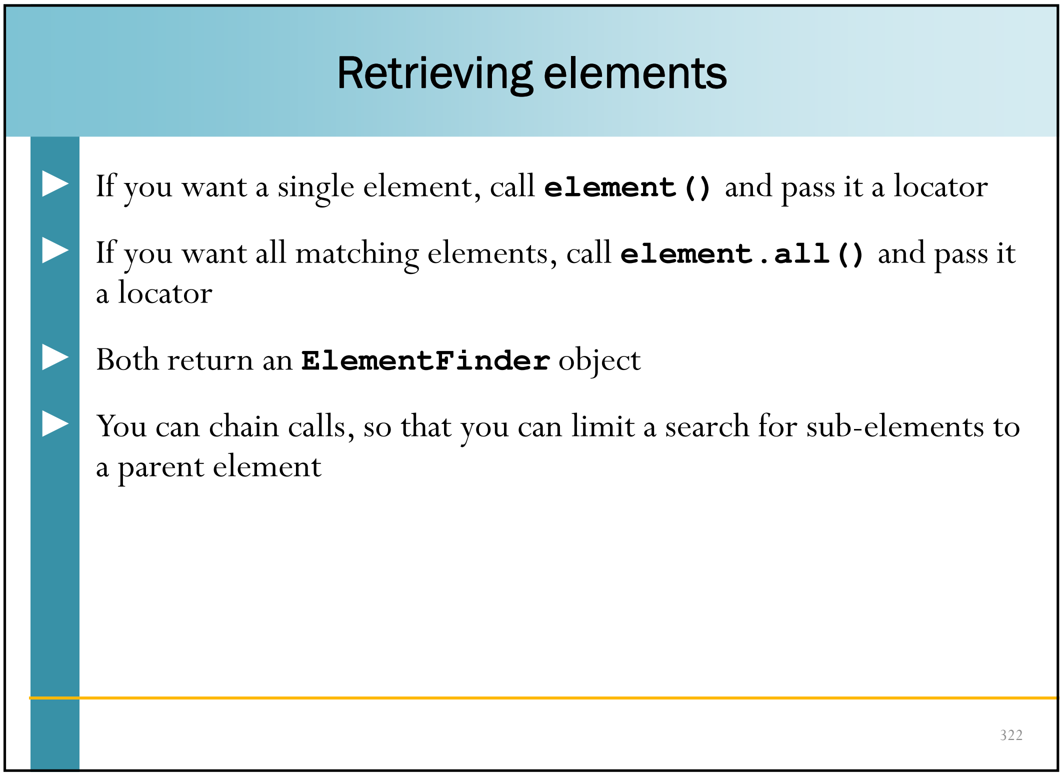
Demo: Protractor tests

- A basic combination of a config file, a test, and some code under test
- Chapter: EndToEndTesting
- Demos/BasicProtractor



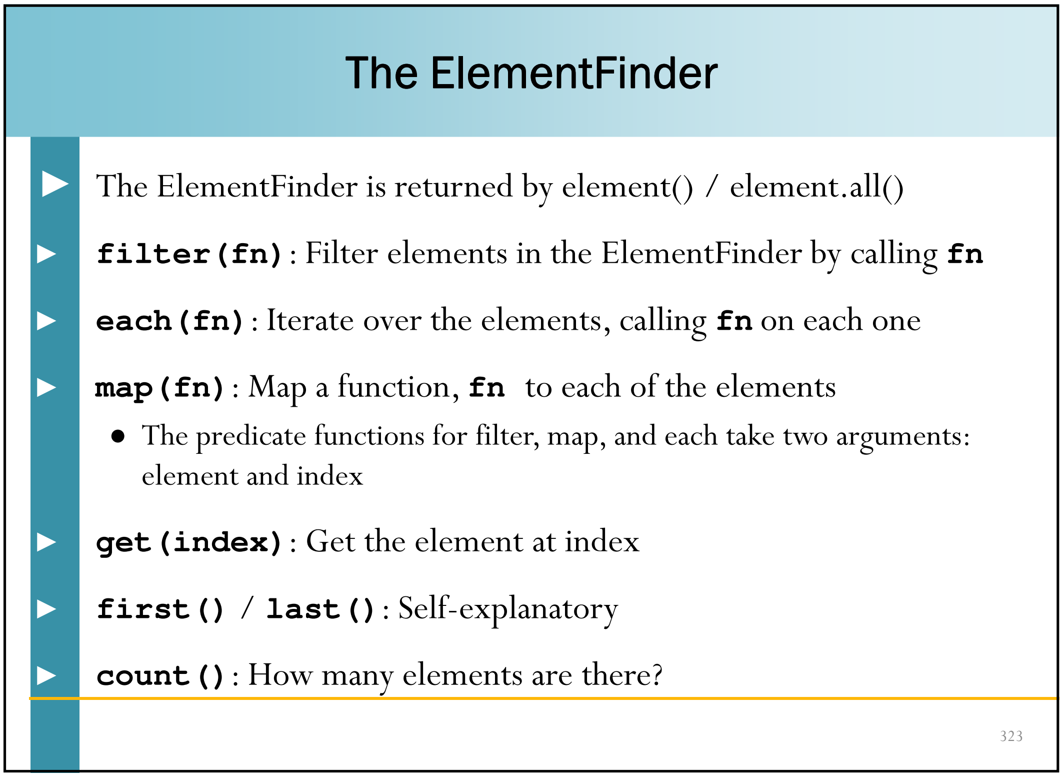
Global Protractor variables

- **`browser`**: A hook to the browser on which you are running tests
  - Call **`browser.get(url)`** to load a particular page
- **`element`**: Accessor to particular elements on the page
- **`by`**: Locators for elements; pass a locator to an element call to control how elements will be found
- **`protractor`**: Protractor's namespace which wraps the Selenium server



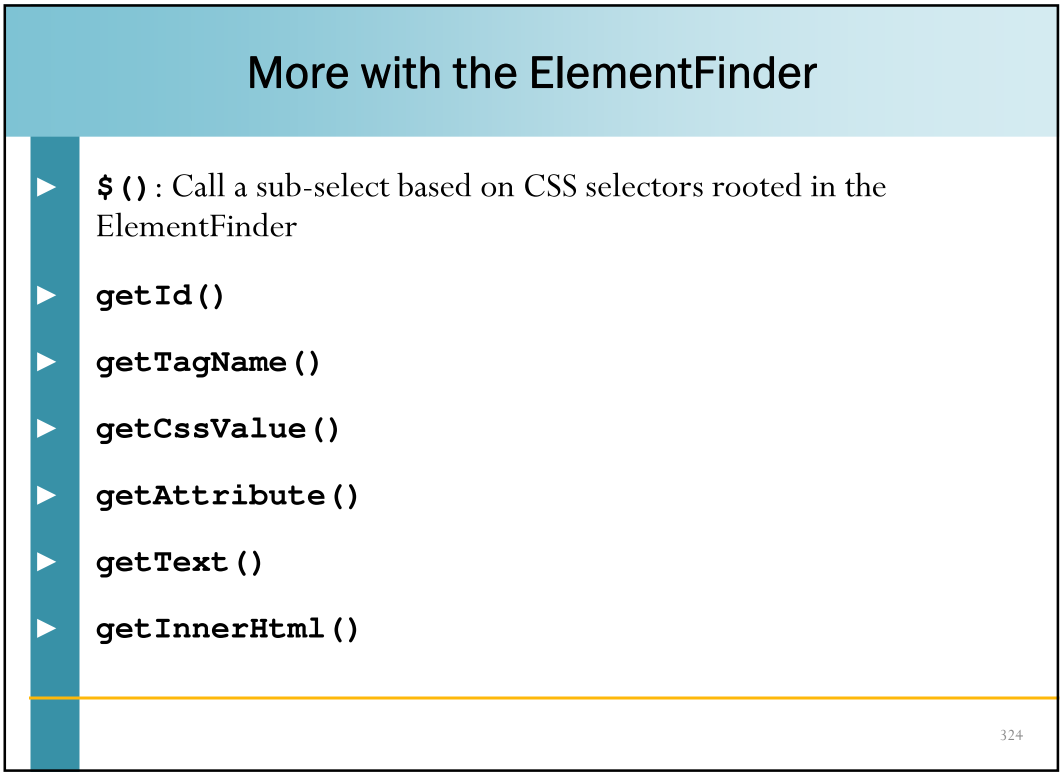
Retrieving elements

- If you want a single element, call **`element()`** and pass it a locator
- If you want all matching elements, call **`element.all()`** and pass it a locator
- Both return an **`ElementFinder`** object
- You can chain calls, so that you can limit a search for sub-elements to a parent element



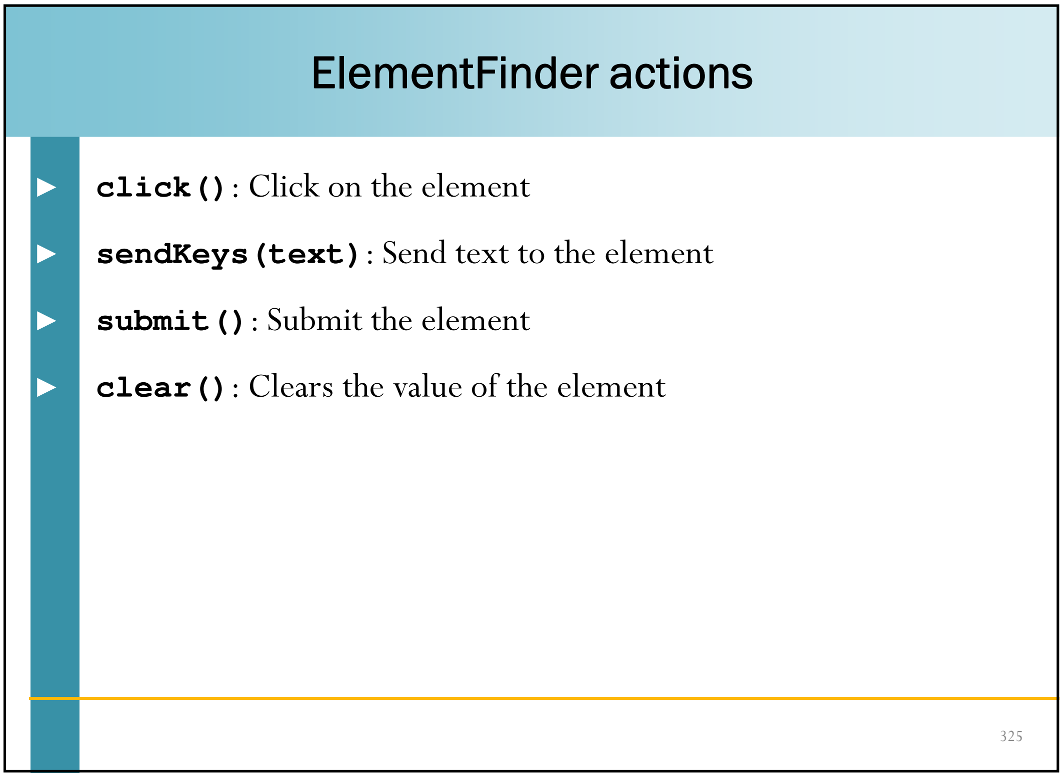
The ElementFinder

- The ElementFinder is returned by element() / element.all()
- **`filter(fn)`** : Filter elements in the ElementFinder by calling **`fn`**
- **`each(fn)`** : Iterate over the elements, calling **`fn`** on each one
- **`map(fn)`** : Map a function, **`fn`** to each of the elements
  - The predicate functions for filter, map, and each take two arguments: element and index
- **`get(index)`** : Get the element at index
- **`first()`** / **`last()`** : Self-explanatory
- **`count()`** : How many elements are there?



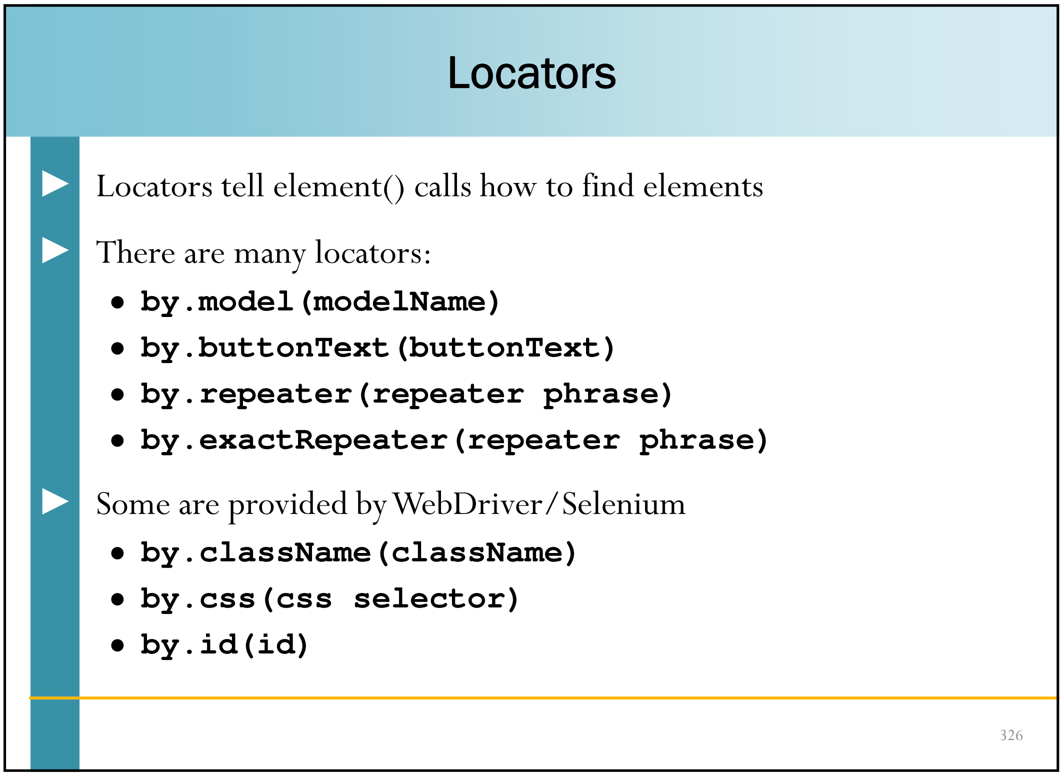
More with the ElementFinder

- **`$()`** : Call a sub-select based on CSS selectors rooted in the ElementFinder
- **`getId()`**
- **`getTagName()`**
- **`getCssValue()`**
- **`getAttribute()`**
- **`getText()`**
- **`getInnerHtml()`**



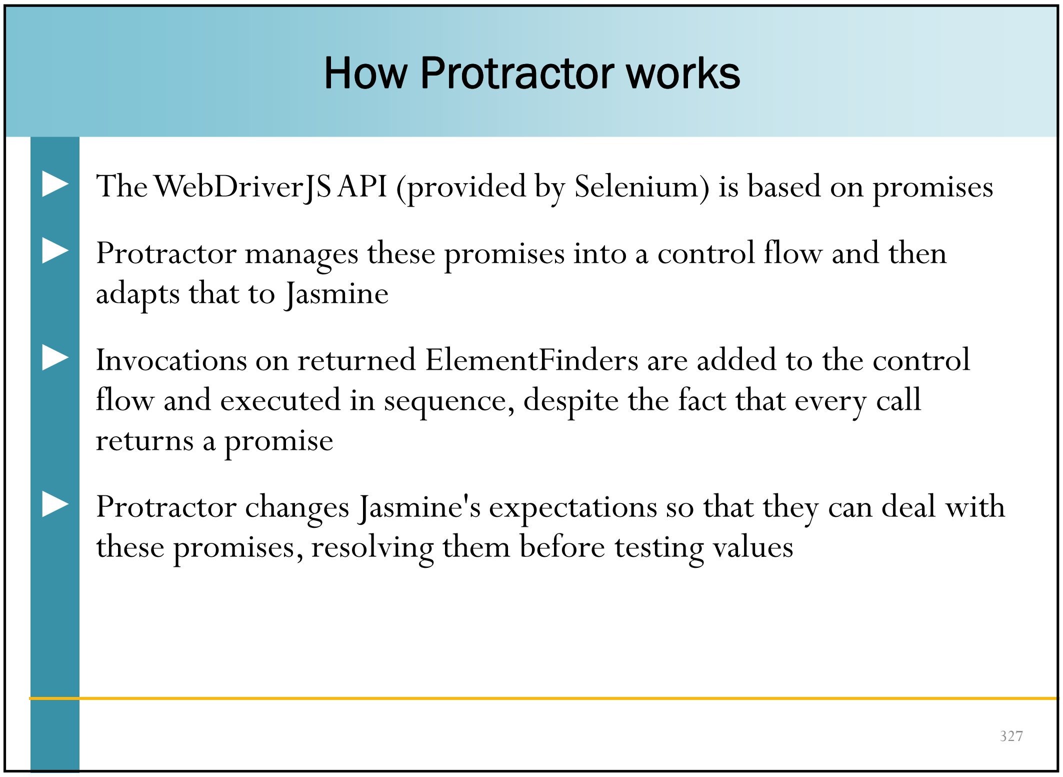
ElementFinder actions

- **`click()`** : Click on the element
- **`sendKeys(text)`** : Send text to the element
- **`submit()`** : Submit the element
- **`clear()`** : Clears the value of the element



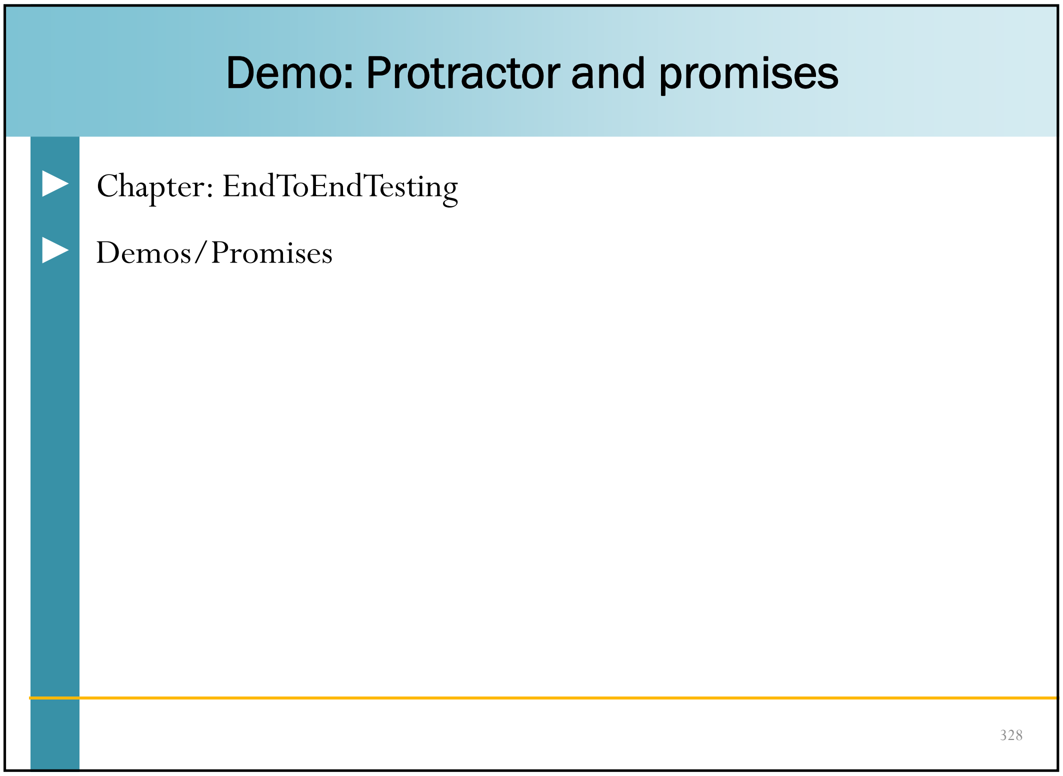
Locators

- Locators tell element() calls how to find elements
- There are many locators:
  - **`by.model(modelName)`**
  - **`by.buttonText(buttonText)`**
  - **`by.repeater(repeater phrase)`**
  - **`by.exactRepeater(repeater phrase)`**
- Some are provided by WebDriver/Selenium
  - **`by.className(className)`**
  - **`by.css(css selector)`**
  - **`by.id(id)`**



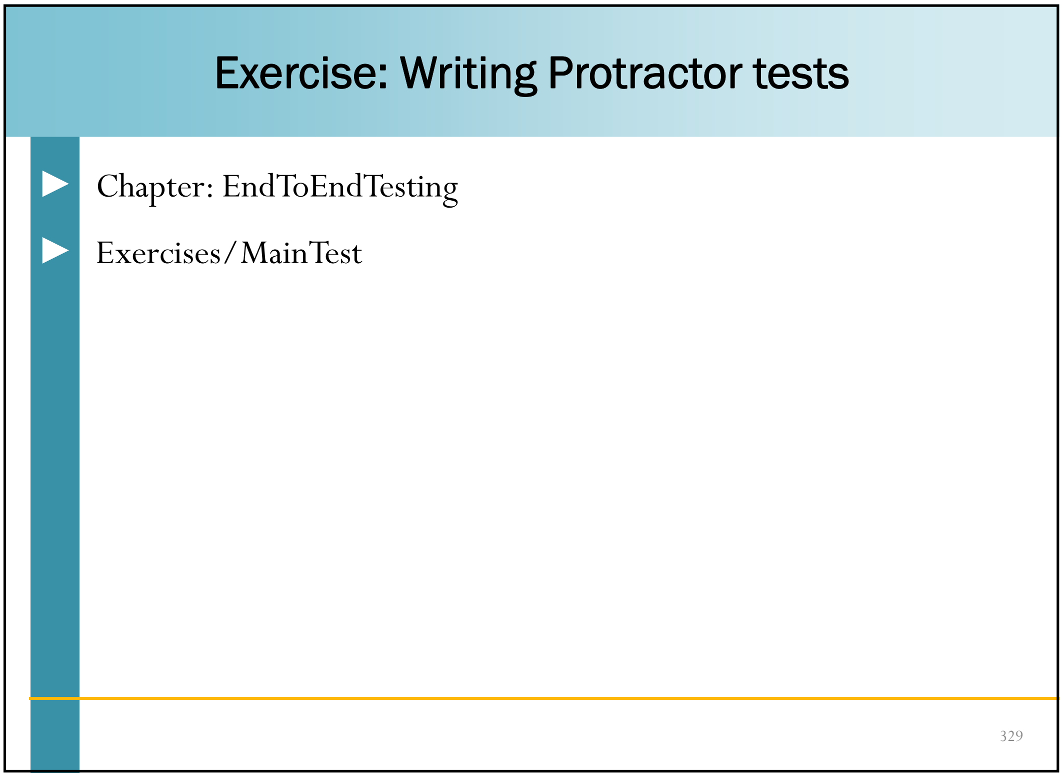
How Protractor works

- The WebDriverJS API (provided by Selenium) is based on promises
- Protractor manages these promises into a control flow and then adapts that to Jasmine
- Invocations on returned ElementFinders are added to the control flow and executed in sequence, despite the fact that every call returns a promise
- Protractor changes Jasmine's expectations so that they can deal with these promises, resolving them before testing values



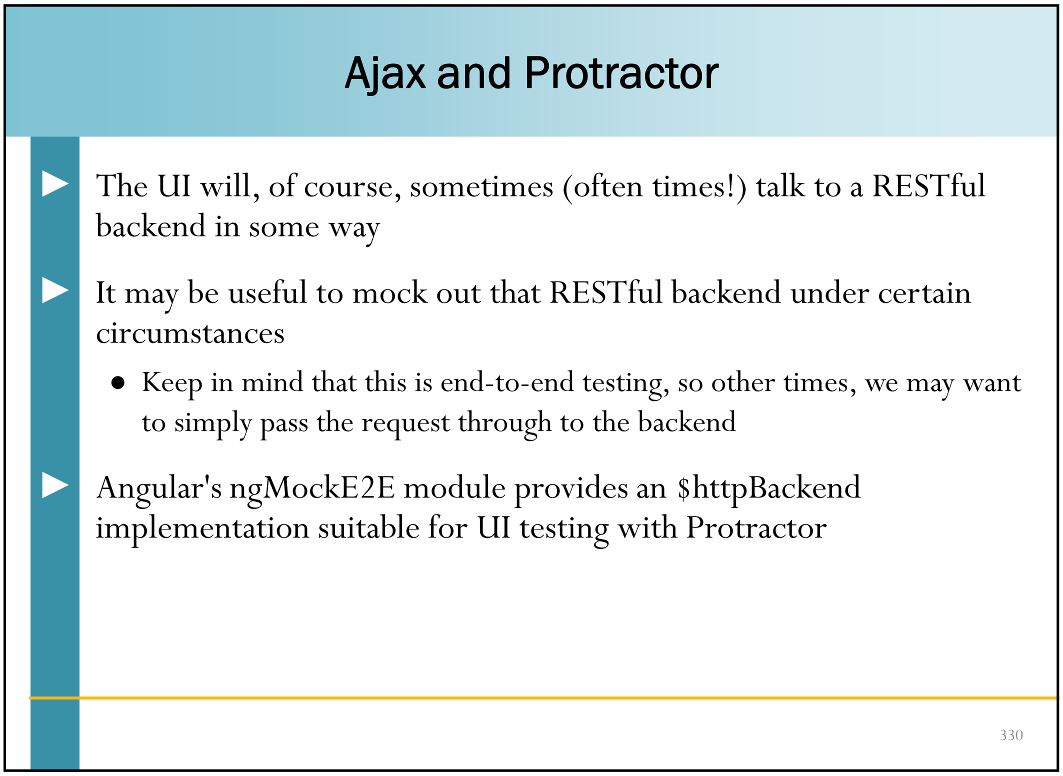
Demo: Protractor and promises

- Chapter: EndToEndTesting
- Demos/Promises



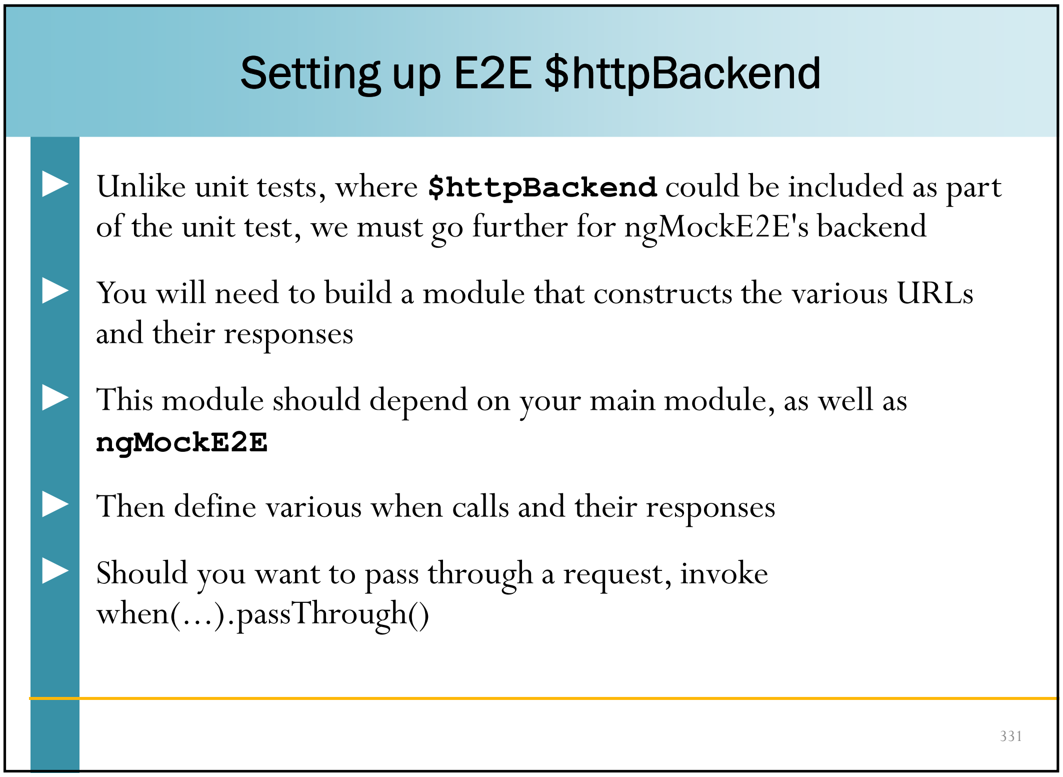
Exercise: Writing Protractor tests

- Chapter: EndToEndTesting
- Exercises/MainTest



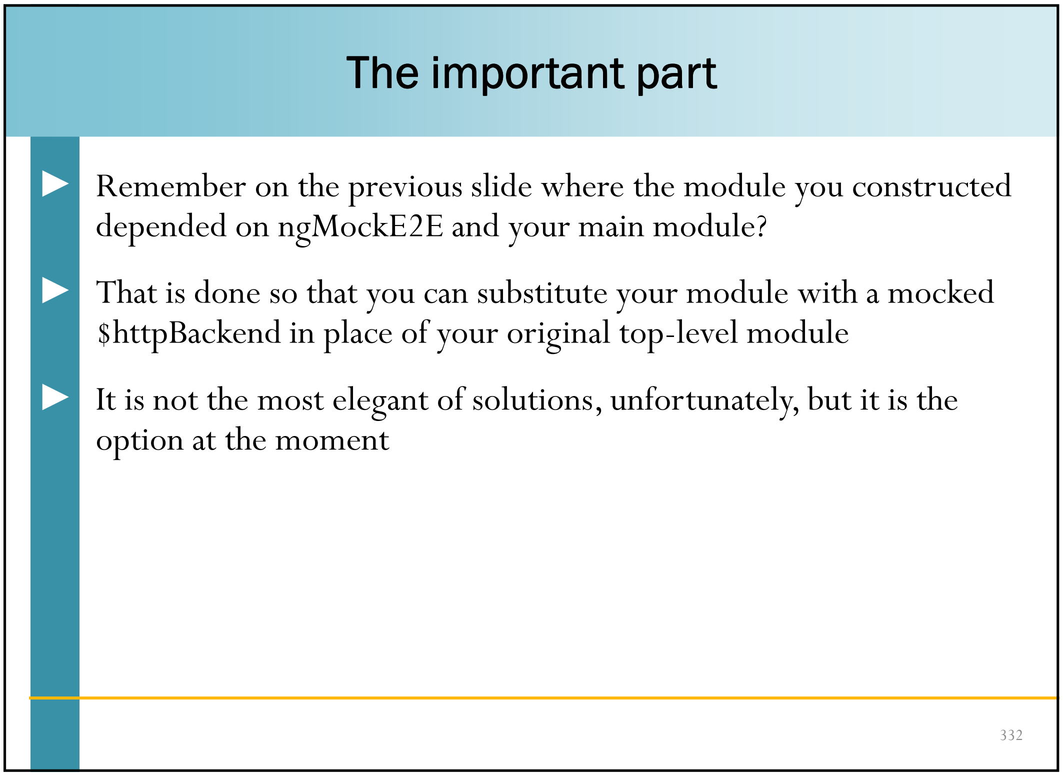
Ajax and Protractor

- The UI will, of course, sometimes (often times!) talk to a RESTful backend in some way
- It may be useful to mock out that RESTful backend under certain circumstances
  - Keep in mind that this is end-to-end testing, so other times, we may want to simply pass the request through to the backend
- Angular's ngMockE2E module provides an $httpBackend implementation suitable for UI testing with Protractor



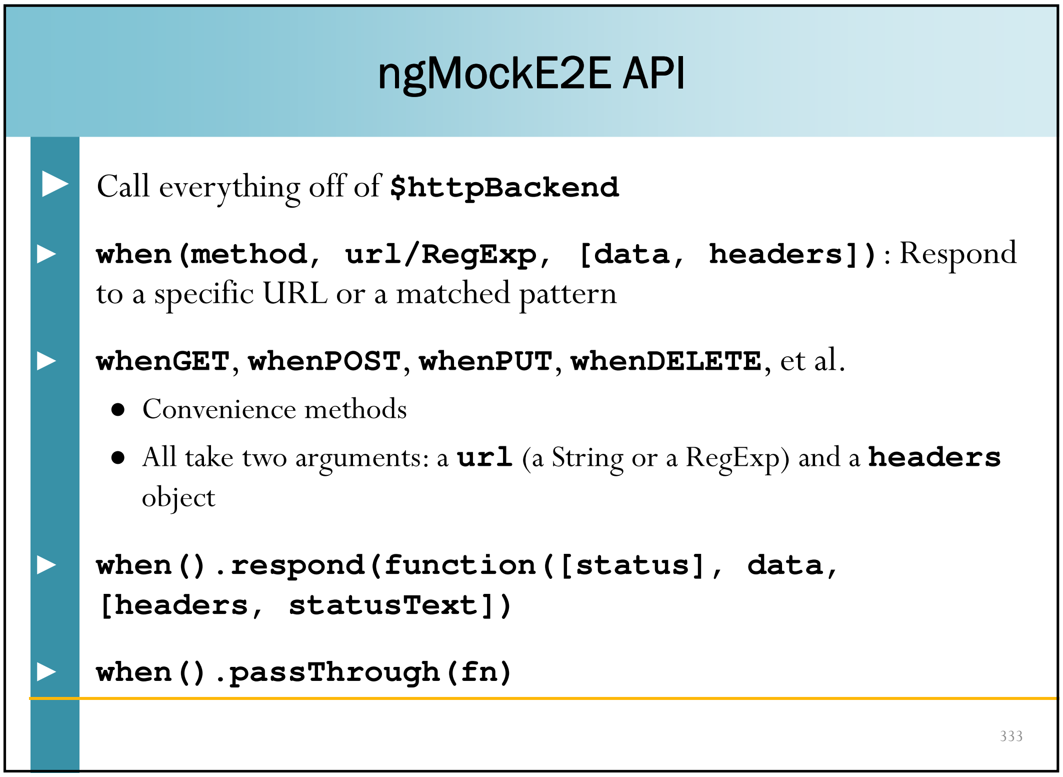
Setting up E2E $httpBackend

- Unlike unit tests, where **`$httpBackend`** could be included as part of the unit test, we must go further for ngMockE2E's backend
- You will need to build a module that constructs the various URLs and their responses
- This module should depend on your main module, as well as **`ngMockE2E`**
- Then define various when calls and their responses
- Should you want to pass through a request, invoke when(…).passThrough()



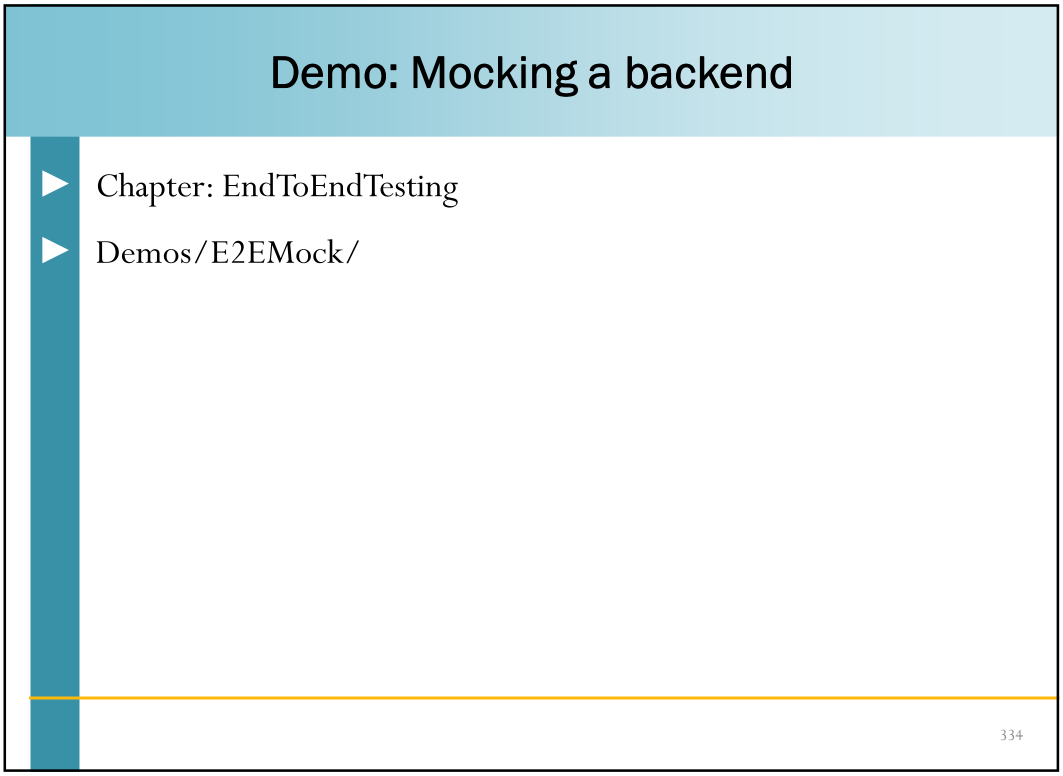
The important part

- Remember on the previous slide where the module you constructed depended on ngMockE2E and your main module?
- That is done so that you can substitute your module with a mocked $httpBackend in place of your original top-level module
- It is not the most elegant of solutions, unfortunately, but it is the option at the moment



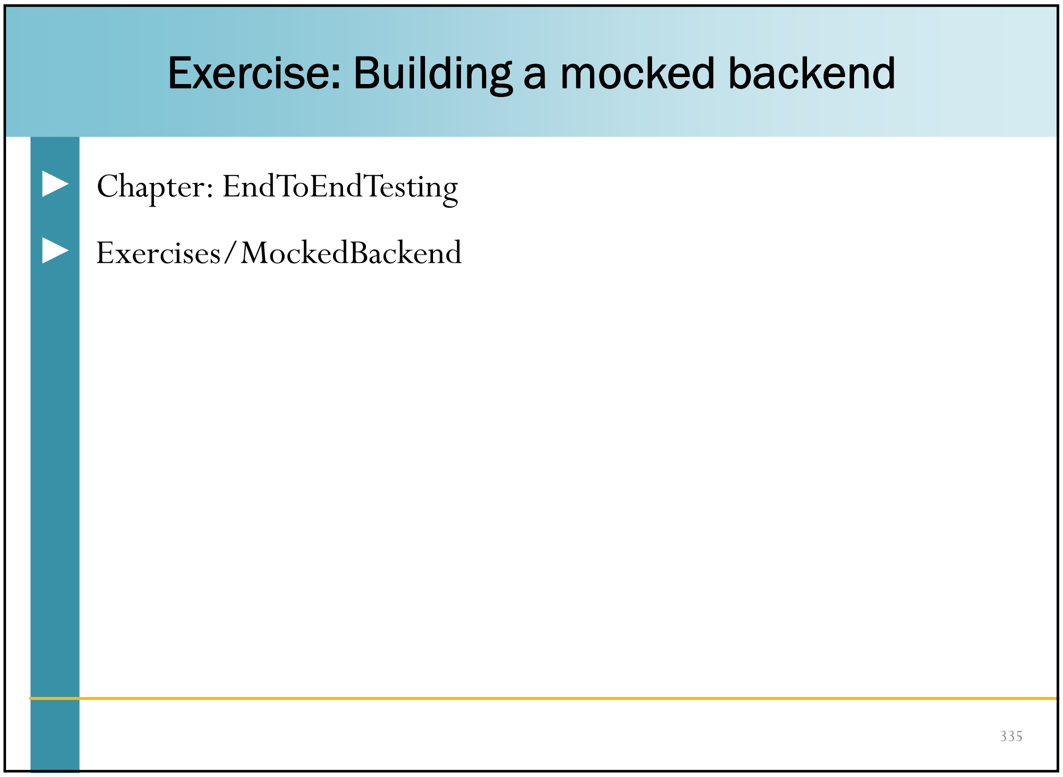
ngMockE2E API

- Call everything off of **`$httpBackend`**
- **`when(method, url/RegExp, [data, headers])`**: Respond to a specific URL or a matched pattern
- **`whenGET`**, **`whenPOST`**, **`whenPUT`**, **`whenDELETE`**, et al.
  - Convenience methods
  - All take two arguments: a **`url`** (a String or a RegExp) and a **`headers`** object
- **`when().respond(function([status], data, [headers, statusText])`**
- **`when().passThrough(fn)`**



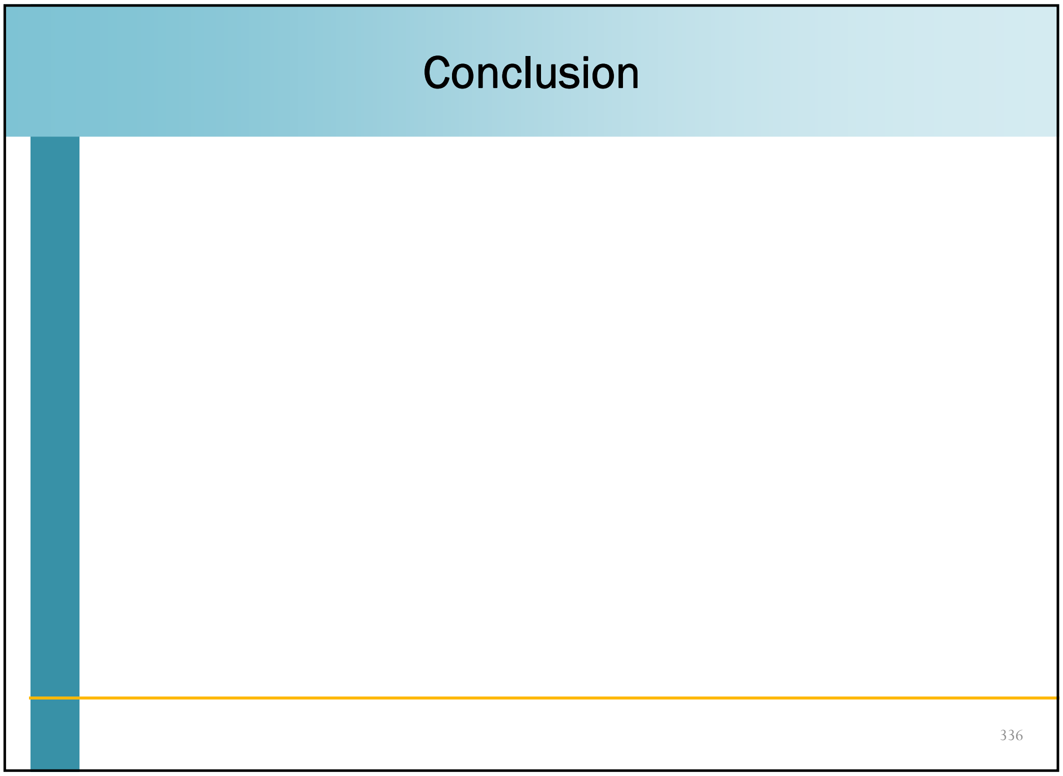
Demo: Mocking a backend

- Chapter: EndToEndTesting
- Demos/E2EMock/

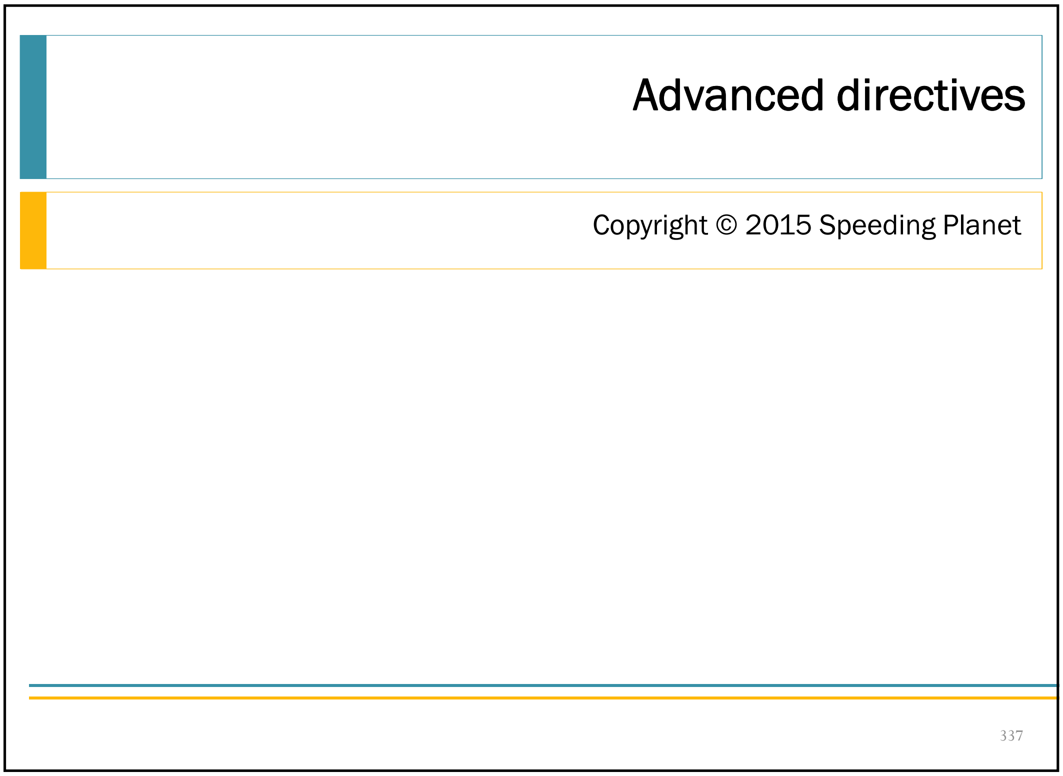


Exercise: Building a mocked backend

- Chapter: EndToEndTesting
- Exercises/MockedBackend

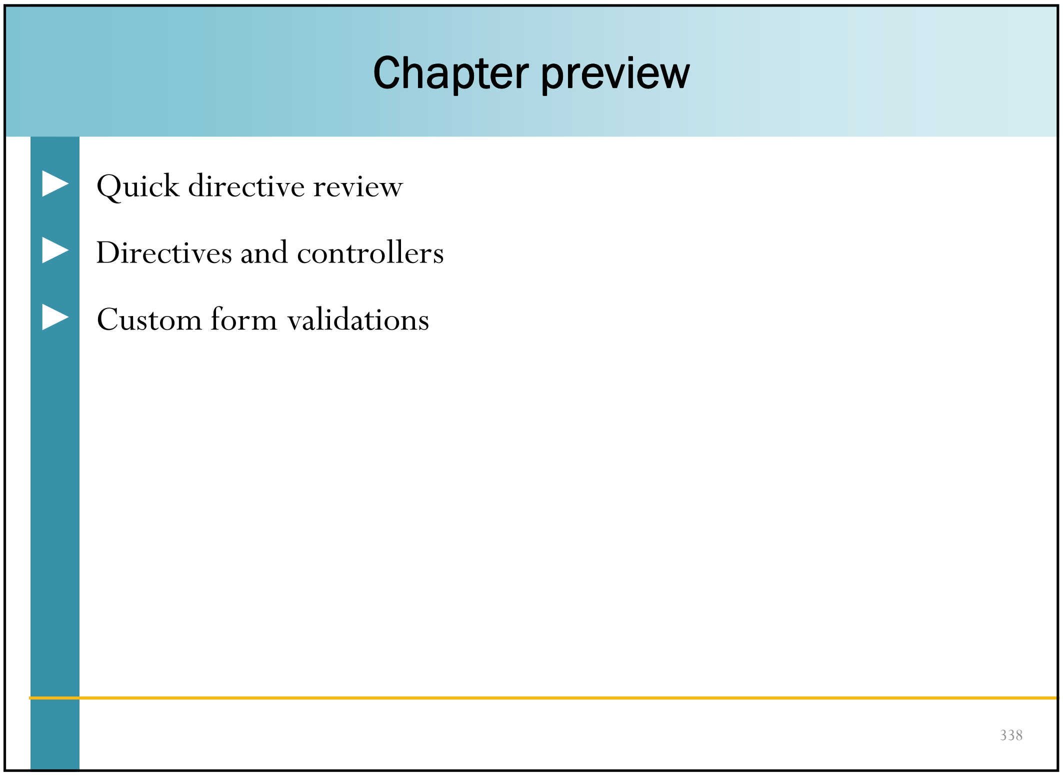


Conclusion



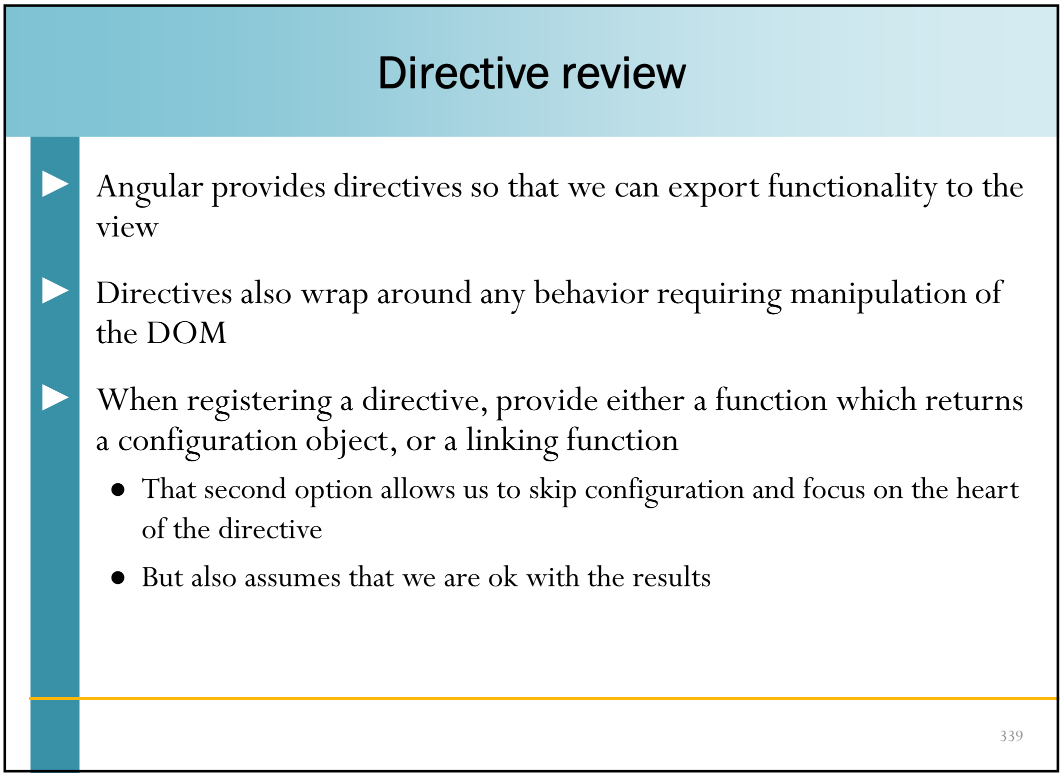
Advanced directives

Copyright © 2015 Speeding Planet



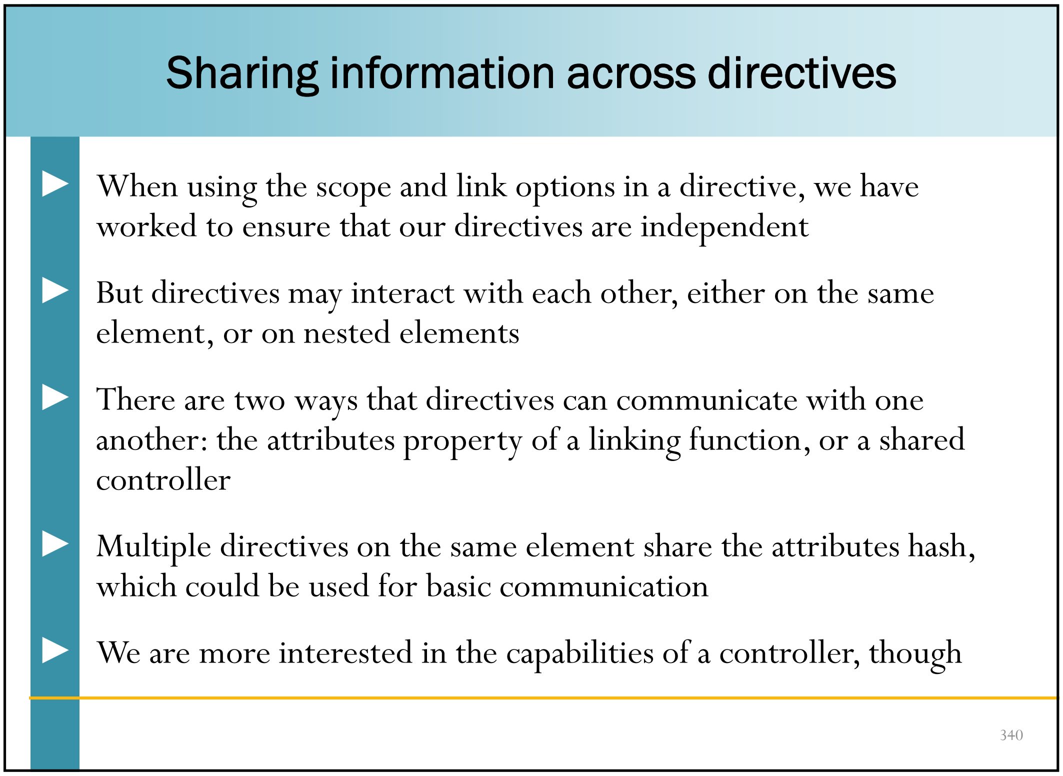
Chapter preview

- Quick directive review
- Directives and controllers
- Custom form validations



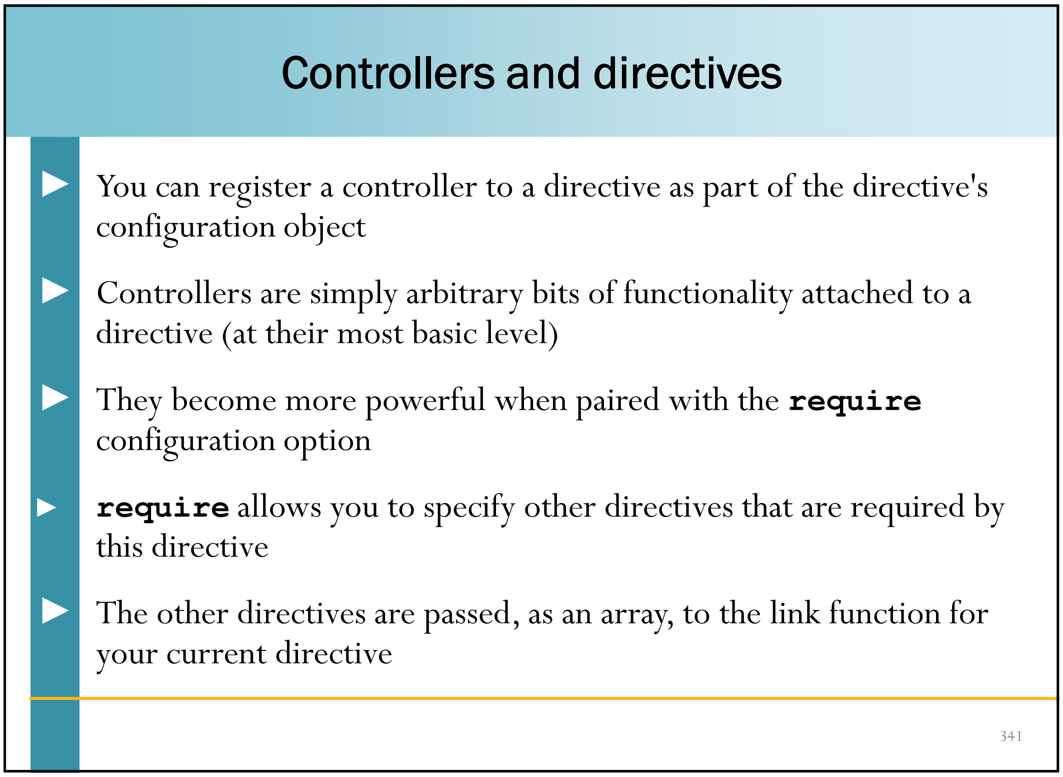
Directive review

- Angular provides directives so that we can export functionality to the view
- Directives also wrap around any behavior requiring manipulation of the DOM
- When registering a directive, provide either a function which returns a configuration object, or a linking function
  - That second option allows us to skip configuration and focus on the heart of the directive
  - But also assumes that we are ok with the results



Sharing information across directives

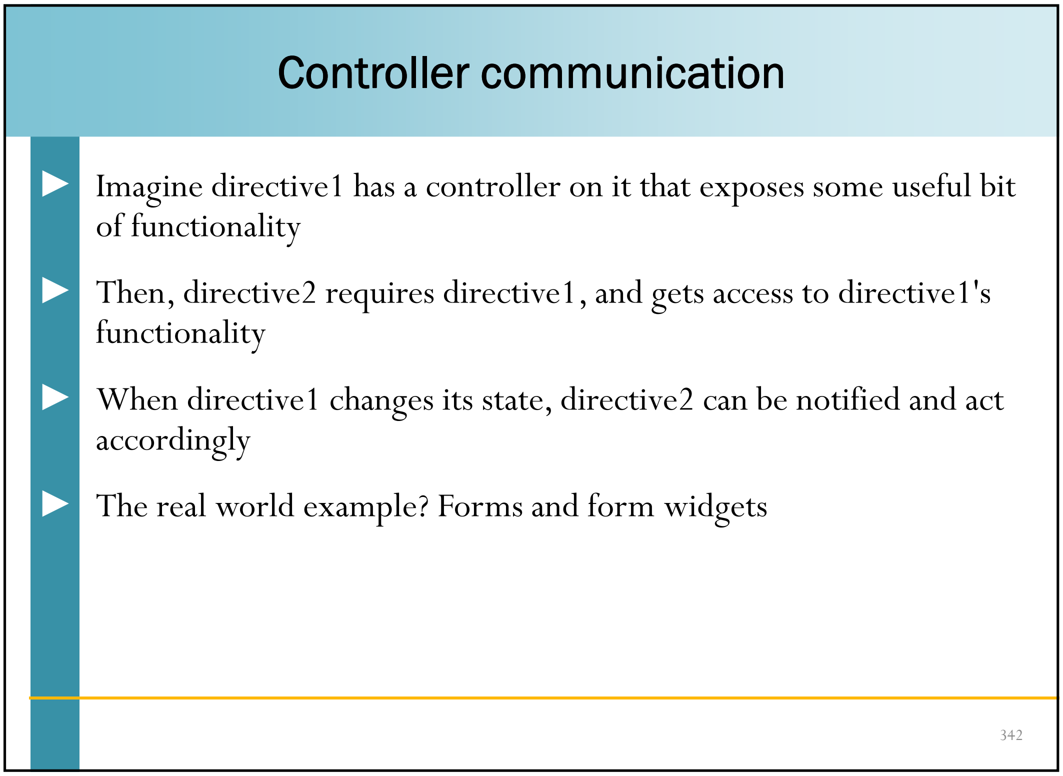
- When using the scope and link options in a directive, we have worked to ensure that our directives are independent
- But directives may interact with each other, either on the same element, or on nested elements
- There are two ways that directives can communicate with one another: the attributes property of a linking function, or a shared controller
- Multiple directives on the same element share the attributes hash, which could be used for basic communication
- We are more interested in the capabilities of a controller, though



Controllers and directives

- You can register a controller to a directive as part of the directive's configuration object
- Controllers are simply arbitrary bits of functionality attached to a directive (at their most basic level)
- They become more powerful when paired with the **`require`** configuration option
- **`require`** allows you to specify other directives that are required by this directive
- The other directives are passed, as an array, to the link function for your current directive

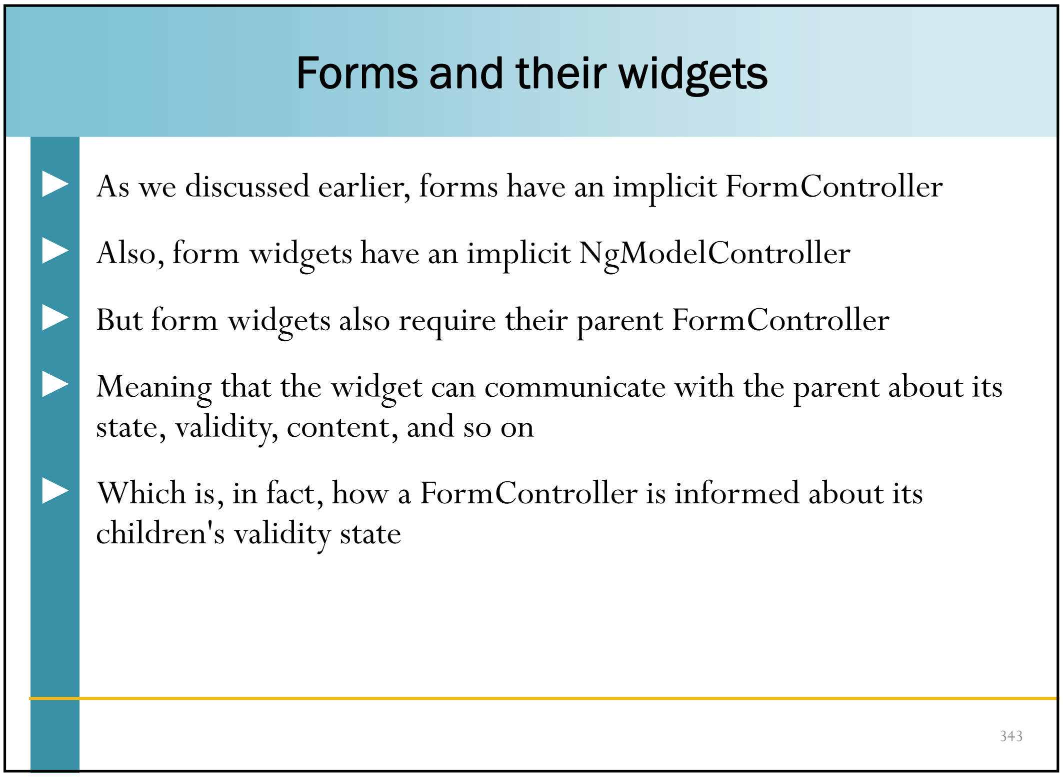
341



Controller communication

- Imagine directive1 has a controller on it that exposes some useful bit of functionality
- Then, directive2 requires directive1, and gets access to directive1's functionality
- When directive1 changes its state, directive2 can be notified and act accordingly
- The real world example? Forms and form widgets

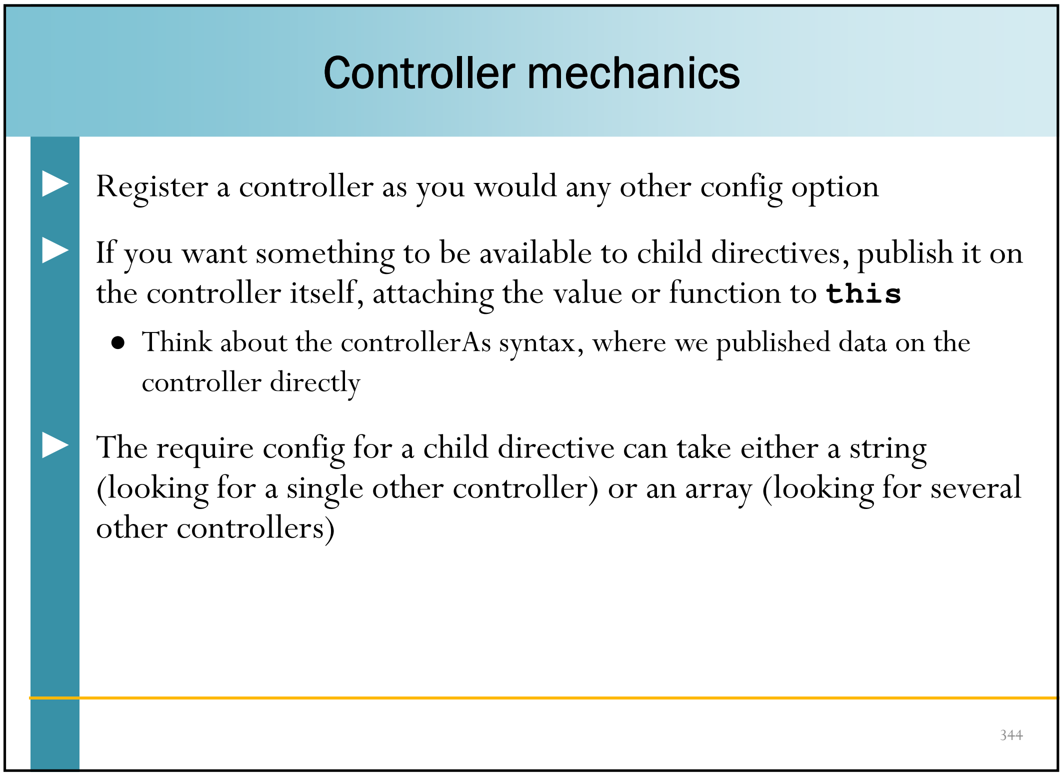
342



Forms and their widgets

- As we discussed earlier, forms have an implicit FormController
- Also, form widgets have an implicit NgModelController
- But form widgets also require their parent FormController
- Meaning that the widget can communicate with the parent about its state, validity, content, and so on
- Which is, in fact, how a FormController is informed about its children's validity state

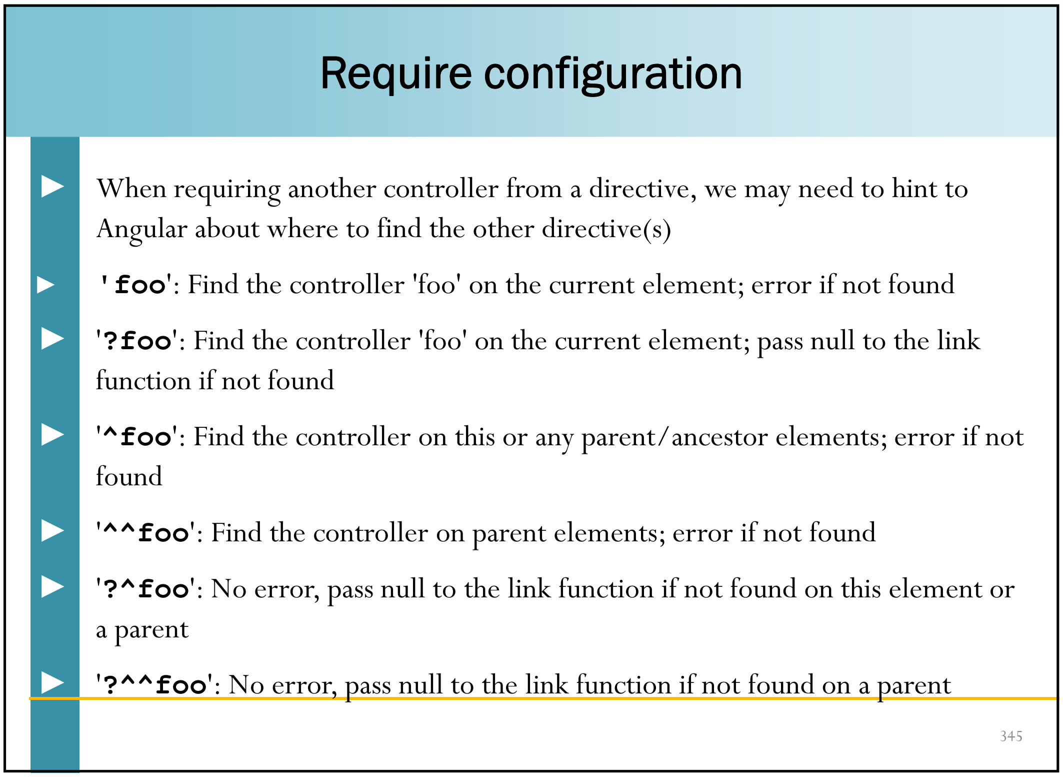
343



Controller mechanics

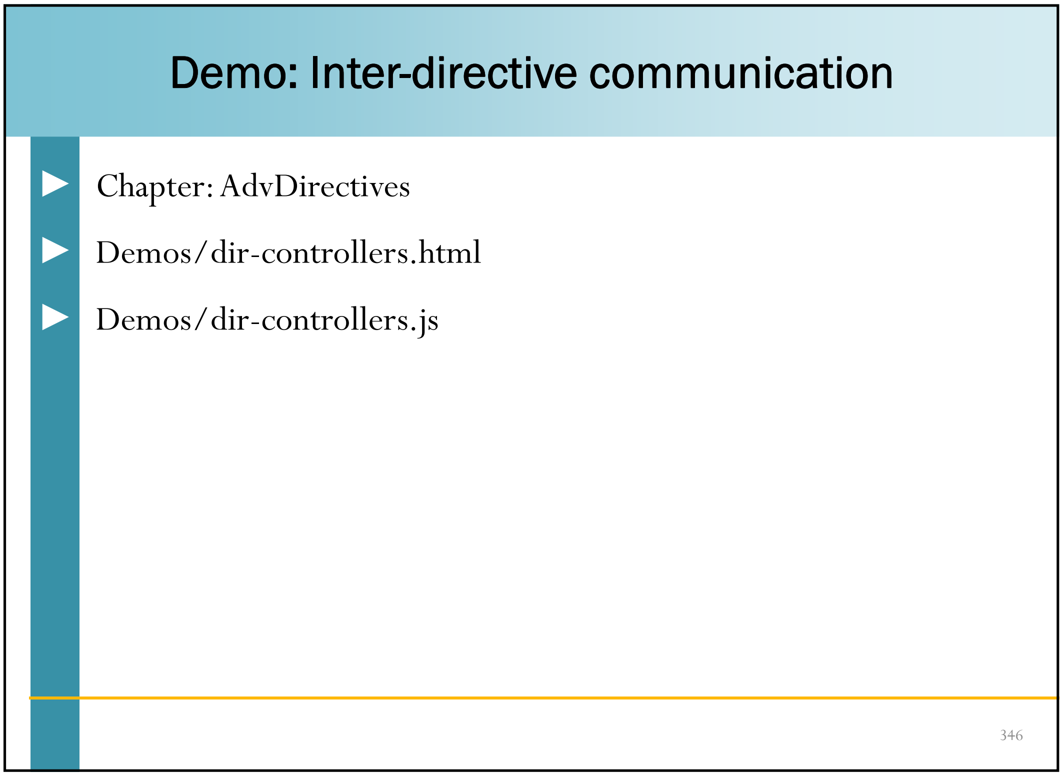
- Register a controller as you would any other config option
- If you want something to be available to child directives, publish it on the controller itself, attaching the value or function to **`this`**
  - Think about the controllerAs syntax, where we published data on the controller directly
- The require config for a child directive can take either a string (looking for a single other controller) or an array (looking for several other controllers)

344



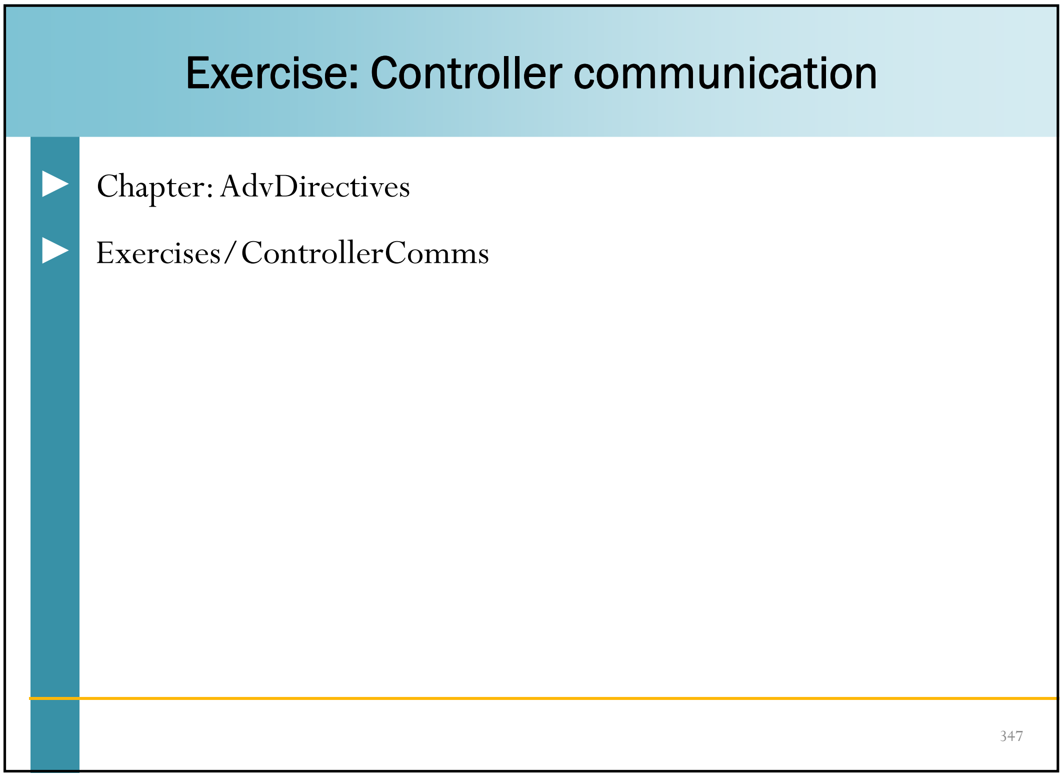
Require configuration

- When requiring another controller from a directive, we may need to hint to Angular about where to find the other directive(s)
- '**foo**': Find the controller 'foo' on the current element; error if not found
- '**?foo**': Find the controller 'foo' on the current element; pass null to the link function if not found
- '**^foo**': Find the controller on this or any parent/ancestor elements; error if not found
- '**^^foo**': Find the controller on parent elements; error if not found
- '**?^foo**': No error, pass null to the link function if not found on this element or a parent
- '**?^^foo**': No error, pass null to the link function if not found on a parent



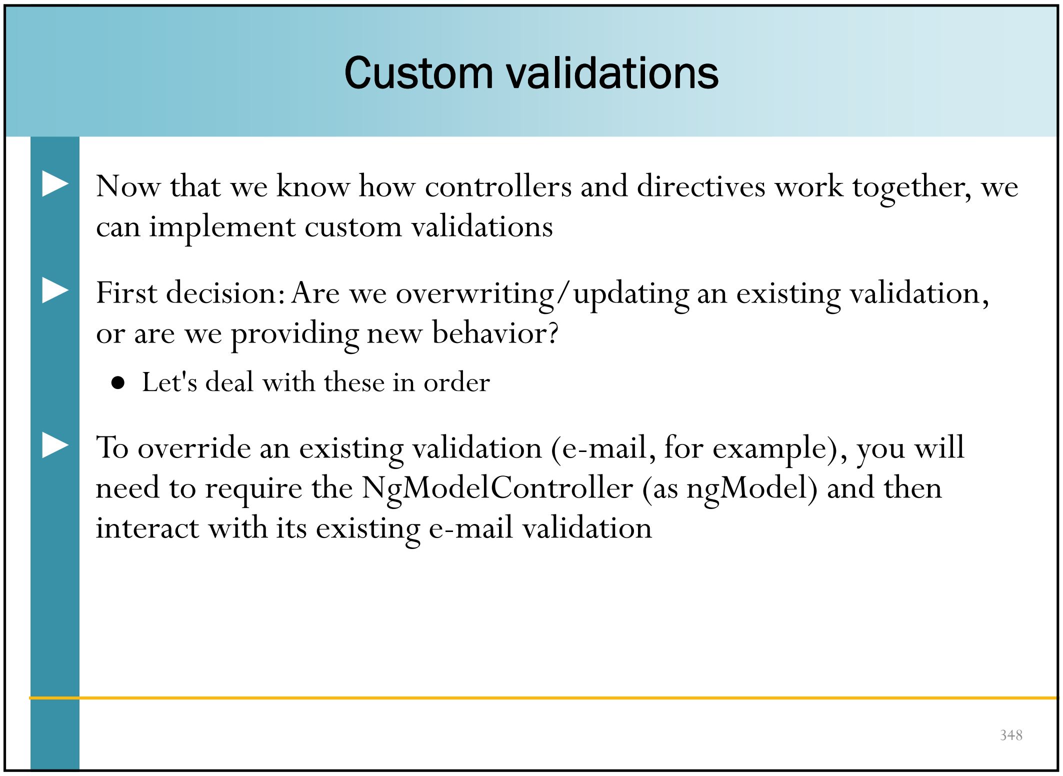
Demo: Inter-directive communication

- Chapter: AdvDirectives
- Demos/dir-controllers.html
- Demos/dir-controllers.js



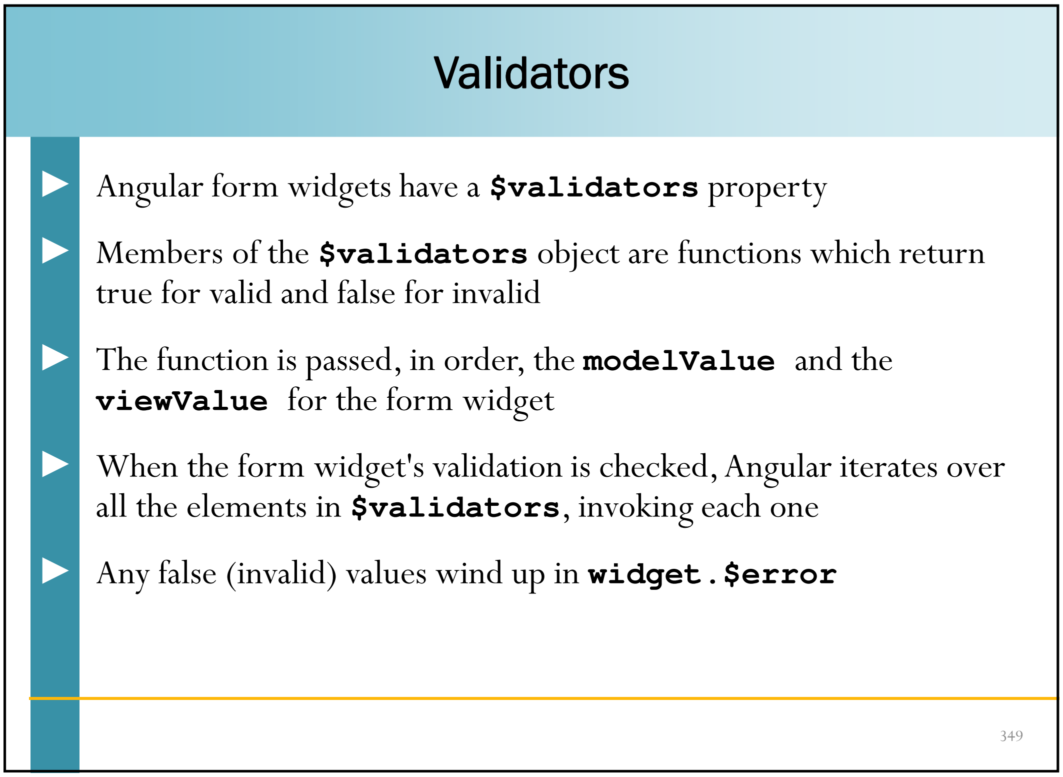
Exercise: Controller communication

- Chapter: AdvDirectives
- Exercises/ControllerComms



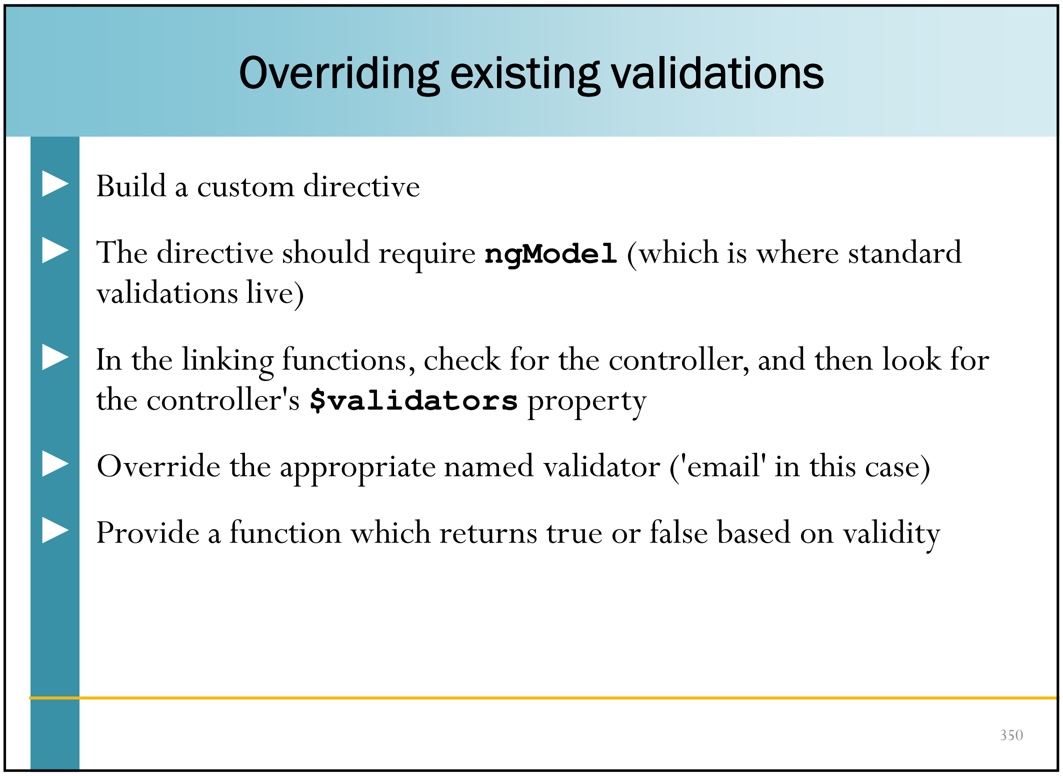
Custom validations

- Now that we know how controllers and directives work together, we can implement custom validations
- First decision: Are we overwriting/updating an existing validation, or are we providing new behavior?
  - Let's deal with these in order
- To override an existing validation (e-mail, for example), you will need to require the NgModelController (as ngModel) and then interact with its existing e-mail validation



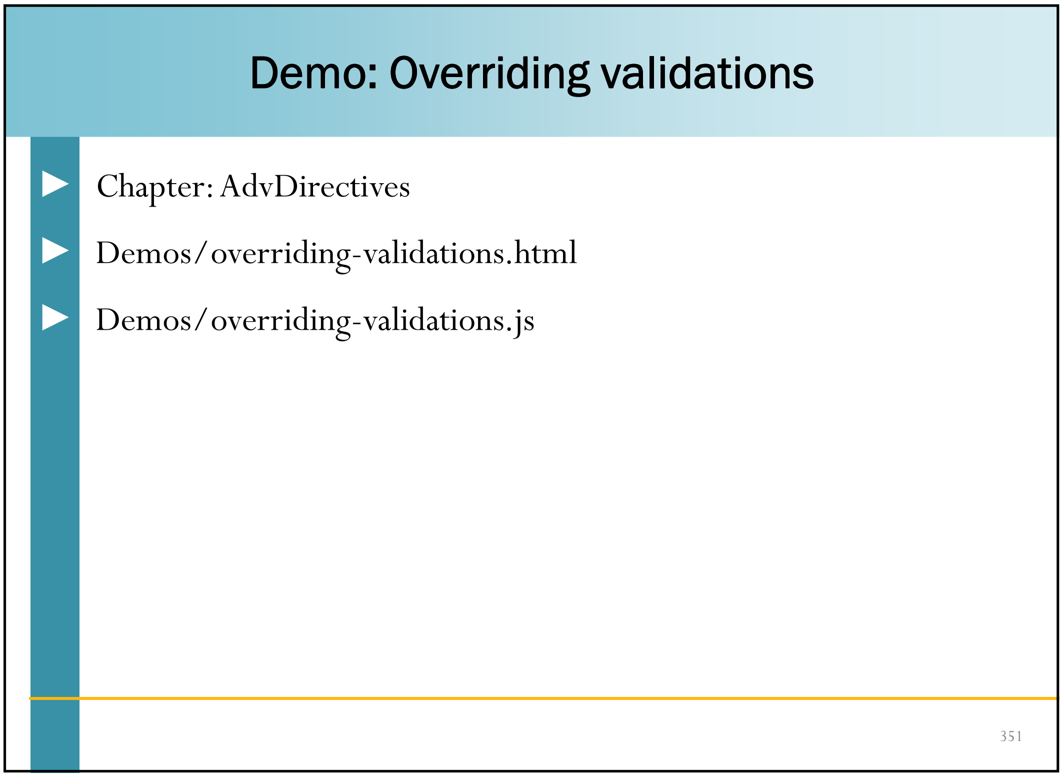
Validators

- Angular form widgets have a **`$validators`** property
- Members of the **`$validators`** object are functions which return true for valid and false for invalid
- The function is passed, in order, the **`modelValue`** and the **`viewValue`** for the form widget
- When the form widget's validation is checked, Angular iterates over all the elements in **`$validators`**, invoking each one
- Any false (invalid) values wind up in **`widget.$error`**



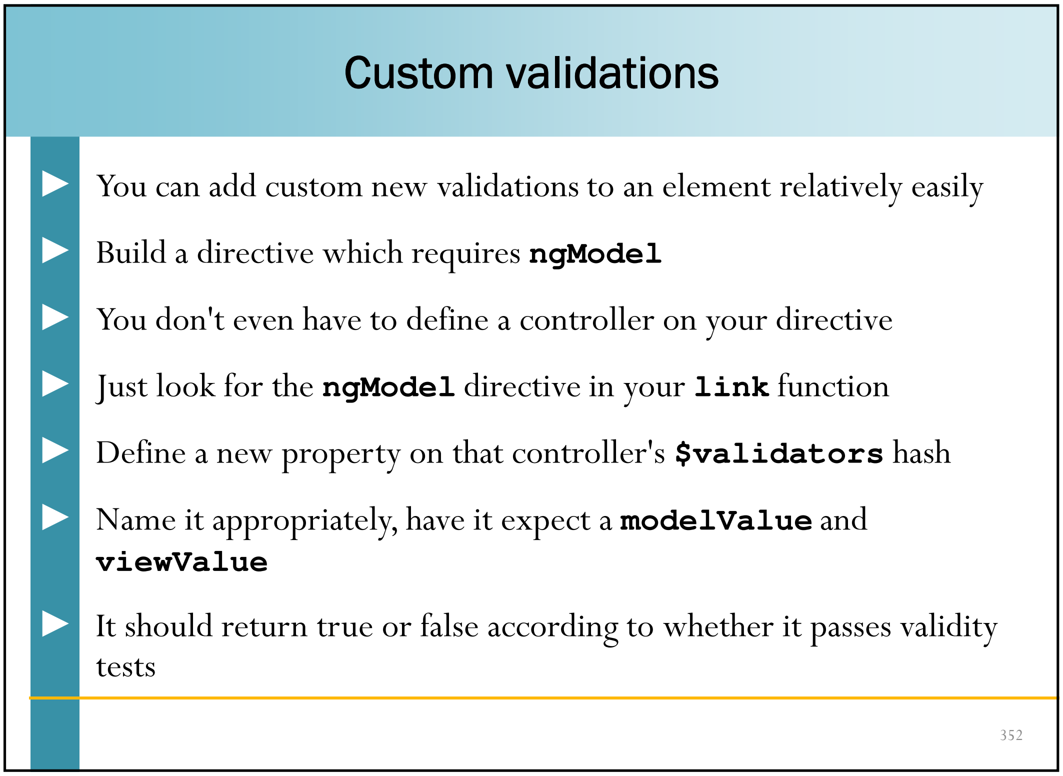
Overriding existing validations

- Build a custom directive
- The directive should require **`ngModel`** (which is where standard validations live)
- In the linking functions, check for the controller, and then look for the controller's **`$validators`** property
- Override the appropriate named validator ('email' in this case)
- Provide a function which returns true or false based on validity



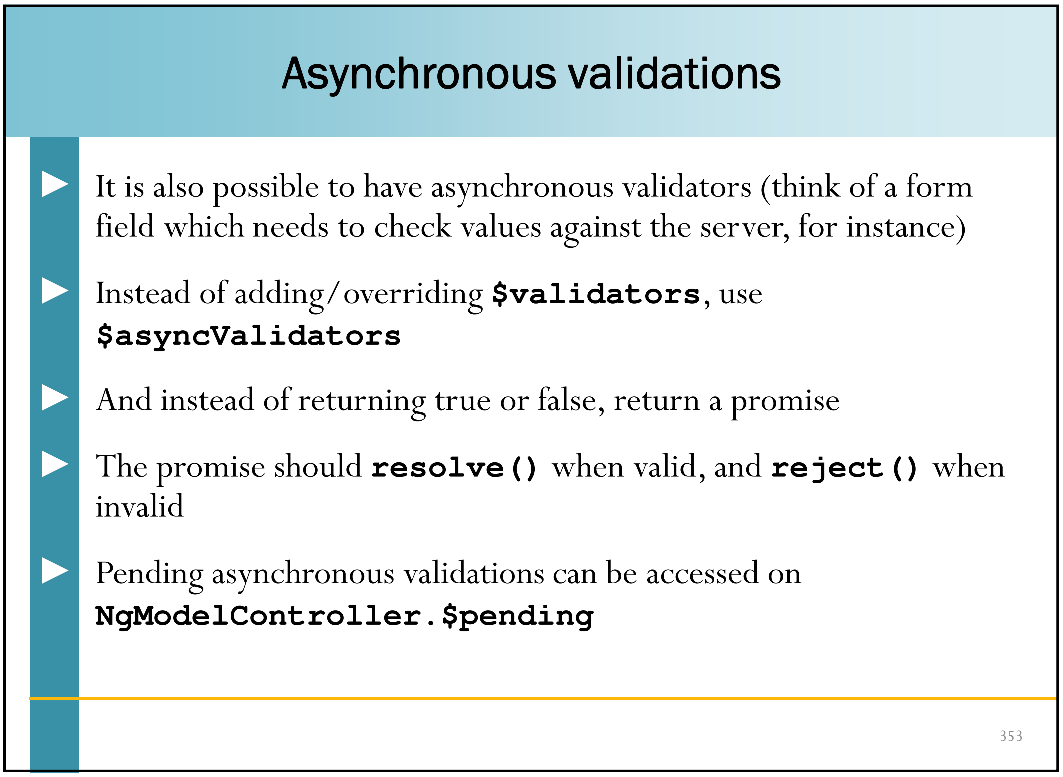
Demo: Overriding validations

- Chapter: AdvDirectives
- Demos/overriding-validations.html
- Demos/overriding-validations.js



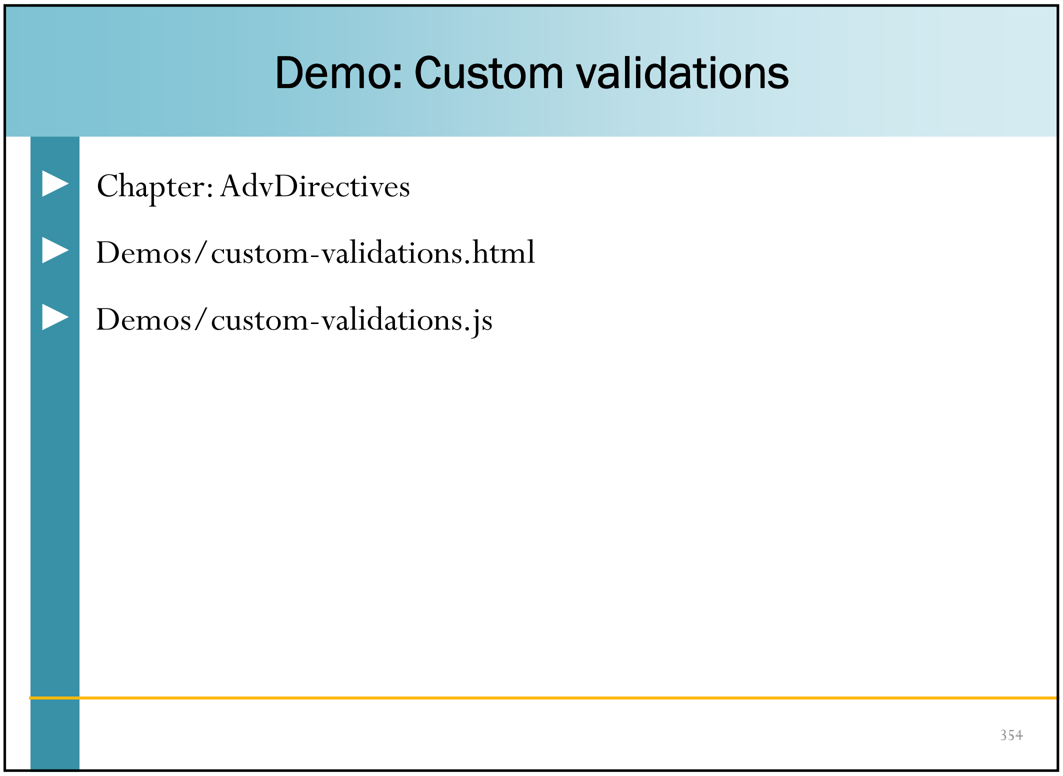
Custom validations

- You can add custom new validations to an element relatively easily
- Build a directive which requires **`ngModel`**
- You don't even have to define a controller on your directive
- Just look for the **`ngModel`** directive in your **`link`** function
- Define a new property on that controller's **`$validators`** hash
- Name it appropriately, have it expect a **`modelValue`** and **`viewValue`**
- It should return true or false according to whether it passes validity tests



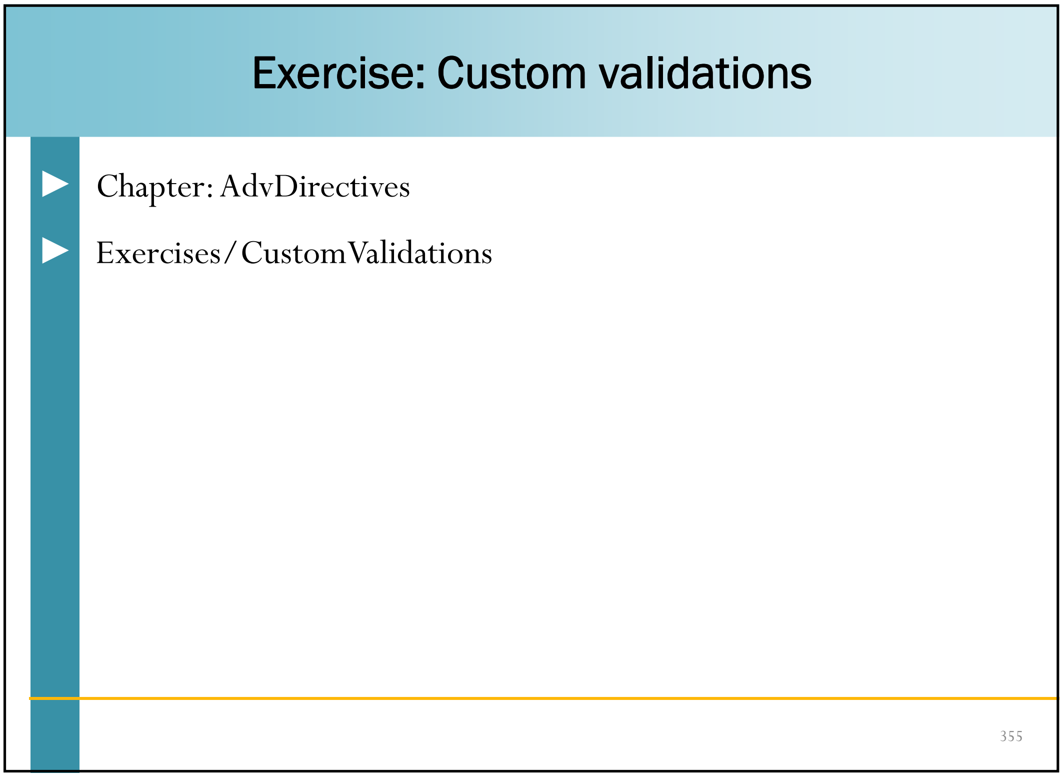
Asynchronous validations

- It is also possible to have asynchronous validators (think of a form field which needs to check values against the server, for instance)
- Instead of adding/overriding **`$validators`**, use **`$asyncValidators`**
- And instead of returning true or false, return a promise
- The promise should **`resolve()`** when valid, and **`reject()`** when invalid
- Pending asynchronous validations can be accessed on **`NgModelController.$pending`**



Demo: Custom validations

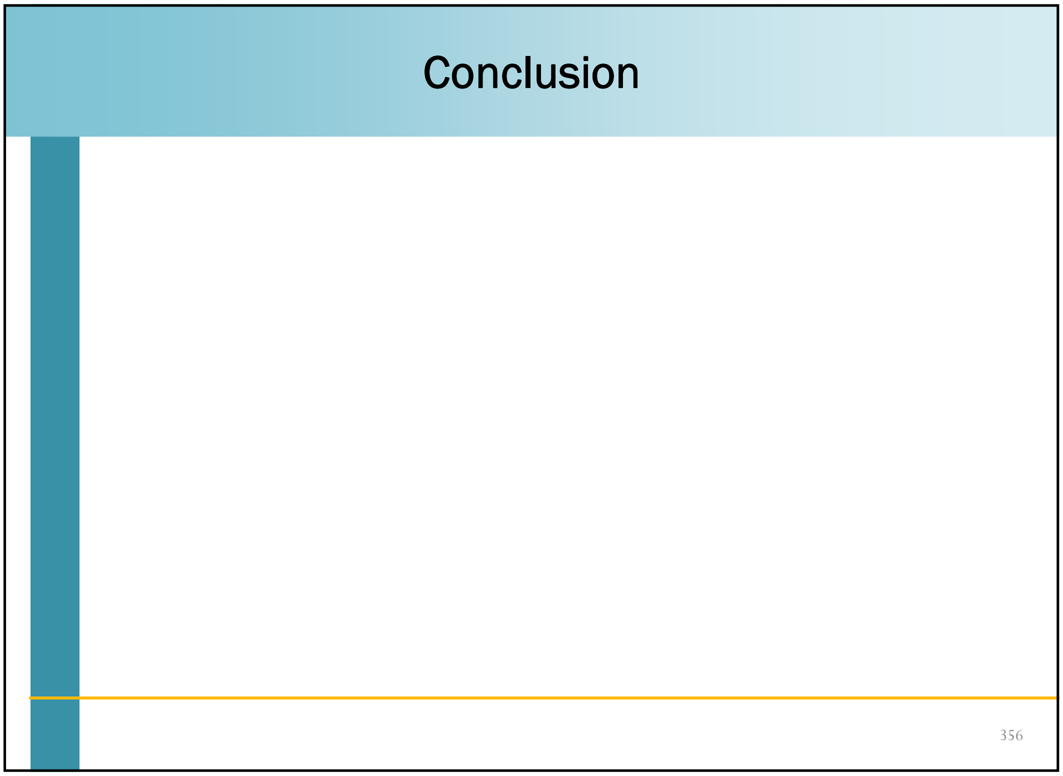
- Chapter: AdvDirectives
- Demos/custom-validations.html
- Demos/custom-validations.js



Exercise: Custom validations

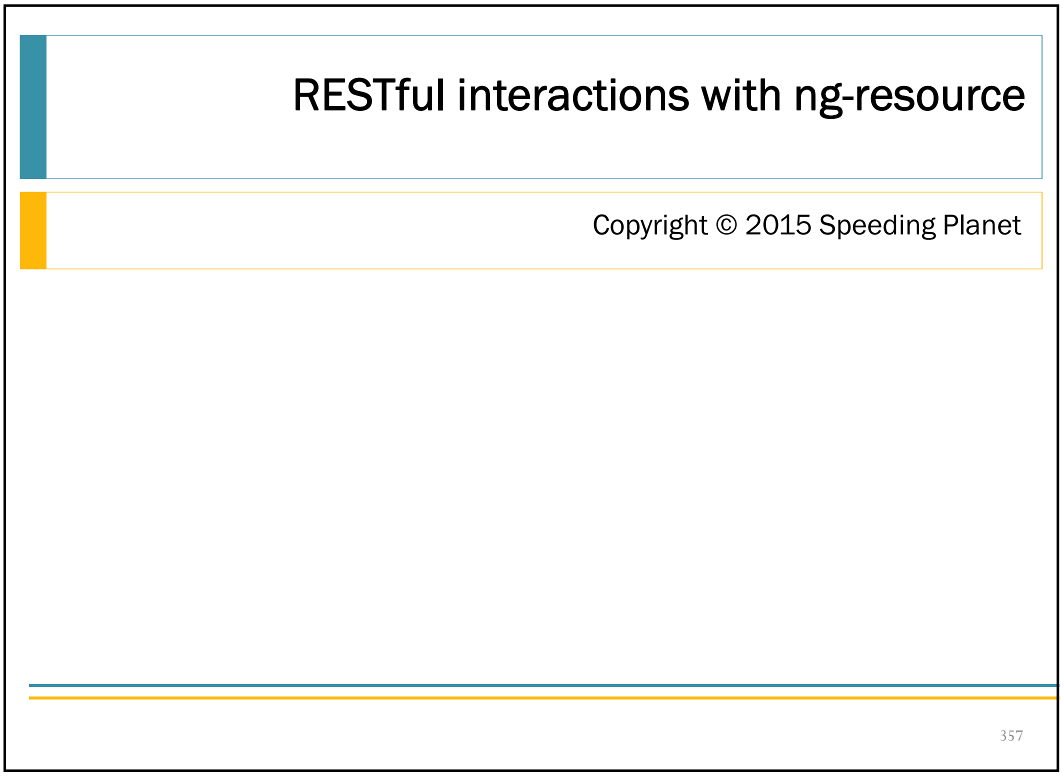
- Chapter: AdvDirectives
- Exercises/CustomValidations

355



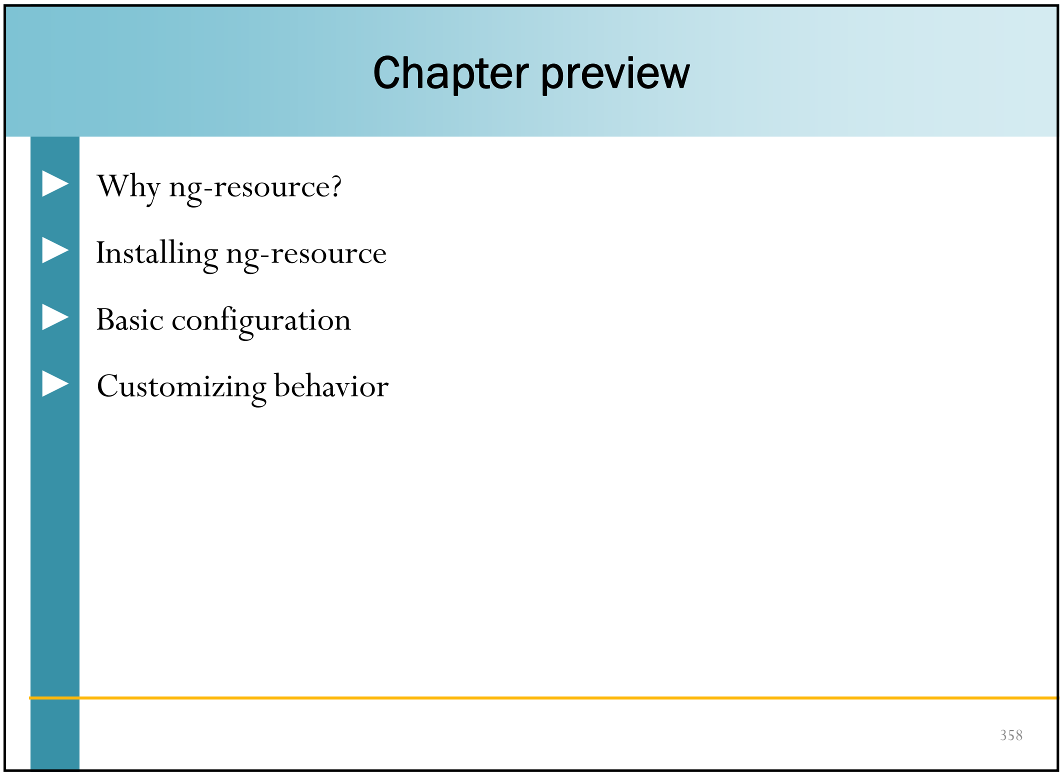
Conclusion

356



RESTful interactions with ng-resource

357



Chapter preview

- Why ng-resource?
- Installing ng-resource
- Basic configuration
- Customizing behavior

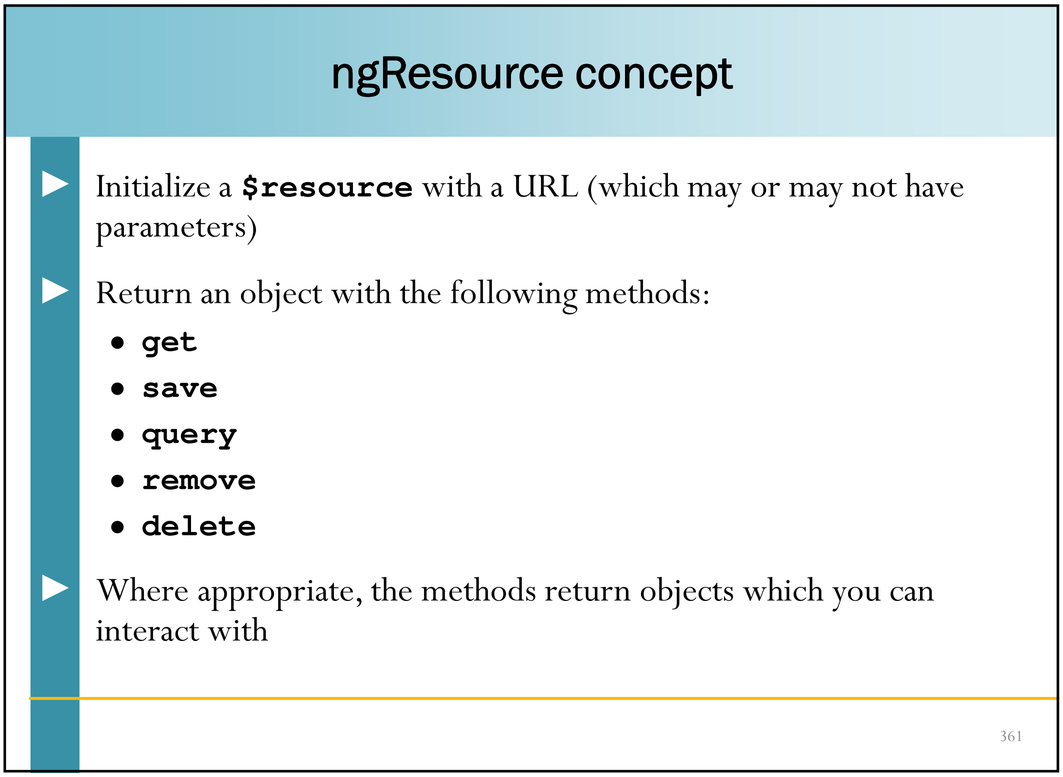
358

Why ng-resource?

- Up to this point, most of our interactions with a server have been through either $http directly, or a provider which wraps $http
- This is useful for one-off or custom behavior, but we have had to implement a somewhat extensive array of functionality
- We know what RESTful patterns of behavior are, yet we still had to implement CRUD operations on the client side ourselves
- Wouldn't it be nice to be able to point to a RESTful URL and tell Angular to configure a service based on reasonable assumptions?
  - Hint: yes, and that's exactly what ng-resource does

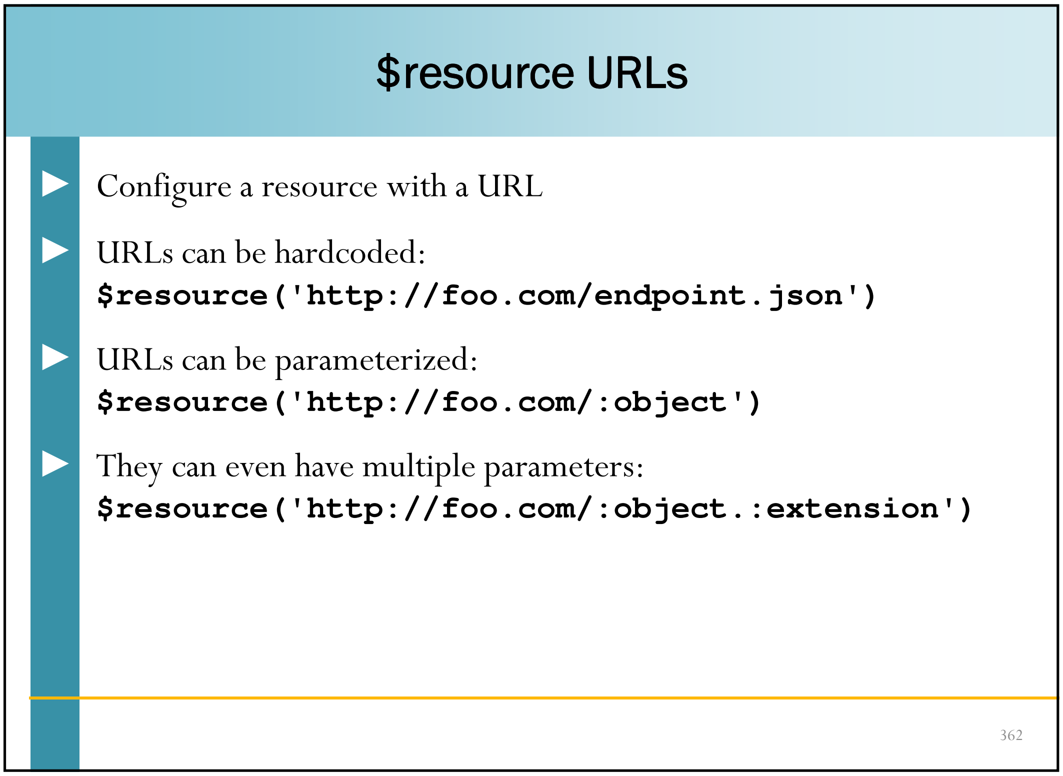
Adding ng-resource

- Download angular-resource.js
- Include angular-resource.js in your HTML, sometime after angular.js
- Have the appropriate modules (probably your top-level module) depend on ngResource



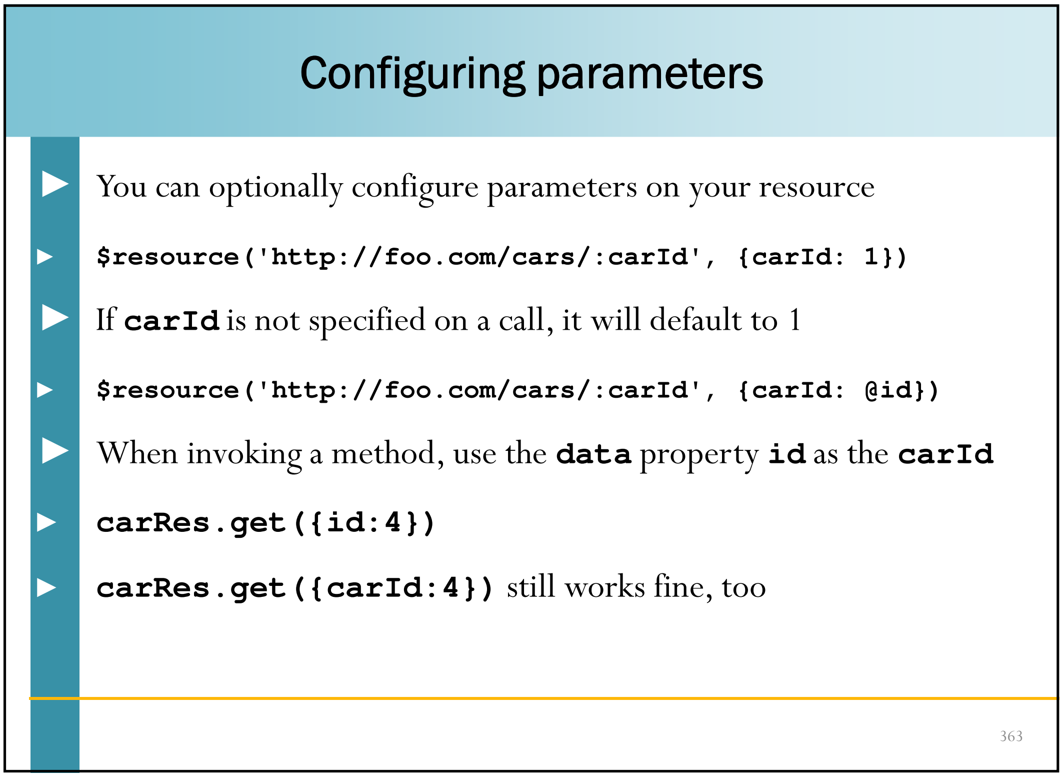
ngResource concept

- Initialize a **`$resource`** with a URL (which may or may not have parameters)
- Return an object with the following methods:
  - **`get`**
  - **`save`**
  - **`query`**
  - **`remove`**
  - **`delete`**
- Where appropriate, the methods return objects which you can interact with



$resource URLs

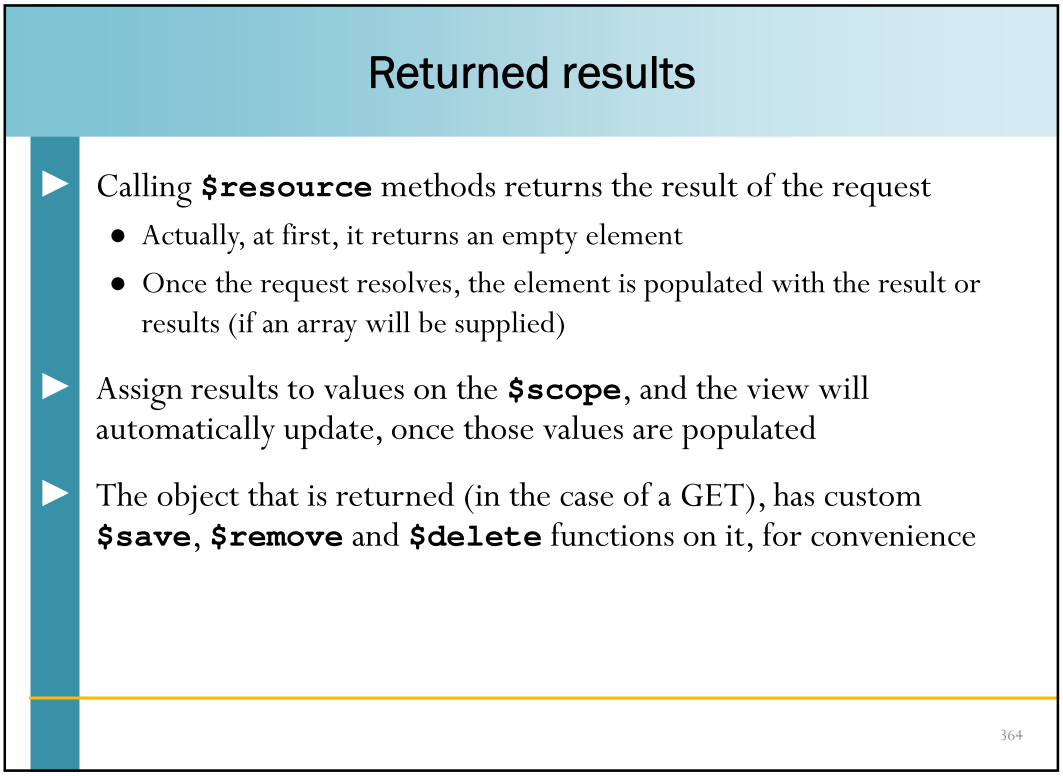
- Configure a resource with a URL
- URLs can be hardcoded:
  `$resource('http://foo.com/endpoint.json')`
- URLs can be parameterized:
  `$resource('http://foo.com/:object')`
- They can even have multiple parameters:
  `$resource('http://foo.com/:object.:extension')`



Configuring parameters

- You can optionally configure parameters on your resource
- `$resource('http://foo.com/cars/:carId', {carId: 1})`
- If **`carId`** is not specified on a call, it will default to 1
- `$resource('http://foo.com/cars/:carId', {carId: @id})`
- When invoking a method, use the **`data`** property **`id`** as the **`carId`**
- **`carRes.get({id:4})`**
- **`carRes.get({carId:4})`** still works fine, too

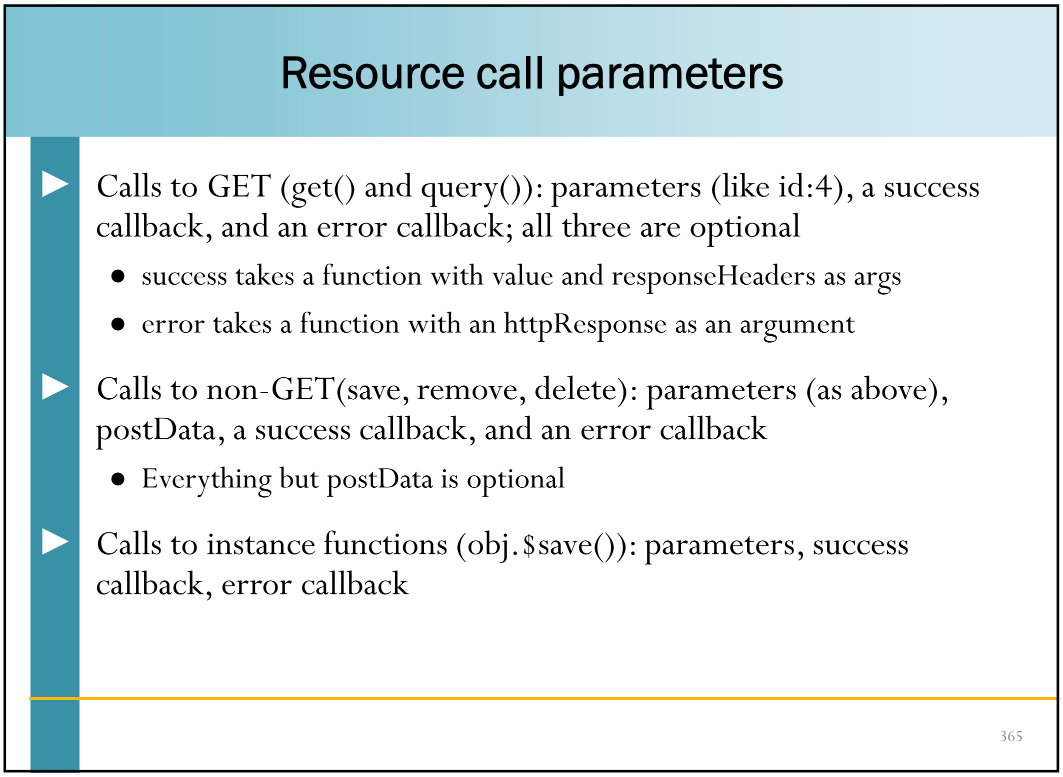
363



Returned results

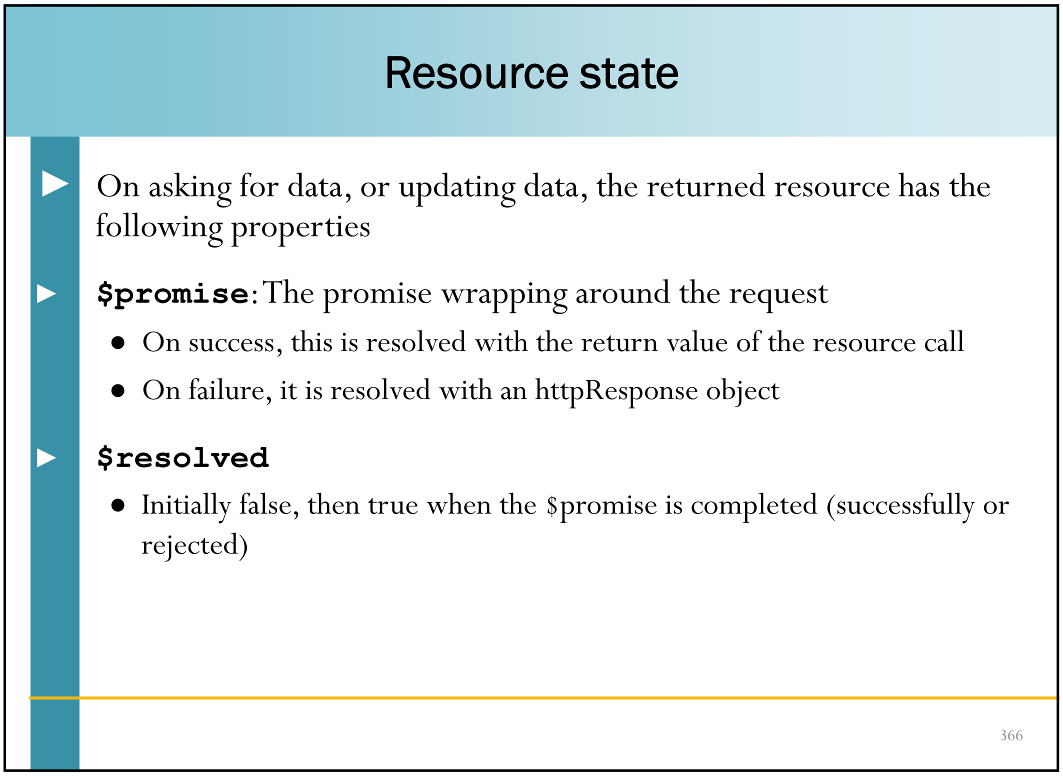
- Calling **`$resource`** methods returns the result of the request
  - Actually, at first, it returns an empty element
  - Once the request resolves, the element is populated with the result or results (if an array will be supplied)
- Assign results to values on the **`$scope`**, and the view will automatically update, once those values are populated
- The object that is returned (in the case of a GET), has custom **`$save`**, **`$remove`** and **`$delete`** functions on it, for convenience

364



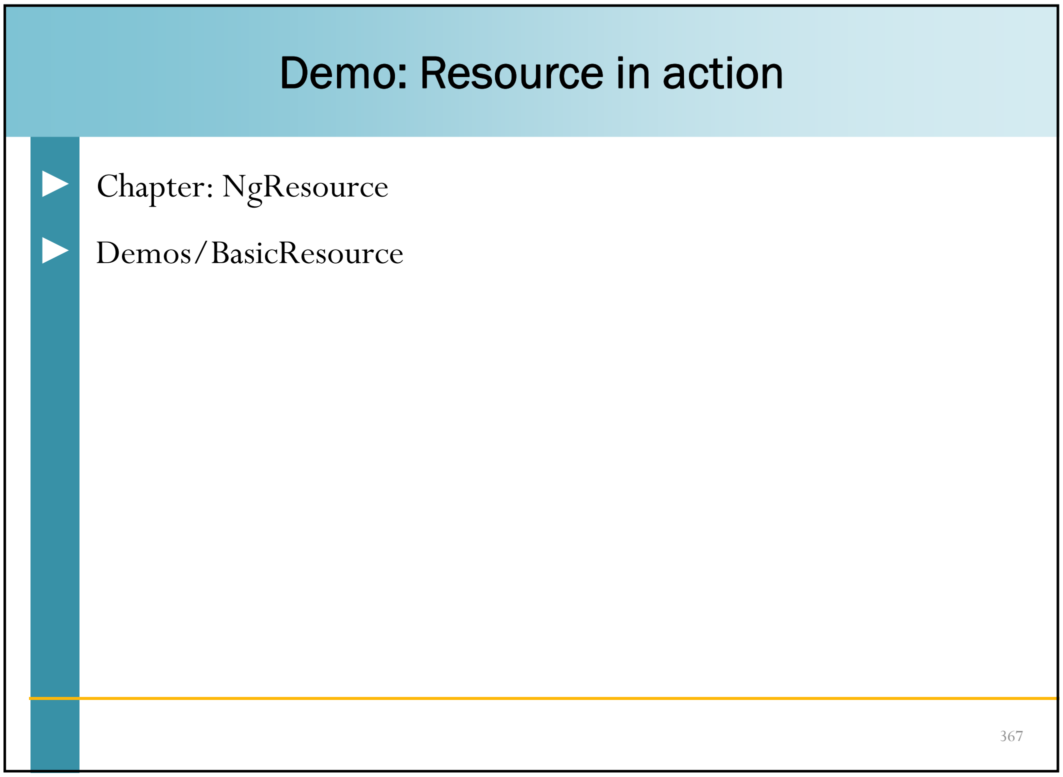
Resource call parameters

- Calls to GET (get() and query()): parameters (like id:4), a success callback, and an error callback; all three are optional
  - success takes a function with value and responseHeaders as args
  - error takes a function with an httpResponse as an argument
- Calls to non-GET(save, remove, delete): parameters (as above), postData, a success callback, and an error callback
  - Everything but postData is optional
- Calls to instance functions (obj.$save()): parameters, success callback, error callback



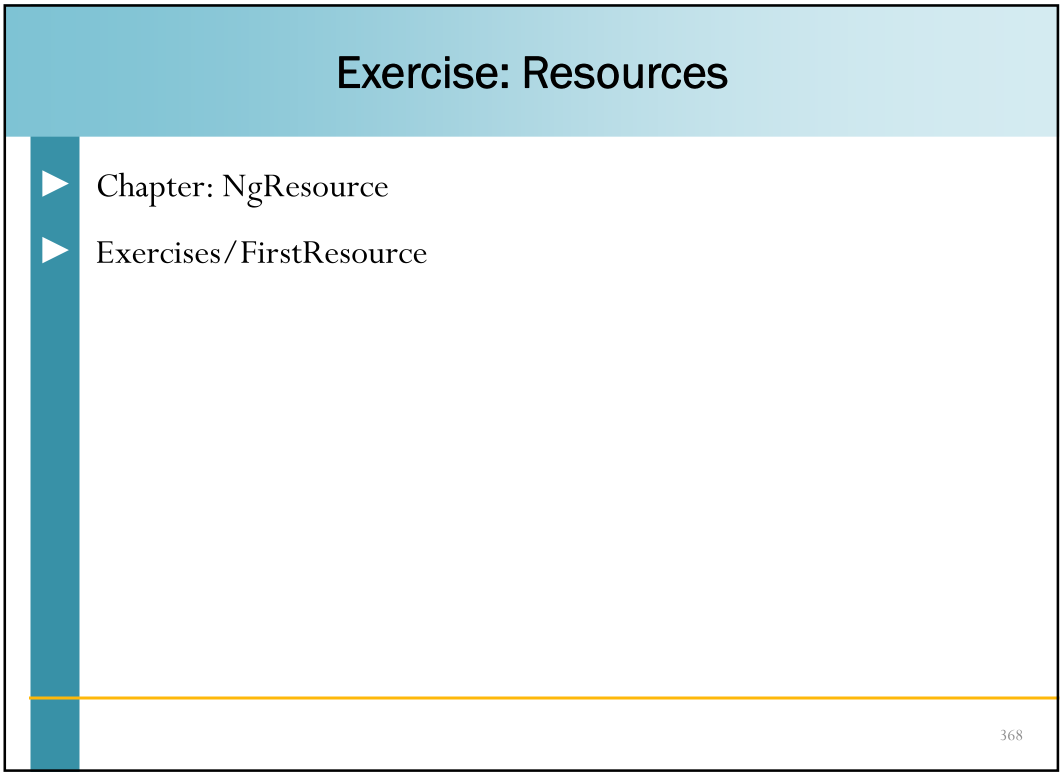
Resource state

- On asking for data, or updating data, the returned resource has the following properties
- **`$promise`**: The promise wrapping around the request
  - On success, this is resolved with the return value of the resource call
  - On failure, it is resolved with an httpResponse object
- **`$resolved`**
  - Initially false, then true when the $promise is completed (successfully or rejected)



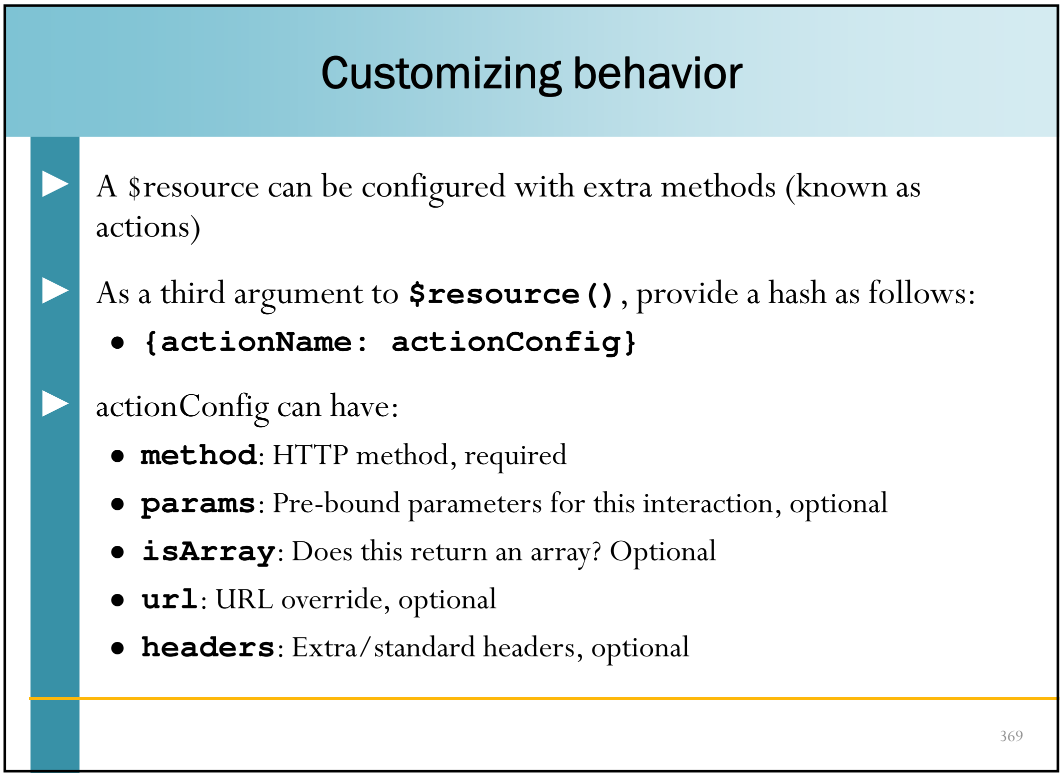
Demo: Resource in action

- Chapter: NgResource
- Demos/BasicResource



Exercise: Resources

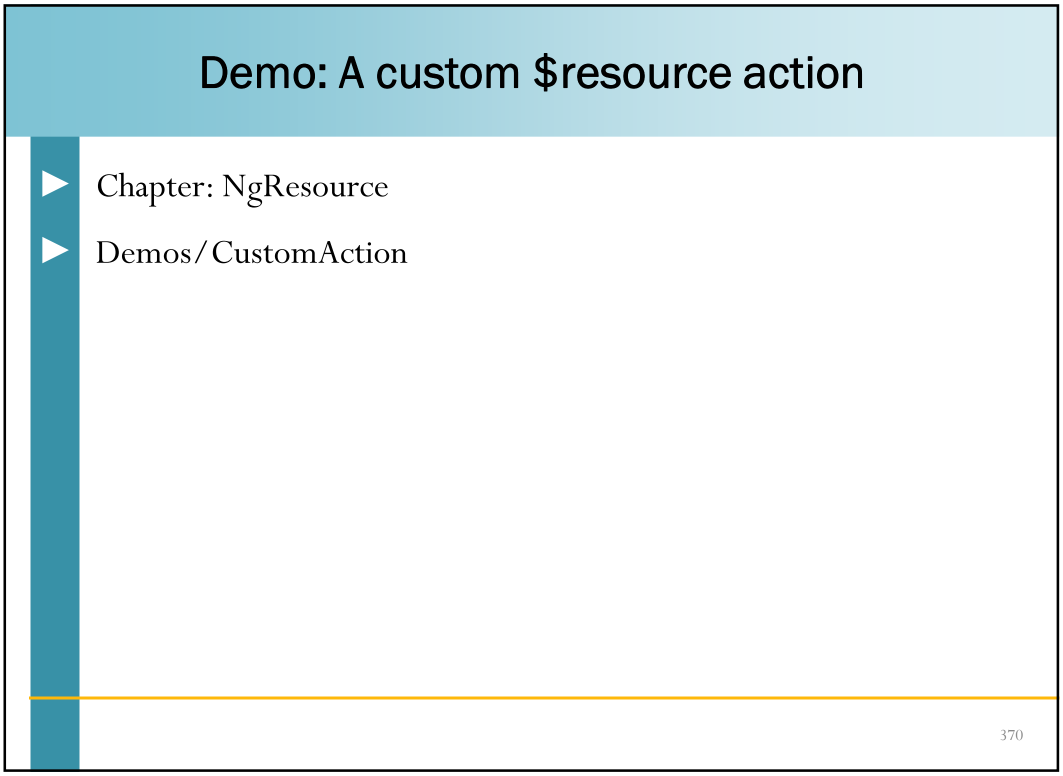
- Chapter: NgResource
- Exercises/FirstResource



Customizing behavior

- A $resource can be configured with extra methods (known as actions)
- As a third argument to **`$resource()`**, provide a hash as follows:
  - **`{actionName: actionConfig}`**
- actionConfig can have:
  - **`method`**: HTTP method, required
  - **`params`**: Pre-bound parameters for this interaction, optional
  - **`isArray`**: Does this return an array? Optional
  - **`url`**: URL override, optional
  - **`headers`**: Extra/standard headers, optional

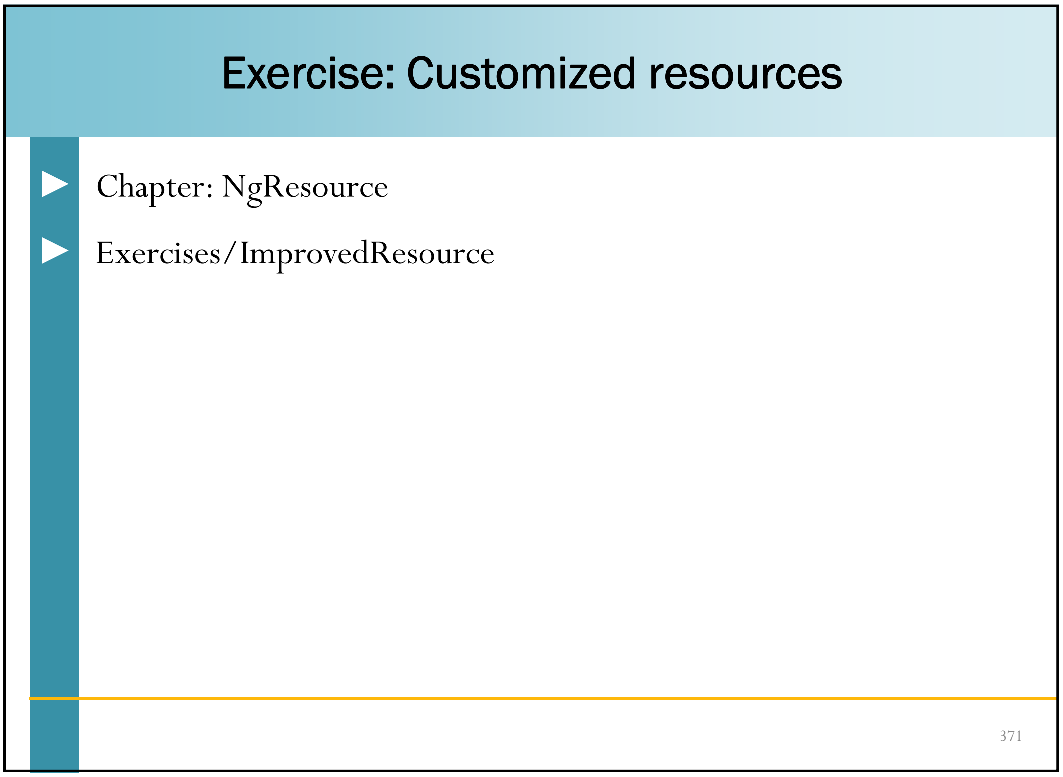
369



Demo: A custom $resource action

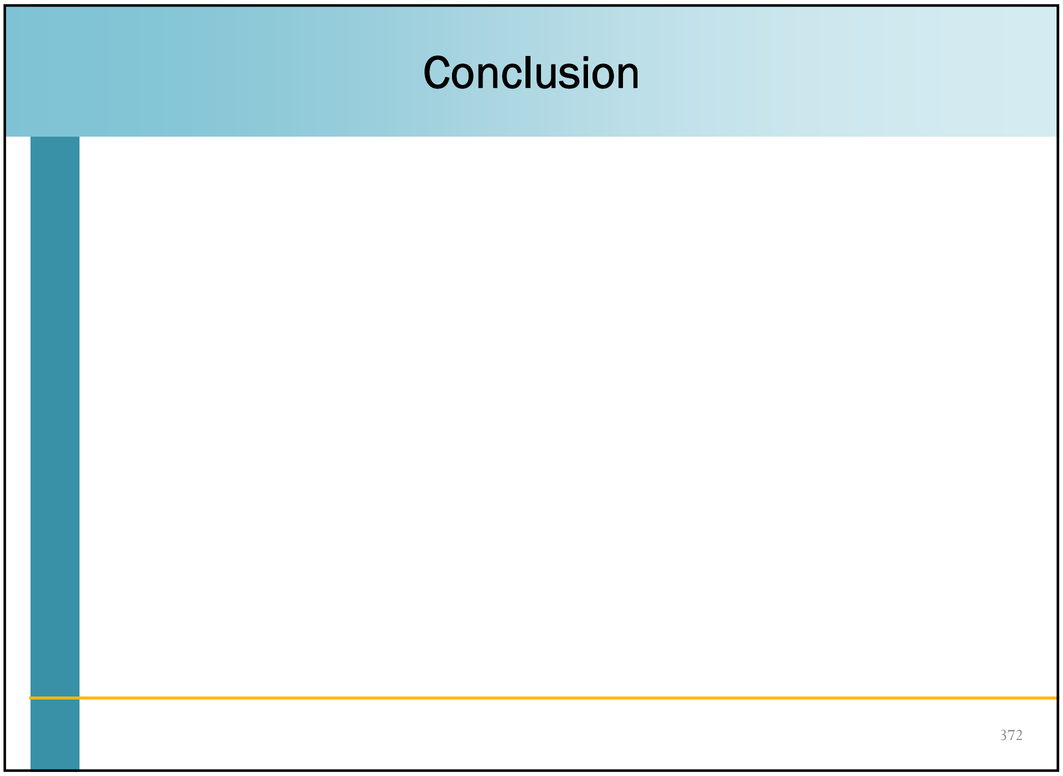
- Chapter: NgResource
- Demos/CustomAction

370



Exercise: Customized resources

- Chapter: NgResource
- Exercises/ImprovedResource



Conclusion