

23-Sep-2018

26-Oct-2018 first round has completed

.....

12- Nov-2018 second round has started..

Angular 6 by Example

Third Edition

Get up and running with Angular by building modern
real-world web apps



Packt

www.packt.com

By Chandermiani Arora and Kevin Hennessy

Angular 6 by Example
Third Edition

Get up and running with Angular by building modern
real-world web apps

Chandermanni Arora
Kevin Hennessy

Packt

BIRMINGHAM - MUMBAI

Angular 6 by Example Third Edition

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Amarabha Banerjee

Acquisition Editor: Larissa Pinto

Content Development Editor: Onkar Wani

Technical Editor: Akhil Nair

Copy Editor: Safis Editing

Project Coordinator: Devanshi Doshi

Proofreader: Safis Editing

Indexer: Rekha Nair

Graphics: Jason Monteiro

Production Coordinator: Nilesh Mohite

First published: October 2016

Second edition: October 2017

Third edition: **June 2018**

Production reference: 1190618

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78883-517-6

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Chandermani Arora is a software craftsman, with love for technology and expertise on web stack. With years of experience, he has architected, designed, and developed various solutions for Microsoft platforms. He has been building apps on Angular 1 since its early days. Having a passion for the framework every project of his has an Angular footprint.

He tries to support the platform in every possible way by writing blogs on various Angular topics or helping fellow developers on StackOverflow, where he is an active member on the Angular channel. He also authored the first edition of this book.

Writing this book has just been a surreal experience and I would like to thank everyone on the Packt team including the reviewers who have helped me with this book.

I also want to express my gratitude towards my wife, my daughter and everyone else in my family. I am blessed to have you all in my life.

Kevin Hennessy is a Senior Software Engineer with Applied Information Sciences. He has 20 years' experience as a developer, team lead, and solutions architect, working on web-based projects, primarily using the Microsoft technology stack. Over the last several years, he has presented and written about single-page applications and JavaScript frameworks, including Knockout, Meteor, and Angular. He has spoken about Angular at the All Things Open conference.

I would like to thank my wife, Mary Gene Hennessy. Her unstinting love and support (and editorial suggestions) through the time that I spent writing this book, have made me ever more aware and appreciate how truly amazing it is to be married to her. I also want to thank my son, Aidan, who has informally tested the code in this book. He is a recent college graduate and his enthusiasm for technology has inspired me in writing this book.

About the reviewer

Phodal Huang is a developer, creator, and author. He works in ThoughtWorks as a Senior Consultant, and focuses on IoT and frontend. He is the author of Design IoT System and Growth: Thinking in Full Stack in Chinese.

He is an open source enthusiast, and has created a series of projects in GitHub. After daily work, he likes to reinvent some wheels for fun. He created the micro-frontend framework `Mooa` for Angular. You can find out more wheels on his GitHub page, `/phodal`.

He loves designing, writing, hacking, traveling, you can also find out more about him on his personal website at `phodal(dot)com`.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

[Title Page](#)

[Copyright and Credits](#)

[Angular 6 by Example](#)

[Third Edition](#)

[Packt Upsell](#)

[Why subscribe?](#)

[PacktPub.com](#)

[Contributors](#)

[About the authors](#)

[About the reviewer](#)

[Packt is searching for authors like you](#)

[Preface](#)

[Who this book is for](#)

[What this book covers](#)

[To get the most out of this book](#)

[Download the example code files](#)

[Conventions used](#)

[Get in touch](#)

[Reviews](#)

[1. Getting Started](#)

[Angular basics](#)

[The component pattern](#)

[Using the component pattern in web applications](#)

[Why weren't components used before in Angular?](#)

[What's new that enables Angular to use the component pattern?](#)

[Web Components](#)

Angular and Web Components

Language support in Angular

ES2015

TypeScript

Putting it all together

Angular modules

The basic steps to building Angular applications

The customary Hello Angular app – Guess the Number!

Building Guess the Number!

Designing our first component

Developing our first component

Installing Bootstrap

What do we have so far?

The host file - index.html

Custom element

The component file

The import statement

Decorator

Defining the class

The module file

Bootstrapping

We're up and running!

Digging deeper

Interpolation

Tracking changes in the number of tries

Expressions

The safe navigation operator

[Data binding](#)

[Property binding](#)

[Event binding](#)

[Structural directives](#)

[Revisiting our app](#)

[Looking at how our code handles updates](#)

[Maintaining state](#)

[Component as the container for state](#)

[Change detection](#)

[Tools](#)

[Resources](#)

[Summary](#)

2. Building Our First App – 7 Minute Workout

What is 7 Minute Workout?

Downloading the code base

Setting up the build

Angular CLI

Code transpiling

Organizing code

Feature folders

The 7 Minute Workout model

First feature module

App bootstrapping

Exploring Angular modules

Comprehending Angular modules

Our first component - WorkoutRunnerComponent

Component lifecycle hooks

Building the 7 Minute Workout view

The Angular binding infrastructure

Interpolations

Property binding

Property versus attribute

Property binding continued...

Interpolation syntactic sugar over property binding

Quick expression evaluation

Side effect-free binding expressions

Angular directives

Target selection for binding

Attribute binding

Style and class binding

Attribute directives

Styling HTML with ngClass and ngStyle

Learning more about an exercise

Adding descriptions and video panels

Component with inputs

Structural directives

The ever-so-useful NgForOf

Asterisk (*) in structural directives

NgForOf performance

Angular security

Trusting safe content

OnChange life cycle event

Formatting exercise steps with innerHTML binding

Displaying the remaining workout duration using pipes

Angular pipes

Pipe chaining

Implementing a custom pipe - SecondsToTimePipe

Adding the next exercise indicator using ngIf

Pausing an exercise

The Angular event binding infrastructure

Event bubbling

Event binding an \$event object

Two-way binding with ngModel

Summary

3. More Angular – SPA and Routing

Exploring Single Page Application capabilities

The Angular SPA infrastructure

Angular routing

Angular router

Routing setup

Adding start and finish pages

Route configuration

Pushstate API and server-side url-rewrites

Rendering component views with router-outlet

Route navigation

The link parameter array

Using the router service for component navigation

Using the ActivatedRoute service to access route params

Angular dependency Injection

Dependency injection 101

Exploring dependency injection in Angular

Tracking workout history

Building the workout-history-tracker service

Integrating with WorkoutRunnerComponent

Injecting dependencies with Constructor Injection

Dependency injection in depth

Registering dependencies

Angular providers

Value providers

Factory providers

- Explicit injection using injector
 - Dependency tokens
 - Using `InjectionToken`
 - Using string tokens
 - Adding the workout history page
 - Sorting and filtering history data using pipes
 - The `orderBy` pipe
 - Pipe chaining with search pipe
 - Pipe gotcha with arrays
 - Angular change detection overview
- Hierarchical injectors
 - Registering component-level dependencies
 - Angular DI dependency walk
 - Dependency injection with `@Injectable`
 - Tracking route changes using the router service
- Fixing the video playback experience
 - Using thumbnails for video
 - Using the `ngx-modialog` library
 - Creating custom dialogs
- Cross-component communication using Angular events
 - Tracking exercise progress with audio
 - Building Angular directives to wrap HTML audio
 - Creating `WorkoutAudioComponent` for audio support
 - Understanding template reference variables
 - Template variable assignment
 - Using the `@ViewChild` decorator
 - The `@ViewChildren` decorator

[Integrating WorkoutAudioComponent](#)

[Exposing WorkoutRunnerComponent events](#)

[The @Output decorator](#)

[Eventing with EventEmitter](#)

[Raising events from WorkoutRunnerComponent](#)

[Component communication patterns](#)

[Injecting a parent component into a child component](#)

[Using component life cycle events](#)

[Sibling component interaction using events and template variables](#)

[Summary](#)

4. Personal Trainer

The Personal Trainer app - the problem scope

Personal Trainer requirements

The Personal Trainer model

Getting started with the code for Personal Trainer

Using the Personal Trainer model in Workout Builder services

The Personal Trainer layout

Personal Trainer navigation with routes

Getting started with Personal Trainer navigation

Introducing child routes to Workout Builder

Adding the child routing component

Updating the WorkoutBuilder component

Updating the Workout Builder module

Updating App Routing module

Putting it all together

Lazy loading of routes

Integrating sub- and side-level navigation

Sub-level navigation

Side navigation

Implementing workout and exercise lists

WorkoutService as a workout and exercise repository

Workout and exercise list components

Workout and exercise list views

Workouts list views

Exercises list views

Building a workout

```
Finishing left nav

Adding WorkoutBuilderService

Adding exercises using ExerciseNav

Implementing the Workout component

Route parameters

Route guards

    Implementing the resolve route guard

Implementing the Workout component continued...

Implementing the Workout template

Angular forms

    Template-driven and reactive forms

    Template-driven forms

    Getting started

        Using NgForm

        ngModel

            Using ngModel with input and textarea

            Using ngModel with select

Angular validation

ngModel

    The Angular model state

Angular CSS classes

Workout validation

    Displaying appropriate validation messages

    Adding more validation

    Managing multiple validation messages

        Custom validation messages for an&#xA0;exercise

Saving the workout
```

[More on NgForm](#)

[Fixing the saving of forms and validation messages](#)

[Reactive forms](#)

[Getting started with reactive forms](#)

[Using the FormBuilder API](#)

[Adding the form model to our HTML view](#)

[Adding form controls to our form inputs](#)

[Adding validation](#)

[Adding dynamic form controls](#)

[Saving the form](#)

[Custom validators](#)

[Integrating a custom validator into our forms](#)

[Configuration options for running validation](#)

[Summary](#)

5. Supporting Server Data Persistence

Angular and server interactions

Setting up the persistence store

Seeding the database

The basics of the HttpClient module

Personal Trainer and server integration

Loading exercise and workout data

Loading exercise and workout lists from a server

Adding the HttpClient module and RxJS to our project

Updating workout-service to use the HttpClient module and RxJS

Modifying getWorkouts() to use the HttpClient module

Updating the workout/exercise list pages

Mapping server data to application models

Loading exercise and workout data from the server

Fixing the builder services

Updating the resolvers

Fixing the Workout and Exercise components

Performing CRUD on exercises/workouts

Creating a new workout

Updating a workout

Deleting a workout

Fixing the upstream code

Using promises for HTTP requests

The async pipe

Cross-domain access and Angular

Using JSONP to make cross-domain requests

Cross-origin resource sharing

Handling workouts not found

Fixing the 7 Minute Workout app

Summary

6. Angular Directives in Depth

Classifying directives

Components

Attribute directives

Structural directives

Building a remote validator directive

Validating workout names using async validators

Building a busy indicator directive

Injecting optional dependencies with the @Optional decorator

Implementation one – using renderer

Angular renderer, the translation layer

Host binding in directives

Property binding using @HostBinding

Attribute binding

Event binding

Implementation two - BusyIndicatorDirective with host bindings

Directive injection

Injecting directives defined on the same element

Injecting directive dependency from the parent

Injecting a child directive (or directives)

Injecting descendant directive(s)

Building an Ajax button component

Transcluding external components/elements into a component

Content children and view children

Injecting view children using @ViewChild and @ViewChildren

Tracking injected dependencies with QueryList

[Injecting content children using @ContentChild and @ContentChildren](#)

[Dependency injection using viewProvider](#)

[Understanding structural directives](#)

[TemplateRef](#)

[ViewContainerRef](#)

[Component styling and view encapsulation](#)

[Overview of Shadow DOM](#)

[Shadow DOM and Angular components](#)

[Summary](#)

7. Testing Personal Trainer

The need for automation

Testing in Angular

Types of testing

Testing – who does it and when?

The Angular testing ecosystem

Getting started with unit testing

Setting up Karma and Jasmine for unit testing

Organization and naming of our test files

Unit-testing Angular applications

Unit-testing pipes

Running our test files

Unit-testing components

Angular testing utilities

Managing dependencies in our tests

Unit-testing WorkoutRunnerComponent

Setting up component dependencies

Mocking dependencies - workout history tracker

Mocking dependencies – workout service

Mocking dependencies - router

Configuring our test using TestBed

Starting unit testing

Debugging unit tests in Karma

Unit-testing WorkoutRunner continued...

Using Jasmine spies to verify method invocations

Using Jasmine spies to verify dependencies

[Testing event emitters](#)

[Testing interval and timeout implementations](#)

[Testing workout pause and resume](#)

[Unit-testing services](#)

[Mocking HTTP request/response with HttpTestingController](#)

[Unit-testing directives](#)

[The TestBed class](#)

[Testing remote validator](#)

[Getting started with E2E testing](#)

[Introducing Protractor](#)

[Setting up Protractor for E2E testing](#)

[Writing E2E tests for the app](#)

[Setting up backend data for E2E testing](#)

[More E2E tests](#)

[Testing Workout Runner](#)

[Using page objects to manage E2E testing](#)

[Summary](#)

8. Some Practical Scenarios

Building a new app

Seed projects

Angular performance

Byte size

Initial load time and memory utilization

The Angular rendering engine

Server-side rendering

Offloading work to a web worker

Performant mobile experience

Change detection improvements

Change detection

Change detection setup

When does change detection kick in?

How does change detection work?

Change detection performance

Using immutable data structures

Using Observables

Manual change detection

Handling authentication and authorization

Cookie-based authentication

Token-based authentication

Handling authorization

Adding authorization support

Sharing user authentication context

Restricting routes

Conditionally rendering content based on roles

Migrating AngularJS apps

Should I migrate?

Advantages of Angular

Developing AngularJS apps today for easy migration

One component per file

Avoiding inline anonymous functions

Avoiding \$scope!

Using controller as (controller aliasing) syntax everywhere

Avoiding ng-controller

Building using the AngularJS 1.5+ component API

What to migrate?

Preparing for Angular migration

Identifying third-party dependencies

jQuery libraries

AngularJS libraries

Choice of language

Migrating AngularJS's Personal Trainer

Setting up AngularJS's Personal Trainer locally

Identifying dependencies

Setting up the module loader

Enabling TypeScript

Adding Angular

The ngUpgrade Library

Bootstrapping the hybrid app

Injecting Angular components into AngularJS views

Migrating our first view to Angular component

[Injecting AngularJS dependencies into Angular](#)

[Registering Angular components as directives](#)

[Rules of engagement](#)

[AngularJS directives and Angular components](#)

[Resource sharing and dependency injection](#)

[Sharing an AngularJS service](#)

[Sharing an Angular service](#)

[Change detection](#)

[Migrating the start and finish pages](#)

[AngularJS directive upgrade](#)

[Replacing angular-translate with ngx-translate](#)

[Using ngDoBootstrap for initialization](#)

[Integrating the start and finish pages](#)

[Getting rid of angular-translate](#)

[Learning](#)

[Summary](#)

[Other Books You May Enjoy](#)

[Leave a review - let other readers know what you think](#)

Preface

Angular 6 is here, and we are super excited! This book allows us to reach out to you and lend a helping hand in your quest to learn Angular. Angular has gone mainstream and has become the ubiquitous platform for web and mobile development.

If you are an AngularJS developer, then there is loads of exciting stuff to learn, and there is a whole new world to explore for developers getting started. Getting started with Angular can be overwhelming even for a seasoned AngularJS developer. Too many terms will be thrown at you: such as TypeScript, Transpiler, Shim, Observable, Immutable, Modules, Exports, Decorators, Components, Web Component, and Shadow DOM. Relax! We are trying to embrace the modern web, and everything new that is here is to make our lives easier. Lots of these concepts are not specific to Angular itself but highlight the direction in which the web platform development is moving. We will try our best to present these concepts in a clear and concise manner, helping everyone understand how these pieces fit into this big ecosystem. Learning by examples has its advantages, for example, you will immediately get to see the concept explained in action. This book follows the same pattern as its predecessor. Using the **Do It Yourself (DIY)** approach, we build multiple simple and complex applications using Angular.

Who this book is for

Angular helps you build faster, more efficient, and more flexible cross-platform applications. Angular is now at release 6, with significant changes through its previous versions. This is a unique web development book that will help you get to grips with Angular and explore a powerful solution for developing single page applications.

What this book covers

[Chapter 1](#), *Getting Started*, introduces you to the Angular framework. We create a super simple app in Angular that highlights some core features of the framework.

[Chapter 2](#), *Building Our First App – 7 Minute Workout*, teaches us how to build our first real Angular app. In the process, we will learn more about one of the primary building blocks of Angular, components. We will also be introduced to Angular's templating constructs, databinding capabilities, and services.

[Chapter 3](#), *More Angular – SPA and Routing*, covers the routing constructs in the framework where we build multiple pages for *7 Minute Workout*. This chapter also explores a number of patterns around inter-component communication.

[Chapter 4](#), *Personal Trainer*, introduces a new exercise where we morph the *7 Minute workout* into a generic Personal Trainer app. This app has the capability to create new workout plans other than the original 7 minute workout. This chapter covers Angular's form capabilities and how we can use them to build custom workouts.

[Chapter 5](#), *Supporting Server Data Persistence*, deals with saving and retrieving workout data from the server. We augment Personal Trainer with persistence capabilities as we explore Angular's http client library and how it uses RxJS Observables.

[Chapter 6](#), *Angular Directives in Depth*, goes deep into the inner workings of Angular directives and components. We build a number of directives to support Personal Trainer.

[Chapter 7](#), *Testing Personal Trainer*, introduces you to the testing world in Angular. You build a suite of unit and end-to-end tests that verify the working of Personal Trainer.

[Chapter 8](#), *Some Practical Scenarios*, provides some practical tips and guidance around scenarios that we might encounter while developing apps on this

framework. We cover scenarios such as authentication and authorization, performance, and the most important case, migrating apps from AngularJS to the latest version of Angular.

To get the most out of this book

We will be building our apps in the TypeScript language; therefore, it would be preferable if you have an IDE that makes development with TypeScript easy. IDEs such as Atom, Sublime, WebStorm, and Visual Studio (or VS Code) are great tools for this purpose.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/chandermani/angular6byexample>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`codeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "We can see how the router combines the routes in `app.routes.ts` with the default route in `workout-builder.routes.ts`".

A block of code is set as follows:

```
| "styles": [
|   "node_modules/bootstrap/dist/css/bootstrap.min.css",
|   "src/styles.css"
| ],
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
| const routes: Routes = [
|   ...
|   { path: 'builder', loadChildren: './workout-builder/workout-
|     builder.module#WorkoutBuilderModule'},
|     { path: '**', redirectTo: '/start' }
| ];
```

Any command-line input or output is written as follows:

```
| ng new guessthenumber --inlineTemplate
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "With the Developer tools open in the Sources tab"



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

23-Sep-2018

12-Nov-2018

Getting Started

Developing applications in JavaScript is always a challenge. Due to its malleable nature and lack of type checking, building a decent-sized application in JavaScript is difficult. Moreover, we use JavaScript for all types of processes, such as **user interface (UI)** manipulation, client-server interaction, and business processing/validations. As a result, we end up with spaghetti code that is difficult to maintain and test.

Libraries such as jQuery do a great job of taking care of various browser quirks and providing constructs that can lead to an overall reduction in the lines of code. However, these libraries lack any structural guidance that can help us when the code base grows.

In recent years, JavaScript frameworks have emerged to manage this complexity. Many of these frameworks, including earlier versions of Angular, use a design pattern called **Model-View-Controller** to separate the elements of the application into more manageable pieces. The success of these frameworks and their popularity in the developer community have established the value of using this pattern.

Web development, however, is constantly evolving and has changed a lot since Angular was first introduced in 2009. Technologies such as Web Components, the new version of JavaScript (ES2015), and TypeScript have all emerged. Taken together, they offer the opportunity to build a new, forward-looking framework. And with this new framework comes a new design pattern—the component pattern.

This chapter is dedicated to understanding the component pattern and how to put it into practice as we build a simple app using Angular.

The topics that we will cover in this chapter are as follows:

- **Angular basics:** We will briefly talk about the component pattern that is used to build Angular applications
- **Building our first Angular app:** We will build a small game—*Guess the*

Number!—in Angular

- **An introduction to some Angular constructs:** We will review some of the constructs that are used in Angular, such as interpolation, expressions, and the data binding syntax
- **Change detection:** We will discuss how change detection is managed in an Angular app
- **Tools and resources:** Lastly, we will provide some resources and tools that will come in handy during Angular development and debugging

Angular basics

Let's get started by looking at how Angular implements the component pattern.

The component pattern

Angular applications use the component pattern. You may not have heard of this pattern, but it is all around us. It is used not only in software development but also in manufacturing, construction, and other fields. Put simply, it involves combining smaller, discrete building blocks into larger finished products. For example, a battery is a component of an automobile.

In software development, components are logical units that can be combined into larger applications. Components tend to have internal logic and properties that are shielded or hidden from the larger application. The larger application then consumes these building blocks through specific gateways, called **interfaces**, which expose only what is needed to make use of the component. In this way, the component's internal logic can be modified without affecting the larger application, as long as the interfaces are not changed.

Getting back to our battery example, the car consumes the battery through a series of connectors. If the battery dies, however, it can be replaced by an entirely new battery, as long as that battery has the same connectors. This means that the builder of the car does not have to worry about the internals of the battery, which simplifies the process of building the car. Even more importantly, the car owner does not have to replace their car every time the battery dies.

To extend the analogy, manufacturers of batteries can market them for a range of different vehicles, for example, ATVs, boats, or snowmobiles. So the component pattern enables them to realize even greater economies of scale.

Using the component pattern in web applications

As web applications continue to become more sophisticated, the need to be able to construct them out of smaller and discrete components becomes more compelling. Components allow applications to be built in a way that prevents them from becoming messes of spaghetti code. Instead, component-based design allows us to reason about specific parts of the application in isolation from the other parts, and then we can stitch the application together into a finished, whole through agreed-upon points of connection.

Also, maintenance costs are less because each component's internal logic can be managed separately without affecting the other parts of the application. And putting applications together using self-describing components makes the application easier to understand at a higher level of abstraction.

Why weren't components used before in Angular?

If this idea makes so much sense, why was the component pattern not adopted in earlier versions of Angular? The answer is that the technologies that existed when Angular was first released did not fully support the implementation of this pattern in web applications.

Earlier versions of Angular, however, made substantial steps in the direction of enabling more intelligent web application design and organization. For example, they implemented the MVC pattern, which separates an application into a model, view, and controller (you will see the use of the MVC pattern continuing within the components that we will build in Angular).

With the MVC pattern, the model is the data, the view is a web page (or a mobile app screen or even a Flash page), and the controller populates the view with data from the model. In this way, separation of concerns is achieved. Following this pattern along with an intelligent use of directives will get you pretty close to components.

So, the earlier versions of Angular allowed applications to be designed and built more logically. However, this approach was limited by the fact that the technologies used were not truly isolated. Instead, they all ended up being rendered without any true separation from other elements on the screen.

What's new that enables Angular to use the component pattern?

By contrast, the newest version of Angular embraces recently emerging technologies, which make it possible to implement the component pattern more fully. These technologies include Web Components, ES2015 (the new version of JavaScript), and TypeScript. Let's discuss what each of these technologies brings to the mix that makes this possible.

Web Components

Web Components is an umbrella term that actually covers four emerging standards for web browsers:

- Custom elements
- Shadow DOM
- Templates
- HTML imports



More information on Web Components can be found at <https://www.webcomponents.org/introduction>

Let's now discuss each of these in detail:

- **Custom elements** enable new types of DOM elements to be created other than the standard HTML tags such as `<div>` and `<p>`. You will see the use of these custom elements throughout this book. For example, the application that we are building in this chapter will have a root element named `<app-root>`, but you can give this element any name you like. Individual components will also use custom elements. For example, in the following chapters we are building a more sophisticated application that breaks the screen down into components. The header of the page will use a custom element `<abc-header>` to display its content (the prefix `abc` is unique to our application and helps to avoid naming collisions with native HTML elements or custom elements in other apps). The ability to add custom tags provides a location on the screen that can be reserved for binding a component. In short, this is the first step towards separating a component from the rest of the page and making it possible to become truly self-contained.
- **Shadow DOM** provides a hidden area on the page for scripts, CSS, and HTML. Markup and styles that are within this hidden area will not affect the rest of the page, and equally importantly they will not be affected by the markup and styles on other parts of the page. Our component can use this hidden area to render its display. So, this is the second step in making our component self-contained.
- **Templates** are fragments of HTML that do not initially render in a web

page, but can be activated at runtime using JavaScript. Many JavaScript frameworks already support some form of templating. Web Components standardize this templating and provide direct support for it in the browser. Templates can be used to make the HTML and CSS inside the Shadow DOM used by our component dynamic. So, this is the third step in making our component.

- The final standard that makes up Web Components is **HTML imports**. They provide a way to load resources such as HTML, CSS, and JavaScript in a single bundle. Angular does not use HTML imports. Instead, it relies on JavaScript module loading, which we will discuss a little later in this chapter.

Angular and Web Components

Web Components are not fully supported in current web browsers. For that reason, Angular components are not strictly Web Components. It is probably more accurate to say that Angular components implement the design principles behind Web Components. They also make it possible to build components that can run in today's browsers.



At the time of writing, Angular supports evergreen browsers, such as Chrome, Firefox, Safari, and Edge, as well as IE 9 and above. It also has support for Android and IOS. For a list of browsers supported by Angular, visit <https://angular.io/guide/browser-support>.

Therefore, throughout the rest of this book, we will focus on building Angular components and not Web Components. Despite this distinction, Angular components align closely with Web Components and can even inter-operate with them. As browsers begin to support Web Components more fully, the differences between Angular components and Web Components will begin to disappear. So, if you want to begin adopting the Web Component standards of the future, Angular provides you with the opportunity to do so today.

Language support in Angular

You can develop components with ES5 (the version of JavaScript supported in all current browsers), but Angular enhances the ability to develop components by adding support for key features that are found in the latest languages, such as ES2015 and TypeScript.

ES2015

ES2015 is the new version of JavaScript; it was approved in June 2015. It adds many improvements to the language, which we will see throughout this book, but the two that interest us the most at this point are the following:

- Classes
- Module loading

Classes did not previously exist in JavaScript. The key advantage of using them, now that they do exist, is that they offer a simple, clear syntax that we can use to create convenient containers for the code in our components. As you will find when you begin working on the applications in this book. Classes also provide a convenient shorthand designation for our components that makes it easier to stitch them together with each other through things such as dependency injection.



To be clear, JavaScript classes do not introduce something that is completely new. The Mozilla Developer Network (MDN) describes them as primarily syntactical sugar over JavaScript's existing prototype-based inheritance. For more information visit <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>.

We'll explore classes throughout the examples in this book. If you have not worked with object-oriented languages, you may not be familiar with classes, so we will cover them as we work through the examples in this chapter.

ES2015 also introduces a new approach to **module loading**. A module provides a way for JavaScript files to be encapsulated. When they are encapsulated, they do not pollute the global namespace and can interact with other modules in a controlled manner.

Once we have our modules defined, we need a way to load them into our application for execution. Module loading allows us to select just what we need for our application from the modules that make up Angular and other components that we create or use.

Currently, a range of approaches and libraries exists to support module loading

in JavaScript. ES2015 adds a new, consistent syntax for loading modules as part of the language. The syntax is straightforward and involves prefixing modules with the `export` keyword (or using the default export) and then using `import` to consume them elsewhere in our application.

ES 2015 module loading enables us to combine components into useful bundles or features that can be imported or exported within our applications. In fact, modules are at the core of Angular itself. We will see that modules are used extensively both in Angular itself and in the applications that we are building throughout this book.



It is important to understand that, while Angular uses syntax that has similarities to ES2015 module-loading syntax, Angular modules (which we will discuss a little later in this chapter) are not the same as JavaScript modules. For further details on these differences, see the Angular documentation at <https://angular.io/guide/architecture#ngmodules-vs-javascript-modules>.

Because ES2015 is not fully supported by today's browsers, we will need to convert ES2015 into ES5 in order to use features such as classes and module loading in our applications. We do this through a process called **transpilation**.

Transpilation is like compilation, except that instead of converting our code into a machine language as compilation does, transpilation converts one type of source code to another type of source code. In this case, it converts ES2015 to ES5. There are several tools called **transpilers** that enable us to do that. Common transpilers include Traceur and **Babel**. TypeScript (which we will discuss next) is also a transpiler, and it is the one that we will use for the examples in this book.

Once ES2015 is transpiled to ES5, we can then use a module loader such as **SystemJS** to load our modules. **SystemJS** follows the ES2015 syntax for module loading and gives us the ability to do module loading in today's browsers. Alternatively, we can use a module bundler such as **webpack** to load and combine our modules. For the projects in this book we will be using **webpack** to load, bundle, and deploy the modules in our applications.



*Since the release of ES2015, the schedule for releases of new versions of ECMAScript (the official name for JavaScript) is on an annual basis—so we now have **ES2016** and **ES2017** and will soon have **ES2018**. Since the features that we are highlighting throughout this book were first introduced in ES2015, we will be referring to ES2015 instead of any of the newer versions. The newer versions, however, are fully compatible with the language features that we are highlighting here and elsewhere in the book.*

TypeScript

TypeScript was created by Microsoft as a superset of JavaScript, which means that it contains the features of ES2015 (such as classes and module loading) and adds the following:

- Types
- Decorators

Types allow us to mark variables, properties, and parameters in our classes to indicate that they are numbers, strings, Booleans, or various structures such as arrays and objects. This enables us to perform type checking at design time to make sure that the proper types are being used in our application.

Decorators are simple annotations that we can add to our classes using the @ symbol along with a function. They provide instructions (called metadata) for the use of our classes. In the case of Angular, decorators allow us to identify our classes as Angular components. Decorators also enable us to specify a custom element to which to bind our component and to identify a template that adds an HTML view to our component. We will cover much more about the use of decorators as we go through this book.

Decorators are not part of ES2015, but are part of a proposal to include them in the JavaScript language in the future. They were added to TypeScript as part of a collaboration between Microsoft and Google. As mentioned earlier, TypeScript compiles into ES5, so we are able to use both types and decorators in browsers that do not fully support ES2015 or the proposed standard for decorators.



As mentioned previously, it is not necessary to use either ES2015 or TypeScript with Angular. However, we think that you will see the advantages of using them as we work through the examples in this book.

Putting it all together

By following the Web Component standards and adding support for ES2015 and TypeScript, Angular gives us the ability to create web applications that implement the component design pattern. These components help realize the vision behind the standards of building large-scale applications through collections of self-describing and self-contained building blocks.

We hope that you will see in the examples in this book that Angular enables components to be constructed in a straightforward and declarative way that makes it easier for developers to implement them. As we proceed through the examples in this book, we will highlight where each of these technologies is being used.

Angular modules

Components are the basic building block of an Angular application. But how do we then organize these building blocks into complete applications? Angular modules provide the answer to this question. They enable us to combine our components into reusable groups of functionality that can be exported and imported throughout our application. For example, in a more sophisticated application we would want to have modules for things such as authentication, common utilities, and external service calls. At the same time, modules enable us to group features within an application in a way that allows us to load them on demand. This is called lazy loading, a topic that we will cover in [chapter 4, Building Personal Trainer](#).

Each Angular application will have one or more modules that contain its components. Angular has introduced `NgModule` as a way to conveniently specify the components that make up a module. Every Angular application must have at least one of these modules—the root module.



Angular itself is built as modules that we import into our application. So you will see the use of modules all over as you build Angular apps.

The basic steps to building Angular applications

To sum up: at a basic level, you will see that to develop applications in Angular, you will do the following:

1. Create components
2. Bundle them into modules
3. Bootstrap your application

The best way to understand Angular and the component design pattern is by seeing it in action. Hence, we are going to build our first Hello World app in Angular. This app will help you become familiar with the Angular framework and see the component design pattern in action.

Let's get started doing that.

The customary Hello Angular app – Guess the Number!

As our first exercise, we want to keep things simple, but still showcase the framework's capabilities. Therefore, we are going to build a very simple game called *Guess the Number!*. The objective of the game is to guess a random computer-generated number in as few tries as possible.

This is how the game looks:

Guess the Number !

Guess the computer generated random number between 1 and 1000.

Your Guess: Verify Restart

No of guesses : 0

Let's now build *Guess the Number!*.

Building Guess the Number!

The standard practice while building user interfaces is to build them top-down. Start by designing the UI and then plug in the data and behavior according to your needs. With such an approach, the UI, data, and behavioral aspects of the app are all tightly coupled, which is a less than ideal situation!

With component-based design, we work differently. We start by looking at the UI and expected behavior, and then we encapsulate all of this into a building block that we call a **component**. This component is then hosted on our page. Within the component, we separate the UI into a view and the behavior into a class, with the appropriate properties and methods needed to support the behavior. If you are not familiar with classes, don't worry. We'll be discussing what they are in detail as we move through the example.

Okay, so let's identify the UI and behavior that we will need for our application.

Designing our first component

To determine what needs to go into our component, we will start by detailing the features that we want the app to support:

- Generating random numbers (`original`)
- Providing input for a user to guess the value (`guess`)
- Tracking the number of guesses already made (`noOfTries`)
- Giving the user hints to improve their guess based on their input (`deviation`)
- Giving a success message if the user guesses the number correctly (`deviation`)

Now that we have our features, we can determine what we need to display to the user and what data we need to track. For the preceding feature set, the elements in parentheses denote the properties that will support those features and will need to be included in our component.

Designing the component is a very crucial process. If it is done right, we can logically organize our application in a way that makes it understandable and easy to maintain.



While building any app, we urge you to first think about the functionality you want to offer, and then the data and behavior that can support the functionality. Lastly, think about how to build a user interface for it. This is a good practice irrespective of the library or framework that you use to build your app.

Developing our first component

Now that we have a design for our first component, we'll start developing it using the **Angular Command Line Interface (Angular CLI)**. The Angular CLI enables us to start building Angular applications and deploying them through a series of console commands. We'll be covering the **Angular CLI** in greater detail in future chapters. For now, we will install it and use it to generate a basic application as the beginning point for our first component.

To use the **Angular CLI** you must first install **Node.js** and **npm** (**Node's** package manager). **Node** is available cross-platform and you can download it from <http://nodejs.org>. Installing **Node** also installs **npm**. For this book, we are using **Node.js** version 8.9.4 and **npm** version 5.6.0. You can find more information about installing **Node** and updating **npm** to the latest version at <https://docs.npmjs.com/getting-started/installing-node>.

Once **Node** and **npm** are installed, open a Command Prompt and type the following: **npm install -g @angular/cli**

This installs the **Angular CLI** that we will use to start building our application. Now from a directory on your local machine, enter the following commands: **ng new guessthenumber --inlineTemplate**
cd guessthenumber
ng serve

With the first command the **Angular CLI** will create a new Angular project on your local machine (the `--inlineTemplate` flag creates a template within our component, which is perfect for what we want to show in this chapter). With the second command, you are navigating to the directory that the **Angular CLI** has created for your new project. The third command launches the application, which you can view at <http://localhost:4200/>. If you do that you should see a standard default Angular page in the browser.

Installing Bootstrap

There is one more step before we build out the specifics of our application. Let's add the Bootstrap library to enhance the look and feel of our application. First, stop the application by typing *Ctrl + C* in the Terminal from which it was launched and enter *Y* when asked if you want to terminate the batch job. Next from the `guessthenumber` directory, enter the following command: **npm install bootstrap --save**

This will install the latest release of Bootstrap (which at the time of writing was version 4.0.0). You may see a few warning messages about unmet dependencies. You can ignore them.

Next configure your new project to include the Bootstrap style sheet:

1. In the `guessthenumber` directory find and open the file `angular.json`
2. In that file find the `projects` property, which contains the settings in our new project
3. Then find the `styles` property within `architect.build.options` and you will see that it contains an array that holds `styles.css`, the default style sheet for our new project
4. Add to that array the location of the `bootstrap.min.css` style sheet like so:

```
"styles": [  
    "node_modules/bootstrap/dist/css/bootstrap.min.css",  
    "src/styles.css"  
],
```



*These instructions for including Bootstrap using the **Angular CLI** can be found at <https://github.com/angular/angular-cli/wiki/stories-include-bootstrap>.*

What do we have so far?

If you take a look in the `guessthenumber` directory that the **Angular CLI** has been created, you will see a large number of files. This may look overwhelming at first, but the important thing to understand is that the **Angular CLI** has generated all these files for us with just a few command line statements. In that way it makes getting started with an Angular application much smoother and easier. It takes the grunt work out of the process and enables us able to build and serve our application with minimal effort. In this chapter, we will be focusing on just a few files that we will need to touch in order to create our application.



If you are running the application in `Internet Explorer`, there is one file that you will need to look at—`polyfill.ts`. This adds various other files that are needed to run the application in Internet Explorer. You will need to uncomment several sections in that file to add these necessary files. Instructions for doing this are contained in the file itself.

Before turning to building out the specifics of our application, let's take a look at one of the key files that will be used to get our application up and running.



Downloading the example code The code in this book is available on GitHub at <https://github.com/chandermani/angular6byexample>. It is organized in checkpoints that allow you to follow along step by step as we build our sample projects in this book. The branch to download for this chapter is GitHub's Branch: `checkpoint1.1`. Look in the `guessthenumber` folder for the code we are covering here. If you are not using Git, download the snapshot of Checkpoint 1.1 (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/tree/checkpoint1.1>. Refer to the `readme.md` file in the `guessthenumber` folder when setting up the snapshot for the first time.

The host file - index.html

Navigate to the `src` folder in the `guessthenumber` directory and open `index.html`. You will see the following:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Guessthenumber</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

`index.html` is the host file for our application. It will be launched by the browser when the application is first run and will host the components in our application. If you have any exposure to web development, most of the HTML code in this file should look familiar . It has standard `html`, `head`, and `body` tags along with a couple of optional tags, one a meta tag for the viewport, which configures how the app will display in a mobile device, and the other a link to an Angular favicon image that will display on the tab in the browser in which the application is loaded.

Custom element

However, there is one more important tag on the page that may not look as familiar to you: <app-root></app-root>

This tag is a **custom element**. It instructs Angular where to inject the component that we will be building.



Guess the Number! and all the other apps that are part of this book have been tested against the Angular 6 final release.

The component file

Now let's turn to building out the specifics of our application. Given the previous discussion of the component pattern, you will not be surprised that to do this we will be constructing a component. In this case our application will be simple enough that we will need only one component (later in this book you will see the use of multiple components as we build more complex applications). The Angular CLI has already given us a start by generating a component file for us. Of course, that file does not contain any of the particulars of our application, so we will have to modify it. To do that navigate to the `src` folder in the `app` directory and open `app.component.ts`.

The import statement

At the top of the page, you will find the following line: `import { Component } from '@angular/core';`

This is an import statement. It tells us what **modules** we will be loading and using in our component. In this case, we are selecting one module that we need to load from `@angular:Component`. Angular has many other modules, but we load only what we need.

You'll notice that the location from which we are importing is not identified as a path or directory within our application. Instead, it is identified as `@angular/core`. Angular has been divided into **barrel modules** that are prefixed with `@angular`.

These barrels combine several modules that are logically related. In this case, we are indicating that we want to import the `core` barrel module, which in turn brings in the `Component` module.

The Angular documentation describes a barrel as:



A way to roll up exports from several ES2015 modules into a single convenient ES2015 module. The barrel itself is an ES2015 module file that re-exports selected exports of other ES2015 modules.

For more information about barrels, see <https://angular.io/guide/glossary#barrel>.

Decorator

Next, replace the code block that starts with `@component` with the following:

```
@Component({
  selector: 'app-root',
  template: `
    <div class="container">
      <h2>Guess the Number !</h2>
      <div class="card bg-light mb-3">
        <div class="card-body">
          <p class="card-text">Guess the computer generated random number between 1
           and 1000.</p>
        </div>
      </div>
      <div>
        <label>Your Guess: </label>
        <input type="number" [value]="guess" (input)="guess = $event.target.value" />
        <button (click)="verifyGuess()" class="btn btn-primary btn-sm">Verify</button>
        <button (click)="initializeGame()" class="btn btn-warning btn-
           sm">Restart</button>
      </div>
      <div>
        <p *ngIf="deviation<0" class="alert alert-warning">Your guess is higher.</p>
        <p *ngIf="deviation>0" class="alert alert-warning">Your guess is lower.</p>
        <p *ngIf="deviation==0" class="alert alert-success">Yes! That's it.</p>
      </div>
      <p class="text-info">No of guesses :
        <span class="badge">{{noOfTries}}</span>
      </p>
    </div>
  `)
})
```

This is the decorator for our component and it is placed directly above the class definition, which we will discuss soon. The `@` symbol is used to identify a decorator. The `@component` decorator has a property called selector, and you may not be surprised to see that it is set to the `<app-root>` tag in our HTML page. This setting tells Angular to inject this component into that tag on the HTML page.

The decorator also has a property called `template`, and this property identifies the HTML markup for our component. Notice the use of back ticks (introduced by ES2015) for rendering the template string over multiple lines. Alternatively, we can set a `templateUrl` property that would point to a separate file.

Defining the class

Now replace the code block that begins with `export class AppComponent` with the following:

```
export class AppComponent {
  deviation: number;
  noOfTries: number;
  original: number;
  guess: number;

  constructor() {
    this.initializeGame();
  }
  initializeGame() {
    this.noOfTries = 0;
    this.original = Math.floor((Math.random() * 1000) + 1);
    this.guess = null;
    this.deviation = null;
  }
  verifyGuess() {
    this.deviation = this.original - this.guess;
    this.noOfTries = this.noOfTries + 1;
  }
}
```

If you have been developing in ES5, the version of JavaScript that is supported in all current browsers, you may not be familiar with the use of classes here. So, we will take a few moments to walk through what makes up a class (for those of you who have developed using an object-oriented programming language, such as C# or Java, this should be familiar territory).

The class file holds the code that we will use to run our component. At the top, we give the class a name, which is `AppComponent`. Then, inside the curly braces, we have four lines that declare the properties for our class. These are similar to ES5 variables, and we will use them to hold the values that we will need to run the application (you'll notice that these are the four values that we identified when we designed our component).

What makes these properties different from standard JavaScript variables is that each property name is followed by `:` and a number. These set the type of the property. In this case, we are indicating that each of these four properties will be set to the `number` type, which means we are expecting the values of all of these properties to be numbers. The ability to specify types for our properties is

provided by TypeScript and it is not available in standard JavaScript.

As we move down, we will see three blocks of script that have names, followed by parentheses, and then curly braces with several lines of script inside them. These are the methods for our class, and they contain the operations that our component will support. They are a lot like standard JavaScript functions.

The first of these methods is `constructor()`, which is a special method that will run when an instance of our component is first created. In our example, the constructor does only one thing when the class is created; it calls another method in our class, called `initializeGame()`.

The `initializeGame()` method sets the starting values of the four properties in the class using the assignment operator `=`. We set these values to `null` or `zero`, except for `original`, in which we use a random number generator to create the number to be guessed.

The class holds one more method called `verifyGuess()`, which updates the `deviation` and `nooftries` properties. This method is not being called from within the component class; instead, it will be called from the view, as we will see when we examine the view more closely later. You'll also notice that our methods refer to properties in the same class by prepending `this` to them.

The module file

As we mentioned earlier, every Angular component must be contained within an Angular module. This means that at a minimum we must add at least one Angular module file to the root of our application. We call this the **root module**. For a simple application like *Guess the Number!*, the root module may be the only module we will need. However, as an Angular application increases in size, it will often make sense to have multiple Angular module files broken down by features. We will cover that situation as we move into building more complex applications in later chapters in this book.

Let's go ahead and take a look at our Angular module file. Again the Angular CLI has created this file for us. Open `app.module.ts` in the `app` directory within the `sre` folder and you will see the following:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The first two statements import `BrowserModule` and `NgModule`. Notice that, while `NgModule` is being imported from `@angular/core`, `BrowserModule` is being imported from a different module: `@angular/platform-browser`. What's significant here is that the import is not coming from `@angular/core`, but from a separate module that is specific to browser-based applications. This is a reminder that Angular can support devices other than browsers, such as mobile devices, hence the need to place `BrowserModule` into a separate module.

The other import in this file is our component `AppComponent`. If you go back to that component you will notice that `export` is added in front of the class definition,

which means we are using module loading within our own application.

We next define a new component `AppModule`. There is nothing in the class itself other than a few imports and a decorator: `@NgModule`. We can use this decorator to configure the module in our application. The first property is `declarations` and with that property we provide an array of the components that will be used in our application. In this case, we have just one component: `AppComponent`.

We next add imports, which in this case include the `BrowserModule`. As the name suggests, this module will provide the functionality needed to run our application in a browser. The next property is `providers`. This property is used to register providers (such as services and other objects) that will be available to be used throughout our application through dependency injection. We have no need for providers in the simple application we are building here, so this property is empty. We will be discussing providers and dependency injection in detail in [Chapter 3, More Angular – SPA, Routing](#).

Finally, we set the `bootstrap` property. This indicates the first component that will be loaded when our application starts up. Again this is the `AppComponent`.

With this configuration in place, we are now ready to bootstrap our component.

Bootstrapping

The class definition for `AppComponent` operates as a blueprint for the component, but the script inside it does not run until we have created an instance of the component. In order to run our application then, we need something in our application that creates this instance. The process of doing that requires us to add code that bootstraps our component.

In the `src` folder, look for a file named `main.ts`. Open it and you will see the following code:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

As you can see, we are importing `enableProdMode` from `@angular/core` and the `platformBrowserDynamic` module from `@angular/platform-browser-dynamic`. Like the import of `BrowserModule` in the `appModule` file, this latter import is specifically for browser-based applications. Next we add an import of our `AppModule` and a file called `environment` that is located in the `environments` directory of our application.

In the next lines of code we check to see if the constant `environment` in the `environment` file has its `production` property set to `true`, and if so, call `enableProdMode()`, which as the name suggests enables production mode. The default setting for `environment.production` is `false`, which is fine for our purposes here since we are not running the application in production mode.



If you open `environments.ts`, you will see some comments that provide guidance for overwriting the settings in this file as part of the build process. We won't be covering the Angular build process until chapter 2, Building Our First App – 7 Minute Workout; so we won't cover that material here.

Finally, we call the `platformBrowserDynamic().bootstrapModule` method with our `AppModule`

as a parameter. The `bootstrapModule` method then creates a new instance of our `AppModule` component, which in turn initializes our `AppComponent`, which we have marked as the `component to bootstrap`. It does that by calling our component's `constructor` method.

We're up and running!

Well, the app is complete and ready to be tested! From the `guessthenumber` directory again type the following: `ng serve`

The app should appear on your browser.



If you are having trouble running the app, you can check out a working version available on GitHub at <https://github.com/chandermani/angular6byexample>. If you are not using Git, download the snapshot of Checkpoint 1.1 (a ZIP file) from the following GitHub location: https://github.com/chandermani/angular6byexample/tree/checkpoint1_1. Refer to the `readme.md` file in the `guessthenumber` folder when setting up the snapshot for the first time

If we glance at our component file and template, we should be mightily impressed with what we have achieved. We are not writing any code to update the UI when the application is running. Still, everything works perfectly.

Digging deeper

To understand how this app functions in the Angular context, we need to delve a little deeper into our component. While the class definition in the component is pretty simple and straightforward, we need to look more closely at the HTML in the template to understand how Angular is working here. It looks like standard HTML with some new symbols, such as `[]`, `()`, `{ { }}`, and `{} { }`.

In the Angular world, these symbols mean the following:

- `{ { }}` and `{} { }` are interpolation symbols
- `[]` represents property bindings
- `()` represents event bindings

Clearly, these symbols have some behavior attached to them and seem to be linking the view HTML and component code. Let's try to understand what these symbols actually do.

Interpolation

Look at this HTML fragment from the template in `app.component.ts`:

```
| <p class="text-info">No of guesses :  
|   <span class="badge">{{noOfTries}}</span>  
| </p>
```

The term `noOfTries` is sandwiched between two interpolation symbols.

Interpolation works by replacing the content of the interpolation markup with the value of the expression (`noOfTries`) inside the interpolation symbol. In this case, `noOfTries` is the name of a component property. So the value of the component property will be displayed as the contents inside the interpolation tags.

Interpolations are declared using this syntax: `{{expression}}`. This expression looks similar to a JavaScript expression, but is always evaluated in the context of the component. Notice that we did not do anything to pass the value of the property to the view. Instead, the interpolation tags read the value of the property directly from the component without any need for additional code.

Tracking changes in the number of tries

Another interesting aspect of interpolation is that changes made to component properties are automatically synchronized with the view. Run the app and make some guesses; the `noOfTries` value changes after every guess and so does the view content:



Interpolation is an excellent **debugging** tool in scenarios where we need to see the state of the model. With interpolation, we don't have to put a breakpoint in code just to know the value of a component property. Since interpolation can take an expression, we can pass a component's method call or a property and see its value.

Expressions

Before going any further, we need to spend a few moments understanding what template expressions are in Angular.

Template expressions in Angular are nothing but pieces of plain JavaScript code that are evaluated in the context of the component instance associated with the template in which they are used. But as the documentation at <https://angular.io/docs/ts/latest/guide/template-syntax.html#template-expressions> makes it clear, there are some differences:

- Assignment is prohibited
- The `new` operator is prohibited
- The bitwise operators `|` and `&` are not supported
- Increment and decrement operators, `++` and `--`, aren't supported
- Template expression operators, such as `|` and `?`, add new meanings

In the light of our discussion on component-based design, you probably won't be surprised to learn that the documentation also makes some things clear; template expressions cannot:

- Refer to anything in the global namespace
- Refer to a window or document
- Call `console.log`

Instead, these expressions are confined to the expression context, which is typically the component instance supporting a particular template.

However, these limitations do not stop us from doing some nifty stuff with expressions. As we can see in the following examples, these all are valid expressions: // outputs the value of a component property `{{property}}` // adds two values `{{ 7 + 9 }}` //outputs the result of boolean comparison. Ternary operator `{{property1 >=0?'positive': 'negative'}}` //call a component's testMethod and outputs the return value `{{testMethod()}}`

 Having looked into expressions, we strongly advise you to keep your expressions simple, thus keeping the HTML readable. The `*ngIf="formHasErrors()"` expression is always better than `*ng-`



if="name==null || email==null || emailformatInvalid(email) || age < 18". So, when an expression starts to become complex, move it into a method in your component.

The safe navigation operator

Before we move on there is one other expression that we should touch on: the Angular safe navigation operator (`?.`). This operator provides a convenient way to check for null values in lengthy property paths, like so:

```
| {{customer?.firstName }}
```

If the safe navigation operator finds a null value (here the `customer`), it stops processing the path, but lets the application continue running. Without it, the application will crash when it reaches anything after the first null (here the customer name) and the view will not display. The safe navigation operator is especially helpful in situations where you are loading data asynchronously and it might not be immediately available to the view. The safe navigation operator will prevent the application from crashing and then load the data when it is available.

Data binding

Learning interpolation and expressions was easy. Now let's look at another framework construct that is being used by our sample app-data binding. We will be covering data binding in far more detail in the upcoming chapters. At this point, we will just touch briefly on the bindings that are used in the sample app we are building.

Property binding

If we look through the template in `app.component.ts`, we will see several places where square brackets `[]` are used. These are **property bindings**.

Let's look at the first of the bindings that we created: `<input type="number" [value]="guess" (input)="guess = $event.target.value" />`

This binding works by linking the value of the `guess` property in our component class to the value of the input field in the view. The binding is dynamic; so, as the value of the `guess` property changes, the value of the input field will be synchronized to the same value; and we do not have to write any code to do that.

At the outset, when we initialize the game, this property is set to null in the initialization method of the component class, so we will not see anything in the input field. However, as the game progresses, this number will be updated with the value of the `guess` as it changes.

Event binding

Looking again at the template in `app.component.ts`, we find several places where parentheses `()` appear. These are **event bindings**.

Let's look at the HTML code line that we created for the first of these event bindings. It should be familiar since the event binding is on the same tag that we first looked at for property binding: the `input` tag:

```
|<input type="number" [value]="guess" (input)="guess = $event.target.value" />
```

In this case, the `input` event of the `input` element is bound to an expression. The expression sets the `guess` property in our component class to `$event.target.value`, which is the value being entered by the user. Behind the scenes, when we use this syntax, Angular sets up an event handler for the event that we are binding to. In this case, the handler updates the `guess` property in our component class whenever the user enters a number in the `input` field.

There are a couple of other places in our code where the `()` parentheses appear:

```
|<button (click)="verifyGuess()" class="btn btn-primary btn-sm">Verify</button>
|<button (click)="initializeGame()" class="btn btn-warning    btn-sm">Restart</button>
```

These two event bindings tie the `click` events for the buttons on the screen to methods in our component. So in this case, behind the scenes, Angular sets up event handlers that bind directly to the methods in our component. When the Verify button is clicked, the `verifyGuess` method is called, and when the Restart button is clicked, the `initializeGame` method is called.

This is advantage of functional programming.

As you work through the samples in this book, you will see many places where the `[]` tags for property bindings are combined with the `()` tags for events. In fact, this pairing is so common that, as we will see later, Angular has come up with a shorthand syntax to combine these tags into one.

Structural directives

Next, we'll examine something that looks similar to data binding but incorporates an Angular feature that we haven't seen before: **structural directives**:

```
<div>
  <p *ngIf="deviation<0" class="alert alert-warning"> Your guess is higher.</p>
  <p *ngIf="deviation>0" class="alert alert-warning"> Your guess is lower.</p>
  <p *ngIf="deviation==0" class="alert alert-success"> Yes! That's it.</p>
</div>
```

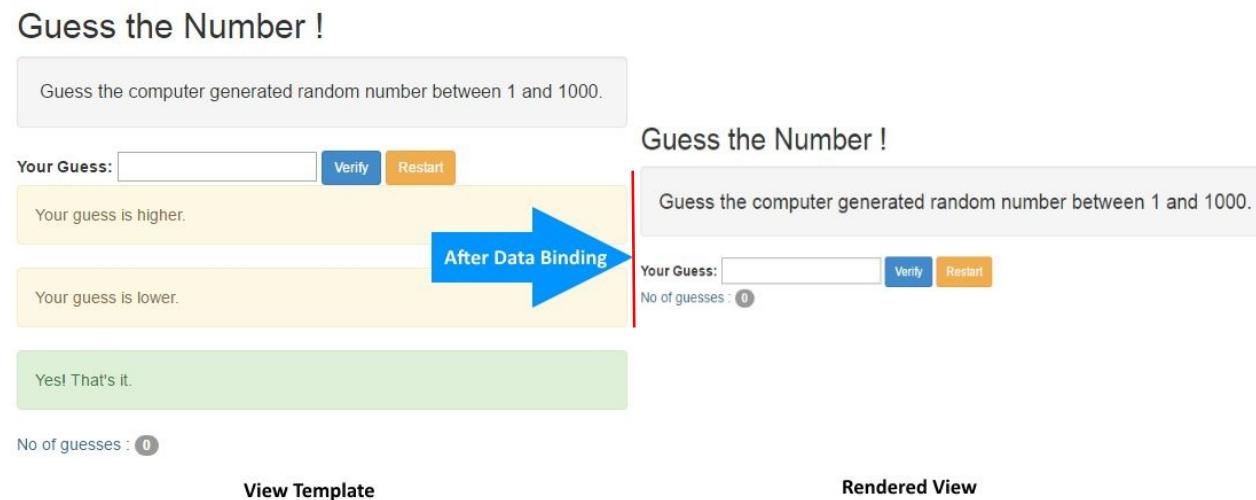
*ngIf inside the <p> tags is the ngIf structural directive. Structural directives allow us to manipulate the structure of DOM elements. The ngIf directive removes or adds DOM elements based on the result of an expression that is assigned to it.

 *The asterisk * in front of `ngIf` is a simplified syntax that Angular, under the hood, expands into `ng-template`, which is Angular's implementation of the Web Components template that we discussed earlier. We'll be learning a lot more about this syntax and about structural directives in the next chapter.*

In this case we are using `ngIf` with a simple expression, similar to the types of expression we saw with interpolation. The expression resolves to either `true` or `false` based on the value of the guess being made and its relation (higher, lower, or equal) to the correct number. It then assigns that result to `ngIf`, which will either add the DOM element if the result is `true` or remove it if it is `false`.

Revisiting our app

So now that we have looked more closely at what makes up our view, let's take another look at our app when it is up and running. When we run our app, Angular binding starts up once the browser has rendered the raw HTML in our view. The framework then compiles this view template and, in the process, sets up the necessary binding. Next, it does the necessary synchronization between our component class and the view template that produces the final rendered output. The following screenshot depicts the transformations that happen to the view template after data binding is done for our app:



We can ourselves see the untransformed view template of the app (what is shown on the left-hand side of the preceding screenshot) by removing the three `*ngIf` directives and the expressions assigned to them from the paragraphs below the input box and refreshing the app in the browser.

Angular differs from other template frameworks, in that these bindings between a component and its view are live. Changes made to the properties of the component update the view. Angular never regenerates the HTML; it just works on the relevant part of the HTML and updates only the HTML elements that need to change as component properties change. This data binding capability makes Angular an exceptional view templating engine too.

Looking at how our code handles updates

If we go back and look at the code for our class, we will see that the properties and methods in the class do not directly reference the view. Instead, the methods simply operate on the properties in the class. As a consequence, the code for our class is more readable, hence more maintainable (and of course, testable):

Guess the Number !

The screenshot shows the browser's developer tools with the 'Sources' tab selected, displaying the `app.component.ts` file. The code defines a class `AppComponent` with properties like `title`, `deviation`, `noOfTries`, `original`, `guess`, and methods `constructor()`, `initializeGame()`, `verifyGuess()`, and `getDeviation()`. Annotations highlight specific parts of the code and the UI:

- A red arrow points from the `noOfTries` property in the code to the "No of guesses : 0" text in the UI.
- A blue arrow points from the `guess` property in the code to the "Your Guess:" input field in the UI.
- A blue arrow points from the `guess` property in the code to the `<input type="number" [value]="guess" (input)="guess = $event.target.value" />` line in the template.
- A blue arrow points from the `guess` property in the code to the `this.guess = null;` line in the `initializeGame()` method.
- A blue arrow points from the `guess` property in the code to the `this.guess` parameter in the `verifyGuess()` method.
- A blue arrow points from the `Methods operate on properties` annotation to the `this.guess` parameter in the `verifyGuess()` method.
- A blue arrow points from the `property binding` annotation to the `<input type="number" [value]="guess" (input)="guess = $event.target.value" />` line in the template.

So far, we have discussed how Angular updates the view based on changes in a component's properties. This is an important concept to understand as it can save us from countless hours of debugging and frustration. The next section is dedicated to learning about change detection and how these updates are managed.

Maintaining state

First, let's look at how we maintain state in our Angular application. Since Angular apps are dynamic and not static, we need to understand the mechanisms that are used to make sure that these dynamic values are kept up to date as the data in an application gets updated. For example, in our application, how does the number of guesses get updated on the screen? How does the application decide to display the correct message about whether the guess is correct based on the user input?

Component as the container for state

Since we have been emphasizing so far that Angular uses the component design pattern, you will probably not be surprised to know that the basic container for the application state is the component itself. This means that when we have a component instance, all the properties in the component and their values are available for the template instance that is referenced in the component. At a practical level, this means that we can use these values directly in expressions and bindings in the template without having to write any plumbing code to wire them up.

In the sample app, for example, to determine what message to display, we can use `deviation` directly in the template expression. Angular will scan our component to find a property with that name and use its value. The same is true for `noOfTries`; Angular will look for the value of this property within our component and then use it to set its value in the interpolation within the template. We don't have to write any other code:

```
<div>

    <p *ngIf="deviation<0" class="alert alert-warning"> Your guess is higher.</p>

    <p *ngIf="deviation>0" class="alert alert-warning"> Your guess is lower.</p>

    <p *ngIf="deviation==0" class="alert alert-success"> Yes! That's it.</p></div>

    <p class="text-info">No of guesses :

        <span class="badge">{{noOfTries}}</span>

    </p>

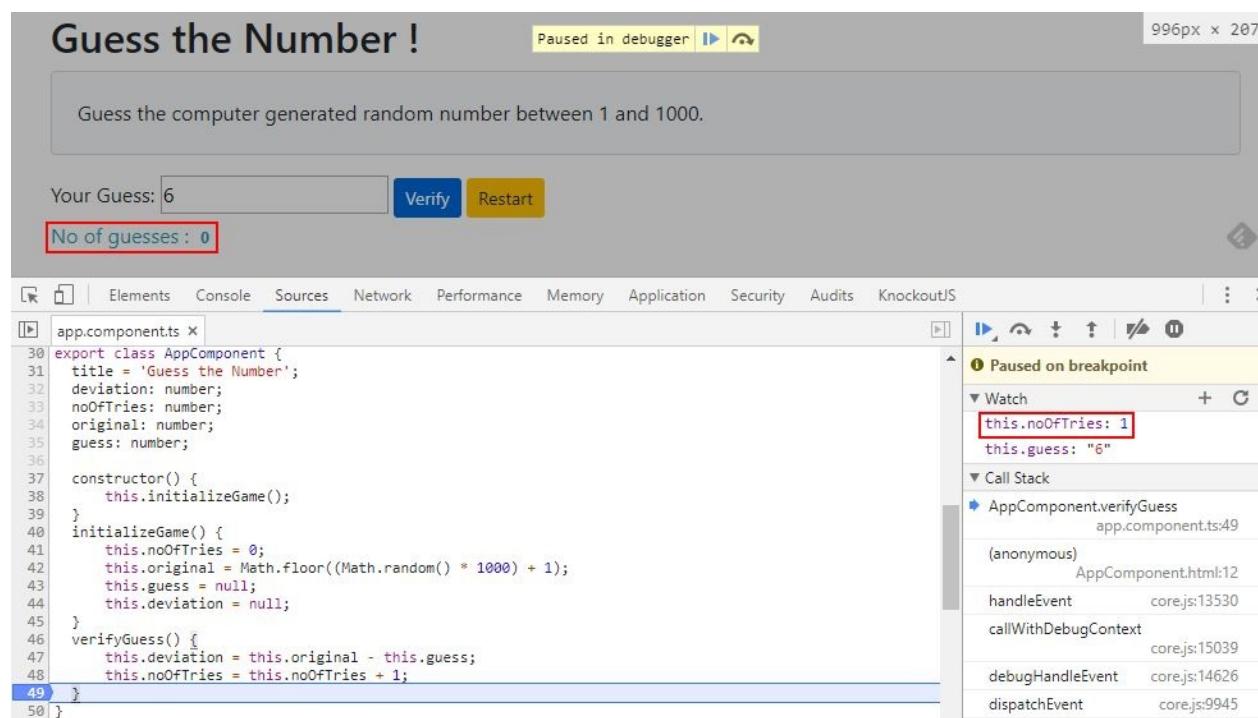
</div>
```


Change detection

So how does Angular keep track of changes in our component as it runs? So far, it appears as if this is all done by magic. We just set up our component properties and methods, and then we bind them to the view using interpolation along with property and event binding. Angular does the rest!

But this does not happen by magic, of course, and in order to make effective use of Angular, you need to understand how it updates these values as they change. This is called **change detection**, and Angular has a very different approach to doing this than what previously existed.

If you use the debugger tool in your browser to walk through the application, you will see how change detection works. Here, we are using Chrome's Developer tools and setting a watch for the `noOfTries` property. If you place a breakpoint at the end of the `verifyGuess()` method, you will see that when you enter a guess, the `noOfTries` property is first updated as soon as you hit the breakpoint, as follows:



Once you move past the breakpoint, the display on the screen updates with the correct number of guesses, as seen in the following screenshot:

Guess the Number !

The screenshot shows the Chrome DevTools interface. On the left, the 'Sources' tab is active, displaying the code for `app.component.ts`. The code defines a class `AppComponent` with properties like `title`, `deviation`, `noOfTries`, `original`, `guess`, and methods `constructor()`, `initializeGame()`, `verifyGuess()`, and a constructor for the component itself. A breakpoint is set at line 49. On the right, the 'Elements' tab shows the DOM structure of the application. It includes a heading 'Guess the Number !', a text input field with the value '6', a blue 'Verify' button, and a yellow 'Restart' button. Below these are two message boxes: 'Your guess is lower.' and 'No of guesses : 1'. The 'Elements' tab also displays the current state of variables in the scope, including `this.noOfTries` and `this.guess`, both currently undefined.

What is really going on under the hood is that Angular is reacting to events in the application and using change detectors, which go through every component, to determine whether anything has changed that affects the view. In this case, the event is a button click. The event generated by the button click calls the `verifyGuess()` method on the component that updates the `noOfTries` property.

That event triggers the change detection cycle, which identifies that the `noOfTries` property that is being used in the view has changed. As a result, Angular updates the element in the view that is bound to `noOfTries` with the new value of that property.

As you can see, this is a multistep process where Angular first updates the components and domain objects in response to an event, then runs change detection, and finally rerenders elements in the view that have changed. And, it does this on every browser event (as well as other asynchronous events, such as XHR requests and timers). Change detection in Angular is reactive and one way.

This approach allows Angular to make just one pass through the change detection graph. It is called **one-way data binding**, and it vastly improves the

performance of Angular.



We'll be covering Angular change detection in depth in chapter 8, Some Practical Scenarios. For a description of this process by the Angular team, visit <https://vsavkin.com/two-phases-of-angular-2-applications-fda2517604be#.fabhc0ynb>.

Tools

Tools make our lives easy, and we are going to share some tools that will help you with different aspects of Angular development, from code writing to debugging:

- **Visual Studio Code:** This is a new IDE that Microsoft has developed (<https://code.visualstudio.com/>). It provides excellent IntelliSense and code completion support for Angular and TypeScript. Visual Studio 2017 (<https://www.visualstudio.com/>) also includes support for Angular and TypeScript.
- **IDE extensions:** Many of the popular IDEs on the market have plugins/extensions that make Angular development easy for us. Examples include:
 - **JetBrains WebStorm:** <https://www.jetbrains.com/webstorm/>
 - **Angular2 Snippets for Sublime Text:** <https://github.com/evanplaice/angular2-snippets> and the **TypeScript plugin for Sublime** <https://github.com/Microsoft/TypeScript-Sublime-Plugin>
 - **Angular 2 TypeScript snippets for Atom** <https://atom.io/packages/angular2-typescript-snippets> and the **TypeScript plugin for Atom:** <https://atom.io/packages/atom-typescript>
- The **Angular language service:** One of the exciting new Developer tools, it provides auto-completions, error checking, and F12 navigation inside Angular templates that are placed in component decorators or external HTML files. You can find installation instructions and more information about the service at <https://angular.io/guide/language-service>.
- **Browser developer console:** All current browsers have excellent capabilities when it comes to JavaScript debugging. Since we are working with JavaScript, we can put in breakpoints, add a watch, and do everything that is otherwise possible with JavaScript. Remember that a lot of errors with code can be detected just by looking at the browser's console window.
- **Augury** (<https://augury.angular.io/>): This is a Chrome Dev Tools extension for debugging Angular applications.
- **Component vendors** are starting to offer support for Angular as well. For example, Telerik has released Kendo UI for Angular: <http://www.telerik.com/kendo-angular-ui-components>

[ndo-angular-ui/](#).

Resources

Angular is a new framework, but already a vibrant community is starting to emerge around it. Together with this book, there are also blogs, articles, support forums, and plenty of help. Some of the prominent resources that will be useful are explained as follows:

- **Framework code and documentation:** The Angular documentation can be found at <https://angular.io/docs>. Then, there is always the Angular source code, a great source of learning. It can be found at <https://github.com/angular/angular>.
- **The Angular team's blog:** You can refer to the Angular team's blog for more information about Angular at <https://blog.angular.io/>.
- **Awesome Angular: A curated list of awesome Angular resources:** This is a community-driven effort that is maintained at <https://github.com/gdi2290/awesome-angular>.
- **The Angular gitter chat room** (<https://gitter.im/angular/angular>) is very active. Also check out **Angular on Reddit**: <https://www.reddit.com/r/Angular>.
- **The Angular Google group** (<https://groups.google.com/forum/#!forum/angular>) and the **Stack Overflow channel** (<http://stackoverflow.com/questions/tagged/angular>): Head over here if you have any questions or are stuck with some issue.
- **Angular Expo** (<http://angularexpo.com/>): People have created some amazing apps using Angular. This site showcases such apps, and most of them have source code available for us to take a look at.

That's it! The chapter is complete and it's now time to summarize what you've learned.

25-Sep-2018

16-Nov-2018

Summary

The journey has started and we have reached the first milestone. Despite this chapter being named *Getting Started*, we have covered a lot of concepts that you will need to know in order to understand the bigger picture. Your learning was derived from our *Guess the Number!* app, which we built and dissected throughout the chapter.

You learned how Angular implements the component design pattern using the emerging standards for Web Components, along with the latest versions of JavaScript and TypeScript. We also reviewed some of the constructs that are used in Angular, such as interpolation, expressions, and the data binding syntax. Finally, we took a look at change detection and some useful tools and resources that will help you get started with Angular development.

The groundwork has been laid, and now we are ready for some serious app development on the Angular framework. In the next chapter, we will start working on a more complex exercise and expose ourselves to a number of new Angular constructs.

Building Our First App – 7 Minute Workout

I hope that the first chapter was intriguing enough and that you want to learn more about Angular—believe me, we have just scratched the surface! The framework has a lot to offer, and together with TypeScript, it strives to make frontend development more organized and hence manageable.

Keeping up with the theme of this book, we will be building a new app in Angular, and in the process, become more familiar with the framework. This app will also help us explore some new capabilities of Angular.

The topics that we will cover in this chapter include the following:

- **7 Minute Workout problem description:** We detail the functionality of the app that we build in this chapter.
- **Code organization:** For our first real app, we will try to explain how to organize code, specifically Angular code.
- **Designing the model:** One of the building blocks of our app is its model. We design the app model based on the app's requirements.
- **Understanding the data binding infrastructure:** While building the *7 Minute Workout* view, we will look at the data binding capabilities of the framework, which include *property*, *attribute*, *class*, *style*, and *event* bindings.
- **Exploring the Angular platform directives:** Some of the directives that we will cover are `ngFor`, `ngIf`, `ngClass`, `ngStyle`, and `ngSwitch`.
- **Cross-component communication with input properties:** As we build nested components, we learn how input properties can be used to pass data from the parent to its child components.
- **Cross-component communication with events:** Angular components can subscribe to and raise events. We get introduced to event binding support in Angular.

- **Angular pipes:** Angular pipes provide a mechanism to format view content. We explore some standard Angular pipes and build our own pipe to support conversions from seconds to hh:mm:ss.

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

What is 7 Minute Workout?

We want everyone reading this book to be physically fit. Therefore, this book should serve a dual purpose; it should not only stimulate your grey matter, but also urge you to look after 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 app that requires us to perform a set of 12 exercises in quick succession within the seven-minute time span. *7 Minute Workout* has become quite popular due to its bite-sized length and great benefits. We 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 to know more about the workout, then check out <http://well.blogs.nytimes.com/2013/05/09/the-scientific-7-minute-workout/>.

The technicalities of the app include performing a set of 12 exercises, dedicating 30 seconds for each of the exercises. This is followed by a brief rest period before starting the next exercise. For the app that we are building, we will be taking rest periods of 10 seconds each. So, the total duration comes out at a little more than seven minutes.

At the end of the chapter, we will have the *7 Minute Workout* app ready, which will look something like the following:

7 Minute Workout

Description

A jumping jack or star jump, also called side-straddle hop is a physical jumping exercise.

Steps

Assume an erect position with feet together and arms at your side. Slightly bend your knees and propel yourself a little, increase into the air. While in air bring your legs out to the side about shoulder width or slightly wider. As you are moving your legs outward, you should raise your arms up over your head, arms should be slightly bent but not too much, hands should be open. Your feet should land shoulder width or wider as your hands meet above your head with arms slightly bent.

Workout Remaining - 00:07:40

Jumping Jacks



Time Remaining: 20

video-player 442.5px x 233px

Basic Exercise Plans : How ... 🔍

In-Home Cardio Workouts : 🔍 ↗

The 7 Minute Workout app

Downloading the code base

The code for this app can be downloaded from the GitHub site (<https://github.com/chandermani/angular6byexample>) dedicated to this book. Since we are building the app incrementally, we have created **multiple checkpoints** that map to **GitHub branches** such as `checkpoint2.1`, `checkpoint2.2`, and so on. During the narration, we will highlight the branch for reference. These branches will contain the work done on the app up until that point in time.



The 7 Minute Workout code is available in the repository folder named `trainer`.

So, let's get started!

Setting up the build

Remember that we are building on a modern platform for which browsers still lack support. Therefore, directly referencing script files in HTML is out of the question (while common, it's a dated approach that we should avoid anyway). Browsers do not understand **TypeScript**; this implies that there has to be a process that converts code written in TypeScript into standard **JavaScript (ES5)**. Hence, having a build set up for any Angular app becomes imperative. And thanks to the growing popularity of Angular, we are never short of options.

If you are a frontend developer working on the web stack, you cannot avoid **Node.js**. This is the most widely used platform for web/JavaScript development. So, no prizes for guessing that most of the Angular build solutions out there are supported by Node. Packages such as **Grunt**, **Gulp**, **JSPM**, and **webpack** are the most common building blocks for any build system.



Since we too are building on the Node.js platform, install Node.js before starting.

For this book and this sample app, we endorse **Angular CLI** (<http://bit.ly/ng6be-a>). A command line tool, it has a build system and a scaffolding tool that hugely simplifies Angular's development workflow. It is popular, easy to set up, easy to manage, and supports almost everything that a modern build system should have. More about it later.

As with any mature framework, Angular CLI is not the only option out there on the web. Some of the notable starter sites plus build setups created by the community are as follows:

Start site	Location
angular2-webpack-starter	http://bit.ly/ng2webpack
angular-seed	https://github.com/mgechev/angular-seed

Let's start with installing Angular CLI. On the command line, type the following:

```
| npm i -g @angular/cli
```

Once installed, Angular CLI adds a new command `ng` to our execution environment. To create a new Angular project from the command line, run the following command:

```
| ng new PROJECT-NAME
```

This generates a folder structure with a bunch of files, a boilerplate Angular application, and a preconfigured build system. To run the application from the command line, execute the following:

```
| ng serve --open
```

And you can see a basic Angular application in action!

For our *7 Minute Workout* app, instead of starting from scratch, we are going to start from a version that is based on the project structure generated by `ng new` with minor modification. Start with the following steps:



Curious about what the default project includes? Go ahead and run `ng new PROJECT-NAME`. Look at the generated content structure and the Angular CLI documentation to get an idea of what's part of a default setup.

1. Download the base version of this app from http://bit.ly/ngbe_base and unzip it to a location on your machine. If you are familiar with how Git works, you can just clone the repository and check out the `base` branch:

```
| git checkout base
```

This code serves as the starting point for our app.

2. Navigate to the `trainer` folder from the command line and execute the command `npm install` from the command line to install the **package dependencies** for our application.



Packages in the Node.js world are third-party libraries (such as Angular for our app) that are either used by the app or support the app's building process. npm is a command-line tool for pulling these packages from a remote repository.

3. Once npm pulls the app dependencies from the npm store, we are ready to build and run the application. From the command line, enter the following

command:

```
| ng serve --open
```

This compiles and runs the app. If the build process goes fine, the default browser window/tab will open with a rudimentary app page (<http://localhost:4200/>). We are all set to begin developing our app in Angular!

But before we do that, it would be interesting to know a bit more about Angular CLI and the customization that we have done on the default project template that Angular CLI generates.

Angular CLI

Angular CLI was created with the aim of standardizing and simplifying the development and deployment workflow for Angular apps. As the documentation suggests: "The Angular CLI makes it easy to create an application that already works, right out of the box. It already follows our best practices!"

It incorporates:

- A build system based on **webpack**
- A **scaffolding tool** to generate all standard Angular artifacts including modules, directives, components, and pipes
- Adherence to **Angular style guide** (<http://bit.ly/ng6be-styleguide>), making sure we use community-driven standards for projects of every shape and size



You may have never heard the term **style guide**, or may not understand its significance. A **style guide** in any technology is a set of guidelines that help us organize and write code that is easy to develop, maintain, and extend. To understand and appreciate Angular's own style guide, some familiarity with the framework itself is desirable, and we have started that journey.

- A targeted **linter**; Angular CLI integrates with **codelyzer** (<http://bit.ly/ng6be-codelyzer>), a **static code analysis tool** that validates our Angular code against a set of rules to make sure that the code we write adheres to standards laid down in the Angular style guide
- Preconfigured **unit** and **end-to-end (e2e)** test framework

And much more!

Imagine if we had to do all this manually! The steep learning curve would have quickly overwhelm us. Thankfully, we don't have to deal with it, Angular CLI does it for us.



The Angular CLI build setup is based on webpack, but it does not expose the underlying webpack configuration; this is intentional. The Angular team wanted to shield developers from the complexities and internal workings of webpack. The ultimate aim of Angular CLI is to eliminate any entry level barriers and make setting up and running Angular code simple. It doesn't mean Angular CLI is not configurable. There is a config file (`angular.json`) that we can use to alter the build setup. We will not cover that here. Check the configuration file for 7 Minute Workout and read the documentation here: <http://bit.ly/ng6be-angular-cli-config>.

The tweaks that we have done to the default generated project template are:

- Referenced Bootstrap CSS in the `style.css` file.
- Upgraded some npm library versions.
- Changed the prefix configuration for generated code to use `abe` (short for Angular By Example) from `app`. With this change, all our components and directive selectors will be prefixed by `abe` instead of `app`. Check `app.component.ts`; the selector is `abe-root` instead of `app-root`.

~~While on the topic of Angular CLI and builds, there is something that we should understand before proceeding.~~

What happens to the TypeScript code we write?

Code transpiling

Browsers, as we all know, only work with JavaScript, they don't understand TypeScript. We hence need a mechanism to convert our TypeScript code into plain JavaScript (**ES5** is our safest bet). The **TypeScript compiler** does this job. The compiler takes the TypeScript code and converts it into JavaScript. This process is commonly referred to as **transpiling**, and since the TypeScript compiler does it, it's called a **transpiler**.



JavaScript as a language has evolved over the years with every new version adding new features/capabilities to the language. The latest avatar, ES2015, succeeds ES5 and is a major update to the language. While released in June 2015, some of the older browsers still lack support for the ES2015 flavor, of JavaScript making its adoption a challenge.

When transpiling code from TypeScript to JavaScript, we can specify the flavor of JavaScript to use. As mentioned earlier, ES5 is our safest bet, but if we plan to work with only the latest and greatest browsers, go for ES2015. For 7 Minute Workout, our code to transpile to is ES5 format. We set this TypeScript compiler configuration in `tsconfig.json` (see the `target` property).

Interestingly, transpilation can happen at both build/compile time and at runtime:

- **Build-time transpilation:** Transpilation as part of the build process takes the script files (in our case, TypeScript `.ts` files) and compiles them into plain JavaScript. Angular CLI does build-time transpilation.
- **Runtime transpilation:** This happens in the browser at runtime. We directly reference the TypeScript files (`.ts` in our case), and the **TypeScript compiler**, which is loaded in the browser beforehand, compiles these script files on the fly. This is a workable setup only for small examples/code snippets, as there is an additional performance overhead involved in loading the transpiler and transpiling the code on the fly.

The process of transpiling is not limited to TypeScript. Every language targeted towards the web, such as **CoffeeScript**, **ES2015**, (yes JavaScript itself!) or any other language that is not inherently understood by a browser needs transpilation. There are transpilers for most languages, and the prominent ones (other than TypeScript) are **tracuer** and **babel**.

The Angular CLI build system takes care of setting up the TypeScript compiler and sets up file watchers that recompile the code every time we make changes to

our TypeScript file.

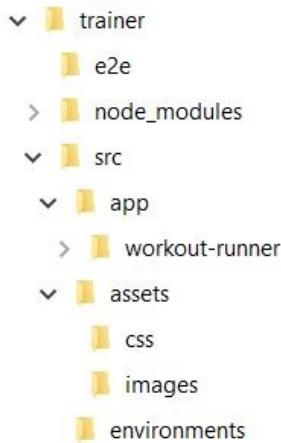


If you are new to TypeScript, remember that TypeScript does not depend on Angular; in fact, Angular has been built on TypeScript. I highly recommend that you look at the official documentation on TypeScript (<https://www.typescriptlang.org/>) and learn the language outside the realms of Angular.

Let's get back to the app we are building and start exploring the code setup.

Organizing code

The advantage of Angular CLI is that it dictates a code organization structure that works for applications of all sizes. Here is how the current code organization looks:



- `trainer` is the application root folder.
- The files inside `trainer` are configuration files and some standard files that are part of every standard `node` application.
- The `e2e` folder will contain end to end tests for the app.
- `src` is the primary folder where all the development happens. All the application artifacts go into `src`.
- The `assets` folder inside `src` hosts static content (such as images, CSS, audio files, and others).
- The `app` folder has the app's source code.
- The `environments` folder is useful to set configurations for different deployment environments (such as `dev`, `qa`, `production`).

To organize Angular code inside the `app` folder, we take a leaf from the Angular style guide (<http://bit.ly/ng6bc-style-guide>) released by the Angular team.

Feature folders

The style guide recommends use of **feature folders** to organize code. With feature folders, files linked to a single feature are placed together. If a feature grows, we break it down further into sub features and tuck the code into sub folders. Consider the `app` folder to be our first feature folder! As the application grows, `app` will add sub features for better code organization.

Let's get straight into building the application. Our first focus area, the app's model!

The 7 Minute Workout model

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

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

Some other valuable features that we will add to this app are as follows:

- 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 themes for this app are **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:

- The name. This should be unique.
- The title. This is shown to the user.
- The 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.

With TypeScript, we can define the classes for our model.

The `Exercise` class looks as follows:

```
export class Exercise {
  constructor(
    public name: string,
    public title: string,
    public description: string,
    public image: string,
    public nameSound?: string,
    public procedure?: string,
    public videos?: Array<string>) { }
```

TypeScript tips

 Declaring constructor parameters with `public` or `private` is a shorthand for creating and initializing class members at one go. The `? suffix` after `nameSound`, `procedure`, and `videos` implies that these are optional parameters.

For the workout, we need to track the following properties:

- The name. This should be unique.
- The title. This is shown to the user.
- The exercises that are part of the workout.
- The duration for each exercise.
- The rest duration between two exercises.

The model class to track workout progress (`WorkoutPlan`) looks as follows:

```
export class WorkoutPlan {
  constructor(
    public name: string,
    public title: string,
    public restBetweenExercise: number,
    public exercises: Exercise[],
    public description?: string) { }

  totalWorkoutDuration(): number { ... }
```

The `totalWorkoutDuration` function returns the total duration of the workout in seconds.

`workoutPlan` has a reference to another class in the preceding definition, `ExercisePlan`. It tracks the exercise and the duration of the exercise in a workout, which is quite apparent once we look at the definition of `ExercisePlan`:

```
export class ExercisePlan {  
    constructor(  
        public exercise: Exercise,  
        public duration: number) { }  
}
```

Let me save you some typing and tell you where to get the model classes, but before that, we need to decide where to add them. We are ready for our first feature.

First feature module

The primary feature of *7 Minute Workout* is to execute a predefined set of exercises. Hence we are going to create a feature module now and later add the feature implementation to this module. We call this module `workout-runner`. Let's initialize the feature with Angular CLI's scaffolding capabilities.

From the command line, navigate to the `trainer/src/app` folder and run the following:

```
| ng generate module workout-runner --module app.module.ts
```

Follow the console logs to know what files are generated. The command essentially:

- Creates a new Angular `WorkoutRunnerModule` module inside a new `workout-runner` folder
- Imports the newly created module into the main application module app (app.module.ts)

We now have a new **feature module**.



TIP *Give every feature its own module.*



Make special note of the conventions Angular CLI follows when scaffolding Angular artifacts. From the preceding example, the module name provided with the command line was `workout-runner`. While the generated folder and filenames use the same name, the class name for the generated module is `WorkoutRunnerModule` (pascal case with the module suffix).

Open the newly generated module definition (`workout-runner.module.ts`) and look at the generated content. `WorkoutRunnerModule` imports `CommonModule`, a module with common Angular directives such as `ngIf` and `ngFor`, allowing us to use these common directives across any component/directive defined in `WorkoutRunnerModule`.



Modules are Angular's way of organizing code. We will touch upon Angular modules shortly.

Copy the `model.ts` file from <http://bit.ly/ng6be-2-1-model-ts> into the `workout-runner` folder. Shortly, we will see how these model classes are utilized.

Since we have started with a preconfigured Angular app, we just need to understand how the app starts.

App bootstrapping

~~Chapter 1, *Getting Started*, had a good introduction to the app bootstrapping process. The app bootstrapping process for *7 Minute Workout* remains the same; look at the `src` folder. There is a `main.ts` file that bootstraps the application by calling the following:~~

```
| platformBrowserDynamic().bootstrapModule(AppModule)  
|   .catch(err => console.log(err));
```

~~The heavy lifting is done by the Angular CLI, which compiles the application, includes the script and CSS reference into `index.html`, and runs the application. We don't need to configure anything. These configurations are part of the default Angular CLI configuration (`.angular-cli.json`).~~

~~We have created a new module and added some model classes to the `module` folder. Before we go any further and start implementing the feature, let's talk a bit about **Angular modules**.~~

Exploring Angular modules

As the *7 Minute Workout* app grows and we add new components/directives/pipes/other artifacts to it, a need arises to organize these items. Each of these items needs to be part of an Angular module.

A naïve approach would be to declare everything in our app's root module (`AppModule`), as we did with `WorkoutRunnerComponent`, but this defeats the whole purpose of Angular modules.

To understand why a single-module approach is never a good idea, let's explore Angular modules.

Comprehending Angular modules

In Angular, **modules** are a way to organize code into chunks that belong together and work as a cohesive unit. Modules are Angular's way of grouping and organizing code.

An Angular module primarily defines:

- The components/directives/pipes it owns
- The components/directives/pipes it makes public for other modules to consume
- Other modules that it depends on
- Services that the module wants to make available application-wide

Any decent-sized Angular app will have modules interlinked with each other: some modules consuming artifacts from other, some providing artifacts to others, and some modules doing both.

As a standard practice, module segregation is feature-based. One divides the app into features or subfeatures (for large features) and modules are created for each of the features. Even the framework adheres to this guideline as all of the framework constructs are divided across modules:

- There is `commonModule` that aggregates the standard framework constructs used in every browser-based Angular app
- There is `RouterModule` if we want to use the Angular routing framework
- There is `HttpModule` if our app needs to communicate with the server over HTTP

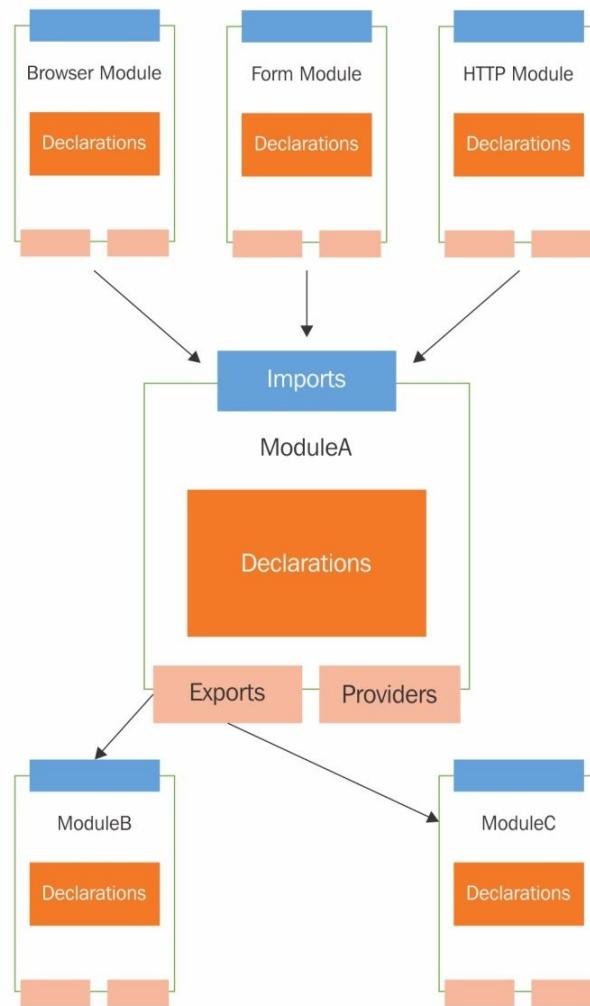
Angular modules are created by applying the `@NgModule` decorator to a TypeScript class, something we learned in [Chapter 1, Getting Started](#). The decorator definition exposes enough metadata, allowing Angular to load everything the module refers to.

The decorator has multiple attributes that allow us to define:

- External dependencies (using `imports`).

- Module artifacts (using `declarations`).
- Module exports (using `exports`).
- The services defined inside the module that need to be registered globally (using `providers`).
- The main application view, called the **root component**, which hosts all other app views. Only the root module should set this using the `bootstrap` property.

This diagram highlights the internals of a module and how they link to each other:



*Modules defined in the context of Angular (using the `@NgModule` decorator) are different from modules we import using the `import` statement in our TypeScript file. Modules imported through the `import` statement are **JavaScript modules**, which can be in different formats adhering to CommonJS, AMD, or ES2015 specifications, whereas Angular modules are constructs used by Angular to segregate and organize its artifacts. Unless the context of the discussion is specifically a JavaScript module, any reference to module implies an Angular*

module. We can learn more about this here: <http://bit.ly/ng2be6-module-vs-ngmodule>.

We hope one thing is clear from all this discussion: creating a single application-wide module is not the right use of Angular modules unless you are building something rudimentary.

It's time to get into the thick of the action; let's build our first component.

Our first component - WorkoutRunnerComponent

`workoutRunnerComponent`, is the central piece of our *7 Minute Workout* app and it will contain the logic to execute the workout.

What we are going to do in the `workoutRunnerComponent` implementation is as follows:

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 until all the exercises are over

We are ready to create (or scaffold) our component.

From the command line, navigate to the `src/app` folder and execute the following `ng` command:

```
| ng generate component workout-runner -is
```

The generator generates a bunch of files (three) in the `workout-runner` folder and updates the module declaration in `workoutRunnerModule` to include the newly created `WorkoutRunnerComponent`.



The `-is` flag is used to stop generation of a separate CSS file for the component. Since we are using global styles, we do not need component-specific styles.



*Remember to run this command from the `src/app` folder and **not** from the `src/app/workout-runner` folder. If we run the preceding command from `src/app/workout-runner`, Angular CLI will create a new subfolder with the `workout-runner` component definition.*

The preceding `ng generate` command for component generates these three files:

- `<component-name>.component.html`: This is the component's view HTML.
- `<component-name>.component.spec.ts`: Test specification file used in unit testing.
We will dedicate a complete chapter to unit testing Angular applications.
- `<component-name>.component.ts`: Main component file containing component

implementation.

Again, we will encourage you to have a look at the generated code to understand what gets generated. The Angular CLI component generator saves us some keystrokes and once generated, the boilerplate code can evolve as desired.



We touched upon the component decorator (`@component`) in chapter 1, Getting Started, and the decorator applied here is no different. While we see only four decorator metadata properties (such as `templateUrl`), the component decorator supports some other useful properties too. Look at the Angular documentation for `component` to learn more about these properties and their application. In the coming chapters, we will utilize some metadata attributes other than the standard ones used on every component.

An observant reader might have noticed that the generated `selector` property value has a prefix `abc`; this is intentional. Since we are extending the **HTML domain-specific language (DSL)** to incorporate a new element, the prefix `abc` helps us demarcate HTML extensions that we have developed. So instead of using `<workout-runner></workout-runner>` in HTML we use `<abc-workout-runner></abc-workout-runner>`. The prefix value has been configured in `angular.json`, see the `prefix` property.



Always add a prefix to your component selector.

We now have the `workoutRunnerComponent` boilerplate; let's start adding the implementation, starting with adding the model reference.

In `workout-runner.component.ts`, import all the workout models:

```
| import {WorkoutPlan, ExercisePlan, Exercise} from '../model';
```

Next, we need to set up the workout data. Let's do that by adding some code in the generated `ngOnInit` function and related class properties to the `WorkoutRunnerComponent` class:

```
workoutPlan: WorkoutPlan;
restExercise: ExercisePlan;
ngOnInit() {
  this.workoutPlan = this.buildWorkout();
  this.restExercise = new ExercisePlan(
    new Exercise('rest', 'Relax!', 'Relax a bit', 'rest.png'),
    this.workoutPlan.restBetweenExercise);
}
```

`ngOnInit` is a special function that Angular calls when a component is initialized.

We will talk about `ngOnInit` shortly.

The `buildWorkout` on `workoutRunnerComponent` sets up the complete workout, as we will define shortly. We also initialize a `restExercise` variable to track even the rest periods as exercise (note that `restExercise` is an object of type `ExercisePlan`).

The `buildWorkout` function is a lengthy function, so it's better to copy the implementation from the workout runner's implementation available in Git branch `checkpoint2.1` (<http://bit.ly/ng6be-2-1-workout-runner-component-ts>). The `buildWorkout` code looks as follows:

```
buildWorkout(): WorkoutPlan {
  let workout = new WorkoutPlan('7MinWorkout',
    "7 Minute Workout", 10, []);
  workout.exercises.push(
    new ExercisePlan(
      new Exercise(
        'jumpingJacks',
        'Jumping Jacks',
        'A jumping jack or star jump, also called side-straddle hop
         is a physical jumping exercise.',
        'JumpingJacks.png',
        'jumpingjacks.wav',
        `Assume an erect position, with feet together and
         arms at your side. . . .`,
        ['dmYwZH_BNd0', 'BAB0dJ-2Z6o', 'c4DAnQ6DtF8']),
      30));
  // (TRUNCATED) Other 11 workout exercise data.
  return workout;
}
```

This code builds the `workoutPlan` object and pushes the exercise data into the `exercises` array (an array of `ExercisePlan` objects), returning the newly built `workout`.

The initialization is complete; now, it's time to actually implement the `start` workout. Add a `start` function to the `workoutRunnerComponent` implementation, as follows:

```
start() {
  this.workoutTimeRemaining =
  this.workoutPlan.totalWorkoutDuration();
  this.currentExerciseIndex = 0;
  this.startExercise(this.workoutPlan.exercises[this.currentExerciseIndex]);
}
```

Then declare the new variables used in the function at the top, with other variable declarations:

```
| workoutTimeRemaining: number;  
| currentExerciseIndex: number;
```

The `workoutTimeRemaining` variable tracks the total time remaining for the workout, and `currentExerciseIndex` tracks the currently executing exercise index. The call to `startExercise` actually starts an exercise. This is how the code for `startExercise` looks:

```
startExercise(exercisePlan: ExercisePlan) {  
    this.currentExercise = exercisePlan;  
    this.exerciseRunningDuration = 0;  
    const intervalId = setInterval(() => {  
        if (this.exerciseRunningDuration >= this.currentExercise.duration) {  
            clearInterval(intervalId);  
        }  
        else { this.exerciseRunningDuration++; }  
    }, 1000);  
}
```

We start by initializing `currentExercise` and `exerciseRunningDuration`. The `currentExercise` variable tracks the exercise in progress and `exerciseRunningDuration` tracks its duration. These two variables also need to be declared at the top:

```
| currentExercise: ExercisePlan;  
| exerciseRunningDuration: number;
```

We use the `setInterval` JavaScript function with a delay of one second (1,000 milliseconds) to make progress. Inside the `setInterval` callback, `exerciseRunningDuration` is incremented with each passing second. The nested `clearInterval` call stops the timer once the exercise duration lapses.



TypeScript arrow functions

The callback parameter passed to `setInterval (()=>{...})` is a lambda function (or an arrow function in ES 2015). Lambda functions are short-form representations of anonymous functions, with added benefits. You can learn more about them at <http://bit.ly/ng2be-ts-arrow-functio>

ns.

The first cut of the component is almost complete, except it currently has a static view (UI) and hence we cannot verify the implementation. We can quickly rectify this situation by adding a rudimentary view definition. Open `workout-runner.component.ts`, comment out the `templateUrl` property, and add an inline template property (`template`) and set it to the following:

```
| template: `<pre>Current Exercise: {{currentExercise | json}}</pre>  
| <pre>Time Left: {{currentExercise.duration - exerciseRunningDuration}}</pre>`  
| Strings enclosed in backticks (` `) are a new addition to ES2015. Also called template literals,  
| such string literals can be multiline and allow expressions to be embedded inside (not to be
```



 confused with Angular expressions). Look at the MDN article at <http://bit.ly/template-literals> for more details.

Inline versus external view template

The preceding `template` property is an example of **inline component template**. This allows the component developer to specify the component template inline instead of using a separate HTML file. The inline template approach generally works for components with a trivial view.

 **Inline templates have a disadvantage:** formatting HTML becomes difficult and IDE support is very limited as the content is treated as a string literal. When we externalize HTML, we can develop a template as a normal HTML document. We recommend you use an **external template file** (specified using `templateUrl`) for elaborate views. Angular CLI by default generates an external template reference, but we can affect this behavior by passing the `inline-template` flag to the `ng component` generation command, such as `inline-template=true`.

The preceding template HTML will render the raw `ExercisePlan` object and the exercise time remaining. It has an interesting expression inside the first interpolation: `currentExercise | json`. The `currentExercise` property is defined in `WorkoutRunnerComponent`, but what about the `|` symbol and what follows it (`json`)? In the Angular world, it is called a **pipe**. The sole purpose of a pipe is to transform/format template data.

The `json` pipe here does JSON data formatting. You will learn more about pipes later in this chapter, but to get a general sense of what the `json` pipe does, we can remove the `json` pipe plus the `|` symbol and render the template; we are going to do this next.

To render the new `WorkoutRunnerComponent` implementation, it has to be added to the root component's view. Modify `src/components/app/app.component.html` and replace the `h3` tag with the following code:

```
<div class="container body-content app-container">
    <abe-workout-runner></abe-workout-runner>
</div>
```

While the implementation may look complete, there is a crucial piece missing. Nowhere in the code do we actually start the workout. The workout should start as soon as we load the page.

Component lifecycle hooks are going to rescue us!

Component lifecycle hooks

The life of an Angular component is eventful. Components get created, change state during their lifetime, and finally, they are destroyed. Angular provides some **lifecycle hooks/functions** that the framework invokes (on the component) when such an event occurs. Consider these examples:

- When a component is initialized, Angular invokes `ngOnInit`
- When a component's data-bound properties change, Angular invokes `ngOnChanges`
- When a component is destroyed, Angular invokes `ngOnDestroy`

As developers, we can tap into these key moments and perform some custom logic inside the respective component.

The hook we are going to utilize here is `ngOnInit`. The `ngOnInit` function gets fired the first time the component's data-bound properties are initialized, but before the view initialization starts.



While `ngOnInit` and the class constructor seem to look similar, they have a different purpose. A constructor is a language feature and it is used to initialize class members. `ngOnInit`, on the other hand, is used to do some initialization stuff once the component is ready. Avoid use of a constructor for anything other than member initialization.

Update the `ngOnInit` function to the `WorkoutRunnerComponent` class with a call to start the workout:

```
| ngOnInit() {  
|   ...  
|   this.start();  
| }
```

Angular CLI as part of component scaffolding already generates the signature for `ngOnInit`. The `ngOnInit` function is declared on the `OnInit` interface, which is part of the core Angular framework. We can confirm this by looking at the import section of `WorkoutRunnerComponent`:

```
import {Component, OnInit} from '@angular/core';  
...  
export class WorkoutRunnerComponent implements OnInit {  
  There are a number of other lifecycle hooks, including ngOnDestroy, ngOnChanges, and
```



ngAfterViewInit, that components support, but we are not going to dwell on any of them here.

Look at the developer guide (<https://angular.io/guide/lifecycle-hooks>) on lifecycle hooks to learn more about other such hooks.



Implementing the interface (*OnInit* in the preceding example) is optional. These lifecycle hooks work as long as the function name matches. We still recommend you use interfaces to clearly communicate the intent.

Time to run our app! Open the command line, navigate to the `trainer` folder, and type this line:

```
| ng serve --open
```

The code compiles, but no UI is rendered. What is failing us? Let's look at the browser console for errors.

Open the browser's dev tools (common keyboard shortcut `F12`) and look at the console tab for errors. There is a template parsing error. Angular is not able to locate the `abe-workout-runner` component. Let's do some sanity checks to verify our setup:

- `WorkoutRunnerComponent` implementation complete - check
- Component declared in `WorkoutRunnerModule` - check
- `WorkoutRunnerModule` imported into `AppModule` - check

Still, the `AppComponent` template cannot locate the `WorkoutRunnerComponent`. Is it because `WorkoutRunnerComponent` and `AppComponent` are in different modules? Indeed, that is the problem! While `WorkoutRunnerModule` has been imported into `AppModule`, `WorkoutRunnerModule` still does not export the new `WorkoutRunnerComponent` that will allow `AppComponent` to use it.



Remember, adding a component/directive/pipe to the declaration section of a module makes them available inside the module. It's only after we export the component/directive/pipe that it becomes available to be used across modules. but, not providers..

Let's export `WorkoutRunnerComponent` by updating the export array of the `WorkoutRunnerModule` declaration to the following:

```
| declarations: [WorkoutRunnerComponent],  
| exports: [WorkoutRunnerComponent]
```

This time, we should see the following output:

```
Current Exercise: {  
  "exercise": {  
    "name": "jumpingJacks",  
    "title": "Jumping Jacks",  
    "description": "A jumping jack or star jump, also called side-  
    "image": "JumpingJacks.png",  
    "nameSound": "jumpingjacks.wav",  
    "procedure": "Assume an erect position, with feet together an  
    "videos": [  
      "dmYwZH_BNd0",  
      "BABOdJ-2Z6o",  
      "c4DAnQ6DtF8"  
    ]  
  },  
  "duration": 30  
}  
  
Time Left: 28
```



Always export artifacts defined inside an Angular module if you want them to be used across other modules.

The model data updates with every passing second! Now you'll understand why interpolations (`{{ }}`) are a great debugging tool.



This will also be a good time to try rendering `currentExercise` without the `json` pipe and see what gets rendered.

We are not done yet! Wait long enough on the page and we realize that the timer stops after 30 seconds. The app does not load the next exercise data. Time to fix it!

Update the code inside the `setInterval` function:

```
if (this.exerciseRunningDuration >= this.currentExercise.duration) {  
  clearInterval(intervalId);  
  const next: ExercisePlan = this.getNextExercise();  
  if (next) {  
    if (next !== this.restExercise) {  
      this.currentExerciseIndex++;  
    }  
    this.startExercise(next);}  
  else { console.log('Workout complete!'); }  
}
```

The `if` condition `if (this.exerciseRunningDuration >= this.currentExercise.duration)` is used to transition to the next exercise once the time duration of the current exercise lapses. We use `getNextExercise` to get the next exercise and call

`startExercise` again to repeat the process. If no exercise is returned by the `getNextExercise` call, the workout is considered complete.

During exercise transitioning, we increment `currentExerciseIndex` only if the next exercise is not a rest exercise. Remember that the original workout plan does not have a rest exercise. For the sake of consistency, we have created a rest exercise and are now swapping between rest and the standard exercises that are part of the workout plan. Therefore, `currentExerciseIndex` does not change when the next exercise is rest.

Let's quickly add the `getNextExercise` function too. Add the function to the `WorkoutRunnerComponent` class:

```
getNextExercise(): ExercisePlan {
  let nextExercise: ExercisePlan = null;
  if (this.currentExercise === this.restExercise) {
    nextExercise = this.workoutPlan.exercises[this.currentExerciseIndex + 1];
  }
  else if (this.currentExerciseIndex < this.workoutPlan.exercises.length - 1) {
    nextExercise = this.restExercise;
  }
  return nextExercise;
}
```

The `getNextExercise` function returns the next exercise that needs to be performed.



Note that the returned object for `getNextExercise` is an `ExercisePlan` object that internally contains the exercise details and the duration for which the exercise runs.

The implementation is quite self-explanatory. If the current exercise is rest, take the next exercise from the `workoutPlan.exercises` array (based on `currentExerciseIndex`); otherwise, the next exercise is rest, given that we are not on the last exercise (the `else if` condition check).

With this, we are ready to test our implementation. The exercises should flip after every 10 or 30 seconds. Great!



The current build setup automatically compiles any changes made to the script files when the files are saved; it also refreshes the browser after these changes. But just in case the UI does not update or things do not work as expected, refresh the browser window. If you are having a problem with running the code, look at the Git branch `checkpoint2.1` for a working version of what we have done thus far. Or if you are not using Git, download the snapshot of Checkpoint 2.1 (a ZIP file) from <http://bit.ly/ng6be-checkpoint2-1>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

We have done enough work on the component for now, let's build the view.

Building the 7 Minute Workout view

Most of the hard work has already been done while defining the model and implementing the component. Now, we just need to skin the HTML using the super-awesome data binding capabilities of Angular. It's going to be simple, sweet, and elegant!

For the *7 Minute Workout* view, we need to show the exercise name, the exercise image, a progress indicator, and the time remaining. Replace the local content of the `workout-runner.component.html` file with the content of the file from the Git branch `checkpoint2.2`, (or download it from <http://bit.ly/ng6be-2-2-workout-runner-component-hm> 1). The view HTML looks as follows:

```
<div class="row">
<div id="exercise-pane" class="col-sm">
<h1 class="text-center">{{currentExercise.exercise.title}}</h1>
<div class="image-container row">
<img class="img-fluid col-sm" [src]="/assets/images/" +
currentExercise.exercise.image" />
</div>
<div class="progress time-progress row">
<div class="progress-bar col-sm"
role="progressbar"
[attr.aria-valuenow]="exerciseRunningDuration"
aria-valuemin="0"
[attr.aria-valuemax]="currentExercise.duration"
[ngStyle]="{{'width':(exerciseRunningDuration/currentExercise.duration) *
100 + '%'}}">
</div>
</div>
<h1>Time Remaining: {{currentExercise.duration-exerciseRunningDuration}}</h1>
</div>
</div>
```

`WorkoutRunnerComponent` currently uses an inline template; instead, we need to revert back to using an external template. Update the `workout-runner.component.ts` file and

get rid of the `template` property, then uncomment `templateurl`, which we commented out earlier.

Before we understand the Angular pieces in the view, let's just run the app again. Save the changes in `workout_runner.component.html` and if everything went fine, we will see the workout app in its full glory:



The basic app is now up and running. The exercise image and title show up, the progress indicator shows the progress, and exercise transitioning occurs when the exercise time lapses. This surely feels great!



If you are having a problem with running the code, look at the Git branch `checkpoint2.2` for a working version of what we have done thus far. You can also download the snapshot of `checkpoint2.2` (a ZIP file) from this GitHub location: <http://bit.ly/ngobe-checkpoint-2-2>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Looking at the view HTML, other than some Bootstrap styles, there are some interesting Angular pieces that need our attention. Before we dwell on these view constructs in detail, let's break down these elements and provide a quick summary:

- `<h1 ...>{{currentExercise.exercise.title}}</h1>`: Uses **interpolation**
- ``: Uses **property binding** to bind the `src` property of the image to the component model property `currentExercise.exercise.image`
- `<div ... [attr.aria-valuenow]="exerciseRunningDuration" ... >`: Uses **attribute binding** to bind the `aria` attribute on `div` to `exerciseRunningDuration`
- `<div ... [ngStyle]="{{'width':(exerciseRunningDuration/currentExercise.duration) * 100 + '%'}}>`: Uses a **directive** `ngStyle` to bind the `style` property on the progress bar `div` to an expression that evaluates the exercise progress

Phew! There is a lot of binding involved. Let's dig deeper into the binding infrastructure.

The Angular binding infrastructure

Most modern JavaScript frameworks today come with strong model-view binding support, and Angular is no different. The primary aim of any binding infrastructure is to reduce the boilerplate code that a developer needs to write to keep the model and view in sync. A robust binding infrastructure is always declarative and terse.

The Angular binding infrastructure allows us to transform template (raw) HTML into a live view that is bound to model data. Based on the binding constructs used, data can flow and be synced in both directions: from model to view and view to model.

The link between the component's model and its view is established using the `template` OR `templateUrl` property of the `@Component` decorator. With the exception of the `script` tag, almost any piece of HTML can act as a template for the Angular binding infrastructure.

To make this binding magic work, Angular needs to take the view template, compile it, link it to the model data, and keep it in sync with model updates without the need for any custom boilerplate synchronization code.

Based on the data flow direction, these bindings can be of three types:

- **One-way binding from model to view:** In model-to-view binding, changes to the model are kept in sync with the view. Interpolations, property, attribute, class, and style bindings fall in this category.
- **One-way binding from view to model:** In this category, view changes flow towards the model. Event bindings fall in this category.
- **Two-way/bidirectional binding:** Two-way binding, as the name suggests, keeps the view and model in sync. There is a special binding construct used for two-way binding, `ngModel`, and some standard HTML data entry elements such as `input` and `select` support two-way binding.

Let's understand how to utilize the binding capabilities of Angular to support view templatization. Angular provides these binding constructs:

- Interpolations
- Property binding
- Attribute binding
- Class binding
- Style binding
- Event binding



We have already touched upon a number of binding capabilities in chapter 1, Getting Started, so here we strive to reduce repetition and build upon the knowledge that we acquired in the last chapter.

This is a good time to learn about all these binding constructs. **Interpolation** is the first one.

Interpolations

Interpolations are quite simple. The expression (commonly known as a **template expression**) inside the interpolation symbols (`{{ }}`) is evaluated in the context of the model (or the component class members), and the outcome of the evaluation (string) is embedded in HTML. A handy framework construct to display a component's data/properties. We have seen these all along in [Chapter 1, Getting Started](#), and also in the view we just added. We render the exercise title and the exercise time remaining using interpolation:

```
<h1>{{currentExercise.exercise.title}}</h1>
... <h1>Time Remaining: {{currentExercise.duration?-
exerciseRunningDuration}}</h1>
```

Remember that interpolations synchronize model changes with the view. Interpolation is one way of binding from a model to a view.



View bindings in Angular are always evaluated in the context of the component's scope.

Interpolations, in fact, are a special case of property binding, which allows us to bind any HTML element/component properties to a model. We will shortly discuss how an interpolation can be written using property binding syntax. Consider interpolation as syntactical sugar over property binding.

Property binding

As discussed in [chapter 1 Getting Started](#), property bindings allow us to bind native HTML/component properties to the component's model and keep them in sync (from model->view). Let's look at property binding from a different context.

Look at this view excerpt from the 7 Minute Workout's component view (`workout-runner.component.html`): ``

It seems that we are setting the `src` attribute of `img` to an expression that gets evaluated at runtime. But are we really binding to an attribute? Or is this a property? Are properties and attributes different?

In Angular realms, while the preceding syntax looks like it is setting an HTML element's attribute, it is in fact doing **property binding**. Moreover, since many of us are not aware of the difference between an HTML element's properties and its attributes, this statement is very confusing. Therefore, before we look at how property bindings work, let's try to grasp the difference between an element's property and its attribute.

Property versus attribute

Take any DOM element API and you will find attributes, properties, functions, and events. While events and functions are self-explanatory, it is difficult to understand the difference between properties and attributes. In daily use, we use these words interchangeably, which does not help much either. Take, for example, this line of code:

```
|<input type="text" value="Awesome Angular">
```

When the browser creates a DOM element (`HTMLInputElement` to be precise) for this input textbox, it uses the value attribute on `input` to set the initial state of the `value` property of `input` to `Awesome Angular`.

After this initialization, any changes to the `value` property of `input` do not reflect on the `value` attribute; the attribute always has `Awesome Angular` (unless set explicitly again). This can be confirmed by querying the `input` state.

Suppose we change the `input` data to `Angular rocks!` and query the `input` element state:

```
|input.value // value property
```

The `value` property always returns the current input content, which is `Angular rocks!`. Whereas this DOM API function:

```
|input.getAttribute('value') // value attribute
```

Returns the `value` attribute, and is always the `Awesome Angular` that was set initially.

The primary role of an element attribute is to initialize the state of the element when the corresponding DOM object is created.

There are a number of other nuances that add to this confusion. These include the following:

- Attribute and property synchronization is not consistent across properties. As we saw in the preceding example, changes to the `value` property on `input`

do not affect the `value` attribute, but this is not true for all property-value pairs. The `src` property of an image element is a prime example of this; changes to property or attribute values are always kept in sync.

- It's surprising to learn that the mapping between attributes and properties is also not one-to-one. There are a number of properties that do not have any backing attribute (such as `innerHTML`), and there are also attributes that do not have a corresponding property defined on the DOM (such as `colspan`).
- Attribute and property mapping adds to this confusion too, as they do not follow a consistent pattern. An excellent example of this is available in the Angular developer's guide, which we are going to reproduce here verbatim:



The `disabled` attribute is another peculiar example. A button's `disabled` property is `false` by default so the button is enabled. When we add the `disabled` attribute, its presence alone initializes the button's `disabled` property to `true` so the button is disabled. Adding and removing the `disabled` attribute disables and enables the button. The value of the attribute is irrelevant, which is why we cannot enable a button by writing `<button disabled="false">Still Disabled</button>`.

The aim of this discussion is to make sure that we understand the difference between the properties and attributes of a DOM element. This new mental model will help us as we continue to explore the framework's property and attribute binding capabilities. Let's get back to our discussion on property binding.

Property binding continued...

Now that we understand the difference between a property and an attribute, let's look at the binding example again:

```
|<img class="img-responsive" [src]="'/static/images/' + currentExercise.exercise.image"  
|/>
```

The `[propertyName]` square bracket syntax is used to bind the `img.src` property to an Angular expression.

The general syntax for property binding looks as follows:

```
| [target]="sourceExpression";
```

In the case of property binding, the `target` is a property on the DOM element or component. With property binding, we can literally bind to any property on the element's DOM. The `src` property on the `img` element is what we use; this binding works for any HTML element and every property on it.



Expression target can also be an event, as we will see shortly when we explore event binding.

Binding source and target

It is important to understand the difference between source and target in an Angular binding.



*The property appearing inside `[]` is a target, sometimes called **binding target**. The target is the consumer of the data and always refers to a property on the component/element. The source expression constitutes the data source that provides data to the target.*

At runtime, the expression is evaluated in the context of the component's/element's property (the

`WorkoutRunnerComponent.currentExercise.exercise.image` property in the preceding case).



Always remember to add square brackets `[]` around the target. If we don't, Angular treats the expression as a string constant and the target is simply assigned the string value.

Property binding, event binding, and attribute binding do not use the interpolation symbol.

The following is invalid: `[src]="{{'/static/images/' + currentExercise.exercise.image}}"`.

If you have worked on AngularJS, property binding together with event binding allows Angular to get rid of a number of directives, such as `ng-disable`, `ng-src`, `ng-key`, `ng-mouse*`, and a few others.*

From a data binding perspective, **Angular treats components in the same way as**

it treats native elements. Hence, property binding works on component properties too! Components can define input and output properties that can be bound to the view, such as this:

```
| <workout-runner [exerciseRestDuration]="restDuration"></workout-runner>
```

This hypothetical snippet binds the exerciseRestDuration property on the WorkoutRunnerComponent class to the restDuration property defined on the container component (parent), allowing us to pass the rest duration as a parameter to the WorkoutRunnerComponent. As we enhance our app and develop new components, you will learn how to define custom properties and events on a component.



We can enable property binding using the `bind-` syntax, which is a canonical form of property binding. This implies that `[src]="/assets/images/' + currentExercise.exercise.image"` is equivalent to the following: `bind-src="/static/images/' + currentExercise.exercise.image"`.



Property binding, like interpolation, is unidirectional, from the component/element source to the view. Changes to the model data are kept in sync with the view.

The template view that we just created has only one property binding (on `[src]`). The other bindings with square brackets aren't property bindings. We will cover them shortly.

Interpolation syntactic sugar over property binding

We concluded the section on interpolations by describing interpolation as syntactical sugar over property binding. The intent was to highlight how both can be used interchangeably. The interpolation syntax is terser than property binding and hence is very useful. This is how Angular interprets an interpolation: `<h3>Main heading - {{heading}}</h3>`
`<h3 [text-content]="" Main heading - '+ heading"></h3>`

Angular **translates** the interpolation in the first statement into the `textContent` property binding (second statement).

Interpolation can be used in more places than you can imagine. The following example contrasts the same binding using interpolation and property binding:
``
` //`
interpolation on attribute

```
<span [text-content]="helpText"></span>
<span>{{helpText}}</span>
```

While property binding (and interpolations) makes it easy for us to bind any expression to the target property, we should be careful with the expression we use. Angular's change detection system will evaluate your expression binding multiple times during the life cycle of the application, as long as our component is alive. Therefore, while binding an expression to a property target, keep these two guidelines in mind.

Quick expression evaluation

A property binding expression should evaluate quickly. Slow expression evaluation can kill your app's performance. This happens when a function performing CPU intensive work is part of an expression. Consider this binding:

```
| <div>{{doLotsOfWork()}}</div>
```

Angular will evaluate the preceding `doLotsOfWork()` expression every time it performs a change detection run. These change detection runs happen more often than we imagine and are based on some internal heuristics, so it becomes imperative that the expressions we use evaluate quickly.

Side effect-free binding expressions

If a function is used in a binding expression, it should be side effect-free.
Consider yet another binding:

```
|<div [innerHTML]="getContent()"></div>
```

And the underlying function, `getContent`:

```
getContent() {  
  var content=buildContent();  
  this.timesContentRequested +=1;  
  return content;  
}
```

The `getContent` call changes the state of the component by updating the `timesContentRequested` property every time it is called. If this property is used in views such as: `<div>{{timesContentRequested}}</div>`

Angular throws errors such as:

```
|Expression '{{getContent()}}' in AppComponent@0:4' has changed after it was checked.  
|Previous value: '1'. Current value: '2'
```



The Angular framework works in two modes, dev and production. If we enable production mode in the application, the preceding error does not show up. Look at the framework documentation at <http://bit.ly/enableProdMode> for more details.

The bottom line is that your expression used inside property binding should be side effect-free.

Let's now look at something interesting, `[ngStyle]`, which looks like a property binding, **but it's not**. The target specified in `[]` is **not a** component/element property (`div` does not have an `ngStyle` property), **it's a directive**.

Two new concepts need to be introduced, **target selection** and **directives**.

Angular directives

As a framework, Angular tries to enhance the HTML **DSL** (short for **Domain-Specific Language**):

- Components are referenced in HTML using custom tags such as `<abe-workout-runner></abe-workout-runner>` (not part of standard HTML constructs). This highlights the **first** extension point.
- The use of `[]` and `()` for property and event binding defines the **second**.
- And then there are **directives**, the **third** extension point which are further classified into **attribute** and **structural directives**, and **components** (components are directive too!).

While components come with their own view, **attribute directives** are there to enhance the appearance and/or behavior of existing elements/components.

Structural directives do not have their own view too; they change the DOM layout of the elements on which they are applied. We will dedicate a complete section later in the chapter to understanding these structural directives.

The `ngStyle` directive used in the `workout-runner` view is, in fact, an attribute directive: `<div class="progress-bar" role="progressbar" [ngStyle] = "{'width': (exerciseRunningDuration/currentExercise.duration) * 100 + '%'}"></div>`

The `ngStyle` directive does not have its own view; instead, it allows us to set multiple styles (`width` in this case) on an HTML element using binding expressions. We will be covering a number of framework attribute directives later in this book.

Directive nomenclature



Directives is an umbrella term used for component directives (also known as components), attribute directives, and structural directives. Throughout the book, when we use the term directive, we will be referring to either an attribute directive or a structural directive depending on the context. Component directives are always referred to as components.

With a basic understanding of the directive types that Angular has, we can comprehend the process of target selection for binding.

Target selection for binding

The target specified in `[]` is not limited to a component/element property. While the property name is a common target, the Angular templating engine actually does heuristics to decide the target type. Angular first searches the registered known directives (attribute or structural) that have matching selectors before looking for a property that matches the target expression. Consider this view fragment: `<div [ngStyle]='expression'></div>`

The search for a target starts with a framework looking at all internal and custom directives with a matching selector (`ngStyle`). Since Angular already has an `ngStyle` directive, it becomes the target (the directive class name is `ngStyle`, whereas the selector is `ngStyle`). If Angular did not have a built-in `ngStyle` directive, the binding engine would have looked for a property called `ngStyle` on the underlying component.

If nothing matches the target expression, an unknown directive error is thrown.

That completes our discussion on target selection. The next section is about attribute binding.

Attribute binding

The only reason attribute binding exists in Angular is that there are HTML attributes that do not have a backing DOM property. The `colspan` and `aria` attributes are some good examples of attributes without backing properties. The progress bar div in our view uses attribute binding.



If attribute directives are still playing your head, I cannot blame you, it can become a bit confusing. Fundamentally, they are different. Attribute directives (such as `[ngStyle]`) change the appearance or behavior of DOM elements and as the name suggests are directives. There is no attribute or property named `ngStyle` on any HTML element. Attribute binding, on the other hand, is all about binding to HTML attributes that do not have backing for a DOM property.

The `7 Minute Workout` uses attribute binding at two places, `[attr.aria-valuenow]` and `[attr.aria-valuemax]`. We may ask a question: can we use standard interpolation syntax to set an attribute? No, that does not work! Let's try it: open `workout-runner.component.html` and replace the two aria attributes `attr.aria-valuenow` and `attr.aria-valuemax` enclosed in `[]` with this highlighted code:

```
<div class="progress-bar" role="progressbar"
  aria-valuenow = "{{exerciseRunningDuration}}"
  aria-valuemin="0"
  aria-valuemax= "{{currentExercise.duration}}" ...> </div>
```

Save the view and if the app is not running, run it. This error will pop up in the browser console:

```
| Can't bind to 'ariaValuenow' since it isn't a known native property in
| WorkoutRunnerComponent ...
```

Angular is trying to search for a property called `ariavaluenow` in the `div` that does not exist! Remember, interpolations are actually property bindings.

We hope that this gets the point across: to bind to an HTML attribute, use attribute binding.



Angular binds to properties by default and not to attributes.

To support attribute binding, Angular uses a prefix notation, `attr`, within `[]`. An attribute binding looks as follows:

```
| [attr.attribute-name]="expression"
```

Revert to the original aria setup to make attribute binding work:

```
| <div ... [attr.aria-valuenow]="exerciseRunningDuration"  
|   [attr.aria-valuemax]="currentExercise.duration" ...>
```



Remember that unless an explicit `attr.` prefix is attached, attribute binding does not work.

While we have not used style and class-based binding in our workout view, these are some binding capabilities that can come in handy. Hence, they are worth exploring.

Style and class binding

We use **class binding** to set and remove a specific class based on the component state, as follows:

```
| [class.class-name]="expression"
```

This adds `class-name` when `expression` is `true` and removes it when it is `false`. A simple example can look as follows: `<div [class.highlight]="isPreferred">Jim</div>` // Toggles the highlight class

Use style bindings to set inline styles based on the component state:

```
| [style.style-name]="expression";
```

While we have used the `ngStyle` directive for the workout view, we could have easily used style binding as well, as we are dealing with a single style. With style binding, the same `ngStyle` expression would become the following:
`[style.width.%] = "(exerciseRunningDuration/currentExercise.duration) * 100"`

`width` is a style, and since it takes units too, we extend our target expression to include the `%` symbol.



Remember that `style.` and `class.` are convenient bindings for setting a single class or style. For more flexibility, there are corresponding attribute directives: `ngClass` and `ngStyle`.

Earlier in the chapter, we formally introduced directives and their classifications. One of the directives types, attribute directives (again, don't confuse them with attribute binding, which we introduced in the preceding section) are the focus of our attention in the next section.

Attribute directives

Attribute directives are HTML extensions that change the look, feel, or behavior of a component/element. As described in the section on Angular directives, these directives do not define their own view.

Other than `ngStyle` and `ngClass` directives, there are a few more attribute directives that are part of the core framework. `ngValue`, `ngModel`, `ngSelectOptions`, `ngControl`, and `ngFormControl` are some of the attribute directives that Angular provides.

Since *7 Minute Workout* uses the `ngStyle` directive, it would be wise to dwell more on this directive and its close associate `ngClass`.



*While the next section is dedicated to learning how to use the `ngClass` and `ngStyle` attribute directives, it is not until chapter 6, *Angular Directives in Depth*, that we learn how to create our own attribute directives.*

Styling HTML with ngClass and ngStyle

Angular has two excellent directives that allow us to dynamically set styles on any element and toggle CSS classes. For the bootstrap progress bar, we use the `ngStyle` directive to dynamically set the element's style, `width`, as the exercise progresses:

```
|<div class="progress-bar" role="progressbar" ...  
|  [ngStyle]="{'width':(exerciseRunningDuration/currentExercise.duration) * 100 +  
|  '%' }"> </div>
```

`ngStyle` allows us to bind one or more styles to a component's properties at once. It takes an object as a parameter. Each property name on the object is the style name, and the value is the Angular expression bound to that property, such as the following example:

```
|<div [ngStyle]= "{  
|  'width':componentWidth,  
|  'height':componentHeight,  
|  'font-size': 'larger',  
|  'font-weight': ifRequired ? 'bold': 'normal' }"></div>
```

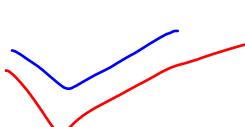
The styles can not only bind to component properties (`componentWidth` and `componentHeight`), but also be set to a constant value ('`larger`'). The expression parser also allows the use of the ternary operator (`?:`); check out `isRequired`.

If styles become too unwieldy in HTML, we also have the option of writing in our component a function that returns the object hash, and setting that as an expression:

```
|<div [ngStyle]= "getStyles()"></div>
```

Moreover, `getStyles` on the component looks as follows:

```
getStyles () {  
  return {  
    'width':componentWidth,  
    ...  
  }  
}
```



`ngClass` works on the same lines too, except that it is used to toggle one or multiple classes. For example, check out the following code:

```
|<div [ngClass]= "{'required':inputRequired, 'email':whenEmail}"></div>  
it is used, when we consume any CSS framework defined class like bootstrap
```

The required class is applied when inputRequired is true and is removed when it evaluates to false.



Directives (custom or platform) like any other Angular artifact, always belong to a module. To use them across modules, the module needs to be imported. Wondering where `ngStyle` is defined? `ngStyle` is part of the core framework module, `commonModule`,, and has been imported in the workout runner module definition (`workout-runner.module.ts`). `commonModule` defines a number of handy directives that are used across Angular.

Well! That covers everything we had to learn about our newly developed view.



And as described earlier, if you are having a problem with running the code, look at the Git branch `checkpoint2-2`. If not using Git, download the snapshot of `checkpoint2-2` (a ZIP file) from <http://bit.ly/ng2bc-checkpoint2-2>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Time to add some enhancements and learn a bit more about the framework!

Learning more about an exercise

For people who are doing this workout for the first time, it will be good to detail the steps involved in each exercise. We can also add references 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 in the left panel and call it the **description panel**. We will also add references to YouTube videos in the right panel, which is the video player panel. To make things more modular and learn some new concepts, we are going to create independent components for each description panel and video panel.

The model data for this is already available. The `description` and `procedure` properties in the `Exercise` class (see `model.ts`) provide the necessary details about the exercise. The `videos` array contains some related YouTube video IDs, which will be used to fetch these videos.

Adding descriptions and video panels

An Angular app is nothing but a hierarchy of components, similar to a tree structure. As of now, *7 Minute Workout* has two components, the root component, `AppComponent`, and its child, `workoutRunnerComponent`, in line with the HTML component layout, which now looks as follows:

```
<abe-root>...</abe-root>
<abe-workout-runner>...</abe-workout-runner>
```

Run the app and do a view source to verify this hierarchy. As we add more components to implement new features in the application this component tree grows and branches out.

We are going to add two subcomponents to `workoutRunnerComponent`, one each to support the exercise description and exercise videos. While we could have added some HTML directly to the `workoutRunnerComponent` view, what we are hoping here is to learn a bit more about cross-component communication. Let's start with adding the description panel on the left and understand how a component can accept inputs.

Component with inputs

Navigate to the `workour-runner` folder and generate a boilerplate exercise description component:

```
| ng generate component exercise-description -is
```

To the generated `exercise-description.component.ts` file, add the highlighted code:

```
import { Component, OnInit, Input } from '@angular/core';
...
export class ExerciseDescriptionComponent {
  @Input() description: string;
  @Input() steps: string;
}
```

The `@Input` decorator signifies that the component property is available for data binding. Before we dig into the `@Input` decorator, let's complete the view and integrate it with `workoutRunnerComponent`.

Copy the view definition for exercise description, `exercise-description.component.html`, from the Git branch `checkpoint2.3`, in the `workout-runner/exercise-description` folder (GitHub location: <http://bit.ly/ng6be-2-3-exercise-description-component.html>). Look at the highlighted HTML for the exercise description:

```
<div class="card-body">
  <div class="card-text">{{description}}</div>
</div>
...
<div class="card-text">
  {{steps}}
</div>
```

The preceding interpolation references the input properties of `ExerciseDescriptionComponent`: `description` and `steps`.

The component definition is complete. Now, we just need to reference `ExerciseDescriptionComponent` in `workoutRunnerComponent` and provide values for `description` and `steps` for the `ExerciseDescriptionComponent` view to render correctly.

Open `workout-runner.component.html` and update the HTML fragments as highlighted

in the following code. Add a new div called `description-panel` before the `exercise-pane` div and adjust some styles on the `exercise-pane` div, as follows:

```
<div class="row">
  <div id="description-panel" class="col-sm-3">
    <abe-exercise-description
      [description]="currentExercise.exercise.description"
      [steps]="currentExercise.exercise.procedure"></abe-exercise-description>
  </div>
  <div id="exercise-pane" class="col-sm-6">
    ...
  </div>
```

If the app is running, the description panel should show up on the left with the relevant exercise details.



WorkoutRunnerComponent was able to use ExerciseDescriptionComponent because it has been declared on WorkoutRunnerModule (see the `workout-runner.module.ts` declaration property). The Angular CLI component generator does this work for us.

Look back at the `abe-exercise-description` declaration in the preceding view. We are referring to the `description` and `steps` properties in the same manner as we did with the HTML element properties earlier in the chapter (``). Simple, intuitive, and very elegant!

The Angular data binding infrastructure makes sure that whenever the `currentExercise.exercise.description` and `currentExercise.exercise.procedure` properties on `WorkoutRunnerComponent` change, the bound properties on `ExerciseDescriptionComponent`, `description`, and `steps` are also updated.



The `@Input` decoration can take a property alias as a parameter, which means the following: consider a property declaration such as: `@Input("myAwesomeProperty") myProperty:string`. It can be referenced in the view as follows: `<my-component [myAwesomeProperty]="'expression'"...>`.

The power of the Angular binding infrastructure allows us to use any component property as a bindable property by attaching the `@Input` decorator (and `@Output` too) to it. We are not limited to basic data types such as `string`, `number`, and `boolean`; there can be complex objects too, which we will see next as we add the video player:



The `@Input` decorator can be applied to complex objects too.

Generate a new component in the `workout-runner` directory for the video player:

```
| ng generate component video-player -is
```

~~Update the generated boilerplate code by copying implementation from `video-player.component.ts` and `video-player.component.html` available in the Git branch `checkpoint2.3` in the `trainer/src/components/workout-runner/video-player` folder (GitHub location: <http://bit.ly/ng6be-2-3-video-player>).~~

Let's look at the implementation for the video player. Open `video-player.component.ts` and check out the `VideoPlayerComponent` class:

```
| export class VideoPlayerComponent implements OnInit, OnChanges {
|   private youtubeUrlPrefix = '//www.youtube.com/embed/';
|
|   @Input() videos: Array<string>;
|   safeVideoUrls: Array<SafeResourceUrl>;
|
|   constructor(private sanitizer: DomSanitizationService) { }
|
|   ngOnChanges() {
|     this.safeVideoUrls = this.videos ?
|       this.videos
|         .map(v =>
|           this.sanitizer.bypassSecurityTrustResourceUrl(this.youtubeUrlPrefix + v))
|         : this.videos;
|   }
| }
```

The `videos` input property here takes an array of strings (YouTube video codes). While we take the `videos` array as input, we do not use this array directly in video player view; instead, we transform the input array into a new array of `safeVideoUrls` and bind it. This can be confirmed by looking at the view implementation:

```
| <div *ngFor="let video of safeVideoUrls">
|   <iframe width="198" height="132" [src]="video" frameborder="0" allowfullscreen>
| </div>
```

The view also uses a new Angular directive called `ngFor` to bind to the `safeVideoUrls` array. The `ngFor` directive belongs to a class of directives called **structural directives**. The directive's job is to take an HTML fragment and regenerate it based on the number of elements in the bound collection.

If you are confused about how the `ngFor` directive works with `safeVideoUrls`, and why we need to generate `safeVideoUrls` instead of using the `videos` input array, wait for a while as we are shortly going to address these queries. But, let's first complete the integration of `VideoPlayerComponent` with `WorkoutRunnerComponent` to see the final outcome.

Update the `WorkoutRunnerComponent` view by adding the component declaration after the `exercise-pane` `div`:

```
|<div id="video-panel" class="col-sm-3">
|  <abe-video-player [videos]="currentExercise.exercise.videos"></abe-video-player>
|</div>
```

The `VideoPlayerComponent`'s `videos` property binds to the exercise's videos collection.

Start/refresh the app and the video thumbnails should show up on the right.



If you are having a problem with running the code, look at the Git branch `checkpoint2.3` for a working version of what we have done thus far. You can also download the snapshot of `checkpoint2.3` (a ZIP file) from <http://bit.ly/ng6be-checkpoint-2-3>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Now, it's time to go back and look at the parts of the `VideoPlayerComponent` implementation. We specifically need to understand:

- How the `ngFor` directive works
- Why there is a need to transform the input `videos` array into `safeVideoUrls`
- The significance of the Angular component life cycle event `onChanges` (used in the video player)

To start with, it's time to formally introduce `ngFor` and the class of directives it belongs to: structural directives.

Structural directives

The third categorization of directives, structural directives, work on the components/elements to manipulate their layout.

The Angular documentation describes structural directives in a succinct manner: "Instead of defining and controlling a view like a Component Directive, or modifying the appearance and behavior of an element like an Attribute Directive, the Structural Directive manipulates the layout by adding and removing entire element sub-trees."

Since we have already touched upon component directives (such as `workout-runner` and `exercise-description`) and attribute directives (such as `ngClass` and `ngStyle`), we can very well contrast their behaviors with structural directives.

The `ngFor` directive belongs to this class. We can easily identify such directives by the `*` prefix. Other than `ngFor`, Angular comes with some other structural directives such as `ngIf` and `ngSwitch`.

The ever-so-useful NgForOf

Every templating language has constructs that allow the templating engine to generate HTML (by repetition). Angular has `NgForOf`. The `NgForOf` directive is a super useful directive used to duplicate a piece of an HTML fragment n number of times. Let's again look at how we have used `NgForOf` in the video player:

```
<div *ngFor="let video of safeVideoUrls">
  <iframe width="198" height="132" [src]="video" frameborder="0" allowfullscreen>
</iframe>
</div>
```



The directive selector for `NgForOf` is `{selector: '[ngFor][ngForOf]'}, so we can use either ngFor or ngForOf in the view template. We also at times refer to this directive as ngFor.`

The preceding code repeats the `div` fragment for each exercise video (using the `safeVideoUrls` array). The `let video of safeVideoUrls` string expression is interpreted as follows: take each video in the `safeVideoUrls` array and assign it to a **template input variable**, `video`.

This input variable can now be referenced inside the `ngFor` template HTML, as we do when we set the `src` property binding.

Interestingly, the string assigned to the `ngFor` directive is not a typical Angular expression. Instead, it's a **microsyntax**—a micro language, which the Angular engine can parse.



You can learn more about microsyntax in Angular's developer guide: <http://bit.ly/ng6be-micro-syntax>.

This microsyntax exposes a number of iteration context properties that we can assign to template input variables and use them inside the `ngFor` HTML block.

One such example is `index`. `index` increases from 0 to the length of the array for each iteration, something similar to a `for` loop, in any programming language. The following example shows how to capture it:

```
<div *ngFor="let video of videos; let i=index">
  <div>This is video - {{i}}</div>
</div>
```

Other than `index`, there are some more iteration context variables; these include `first`, `last`, `even`, and `odd`. This context data allows us to do some nifty stuff. Consider this example:

```
| <div *ngFor="let video of videos; let i=index; let f=first">
|   <div [class.special]="f">This is video - {{i}}</div>
| </div>
```

It applies a `special` class to the first video `div`.

The `NgForOf` directive can be applied to HTML elements as well as our custom components. This is a valid use of `NgForOf`:

```
| <user-profile *ngFor="let userDetail of users" [user]= "userDetail"></user-profile>
```

Always remember to add an asterisk (*) before `ngFor` (and other structural directives). * has a significance.

Asterisk (*) in structural directives

The * prefix is a terser format to represent a structural directive. Take, for example, the usage of `ngFor` by the video player. The `ngFor` template: `<div *ngFor="let video of safeVideoUrls"> <iframe width="198" height="132" [src]="video" frameborder="0" allowfullscreen></iframe> </div>`

Actually expands to the following:

```
| <ng-template ngFor let-video [ngForOf]="safeVideoUrls">
|   <div>
|     <iframe width="198" height="132" [src]="video" ...></iframe>
|   </div>
| </ng-template>
```

The `ng-template` tag is an Angular element that has a declaration for `ngFor`, a template input variable (`video`), and a property (`ngForOf`) that points to the `safeVideoUrls` array. Both the preceding declarations are a valid usage of `ngFor`.

Not sure about you, but I prefer the terser first format for `ngFor`!

NgForOf performance

Since `NgForOf` generates HTML based on collection elements, it is notorious for causing performance issues. But we cannot blame the directive. It does what it is supposed to do: iterate and generate elements! If the underlying collection is huge, UI rendering can take a performance hit, especially if the collection changes too often. The cost of continuously destroying and creating elements in response to a changing collection can quickly become prohibitive.

One of the performance tweaks for `NgForOf` allows us to alter the behavior of `ngForOf` when it comes to creating and destroying DOM elements (when the underlying collection elements are added or removed).

Imagine a scenario where we frequently get an array of objects from the server and bind it to the view using `NgForOf`. The default behavior of `NgForOf` is to regenerate the DOM every time we refresh the list (since Angular does a standard object equality check). However, as developers, we may very well know not much has changed. Some new objects may have been added, some removed, and maybe some modified. But Angular just regenerates the complete DOM.

To alleviate this situation, Angular allows us to specify a custom **tracking function**, which lets Angular know when two objects being compared are equal. Have a look at the following function:

```
| trackByUserId(index: number, hero: User) { return user.id; }
```

A function such as this can be used in the `NgForOf` template to tell Angular to compare the *user* object based on its `id` property instead of doing a reference equality check.

This is how we then use the preceding function in the `NgForOf` template:

```
|<div *ngFor="let user of users; trackBy: trackByUserId">{{user.name}}</div>
```

`NgForOf` will now avoid recreating DOM for users with IDs already rendered.

Remember, Angular may still update the existing DOM elements if the bound properties of a user have changed.

That's enough on the `ngFor` directive; let's move ahead.

We still need to understand the role of the `safevideoUrls` and the `onchange` life cycle events in the `VideoPlayerComponent` implementation. Let's tackle the former first and understand the need for `safevideoUrls`.

Angular security

The easiest way to understand why we need to bind to `safeVideoUrls` instead of the `videos` input property is by trying the `videos` array out. Replace the existing `ngFor` fragment HTML with the following:

```
<div *ngFor="let video of videos">
  <iframe width="198" height="132"
    [src]="'//www.youtube.com/embed/' + video" frameborder="0" allowfullscreen>
</iframe>
</div>
```

And look at the browser's console log (a page refresh may be required). There are a bunch of errors thrown by the framework, such as:

```
Error: unsafe value used in a resource URL context (see http://g.co/ng/security#xss)
```

No prize for guessing what is happening! Angular is trying to safeguard our application against a **Cross-Site Scripting (XSS)** attack.

Such an attack enables the attacker to inject malicious code into our web pages. Once injected, the malicious code can read data from the current site context. This allows it to steal confidential data and also impersonate the logged-in user, hence gaining access to privileged resources.

Angular has been designed to block these attacks by sanitizing any external code/script that is injected into an Angular view. Remember, content can be injected into a view through a number of mechanisms, including property/attribute/style bindings or interpolation.

Consider an example of binding HTML markup through a component model to the `innerHTML` property of an HTML element (property binding):

```
this.htmlContent = '<span>HTML content.</span>'      // Component
<div [innerHTML]="htmlContent"> <!-- View -->
```

While the HTML content is emitted, any unsafe content (such as a `script`) if present is stripped.

But what about Iframes? In our preceding example, Angular is blocking property binding to Iframe's `src` property too. This is a warning against third-party content being embedded in our own site using Iframe. Angular prevents this too.

All in all, the framework defines four security contexts around content sanitization. These include:

1. **HTML content sanitization**, when HTML content is bound using the `innerHTML` property
2. **Style sanitization**, when binding CSS into the `style` property
3. **URL sanitization**, when URLs are used with tags such as `anchor` and `img`
4. **Resource sanitization**, when using `iframes` or `script` tags; in this case, content cannot be sanitized and hence it is blocked by default

Angular is trying its best to keep us out of danger. But at times, we know that the content is safe to render and hence want to circumvent the default sanitization behavior.

Trusting safe content

To let Angular know that the content being bound is safe, we use `DomSanitizer` and call the appropriate method based on the security contexts just described. The available functions are as follows:

- `bypassSecurityTrustHtml`
- `bypassSecurityTrustScript`
- `bypassSecurityTrustStyle`
- `bypassSecurityTrustUrl`
- `bypassSecurityTrustResourceUrl`

In our video player implementation, we use `bypassSecurityTrustResourceUrl`; it converts the video URL into a trusted `SafeResourceUrl` object:

```
| this.videos.map(v =>
|   this.sanitizer.bypassSecurityTrustResourceUrl(this.youtubeUrlPrefix + v))
```

The `map` method transforms the `videos` array into a collection of `SafeResourceUrl` objects and assigns it to `safeVideoUrls`.

Each of the methods listed previously takes a string parameter. This is the content we want Angular to know is safe. The return object, which could be any of `SafeStyle`, `SafeHtml`, `SafeScript`, `SafeUrl`, or `SafeResourceUrl`, can then be bound to the view.



A comprehensive treatment of this topic is available in the framework security guide available at <http://bit.ly/ng6be-security>. A highly recommended read!

The last question to answer is why do this in the onChanges Angular life cycle event?

OnChange life cycle event

The `onchanges` life cycle event is triggered whenever the component's input(s) change. In the case of `videoPlayerComponent`, it is the `videos` array input property that changes whenever a new exercise is loaded. We use this life cycle event to recreate the `safevideoUrls` array and re-bind it to the view. Simple!

Video panel implementation is now complete. Let's add a few more minor enhancements and explore it a bit more in Angular.

Formatting exercise steps with innerHTML binding

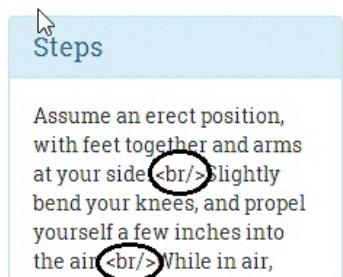
One of the sore points in the current app is the formatting of the exercise steps. It's a bit difficult to read these steps.

The steps should either have a line break (`
`) or be 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 bound to the step interpolation, or write a pipe that can add some HTML formatting using the line delimiting convention ().

For a quick verification, let's update the first exercise steps in `workout-runner.component.ts` by adding a break (`
`) after each line:

```
| `Assume an erect position, with feet together and arms at your side. <br>
|   Slightly bend your knees, and propel yourself a few inches into the air. <br>
|   While in air, bring your legs out to the side about shoulder width or slightly wider.
|   <br>
|   ...
```

As the workout restarts, look at the first exercise steps. The output does not match our expectations, as shown here:



The break tags were literally rendered in the browser. Angular did not render the interpolation as HTML; instead, it escaped the HTML characters, and we know why, security!

How to fix it? Easy! Replace the interpolation with the property binding to bind step data to the element's `innerHTML` property (in `exercise-description.html`), and you are done!

```
| <div class="card-text" [innerHTML]="steps">
```

Refresh the workout page to confirm.



Preventing Cross-Site Scripting Security (XSS) issues

By using `innerHTML`, we instruct Angular to not escape HTML, but Angular still sanitizes the input HTML as described in the security section earlier. It removes things such as `<script>` tags and other JavaScript to safeguard against XSS attacks. If you want to dynamically inject styles/scripts into HTML, use the `DomSanitizer` to bypass this sanitization check.

Time for another enhancement! It's time to learn about Angular pipes.

Displaying the remaining workout duration using pipes

It will be nice if we can tell the user the time left to complete the workout and not just the duration of the exercise in progress. We can add a countdown timer somewhere in the exercise pane to show the overall time remaining.

The approach that we are going to take here is to define a component property called `workoutTimeRemaining`. This property will be initialized with the total time at the start of the workout and will reduce with every passing second until it reaches zero. Since `workoutTimeRemaining` is a numeric value, but we want to display a timer in the `hh:mm:ss` format, we need to make a conversion between the seconds data and the time format. **Angular pipes** are a great option for implementing such a feature.

Angular pipes

The primary aim of a pipe is to format the data displayed in the view. **Pipes** allow us to package this content transformation logic (formatting) as a reusable element. The framework itself comes with multiple predefined pipes, such as date, currency, lowercase, uppercase, slice, and others.

This is how we use a pipe with a view:

```
| {{expression | pipeName:inputParam1}}
```

An expression is followed by the pipe symbol (|), which is followed by the pipe name and then an optional parameter (inputParam1) separated by a colon (:). If the pipe takes multiple inputs, they can be placed one after another separated by a colon, such as the inbuilt slice pipe, which can slice an array or string:

```
| {{fullName | slice:0:20}} //renders first 20 characters
```

The parameter passed to the pipe can be a constant or a component property, which implies we can use template expressions with pipe parameter. See the following example:

```
| {{fullName | slice:0:truncateAt}} //renders based on value truncateAt
```

Here are some examples of the use of the `date` pipe, as described in the Angular `date` documentation. Assume that `dateObj` is initialized to *June 15, 2015 21:43:11* and locale is *en-US*:

```
| {{ dateObj | date }}           // output is 'Jun 15, 2015      '
| {{ dateObj | date:'medium' }}   // output is 'Jun 15, 2015, 9:43:11 PM'
| {{ dateObj | date:'shortTime' }} // output is '9:43 PM          '
| {{ dateObj | date:'mmss' }}     // output is '43:11'
```

Some of the most commonly used pipes are the following:

- **date**: As we just saw, the date filter is used to format the date in a specific manner. This filter supports quite a number of formats and is locale-aware too. To know about the other formats supported by the date pipe, check out the framework documentation at <http://bit.ly/ng2-date>.
- **uppercase** and **lowercase**: These two pipes, as the name suggests, change

the case of the string input.

- **decimal** and **percent**: `decimal` and `percent` pipes are there to format decimal and percentage values based on the current browser locale.
- **currency**: This is used to format numeric values as a currency based on the current browser locale:

```
| {{14.22|currency:"USD" }} <!-Renders USD 14.22 -->
| {{14.22|currency:"USD":'symbol'}} <!-Renders $14.22 -->
```

- **json**: This is a handy pipe for debugging that can transform any input into a string using `JSON.stringify`. We made good use of it at the start of this chapter to render the `workoutPlan` object (see the Checkpoint 2.1 code).
- **slice**: This pipe allows us to split a list or a string value to create a smaller trimmed down list/string. We saw an example in the preceding code.

We are not going to cover the preceding pipes in detail. From a development perspective, as long as we know what pipes are there and what they are useful for, we can always refer to the platform documentation for exact usage instructions.

Pipe chaining

A really powerful feature of pipes is that they can be chained, where the output from one pipe can serve as the input to another pipe. Consider this example:

```
{fullName | slice:0:20 | uppercase}
```

The first pipe slices the first 20 characters of `fullName` and the second pipe transforms them to uppercase.

~~Now that we have seen what pipes are and how to use them, why not implement one for the *7 Minute Workout* app: a **seconds to time** pipe?~~

Implementing a custom pipe - SecondsToTimePipe

`SecondsToTimePipe`, as the name suggests, should convert a numeric value into the `hh:mm:ss` format.

Create a folder `shared` in the `workout-runner` folder and from the shared folder invoke this CLI command to generate the pipe boilerplate:

```
| ng generate pipe seconds-to-time
```

 *The shared folder has been created to add common components/directives/pipes that can be used in the `workout-runner` module. It is a convention we follow to organize shared code at different levels. In the future, we can create a shared folder at the app module level, which has artifacts shared globally. In fact, if the second to time pipe needs to be used across other application modules, it can also be moved into the app module.*

Copy the following `transform` function implementation into `seconds-to-time.pipe.ts`(the definition can also be downloaded from the Git branch `checkpoint.2.4` on the GitHub site at <http://bit.ly/nng6be-2-4-seconds-to-time-pipe-ts>):

```
export class SecondsToTimePipe implements PipeTransform {
  transform(value: number): any {
    if (!isNaN(value)) {
      const hours = Math.floor(value / 3600);
      const minutes = Math.floor((value - (hours * 3600)) / 60);
      const seconds = value - (hours * 3600) - (minutes * 60);

      return ('0' + hours).substr(-2) + ':'
        + ('0' + minutes).substr(-2) + ':'
        + ('0' + seconds).substr(-2);
    }
    return;
  }
}
```

In an Angular pipe, the implementation logic goes into the `transform` function. Defined as part of the `PipeTransform` interface, the preceding `transform` function transforms the input seconds value into an `hh:mm:ss` string. The first parameter to the `transform` function is the pipe input. The subsequent parameters, if provided, are the arguments to the pipe, passed using a colon separator (`pipe:argument1:arugment2..`) from the view.

For `secondsToTimePipe`, while Angular CLI generates a boilerplate argument `(args?:any)`, we do not make use of any pipe argument as the implementation does not require it.

The pipe 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 addition of 0 on the left for each of the `hours`, `minutes`, and `seconds` variables is done to format the value with a leading 0 in case the calculated value for hours, minutes, or seconds is less than 10.

The pipe that we just created is just a standard TypeScript class. It's the Pipe decorator (`@Pipe`) that instructs Angular to treat this class as a pipe:

```
| @Pipe({  
|   name: 'secondsToTime'  
| })
```

The pipe definition is complete, but to use the pipe in `WorkoutRunnerComponent` the pipe has to be declared on `WorkoutRunnerModule`. Angular CLI has already done this for us as part of the boilerplate generation (see the declaration section in `workout-runner.module.ts`).

Now we just need to add the pipe in the view. Update `workout-runner.component.html` by adding the highlighted fragment:

```
<div class="exercise-pane" class="col-sm-6">  
  <h4 class="text-center">Workout Remaining - {{workoutTimeRemaining |  
  secondsToTime}}</h4>  
  <h1 class="text-center">{{currentExercise.exercise.title}}</h1>
```

Surprisingly, the implementation is still not complete! There is one more step left. We have a pipe definition, and we have referenced it in the view, but `workoutTimeRemaining` needs to update with each passing second for `SecondsToTimePipe` to be effective.

We have already initialized `WorkoutRunnerComponent`'s `workoutTimeRemaining` property in the `start` function with the total workout time:

```
start() {  
  this.workoutTimeRemaining = this.workoutPlan.totalWorkoutDuration();  
  ...  
}
```

Now the question is: how to update the `workoutTimeRemaining` variable with each passing second? Remember that we already have a `setInterval` set up that updates `exerciseRunningDuration`. While we can write another `setInterval` implementation for `workoutTimeRemaining`, it will be better if a single `setInterval` setup can take care of both the requirements.

Add a function called `startExerciseTimeTracking` to `workoutRunnerComponent`; it looks as follows:

```
startExerciseTimeTracking() {
  this.exerciseTrackingInterval = window.setInterval(() => {
    if (this.exerciseRunningDuration >= this.currentExercise.duration) {
      clearInterval(this.exerciseTrackingInterval);
      const next: ExercisePlan = this.getNextExercise();
      if (next) {
        if (next !== this.restExercise) {
          this.currentExerciseIndex++;
        }
        this.startExercise(next);
      }
      else {
        console.log('Workout complete!');
      }
      return;
    }
    ++this.exerciseRunningDuration;
    --this.workoutTimeRemaining;
  }, 1000);
}
```

As you can see, the primary purpose of the function is to track the exercise progress and flip the exercise once it is complete. However, it also tracks `workoutTimeRemaining` (it decrements this counter). The first `if` condition setup just makes sure that we clear the timer once all the exercises are done. The inner `if` conditions are used to keep `currentExerciseIndex` in sync with the running exercise.

This function uses a numeric instance variable called `exerciseTrackingInterval`. Add it to the class declaration section. We are going to use this variable later to implement an exercise pausing behavior.

Remove the complete `setInterval` setup from `startExercise` and replace it with a call to `this.startExerciseTimeTracking();`. We are all set to test our implementation. If required, refresh the browser and verify the implementation:

Description

A jumping jack or star jump, also called side-straddle hop is a

Workout Remaining - 00:07:38

Jumping Jacks

Videos



The next section is about another inbuilt Angular directive, `ngIf`, and another small enhancement.

Adding the next exercise indicator using ngIf

It will be nice for the user to be told what the next exercise is during the short rest period between exercises. This will help them prepare for the next exercise. So let's add it.

To implement this feature, we can simply output the title of the next exercise from the `workoutPlan.exercises` array. We show the title next to the `Time Remaining` countdown section.

Change the workout div (`class="exercise-pane"`) to include the highlighted content, and remove existing `Time Remaining h1`:

```
<div class="exercise-pane">
  <!-- Exiting html -->
  <div class="progress time-progress">
    <!-- Exiting html -->
  </div>  <div class="row">
    <h4 class="col-sm-6 text-left">Time Remaining:
      <strong>{{currentExercise.duration-exerciseRunningDuration}}</strong>
    </h4>
    <h4 class="col-sm-6 text-right"
*ngIf="currentExercise.exercise.name=='rest'">Next up:
      <strong>{{workoutPlan.exercises[currentExerciseIndex + 1].exercise.title}}</strong>
    </h4>
  </div>
</div>
```

We wrap the existing `Time Remaining h1` and add another `h3` tag to show the next exercise inside a new `div` with some style updates. Also, there is a new directive, `ngIf`, in the second `h3`. The `*` prefix implies that it belongs to the same set of directives that `ngFor` belongs: **structural directives**. Let's talk a bit about `ngIf`.

The `ngIf` directive is used to add or remove a specific section of the DOM based on whether the expression provided to it returns `true` or `false`. The DOM element is added when the expression evaluates to `true` and is destroyed otherwise. Isolate the `ngIf` declaration from the preceding view:

```
| ngIf="currentExercise.details.name=='rest'"
```

The directive expression checks whether we are currently in the rest phase and accordingly shows or hides the linked `h3`.

Also in the same `h3`, we have an interpolation that shows the name of the exercise from the `workoutPlan.exercises` array.

A word of caution here: `ngIf` adds and destroys the DOM element, and hence it is not similar to the visibility constructs that we employed to show and hide elements. While the end result of `style, display:none` is the same as that of `ngIf`, the mechanism is entirely different:

```
| <div [style.display]="isAdmin" ? 'block' : 'none'">Welcome Admin</div>
```

Versus this line:

```
| <div *ngIf="isAdmin" ? 'block' : 'none'">Welcome Admin</div>
```

With `ngIf`, whenever the expression changes from `false` to `true`, a complete re-initialization of the content occurs. Recursively, new elements/components are created and data binding is set up, starting from the parent down to the children. The reverse happens when the expression changes from `true` to `false`: all of this is destroyed. Therefore, using `ngIf` can sometimes become an expensive operation if it wraps a large chunk of content and the expression attached to it changes very often. But otherwise, wrapping a view in `ngIf` is more performant than using CSS/style-based show or hide, as neither the DOM is created nor the data binding expressions are set up when the `ngIf` expression evaluates to `false`.

New version of Angular support branching constructs too. This allows us to implement the **if then else** flow in the view HTML. The following sample has been lifted directly from the platform documentation of `ngIf`:

```
| <div *ngIf="show; else elseBlock">Text to show</div>
| <ng-template #elseBlock>Alternate text while primary text is hidden</ng-template>
```

The `else` binding points to a `ng-template` with template variable `#elseBlock`.

There is another directive that belongs in this league: `ngSwitch`. When defined on the parent HTML, it can swap the child HTML elements based on the `ngSwitch` expression. Consider this example:

```
| <div id="parent" [ngSwitch] ="userType">
```

```
<div *ngSwitchCase="'admin'">I am the Admin!</div>
<div *ngSwitchCase="'powerUser'">I am the Power User!</div>
<div *ngSwitchDefault>I am a normal user!</div>
</div>
```

We bind the `userType` expression to `ngSwitch`. Based on the value of `userType` (`admin`, `powerUser`, or any other `userType`), one of the inner `div` elements will be rendered. The `ngSwitchDefault` directive is a wildcard match/fallback match, and it gets rendered when `userType` is neither `admin` nor `powerUser`.

If you have not realized it yet, note that there are three directives working together here to achieve switch-case-like behavior:

- `ngSwitch`
- `ngSwitchCase`
- `ngSwitchDefault`

Coming back to our next exercise implementation, we are ready to verify the implementation, start the app, and wait for the rest period. There should be a mention of the next exercise during the rest phase, as shown here:



The app is shaping up well. If you have used the app and done some physical workouts along with it, you will be missing the exercise pause functionality badly. The workout just does not stop until it reaches the end. We need to fix this behavior.

Pausing an exercise

To pause an exercise, we need to stop the timer. We also 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 on, 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 update the component.

Update the `WorkoutRunnerComponent` class, add these three functions, and add a declaration for the `workoutPaused` variable:

```
workoutPaused: boolean;
...
pause() {
  clearInterval(this.exerciseTrackingInterval);
  this.workoutPaused = true;
}

resume() {
  this.startExerciseTimeTracking();
  this.workoutPaused = false;
}

pauseResumeToggle() {
  if (this.workoutPaused) { this.resume(); }
  else { this.pause(); }
}
```

The implementation for pausing is simple. The first thing we do is cancel the existing `setInterval` setup by calling `clearInterval(this.exerciseTrackingInterval);`. While resuming, we again call `startExerciseTimeTracking`, which again starts tracking the time from where we left off.

Now we just need to invoke the `pauseResumeToggle` function for the view. Add the following content to `workout-runner.html`:

```
<div id="exercise-pane" class="col-sm-6">
  <div id="pause-overlay" (click)="pauseResumeToggle()"> <span class="pause absolute-center">
    [ngClass]="{{'ion-md-pause' : !workoutPaused, 'ion-md-play' :
    workoutPaused}}>
    </span>
  </div>
  <div class="row workout-content">
```

The `click` event handler on the div toggles the workout running state, and the `ngClass` directive is used to toggle the class between `ion-md-pause` and `ion-md-play`. What is missing now is the ability to pause and resume on a `P` key press.

One approach could be to apply a `keyup` event handler on the div:

```
| <div id="pause-overlay" (keyup)="onKeyPressed($event)">
```

But there are some shortcomings to this approach:

- The `div` element does not have a concept of focus, so we also need to add the `tabIndex` attribute on the div to make it work
- Even then, it works only when we have clicked on the div at least once

There is a better way to implement this; attach the event handler to the global `window` event `keyup`. This is how the event binding should be applied on the `div`:

```
| <div id="pause-overlay" (window:keyup)="onKeyPressed($event)">
```

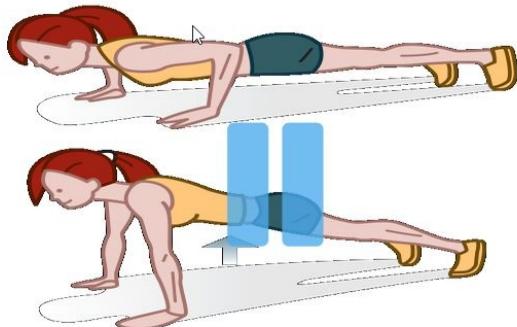
Make note of the special `window:` prefix before the `keyup` event. We can use this syntax to attach events to any global object, such as the `document`. A handy and very powerful feature of Angular binding infrastructure! The `onKeyPressed` event handler needs to be added to `workoutRunnerComponent`. Add this function to the class:

```
onKeyPressed(event: KeyboardEvent) {
  if (event.which === 80 || event.which === 112) {
    this.pauseResumeToggle();
  }
}
```

The `$event` object is the standard **DOM event object** that Angular makes available for manipulation. Since this is a keyboard event, the specialized class is `KeyboardEvent`. The `which` property is matched to ASCII values of `p` or `P`. Refresh the page and you should see the play/pause icon when your mouse hovers over the exercise image, as follows:

Workout Remaining - **00:06:02**

Push Up



While we are on the topic of **event binding**, it would be a good opportunity to explore Angular's event binding infrastructure

The Angular event binding infrastructure

Angular event binding allows a component to communicate with its parent through events.

If we look back at the app implementation, what we have encountered thus far are the property/attribute bindings. Such bindings allow a component/element to take inputs from the outside world. The data flows into the component.

Event bindings are the reverse of property bindings. They allow a component/element to inform the outside world about any state change.

As we saw in the pause/resume implementation, event binding employs round brackets () to specify the target event:

```
| <div id="pause-overlay" (click)="pauseResumeToggle()">
```

This attaches a click event handler to the div that invokes the expression pauseResumeToggle() when the div is clicked.



Like properties, there is a canonical form for events too. Instead of using round brackets, the on- prefix can be used: on-click="pauseResumeToggle()"

Angular supports all types of events. Events related to keyboard inputs, mouse movements, button clicks, and touches. The framework even allows us to define our own event for the components we create, such as:

```
| <workout-runner (paused)= "stopAudio()"></workout-runner>
```

We will be covering custom component events in the next chapter, where we will add audio support to 7 Minute Workout.

It is expected that events have side effects; in other words, an event handler may change the state of the component, which in turn may trigger a chain reaction in which multiple components react to the state change and change their own state. This is unlike a property binding expression, which should be side-effect-free.

Even in our implementation, clicking on the `div` element toggles the exercise run state.

Event bubbling

When Angular attaches event handlers to standard HTML element events, the event propagation works in the same way as standard DOM event propagation works. This is also called **event bubbling**. Events on child elements are propagated upwards, and hence event binding is also possible on a parent element, as follows: <div id="parent" (click)="doWork(\$event)"> Try <div id="child">me!</div> </div>

Clicking on either of the divs results in the invocation of the `doWork` function on the parent `div`. Moreover, `$event.target` contains the reference to the `div` that dispatched the event.



Custom events created on Angular components do not support event bubbling.

Event bubbling stops if the expression assigned to the `target` evaluates to a falsey value (such as `void`, `false`). Therefore, to continue propagation, the expression should evaluate to `true`: <div id="parent" (click)="doWork(\$event) || true">

Here too, the `$event` object deserves some special attention.

Event binding an \$event object

Angular makes an `$event` object available whenever the target event is triggered. This `$event` contains the details of the event that occurred.

The important thing to note here is that the shape of the `$event` object is decided based on the event type. For HTML elements, it is a DOM event object (<https://developer.mozilla.org/en-US/docs/Web/Events>), which may vary based on the actual event.

But if it is a custom component event, what is passed in the `$event` object is decided by the component implementation. We will return to this discussion again, in the next chapter.

We have now covered most of the data binding capabilities of Angular, with the exception of two-way binding. A quick introduction to the two-way binding constructs is warranted before we conclude the chapter.

Two-way binding with ngModel

Two-way binding helps us keep the model and view in sync. Changes to the model update the view and changes to the view update the model. The obvious area where two-way binding is applicable is form input. Let's look at a simple example: `<input [(ngModel)]="workout.name">`

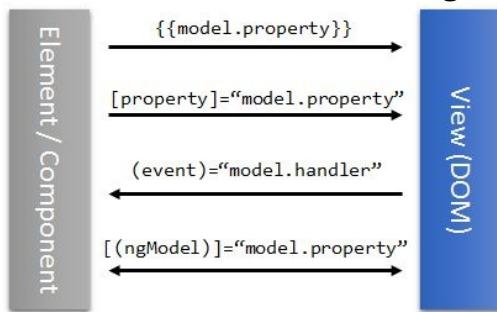
The `ngModel` directive here sets a two-way binding between the `input`'s `value` property and the `workout.name` property on the underlying component. Anything that the user enters in the preceding `input` is synced with `workout.name`, and any changes to `workout.name` are reflected back on the preceding `input`.

Interestingly, we can achieve the same result without using the `ngModel` directive too, by combining both property and event binding syntax. Consider the next example; it works in the same way as `input` before: `<input [value]="workout.name" (input)="workout.name=$event.target.value" >`

There is a property binding set up on the `value` property and an event binding set up on the `input` event that make the bidirectional sync work.

We will get into more details on two-way binding in [chapter 4, Personal Trainer](#), where we build our own custom workouts.

We have created a diagram that summarizes the data flow patterns for all the bindings that we have discussed thus far. Here is a handy diagram to help you memorize each of the binding constructs and how data flows:



We now have a fully functional *7 Minute Workout*, with some bells and whistles too, and hopefully you had fun creating the app. It's time to conclude the chapter

and summarize the lessons.



If you are having a problem with running the code, look at the Git branch `checkpoint2.4` for a working version of what we have done thus far. You can also download a snapshot of `checkpoint2.4` (a ZIP file) from this GitHub location: <http://bit.ly/ng6be-checkpoint-2-4>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Summary

We started this chapter with the aim of creating an Angular app that is more complex than the sample we created in the first chapter. The *7 Minute Workout* app fitted the bill, and you learned a lot about the Angular framework while building this app.

We started by defining the functional specifications of the *7 Minute Workout* app. We then focused our efforts on defining the code structure for the app.

To build the app, we started off by defining the model of the app. Once the model was in place, we started the actual implementation by building an **Angular component**. Angular components are nothing but classes that are decorated with a framework-specific decorator, `@Component`.

We also learned about **Angular modules** and how Angular uses them to organize code artifacts.

Once we had a fully functional component, we created a supporting view for the app. We also explored the data binding capabilities of the framework, including **property**, **attribute**, **class**, **style**, and **event binding**. Plus, we highlighted how **interpolations** are a special case of property binding.

Components are a special class of directives that have an attached view. We touched upon what directives are and the special classes of directives, including **attribute** and **structural directives**.

We learned how to perform cross-component communication using **input properties**. The two child components that we put together (`ExerciseDescriptionComponent` and `VideoPlayerComponent`) derived their inputs from the parent `WorkoutRunnerComponent` using input properties.

We then covered another core construct in Angular, **pipes**. We saw how to use pipes such as the date pipe and how to create one of our own.

Throughout the chapter, we touched upon a number of Angular directives,

including the following:

- `ngClass/ngStyle`: For applying multiple styles and classes using Angular binding capabilities
- `ngFor`: For generating dynamic HTML content using a looping construct
- `ngIf`: For conditionally creating/destroying DOM elements
- `ngSwitch`: For creating/destroying DOM elements using the switch-case construct

We now have a basic *7 Minute Workout* app. For a better user experience, we have added a number of small enhancements to it too, but we are still missing some good-to-have features that would make our app more usable. From the framework perspective, we have purposefully ignored some core/advanced concepts such as **change detection**, **dependency injection**, **component routing**, and data flow patterns, which we plan to cover in the next chapter.

5-Oct-2018

More Angular – SPA and Routing

The previous chapter was about building our first useful app in Angular, then this chapter is about adding a whole lot of Angular goodness to it. Within the learning curve, we have made a start in exploring a technology platform and now we can build some rudimentary apps using Angular. But that's just the start! There is a lot more to learn before we can make effective use of Angular in a decent-sized application. This chapter takes us one step closer to realizing this goal.

The *7-Minute Workout* app still has some rough edges that we can fix while making the overall app experience better. This chapter is all about adding those enhancements and features. And as always, this app-building process provides us with enough opportunities to enhance our understanding of the framework and learn new things about it.

The topics we cover in this chapter include:

- **Exploring Angular's Single Page Applications (SPAs):** We explore Angular's SPA capabilities, which include route navigation, link generation, and routing events.
- **Understanding dependency injection:** One of the core platform features. In this chapter, we learn how Angular makes effective use of dependency injection to inject components and services across the application.
- **Angular pure (stateless) and impure (stateful) pipes:** We explore the primary data transformation construct of Angular, pipes, in more detail as we build some new pipes.
- **Cross-component communication:** Since Angular is all about components and their interactions, we look at how to do cross-component communication in a parent-child and sibling component setup. We learn how Angular's *template variables* and *events* facilitate this communication.
- **Creating and consuming events:** We learn how a component can expose its own events and how to bind to these events from the HTML template and from other components.

As a side note, I expect you are using the *7-Minute Workout* on a regular basis and working on your physical fitness. If not, take a seven-minute exercise break and exercise now. I insist!

Hope the workout was fun! Now let's get back to some serious business. Let's start with exploring Angular's **Single Page Application (SPA)** capabilities.



We are starting from where we left off in chapter 2, Building Our First App - 7-Minute Workout. The checkpoint2.4 Git branch can serve as the base for this chapter. The code is also available on GitHub (<https://github.com/chandermani/angular6byexample>) for everyone to download. Checkpoints are implemented as branches in GitHub. If you are not using Git, download the snapshot of checkpoint2.4 (a ZIP file) from the GitHub location: <http://bit.ly/ng6be-checkpoint-2-4>. Refer to the README.md file in the trainer folder when setting up the snapshot for the first time.

Exploring Single Page Application capabilities

The *7-Minute Workout* starts when we load the app, 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 makes the app look more professional and allows us to understand the single-page nomenclature of Angular.

The Angular SPA infrastructure

With modern web frameworks, such as Angular and [Vue.js](#), we are now getting used to apps that do not perform full-page refreshes. But if you are new to this scene, it's worth mentioning what *SPAs* are.

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 and HTML fragments and injected into the already loaded view. Google Mail is a great example of an SPA. SPAs make for a great user experience as the user gets a desktop app-like feel, with no constant post-backs and page refreshes, which are typically associated with traditional web apps.

Like any modern JavaScript framework, [Angular also provides the necessary constructs for SPA implementation](#). Let's understand them and add our app pages too.

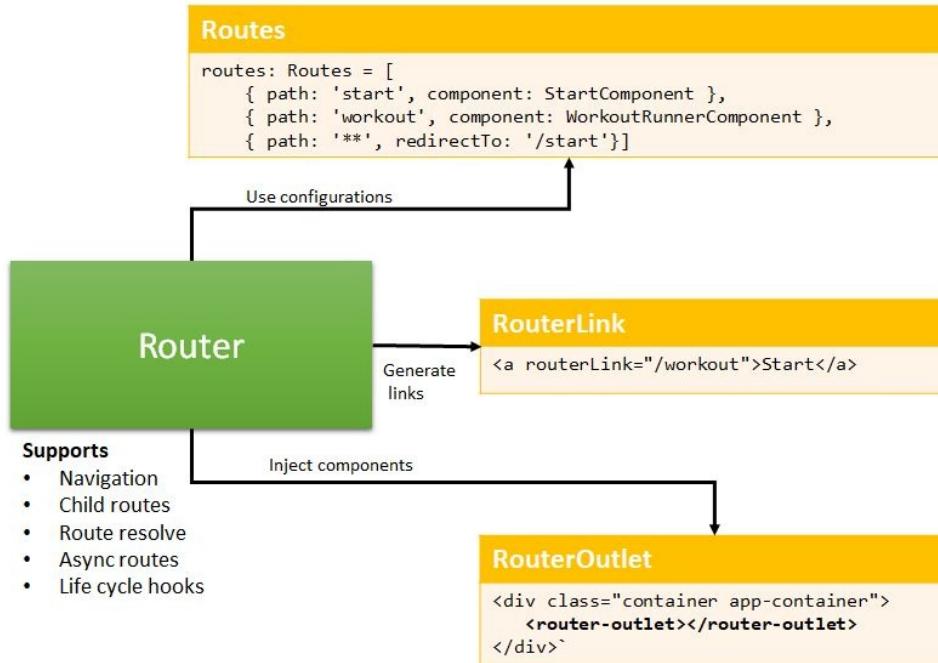
Angular routing

Angular supports SPA development using its **routing infrastructure**. This infrastructure tracks browser URLs, enables hyperlink generation, exposes routing events, and provides a set of directives/components for views that react to URL changes.

There are four major framework pieces that work together to support the Angular routing infrastructure:

- **The Router (Router)**: The primary infrastructure piece that actually provides component navigation
- **The Routing configuration (Route)**: The component router is dependent upon the routing configuration for setting up routes
- **The RouterOutlet component**: The `RouterOutlet` component is the placeholder container (*host*) where route-specific views are loaded
- **The RouterLink directive**: This generates hyperlinks that can be embedded in the anchor tags for navigation

The following diagram highlights the roles these components play within the routing setup:



I highly encourage everyone to keep revisiting this diagram as we set up routing for *7-Minute Workout*.

The router is the central piece of this complete setup; therefore a quick overview of the router will be helpful.

Angular router

If you have worked with any JavaScript framework with SPA support, this is how things work. The framework **watches** the browser URL and serves views based on the URL loaded. There are dedicated framework components for this job. In the Angular world, this tracking is done by a *framework service, the router*.



*In Angular, any class, object, or function that provides some generic functionality is termed a **service**. Angular does not provide any special construct to declare a service as it does for components, directives, and pipes. Anything that can be consumed by components/directives/pipes can be termed a service. The router is one such service. And there are many more services that are part of the framework. If you are from the Angular 1 realm, this is a pleasant surprise-no service, factory, provider, value, or constant!*

The Angular router is there to:

- Enable navigation between components on route change
- Pass routing data between component views
- Make the state of the current route available to active/loaded components
- Provide APIs that allow navigation from component code
- Track the navigation history, allowing us to move back and forward between component views using browser buttons
- Provide life cycle events and guard conditions that allow us to affect navigation based on some external factors



The router also supports some advanced routing concepts, such as parent-child routes. This gives us the ability to define routes at multiple levels inside the component tree. The parent component can define routes and child components can further add more sub-routes to the parent route definition. This is something that we cover in detail in chapter 4, Building Personal Trainer.

The router does not work alone. As highlighted in the preceding diagram, it depends upon other framework pieces to achieve the desired results. Let's add some app pages and work with each piece of the puzzle.

Routing setup

The Angular router is not part of the core Angular framework. It has a separate Angular module and its own npm package. Angular CLI has already installed this package as part of the project setup. Look at `package.json` to confirm this:

```
"@angular/platform-browser-dynamic": "6.0.0",  
"@angular/router": "6.0.0",
```

Since the router is already installed, we just need to integrate it into *7-Minute Workout*.

We can start by adding the `base` reference (highlighted) to the `head` section of index.html, if not present: `<title>Trainer</title>`
<base href="/">

The router requires base href to be set. The `href` value specifies the base URL to use for all relative URLs within an HTML document, including links to CSS, scripts, images, and any other resource. This setting helps the router to create navigation URLs.

Adding start and finish pages

The plan here is to have three pages for *7-Minute Workout*:

- **Start page:** This becomes the landing page for the app
- **Workout page:** What we have currently
- **Finish page:** We navigate to this once the workout is complete

The workout component and its view (`workout-runner.component.ts` and `workout-runner.component.html`) are already there. So let's create `startComponent` and `FinishComponent`.

Again, using the Angular CLI generates the boilerplate for the start and finish components. Navigate to the `trainer/src/app` folder and execute the component-generation command:

```
| ng generate component start -is  
| ng generate component finish -is
```

Next, copy the views for the *start* and *finish* components from the `checkpoints.1` Git branch (the GitHub location to download from is <http://bit.ly/ng6be-3-1-app>).

Both the *start* and *finish* components' implementations are empty. The interesting bits are in the view. The start component view has a link to navigate to the workout runner component (``) and so does finish. We have yet to define the routes.



The start and finish components have been added to app module, as they are rudimentary views, unlike workout runner, which has its own `workoutRunnerModule` module.

All three components are ready. Time to define the route configurations!

Route configuration

To set up the routes for *7-Minute Workout*, we are going to create a *route definition module file*. Create a file called `app-routing.module.ts` in the `trainer/src/app` folder defining the top-level routes for the app. Add the following routing setup or copy it from the `checkpoint3.1` Git branch:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { WorkoutRunnerComponent } from './workout-runner/workout-runner.component';
import { StartComponent } from './start/start.component';
import { FinishComponent } from './finish/finish.component';
```

```
const routes: Routes = [
  { path: 'start', component: StartComponent },
  { path: 'workout', component: WorkoutRunnerComponent },
  { path: 'finish', component: FinishComponent },
  { path: '**', redirectTo: '/start' }
];
```

```
@NgModule({
  imports: [RouterModule.forRoot(routes, { enableTracing: true })],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```



The Angular CLI also supports boilerplate route-generation for modules. We have not used that feature. We can learn about it from the CLI documentation at <http://bit.ly/ng-cli-routing>.

The `routes` variable is an array of `Route` objects. Each `Route` defines the configuration of a single route, which contains:

- `path`: The target path to match
- `component`: The component to be loaded when the path is hit

Such a route definition can be interpreted as, "when the user navigates to a URL

(defined in `path`), load the corresponding component defined in the `component` property." Take the first route example; navigating to `http://localhost:9000/start` loads the component view for `startComponent`.

You may have noticed that the last `Route` definition looks a bit different. `path` looks odd and it does not have a `component` property either. A path with `**` denotes a catch-all path or the **wildcard route** for our app. Any navigation that does not match one of the first three routes matches the catch-all route, causing the app to navigate to the start page (defined in the `redirectTo` property).



We can try this once the routing setup is complete. Type any random route, such as `http://localhost:9000/abcd`, and the app automatically redirects to `http://localhost:9000/start`.

We finally create and import a new module into `AppRoutingModule` with the call to `RouterModule.forRoot`. And by re-exporting Angular's `RouterModule`, we can import `AppRoutingModule` instead of `RouterModule` and have access to all of the routing constructs together with our app routes available in `AppModule`.



The `enableTracing: true` property on the `forRoot` function parameter allows us to monitor the router events (such as `NavigationStart`, `NavigationEnd`, and `NavigationCancel`) that happen when navigation takes place and the correct route is resolved. The logs are visible in the browser's debugger console. Use it for debugging purposes only, remove it from production builds.
Could the preceding routing setup could have been done inside `AppModule`? Yes, it's definitely possible, but we would recommend against it. As the number of routes grow and the routing setup becomes more complex, having a separate routing module helps us organize the code better.

An important thing to highlight here: **route ordering is important in route definition**. Since route matching is done in a top-down fashion, it stops at the first match to define your specific routes before any generic catch-all route, such as the `** wildcard route` in our definition, which is declared at the last.

The default router setup uses the **pushstate** mechanism for URL navigation. In such a setup, URLs look like:

- `localhost:4200/start`
- `localhost:4200/workout`
- `localhost:4200/finish`

This may not seem like a big deal, but remember that we are doing client-side navigation, not the full-page redirects that we are so used to. As the **developer**

guide states: Modern HTML 5 browsers support `history.pushState`, a technique that changes a browser's location and history without triggering a server page request. The router can compose a "natural" URL that is indistinguishable from one that would otherwise require a page load.

Pushstate API and server-side url-rewrites

The router uses the pushstate API in one of two cases:

- When we click on links embedded in the view (`<a>` tag)
- When we use the router API

In both cases, the router intercepts any navigation events, loads the appropriate component view, and finally updates the browser URL. The request never goes to the server.

But what if we refresh the browser window?

The Angular router **cannot** intercept the browser's refresh event, and hence a complete page refresh happens. In such a scenario, the server needs to respond to a resource request (URL) that only exists on the client side. A typical server response is to send the app entry file (such as `index.html`) for any arbitrary request that may result in a `404 (Not Found)` error. This **is what we call the server url-rewrite**. This implies requests to any non-existent URLs, such as `/start`, `/workout`, or `/finish` loads the index page.



Each server platform has a different mechanism to support url-rewrite. We suggest you look at the documentation for the server stack you use to enable url-rewrite for your Angular apps.

We can see the server-side rewrites in action once the app routing is complete. Once completed, try to refresh the app and see the browser's network log; *the server sends the same generated `index.html` content every time irrespective of the URL requested.*

The routing module definition is complete now. Before proceeding further, open `app.module.ts` and import `AppRoutingModule`:

```
import { FinishComponent } from './finish/finish.component';
import { AppRouterModule } from './app-routing.module';
@NgModule({
  imports: [..., StartModule, FinishModule, AppRouterModule],
```

Now that we have all the required components and all the routes defined, where do we inject these components on route change? We just need to define a placeholder for that in the host view.

Rendering component views with router-outlet

Look at the current `AppComponent` template (`app.component.html`), it has an embedded `WorkoutRunnerComponent`:

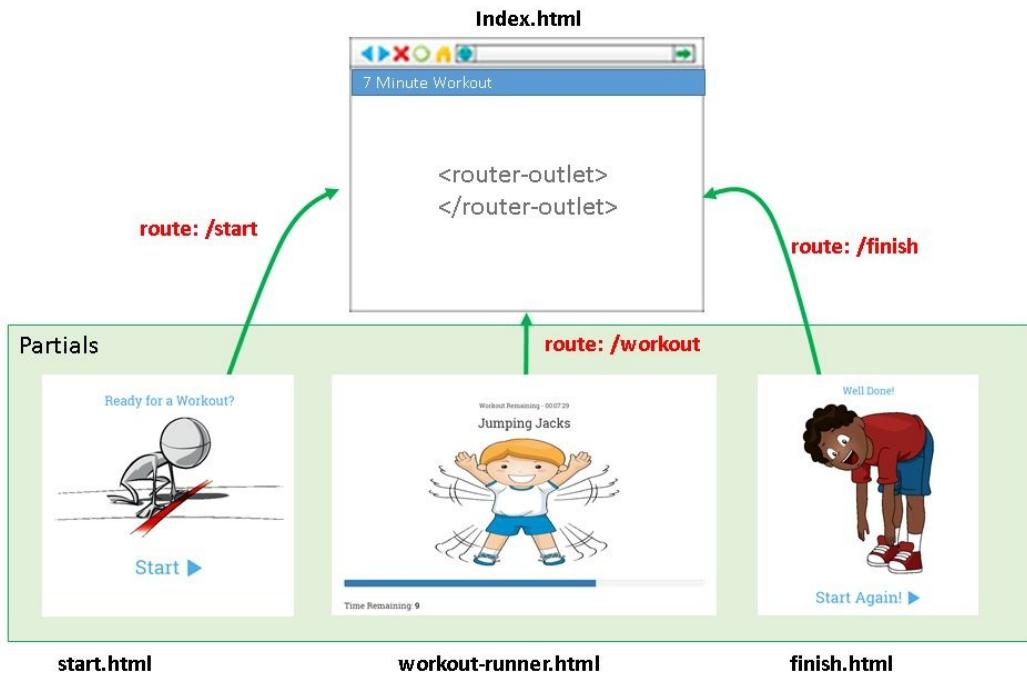
```
| <abe-workout-runner></abe-workout-runner>
```

This needs to change as we need to render different components based on the URL (`/start`, `/workout`, or `/finish`). Remove the preceding declaration and replace it with a **router directive**:

```
| <router-outlet></router-outlet>
```

`RouterOutlet` is an **Angular component directive** that acts as a placeholder for loading route-specific components when the route changes. It integrates with the **router service** to load the appropriate component based on the current browser URL and the route definition.

The following diagram helps us to easily visualize what is happening with the `router-outlet` setup:



We are almost done; it's time to trigger navigation.

Route navigation

Like standard browser navigation, Angular navigation happens:

- When a user enters a URL directly into the browser address bar
- On clicking on a link on the anchor tag
- On using a script/code to navigate

If not started, start the app and load `http://localhost:4200` or `http://localhost:4200/start`. The start page should be loaded.

Click on the Start button and the workout should start under the `http://localhost:4200/workout` URL.

The Angular router also supports the old style of hash (#)-based routing. When hash-based routing is enabled, the routes appear as follows:



- `localhost:9000/#/start`
- `localhost:9000/#/workout`
- `localhost:9000/#/finish`

To change it to hash-based routing, the route configuration for the top-level routes should be augmented with an extra `useHash:true` property in the `RouterModule.forRoot` function (second parameter).

Interestingly, the anchor link in the `StartComponent` view definition does not have an `href` attribute. Instead, there is a `RouterLink` directive (the directive name is `RouterLink`, the selector is `routerLink`):

```
|<a routerLink="/workout">
```

In the preceding case, since the route is fixed, the directive takes a constant expression (`"/workout"`). We are not using the standard square bracket notation (`[]`) here, instead are assigning the directive a fixed value. This is known as **one-time binding**. For **dynamic routes**, we can use template expressions and the link parameter array. We'll touch upon dynamic routes and the link parameter array shortly.



Notice the / prefix in the preceding route path. / is used to specify an absolute path. The Angular router also supports relative paths, which are useful when working with child routes.

— We will explore the concept of child routes in the next few chapters.

Refresh the app and check the rendered HTML for `startComponent`; the preceding anchor tag is rendered with the correct `href` value:

| <a ... href="/workout">
| Avoid hardcoding route links
 While you could have directly used , prefer routerLink to avoid hardcoding routes.

The link parameter array

The route setup for the current *7-Minute Workout* is quite simple, and there isn't a need to pass parameters as part of link generation. But the capability is there for non-trivial routes that require dynamic parameters. See this example:

```
@RouteConfig([ { path: '/users/:id', component: UserDetail }, { path: '/users', component: UserList}, ])
```

This is how the first route can be generated:

```
| <a [routerLink]="/users", 2] // generates /users/2
```

The array assigned to the `RouterLink` directive is what we called the **link parameter array**. The array follows a specific pattern: ['routePath', param1, param2, {prop1:val1, prop2:val2}]

The first element is always the route path, and the next set of parameters is there to replace placeholder tokens defined in a route template.



The Angular router is quite a beast and supports almost everything that we expect from a modern router library. It supports child routes, async routes, lifecycle hooks, secondary routes, and some other advanced scenarios. We'll delay discussion on these topics until later chapters. This chapter just gets us started with Angular routing, but there is more to come!

The router link parameter can also be an object. Such objects are used to supply **optional parameters** to the route. See this example: `<a [routerLink]="/users", {id:2}"// generates /users;id=2`

Note that the generated link contains a semicolon to separate optional parameters from the route and other parameters.

The last missing part of the implementation is routing to the finish page once the workout completes.

Using the router service for component navigation

Navigation from the workout page to the finish page is not triggered manually but on completion of the workout. `WorkoutRunnerComponent` needs to trigger this transition.

For this, `WorkoutRunnerComponent` needs to get hold of the router and invoke the `navigate` method on it.

How does `WorkoutRunnerComponent` get the router instance? Using Angular's *dependency injection framework*. We have been shying away from this topic for some time now. We have achieved a lot without even knowing that there's a dependency injection framework in play all this time. Let's wait a tad longer and first concentrate on fixing the navigation issue.

For `WorkoutRunnerComponent` to get hold of the router service instance, it just needs to declare the service on the constructor. Update the `WorkoutRunnerComponent` constructor and add the imports:

```
| import {Router} from '@angular/router';
| ...
| constructor(private router: Router) {
```

Angular now magically injects the current router into the `router` private variable when `WorkoutRunnerComponent` is instantiated. The magic is done by dependency injection framework.

It's now just a matter of replacing the `console.log("Workout complete!");` statement with the call to the `navigate` router:

```
| this.router.navigate( ['/finish'] );
```

The `navigate` method takes the same *link parameter array* as the RouterLink directive. We can verify the implementation by patiently waiting for the workout to complete!



If you are having a problem running the code, look at the `checkpoint3.1` Git branch for a working version of what we have done thus far. Or if you are not using Git, download the snapshot of `checkpoint3.1` (a ZIP file) from <http://bit.ly/ng6be-checkpoint-3-1>. Refer to the `README.ma` file in the trainer folder when setting up the snapshot for the first time.

The routes we have defined in *7-Minute Workout* are standard simple routes. But if there are dynamic routes that take parameters, how do we make the parameters available in our components? Angular has a service for that, the `ActivatedRoute` service.

Using the ActivatedRoute service to access route params

There are times when the app requires access to the active route state. Information such as the current URL fragment, the current route parameters, and other route-related data may come in handy during a component's implementation.

The `ActivatedRoute` service is a one-stop shop for all current route-related queries. It has a number of properties, including `url` and `paramMap`, that can be used to query the route's current state.

Let's look at an example of a parameterized route and how to access a parameter passed from a component, given this route:

```
| { path: '/users/:id', component: UserDetailComponent },
```

When the user navigates to `/user/5`, the underlying component can access the `:id` parameter value by first injecting `ActivatedRoute` into its constructor:

```
| export class UserDetailComponent {  
|   constructor( private route: ActivatedRoute ...
```

And then, anywhere in the code where the parameter is required, call `get` methods on the `ActivatedRoute.paramMap` property:

```
| ngOnInit() {  
|   let id = +this.route.paramMap.get('id'); // (+) converts string 'id' to a number  
|   var currentUser=this.getUser(id)  
}
```



*The `paramMap` property on `ActivatedRoute` is actually an **observable**. We will learn more about observables later in the chapter, but for now, it is enough to understand that observables are objects that let the outside world know about their state changes by raising events that others can listen into.*

We'll use this router capability in the later chapters where we build a new app that can create workouts and edit existing workouts. In the upcoming chapters, we also look at some advanced routing concepts, including *child routes*, *lazy loaded routes*, and *guard conditions*.

We have covered the basics of Angular routing, it's now time to concentrate on a much overdue topic: *Dependency Injection*.

Angular dependency Injection

Angular makes heavy use of dependency injection to manage app and framework dependencies. The surprising part is that we could ignore this topic until we started our discussion on the router. All this time, the Angular dependency injection framework has been supporting our implementation. The hallmark of a good dependency injection framework is that the consumer can use it without bothering too much about the internals and with little ceremony.

If you are not sure what dependency injection is or just have a vague idea about it, an introduction to DI surely does not hurt anyone.

Dependency injection 101

For any application, its components (not to be confused with Angular components) do not work in isolation. There are dependencies between them. A component may use other components to achieve its desired functionalities.

Dependency injection is a pattern for managing such dependencies.

The DI pattern is popular in many programming languages as it allows us to manage dependencies 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.

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, which is called **constructor injection** (Angular also does this) or by directly setting the object properties, which is called **property injection**.

Here is a rudimentary example of DI in action. Consider a class called `Tracker` that requires `Logger` for a logging operation:

```
class Tracker() {  
    logger:Logger;  
    constructor() {  
        this.logger = new Logger();  
    }  
}
```

The dependency of the `Logger` class is hardwired inside `Tracker` as `Tracker` itself instantiates the `Logger` instance. What if we externalize this dependency? So the class becomes:

```
class Tracker {  
    logger:Logger;  
    constructor(logger:Logger) {  
        this.logger = logger;  
    }  
}
```

This innocuous-looking change has a major impact. By adding the ability to provide the dependency externally, we can now:

- **Decouple these components and enable extensibility.** The DI pattern allows us to alter the logging behavior of the `Tracker` class without touching the class itself. Here is an example:

```
| var trackerWithDBLog=new Tracker(new DBLogger());
  var trackerWithMemoryLog=new Tracker(new MemoryLogger());
```

The two `Tracker` objects we just saw have different logging capabilities for the same `Tracker` class implementation. `trackerWithDBLog` logs to a DB and `trackerWithMemoryLog` to the memory (assuming both `DBLogger` and `MemoryLogger` are derived from the `Logger` class). Since `Tracker` is not dependent on a specific implementation of `Logger` (`DBLogger` or `MemoryLogger`), this implies `Logger` and `Tracker` are loosely coupled. In the future, we can derive a new `Logger` class implementation and use that for logging without changing the `Tracker` implementation.

- **Mock dependencies:** The ability to mock dependencies makes our components more testable. The `Tracker` implementation can be tested in isolation (unit testing) by providing a mock implementation for `Logger`, such as `MockLogger`, or by using a mocking framework that can easily mock the `Logger` interface.

We can now understand how powerful DI is.

Think carefully: once DI is in place, the responsibility for resolving the dependencies now falls on the calling/consumer code. In the preceding example, a class that was earlier instantiating `Tracker` now needs to create a `Logger` derivation and inject it into `Tracker` before using it.

Clearly, this flexibility in swapping internal dependencies of a component comes at a price. The calling code implementation can become overly complex as it now has to manage child dependencies too. This may seem simple at first, but given the fact that dependent components may themselves have dependencies, what we are dealing with is a complex dependency-tree structure.

This is where DI containers/frameworks add value. They make managing dependencies less cumbersome for the calling code. These containers then construct/manage dependencies and provide it to our client/consumer code.

The Angular DI framework manages dependencies for our Angular components,

directives, pipes, and services.

Exploring dependency injection in Angular

Angular employs its very own DI framework to manage dependencies across the application. The very first example of visible dependency injection was the injection of the component router into `WorkoutRunnerComponent`:

```
|constructor(private router: Router) {
```

When the `WorkoutRunnerComponent` class gets instantiated, the DI framework internally locates/creates the correct router instance and injects it into the caller (in our case, `WorkoutRunnerComponent`).

While Angular does a good job of keeping the DI infrastructure hidden, it's imperative that we understand how Angular DI works. Otherwise, everything may seem rather magical.

DI is about creating and managing dependencies, and the framework component that does this is dubbed the **the injector**. For the injector to manage dependencies, it needs to understand the following:

- **The what:** What is the dependency? The dependency could be a class, an object, a factory function, or a value. Every dependency needs to be registered with the DI framework before it can be injected.
- **The where/when:** The DI framework needs to know where to inject a dependency and when.
- **The how:** The DI framework also needs to know the recipe for creating the dependency when requested.

Any injected dependency needs to answer these questions, irrespective of whether it's a framework construct or an artifact created by us.

Take, for example, the `Router` instance injection done in `WorkoutRunnerComponent`. To answer the what and how parts, we register the `Router` service by importing the `RouterModule` into `AppRoutingModule`:

```
| imports: [..., AppRoutingModule];
```

The `AppRoutingModule` is a module that exports multiple routes together with all the Angular-router-related services (technically it re-exports `RouterModule`).

The where and when are decided based on the component that requires the dependencies. The constructor of `WorkoutRunnerComponent` takes a dependency of `Router`. This informs the injector to inject the current `Router` instance when `WorkoutRunnerComponent` is created as part of route navigation.



Internally, the injector determines the dependencies of a class based on the metadata reflected from it when converting TypeScript to ES5 code (done by the TypeScript compiler). The metadata is generated only if we add a decorator, such as `@component` or `@Pipe`, on the class.

What happens if we inject `Router` into another class? Is the same `Router` instance used? The short answer is yes. The Angular injector creates and caches dependencies for future reuse, and hence these services are singleton in nature.



While dependencies in an injector are singleton, at any given time, there can be multiple injectors active throughout an Angular app. You'll learn about the injector hierarchy shortly. With the router, there is another layer of complexity. Since Angular supports the child route concept, each of these child routes has its own router instance. Wait until we cover child routers in the next chapter so that you can understand the intricacies!

Let's create an Angular service to track workout history. This process will help you understand how dependencies are wired using Angular DI.

Tracking workout history

It would be a great addition to our app if we could track our workout history. When did we last exercise? Did we complete it? How much time did we spend on it?

To answer these questions, we need to track when the workout starts and when it ends. This tracking data then needs to be persisted somewhere.

A possible solution could be to extend our `WorkoutRunnerComponent` with the desired functionality. But that adds unnecessary complexity to `WorkoutRunnerComponent` and that's not its primary job.

~~We need a dedicated history-tracking service for this job, a service that tracks historical data and shares it throughout the app. Let's start building the workout-history-tracker service.~~

Building the workout-history-tracker service

The workout-history-tracker service is going to track workout progress. The service will also expose an interface, allowing `WorkoutRunnerComponent` to start and stop workout tracking.

Inspired again by the *Angular style guide*, we are going to create a new module, **core module**, and add the service to this module. The role of the core module is to host services that are available across the application. It is also a good place to add single-use components that are required when the application starts. A nav bar and busy indicator are good examples of such components.

On the command line, navigate to the `trainer/src/app` folder and generate a new module:

```
| ng generate module core --module app
```

This creates a new `coreModule` module and imports it into `AppModule`. Next, create a new service inside the `trainer/src/app/core` folder, again using Angular CLI:

```
| ng generate service workout-history-tracker
```

The generated code is quite simple. The generator creates a new class `WorkoutHistoryTrackerService` (`workout-history-tracker.service.ts`) with a `@Injectable` decorator applied on the class:

```
@Injectable({
  providedIn: 'root'
})
export class WorkoutHistoryTrackerService {
  ...
}
```

The `providedIn: 'root'` property on `Injectable` instructs Angular to create a **provider** with the *root injector*. The sole job of this provider is to create the `WorkoutHistoryTrackerService` service and return it when Angular's DI injector desires. Any service that we create/use needs to be registered on an injector. As

the Angular documentation on *providers* describes,

Providers tell the injector how to create the service. Without a provider, the injector would not know that it is responsible for injecting the service nor be able to create the service.



A service in Angular is just a class that has been registered with Angular's DI framework.
Nothing special about them!

Sometimes it is desirable to include the service as part of a module instead of registering it with the root injector. In such a case, the service can be registered at the module level. There are two ways to achieve this:

- **Option 1:** Reference the module with the `providedIn` property:

```
|@Injectable({  
|  providedIn: CoreModule  
|})  
export class WorkoutHistoryTrackerService {
```

- **Option 2:** Register the service on the module, using the `providers` array:

```
|@NgModule({  
|  providers: [WorkoutHistoryTrackerService],  
|})  
export class CoreModule { }
```

Registering services at module level is advantageous in scenarios where a module is lazy loaded.

Registering the service using `Injectable` (*option 1*) has another advantage. It enables Angular CLI build to perform advanced optimization with code bundling, leaving out any service that is declared but never used (a process called **tree shaking**).



Irrespective of the two options we use, the service is still registered (via a provider) with the root injector.

We are going to use the `Injectable` approach to registering dependency throughout the book, unless stated otherwise. Open `workout-history-tracker.service.ts` and add the following implementation:

```
|import { ExercisePlan } from '../workout-runner/model';
```

```

import { CoreModule } from './core.module';
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: CoreModule
})
export class WorkoutHistoryTrackerService {
  private maxHistoryItems = 20; //Tracking last 20 exercises
  private currentWorkoutLog: WorkoutLogEntry = null;
  private workoutHistory: Array<WorkoutLogEntry> = [];
  private workoutTracked: boolean;

  constructor() { }

  get tracking(): boolean {
    return this.workoutTracked;
  }
}

export class WorkoutLogEntry {
  constructor(
    public startedOn: Date,
    public completed: boolean = false,
    public exercisesDone: number = 0,
    public lastExercise?: string,
    public endedOn?: Date) { }
}

```

There are two classes defined: `WorkoutHistoryTrackerService` and `WorkoutLogEntry`. As the name suggests, `WorkoutLogEntry` defines log data for one workout execution. `maxHistoryItems` allows us to configure the maximum number of items to store in the `workoutHistory` array, the array that contains the historical data. The `get tracking()` method defines a getter property for `workoutTracked` in TypeScript. `workoutTracked` is set to `true` during workout execution.

Let's add the start tracking, stop tracking, and exercise complete functions to `WorkoutHistoryTrackerService`:

```

startTracking() {
  this.workoutTracked = true;
  this.currentWorkoutLog = new WorkoutLogEntry(new Date());
  if (this.workoutHistory.length >= this.maxHistoryItems) {
    this.workoutHistory.shift();
  }
  this.workoutHistory.push(this.currentWorkoutLog);
}

exerciseComplete(exercise: ExercisePlan) {
  this.currentWorkoutLog.lastExercise = exercise.exercise.title;
  ++this.currentWorkoutLog.exercisesDone;
}

endTracking(completed: boolean) {
  this.currentWorkoutLog.completed = completed;
  this.currentWorkoutLog.endedOn = new Date();
  this.currentWorkoutLog = null;
  this.workoutTracked = false;
}

```

The `startTracking` function creates a `WorkoutLogEntry` and adds it to the `workoutHistory` array. By setting the `currentWorkoutLog` to the newly created log entry, we can manipulate it later during workout execution. The `endTracking` function and the `exerciseComplete` function just alter `currentWorkoutLog`. The `exerciseComplete` function should be called on completion of each exercise that is part of the workout. To save you some keystrokes, get the complete code for implementation done thus far from this gist: <http://bit.ly/ng6be-gist-workout-history-tracker-v1-ts>.

The service implementation now also includes a function to get workout history data:

```
|getHistory(): Array<WorkoutLogEntry> {  
|  return this.workoutHistory;  
|}
```

That completes the `WorkoutHistoryTrackerService` implementation; now it's time to integrate it into the workout execution.

Integrating with WorkoutRunnerComponent

`WorkoutRunnerComponent` requires `WorkoutHistoryTrackerService` to track workout history; hence there is a dependency to be fulfilled. We already have registered `WorkoutHistoryTrackerService` with Angular's DI framework using the `Injectable` decorator and now it's time to consume the service.

Injecting dependencies with Constructor Injection

~~Consuming dependency is easy! More often than not, we use constructor injection to consume a dependency.~~

Add the `import` statement at the top and update the `WorkoutRunnerComponent` constructor, as follows:

```
import { WorkoutHistoryTrackerService } from '../core/workout-history-tracker.service';
...
constructor(private router: Router,
            private tracker: WorkoutHistoryTrackerService
) {
```

~~As with `router`, Angular injects `workoutHistoryTrackerService` too when `WorkoutRunnerComponent` is created. Easy!~~

Once the service is injected and available to `WorkoutRunnerComponent`, the service instance (`tracker`) needs to be called when the workout starts, when an exercise is complete, and when the workout finishes.

Add this as the first statement in the `start` function:

```
| this.tracker.startTracking();
```

In the `startExerciseTimeTracking` function, add the highlighted code after the `clearInterval` call:

```
clearInterval(this.exerciseTrackingInterval);
if (this.currentExercise !== this.restExercise) {
  this.tracker.exerciseComplete(this.workoutPlan.exercises[this.currentExerciseIndex]);
}
```

And the highlighted code inside the workout to complete the `else` condition in the same function:

```
this.tracker.endTracking(true);
this.router.navigate(['/finish']);
```

History tracking is almost complete except for one case. What if the user manually navigates away from the workout page? How do we stop tracking?

When that happens, we can always rely on the component's life cycle hooks/events to help us. Workout tracking can be stopped when the `ngOnDestroy` event is fired. An appropriate place to perform any cleanup work is before the component is removed from the component tree. Let's do it.

Add this function and the corresponding life cycle event interface to `workout-runner.component.ts`:

```
export class WorkoutRunnerComponent implements OnInit, OnDestroy {
  ...
  ngOnDestroy() {
    this.tracker.endTracking(false);
  }
}
```

The workout-history-tracking implementation is complete. We are eager to start the workout history page/component implementation, but not before completing our discussion on Angular's DI capabilities.



Feel free to skip the next section for now if you want to maintain the app-building velocity. Come back to this section with a fresh and relaxed mind. There are some very important core concepts we'll share in the next section.

Dependency injection in depth

Let's first try to understand the different places we can register a dependency using `WorkoutHistoryTrackerService` as an example.

Registering dependencies

The standard way to register a dependency is to register it at the root/global level. This can be done either by passing the dependency type into the `providers` attribute (array) in the `NgModule` decorator, or by using the `providedIn` attribute on the `Injectable` service decorator.

Remember our `WorkoutHistoryTrackerService` registration? Check the following:

```
| @Injectable({  
|   providedIn: CoreModule  
| })  
| export class WorkoutHistoryTrackerService {
```

The same thing can be done on the module declaration too, as shown here:

```
| @NgModule({...providers: [WorkoutHistoryTrackerService],})
```

Technically speaking, when using any of the preceding mechanisms, the service gets registered with the app's root injector, irrespective of the Angular module it is declared in. Any Angular artifact across modules henceforth can use the service (`WorkoutHistoryTrackerService`). No module imports are required at all.



This behavior is different from component/directive/pipe registration. Such artifacts have to be exported from a module for another module to use them.

Another place where dependencies can be registered is on the component. The `@component` decorator has a `providers` array parameter to register dependencies. With these two levels of dependency registration, the obvious question that we need to answer is, which one to use?

It seems obvious that if the dependency is exclusively used by a component and its children, it should be registered at the `@Component` decorator level. Not really! There is much more we need to understand before we can answer this question. There is a whole new world of hierarchical injectors that need to be introduced. Let's wait, and instead learn other ways of registering dependencies, by continuing our discussion on providers.

Providers create dependencies when the Angular injector requests them. These providers have the recipe to create these dependencies. While a class seems to be the obvious dependency that can be registered, we can also register:

- A specific object/value
- A factory function

Registering `WorkoutHistoryTrackerService` using the `Injectable` decorator is the most common pattern of registration. But at times we need some flexibility with our dependency registrations. To register an object or a factory function, we need to use the expanded version of provider registrations available on `NgModule`.

To learn about these variations, we need to explore providers and dependency registration in a little more detail.

Angular providers

Providers create dependencies that are served by the DI framework.

Look at this `WorkoutHistoryTrackerService` dependency registration done on `NgModule`:
providers: [`WorkoutHistoryTrackerService`],

This syntax is a short-form notation for the following version:

```
| providers:{ provide: WorkoutHistoryTrackerService, useClass:  
| WorkoutHistoryTrackerService })
```

The first property (`provide`) is a token that acts as a key for registering a dependency. This key also allows us to locate the dependency during dependency injection.

The second property (`useClass`) is a provider definition object that defines the recipe for creating the dependency value.

With `useClass`, we are registering a **class provider**. A class provider creates dependencies by instantiating the type of object requested for. There are some other provider types.

Value providers

A *class provider* creates class objects and fulfills the dependency, but at times we want to register a specific object/primitive with the DI provider instead. **Value providers** solve this use case.

Had we registered `WorkoutHistoryTrackerService` using this technique, the registration would have looked like this:

```
| {provide: WorkoutHistoryTrackerService, useValue: new WorkoutHistoryTrackerService()};
```

With the value provider, we have the responsibility to provide an instance of a service/object/primitive to Angular DI.

With the *value provider*, since we are creating the dependency manually, we are also responsible for constructing any child dependencies if there are dependencies further down the lineage. Take the example of `WorkoutHistoryTrackerService` again. If `WorkoutHistoryTrackerService` has some dependencies, those too need to be fulfilled by manual injection:

```
| {provide: WorkoutHistoryTrackerService, useValue: new WorkoutHistoryTrackerService(new LocalStorage())};
```

In the preceding example, we not only have to create an instance of `WorkoutHistoryTrackerService`, we also have to create an instance of the `LocalStorage` service. For a service with a complex dependency graph, setting up that service with a value provider becomes challenging.



Wherever possible, prefer class provider over value provider.

Value providers still come in handy in specific scenarios. For example, we can register a common app configuration using a value provider:

```
| {provide: AppConfig, {useValue: {name:'Test App', gridSetting: {...} ...}}}
```

Or register a mock dependency while unit testing:

```
| {provide:WorkoutHistoryTrackerService, useValue: new MockWorkoutHistoryTracker()}
```

Factory providers

There are times when dependency construction is not a trivial affair.

Construction depends upon external factors. These factors decide what objects or class instances are created and returned. **Factory providers** do this heavy lifting.

Take an example where we want to have different configurations for dev and production releases. We can very well use a factory implementation to select the right configuration:

```
{provide: AppConfig, useFactory: () => {
  if(PRODUCTION) {
    return {name:'Production App', gridSetting: {...} ...}
  }
  else {
    return {name:'Test App', gridSetting: {...} ...}
  }
}}
```

A factory function can have its own dependencies too. In such a case, the syntax changes a bit:

```
{provide: WorkoutHistoryTrackerService, useFactory: (environment:Environment) => {
  if(Environment.isTest) {
    return new MockWorkoutHistoryTracker();
  }
  else {
    return new WorkoutHistoryTrackerService();
  },
  deps:[Environment]
}}
```

The dependencies are passed as parameters to the factory function and registered on the provider definition object property, deps (Environment is the dependency injected in the preceding example).

Use the useFactory provide if the construction of the dependency is complex and not everything can be decided during dependency wire-up.

While we have a number of options to declare dependencies, consuming dependencies is far simpler. We saw one way of constructor injection in the *Injecting dependencies with Constructor Injection* section.

Explicit injection using injector

We can even do explicit injections using Angular's **Injector service**. This is the same injector Angular uses to support DI. Here is how to inject the `WorkoutHistoryTrackerService` service using `Injector: constructor(private router: Router, private injector:Injector) {`

```
this.tracker=injector.get(WorkoutHistoryTrackerService);
```

We inject the `Injector` service and then ask for the `WorkoutHistoryTrackerService` instance explicitly.

When would someone want to do this? Well, almost never. Avoid this pattern as it exposes the DI container to your implementation and adds a bit of noise.

We now know how to register dependency and how to consume it, but how does the DI framework locate these dependencies?

Dependency tokens

Remember this expanded version of dependency registration shown earlier:

```
| { provide: WorkoutHistoryTrackerService, useClass: WorkoutHistoryTrackerService }
```

The `provide` property value is a **token**. This token is used to identify the dependency to inject. In the preceding example, we use class name or type to identify dependency and hence the token is called a **class token**.

Based on the preceding registration, whenever Angular sees a statement similar to the following, it injects the correct dependency based on the class type, here `WorkoutHistoryTrackerService`:

```
| constructor(tracker: WorkoutHistoryTrackerService)
```

Angular supports some other tokens too.

Using InjectionToken

There are times when the dependency we define is either a primitive, object, or function. In such a scenario, the class token cannot be used as there is no class. Angular solves this problem using `InjectionToken` (or **string tokens** as we'll see later). The app configuration registration examples that we shared earlier can be rewritten using string tokens if there is no `AppConfig` **class**.

To register a dependency using `InjectionToken`, we first need to create the `InjectionToken` class instance: `export const APP_CONFIG = new InjectionToken('Application Configuration');`

Then, use the token to register the dependency:

```
| { provide: APP_CONFIG, useValue: {name:'Test App', gridSetting: {...} ...}};
```

And finally, inject the dependency anywhere using the `@Inject` decorator:

```
| constructor(@Inject(APP_CONFIG) config) { }
```

 Interestingly, when `@Inject()` is not present, the injector uses the type/class name of the parameter (class token) to locate the dependency.

Using string tokens

Angular also supports **string tokens**, allowing us to use string literals to identify and inject dependencies. The preceding example with a string token becomes:

```
| { provide: 'appconfig', useValue: {name:'Test App', gridSetting: {...} ...});  
| ...  
| constructor(@Inject('appconfig') config) { }
```

A downside to string tokens is that you can misspell the token across the declaration and injection.

Phew! This was a really long section on Angular's Dependency Injection and there's still stuff left to cover. For now, let's get back on track and add the workout history page.

Adding the workout history page

The workout history data that we are collecting during the execution of the workout can now be rendered in a view. Let's add a `History` component. The component will be available at the `/history` location and can be loaded by clicking on a link in the app header section.

Update the route definition in `app.routes.ts` to include a new route and the related import:

```
import { WorkoutHistoryComponent } from './workout-history/workout-history.component';
...
export const routes: Routes = [
  ...
  { path: 'history', component: WorkoutHistoryComponent },
  { path: '**', redirectTo: '/start' }
])
```

The History link needs to be added to the app header section. Let's refactor the header section into its own component. Update the `app.component.html` template and replace the `nav` element with the following code:

```
<div id="header">
  <abe-header></abe-header>
</div>
```

The `nav` element moves into a header component, which we still need to create. Use `ng generate` to generate a new `HeaderComponent` component by running the command inside the `trainer/src/app/core` folder:

```
| ng generate component header -is
```

This statement creates a new header component and declares it on the core module. Next, update the definition for the header component (`header.component.ts`) and its view (`header.component.html`) from the `checkpoint3.2` Git branch (GitHub location: <http://bit.ly/ng6be-3-2-header>).

While we have added the `header` element to `app.component.html`, the `header` component will not render unless we import the `core` module and export the component from the `core` module. Angular CLI did the first part for us, for the second part, update `core.module.ts` to:

```
imports: [ CommonModule, RouterModule ],
declarations: [ HeaderComponent ],
exports: [ HeaderComponent ]
```

If you look at the `HeaderComponent` view, the history link is now there. We had to import `RouterModule` as the following link is generated using the `RouterLink` directive, which is part of `RouterModule`:

```
| <a class="nav-link" routerLink="/history" title="Workout History">History</a>
```

Let's add the workout history component by first generating the component's boilerplate. From the command line, navigate to `trainer/src/app` and run:

```
| ng generate component workout-history -is
```

The `WorkoutHistoryComponent` implementation is available in the `checkpoint3.2` Git branch; the folder is `workout-history` (GitHub location: <http://bit.ly/ng6be-3-2-workout-history>).

The `WorkoutHistoryComponent` view code is trivial, to say the least: a few Angular constructs, including `ngFor` and `ngIf`. The component implementation too is pretty straightforward. Inject the `WorkoutHistoryTrackerService` service dependency and load the history data when `WorkoutHistoryComponent` is initialized:

```
| ngOnInit() {
  this.history = this.tracker.getHistory();
}
```

And this time, we use the `Location` service instead of `Router` to navigate away from the `history` component:

```
| goBack() {
  this.location.back();
}
```

The `Location` service is used to interact with the browser URL. Depending upon the URL strategy, either URL paths (such as `/start` or `/workout`) or URL hash segments (such as `#/start` or `#/workout`) are used to track location changes. The router service internally uses the location service to trigger navigation.



Router versus Location

While the `Location` service allows us to perform navigation, using `Router` is a preferred way of performing route navigation. We used the location service here because the need was to navigate to the last route without bothering about how to construct the route.

We are ready to test our workout history implementation. Load the start page and click on the History link. The history page is loaded with an empty grid. Go back, start a workout, and let an exercise complete. Check the history page again; there should be a workout listed:

The screenshot shows a web application interface. At the top, a blue header bar contains the text "7 Minute Workout" on the left and "History" on the right. Below the header is a table titled "All Workouts:". The table has six columns: "No", "Started", "Ended", "Last Exercise", "Exercises Done", and "Completed". A single row of data is present in the table, corresponding to the session listed in the header.

No	Started	Ended	Last Exercise	Exercises Done	Completed
1	1/10/2016, 9:49 AM	1/10/2016, 9:57 AM	Side Plank	12	Yes

Looks good! If we run the workout multiple times and let the history list build, we realize there is one sore point in this listing. Historical data is not sorted in reverse-chronological order, with the newest at the top. Also, it would be great if we had some filtering capabilities.

Sorting and filtering history data using pipes

In [Chapter 2](#), *Building Our First App – 7-Minute Workout*, we explored pipes. We even built our own pipe to format the seconds values as hh:mm:ss. The primary purpose of pipes is to transform data and, surprisingly, they work on arrays too! For arrays, pipes can sort and filter data. We create two pipes, one for each sorting and filtering.



AngularJS has prebuilt filters ([filters are pipes in Angular](#)), `orderBy` and `filter`, for this very purpose. Angular does not come with these pipes and there is a good reason. These pipes are prone to poor performance. Learn the rationale behind this decision in the framework documentation on pipes at <http://bit.ly/ng-no-filter-orderby-pipe>.

Let's start with the `orderBy` pipe.

The orderBy pipe

The `orderBy` pipe we implement is going to order an array of objects based on any of the object's properties. The usage pattern for sorting items in ascending order based on the `fieldName` property is going to be:

```
| *ngFor="let item of items| orderBy:fieldName"
```

And for sorting items in descending order, the usage pattern is:

```
| *ngFor="let item of items| orderBy:-fieldName"
```

Make note of the extra hyphen (-) before `fieldName`.

We plan to add `orderByPipe` in a new shared module. Are you thinking, why not add it to core module? By convention, the core module contains global services and one-time-use components. This is exactly one core module per application. Shared modules, on the other hand, have components/directives/pipes that are shared across modules. Such shared modules can be also be defined at multiple levels, across the parent and child modules. In this case, we will define the shared module inside `AppModule`.

Create a new `SharedModule` module by running this command in the `trainer/src/app` directory:

```
| ng generate module shared --module app
```

From the command line, navigate to the `trainer/src/app/shared` folder and generate the order by pipe boilerplate:

```
| ng generate pipe order-by
```

Open `order-by.pipe.ts` and update the definition from the checkpoint3.2 code (GitHub location: <http://bit.ly/ng6be-3-2-order-by-pipe>). While we are not going to delve into the pipe's implementation details, some relevant parts need to be highlighted. Look at this pipe outline:

```
| @Pipe({ name: 'orderBy' })
| export class OrderByPipe {
|   transform(value: Array<any>, field:string): any {
```

```
| } ...
```

The preceding `field` variable receives the field on which sorting is required. If the field has a `-` prefix, we truncate the prefix before sorting the array in descending order.



The pipe also uses the [spread operator](#), `[...]`, which may be new to you. Learn more about the spread operator on MDN at <http://bit.ly/js-spread>.

To use `orderByPipe`, update the template view for workout history:

```
|<tr *ngFor="let historyItem of history|orderBy:'-startedOn'; let i = index">
```

And again, we need to export the pipe from the shared module allowing `WorkoutHistoryComponent` to use it. Add an `exports` property on `SharedModule` and set it to `OrderByPipe`:

```
| declarations:[...],  
| exports:[OrderByPipe]
```

The historical data will now be sorted in descending order on the `startedOn` field.



Make note of the single quotes around the pipe parameter (`'-startedOn'`). We are passing a literal string to the `orderBy` pipe. Pipe parameters support data binding and can be bound to component properties too.

That's enough for the `orderBy` pipe. Let's implement filtering.

Pipe chaining with search pipe

We start by creating the search pipe boilerplate by running the following command from the `trainer/src/app/shared` folder:

```
| ng generate pipe search
```

The implementation can now be copied from [checkpoint3.2](#) (GitHub location: <https://github.com/microsoft/TypeScript-DotNet/tree/3.2/search-pipe>). `SearchPipe` does a basic equality-based filtering. Nothing special.

Look at the pipe code; the pipe takes two arguments, the first being the field to search, and the second the value to search. We use the [JavaScript array's filter](#) function to filter the record, doing a strict equality check. Wondering about the `pure` attribute on the `Pipe` decorator? This is going to be the subject of discussion in the next section.

Let's update the workout history view and incorporate the search pipe too. Open `workout-history.component.html` and uncomment the div with radio buttons. These radio buttons filter workouts based on whether they were completed or not. This is how the HTML filter selection looks:

```
<label><input type="radio" name="searchFilter" value=""  
  (change)="completed = null" checked="">All </label>  
<label><input type="radio" name="searchFilter" value="true"  
  (change)="completed = $event.target.value=='true'"> Completed </label>  
<label><input type="radio" name="searchFilter" value="false"  
  (change)="completed = $event.target.value=='true'"> Incomplete </label>
```

We define three filters: `all`, `completed`, and `incomplete` workouts. The radio selection sets the component's `completed` property using the `change` event expression. `$event.target` is the radio button that was clicked.

The `search` pipe can now be added to the `ngFor` directive expression. We are going to chain the `search` and `orderBy` pipes. Update the `ngFor` expression to:

```
<tr *ngFor="let historyItem of history |search:'completed':completed |orderBy:'-
```

A great example of Angular's pipe chaining capabilities!

As we did with `orderByPipe`, `searchPipe` too needs to be exported from the shared module before using it.

The `search` pipe first filters the historical data, followed by the `orderBy` pipe reordering it. Pay close attention to the `search` pipe parameters: the first parameter is a string literal denoting the field to search (`historyItem.completed`), whereas the second parameter is derived from the component's `completed` property. Having the ability to bind pipe parameters to component properties allows us great flexibility.

Go ahead and verify the search capabilities of the history page. Based on the radio selection, the history records are filtered, and of course, they are sorted in reverse-chronological order based on the workout start dates.

While pipe usage with arrays looks simple, it can throw up some surprises if we do not understand when pipes are evaluated.

Pipe gotcha with arrays

To understand the issue with pipes applied to arrays, let's reproduce the problem.

Open `search.pipe.ts` and remove the `@Pipe` decorator's `pure` attribute. Also, take the following statement:

```
| if (searchTerm == null) return [...value];
```

And change it into this:

```
| if (searchTerm == null) return [value];
```

Add a button at the end of the radio list (in `workout-history.component.html`) that adds a new log entry to the `history` array:

```
| <button (click)="addLog()">Add Log</button>
```

Add a function to `workoutHistoryComponent`:

```
| addLog() {  
| }  
|   this.history.push(Object.assign({}, this.history[this.history.length-1]));
```

The preceding function duplicates the first history item and adds back to the `history` array. If we load the page and click on the button, a new log entry gets added to the history array, but it does not show up on the view unless we change the filter (by clicking on the other radios). Interesting!



Before calling `addLog` make sure at least one history log is already there; otherwise the `addLog` function will fail.

The pipes that we have built thus far are **stateless** (also called **pure**) in nature. They simply transform input data into output. **Stateless pipes** are reevaluated only if the pipe input changes (the expression on the left side of pipe symbol) or any pipe argument is updated.

For arrays, this happens on an array assignment/reference change and not on the addition or deletion of elements. Switching the filter condition works, as it

causes the search pipe to evaluate again as the search parameter (the completed status) changes. This behavior is something to be aware of.

What's the fix? For starters, we can make the history array immutable, which implies that it cannot be changed once created. To add a new element, we need to create a new array with the new value, something like:

```
| this.history = [...this.history, Object.assign({}, this.history[0])];
```

This works perfectly, but we are changing our implementation to make it work with pipes which is incorrect. Instead, we can change the pipe. The pipe should be marked stateful.

The difference between a stateless and stateful pipe is that stateful pipes are evaluated by Angular every time the framework does a change-detection run, which involves checking the complete application for changes. Therefore, with stateful pipes, the check is not limited to the pipe input/argument changes.

To make a `search` pipe stateless, just revert the first change we made and add back `pure: false` on the `Pipe` decorator:

```
| @Pipe({
|   name: 'search',
|   pure:false
| })
```

It still does not work! The `search` pipe has one more quirk that needs a fix. The All radio selection does not work perfectly. Add a new workout log, and it still will not show up, unless we switch filters.

The fix here is to revert the second change. Isolate this line in the `search` pipe:

```
| if (searchTerm == null) return value;
```

And change it to the following:

```
| if (searchTerm == null) return [...value];
```

We changed the `if` condition to return a new array every time (using the spread operator), even when `searchTerm` is `null`. If we return the same array reference, Angular does not check for a size change in the array and hence does not update the UI.

That completes our History page implementation. You may now be wondering what the last few fixes on pipes have to do with how change detection works. Or you may be wondering what change detection is. Let's put all of these doubts to rest and introduce everyone to *Angular's change-detection system*.



Angular's change detection will be covered extensively in [Chapter 8, Some Practical Scenarios](#). The aim of the next section is to introduce the concept of change detection and how Angular performs this process.

Angular change detection overview

To put it succinctly, change detection is all about tracking changes done to the component's model during app execution. This helps Angular's data-binding infrastructure to identify what parts of the view need to be updated. Every data binding framework needs to address this issue, and the approach these frameworks take for tracking changes differs. It even differs from [AngularJS to Angular](#).

To understand how change detection works in Angular, there are a few things that we need to keep in mind.

- An Angular app is nothing but a hierarchy of components, from the root to the leaf.
- There is nothing special about the component properties that we bind to view; therefore Angular needs an efficient mechanism to know when these properties change. It cannot keep polling for changes in these properties.
- To detect changes in a property value, Angular does a [*strict comparison*](#) (`==`) between the previous and current value. For reference types, it means only the references are compared. No deep comparison is done.



For precisely this reason, we had to mark our search pipe [as stateful](#). Adding elements to an existing array does not change the array reference and hence Angular fails to detect any change in the array. Once the pipe is marked [as stateful](#), the pipe is evaluated, irrespective of whether the array has changed or not.

Since Angular cannot know when any bound property is updated automatically, it instead resorts to checking every bound property when a change detection run is triggered. Starting [from the root of the component tree](#), Angular checks each bound property for changes as it goes down the component hierarchy. If a change is detected, that component is [marked for refresh](#). It's worth reiterating that changes in a bound property [do not immediately update the view](#). Instead, a change-detection run works [in two phases](#).

- In the *first phase*, it does the component tree walk and [marks](#) components that need to be refreshed due to model updates

- In the *second phase*, the actual view is synchronized with the underlying model



Model changes and view updates are never interleaved during a change-detection run.

We now just need to answer two more questions:

- When is a change-detection run triggered?
- How many times does it run?

An Angular change-detection run is triggered when any of these events are triggered:

- **User input/browser events:** We click on a button, enter some text, scroll the content. Each of these actions can update the view (and the underlying model).
- **Remote XHR requests:** This is another common reason for view updates. Getting data from a remote server to show on the grid and getting user data to render a view are examples of this.
- **setTimeout and setInterval:** As it turns out, we can use `setTimeout` and `setInterval` to execute some code asynchronously and at specific intervals. Such code can also update the model. For example, a `setInterval` timer may check for stock quotes at regular intervals and update the stock price on the UI.

To answer how many times, **it's one**. Each component model is checked only once, in a top-down fashion, starting from the root component to the tree leaves.



The last statement is true when Angular is configured to run in production mode. In development mode, the component tree is traversed twice for changes. Angular expects the model to be stable after the first tree walk. If that is not the case, Angular throws an error in development mode, and ignores the changes in production mode. We can enable the production mode by invoking the `enableProdMode` function before the `bootstrap` function call.

It's time now to pick another topic linked to Angular's dependency injection. The concept of **hierarchical injectors** will be our next topic of discussion. It is a very powerful feature that can come in handy as we build bigger and better apps using Angular.

Hierarchical injectors

An **injector** in Angular's dependency injection setup is a container that is responsible for storing dependencies and dispensing them when asked for. The provider registration examples shared earlier actually register the dependencies with a global injector.

Registering component-level dependencies

All of the dependency registrations that we have done thus far were done on a module. Angular goes one step further and allows registration of dependencies at the component level too. There is a similar `providers` attribute on the `@Component` decorator that allows us to register dependency at the component level.

We could've very well registered the `WorkoutHistoryTrackerService` dependency on `WorkoutRunnerComponent`. Something along these lines:

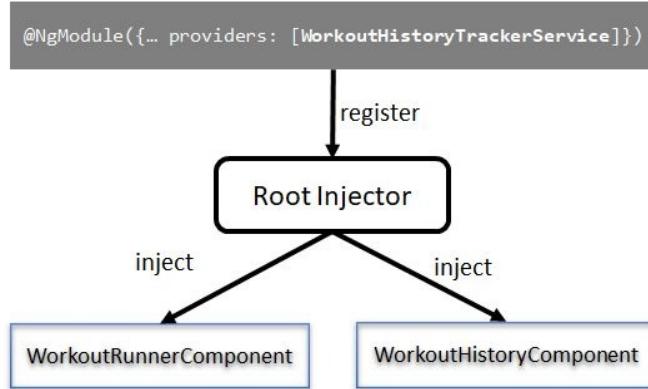
```
 @Component({
  selector: 'abe-workout-runner',
  providers: [WorkoutHistoryTrackerService]
  ...
})
```

But whether we should do it or not is something that we discuss here.

In the context of this discussion on hierarchical injectors, it's important to understand that Angular creates an injector per component (oversimplified). Dependency registration done at the component level is available on the component and its descendants.

We also learned that dependencies are singleton in nature. Once created, the injector will always return the same dependency every time. This feature is evident from the workout history implementation.

`WorkoutHistoryTrackerService` was registered with the `coreModule` and then injected into two components: `WorkoutRunnerComponent` and `WorkoutHistoryComponent`. Both components get the same instance of `WorkoutHistoryTrackerService`. The next diagram highlights this registration and injection:



To confirm, just add a `console.log` statement in the `WorkoutHistoryTrackerService` constructor:

```
| console.log("WorkoutHistoryTrackerService instance created.")
```

Refresh the app and open the history page by clicking on the header link. The message log is generated once, irrespective of how many times we run the workout or open the history page.

That's also a new interaction/data flow pattern!

Think carefully; a service is being used to share state between two components. `WorkoutRunnerComponent` is generating data and `WorkoutHistoryComponent` is consuming it. And without any interdependence. We are exploiting the fact that dependencies are singleton in nature. This data-sharing/interaction/data-flow pattern can be used to share state between any number of components. Indeed, this is a very powerful weapon in our arsenal. The next time there is a need to share state between unrelated components, think of services.

But what does all this have to do with hierarchical injectors? OK, let's not beat around the bush; let's get straight to the point.

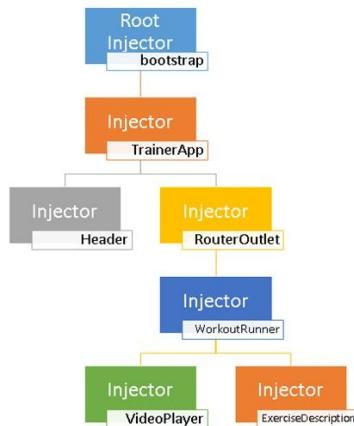
While dependencies registered with the injector are singleton, `Injector` itself is not! At any given point in time, there are multiple injectors active in the application. In fact, injectors are created in the same hierarchy as the component tree. Angular creates an `Injector` instance for every component in the component tree (oversimplification; see the next information box).

Angular does not literally create an injector for each component. As explained in the Angular



developer guide: every component doesn't need its own injector and it would be horribly inefficient to create masses of injectors for no good purpose. But it is true that every component has an injector (even if it shares that injector with another component) and there may be many different injector instances operating at different levels of the component tree. It is useful to pretend that every component has its own injector.

The component and injector tree looks something like this when a workout is running:



The insert textbox denotes the component name. The **root injector** is the injector created as part of the application bootstrap process.

What is the significance of this injector hierarchy? To understand the implications, we need to understand what happens when a component requests a dependency.

Angular DI dependency walk

Whenever requested for a dependency, Angular first tries to satisfy the dependency from the component's own injector. If it fails to find the requested dependency, it queries the parent component injector for the dependency, and its parent if the probing fails again, and so on and so forth till it finds the dependency or reaches the root injector. The takeaway: any dependency search is hierarchy-based.

Earlier when we registered `WorkoutHistoryTrackerService`, it was registered with the root injector. The `WorkoutHistoryTrackerService` dependency request from both `WorkoutRunnerComponent` and `WorkoutHistoryComponent` gets satisfied by the root injector, and not by their own component injectors.

This hierarchical injector structure brings a lot of flexibility. We can configure different providers at different component levels and override the parent provider configuration in child components. This only applies to dependencies registered on components. If the dependency is added to a module, it gets registered on the root injector.

Also, if a dependency is registered at the component level, its life cycle is bound to the component's life cycle. The dependency is created every time the component is loaded, and destroyed when the component is destroyed. Unlike module-level dependencies that are created only once: when requested for the first time.

Let's try to override the global `WorkoutHistoryTrackerService` service in components that use it to learn what happens on such overrides. It's going to be fun and we will learn a lot!

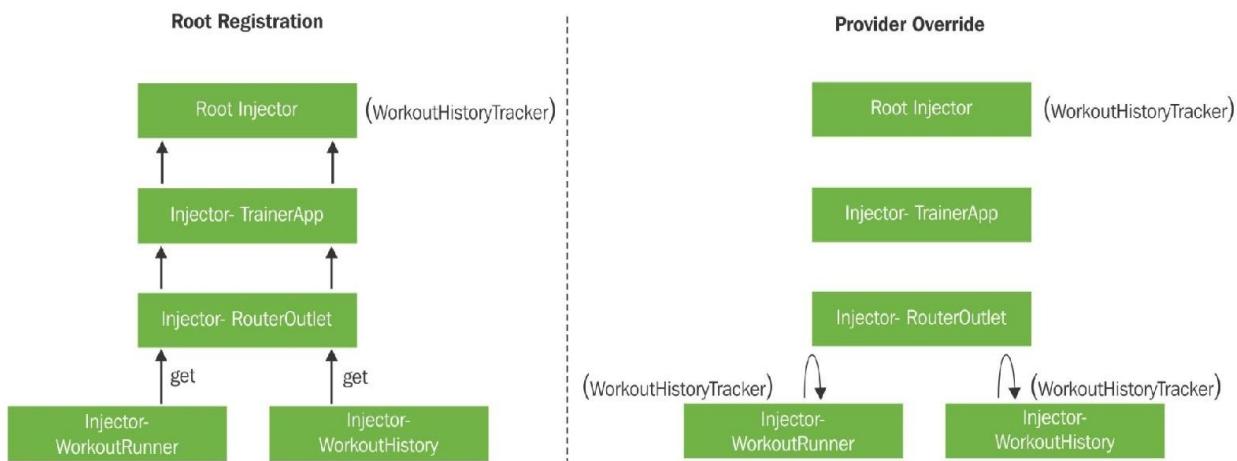
Open `workout-runner.component.ts` and add a `providers` attribute to the `@Component` decorator:

```
| providers: [WorkoutHistoryTrackerService]
```

Do this in `workout-history.component.ts` too. Now if we refresh the app, start a

workout, and then load the history page, the grid is empty. Irrespective of the times we try to run the workout, the history grid is always empty.

The reason is quite obvious. After setting the `WorkoutHistoryTrackerService` provider on each `WorkoutRunnerComponent` and `WorkoutHistoryComponent`, the dependency is being fulfilled by the respective component injectors themselves. Both component injectors create their own instance of `WorkoutHistoryTrackerService` when requested, and hence the history tracking is broken. Look at the following diagram to understand how the request is fulfilled in both scenarios:



A quick question: what happens if we register the dependency in the root component, `TrainerAppComponent`, instead of doing it on the module? Something like this:

```
  @Component({
    selector: 'abe-root',
    providers:[WorkoutHistoryTrackerService]
  }
export class AppComponent {
```

Interestingly, with this setup too, things work perfectly. That's pretty evident; `TrainerAppComponent` is a parent component for `RouterOutlet` that internally loads `WorkoutRunnerComponent` and `WorkoutHistoryComponent`. Hence in such a setup, the dependency gets fulfilled by the `TrainerAppComponent` injector.



Dependency lookup up on the component hierarchy can be manipulated if an intermediate component has declared itself as a host component. We will learn more about it in later chapters.

Hierarchical injectors allow us to register dependencies at a component level, avoiding the need to register all dependencies globally.

The predominant use case for this functionality is when building an Angular library component. Such components can register their own dependencies without requiring the consumer of the library to register library-specific dependencies.



Remember: if you are having trouble loading the right service/dependency, make sure you check the component hierarchy for overrides done at any level.

We now understand how dependency resolution works in components. But what happens if a service has a dependency? Yet more uncharted territory to explore. Let's extend our app further.



Remove any provider registration done on components before continuing further.

Dependency injection with `@Injectable`

`WorkoutHistoryTrackerService` has a fundamental flaw: the history is not persisted. Refresh the app and the history is lost. We need to add persistence logic to store historical data. To avoid any complex setup, we are going to use the browser local storage to store historical data.

Add a new `LocalStorageService` service by invoking this CLI command from the `trainer/src/app/core` folder:

```
| ng generate service local-storage
```

Copy the following two functions to the generated class, or copy them from the `checkpoint3.2` GitHub branch:

```
getItem<T>(key: string): T {
  if (localStorage[key]) {
    return <T>JSON.parse(localStorage[key]);
  }
  return null;
}

setItem(key: string, item: any) {
  localStorage[key] = JSON.stringify(item);
}
```

A simple wrapper over the browser's `localStorage` object.

Like any other dependency, inject it into the `WorkoutHistoryTrackerService` constructor (the `workout-history-tracker.ts` file) with the necessary import:

```
import {LocalStorage} from './local-storage';
...
constructor(private storage: LocalStorageService) {
```

~~It is advisable that the default `Injectable` decorator on the service is applied, even if we register the dependency on module (`NgModule` provider registrations syntax). Especially when the service itself has a dependency, as it does with the preceding example of `WorkoutHistoryTrackerService`. Do not use the `providedIn` decorator attribute of `Injectable` when using the module-based service~~

registration.

By putting in the `@Injectable` decorator, we are forcing the TypeScript transpiler to generate metadata for the `WorkoutHistoryTrackerService` class. This includes details about the constructor arguments. Angular DI consumes this generated metadata to determine the types of dependency the service has, and in future, it fulfills these dependencies when the service is created.

What about `WorkoutRunnerComponent`, which that uses `WorkoutHistoryTrackerService`? We have not used `@Injectable` there but still, the DI works. We don't need to. Any decorator works and there is already an `@Component` decorator applied to all components.

The actual integration between the `LocalStorage` service and `WorkoutHistoryTrackerService` is a mundane process.

Update the constructor for `WorkoutHistoryTrackerService` as follows:

```
| constructor(private storage: LocalStorage) {  
|   this.workoutHistory = (storage.getItem<Array<WorkoutLogEntry>>(this.storageKey) ||  
|   [])  
|     .map((item: WorkoutLogEntry) => {  
|       item.startedOn = new Date(item.startedOn.toString());  
|       item.endedOn = item.endedOn == null ? null : new Date(item.endedOn.toString());  
|       return item;  
|     });  
|}
```

And add a declaration for `storageKey`:

```
| private storageKey = 'workouts';
```

The constructor loads the workout logs from the local storage. The `map` function call is necessary as everything stored in `localStorage` is a string. Therefore, while de-serializing, we need to convert the string back to the date value.

Add this statement last in the `startTracking`, `exerciseComplete`, and `endTracking` functions:

```
| this.storage.setItem(this.storageKey, this.workoutHistory);
```

We save the workout history to local storage every time the historical data changes.

That's it! We have built workout history tracking over `localStorage`. Verify it!

Before we move on to our big-ticket item, audio support, there are a few minor fixes that are needed for a better user experience. The first one is related to the [History link](#).

Tracking route changes using the router service

The History link in the `Header` component is visible for all routes other than when a workout is in progress. We don't want to lose an in-progress workout by accidentally clicking on the History link. Moreover, no one is interested in knowing about the workout history while doing a workout.

The fix is easy. We just need to determine whether the current route is the workout route and hide the link. The `Router` service is going to help us with this job.

Open `header.component.ts` and look at the highlighted implementation:

```
import { Router, NavigationEnd } from '@angular/router';
import 'rxjs/add/operator/filter';
...
export class HeaderComponent {
  private showHistoryLink = true;
  constructor(private router: Router) {
    this.router.events.pipe(
      filter(e => e instanceof NavigationEnd)
      .subscribe((e: NavigationEnd) => {
        this.showHistoryLink = !e.url.startsWith('/workout');
      });
  }
}
```

The `showHistoryLink` property binds to the view and decides whether the history link is shown to the user or not. In the constructor, we inject the `Router` service and subscribe to the observable events using the `subscribe` function.

We will learn more about observables later in the chapter, but for now, it is enough to understand that observables are objects that raise events and can be subscribed to. Since the router raises a number of events throughout the component's life cycle, the `filter` operator allows us to filter the event we are interested in and the `subscribe` function registers a callback function that is invoked every time the route changes.

To learn about the other router events, including `NavigationStart`, `NavigationEnd`, `NavigationCancel`, and `NavigationError`, look at the router documentation (<http://bit.ly/>

`ng-router-events`) to understand when the events are raised.

The callback implementation just toggles the `showHistoryLink` state based on the current route URL. To use `showHistoryLink` in the view, just update the header template line with the anchor tag to:

```
|<li *ngIf="showHistoryLink"><a routerLink="/history" ...>...</a></li>
```

And that's it! The History link does not show up on the workout page.



If you are having a problem with running the code, look at the `checkpoint3.2` Git branch for a working version of what we have done thus far. Or if you are not using Git, download the snapshot of `checkpoint3.2` (a ZIP file) from <http://bit.ly/ng6be-checkpoint-3-2>. Refer to the `README.ma` file in the `trainer` folder when setting up the snapshot for the first time.

Another fix/enhancement is related to the video panel on the workout page.

Fixing the video playback experience

The current video panel implementation can at best be termed 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. This is a clear indication that this approach to video playback needs some fixing.

This is what we are going to do to fix the video panel:

- Show the image thumbnail for the exercise video instead of loading the video player itself
- When the user clicks on the thumbnail, load a popup/dialog with a bigger video player that can play the selected video
- Pause the workout while the video playback is on

Let's get on with the job!

Using thumbnails for video

Replace the `ngFor` HTML template inside `video-player.component.html` with this snippet:

```
<div *ngFor="let video of videos" class="row">
<div class="col-sm-12 p-2">
<img class="video-image" [src]="'//i.ytimg.com/vi/'+video+'/hqdefault.jpg'" />
</div>
</div>
```

We have abandoned iframe, and instead loaded the thumbnail image of the video (check the `img` tag). All other content shown here is for styling the image.



We have referenced the Stack Overflow post (<http://bit.ly/so-yt-thumbnail>) to determine the thumbnail image URL for our videos.

Start a new workout; the images should show up, but the playback functionality is broken. We need to add a video playback dialog.

Using the `ngx-modialog` library

To show the video in a dialog, we are going to incorporate a third-party library, **ngx-modialog**, available on GitHub at <http://bit.ly/ngx-modialog>. Let's install and configure the library.

From the command line (inside the `trainer` folder), run the following command to install the library:

```
| npm i ngx-modialog@5.0.0 --save
```

 Work on an Angular v6-compatible `ngx-modialog` library is in progress (<https://github.com/shlomiassaf/ngx-modialog/issues/426>). To use the version 5 library, which is dependent on older version of RxJS, install the `rxjs-compat` package from command line, `npm i rxjs-compat --save`, before proceeding.

Next import and configure the library in the core module. Open `core.module.ts` and add the highlighted configurations:

```
import { RouterModule } from '@angular/router';
import { ModalModule } from 'ngx-modialog';
import { BootstrapModalModule } from 'ngx-modialog/plugins/bootstrap';
...
imports: [
  ...
  ModalModule.forRoot(),
  BootstrapModalModule
],
```

The library is now ready for use.

While `ngx-modialog` has a number of predefined templates for standard dialogs, such as alert, prompt, and confirm, these dialogs provide little customization in terms of look and feel. To have better control over the dialog UI, we need to create a custom dialog, which thankfully the library supports.

Creating custom dialogs

Custom dialogs in `ngx-modal` are nothing but Angular components with some special library construct incorporated.

Let's start with building a video dialog component that shows the YouTube video in a popup dialog. Generate the component's boilerplate by navigating to `trainer/src/app/workout-runner/video-player` and running the following command:

```
| ng generate component video-dialog -is
```

Copy the video dialog implementation from the `workout-runner/video-player/video-dialog` folder in the `checkpoint3.3` Git branch (GitHub location: <http://bit.ly/ng6be-3-3-video-dialog>) into your local setup. You need to update the component implementation and the view.

Next, update `workout-runner.module.ts` and add a new `entryComponents` attribute to the module decorator:

```
| ...
| declarations: [..., VideoDialogComponent],
| entryComponents:[VideoDialogComponent]
```

The newly created `VideoDialogComponent` needs to be added to `entryComponents` as it is not explicitly used in the component tree.

`VideoDialogComponent` is a standard Angular component, with some modal dialog and specific implementations that we describe later. The `videoDialogContext` class declared inside `VideoDialogComponent` has been created to pass the `videoId` of the YouTube video clicked to the dialog instance. The library uses this context class to pass data between the calling code and the modal dialog. The `VideoDialogContext` class inherits a configuration class that the dialog library uses to alter the behavior and UI of the modal dialog from `BSModalContext`.

To get a better sense of how `VideoDialogContext` is utilized, let's invoke the preceding dialog from the workout runner when the video image is clicked.

Update the `ngFor` div in `video-player.component.html` and add a `click` event handler:

```
|<div *ngFor="let video of videos" (click)="playVideo(video)" ...>
```

The preceding handler invokes the `playVideo` method, passing in the video clicked. The `playVideo` function, in turn, opens the corresponding video dialog. Add the `playVideo` implementation to `video-player.component.ts` as highlighted:

```
import { Modal } from 'ngx-modialog/plugins/bootstrap';
import { VideoDialogComponent, VideoDialogContent } from './video-dialog/video-
dialog.component';
import { overlayConfigFactory } from 'ngx-modialog';
...
export class VideoPlayerComponent {
    @Input() videos: Array<string>;
}

constructor(private modal: Modal) { }

    playVideo(videoId: string) {
        this.modal.open(VideoDialogComponent,
            overlayConfigFactory(new VideoDialogContent(videoId)));
    }
}
```

The `playVideo` function calls the `Modal` class' `open` function, passing in the dialog component to open and a new instance of the `VideoDialogContent` class with the `videoId` of the YouTube video. Before proceeding, delete the `ngOnchange` function and the interface declaration too.

Coming back to the `VideoDialogComponent` implementation, the component implements the `ModalComponent<VideoDialogContent>` interface required by the modal library. Look at how the context (`VideoDialogContent`) to the dialog is passed to the constructor and how we extract and assign the `videoId` property from the context. Then it's just a matter of binding the `videoId` property to the template view (see the HTML template) and rendering the YouTube player.

And we are good to go. Load the app and start the workout. Then click on any workout video images. The video dialog should load and now we can watch the video!

Before we call the dialog implementation complete, there is one small issue that needs to be fixed. When the dialog opens, the workout should pause: that's not happening currently. We will help you fix it at the end of the next section using Angular's eventing infrastructure.

 If you are having a problem with running the code, look at the `checkpoint3.3` Git branch for a working version of what we have done thus far. Or if you are not using Git, download the



snapshot of `checkpoint3.3` (a ZIP file) from <http://bit.ly/ng6be-checkpoint-3-3>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

There is one last feature that we plan to add to *7-Minute Workout* before wrapping up the application and building something new with Angular: audio support. It teaches us some new cross-component communication patterns too.

Cross-component communication using Angular events

We touched upon events in the last chapter when learning about Angular's binding infrastructure. It's time now to look at eventing in more depth. Let's add audio support to *7-Minute Workout*.

Tracking exercise progress with audio

For the *7-Minute Workout* app, adding sound support is vital. One cannot exercise while constantly staring at the screen. Audio clues help the user perform the workout effectively as they can just follow the audio instructions.

Here is how we are going to support exercise tracking using audio clues:

- A ticking clock soundtrack 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

There will be an audio clip for each of these scenarios.

Modern browsers have good support for audio. The HTML5 `<audio>` tag provides a mechanism to embed audio clips into HTML content. We too will use the `<audio>` tag to play back our clips.

Since the plan is to use the HTML `<audio>` element, we need to create a wrapper directive that allows us to control audio elements from Angular. Remember that directives are HTML extensions without a view.



The `checkpoint3.4` Git and the `trainer/static/audio` folder contain all the audio files used for playback; copy them first. If you are not using Git, a snapshot of the chapter code is available at <http://bit.ly/ng6be-checkpoint-3-4>. Download and unzip the contents and copy the audio files.

Building Angular directives to wrap HTML audio

If you have worked a lot with JavaScript and jQuery, you may have realized we have purposefully shied away from directly accessing the DOM for any of our component implementations. There has not been a need to do it. The Angular data-binding infrastructure, including property, attribute, and event binding, has helped us manipulate HTML without touching the DOM.

For the audio element too, the access pattern should be Angularish. In Angular, the only place where direct DOM manipulation is acceptable and practiced is inside directives. Let's create a directive that wraps access to audio elements.

Navigate to `trainer/src/app/shared` and run this command to generate a template directive:

```
| ng generate directive my-audio
```



Since it is the first time we are creating a directive, we encourage you to look at the generated code.

Since the directive is added to the shared module, it needs to be exported too. Add the `MyAudioDirective` reference in the `exports` array too (`shared.module.ts`). Then update the directive definition with the following code:

```
import {Directive, ElementRef} from '@angular/core';

@Directive({
  selector: 'audio',
  exportAs: 'MyAudio'
})
export class MyAudioDirective {
  private audioPlayer: HTMLAudioElement;
  constructor(element: ElementRef) {
    this.audioPlayer = element.nativeElement;
  }
}
```

The `MyAudioDirective` class is decorated with `@Directive`. The `@Directive` decorator is similar to the `@Component` decorator except we cannot have an attached view. Therefore, no `template` or `templateUrl` is allowed!

The preceding `selector` property allows the framework to identify where to apply the directive. We have replaced the generated `[abeMyAudioDirective]` attribute selector with just `audio`. Using `audio` as the selector makes our directive load for every `<audio>` tag used in HTML. The new selector works as an element selector.



In a standard scenario, directive selectors are attribute-based (such as `[abeMyAudioDirective]` for the generated code), which helps us identify where the directive has been applied. We deviate from this norm and use an element selector for the `MyAudioDirective` directive. We want this directive to be loaded for every audio element, and it becomes cumbersome to go to each audio declaration and add a directive-specific attribute. Hence an element selector.

The use of `exportAs` becomes clear when we use this directive in view templates.

The `ElementRef` object injected in the constructor is the Angular element (`audio` in this case) for which the directive is loaded. Angular creates the `ElementRef` instance for every component and directive when it compiles and executes the HTML template. When requested in the constructor, the DI framework locates the corresponding `ElementRef` and injects it. We use `ElementRef` to get hold of the underlying audio element in the code (the instance of `HTMLAudioElement`). The `audioPlayer` property holds this reference.

The directive now needs to expose an API to manipulate the audio player. Add these functions to the `MyAudioDirective` directive:

```
stop() {
  this.audioPlayer.pause();
}

start() {
  this.audioPlayer.play();
}
get currentTime(): number {
  return this.audioPlayer.currentTime;
}

get duration(): number {
  return this.audioPlayer.duration;
}

get playbackComplete() {
  return this.duration == this.currentTime;
}
```

The `MyAudioDirective` API has two functions (`start` and `stop`) and three getters (`currentTime`, `duration`, and a Boolean property called `playbackComplete`). The implementations for these functions and properties just wrap the audio element functions.



Learn about these audio functions from the MDN documentation here: <http://bit.ly/html-media-element>.

To understand how we use the audio directive, let's create a new component that manages audio playback.

Creating WorkoutAudioComponent for audio support

If we go back and look at the audio cues that are required, there are four distinct audio cues, and hence we are going to create a component with five embedded `<audio>` tags (two audio tags work together for next-up audio).

From the command line go to the `trainer/src/app/workout-runner` folder and add a new `WorkoutAudioComponent` component using Angular CLI.

Open `workout-audio.component.html` and replace the existing view template with this HTML snippet:

```
|<audio #ticks="MyAudio" loop src="/assets/audio/tick10s.mp3"></audio>
|<audio #nextUp="MyAudio" src="/assets/audio/nextup.mp3"></audio>
|<audio #nextUpExercise="MyAudio" [src]="/assets/audio/' + nextupSound"></audio>
|<audio #halfway="MyAudio" src="/assets/audio/15seconds.wav"></audio>
|<audio #aboutToComplete="MyAudio" src="/assets/audio/321.wav"></audio>
```

There are five `<audio>` tags, one for each of the following:

- **Ticking audio:** The first audio tag produces the ticking sound and is started as soon as the workout starts.
- **Next up audio and exercise audio:** The next two audio tags work together. The first tag produces the "Next up" sound. And the actual exercise audio is handled by the third tag (in the preceding code snippet).
- **Halfway audio:** The fourth audio tag plays halfway through the exercise.
- **About to complete audio:** The final audio tag plays a piece to denote the completion of an exercise.

Did you notice the usage of the `#` symbol in each of the `audio` tags? There are some variable assignments prefixed with `#`. In the Angular world, these variables are known as **template reference variables** or at times **template variables**.

As the platform guide defines:

A template reference variable is often a reference to a DOM element or directive

within a template.



*Don't confuse them with the template input variables that we have used with the `ngFor` directive earlier, `*ngFor="let video of videos"`. The **template input variable's** (`video` in this case) scope is within the HTML fragment it is declared, whereas the **template reference variable** can be accessed across the entire template.*

Look at the last section where `MyAudioDirective` was defined. The `exportAs` metadata is set to `MyAudio`. We repeat that same `MyAudio` string while assigning the `template reference variable` for each audio tag:

```
| #ticks="MyAudio"
```

The role of `exportAs` is to define the name that can be used in the view to assign this directive to a variable. Remember, a single element/component can have multiple directives applied to it. `exportAs` allows us to select which directive should be assigned to a template-reference variable based on what is on the right side of equals.

Typically, template variables, once declared, give access to the view element/component they are attached to, to other parts of the view, something we will discuss shortly. But in our case, we will use template variables to refer to the multiple `MyAudioDirective` from the parent component's code. Let's understand how to use them.

Update the generated `workout-audio.component.ts` with the following outline:

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { MyAudioDirective } from '../../../../../shared/my-audio.directive';

@Component({
  ...
})
export class WorkoutAudioComponent implements OnInit {
  @ViewChild('ticks') private ticks: MyAudioDirective;
  @ViewChild('nextUp') private nextUp: MyAudioDirective;
  @ViewChild('nextUpExercise') private nextUpExercise: MyAudioDirective;
  @ViewChild('halfway') private halfway: MyAudioDirective;
  @ViewChild('aboutToComplete') private aboutToComplete: MyAudioDirective;
  private nextupSound: string;

  constructor() { }
  ...
}
```

The interesting bit in this outline is the `@viewchild` decorator against the five properties. The `@viewchild` decorator allows us to inject a child

component/directive/element reference into its parent. The parameter passed to the decorator is the template variable name, which helps DI match the element/directive to inject. When Angular instantiates the main `WorkoutAudioComponent`, it injects the corresponding audio directives based on the `@ViewChild` decorator and the template reference variable name passed. Let's complete the basic class implementation before we look at `@ViewChild` in detail.



Without `exportAs` set on the `MyAudioDirective` directive, the `@ViewChild` injection injects the related `ElementRef` instance instead of the `MyAudioDirective` instance. We can confirm this by removing the `exportAs` attribute from `myAudioDirective` and then looking at the injected dependencies in `WorkoutAudioComponent`.

The remaining task is to just play the correct audio component at the right time. Add these functions to `WorkoutAudioComponent`:

```
stop() {
  this.ticks.stop();
  this.nextUp.stop();
  this.halfway.stop();
  this.aboutToComplete.stop();
  this.nextUpExercise.stop();
}
resume() {
  this.ticks.start();
  if (this.nextUp.currentTime > 0 && !this.nextUp.playbackComplete)
    { this.nextUp.start(); }
  else if (this.nextUpExercise.currentTime > 0 &&
!this.nextUpExercise.playbackComplete)
    { this.nextUpExercise.start(); }
  else if (this.halfway.currentTime > 0 && !this.halfway.playbackComplete)
    { this.halfway.start(); }
  else if (this.aboutToComplete.currentTime > 0 &&
!this.aboutToComplete.playbackComplete)
    { this.aboutToComplete.start(); }
}
onExerciseProgress(progress: ExerciseProgressEvent) {
  if (progress.runningFor === Math.floor(progress.exercise.duration / 2)
    && progress.exercise.exercise.name != 'rest') {
    this.halfway.start();
  }
  else if (progress.timeRemaining === 3) {
    this.aboutToComplete.start();
  }
}
onExerciseChanged(state: ExerciseChangedEvent) {
  if (state.current.exercise.name === 'rest') {
    this.nextupSound = state.next.exercise.nameSound;
    setTimeout(() => this.nextUp.start(), 2000);
    setTimeout(() => this.nextUpExercise.start(), 3000);
  }
}
```

Having trouble writing these functions? They are available in the [checkpoint3.3](#) Git

branch.

There are two new model classes used in the preceding code. Add their declarations to `model.ts`, as follows (again available in `checkpoint3.3`):

```
export class ExerciseProgressEvent {  
    constructor(  
        public exercise: ExercisePlan,  
        public runningFor: number,  
        public timeRemaining: number,  
        public workoutTimeRemaining: number) { }  
}  
  
export class ExerciseChangedEvent {  
    constructor(  
        public current: ExercisePlan,  
        public next: ExercisePlan) { }  
}
```

These are model classes to track progress events. The `WorkoutAudioComponent` implementation consumes this data. Remember to import the reference for `ExerciseProgressEvent` and `ExerciseChangedEvent` in `workout-audio.component.ts`.

To reiterate, the audio component consumes the events by defining two event handlers: `onExerciseProgress` and `onExerciseChanged`. How the events are generated becomes clear as we move along.

The `start` and `resume` functions stop and resume audio whenever a workout starts, pauses, or completes. The extra complexity in the `resume` function is to tackle cases when the workout was paused during next up, about to complete, or half-way audio playback. We just want to continue from where we left off.

The `onExerciseProgress` function should be called to report the workout progress. It's used to play the halfway audio and about-to-complete audio based on the state of the workout. The parameter passed to it is an object that contains exercise progress data.

The `onExerciseChanged` function should be called when the exercise changes. The input parameter contains the current and next exercise in line and helps `WorkoutAudioComponent` to decide when to play the next up exercise audio.

We touched upon two new concepts in this section: template reference variables and injecting child elements/directives into the parent. It's worth exploring these two concepts in more detail before we continue with the implementation. We'll

start with learning more about template reference variables.

Understanding template reference variables

Template reference variables are created on the view template and are mostly consumed from the view. As you have already learned, these variables can be identified by the `#` prefix used to declare them.

One of the greatest benefits of template variables is that they facilitate cross-component communication at the view template level. Once declared, such variables can be referenced by sibling elements/components and their children. Check out the following snippet:

```
| <input #emailId type="email">Email to {{emailId.value}}  
|   <button (click)= "MailUser(emailId.value)">Send</button>
```

This snippet declares a template variable, `emailId`, and then references it in the interpolation and the button `click` expression.

The Angular templating engine assigns the DOM object for `input` (an instance of `HTMLInputElement`) to the `emailId` variable. Since the variable is available across siblings, we use it in a button's `click` expression.

Template variables work with components too. We can easily do this:

```
| <trainer-app>  
|   <workout-runner #runner></workout-runner>  
|   <button (click)= "runner.start()">Start Workout</button>  
</trainer-app>
```

In this case, `runner` has a reference to the `WorkoutRunnerComponent` object, and the button is used to start the workout.



The `ref-` prefix is the canonical alternative to `#`. The `#runner` variable can also be declared as `ref-runner`.

Template variable assignment

You may not have noticed but there is something interesting about the template variable assignments described in the last few sections. To recap, the three examples that we have used are: <audio #ticks="MyAudio" loop src="/static/audio/tick10s.mp3"></audio>

```
<input #emailId type="email">Email to {{emailId.value}}  
<workout-runner #runner></workout-runner>
```

What got assigned to the variable depends on where the variable was declared. This is governed by rules in Angular:

- If a directive is present on the element, such as `MyAudioDirective` in the first example shown previously, the directive sets the value. The `MyAudioDirective` directive sets the `ticks` variable to an instance of `MyAudioDirective`.
- If there is no directive present, either the underlying HTML DOM element is assigned or a component object is assigned (as shown in the `input` and `workout-runner` examples).

We will be employing this technique to implement the workout audio component integration with the workout runner component. This introduction gives us the head start that we need.

The other new concept that we promised to cover is child element/directive injection using the `viewchild` and `viewchildren` decorators.

Using the `@ViewChild` decorator

The `@ViewChild` decorator instructs the Angular DI framework to search for some specific child component/directive/element in the component tree and **inject it into the parent**. This allows the parent component to interact with child components/element using the reference to the child, a new communication pattern!

In the preceding code, the `audio` element directive (the `MyAudioDirective` class) is injected into the `workoutAudioComponent` code.

To establish the context, let's recheck a view fragment from `workoutAudioComponent`:

```
| <audio #ticks="MyAudio" loop src="/static/audio/tick10s.mp3"></audio>
```

Angular injects the directive (`MyAudioDirective`) into the `workoutAudioComponent` property: `ticks`. The search is done based on the selector passed to the `@ViewChild` decorator. Let's see the audio example again:

```
| @ViewChild('ticks') private ticks: MyAudioDirective;
```

The selector parameter on `ViewChild` can be a string value, in which case Angular searches for a matching template variable, as before.

Or it can be a type. This is valid and should inject an instance of `MyAudioDirective`:

```
| @ViewChild(MyAudioDirective) private ticks: MyAudioDirective;
```

However, it does not work in our case. Why? Because there are multiple `MyAudioDirective` directives declared in the `workoutAudioComponent` view, one for each of the `<audio>` tags. In such a scenario, the first match is injected. Not very useful. Passing the type selector would have worked if there was only one `<audio>` tag in the view!



*Properties decorated with `@ViewChild` are sure to be set **before** the `ngAfterViewInit` event hook on the component is called. This implies such properties are `null` if accessed inside the constructor.*

Angular also has a decorator to locate and inject multiple child

components/directives: @ViewChildren.

The `@ViewChildren` decorator

`@viewChildren` works similarly to `@ViewChild`, except it can be used to inject multiple child types into the parent. Again taking the previous audio component above as an example, using `@ViewChildren`, we can get all the `MyAudioDirective` directive instances in `WorkoutAudioComponent`, as shown here:

```
| @ViewChildren(MyAudioDirective) allAudios: QueryList<MyAudioDirective>;
```

Look carefully; `allAudios` is not a standard JavaScript array, but a custom class, `QueryList<Type>`. The `QueryList` class is an immutable collection that contains the reference to the components/directives that Angular was able to locate based on the filter criteria passed to the `@viewChildren` decorator. The best thing about this list is that Angular will keep this list in sync with the state of the view. When directives/components get added/removed from the view dynamically, this list is updated too. Components/directives generated using `ng-for` are a prime example of this dynamic behavior. Consider the preceding `@ViewChildren` usage and this view template:

```
| <audio *ngFor="let clip of clips" src="/static/audio/ "+{{clip}}></audio>
```

The number of `MyAudioDirective` directives created by Angular depends upon the number of `clips`. When `@ViewChildren` is used, Angular injects the correct number of `MyAudioDirective` instances into the `allAudios` property and keeps it in sync when items are added or removed from the `clips` array.

While the usage of `@ViewChildren` allows us to get hold of all `MyAudioDirective` directives, it cannot be used to control the playback. You see, we need to get hold of individual `MyAudioDirective` instances as the audio playback timing varies. Hence the distinct `@ViewChild` implementation.

Once we get hold of the `MyAudioDirective` directive attached to each audio element, it is just a matter of playing the audio tracks at the right time.

Integrating WorkoutAudioComponent

While we have componentized the audio playback functionality into `WorkoutAudioComponent`, it is and always will be tightly coupled to the `WorkoutRunnerComponent` implementation. `WorkoutAudioComponent` derives its operational intelligence from `WorkoutRunnerComponent`. Hence the two components need to interact. `WorkoutRunnerComponent` needs to provide the `WorkoutAudioComponent` state change data, including when the workout started, exercise progress, workout stopped, paused, and resumed.

One way to achieve this integration would be to use the currently exposed `WorkoutAudioComponent` API (stop, resume, and other functions) from `WorkoutRunnerComponent`.

Something can be done by injecting `WorkoutAudioComponent` into `WorkoutRunnerComponent`, as we did earlier when we injected `MyAudioDirective` into `WorkoutAudioComponent`.

Declare the `WorkoutAudioComponent` in the `WorkoutRunnerComponent`'s view, such as:

```
| <div class="row pt-4">...</div>
| <abe-workout-audio></abe-workout-audio>
```

Doing so gives us a reference to the `WorkoutAudioComponent` inside the `WorkoutRunnerComponent` implementation:

```
| @ViewChild(WorkoutAudioComponent) workoutAudioPlayer: WorkoutAudioComponent;
```

The `WorkoutAudioComponent` functions can then be invoked from `WorkoutRunnerComponent` from different places in the code. For example, this is how `pause` would change:

```
| pause() {
|   clearInterval(this.exerciseTrackingInterval);
|   this.workoutPaused = true;
|   this.workoutAudioPlayer.stop();
| }
```

And to play the next-up audio, we would need to change parts of the `startExerciseTimeTracking` function:

```
| this.startExercise(next);  
| this.workoutAudioPlayer.onExerciseChanged(new ExerciseChangedEvent(next,  
| this.getNextExercise()));
```

This is a perfectly viable option where `WorkoutAudioComponent` becomes a dumb component controlled by `WorkoutRunnerComponent`. The only problem with this solution is that it adds some noise to the `WorkoutRunnerComponent` implementation. `WorkoutRunnerComponent` now needs to manage audio playback too.

There is an alternative, however.

`WorkoutRunnerComponent` can expose events that are triggered during different times of workout execution, such as workout started, exercise started, and workout paused. The advantage of having `WorkoutRunnerComponent` expose events is that it allows us to integrate other components/directives with `WorkoutRunnerComponent` using the same events. Be it the `WorkoutAudioComponent` or components we create in future.

Exposing WorkoutRunnerComponent events

Till now we have only explored how to consume events. Angular allows us to raise events too. Angular components and directives can expose [custom events using the `EventEmitter` class and the `@Output` decorator.](#)

Add these event declarations to `WorkoutRunnerComponent` at the end of the variable declaration section:

```
workoutPaused: boolean;
@Output() exercisePaused: EventEmitter<number> =
  new EventEmitter<number>();
@Output() exerciseResumed: EventEmitter<number> =
  new EventEmitter<number>();
@Output() exerciseProgress: EventEmitter<ExerciseProgressEvent> =
  new EventEmitter<ExerciseProgressEvent>();
@Output() exerciseChanged: EventEmitter<ExerciseChangedEvent> =
  new EventEmitter<ExerciseChangedEvent>();
@Output() workoutStarted: EventEmitter<WorkoutPlan> =
  new EventEmitter<WorkoutPlan>();
@Output() workoutComplete: EventEmitter<WorkoutPlan> =
  new EventEmitter<WorkoutPlan>();
```

The names of the events are self-explanatory, and within our `WorkoutRunnerComponent` implementation, we need to raise them at the appropriate times.

Remember to add the `ExerciseProgressEvent` and `ExerciseChangeEvent` imports to the `model` already declared on top. And add the `Output` and `EventEmitter` imports to `@angular/core`.

Let's try to understand the role of the `@Output` decorator and the `EventEmitter` class.

The `@Output` decorator

We covered a decent amount of Angular eventing capabilities in [Chapter 2](#), *Building Our First App – 7-Minute Workout*. Specifically, we learned how we can consume any event on a component, directive, or DOM element using the bracketed () syntax. How about raising our own events?

In Angular, we can create and raise our own events, events that signify something noteworthy has happened in our component/directive. Using the `@output` decorator and the `EventEmitter` class, we can define and raise custom events.



*It's also a good time to refresh what we learned about events, by revisiting the Eventing subsection in the Angular event binding infrastructure section from [Chapter 2](#), *Building Our First App – 7-Minute Workout*.*

Remember this: it is through events that components can communicate with the outside world. When we declare:

```
| @Output() exercisePaused: EventEmitter<number> = new EventEmitter<number>();
```

It signifies that `WorkoutRunnerComponent` exposes an event, `exercisePaused` (raised when the workout is paused).

To subscribe to this event, we can do the following:

```
| <abe-workout-runner (exercisePaused)="onExercisePaused($event)"></abe-workout-runner>
```

This looks absolutely similar to how we did the DOM event subscription in the workout runner template. See this sample stripped from the `workout-runner`'s view:

```
| <div id="pause-overlay" (click)="pauseResumeToggle()"  
| (window:keyup)="onKeyPressed($event)">
```

The `@Output` decorator instructs Angular to make this event available for template binding. Events created without the `@Output` decorator cannot be referenced in HTML.



The `@output` decorator can also take a parameter, signifying the name of the event. If not provided, the decorator uses the property name: `@output("workoutPaused") exercisePaused: EventEmitter<number> ...`. This declares a `workoutPaused` event instead of `exercisePaused`.

Like any decorator, the `@output` decorator is there just to provide metadata for the Angular framework to work with. The real heavy lifting is done by the `EventEmitter` class.

Eventing with EventEmitter

Angular embraces **reactive programming** (also dubbed **Rx-style** programming) to support asynchronous operations with events. If you are hearing this term for the first time or don't have much idea about what reactive programming is, you're not alone.

Reactive programming is all about programming against **asynchronous data streams**. Such a stream is nothing but a sequence of ongoing events ordered based on the time they occur. We can imagine a stream as a pipe generating data (in some manner) and pushing it to one or more subscribers. Since these events are captured asynchronously by subscribers, they are called asynchronous data streams.

The data can be anything, ranging from browser/DOM element events to user input to loading remote data using AJAX. With Rx style, we consume this data uniformly.

In the Rx world, there are Observers and Observables, a concept derived from the very popular **Observer design pattern**. **Observables** are streams that emit data. **Observers**, on the other hand, subscribe to these events.

The `EventEmitter` class in Angular is primarily responsible for providing eventing support. It acts both as an *observer* and *observable*. We can fire events on it and it can also listen to events.

There are two functions available on `EventEmitter` that are of interest to us:

- `emit`: As the name suggests, use this function to raise events. It takes a single argument that is the event data. `emit` is the observable side.
- `subscribe`: Use this function to subscribe to the events raised by `EventEmitter`. `subscribe` is the observer side.

Let's do some event publishing and subscriptions to understand how the preceding functions work.

Raising events from WorkoutRunnerComponent

Look at the `EventEmitter` declaration. These have been declared with the `type` parameter. The `type` parameter on `EventEmitter` signifies the type of data emitted.

Let's add the event implementation to `workout-runner.component.ts`, starting from the top of the file and moving down.

Add this statement to the end of the `start` function:

```
| this.workoutStarted.emit(this.workoutPlan);
```

We use the `emit` function of `EventEmitter` to raise a `workoutStarted` event with the current workout plan as an argument.

To `pause`, add this line to raise the `exercisePaused` event:

```
| this.exercisePaused.emit(this.currentExerciseIndex);
```

To `resume`, add the following line:

```
| this.exerciseResumed.emit(this.currentExerciseIndex);
```

Each time, we pass the current exercise index as an argument to `emit` when raising the `exercisePaused` and `exerciseResumed` events.

Inside the `startExerciseTimeTracking` function, add the highlighted code after the call to `startExercise`:

```
| this.startExercise(next);
| this.exerciseChanged.emit(new ExerciseChangedEvent(next, this.getNextExercise()));
```

The argument passed contains the exercise that is going to start (`next`) and the next exercise in line (`this.getNextExercise()`).

To the same function, add the highlighted code:

```
| this.tracker.endTracking(true);
```

```
| this.workoutComplete.emit(this.workoutPlan);  
| this.router.navigate(['finish']);
```

The event is raised when the workout is completed.

In the same function, we raise an event that communicates the workout progress.
Add this statement:

```
--this.workoutTimeRemaining;  
this.exerciseProgress.emit(new ExerciseProgressEvent(  
    this.currentExercise,  
    this.exerciseRunningDuration,  
    this.currentExercise.duration -this.exerciseRunningDuration,  
    this.workoutTimeRemaining));
```

That completes our eventing implementation.

As you may have guessed, `WorkoutAudioComponent` now needs to **consume** these events. The challenge here is how to organize these components so that they can communicate with each other with the minimum dependency on each other.

Component communication patterns

As the implementation stands now, we have:

- A basic `WorkoutAudioComponent` implementation
- Augmented `WorkoutRunnerComponent` by exposing workout life cycle events

These two components just need to talk to each other now.

If the parent needs to communicate **with its children**, it can do this by:

- **Property binding:** The parent component can set up a property binding on the child component to push data to the child component. For example, this property binding can stop the audio player when the workout is paused:

```
| <workout-audio [stopped]="workoutPaused"></workout-audio>
```

Property binding, in this case, works fine. When the workout is paused, the audio is stopped too. But not all scenarios can be handled using property bindings. Playing the next exercise audio or halfway audio requires a bit more control.

- **Calling functions on child components:** The parent component can also call functions on the child component if it can get hold of the child component. We have already seen how to achieve this using the `@ViewChild` and `@ViewChildren` decorators in the `WorkoutAudioComponent` implementation. This approach and its shortcomings have also been discussed briefly in the *Integrating WorkoutAudioComponent* section.

There is one more not-so-good option. Instead of the parent referencing the child component, the child references the parent component. This allows the child component to call the parent component's public functions or subscribe to parent component events.

We are going to try this approach and then scrap the implementation for a better one! A lot of learning can be derived from the not-so-optimal solution we plan to implement.

Injecting a parent component into a child component

Add the `WorkoutAudioComponent` to the `WorkoutRunnerComponent` view just before the last closing `div`:

```
| <abe-workout-audio></abe-workout-audio>
```

Next, inject `WorkoutRunnerComponent` into `WorkoutAudioComponent`. Open `workout-audio.component.ts` and add the following declaration and update the constructor:

```
private subscriptions: Array<any>;  
  
constructor( @Inject(forwardRef(() => WorkoutRunnerComponent))  
    private runner: WorkoutRunnerComponent) {  
    this.subscriptions = [  
        this.runner.exercisePaused.subscribe((exercise: ExercisePlan) =>  
            this.stop()),  
        this.runner.workoutComplete.subscribe((exercise: ExercisePlan) =>  
            this.stop()),  
        this.runner.exerciseResumed.subscribe((exercise: ExercisePlan) =>  
            this.resume()),  
        this.runner.exerciseProgress.subscribe((progress: ExerciseProgressEvent) =>  
            this.onExerciseProgress(progress)),  
  
        this.runner.exerciseChanged.subscribe((state: ExerciseChangedEvent) =>  
            this.onExerciseChanged(state))];  
}
```

And remember to add these imports:

```
import {Component, ViewChild, Inject, forwardRef} from '@angular/core';  
import {WorkoutRunnerComponent} from '../workout-runner.component'
```

Let's try to understand what we have done before running the app. There is some amount of trickery involved in the construction injection. If we directly try to inject `WorkoutRunnerComponent` into `WorkoutAudioComponent`, it fails with Angular complaining of not being able to find all the dependencies. Read the code and think carefully; there is a subtle dependency cycle issue lurking.

`WorkoutRunnerComponent` is already dependent on `WorkoutAudioComponent`, as we have referenced `WorkoutAudioComponent` in the `WorkoutRunnerComponent` view. Now by injecting `WorkoutRunnerComponent` in `WorkoutAudioComponent`, we have created a dependency cycle.

Cyclic dependencies are challenging for any DI framework. When creating a component with a cyclic dependency, the framework has to somehow resolve the cycle. In the preceding example, we resolve the circular dependency issue by using an `@Inject` decorator and passing in the token created using the `forwardRef()` global framework function.

Once the injection is done correctly, inside the constructor, we attach a handler to the `workoutRunnerComponent` events, using the `subscribe` function of `EventEmitter`. The arrow function passed to `subscribe` is called whenever the event occurs with a specific event argument. We collect all the subscriptions into a `subscription` array. This array comes in handy when we unsubscribe, which we need to, to avoid memory leaks.

A bit about `EventEmitter`: the `EventEmitter` subscription (`subscribe` function) takes three arguments:

```
| subscribe(generatorOrNext?: any, error?: any, complete?: any) : any
```

- The first argument is a callback, which is invoked whenever an event is emitted
- The second argument is an error callback function, invoked when the observable (the part that is generating events) errors out
- The final argument takes a callback function that is called when the observable is done publishing events

We have done enough to make audio integration work. Run the app and start the workout. Except for the ticking audio, all the \ audio clips play at the right time. You may have to wait some time to hear the other audio clips. What is the problem?

As it turns out, we never started the ticking audio clip at the start of the workout. We can fix it by either setting the `autoplay` attribute on the `ticks` audio element or using the component life cycle events to trigger the ticking sound. Let's take the second approach.

Using component life cycle events

The injected `MyAudioDirective` in `WorkoutAudioComponent`, shown as follows, is not available till the view is initialized:

```
| <audio #ticks="MyAudio" loop src="/assets/audio/tick10s.mp3"></audio>
| <audio #nextUp="MyAudio" src="/assets/audio/nextup.mp3"></audio>
| ...
| ...
```

We can verify it by accessing the `ticks` variable inside the constructor; it will be null. Angular has still not done its magic and we need to wait for the children of `WorkoutAudioComponent` to be initialized.

The component's life cycle hooks can help us here. The `AfterViewInit` event hook is called once the component's view has been initialized and hence is a safe place from which to access the component's child directives/elements. Let's do it quickly.

Update `WorkoutAudioComponent` by adding the interface implementation, and the necessary imports, as highlighted:

```
import {..., AfterViewInit} from '@angular/core';
...
export class WorkoutAudioComponent implements OnInit, AfterViewInit {
  ngAfterViewInit() {
    this.ticks.start();
  }
}
```

Go ahead and test the app. The app has come to life with full-fledged audio feedback. Nice!

While everything looks fine and dandy on the surface, there is a memory leak in the application now. If, in the middle of the workout, we navigate away from the workout page (to the start or finish page) and again return to the workout page, multiple audio clips play at random times.

It seems that `WorkoutRunnerComponent` is not getting destroyed on route navigation, and due to this, none of the child components are destroyed, including `WorkoutAudioComponent`. The net result? A new `WorkoutRunnerComponent` is being created every time we navigate to the workout page but is never removed from the

memory on navigating away.

The primary reason for this memory leak is the event handlers we have added in `workoutAudioComponent`. We need to unsubscribe from these events when the audio component unloads, or else the `workoutRunnerComponent` reference will never be dereferenced.

Another component lifecycle event comes to our rescue here: `onDestroy`. Add this implementation to the `workoutAudioComponent` class:

```
ngOnDestroy() {  
    this.subscriptions.forEach((s) => s.unsubscribe());  
}
```

Also, remember to add references to the `onDestroy` event interface as we did for `AfterViewInit`.

~~Hope the `subscription` array that we created during event subscription makes sense now. One-shot unsubscribe!~~

This audio integration is now complete. While this approach is not an awfully bad way of integrating the two components, we can do better. Child components referring to the parent component seems to be undesirable.



Before proceeding, delete the code that we have added to `workout-audio.component.ts` from the Injecting a parent component into a child component section onward.

Sibling component interaction using events and template variables

What if `WorkoutRunnerComponent` and `WorkoutAudioComponent` were organized as sibling components?

If `WorkoutAudioComponent` and `WorkoutRunnerComponent` become siblings, we can make good use of Angular's *eventing* and *template reference variables*. Confused? Well, to start with, this is how the components should be laid out:

```
| <workout-runner></workout-runner>
  <workout-audio></workout-audio>
```

Does it ring any bells? Starting from this template, can you guess how the final HTML template would look? Think about it before you proceed further.

Still struggling? As soon as we make them sibling components, the power of the Angular templating engine comes to the fore. The following template code is enough to integrate `WorkoutRunnerComponent` and `WorkoutAudioComponent`:

```
<abe-workout-runner (exercisePaused)="wa.stop()"
  (exerciseResumed)="wa.resume()"
  (exerciseProgress)="wa.onExerciseProgress($event)"
  (exerciseChanged)="wa.onExerciseChanged($event)"
  (workoutComplete)="wa.stop()"
  (workoutStarted)="wa.resume()">
</abe-workout-runner>
<abe-workout-audio #wa></abe-workout-audio>
```

The `WorkoutAudioComponent` template variable, `wa`, is being manipulated by referencing the variable in the event handler expressions on `WorkoutRunnerComponent`. Quite elegant! We still need to solve the biggest puzzle in this approach: Where does the preceding code go? Remember, `WorkoutRunnerComponent` is loaded as part of route loading. Nowhere in the code have we had a statement like this:

```
| <workout-runner></workout-runner>
```

We need to reorganize the component tree and bring in a container component that can host `WorkoutRunnerComponent` and `WorkoutAudioComponent`. The router then loads this container component instead of `WorkoutRunnerComponent`. Let's do it.

Generate a new component code from command line by navigating to `trainer/src/app/workout-runner` and executing:

```
| ng generate component workout-container -is
```

Copy the HTML code with the events described to the template file. The workout container component is ready.

We just need to rewire the routing setup. Open `app-routing.module.ts`. Change the route for the workout runner and add the necessary import:

```
import {WorkoutContainerComponent}  
from './workout-runner/workout-container/workout-container.component';  
  
{ path: '/workout', component: WorkoutContainerComponent },
```

And we have a working audio integration that is clear, concise, and pleasing to the eye!

It's time now to wrap up the chapter, but not before addressing the video player dialog glitch introduced in the earlier sections. The workout does not stop/pause when the video player dialog is open.

We are not going to detail the fix here, and urge the readers to give it a try without consulting the `checkpoint3.4` code.

Here is an obvious hint. Use the eventing infrastructure!

And another one: raise events from `VideoPlayerComponent`, one for each playback started and ended.

And one last hint: the `open` function on the dialog service (`Modal`) returns a promise, which is resolved when the dialog is closed.



If you are having a problem with running the code, look at the `checkpoint3.4` Git branch for a working version of what we have done thus far. Or if you are not using Git, download the snapshot of `checkpoint3.4` (a ZIP file) from <http://bit.ly/ng6be-checkpoint-3-4>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Summary

Bit by bit, piece by piece, we have added 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 the chapter by exploring the Single Page Application (SPA) capabilities of Angular. Here we learned about basic Angular routing, setting up routes, using route configuration, generating links using the `RouterLink` directive, and using the Angular `Router` and `Location` services to perform navigation.

From the app perspective, we added start, finish, and workout pages to *7-Minute Workout*.

We then built a workout history tracker service that was used to track historical workout executions. During this process, we learned about Angular's **Dependency Injection (DI)** in depth. We covered how a dependency is registered, what a dependency token is, and how dependencies are singleton in nature. We also learned about injectors and how hierarchical injectors affect dependency probing.

Lastly, we touched upon an important topic: cross-component communication, primarily using Angular eventing. We detailed how to create custom events using the `@output` decorator and `EventEmitter`.

The `@ViewChild` and `@ViewChildren` decorators that we touched upon in this chapter helped us understand how a parent can get hold of a child component for use. Angular DI also allows injecting a parent component into a child.

We concluded this chapter by building a `WorkoutAudioComponent` and highlighted how sibling-component communication can happen using Angular events and template variables.

What's next? We are going to build a new app, Personal Trainer. This app will allow us to build our own custom workouts. Once we can create 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 chapter, we'll showcase Angular's form capabilities while we build a UI that allows us to create, update, and view our own custom workouts/exercises.

10-Oct-2018

Personal Trainer

The *7 Minute Workout* app has been an excellent opportunity for us to learn about Angular. Working through the app, we have covered a number of Angular constructs. Still, there are areas such as Angular form support and client-server communication that remain unexplored. This is partially due to the fact that *7 Minute Workout*, from a functional standpoint, has limited touch points with the end user. Interactions are limited to starting, stopping, and pausing the workout. Also, the app neither consumes nor produces any data (except workout history).

In this chapter, we plan to delve deeper into one of the two aforementioned areas, Angular 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 we already have.

This chapter is dedicated to understanding Angular forms and how to put them to use as we build out our *Personal Trainer* app.

The topics that we will cover in this chapter are as follows:

- **Defining Personal Trainer requirements:** Since we are building a new app in this chapter, 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 Angular routes and the main view.
- **Adding support pages:** Before we focus on the form capability and build a Workout component, we build some supporting components for workout and exercise listing.
- **Defining the Workout Builder component structure:** We lay out the

Workout Builder components that we will use to manage workouts.

- **Building forms:** We make extensive use of HTML forms and input elements to create custom workouts. In the process, we will learn more about Angular Forms. The concepts that we cover include:

- **Form types:** The two types of form that can be built with Angular are template-driven and reactive. We're working with both template-driven and reactive forms in this chapter.
- **ngModel:** This provides two-way data binding for template driven forms and allows us to track changes and validate form input.
- **Reactive Form Controls:** These include the form builder, form control, form group, and form array. These are used to construct forms programmatically.
- **Data formatting:** These are the CSS classes that permit us to style our feedback to the user.
- **Input validation:** We will learn about the validation capabilities of Angular forms.

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 chapters, 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 fulfill:

- 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, so 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 that it is in the directory for `workout-runner`. This means that in order to use it, we will have to import it from that directory. It makes more sense to move the model into the `core` folder so that it is clear that it can be used across features. We'll do that in this chapter.

Getting started with the code for Personal Trainer

First, download the base version of the new *Personal Trainer* app from `checkpoint4.1` in the GitHub repository for the book.



The code is available on GitHub <https://github.com/chandermani/angular6byexample> for everyone to download. Checkpoints are implemented as branches in GitHub. The branch to download is as follows: GitHub Branch: `checkpoint4.1`. If you are not using Git, download the snapshot of Checkpoint 4.1 (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/archive/checkpoint4.1.zip>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

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` feature. This feature contains implementations pertaining to *Personal Trainer*.
- Updating the layout and styles of the app.
- Adding some components and HTML templates with placeholder content for *Personal Trainer* in the `workout-builder` folder under `trainer/src/app`.
- Defining a new route to the `WorkoutBuilder` feature. We will cover setting up this route within the app in the coming section.
- As we just mentioned, moving the existing `model.ts` file into the `core` folder.

Let's discuss how we will be using the model.

Using the Personal Trainer model in Workout Builder services

In the last chapter, we dedicated a complete section to learning about Angular services, and one thing we found out was that services are useful for sharing data across controllers and other Angular constructs. Open the `model.ts` file present in the `core` folder under `app`. In this class, we essentially do not have any data, but a blueprint that describes the shape of the data. The plan is to use services to expose this model structure. We have already done that in Workout Runner. Now, we will do the same in Workout Builder.



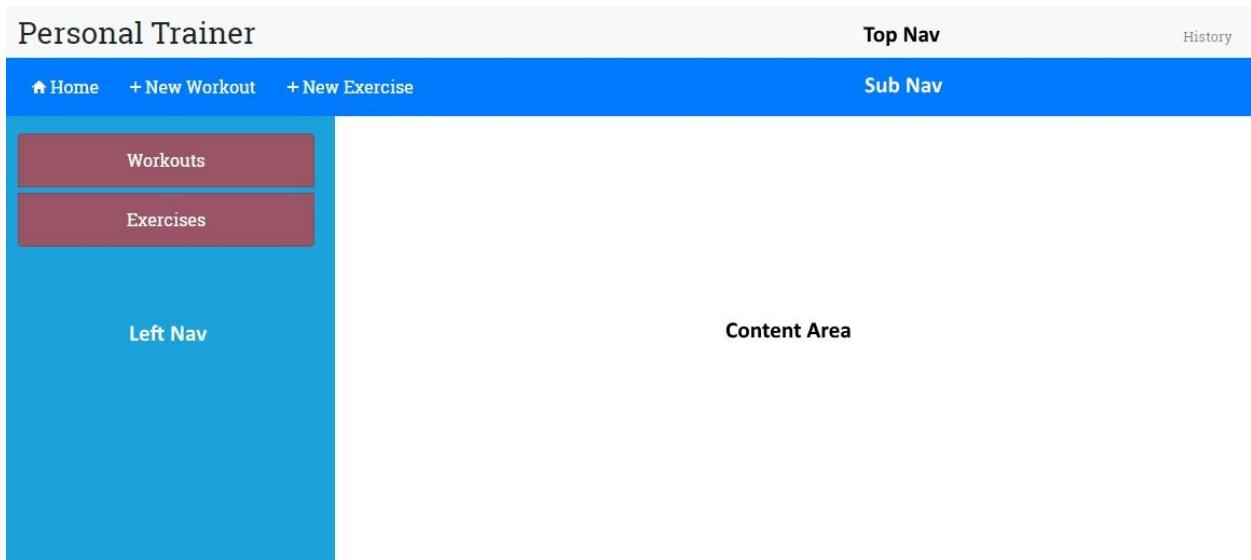
The `model.ts` file has been moved into the `core` folder as it is shared across the Workout Builder and Workout Runner apps. Note: in `checkpoint4.1` we have updated the import statements in `workout-runner.component.ts`, `workout-audio.component.ts`, and `workout-history-tracker-service.ts` to reflect this change.

In [Chapter 2](#), *Building Our First App - 7 Minute Workout*, we reviewed the class definitions in the model file: `Exercise`, `ExercisePlan`, and `workoutPlan`. As we then mentioned, these three classes constitute our base model. We will now start using this base model in our new app.

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 component.
- Left Nav: This contains elements that are dependent upon the active component.
- Content Area: This is where the main view for our component will display. 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 `workout-builder` under `trainer/src/app`. It has files for each component that we described previously, with some placeholder content. We will be building these components as we go along in this chapter.

However, we first need to link up these components within the app. This requires us to define the navigation patterns for the Workout Builder app and accordingly define the app routes.

Personal Trainer navigation with routes

The navigation pattern that we plan to use for the app is the **list-detail pattern**. We will create list pages for the exercises and workouts available in the app. Clicking on any list item takes us to the detailed view for the item where we can perform all CRUD operations (create/read/update/delete). The following routes adhere to this pattern:

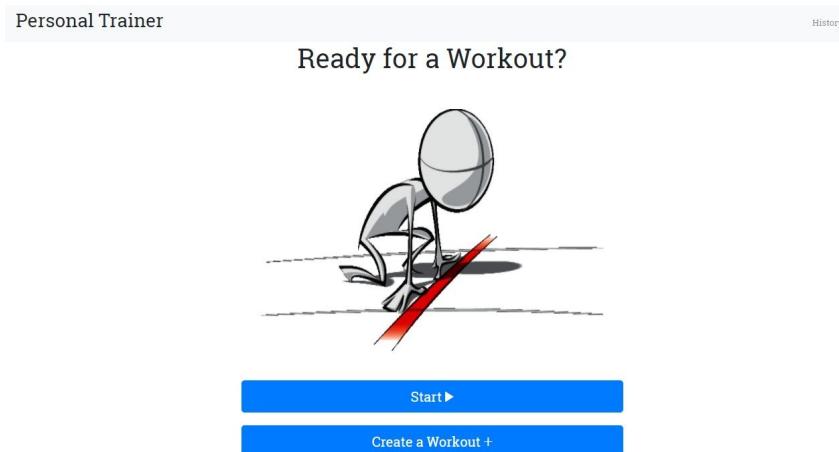
Route	Description
/builder	This just redirects to builder/workouts
/builder/workouts	This lists all the available workouts. It is <u>the landing page for <i>Workout Builder</i></u>
/builder/workout/new	This creates a new workout
/builder/workout/:id	This edits an existing workout with the specific ID
/builder/exercises	This lists all the available exercises
/builder/exercise/new	This creates a new exercise
/builder/exercise/:id	This edits an existing exercise with the specific ID

Getting started with Personal Trainer navigation

At this point, if you look at the route configuration in `app-routing.module.ts` in the `src/app` folder, you will find one new route definition, `builder`:

```
const routes: Routes = [
  ...
  { path: 'builder', component: WorkoutBuilderComponent },
  ...
];
```

And if you run the application, you will see that the start screen shows another link, Create a Workout:



Behind the scenes, we have added another router link for this link into `start.component.html`:

```
<a routerLink="/builder" class="btn btn-primary btn-lg btn-block" role="button" aria-
pressed="true">
  <span>Create a Workout</span>
  <span class="ion-md-add"></span>
</a>
```

And if you click on this link, you will be taken to the following view:

Workout Builder

Again, behind the scenes we have added `workout-builder.component.ts` to the `trainer/src/app/workout-builder` folder with the following inline template:

```
template: `<div class="row">
  <div class="col-sm-3"></div>
  <div class="col-sm-6">
    <h1 class="text-center">Workout Builder</h1>
  </div>
  <div class="col-sm-3"></div>
</div>`
```

And this view is displayed on the screen under the header using the router outlet in our `app.component.html` template:

```
<div class="container body-content app-container">
  <router-outlet></router-outlet>
</div>`
```

We have wrapped this component (along with the other files we have stubbed out for this feature) in a new module named `workout-builder.module.ts`:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { WorkoutBuilderComponent } from './workout-builder.component';
import { ExerciseComponent } from './exercise/exercise.component';
import { ExercisesComponent } from './exercises/exercises.component';
import { WorkoutComponent } from './workout/workout.component';
import { WorkoutsComponent } from './workouts/workouts.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [WorkoutBuilderComponent, ExerciseComponent, ExercisesComponent,
    WorkoutComponent, WorkoutsComponent]
})
export class WorkoutBuilderModule { }
```

The only thing that might look different here from the other modules that we have created is that we are importing `CommonModule` instead of `BrowserModule`. This avoids importing the whole of `BrowserModule` a second time, which would generate

an error when we get to implementing lazy loading for this module.

Finally, we have added an import for this module to `app.module.ts`:

```
...
@NgModule({
  imports: [
    ...
    WorkoutBuilderModule,
    ...
  ]
})
```

So, nothing surprising here. These are the basic component building and routing patterns that we introduced in the previous chapters. Following these patterns, we should now begin to think about adding the additional navigation outlined previously for our new feature. However, before we jump into doing that, there are a couple of things we need to consider.

First, if we start adding our routes to the `app.routing-module.ts` file, then the number of routes stored there will grow. These new routes for *Workout Builder* will also be intermixed with the routes for *Workout Runner*. While the number of routes we are now adding might seem insignificant, over time this could get to be a maintenance problem.

Second, we need to take into consideration that our application now consists of two features—*Workout Runner* and *Workout Builder*. We should be thinking about ways to separate these features within our application so that they can be developed independently of each other.

Put differently, we want **loose coupling** between the features that we build. Using this pattern allows us to swap out a feature within our application without affecting the other features. For example, somewhere down the line we may want to convert the *Workout Runner* into a mobile app but leave the *Workout Builder* intact as a web-based application.

Going back to the first chapter, we emphasized that this ability to separate our components from each other is one of the key advantages of using the **component design pattern** that Angular implements. Fortunately, Angular's router gives us the ability to separate out our routing into logically organized **routing configurations** that closely match the features in our application.

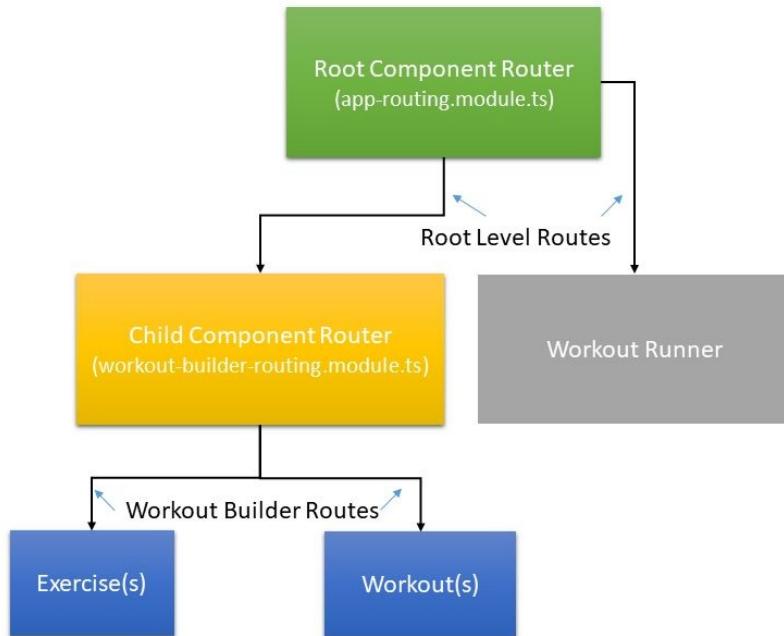
In order to accomplish this separation, Angular allows us to use child routing,

where we can isolate the routing for each of our features. In this chapter, we will use **child routing** to separate out the routing for *Workout Builder*.

Introducing child routes to Workout Builder

Angular supports our goal of isolating the routing for our new *Workout Builder* by providing us with the ability to create a hierarchy of router components within our application. We currently have just one router component, which is in the root component of our application. But Angular allows us to add what are called **child router components** under our root component. This means that one feature can be ignorant of the routes the other is using and each is free to adapt its routes in response to changes within that feature.

Getting back to our application, we can use **child routing** in Angular to match the routing for the two features of our application with the code that will be using them. So in our application, we can structure the routing into the following routing hierarchy for our *Workout Builder* (at this point, we are leaving the *Workout Runner* as is to show the before and after comparison):



With this approach, we can create a logical separation of our routes by feature and make them easier to manage and maintain.

So, let's get started by adding child routing to our application.

From this point on in this section, we'll be adding to the code that we downloaded earlier for this chapter. If you want to see the complete code for this next section, you can download it from `checkpoint 4.2` in the GitHub repository. If you want to work along with us as we build out the code for this section, still be sure to add the changes in `styles.css` in the `trainer/src` folder that are part of this checkpoint, since we won't be discussing them here. Also be sure and add the files for exercise(s), workout(s), and navigation from the `trainer/src/app/workout-builder` folder in the repository. At this stage, these are just stub files, which we will implement later in this chapter. However, you will need these stub files here in order to implement navigation for the Workout Builder module. The code is available for everyone to download on GitHub at <https://github.com/chandermani/angular6byexample>. Checkpoints are implemented as branches in GitHub. The branch to download is as follows: GitHub Branch: `checkpoint4.2`. If you are not using Git, download the snapshot of `checkpoint 4.2` (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/archive/checkpoint4.2.zip>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.



Adding the child routing component

In the `workout-builder` directory, add a new TypeScript file named `workout-builder.routing.module.ts` with the following imports:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { WorkoutBuilderComponent } from './workout-builder.component';
import { WorkoutsComponent } from './workouts/workouts.component';
import { WorkoutComponent } from './workout/workout.component';
import { ExercisesComponent } from './exercises/exercises.component';
import { ExerciseComponent } from './exercise/exercise.component';
```

As you can see, we are importing the components we just mentioned; they will be part of our *Workout Builder* (exercise, exercises, workout, and workouts). Along with those imports, we are also importing `NgModule` from the Angular core module and `Routes` and `RouterModule` from the Angular router module. These imports will give us the ability to add and export child routes.

 *We are not using the Angular CLI here because it does not have a standalone blueprint for creating a routing module. However, you can have the CLI create a routing module at the time that you create a module using the `--routing` option. In this case, we already had an existing module created so we couldn't use that flag. See <https://github.com/angular/angular-cli/blob/master/docs/documentation/stories/routing.md> for more details about how to do this.*

Then, add the following route configuration to the file:

```
const routes: Routes = [
  {
    path: 'builder',
    component: WorkoutBuilderComponent,
    children: [
      {path: '', pathMatch: 'full', redirectTo: 'workouts'},
      {path: 'workouts', component: WorkoutsComponent },
      {path: 'workout/new', component: WorkoutComponent },
      {path: 'workout/:id', component: WorkoutComponent },
      {path: 'exercises', component: ExercisesComponent},
      {path: 'exercise/new', component: ExerciseComponent },
      {path: 'exercise/:id', component: ExerciseComponent }
    ],
  },
];
```

The first configuration, `path: 'builder'`, sets the base URL for the child routes so that each of the child routes prepends it. The next configuration identifies the `WorkoutBuilder` component as the feature area root component for the child

components in this file. This means it will be the component in which each of the child components is displayed using `router-outlet`. The final configuration is a list of one or more children that defines the routing for the child components.

One thing to note here is that we have set up `workouts` as the default for the child routes with the following configuration:

```
| {path: '', pathMatch: 'full', redirectTo: 'workouts'},
```

This configuration indicates that if someone navigates to `builder`, they will be redirected to the `builder/workouts` route. The `pathMatch: 'full'` setting means that the match will only be made if the path after `workout/builder` is an empty string. This prevents the redirection from happening if the routes are something else, such as `workout/builder/exercises` or any of the other routes we have configured within this file.

Finally, add the following class declaration preceded by an `@NgModule` decorator that defines imports and exports for our module:

```
| @NgModule({
|   imports: [RouterModule.forChild(routes)],
|   exports: [RouterModule]
| })
| export class WorkoutBuilderRoutingModule { }
```

This import is very similar to the one in `app.routing-module.ts`, with one difference: instead of `RouterModule.forRoot`, we are using `RouterModule.forChild`. The reason for the difference may seem self-explanatory: we are creating child routes, not the routes in the root of the application, and this is how we signify that. Under the hood, however, there is a significant difference. This is because we cannot have more than one router service active in our application. `forRoot` creates the router service but `forChild` does not.

Updating the WorkoutBuilder component

We next need to update the `WorkoutBuilder` component to support our new child routes. To do so, change the `@Component` decorator for `WorkoutBuilder` to:

1. Remove the `selector`
2. Add a `<abe-sub-nav-main>` custom element to the template
3. Add a `<router-outlet>` tag to the template

The decorator should now look like the following:

```
@Component({
  template: `<div class="container-fluid fixed-top mt-5">
    <div class="row mt-5">
      <abe-sub-nav-main></abe-sub-nav-main>
    </div>
    <div class="row mt-2">
      <div class="col-sm-12">
        <router-outlet></router-outlet>
      </div>
    </div>
  </div>`})
})
```

We are removing the selector because `WorkoutBuilderComponent` will not be embedded in the application root, `app.component.ts`. Instead, it will be reached from `app.routing-module.ts` through routing. And while it will handle incoming routing requests from `app.routes.ts`, it will in turn be routing them to the other components contained in the `WorkoutBuilder` feature.

And those components will display their views using the `<router-outlet>` tag that we have just added to the `WorkoutBuilder` template. Given that the template for `WorkoutBuilderComponent` will be simple, we are using an inline `template` instead of a `templateUrl`.



Typically, for a component's view we recommend using a `templateUrl` that points to a separate HTML template file. This is especially true when you anticipate that the view will involve more than a few lines of HTML. In that situation, it is much easier to work with a view inside its own HTML file.

We are also adding an `<abe-sub-nav-main>` element that will be used to create a secondary top-level menu for navigating within the *Workout Builder* feature. We'll discuss that a little later in this chapter.

Updating the Workout Builder module

Now, let's update `workoutBuilderModule`. First, add the following import to the file:

```
import { WorkoutBuilderRoutingModule } from './workout-builder-routing.module';
```

It imports the child routing that we just set up.

Next, update the `@NgModule` decorator to add `workoutBuilderRoutingModule: ...`

```
@NgModule({
  imports: [
    CommonModule,
    WorkoutBuilderRoutingModule
  ],
  ...
})
```

Finally, add the imports and declarations for the new navigation components that can be found in `checkpoint4.2`:

```
import { LeftNavExercisesComponent } from './navigation/left-nav-exercises.component';
import { LeftNavMainComponent } from './navigation/left-nav-main.component';
import { SubNavMainComponent } from './navigation/sub-nav-main.component';
...
declarations: [
  ...
  LeftNavExercisesComponent,
  LeftNavMainComponent,
  SubNavMainComponent]
```

Updating App Routing module

One last step: return to `app.routing-module.ts` and remove the import of the `WorkoutBuilderComponent` and the route definition that points to the builder:`{ path: 'builder', component: WorkoutBuilderComponent },.`

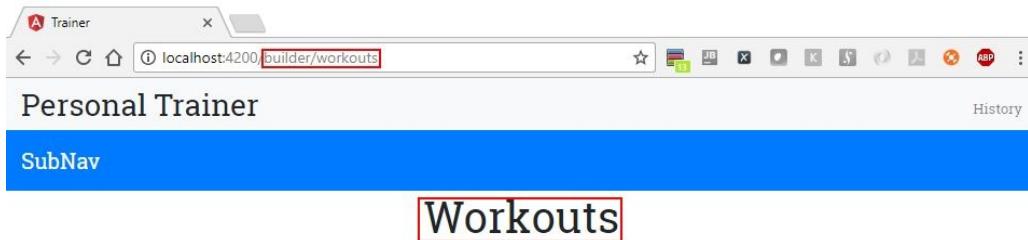


Be sure to leave the import of the `WorkoutBuilderModule` in `app.module.ts` unchanged. We'll discuss removing that in the next section when we cover lazy loading.

Putting it all together

From the previous chapter, we already know how to set up root routing for our application. But now, what we have instead of root routing is area or feature routing that contains child routes. We have been able to achieve the separation of concerns we discussed earlier, so that all the routes related to the *Workout Builder* are now separately contained in their own routing configuration. This means that we can manage all the routing for *Workout Builder* in the `workoutBuilderRoutes` component without affecting other parts of the application.

We can see how the router combines the routes in `app.routes.ts` with the default route in `workout-builder.routes.ts`, if we now navigate from the start page to the *Workout Builder*:



If we look at the URL in the browser, it is `/builder/workouts`. You'll recall that the router link on the start page is `['/builder']`. So how did the router take us to this location?

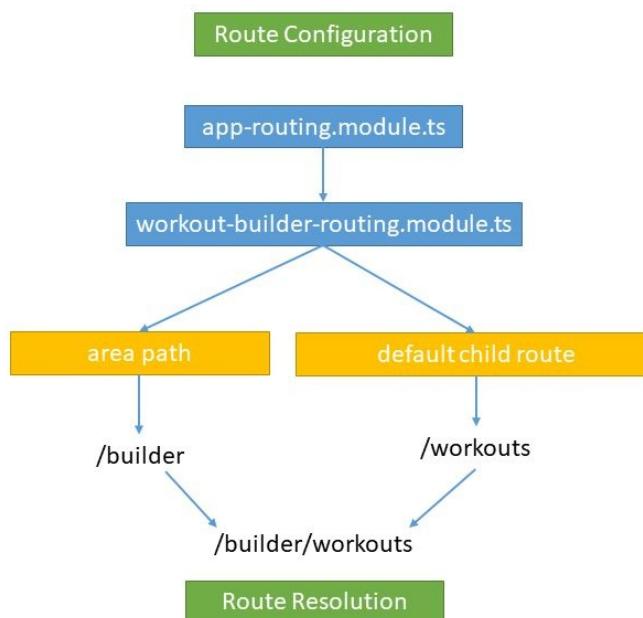
It does it this way: when the link is clicked, the Angular router first looks to `app-routing.module.ts` for the `builder` path because that file contains the configuration for the root routes in our application. The router does not find that path because we have removed it from the routes in that file.

However, `WorkoutBuilderModule` has been imported into our `AppModule` and that `module` in turn imports `workoutBuilderRoutingModule`. The latter file contains the child routes that we just configured. The router finds that `builder` is the parent route in that file and so it uses that route. It also finds the default setting that redirects to the child

path `workouts` in the event that the `builder` path ends with an empty string, which it does in this case.

If you look at the screen, you will see it is displaying the view for `workouts` (and not as previously `Workout Builder`). This means that the router has successfully routed the request to `WorkoutsComponent`, which is the component for the default route in the child route configuration that we set up in `workoutBuilderRoutingModule`.

This process of route resolution is illustrated here:



One final thought on child routing. When you look at our child routing component, `workout-builder.component.ts`, you will see that it has no references to its parent component, `app.component.ts` (as we mentioned earlier, the `<selector>` tag has been removed, so `WorkoutBuilderComponent` is not being embedded in the root component). This means that we have successfully encapsulated `WorkoutBuilderComponent` (and all of its related components that are imported in the `WorkoutBuilderModule`) in a way that will allow us to move all of it elsewhere in the application, or even into a new application.

Now, it's time for us to move on to converting our routing for the Workout Builder to use lazy loading and building out its navigation menus. If you want to see the completed code for this next section, you can download it from the companion codebase in [checkpoint 4.3](#). Again, if you are working along with us as

we build the application, be sure and update the `styles.css` file, which we are not discussing here.



The code is also available for everyone to download on GitHub at <https://github.com/chandermani/angular6byexample>. Checkpoints are implemented as branches in GitHub. The branch to download is as follows: GitHub Branch: `checkpoint4.3` (folder - `trainer`). If you are not using Git, download the snapshot of `checkpoint 4.3` (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/archive/checkpoint4.3.zip>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Lazy loading of routes

When we roll out our application, we expect that our users will be accessing the Workout Runner every day (and we know that this will be the case for you!). But, we anticipate that they will only occasionally be using the Workout Builder to construct their exercises and workout plans. It would therefore be nice if we could avoid the overhead of loading the Workout Builder when our users are just doing their exercises in the Workout Runner. Instead, we would prefer to load Workout Builder only on demand when a user wants to add to or update their exercises and workout plans. This approach is called lazy loading. Lazy loading allows us to employ an asynchronous approach when loading our modules. This means that we can load just what is required to get the application started and then load other modules as we need them.



Under the hood, when we use the Angular CLI to build and serve our application, it uses WebPack's bundling and chunking capabilities to accomplish lazy loading. We'll be discussing these capabilities as we work through how to implement lazy loading in our application.

So in our *Personal Trainer*, we want to change the application so that it only loads the **Workout Builder** on demand. And the Angular **router** allows us to do just that using lazy loading.

But before we get started implementing lazy loading, let's take a look at our current application and how it is loading our modules. With the developer tools open in the Sources tab, start up the application; when the start page appears in your browser, if you look under the webpack node in the source tree, you will see that all the files in the application have loaded, including both the *Workout Runner* and *Workout Builder* files:



So, even though we may just want to use the *Workout Runner*, we have to load the *Workout Builder* as well. In a way, this makes sense if you think of our application as a **Single Page Application (SPA)**. In order to avoid round trips to the server, an SPA will typically load all the resources that will be needed to use the application when it is first started up by a user. But in our case, the important point is that we do not need the *Workout Builder* when the application is first loaded. Instead, we would like to load those resources only when the user decides that they want to add or change a workout or exercise.

So, let's get started with making that happen.

First, modify `app.routing-module.ts` to add the following route configuration for `WorkoutBuilderModule`:

```
| const routes: Routes = [
```

```
    ...
    { path: 'builder', loadChildren: './workout-builder/workout-
builder.module#WorkoutBuilderModule'},
    { path: '**', redirectTo: '/start' }
];
```

Notice that the `loadChildren` property is:

```
| module file path + # + module name
```

This configuration provides the information that will be needed to load and instantiate `WorkoutBuilderModule`.

Next go back to `workout-builder-routing.module.ts` and change the `path` property to an empty string:

```
export const Routes: Routes = [
  {
    path: '',
    ...
  }
];
```

We are making this change because we are now setting the path (`builder`) to the `WorkoutBuilderRoutes` in the new configuration for them that we added in `app.routing-module.ts`.

Finally go back to `app.module.ts` and remove the `WorkoutBuilderModule` import in the `@NgModule` configuration in that file. What this means is that instead of loading the **Workout Builder** feature when the application first starts, we only load it when a user accesses the route to *Workout Builder*.

Let's go build and run the application again using `ng serve`. In the Terminal window, you should see something like the following output:

```
PS C:\Users\Kevin.Hennessy\Github\angular6byexample\trainer> ng serve
** NG Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
40% building modules 256/260 modules 4 active ...dules\cDate: 2018-02-27T20:11:56.286Z
Hash: a40fdab64678f63423b5e
Time: 11933ms
chunk {inline} inline.bundle.js (inline) 5.79 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 205 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 545 kB [initial] [rendered]
chunk {styles} styles.bundle.js (styles) 691 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 8.3 MB [initial] [rendered]
chunk {workout-builder.module} workout-builder.module.chunk.js () 45.9 kB [rendered]

webpack: Compiled successfully.
```

What's interesting here is the last line that shows a separate file for the `workout.builder.module` called `workout-builder.module.chunk.js`. WebPack has used

what is called code splitting to carve out our workout builder module into a separate chunk. This chunk will not be loaded in our application until it is needed (that is, when the router navigates to `WorkoutBuilderModule`).

Now, keeping the Sources tab open in the Chrome developer tools bring up the application in the browser again. When the start page loads, only the files related to the *Workout Runner* appear and **not those** related to the *Workout Builder*, as shown here:



Then, if we clear the Network tab and click on the Create a Workout link, we'll see the `workout-builder.module` chunk load:

Name	Status	Type	Initiator	Size	Time
workout-builder.module.chunk.js	200	script	inline.bundle.js:109	45.1 KB	318 ms

This means that we have achieved encapsulation of our new feature and with asynchronous routing we are able to use lazy loading to load all its components only when needed.

Child and asynchronous routing make it straightforward to implement applications that allow us to have our cake and eat it too. On one hand, we can build SPAs with powerful client-side navigation, while on the other hand we can also encapsulate features in separate child routing components and load them only on demand.

This power and flexibility of the Angular router give us the ability to meet user expectations by closely mapping our application's behavior and responsiveness to the ways they will use the application. In this case, we have leveraged these capabilities to achieve what we set out to do: immediately load *Workout Runner* so that our users can get to work on their exercises right away, but avoid the overhead of loading *Workout Builder* and instead only serve it when a user wants to build a workout.

Now that we have the routing configuration in place in the *Workout Builder*, we will turn our attention to building out the sub-level and left navigation; this will enable us to use this routing. The next sections cover implementing this navigation.

Integrating sub- and side-level navigation

The basic idea around integrating sub- and side-level navigation into the app is to provide context-aware sub-views 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.

Sub-level navigation

We'll start by adding sub-level navigation to the *Workout Builder*. We have already imported our `SubNavMainComponent` into the *Workout Builder*. But, currently it is just displaying placeholder content:

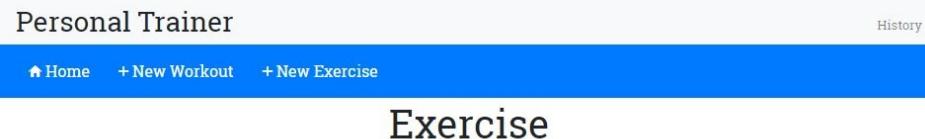


We'll now replace that content with three router links: Home, New Workout, and New Exercise.

Open the `sub-nav-main.component.html` file and change the HTML in it to the following:

```
<nav class="navbar fixed-top navbar-dark bg-primary mt-5">
  <div>
    <a [routerLink]="/builder/workouts'" class="btn btn-primary">
      <span class="ion-md-home"></span> Home
    </a>
    <a [routerLink]="/builder/workout/new'" class="btn btn-primary">
      <span class="ion-md-add"></span> New Workout
    </a>
    <a [routerLink]="/builder/exercise/new'" class="btn btn-primary">
      <span class="ion-md-add"></span> New Exercise
    </a>
  </div>
</nav>
```

Now, rerun the application and you will see the three navigation links. If we click on the New Exercise link button, we will be routed to `ExerciseComponent` and its view will appear in the Router Outlet in the *Workout Builder* view:



The New Workout link button will work in a similar fashion; when clicked on, it will take the user to the `workoutComponent` and display its view in the router outlet.

Clicking on the Home link button will return the user to the `workoutsComponent` and view.

Side navigation

Side-level navigation within the *Workout Builder* will vary depending on the child component that we navigate to. For instance, when we first navigate to the *Workout Builder*, we are taken to the Workouts screen because the `WorkoutsComponent` route is the default route for the *Workout Builder*. That component will need side navigation; it will allow us to select to view a list of workouts or a list of exercises.

The component-based nature of Angular gives us an easy way to implement these context-sensitive menus. We can define new components for each of the menus and then import them into the components that need them. In this case, we have three components that will need side menus: **Workouts**, **Exercises**, and **Workout**. The first two of these components can actually use the same menu so we really only need two side menu components: `LeftNavMainComponent`, which will be like the preceding menu and will be used by the `Exercises` and `Workouts` components, and `LeftNavExercisesComponent`, which will contain a list of existing exercises and will be used by the `Workouts` component.

We already have files for the two menu components, including template files, and have imported them into `WorkoutBuilderModule`. We will now integrate these into the components that need them.

First, modify the `workouts.component.html` template to add the selector for the menu:

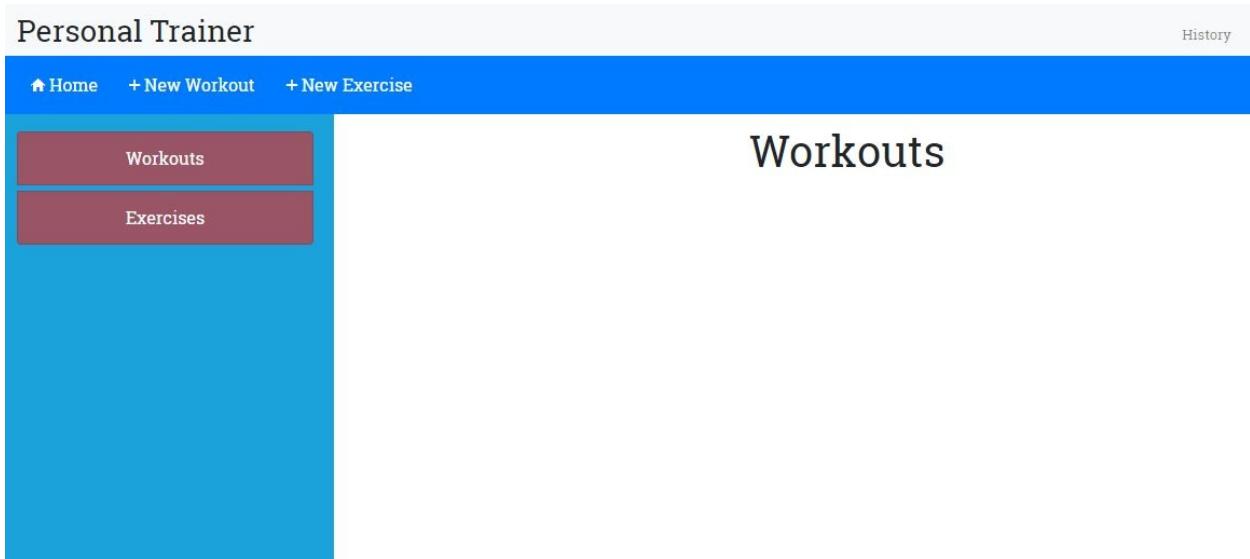
```
<div class="row">
  <div>
    <abe-left-nav-main></abe-left-nav-main>
  </div>
  <div class="col-sm-10 builder-content">
    <h1 class="text-center">Workouts</h1>
  </div>
</div>
```

Then, replace the placeholder text in the `left-nav-main.component.html` with the navigation links to `WorkoutsComponent` and `ExercisesComponent`:

```
<div class="left-nav-bar">
  <div class="list-group">
    <a [routerLink]="/builder/workouts" class="list-group-item list-group-item-action">Workouts</a>
```

```
|     <a [routerLink]="/builder/exercises'" class="list-group-item list-group-item-action">Exercises</a>
|   </div>
|</div>
```

Run the application and you should see the following:



Follow the exact same steps to complete the side menu for the `Exercises` component.



We won't show the code for this menu here, but you can find it in the `workout-builder/exercises` folder under `trainer/src/app` in checkpoint 4.3 of the GitHub repository.

For the menu for the Workout screen, the steps are the same except that you should change `left-nav-exercises.component.html` to the following:

```
| <div class="left-nav-bar">
|   <h3>Exercises</h3>
| </div>
```

We will use this template as the starting point for building out a list of exercises that will appear on the left-hand side of the screen and can be selected for inclusion in a workout.

Implementing workout and exercise lists

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 [Chapter 5](#), *Supporting Server Data Persistence*, 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

The code is available to download on GitHub at <https://github.com/chandermani/angular6byexample>. The branch to download is as follows: **GitHub Branch: checkpoint4.4** (folder—`trainer`). If you are not using Git, download the snapshot of `checkpoint 4.4` (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/archive/checkpoint4.4.zip>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time. Again, if you are working along with us as we build the application, be sure to update the `styles.css` file, which we are not discussing here. Because some of the files in this section are rather long, rather than showing the code here, we are also going to suggest at times that you simply copy the files into your solution.

Locate `workout-service.ts` in the `trainer/src/core` folder. The code in that file should look like the following, except for the implementation of the two methods `setupInitialExercises` and `setupInitialWorkouts`, which we have left out because of their length:

```
import {Injectable} from '@angular/core';
import {ExercisePlan} from './model';
import {WorkoutPlan} from './model';
import {Exercise} from "./model";
import { CoreModule } from './core.module';

@Injectable({
  providedIn: CoreModule
})
export class WorkoutService {
  workouts: Array<WorkoutPlan> = [];
  exercises: Array<Exercise> = [];

  constructor() {
    this.setupInitialExercises();
    this.setupInitialWorkouts();
  }
}
```

```
    }

    getExercises(){
      return this.exercises;
    }

    getWorkouts(){
      return this.workouts;
    }
    setupInitialExercises(){
      // implementation of in-memory store.
    }

    setupInitialWorkouts(){
      // implementation of in-memory store.
    }
  }}
}
```

As we have mentioned before, the implementation of an Angular service is straightforward. Here, we are declaring a class with the name `workoutService` and decorating it with `@Injectable`. Within the `@Injectable` decorator, we have sets the `provided-in` property to `coreModule`. This registers `workoutService` as a provider with Angular's **Dependency Injection** framework and makes it available throughout our application.

In the class definition, we first create two arrays: one for `workouts` and one for `Exercises`. These arrays are of types `WorkoutPlan` and `Exercise` respectively, and we therefore need to import `WorkoutPlan` and `Exercise` from `model.ts` to get the type definitions for them.

The constructor calls two methods to set up the Workouts and Services List. At the moment, we are just using an in-memory store that populates these lists with data.

The two methods, `getExercises` and `getWorkouts`, as the names suggest, return a list of exercises and workouts respectively. Since we plan to use the in-memory store to store workout and exercise data, the `workouts` and `Exercises` arrays store this data. As we go along, we will be adding more functions to the service.

Time to build out the components for the workout and exercise lists!

Workout and exercise list components

First, open the `workouts.component.ts` file in the `trainer/src/app/workout-builder/workouts` folder and update the imports as follows:

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

import { WorkoutPlan } from '../../../../../core/model';
import { WorkoutService } from '../../../../../core/workout.service';
```

This new code imports the Angular `Router` as well as `WorkoutService` and the `WorkoutPlan` type.

Next, replace the class definition with the following code:

```
export class WorkoutsComponent implements OnInit {
  workoutList:Array<WorkoutPlan> = [];

  constructor(
    public router:Router,
    public workoutService:WorkoutService) {}

  ngOnInit() {
    this.workoutList = this.workoutService.getWorkouts();
  }

  onSelect(workout: WorkoutPlan) {
    this.router.navigate( ['./builder/workout', workout.name] );
  }
}
```

This code adds a constructor into which we are injecting the `Router` and the `WorkoutService`. The `ngOnInit` method then calls the `getWorkouts` method on the `WorkoutService` and populates a `workoutList` array with a list of `WorkoutPlans` returned from that method call. We'll use that `workoutList` array to populate the list of workout plans that will display in the `workouts` component's view.

You'll notice that we are putting the code for calling `workoutService` into the `ngOnInit` method. We want to avoid placing this code in the constructor. Eventually, we will be replacing the in-memory store that this service uses with a call to an external data store and we do not want the instantiation of our component to be affected by this call. Adding these method calls to the constructor would also complicate testing the component.

To avoid such unintended side effects, we instead place the code in the `ngOnInit` method. This method implements one of Angular's life cycle hooks, `onInit`, which Angular calls after creating an instance of the service. This way, we rely on Angular to call this method in a predictable way that does not affect the instantiation of the component.

Next, we'll make almost identical changes to the `Exercises` component. As with the `workouts` component, this code injects the workout service into our component. This time, we then use the workout service to retrieve the exercises.



Because it so similar to what we just showed you for the `workouts` component, we won't show that code here. Just add it from the `workout-builder/exercises` folder in checkpoint 4.4.

Workout and exercise list views

Now, we need to implement the list views that have so far been empty!



In this section, we will be updating the code from `checkpoint 4.3` with what is found in `checkpoint 4.4`. So if you are coding along with us, simply follow the steps laid out in this section. If you want to see the finished code, then just copy the files from `checkpoint 4.4` into your solution.

Workouts list views

To get the view working, open `workouts.component.html` and add the following markup:

```
<div class="row">
  <div>
    <abe-left-nav-main></abe-left-nav-main>
  </div>
  <div class="col-sm-10 builder-content">
    <h1 class="text-center">Workouts</h1>
    <div *ngFor="let workout of workoutList|orderBy:'title'" class="workout tile"
(click)="onSelect(workout)">
      <div class="title">{{workout.title}}</div>
      <div class="stats">
        <span class="duration" title="Duration"><span class="ion-md-time"></span>
- {{(workout.totalWorkoutDuration? workout.totalWorkoutDuration(): 0)|secondsToTime}}</span>
        <span class="float-right" title="Exercise Count"><span class="ion-md-list"></span> - {{workout.exercises.length}}</span>
      </div>
    </div>
  </div>
</div>
```

We are using one of the Angular core directives, `ngFor`, to loop through the list of workouts and display them in a list on the page. We add the `*` sign in front of `ngFor` to identify it as an Angular directive. Using a `let` statement, we assign `workout` as a local variable that we use to iterate through the workout list and identify the values to be displayed for each workout (for example, `workout.title`). We then use one of our custom pipes, `orderBy`, to display a list of workouts in alphabetical order by title. We are also using another custom pipe, `secondsToTime`, to format the time displayed for the total workout duration.



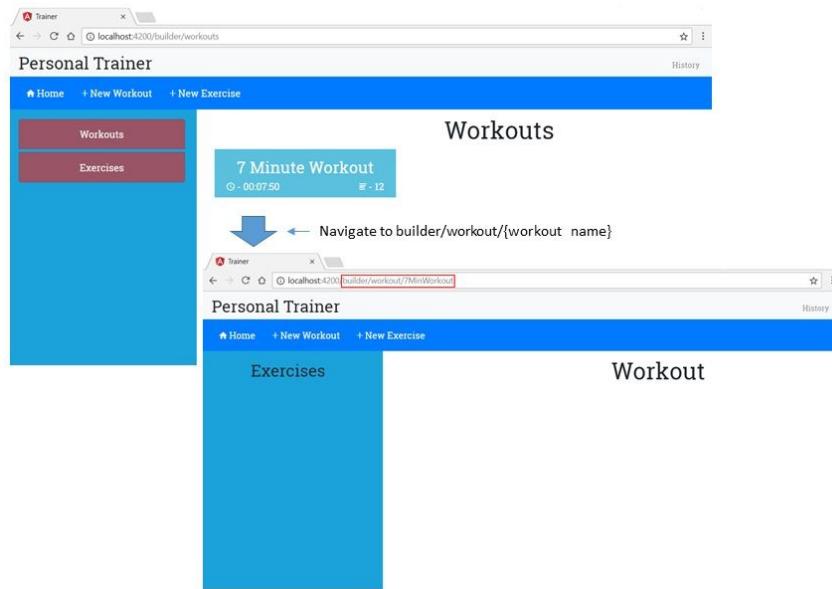
If you are coding along with us, you will need to move the `secondsToTime` pipe into the shared folder and include it in the `sharedModule`. Then, add `sharedModule` to `WorkoutBuilderModule` as an additional import. That change has already been made in `checkpoint 4.4` in the GitHub repository.

Finally, we bind the click event to the following `onSelect` method that we add to our component:

```
onSelect(workout: WorkoutPlan) {
  this.router.navigate( ['/builder/workout', workout.name] );
}
```

This sets up navigation to the workout details page. This navigation happens when we 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.

Go ahead and refresh the builder page (`/builder/workouts`); one workout is listed, the 7 Minute Workout. Click on the tile for that workout. You'll be taken to the **Workout screen** and the workout name, `7MinWorkout`, will appear at the end of the URL:



The Workout screen

Exercises list views

We are going to follow the same approach for the `Exercises` list view as we did for the `Workouts` list view, except that in this case, we will actually be implementing two views: one for the `Exercises` component (which will display in the main content area when a user navigates to that component) and one for the `LeftNavExercisesComponent` `exercises` context menu (which will display when the user navigates to the `workouts` component to create or edit a workout).

For the `Exercises` component, we will follow an approach that is almost identical to what we did to display a list of workouts in the `Workouts` component. So, we won't show that code here. Just add the files for `exercises.component.ts` and `exercises.component.html` from `checkpoint 4.4`.

When you are done copying the files, click on the Exercises link in the left navigation to load the 12 exercises that you have already configured in `WorkoutService`.

As with the `Workouts` list, this sets up the navigation to the exercise detail page. Clicking on an item in the exercises list takes us to the exercise detail page. The selected exercise name is passed as part of the route/URL to the exercise detail page.

In the final list view, we will add a list of exercises that will display in the left context menu for the *Workout Builder* screen. This view is loaded in the left navigation when we create or edit a workout. Using Angular's component-based approach, we will update the `leftNavExercisesComponent` and its related view to provide this functionality. Again we won't show that code here. Just add the files for `left-nav-exercises.component.ts` and `left-nav-exercises.component.html` from the `trainer/src/app/navigation` folder in `checkpoint 4.4`.

Once you are done copying those files, click on the New Workout button on the sub-navigation menu in the *Workout Builder* and you will now see a list of exercises displayed in the left navigation menu—exercises that we have already configured in `WorkoutService`.

Time to add the ability to load, save, and update exercise/workout data!

Building a workout

The core functionality *Personal Trainer* provides centers around workout and exercise building. Everything is there to support these two functions. In this section, we focus on building and editing workouts using Angular.

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:

Personal Trainer

Home + New Workout + New Exercise

Exercises

- Abdominal Crunches >
- High Knees >
- Jumping Jacks >
- Lunges >
- Plank >
- Push up >
- Pushup And Rotate >
- Side Plank >
- Squat >
- Step Up Onto Chair >
- Tricep Dips On Chair >
- Wall Sit >

7 Minute Workout

Jumping Jacks

1

30 seconds

Wall Sit

2

30 seconds

Push up

3

30 seconds

Name: 7MinWorkout

Title: 7 Minute Workout

Description: Enter workout description.

Rest Time (in seconds): 10

Total Exercises: 12

Total Duration: 00:07:50

Save

This screenshot shows the 'Workout Builder' interface. On the left, a sidebar lists various exercises. The main area displays a '7 Minute Workout' plan with three exercises: 'Jumping Jacks', 'Wall Sit', and 'Push up'. Each exercise card includes an illustration, a duration of '30 seconds', and a sequence of images showing the movement. To the right of the exercises, there are fields for 'Name' (7MinWorkout), 'Title' (7 Minute Workout), and 'Description' (Enter workout description). Below these are fields for 'Rest Time (in seconds)' (10), 'Total Exercises' (12), and 'Total Duration' (00:07:50). A large yellow 'Save' button is at the bottom right.

The page has a left navigation that lists 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/workout/new`.
- Edit: This mode is used for editing the existing workout. The URL is `#/builder/workout/: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).

Finishing left nav

At the end of the previous section, we updated the left navigation view for the `workout` component to show a list of exercises. Our intention was to let the user click on an arrow next to an exercise to add it to the workout. At the time, we deferred implementing the `addExercise` method in the `LeftNavExercisesComponent` that was bound to that click event. Now, we will go ahead and do that.

We have a couple of options here. The `LeftNavExercisesComponent` is a child component of the `workoutComponent`, so we can implement child/parent inter-component communication to accomplish that. We covered this technique in the previous chapter while working on *7 Minute Workout*.

However, adding an exercise to the workout is part of a larger process of building the workout and using child/parent inter-component communication would make the implementation of the `AddExercise` method differ from the other functionality that we will be adding going forward.

For this reason, it makes more sense to follow another approach for sharing data, one that we can use consistently throughout the process of building a workout. That approach involves using a service. As we get into adding the other functionality for creating an actual workout, such as save/update logic and implementing the other relevant components, the benefits of going down the service route will become increasingly clear.

So, we introduce a new service into the picture: `WorkoutBuilderService`. The ultimate aim of `WorkoutBuilderService` service is to coordinate between `WorkoutService` (which retrieves and persists the workout) and the components (such as `LeftNavExercisesComponent` and others we will add later), while the workout is being built, hence reducing the amount of code in `workoutComponent` to the bare minimum.

Adding WorkoutBuilderService

`WorkoutBuilderService` monitors the state of the workout that a user of the application is building. It:

- Tracks the current workout
- Creates a new workout
- Loads the existing workout
- Saves the workout

Copy `workout-builder-service.ts` from the `workout-builder/builder-services` folder under `trainer/src/app` in checkpoint 4.5.

 *The code is also available for everyone to download on GitHub at <https://github.com/chandermani/angular6byexample>. Checkpoints are implemented as branches in GitHub. The branch to download is as follows: GitHub Branch: `checkpoint4.5` (folder—`trainer`). If you are not using Git, download the snapshot of `checkpoint 4.5` (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/archive/checkpoint4.5.zip>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time. Again, if you are working along with us as we build the application, be sure to update the `styles.css` file, which we are not discussing here.*

While we normally make services available application-wide, `WorkoutBuilderService` will only be used in the *Workout Builder* feature. Therefore, instead of registering it with the providers in `AppModule`, we have registered it in the provider array of `WorkoutBuilderModule` as follows (after adding it as an import at the top of the file):

```
@NgModule({
  ...
  providers: [WorkoutBuilderService]
})
```

Adding it as a provider here means that it will only be loaded when the *Workout Builder* feature is being accessed and it cannot be reached outside this module. This means that it can be evolved independently of other modules in the application and can be modified without affecting other parts of the application.

Let's look at some of the relevant parts of the service.

`WorkoutBuilderService` needs the type definitions for `WorkoutPlan`, `ExercisePlan`, and

`WorkoutService`, so we import these into the component:

```
| import { WorkoutPlan, ExercisePlan } from '....core/model';
| import { WorkoutService } from '....core/workout.service';
```

`WorkoutBuilderService` has a dependency on `WorkoutService` to provide persistence and querying capabilities. We resolve this dependency by injecting `WorkoutService` into the constructor for `WorkoutBuilderService`:

```
| constructor(public workoutService: WorkoutService) {}
```

`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:

```
startBuilding(name: string){
  if(name){
    this.buildingWorkout = this.workoutService.getWorkout(name)
    this.newWorkout = false;
  }else{
    this.buildingWorkout = new WorkoutPlan("", "", 30, []);
    this.newWorkout = true;
  }
  return this.buildingWorkout;
}
```

The basic idea behind this tracking function is to set up a `WorkoutPlan` object (`buildingWorkout`) that will be made available to components to manipulate the workout details. The `startBuilding` method 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` object 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 save and update situations when the `save` method on this service is called.

The rest of the methods, `removeExercise`, `addExercise`, and `moveExerciseTo`, are self-explanatory and affect the exercise list that is part of the workout (`buildingWorkout`).

`WorkoutBuilderService` is calling a new method, `getWorkout`, on `WorkoutService`, which we have not added yet. Go ahead and copy the `getWorkout` implementation from

~~the `workout-service.ts` file in the `services` folder under `trainer/src` in checkpoint 4.5.
We will not dwell on 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 ExerciseNav

To add exercises to the workout we are building, we just need to import `WorkoutBuilderService` and `ExercisePlan`, inject `WorkoutBuilderService` into the `LeftNavExercisesComponent`, and call its `addExercise` method, passing the selected exercise as a parameter:

```
constructor(  
    public workoutService:WorkoutService,  
    public workoutBuilderService:WorkoutBuilderService) {}  
...  
addExercise(exercise:Exercise) {  
    this.workoutBuilderService.addExercise(new ExercisePlan(exercise, 30));  
}
```

Internally, `workoutBuilderService.addExercise` updates the `buildingWorkout` model data with the new exercise.

The preceding implementation is a classic case of sharing data between independent components. The shared service exposes the data in a controlled manner to any component that requests it. While sharing data, it is always a good practice to expose the state/data using methods instead of directly exposing the data object. We can see that in our component and service implementations too. `LeftNavExercisesComponent` 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, `WorkoutBuilderService` needs to be initialized by calling `startBuilding` before `addExercise` is called. What happens if a component calls `addExercise` before the initialization takes place?

Implementing the Workout component

The `WorkoutComponent` 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 component will be reduced. Other than the primary responsibility of integrating with, exposing, and interacting with its template view, we will delegate most of the other work to `WorkoutBuilderService`.

The `WorkoutComponent` is associated with two routes/views, namely `/builder/workout/new` and `/builder/workout/:id`. These routes handle both creating and editing workout scenarios. The first job of the component is to load or create the workout that it needs to manipulate.

Route parameters

But before we get to building out the `workoutComponent` and its associated view, we need to touch briefly on the navigation that brings a user to the screen for that component. This component handles both creating and editing workout scenarios. The first job of the component is to load or create the workout that it needs to manipulate. We plan to use Angular's routing framework to pass the necessary data to the component, so that it will know whether it is editing an existing workout or creating a new one, and in the case of an existing workout, which component it should be editing.

How is this done? The `workoutComponent` is associated with two routes, namely `/builder/workout/new` and `/builder/workout/:id`. The difference in these two routes lies in what is at the end of these routes; in one case, it is `/new`, and in the other, `/:id`. These are called **route parameters**. The `:id` in the second route is a token for a route parameter. The router will convert the token to the ID for the workout component. As we saw earlier, this means that the URL that will be passed to the component in the case of *7 Minute Workout* will be `/builder/workout/7MinuteWorkout`.

How do we know that this workout name is the right parameter for the ID? As you recall, when we set up the event for handling a click on the Workout tiles on the Workouts screen that takes us to the Workout screen, we designated the workout name as the parameter for the ID, like so:

```
| onSelect(workout: WorkoutPlan) {  
|   this.router.navigate( ['./builder/workout', workout.name] );  
| }
```

Here, we are constructing the route using the programmatic interface for the router (we covered routing in detail in the previous chapter, so we won't go over that again here). The router.navigate method accepts an array. This is called the **link parameters array**. The first item in the array is the path of the route, and the second is a route parameter that specifies the ID of the workout. In this case, we set the `id` parameter to the workout name. From our discussion of routing in the previous chapter, we know that we can also construct the same type of URL as part of a router link or simply enter it in the browser to get to the Workouts screen and edit a particular workout.

The other of the two routes ends with `/new`. Since this route does not have a `token` parameter, the router will simply pass the URL unmodified to the `WorkoutComponent`. The `WorkoutComponent` will then need to parse the incoming URL to identify that it should be creating a new component.

Route guards

But before the link takes the user to the `workoutComponent`, there is another step along the way that we need to consider. The possibility always exists that the ID that is passed in the URL for editing a workout could be incorrect or missing. In those cases, we do not want the component to load, but instead we want to have the user redirected to another page or back to where they came from.

Angular offers a way to accomplish this result with **route guards**. As the name implies, route guards **provide a way to prevent navigation to a route**. A route guard can be used to inject custom logic that can do things such as check authorization, load data, and make other verifications to determine whether the navigation to the component needs to be canceled or not. And all of this is done before the component loads so it is never seen if the routing is canceled.

Angular offers several route guards, including `CanActivate`, `CanActivateChild`, `CanDeActivate`, `Resolve`, and `CanLoad`. At this point, we are interested in the `Resolve` route guard. The `Resolve` guard will allow us not only to check for the existence of a workout, but also to load the data associated with a workout before loading the `workoutComponent`. The advantage of doing the latter is that we avoid the necessity of checking to make sure the data is loaded in the `workoutComponent` and it eliminates adding conditional logic throughout its component template to make sure that the data is there when it is rendered. This will be especially useful when in the next chapter when we start using `observables` where we must wait for the observable to complete before we are guaranteed of having the data that it will provide. The `Resolve` guard will handle waiting for the observable to complete, which means that the `workoutComponent` will be guaranteed to have the data that it needs before it loads.

Implementing the resolve route guard

The `Resolve` guard allows us to prefetch the data for a workout. In our case, what we want to do is use `Resolve` to check the validity of any ID that is passed for an existing workout. Specifically, we will run a check on that ID by making a call to the `WorkoutBuilderService` to retrieve the Workout Plan and see if it exists. If it exists, we will load the data associated with the Workout Plan so that it is available to the `workoutComponent`; if not we will redirect back to the Workouts screen.

Copy `workout.resolver.ts` from the `workout-builder/workout` folder under `trainer/src/app/workout` in checkpoint 4.5 and you will see the following code:

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/take';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { Router, Resolve, ActivatedRouteSnapshot,
         RouterStateSnapshot } from '@angular/router';
import { WorkoutPlan } from '../../../../../core/model';
import { WorkoutBuilderService } from '../builder-services/workout-builder.service';

@Injectable()
export class WorkoutResolver implements Resolve<WorkoutPlan> {
  public workout: WorkoutPlan;

  constructor(
    public workoutBuilderService: WorkoutBuilderService,
    public router: Router) {}

  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): WorkoutPlan {
    let workoutName = route.paramMap.get('id');

    if (!workoutName) {
      workoutName = '';
    }

    this.workout = this.workoutBuilderService.startBuilding(workoutName);

    if (this.workout) {
      return this.workout;
    } else { // workoutName not found
      this.router.navigate(['/builder/workouts']);
      return null;
    }
  }
}
```

As you can see, the `WorkoutResolver` is an injectable class that implements the `Resolve` interface. The code injects the `workoutBuilderService` and `Router` into the class and implements the interface with the `resolve` method. The `resolve` method accepts two parameters; `ActivatedRouteSnapshot` and `RouterStateSnapshot`. In this case, we are only interested in the first of these two parameters, `ActivatedRouteSnapshot`. It contains a `paramMap` from which we extract the ID parameter for the route.

The `resolve` method then calls the `startBuilding` method of `WorkoutBuildingService` using the parameter supplied in the route. If the workout exists, then `resolve` returns the data and the navigation proceeds; if not, it re-routes the user to the `workouts` page and returns false. If `new` is passed as an ID, `workoutBuilderService` will load a new workout and the `Resolve` guard will allow navigation to proceed to the `WorkoutComponent`.

The `resolve` method can return a Promise, an Observable, or a synchronous value. If we return an Observable, we will need to make sure that the Observable completes before proceeding with navigation. In this case, however, we are making a synchronous call to a local in-memory data store, so we are just returning a value.

To complete the implementation of the `WorkoutResolver`, first make sure to import and add it to `WorkoutBuilderModule` as a provider:

```
....  
import { WorkoutResolver } from './workout/workout.resolver';  
  
@NgModule({  
  ....  
  providers: [WorkoutBuilderService, WorkoutResolver]  
})  
....
```

Then, add it to the route configuration for `WorkoutComponent` by updating `workout-builder-routing.module.ts` as follows:

```
....  
import { WorkoutResolver } from './workout/workout.resolver';  
....  
const routes: Routes = [  
  {  
    path: '',  
    component: WorkoutBuilderComponent,  
    children: [  
      {path: '', pathMatch: 'full', redirectTo: 'workouts'},  
      {path: 'workouts', component: WorkoutsComponent },  
      {path: 'workout/new', component: WorkoutComponent, resolve: { workout:
```

```
WorkoutResolver} },
      {path: 'workout/:id', component: WorkoutComponent, resolve: { workout:
WorkoutResolver } },
        {path: 'exercises', component: ExercisesComponent},
        {path: 'exercise/new', component: ExerciseComponent },
        {path: 'exercise/:id', component: ExerciseComponent }
    ],
];
};
```

As you can see, we add `WorkoutResolver` to the routing module's imports. Then, we add `resolve { workout: WorkoutResolver }` to the end of the route configuration for `workout/new` and `workout/:id`. This instructs the router to use the `WorkoutResolver` resolve method and assign its return value to `workout` in the route's data. This configuration means that `WorkoutResolver` will be called prior to the router navigating to `WorkoutComponent` and that the workout data will be available to the `WorkoutComponent` when it loads. We'll see how to extract this data in the `WorkoutComponent` in the next section.

Implementing the Workout component continued...

Now that we have established the routing that takes us to the `workout` component, let's turn to completing its implementation. So, copy the `workout.component.ts` file from the `workout-builder/workout` folder under `trainer/src/app` in checkpoint 4.5. (Also, copy `workout-builder.module.ts` from the `workout-builder` folder. We'll discuss the changes in this file a little later when we get to Angular forms.)

Open `workout.component.ts` and you'll see that we have added a constructor that injects `ActivatedRoute` and `WorkoutBuilderService`:

```
| constructor(  
|   public route: ActivatedRoute,  
|   public workoutBuilderService:WorkoutBuilderService){ }
```

In addition, we have added the following `ngOnInit` method:

```
| ngOnInit() {  
|   this.sub = this.route.data  
|   .subscribe(  
|     (data: { workout: WorkoutPlan }) => {  
|       this.workout = data.workout;  
|     }  
|   );  
| }
```

The method subscribes to the `route` and extracts the `workout` from the `route.data`. There is no need to check the `workout` exists because we have already done that in the `WorkoutResolver`.



We are subscribing to the `route.data` because as an `ActivatedRoute`, the `route` exposes its `data` as an observable, which can change during the lifetime of the component. This gives us the ability to reuse the same component instance with different parameters, even though the `onInit` life cycle event for that component is called only once. We'll cover observables in detail in the next chapter.

In addition to this code, we have also added a series of methods to the `workout` component for adding, removing, and moving a `workout`. These methods all call corresponding methods on the `WorkoutBuilderService` and we will not review them in detail here. We've also added an array of `durations` for populating the duration

drop-down list.

For now, this is enough for the **component** class implementation. Let's update the associated `workout` template.

Implementing the Workout template

Now, copy the `workout.component.html` files from the `workout-builder/workout` folder under `trainer/src/app` in `checkpoint 4.5`. 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 *Building a workout* section.



*In the event of any problem, you can refer to the `checkpoint4.5` code in the GitHub repository:
Branch: `checkpoint4.5` (folder - `trainer`).*

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 the exercises that are part of the workout in order. We display them as top-to-bottom tiles in the left part of the content area. 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

We use `ngFor` to iterate over the list of exercises and display them:

```
|<div *ngFor="let exercisePlan of workout.exercises; let i=index" class="exercise-item">
```

You will notice that we are using the `*` asterisk in front of `ngFor`, which is shorthand for the `<template>` tag. We are also using `let` to set two local variables: `exercisePlan` to identify an item in the list of exercises and `i` to set up an index value that we will use to show a number for the exercises as they are displayed on the screen. We will also use the index value to manage reordering and deleting exercises from the list.

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 workout changes.

The complete list has been wrapped inside the HTML form element so that we can make use of the form-related capabilities that Angular provides. So, what are these capabilities?

almost 50 pages talks about form

Angular forms

Forms are such an integral part of HTML development that any framework that targets client-side development just cannot ignore them. Angular 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 inputs
- Validating those inputs against business rules
- Submitting the data to the backend server

Angular has something to offer for all the preceding use cases.

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 the input before it can be submitted.

Lastly, Angular provides HTTP services for client-server interaction and persisting data to the server. We'll cover those services in [Chapter 5, Supporting Server Data Persistence](#).

Since the first two use cases are our main focus in this chapter, let's learn more about Angular user input and data validation support.

Template-driven and reactive forms

Angular offers two types of forms: **template-driven** and **reactive**. We'll be discussing both types of form in this chapter. Because the Angular team indicates that many of us will primarily use **template-driven forms**, that is what we will start with in this chapter.

Template-driven forms

As the name suggests, **template-driven forms** place the emphasis on developing a form within an HTML template and handling most of the logic for the form inputs, data validation, saving, and updating in-form directives placed within that template. The result is that very little form-related code is required in the component class that is associated with the form's template.

Template-driven forms make heavy use of the `ngModel` form directive. We will be discussing it in the next sections. It provides two-way data binding for form controls, which is a nice feature indeed. It allows us to write much less boilerplate code to implement a form. It also helps us to manage the state of the form (such as whether the form controls have changed and whether these changes have been saved). And, it also gives us the ability to easily construct messages that display if the validation requirements for a form control have not been met (for example, a required field not provided, email not in the right format, and so on).

Getting started

In order to use Angular forms in our `workout` component, we must first add some additional configuration. Open `workout-builder.module.ts` from the `workout-builder` folder under `trainer/src/app` in checkpoint 4.5. You will see that it imports `FormsModule`:

```
...
import { FormsModule } from '@angular/forms';
...
@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    SharedModule,
    workoutBuilderRouting
  ],
})
```

This brings in all that we will need to implement our form, including:

- `NgForm`
- `ngModel`

Let's start using these to build our form.

Using NgForm

In our template (`workout.component.html`), we have added the following `form` tag:

```
| <form #f="ngForm" class="row" name="formWorkout" (ngSubmit)="save(f.form)". . .
| </form>
```

Let's take a look at what we have here. One interesting thing is that we are still using a standard `<form>` tag and not a special Angular tag. We've also used `#` to define a local variable `f` to which we have assigned `ngForm`. Creating this local variable provides us with the convenience of being able to use it for form-related activity in other places within the form. For example, you can see that we are using it at the end of the opening `form` tag in a parameter, `f.form`, which is being passed to the `onsubmit` event bound to `(ngSubmit)`.

That last binding to `(ngSubmit)` should tell us that something different is going on here. Even though we did not explicitly add the `NgForm` directive, our `<form>` now has additional events such as `ngSubmit` to which we can bind actions. How did this happen? Well, this was not triggered by our assigning `ngForm` to a local variable. Instead, it happened *auto-magically* because we imported the forms module into `workout-builder.module.ts`.

With that import in place, Angular scanned our template for a `<form>` tag and wrapped that `<form>` tag within an `NgForm` directive. The Angular documentation indicates that `<form>` elements in the component will be upgraded to use the Angular form system. This is important because it means that various capabilities of `NgForm` are now available to use with the form. These include the `ngSubmit` event, which signals when a user has triggered a form submission and provides the ability to validate the entire form before submitting it.

ngModel

One of the fundamental building blocks for template-driven forms is `ngModel`, and you will find it being used throughout our form. One of the primary roles of `ngModel` is to support two-way binding between user input and an underlying model. With such a setup, changes in the model are reflected in the view, and updates to the view too are reflected back on the model. Most of the other directives that we have covered so far only support one-way binding from models to views. `ngModel` goes both ways. But, be aware that it is **only available within `NgForm` for use with elements that allow user input.**

As you know, we already have a model that we are using for the Workout page, `WorkoutPlan`. Here is the `WorkoutPlan` model from `model.ts`:

```
export class WorkoutPlan {
  constructor(
    public name: string,
    public title: string,
    public restBetweenExercise: number,
    public exercises: ExercisePlan[],
    public description?: string)
  }
  totalWorkoutDuration(): number{
    . . . [code calculating the total duration of the workout]. . .
  }
}
```

Note the use of the `?` after `description`. This means that it is an optional property in our model and is not required to create a `WorkoutPlan`. In our form, this will mean that we will not require that a description be entered and everything will work fine without it.

Within the `WorkoutPlan` model, we also have a reference to an array made up of instances of another type of model: `ExercisePlan`. `ExercisePlan` in turn is made up of a number (`duration`) and another model (`Exercise`), which looks like this:

```
export class Exercise {
  constructor(
    public name: string,
    public title: string,
    public description: string,
    public image: string,
    public nameSound?: string,
    public procedure?: string,
    public videos?: Array<string>) { }
```

```
| }
```

The use of these nested classes shows that we can create complex hierarchies of models that can all be data-bound within our form using `NgModel`. So throughout the form, whenever we need to update one of the values in a `workoutPlan` or an `ExercisePlan`, we can use `NgModel` to do that (the `workoutPlan` model will be represented by a local variable named `workout` in the following examples).

Using ngModel with input and textarea

Open `workout-component.html` and look for `ngModel`. It has been applied to form 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" class="form-control" id="workout-name"
|placeholder="Enter workout name. Must be unique." [(ngModel)]="workout.name">
```

The preceding `[(ngModel)]` directive sets up a two-way binding between the input control and the `workout.name` model property. The brackets and parentheses should each look familiar. Previously, we used them separately from each other: the `[]` brackets for property binding and the `()` parentheses for event binding. In the latter case, we usually bound the event to a call to a method in the component associated with the template. You can see an example of this in the form with the button that a user clicks on to remove an exercise:

```
|<span class="btn float-right trashcan" (click)="removeExercise(exercisePlan)"><span
|class="ion-ios-trash-outline"></span></span>
```

Here, the `click` event is explicitly bound to a method called `removeExercise` in our `workout` component class. But for the `workout.name` input, we do not have an explicit binding to a method on the component. So what's going on here and how does the update happen without us calling a method on the component? The answer to that question is that the combination `[()]` is shorthand for both binding a model property to the input element and wiring up an event that updates the model.

Put differently, if we reference a model element in our form, `ngModel` is smart enough to know that what we want to do is update that element (`workout.name` here) when a user enters or changes the data in the input field to which it is bound. Under the hood, Angular creates an update method similar to what we would otherwise have to write ourselves. Nice! This approach keeps us from having to write repetitive code to update our model.

Angular supports most of the HTML5 input types, including `text`, `number`, `select`,

radio, and checkbox. This means binding between a model and any of these input types just works out of the box.

The `textarea` element works the same as the `input`:

```
| <textarea name="description" . . . [(ngModel)]="workout.description"></textarea>
```

Here, we bind `textarea` to `workout.description`. Under the hood, `ngModel` updates the `workout.description` in our model with every change we type into the text area.

To test out how this works, why don't we verify this binding? Add a model interpolation expression at the end of any of the linked inputs, such as this one:

```
| <input type="text" . . . [(ngModel)]="workout.name">{{workout.name}}
```

Open the Workout page, type something in the input, and see how the interpolation is updated instantaneously. The magic of two-way binding!

Name:

7minwork

7minwork

Using ngModel with select

Let's look at how `select` has been set up:

```
| <select . . . name="duration" [(ngModel)]="exercisePlan.duration">
|   <option *ngFor="let duration of durations" [value]="duration.value">
|     {{duration.title}}</option>
| </select>
```

We are using `ngFor` here to bind to an array, `durations`, which is in the `workout` component class. The array looks like this:

```
| [{ title: "15 seconds", value: 15 },
|   { title: "30 seconds", value: 30 }, ...]
```

The `ngFor` component will loop over the array and populate the drop-down values with the corresponding values in the array with the title for each item being displayed using interpolation, `{{duration.title}}`. And `[(ngModel)]` then binds the drop-down selection to the `exercisePlan.duration` in the model.

Notice here that we are binding to the nested model: `ExercisePlan`. And, we may have multiple exercises to which we will be applying this binding. With that being the case, we have to make use of another Angular form directive —`ngModelGroup`—to handle these bindings. `ngModelGroup` will allow us to create a nested group within our model that will contain the list of exercises included in the workout and then in turn loop over each exercise to bind its duration to the model.

To start with, we will add `ngModelGroup` to the `div` tag that we have created within the form to hold our list of exercises:

```
| <div id="exercises-list" class="col-sm-2 exercise-list" ngModelGroup="exercises">
```

That takes care of creating the nested list of exercises. Now, we have to handle the individual exercises within that list, and we can do that by adding another `ngModelGroup` to the individual divs that contain each exercise:

```
| <div class="exercise tile" [ngModelGroup]="i">
```

Here, we are using the index in our for loop to dynamically create an individual

model group for each of our exercises. These model groups will be nested inside the first model group that we created. Temporarily, add the tag `<pre>{{ f.value | json }}</pre>` to the bottom of the form and you will be able to see the structure of this nested model:

```
{ "exercises": { "0": { "duration": 15 }, "1": { "duration": 60 }, "2": { "duration": 45 }, "exerciseCount": 3 }, "workoutName": "1minworkout", "title": "1 Minute Workout", "description": "desc", "restBetweenExercise": 30 }
```



This is powerful stuff that enables us to create complicated forms with nested models, all of which can use `ngModel` for databinding.



You may have noticed a subtle difference in the two `ngModelGroup` directive tags we just introduced. The second of the two is wrapped in angle brackets, `[]`, while the first is not. This is because with the first tag we are just naming our model group, whereas with the second we are binding it dynamically to each exercise's div tag using the index of our for loop.

Like input, select too supports two-way binding. We saw how changing select updates a model, but the model-to-template binding may not be apparent. To verify that a model to a template binding works, open the *7 Minute Workout* app and verify the duration dropdowns. Each one has a value that is consistent with the model value (30 seconds).

Angular does an awesome job of keeping the model and view in sync using `ngModel`. Change the model and see the view updated; change the view and watch as the model is updated instantaneously.

Now, let's add validation to our form.

The code for the next section is also available for everyone to download on GitHub at <https://github.com/chandermani/angular6byexample>. Checkpoints are implemented as branches in GitHub. The branch to download is as follows: GitHub Branch: `checkpoint4.6` (folder—trainer). Or if you are not using Git, download the snapshot of Checkpoint 4.6 (a ZIP file) from the following GitHub





location: <https://github.com/chandermani/angular6byexample/archive/checkpoint4.6.zip>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time. Again, if you are working along with us as we build the application, be sure and update the `styles.css` file, which we are not discussing here.

Angular validation

As the saying goes, *never trust user input*. Angular has support for validation, including the standard required, min, max, and pattern, as well as custom validators.

ngModel

`ngModel` is the building block that we will use to implement validation. It does two things for us: it maintains the model state and provides a mechanism for identifying validation errors and displaying validation messages.

To get started, we need to assign `ngModel` to a local variable in all of our form controls that we will be validating. In each case, we need to use a unique name for this local variable. For example, for workout name we add `#name="ngModel"` within the `input` tag for that control along with the HTML 5 `required` attribute. The workout name `input` tag should now look like this:

```
<input type="text" name="workoutName" #name="ngModel" class="form-control" id="workout-name" placeholder="Enter workout name. Must be unique." [(ngModel)]="workout.name" required>
```

Continue through the form, assigning `ngModel` to local variables for each of the inputs. Also, add the required attribute for all the required fields.

The Angular model state

Whenever we use `NgForm`, every element within our form, including input, text area, and select, has some states defined on the associated model. `ngModel` tracks these states for us. The states tracked are:

- `pristine`: The value of this is `true` as long as the user does not interact with the input. Any update to the `input` field and `ng-pristine` is set to `false`.
- `dirty`: This is the reverse of `ng-pristine`. This is `true` when the input data has been updated.
- `touched`: This is `true` if the control ever had focus.
- `untouched`: This is `true` if the control has never lost focus. This is just the reverse of `ng-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.

`pristinedirty` OR `toucheduntouched` are useful properties that can help us decide when error labels are shown.

Angular CSS classes

Based on the model state, Angular 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 used when the input is never visited
- `ng-touched`: This is used when the input has focus

To verify it, go back to the `workoutName` input tag and add a template reference variable named `spy` inside the `input` tag: `<input type="text" name="workoutName" #name="ngModel" class="form-control" id="workout-name" placeholder="Enter workout name. Must be unique." [(ngModel)]="workout.name" required #spy>`

Then, below the tag, add the following label:

```
| <label>{{spy.className}}</label>
```

Reload the application and click on the New Workout link in the *Workout Builder*. Before touching anything on the screen, you will see the following

displayed:



Name:
Enter workout name. Must be unique.
`form-control ng-untouched ng-pristine ng-invalid`

Add some content into the Name input box and tab away from it. The label changes to this:



Name:
My New Workout
`form-control ng-dirty ng-valid ng-touched`

What we are seeing here is Angular changing the CSS classes that apply to this

control as the user interacts with it. You can also see these changes by inspecting the `input` element in the developer console.

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; }
```

This draws a red border around any input control that has invalid data.

As you add more validations to the Workout page, you can 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 of validations (before moving on, remove the variable name and label that you just added).

Workout validation

The workout data needs to be validated for a number of conditions.

After taking the step of adding the local variable references for `ngModel` and the required attribute to our `input` fields, we have been able to see how `ngModel` tracks changes in the state of these controls and how it toggles the CSS styles.

Displaying appropriate validation messages

Now, the input needs to have a value; otherwise, the validation fails. But, how can we know if the validation has failed? `ngModel` comes to our rescue here. It can provide the validation state of the particular input. And that gives us what we need to display an appropriate validation message.

Let's go back to the input control for the Workout name. In order to get a validation message to display, we have to first modify the input tag to the following:

```
<input type="text" name="workoutName" #name="ngModel" class="form-control" id="workout-name" placeholder="Enter workout name. Must be unique." [(ngModel)]="workout.name" required>
```

We have added a local variable called `#name` and assigned `ngModel` to it. This is called a template reference variable and we can use it with the following label to display a validation message for the input:

```
<label *ngIf="name.control.hasError('required') && (name.touched)" class="alert alert-danger validation-message">Name is required</label>
```

We are showing the validation message in the event that the name is not provided and the control has been touched. To check the first condition, we retrieve the `hasError` property of the control and see if the error type is `required`. We check to see if the name input has been `touched` because we do not want the message to display when the form first loads for a new workout.



You will notice that we are using a somewhat more verbose style to identify validation errors than is required in this situation. Instead of `name.control.hasError('required')`, we could have used `!name.valid` and it would have worked perfectly fine. However, using the more verbose approach allows us to identify validation errors with greater specificity, which will be essential when we start adding multiple validators to our form controls. We'll look at using multiple validators a little later in this chapter. For consistency, we'll stick with the more verbose approach.

Load the new Workout page (`/builder/workouts/new`) now. Enter a value in the name input box and then delete it. The error label appears as shown in the following

screenshot:

Name:

Enter workout name. Must be unique.

Name is required

Adding more validation

Angular provides several out-of-the-box validators, including:

- required
- minLength
- maxLength
- email
- pattern

For the complete list of out-of-the box validators, see the documentation for the validators class at <https://angular.io/api/forms/Validators>.

We've seen how the `required` validator works. Now, let's look at two of the other out-of-the box validators: `minLength` and `maxLength`. In addition to making it required, we want the title of the workout to be between 5 and 20 characters (we'll look at the `pattern` validator a little later in this chapter).

So, in addition to the `required` attribute we added previously to the title input box, we will add the `minLength` attribute and set it to 5, and add the `maxLength` attribute and set it to 20, like so:

```
|<input type="text" . . . minlength="5" maxlength="20" required>
```

Then, we add another label with a message that will display when this validation is not met:

```
|<label *ngIf="(title.control.hasError('minlength') ||  
|  title.control.hasError('maxlength')) && workout.title.length > 0" class="alert alert-  
| danger validation-message">Title should be between 5 and 20 characters long.</label>
```

Managing multiple validation messages

You'll see that the conditions for displaying the message now test for the length not being zero. This prevents the message from displaying in the event that the control is touched but left empty. In that case, the title required message should display. This message only displays if nothing is entered in the field and we accomplish this by checking explicitly to see if the control's `hasError` type is required:

```
|<label *ngIf="title.control.hasError('required')" class="alert alert-danger validation-
| message">Title is required.</label>
```

Since we are attaching two validators to this input field, we can consolidate the check for the input being touched by wrapping both validators in a div tag that checks for that condition being met:

```
|<div *ngIf="title.touched">
|   . . . [the two validators] . . .
|</div>
```



What we just did shows how we can attach multiple validations to a single input control and also display the appropriate message in the event that one of the validation conditions is not met. However, it's pretty clear that this approach will not scale for more complicated scenarios. Some inputs contain a lot of validations and controlling when a validation message shows up can become complex. As the expressions for handling the various displays get more complicated, we may want to refactor and move them into a custom directive. Creating a custom directive will be covered in detail in chapter 6, Angular 2 Directives in Depth.

Custom validation messages for an exercise

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.

What we will do is add a hidden input box to the form that contains the count of the exercises. We will then bind this to `ngModel` and add a pattern validator that will check to make sure that there is more than one exercise. We will set the value of the input box to the count of the exercises:

```
<input type="hidden" name="exerciseCount" #exerciseCount="ngModel"
ngControl="exerciseCount" class="form-control" id="exercise-count"
[(ngModel)]="workout.exercises.length" pattern="[1-9][0-9]*">
```

Then, we will attach a validation message to it similar to what we just did with our other validators:

```
<label *ngIf="exerciseCount.control.hasError('pattern')" class="alert alert-danger
extended-validation-message">The workout should have at least one exercise!</label>
```

We are not using `ngModel` in its true sense here. There is no two-way binding involved. We are only interested in using it to do custom validation.

Open the new Workout page, add an exercise, and remove it; we should see this error:

My New Workout

The workout should have at least one exercise!

What we did here could have been easily done without involving any model validation infrastructure. But, by hooking our validation into that 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. Most importantly, if our validation fails here, the entire form will be invalidated.



Implementing custom validation the way we just did is not what you would want to do very often. Instead, it will usually make more sense to implement this kind of complicated logic inside a custom directive. We'll cover creating custom directives in detail in chapter 6, Angular 2 Directives in Depth.

One nuisance with our newly implemented `Exercise count` validation is that it shows when the screen for a new `workout` first appears. With this message, we are not able to use `ng-touched` to hide the display. This is because the exercises are being added programmatically and the hidden input we are using to track their count never changes from untouched as exercises are added or removed.

To fix this problem, we need an additional value to check when the state of the exercise list has been reduced to zero, except when the form is first loaded. The only way that situation can happen is if the user adds and then removes exercises from a workout to the point that there are no more exercises. So, we'll add another property to our component that we can use to track whether the remove method has been called. We call that value `removeTouched` and set its initial value to `false`:

```
| removeTouched: boolean = false;
```

Then, in the `remove` method we will set that value to `true`:

```
| removeExercise(exercisePlan: ExercisePlan) {  
  this.removeTouched = true;  
  this.workoutBuilderService.removeExercise(exercisePlan);  
}
```

Next, we will add `removeTouched` to our validation message conditions, like so:

```
| <label *ngIf="exerciseCount.control.hasError('pattern') && (removeTouched)"
```

Now, when we open a new workout screen, the validation message will not display. But if the user adds and then removes all the exercises, then it will display.

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 calling 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 `WorkoutService` and `WorkoutBuilderService`.

`WorkoutService` needs two new methods, `addWorkout` and `updateWorkout`:

```
addWorkout(workout: WorkoutPlan){  
    if (workout.name){  
        this.workouts.push(workout);  
        return workout;  
    }  
}  
  
updateWorkout(workout: WorkoutPlan){  
    for (var i = 0; i < this.workouts.length; i++) {  
        if (this.workouts[i].name === workout.name) {  
            this.workouts[i] = workout;  
            break;  
        }  
    }  
}
```

The `addWorkout` method does a basic check on the workout name and then pushes the workout into the `workouts` array. Since there is no backing store involved, if we refresh the page, the data is lost. We will fix this in the next chapter where we persist the data to a server.

The `updateWorkout` method 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 already tracking the context in which workout construction is going on:

```
save(){  
    let workout = this.newWorkout ?  
        this._workoutService.addWorkout(this.buildingWorkout) :  
        this._workoutService.updateWorkout(this.buildingWorkout);  
    this.newWorkout = false;  
    return workout;  
}
```

The `save` method calls either `addWorkout` or `updateWorkout` in the `Workout` service 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 the `workout` component and learn more about the `form` directive!

Before we look at `NgForm` in more detail, let's add the save method to `workout` to save the workout when the `Save` button is clicked on. Add this code to the `Workout` component:

```
| save(formWorkout:any){  
|   if (!formWorkout.valid) return;  
|   this.workoutBuilderService.save();  
|   this.router.navigate(['/builder/workouts']);  
| }
```

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.

More on NgForm

Forms in Angular have a different role to play as compared to traditional forms that post data to the server. If we go back and look again at the form tag, we will see that it is missing the standard action attribute. The standard form behavior of posting data to the server using full-page post-back does not make sense with an SPA framework such as Angular. In Angular, all server requests are made through asynchronous invocations originating from directives or services.

Under the hood, Angular is also turning off the browser's inbuilt validation. As you have seen in this chapter, we are still using validation attributes such as required that look the same as native HTML validation attributes. However, as the Angular documentation explains, inside an Angular form "Angular uses directives to match these attributes with validator functions in the framework." See <https://angular.io/guide/form-validation#template-driven-validation>.

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, you can add an `ngNoForm` attribute to the `form` element, but this will definitely cause a full-page refresh. You can also turn on the browser's inbuilt validation by adding the `ngNativeValidate` attribute. We'll explore the specifics of the `NgForm` API a little later in this chapter when we look at saving the form and implementing validation.

The state of the `FormControl` objects within the form is being monitored by `NgForm`. If any of them are invalid, then `NgForm` sets the entire form to invalid. In this case, we have been able to use `NgForm` to determine that one or more of the `FormControl` objects is invalid and therefore the state of the form as a whole is invalid too.

Let's look at one more issue before we finish this chapter.

Fixing the saving of forms and validation messages

Open a new Workout 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 binding for the name input element, it looks like this:
*ngIf="name.control?.hasError('required') && name.touched"

Remember that, earlier in the chapter, we explicitly disabled showing validation messages until the user has touched the input control. The same issue has come back to bite us and we need to fix it now.

We do not have a way to explicitly change the touched state of our controls to untouched. Instead, we will resort to a little trickery to get the job done. We'll introduce a new property called `submitted`. Add it at the top of the `workout` class definition and set its initial value to `false`, like so: `submitted: boolean = false;`

The variable will be set to `true` on the Save button click. Update the save implementation by adding the highlighted code:

```
| save(formWorkout){  
|   this.submitted = true;  
|   if (!formWorkout.valid) return;  
|   this._workoutBuilderService.save();  
|   this.router.navigate(['/builder/workouts']);  
| }
```

However, how does this help? Well, there is another part to this fix that requires us to change the error message for each of the controls we are validating. The expression now changes to:

```
| *ngIf="name.control.hasError('required') && (name.touched || submitted)"
```

With this fix, the error message is shown when the control is touched or the form submit button is pressed (`submitted` is `true`). This expression fix now has to be applied to every validation message where a check appears.

If we now open the new Workout page and click on the Save button, we should see all validation messages on the input controls:

Workout Title

The workout should have at least one exercise!

Name:

Enter workout name. Must be unique.

Name is required

Title:

What would be the workout title?

Title is required.

Description:

Enter workout description.

Rest Time (in seconds):

0

Rest time is required

Total Exercises: 0

Total Duration:

Save

Reactive forms

The other type of form that Angular supports is called **reactive** forms. **Reactive forms** start with a model that is constructed in a component class. With this approach, we use the **form builder API** to create a form in code and associate it with a model.

Given the minimal code we have to write to get template-driven forms working, why and when should we consider using reactive forms? There are several situations in which we might want to use them. These include cases where we want to take programmatic control of creating the form. This is especially beneficial, as we will see, when we are trying to create form controls dynamically based on data we are retrieving from the server.

If our validation gets complicated, it is often easier to handle it in code. Using reactive forms, we can keep this complicated logic out of the HTML template, making the template syntax simpler.

Another significant advantage of reactive forms is that they make unit-testing the form possible, which is not the case with **template-driven forms**. We can simply instantiate our form controls in our tests and then test them outside the markup on our page.

Reactive forms use three new form directives that we haven't discussed before: `FormGroup`, `FormControl`, and `FormArray`. These directives allow the form object that is constructed in code to be tied directly to the HTML markup in the template. The form controls that are created in the component class are then directly available in the form itself. Technically speaking, this means that we don't need to use `ngModel` (which is integral to template-driven forms) with reactive forms (although it can be used). The overall approach is a cleaner and less cluttered template with more focus on the code that drives the form. Let's get started with building a reactive form.

Getting started with reactive forms

We'll make use of reactive forms to build the form to add and edit Exercises. Among other things, this form will allow the user to add links to exercise videos on YouTube. And since they can add any number of video links, we will need to be able to add controls for these video links dynamically. This challenge will present a good test of how effective reactive forms can be in developing more complex forms. Here is how the form will look:

Personal Trainer

Home + New Workout + New Exercise

Abdominal Crunches

Name:

Title:

Description:

Exercise Steps:
Exercise Audio:

Exercise Image (Will be scaled to: 540 X 360 px): 

Videos: [Add Video](#) [Save](#)

To get started, open `workout-builder.module.ts` and add the following import:

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
...
@NgModule({
  imports: [
```

```
    CommonModule,
    FormsModule,
ReactiveFormsModule,
    SharedModule,
    workoutBuilderRouting
],
```

ReactiveFormsModule contains what we will need to build reactive forms.

Next, copy exercise-builder-service.ts from the workout-builder/builder-services folder under trainer/src/app in checkpoint 4.6 and import it into workout-builder.module.ts:

```
| import { ExerciseBuilderService } from "./builder-services/exercise-builder-service";
```

Then, add it as an additional provider to the providers array in that same file:

```
@NgModule({
  ...
  providers: [
    WorkoutBuilderService,
    WorkoutResolver,
    ExerciseBuilderService,
    ExerciseResolver
  ]
})
```

 You will notice here that we also have added ExerciseResolver as a provider. We won't be covering that here, but you should copy it from the exercise folder as well and also copy the updated workout-builder-routing.module.ts, which adds it as a route guard for the navigation to ExerciseComponent.

Now, open exercise.component.ts and add the following import statement:

```
| import { Validators, FormArray, FormGroup, FormControl, FormBuilder } from
|   '@angular/forms';
```

This brings in the following, which we will use to construct our form:

- FormBuilder
- FormGroup
- FormControl
- FormArray

Finally, we inject FormBuilder (as well as Router, ActivatedRoute, and ExerciseBuilderService) into the constructor of our class:

```
constructor(
  public route: ActivatedRoute,
  public router: Router,
  public exerciseBuilderService: ExerciseBuilderService,
  public formBuilder: FormBuilder
```

```
|  ) {}
```

With these preliminary steps out of the way, we can now go ahead and start building out our form.

Using the FormBuilder API

The `FormBuilder` API is the foundation for reactive forms. You can think of it as a factory for turning out the forms we are constructing in our code. Go ahead and add the `ngOnInit` life cycle hook to your class, as follows:

```
ngOnInit() {
  this.sub = this.route.data
    .subscribe(
      (data: { exercise: Exercise }) => {
        this.exercise = data.exercise;
      }
    );
  this.buildExerciseForm();
}
```

When `ngOnInit` fires, it will extract the data for an existing or new `exercise` from the route data that has been retrieved and returned by `ExerciseResolver`. This is the same pattern we followed with initializing the `workout` component.

Now, let's implement the `buildExerciseForm` method by adding the following code:

```
buildExerciseForm(){
  this.exerciseForm = this.formBuilder.group({
    'name': [this.exercise.name, [Validators.required,
      AlphaNumericValidator.invalidAlphaNumeric]],
    'title': [this.exercise.title, Validators.required],
    'description': [this.exercise.description, Validators.required],
    'image': [this.exercise.image, Validators.required],
    'nameSound': [this.exercise.nameSound],
    'procedure': [this.exercise.procedure],
    'videos': this.addVideoArray()
  })
}
```

Let's examine this code. To start with, we are using the injected instance of `FormBuilder` to construct the form and assign it to a local variable, `exerciseForm`. Using `formBuilder.group`, we add several form controls to our form. We add each of them by a simple key/value mapping:

```
| 'name': [this.exercise.name, Validators.required],
```

The left side of the mapping is the name of the FormControl, and the right is an array containing as its first element the value of the control (in our case, the corresponding element on our exercise model) and the second a validator (in this

case, the out-of-the-box required validator). Nice and neat! It's definitely easier to see and reason about our form controls by setting them up outside the template.

We can not only build up `FormControls` in our form this way, but also add `FormControlGroups` and `FormControlArray`, which contain `FormControls` within them. This means we can create complex forms that contain nested input controls. In our case, as we have mentioned, we are going to need to accommodate the possibility of our users adding multiple videos to an exercise. We can do this by adding the following code:

```
| 'videos': this.addVideoArray()
```

What we are doing here is assigning a `FormArray` to `videos`, which means we can assign multiple controls in this mapping. To construct this new `FormArray`, we add the following `addVideoArray` method to our class:

```
addVideoArray(){
  if(this.exercise.videos){
    this.exercise.videos.forEach((video : any) => {
      this.videoArray.push(new FormControl(video, Validators.required));
    });
  }
  return this.videoArray;
}
```

This method constructs a `FormControl` for each video; each is then added each to a `FormArray` that is assigned to the `videos` control in our form.

Adding the form model to our HTML view

So far, we have been working behind the scenes in our class to construct our form. The next step is to wire up our form to the view. To do this, we use the same controls that we used to build the form in our code: `formGroup`, `FormControl`, and `formArray`.

Open `exercise.component.html` and add a `form` tag as follows:

```
| <form class="row" [formGroup]="exerciseForm" (ngSubmit)="onSubmit(exerciseForm)">
```

Within the tag, we are first assigning the `exerciseForm` that we just built in code to `formGroup`. This establishes the connection between our coded model and the form in the view. We also wire up the `ngSubmit` event to an `onSubmit` method in our code (we'll discuss this method a little later).

Adding form controls to our form inputs

Next, we start constructing the inputs for our form. We'll start with the input for the name of our exercise:

```
| <input name="name" formControlName="name" class="form-control" id="name"  
| placeholder="Enter exercise name. Must be unique.">
```

We assign the name of our coded form control to formControlName. This establishes the link between the control in our code and the input field in the markup. Another item of interest here is that we do not use the required attribute.

Adding validation

The next thing that we do is add a validation message to the control that will display in the event of a validation error: <label

```
*ngIf="exerciseForm.controls['name'].hasError('required') &&
(exerciseForm.controls['name'].touched || submitted)" class="alert alert-danger
validation-message">Name is required</label>
```

Notice that this markup is very similar to what we used in template-driven forms for validation, except that the syntax for identifying the control is somewhat more verbose Again, it checks the state of the `hasError` property of the control to make sure it is valid.

But wait a minute! How can we validate this input? Haven't we removed the `required` attribute from our tag? This is where the control mappings that we added in our code come into play. If you look back at the code for the form model, you can see the following mapping for the `name` control: 'name': [this.exercise.name, Validators.required],

The second element in the mapping array assigns the required validator to the name form control. This means that we don't have to add anything to our template; instead, the form control itself is attached to the template with a required validator. The ability to add a validator in our code enables us to conveniently add validators outside our template. This is especially useful when it comes to writing custom validators with complex logic behind them.

Adding dynamic form controls

As we mentioned earlier, the Exercise form that we are building requires that we allow the user to add one or more videos to the exercise. Since we don't know how many videos the user may want to add, we will have to build the `input` fields for these videos dynamically as the user clicks on the `Add Video` button. Here's how it will look:

Videos:

Xyd_fa5zoEU	
MKmrqcoCZ-M	
Add a related youtube video identified.	

Add Video

We have already seen the code in our component class that we use to do this. Now, let's take a look at how it is implemented in our template.

We first use `ngFor` to loop through our list of videos. Then, we assign the index in our videos to a local variable, `i`. No surprises so far:

```
<div *ngFor="let video of videoArray.controls; let i=index" class="form-row align-items-center">
```

Inside the loop, we do three things. First, we dynamically add a video `input` field for each of the videos currently in our exercise:

```
<div class="col-sm-10">
  <input type="text" class="form-control" [FormControlName]="i" placeholder="Add a related youtube video identified."/>
</div>
```

Next, we add a button to allow the user to delete a video:

```
<span class="btn alert-danger" title="Delete this video." (click)="deleteVideo(i)">
  <span class="ion-ios-trash-outline"></span>
</span>
```

We bind a `deleteVideo` method in our component class to the button's `click` event

and pass to it the index of the video being deleted.

We then add a validation message for each of the video `input` fields:

```
| <label *ngIf="exerciseForm.controls['videos'].controls[i].hasError('required') &&
| (exerciseForm.controls['videos'].controls[i].touched || submitted)" class="alert alert-
| danger validation-message">Video identifier is required</label>
```

The validation message follows the same pattern for displaying the message that we have used elsewhere in this chapter. We drill into the `exerciseFormControls` group to find the particular control by its index. Again, the syntax is verbose but easy enough to understand.

Saving the form

The final step in building out our reactive form is to handle saving the form. When we constructed the form tag earlier, we bound the `ngSubmit` event to the following `onSubmit` method in our code:

```
onSubmit(formExercise: FormGroup) {
  this.submitted = true;
  if (!formExercise.valid) { return; }
  this.mapFormValues(formExercise);
  this.exerciseBuilderService.save();
  this.router.navigate(['/builder/exercises']);
}
```

This method sets `submitted` to `true`, which will trigger the display of any validation messages that might have been previously hidden because the form had not been touched. It also returns without saving in the event that there are any validation errors on the form. If there are none, then it calls the following `mapFormValues` method, which assigns the values from our form to the `exercise` that will be saved:

```
mapFormValues(form: FormGroup) {
  this.exercise.name = form.controls['name'].value;
  this.exercise.title = form.controls['title'].value;
  this.exercise.description = form.controls['description'].value;
  this.exercise.image = form.controls['image'].value;
  this.exercise.nameSound = form.controls['nameSound'].value;
  this.exercise.procedure = form.controls['procedure'].value;
  this.exercise.videos = form.controls['videos'].value;
}
```

It then calls the `save` method in `ExerciseBuilderService` and routes the user back to the exercise list screen (remember that any new exercise will not display in that list because we have not yet implemented data persistence in our application).

We hope this makes it clear; reactive forms offer many advantages when we are trying to build more complicated forms. They allow programming logic to be removed from the template. They permit validators to be added to the form programmatically. And, they support building forms dynamically at runtime.

Custom validators

Now, we'll take a look at one more thing before we conclude this chapter. As anyone who has worked on building web forms (either in Angular or any other web technology) knows, we are often called on to create validations that are unique to the application we are building. Angular provides us with the flexibility to enhance our reactive form validation by building custom validators.

In building our exercise form, we need to be sure about what is entered, as a name contains only alphanumeric characters and no spaces. This is because when we get to storing the exercises in a remote data store, we are going to use the name of the exercise as its key. So, in addition to the standard required field validator, let's build another validator that checks to make sure that the name entered is in alphanumeric form only.

Creating a custom control is quite straightforward. In its simplest form, an Angular custom validator is a function that takes a control as an input parameter, runs the validation check, and returns true or false. So, let's start by adding a TypeScript file with the name `alphanumeric-validator.ts`. In that file, first import `FormControl` from `@angular/forms`, then add the following class to that file:

```
export class AlphaNumericValidator {  
  static invalidAlphaNumeric(control: FormControl): { [key: string]: boolean } {  
    if (control.value.length && !control.value.match(/^[a-zA-Z0-9]+$/i)) {  
      return {invalidAlphaNumeric: true};  
    }  
    return null;  
  }  
}
```

The code follows the pattern for creating a validator that we just mentioned. The only thing that may be a little surprising is that it returns true when the validation fails! As long as you are clear on this one quirk, you should have no problem writing your own custom validator.

Integrating a custom validator into our forms

So how do we plug our custom validator into our form? If we are using reactive forms, the answer is pretty simple. We add it just like a built-in validator when we build our form in code. Let's do that. Open `exercise.component.ts` and first add an import for our custom validator:

```
| import { AlphaNumericValidator } from '../alphanumeric-validator';
```

Then, modify the form builder code to add the validator to the `name` control:

```
buildExerciseForm(){
  this.exerciseForm = this._formBuilder.group({
    'name': [this.exercise.name, [Validators.required,
      AlphaNumericValidator.invalidAlphaNumeric]],
    . . . [other form controls] . . .
  });
}
```

Since the `name` control already has a required validator, we add `AlphaNumericValidator` as a second validator using an array that contains both validators. The array can be used to add any number of validators to a control.

The final step is to incorporate the appropriate validation message for the control into our template. Open `workout.component.html` and add the following label just below the label that displays the message for the required validator:

```
<label *ngIf="exerciseForm.controls['name'].hasError('invalidAlphaNumeric') &&
(exerciseForm.controls['name'].touched || submitted)" class="alert alert-danger
validation-message">Name must be alphanumeric</label>
```

The exercise screen will now display a validation message if a non-alphanumeric value is entered in the name input box:

Name:

a workout

Name must be alphanumeric

As we hope you can see, reactive forms give us the ability to add custom validators to our forms in a straightforward manner that allows us to maintain the validation logic in our code and easily integrate it into our templates.



You may have noticed that in this chapter, we have not covered how to use custom validators in template-driven forms. That is because implementing them requires the additional step of building a custom directive. We'll cover that in chapter 6, Angular 2 Directives in Depth.

Configuration options for running validation

Before we move on from validation, there is one more topic to cover and that is configuration options for running the validations. So far, we have been using the default option, which runs validation checks on every input event. However, you have the choice of configuring them to run either on "blur"(that is when the user leaves an input control) or when the form is submitted. You can set this configuration at the form level or on a control-by-control basis.

For example, we might decide that to avoid the complexity of handling missing exercises in the workout form, we will set that form to validate only upon submit. We can set this by adding the following highlighted assignment of `NgFormOptions` to the form tag:

```
| <form #f="ngForm" name="formWorkout" (ngSubmit)="save(f.form)" [ngFormOptions]="  
| {updateOn: 'submit'}" class="row">
```

This instructs Angular to run our validations only upon ~~submit~~. Try it and you'll see that no validations appear when you make entries into the form. Leave the form blank and press the Save button, and you will see the validation messages appear. Taking this approach, of course, means that there are no visual cues to the user regarding validation until they press the Save button.

There are also a couple of other unintended side effects to using this approach in our form. The first is that the title no longer updates at the top of the screen as we type into the title input box. That value will only be updated when we press Save. Second, you will also see a validation message appear if you add one or more workouts and then remove all of them. This is because of the special conditions we set up for this control, which cause it to fire outside the normal validation flow.

So, maybe we should take a different approach. Angular provides the option of implementing more fine-grained control of the validation flow by allowing us to make such configurations at the control level using `ngModelOptions`. For example,

let's remove the `ngFormOptions` assignment from the form tag and modify the title input control to add `ngModelOptions` as follows:

```
<input type="text" name="title" class="form-control" #title="ngModel" id="workout-
title" placeholder="What would be the workout title?" [(ngModel)]="workout.title"
[ngModelOptions]="{updateOn: 'blur'}" minlength="5" maxlength="20" required>
```

You'll then notice that as you type the title into the input box, it does not update the title on the screen until you move off it (which triggers the `updateOn` event):

[`ngModelOptions`]`="{updateOn: 'blur'}`"

Workout Title

Title does not update and validation message does not appear while user types within title input box.

Name:
Enter workout name. Must be unique.

Title:
7Min

Description:
Enter workout description.

7Min

Title updates and validation message appears after user leaves title input box.

Name:
Enter workout name. Must be unique.

Title:
7Min

Title should be between 5 and 20 characters long

Description:
Enter workout description.

As you will remember, the default option caused the title to update with every keystroke. This is a contrived example but it illustrates how the differences in these configurations work.

You probably don't see the need to use the on blur setting here. But, incase where you may be doing validation by calling an external data store, this approach could be helpful in limiting the number of calls that are being made.

And making such remote calls is exactly what we will be doing in [chapter 6](#), *Angular Directives in Depth*, when we implement a custom directive. The **directive** will be checking for duplicate names that already exist in our remote data store. So, let's remove this configuration from the title input control and place it instead on the name input control, like so:

```
|<input type="text" name="workoutName" #name="ngModel" class="form-control" id="workout-  
|  name" placeholder="Enter workout name. Must be unique." [(ngModel)]="workout.name"  
|  [ngModelOptions]="{updateOn: 'blur'}" required>
```

We also can set the validation timing options within a reactive form. From what we have already learned about reactive forms, you will not be surprised to learn that we will be applying these settings in our code rather than the template. For example, to set them for a form group you use the following syntax:

```
| new FormGroup(value, {updateOn: 'blur'}));
```

We can also apply them to individual form controls and that is what we will do in the case of our exercise form. Like the workout form, we will want to be able to validate the uniqueness of the name by making a remote call. So, we will want to limit the validation checking in a similar manner. We'll do that by adding the following to the code that creates the name form control:

```
buildExerciseForm() {  
    this.exerciseForm = this.formBuilder.group({  
        'name': [  
            this.exercise.name,  
            {  
                updateOn: 'blur',  
                validators: [Validators.required,  
AlphaNumericValidator.invalidAlphaNumeric]  
            }  
        ],  
        ...  
    });  
}
```

Note that we are putting the setting, along with the **validators array**, in the options object inside a pair of curly braces.

Summary

We now have a *Personal Trainer* app. The process of converting a specific *7 Minute Workout* app to a generic *Personal Trainer* app helped us learn a number of new concepts.

We started the chapter by defining the new app requirements. Then, we designed the model as a shared service.

We defined some new views and corresponding routes for the *Personal Trainer* app. We also used both child and asynchronous routing to separate out *Workout Builder* from the rest of the app.

We then turned our focus to workout building. One of the primary technological focuses in this chapter was on Angular forms. The *Workout Builder* employed a number of form input elements and we implemented a number of common form scenarios using both template-driven and reactive forms. We also explored Angular validation in depth, and implemented a custom validator. We also covered configuring the timing options for running validation.

The next chapter is all about client-server interaction. The workouts and exercises that we create need to be persisted. In the next chapter, we build a persistence layer, which will allow us to save workout and exercise data on the server.

Before we conclude this chapter, here is a friendly reminder. If you have not completed the exercise building routine for *Personal Trainer*, go ahead and do it. You can always compare your implementation with what has been provided in the companion code base. There are also things you can add to the original implementation, such as file uploads for the exercise image, and once you are more familiar with client-server interaction, a remote check to determine whether the YouTube videos actually exist.

15-Oct-2018

Supporting Server Data Persistence

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 did not persist anywhere. We need to fix this.

Seldom are applications self-contained. Any consumer app, irrespective of its size, has parts that interact with elements outside its boundary. 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 chapter explores the constructs that Angular 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 chapter include the following:

- **Provisioning a backend to persist workout data:** We set up a [MongoLab](#) account and use its [Data API](#) to access and store workout data.
- **Understanding the Angular HttpClient:** The `HttpClient` allows us to interact with a server over HTTP. You'll learn how to make all types of `GET`, `POST`, `PUT`, and `DELETE` requests with the `HttpClient`.
- **Implementing the loading and saving of workout data:** We use the `HttpClient` to load and store workout data in the [MongoLab](#) databases.
- **Two ways in which we can use the HttpClient's XMLHttpRequest:** Either [Observables](#) or [with promises](#).
- **Using RxJS and Observables:** To subscribe to and query streams of data.
- **Using promises:** In this chapter, we will see how to use promises as part of HTTP invocation and response.
- **Working with cross-domain access:** As we are interacting with a MongoLab server in a different domain, you will learn about browser restrictions on cross-domain access. You will also learn how [JSONP](#) and CORS help us make cross-domain access easy and about Angular JSONP support.

Let's set the ball rolling.

Angular 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 JavaScript apps, we depend on the AJAX request/response mechanism to communicate with the server. To support AJAX-based communication, Angular provides the Angular HttpClient module. Before we delve into the HttpClient module, we need to set up our server platform that stores the data and allows us to manage it.

Setting up the persistence store

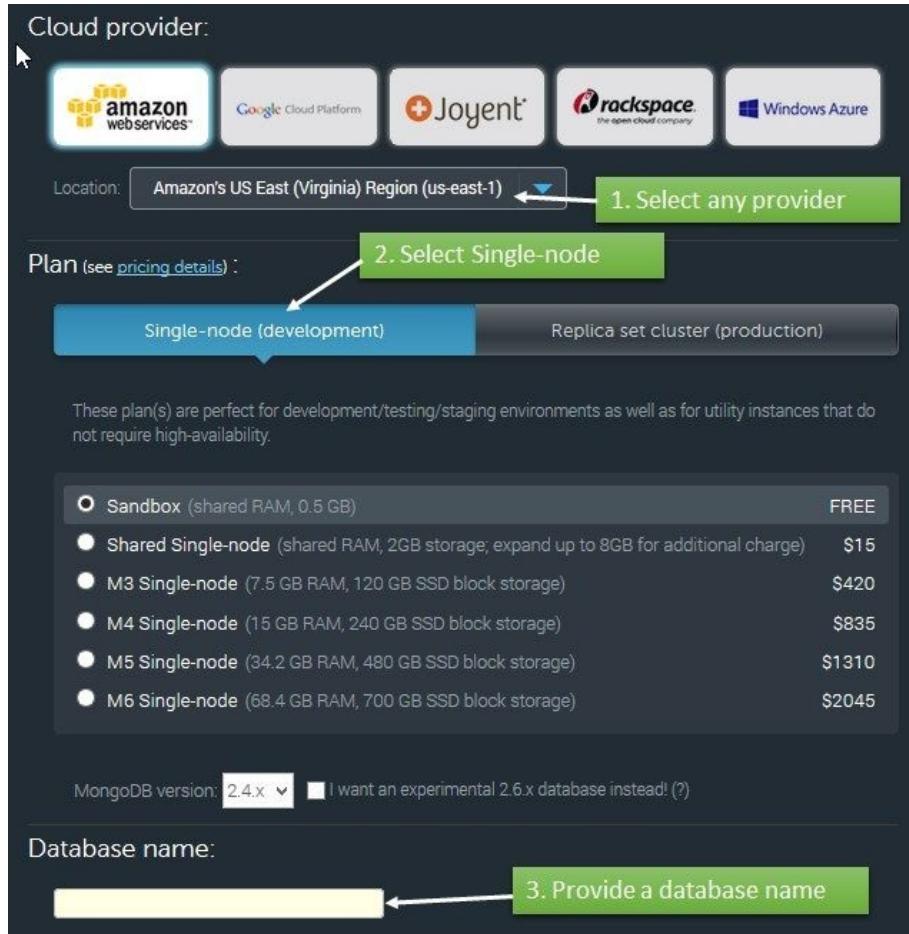
For data persistence, we use a document database called MongoDB (<https://www.mongodb.com/>), hosted over MongoLab (<https://www.mlab.com/>), as our data store. The reason we zeroed in on MongoLab is that it provides an interface to interact with the database directly. This saves us the effort of setting up server middleware to support MongoDB interaction.



*It is never a good idea to expose the data store/database directly to the client. But in this case, since our primary aim is to learn about Angular and client-server interaction, we take this liberty and 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 check out <http://nobackend.org/>.*

Our first task is to provision an account on MongoLab and create a database:

1. Go to <https://mlab.com> and sign up for an mLab account by following the instructions on the website
2. Once the account is provisioned, log in and create a new Mongo database by clicking on the Create New button on the home page
3. On the database creation screen, you need to make some selections to provision the database. See the following screenshot to select the free database tier and other options:



4. Create the database and make a note of the database name that you create
5. 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 called **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 two previously mentioned document collections. We may not even be using the document database in its true sense.*

Once the collections are added, add yourself as a user to the database from the Users tab.

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. At the same time, click on the button below the API key to Enable Data API access. This is disabled by default.

The data store 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` in the `trainer/db` folder for checkpoint 5.1 from the companion code base. 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 the following:

NAME	DOCUMENTS	CAPPED?	SIZE
exercises	12	false	19.30 KB
workouts	1	false	8.97 KB

Everything has been set up now, so let's start our discussion of the `HttpClient` module and implement workout/exercise persistence for the *Personal Trainer* app.

The basics of the HttpClient module

At the core of the `HttpClient` module is the `HttpClient`. It performs HTTP requests using `XMLHttpRequest` as the default backend (JSONP is also available, as we will see later in this chapter). It supports requests such as `GET`, `POST`, `PUT`, and `DELETE`. In this chapter, we will use the `HttpClient` to make all of these types of requests. As we will see, the `HttpClient` makes it easy to make these calls with a minimal amount of setup and complexity. None of this terminology will come as a surprise to anyone who has previously worked with Angular or built JavaScript applications that communicate with a backend data store.

However, there is a substantial change in the way Angular handles HTTP requests. Calling a request now returns an Observable of HTTP responses. It does so by using the RxJS library, which is a well-known open source implementation of the asynchronous Observable pattern.



You can find the RxJS project on GitHub at <https://github.com/Reactive-Extensions/RxJS>. The site indicates that the project is being actively developed by Microsoft in collaboration with a community of open source developers. We will not be covering the asynchronous Observable pattern in great detail here, and we encourage you to visit that site to learn more about the pattern and how RxJS implements it. The version of RxJS that Angular is using is beta 5.

In the simplest of terms, using Observables allows a developer to think about the data that flows through an application as streams of information that the application can dip into and use whenever it wants. These streams change over time, which allows the application to react to these changes. This quality of Observables provides a foundation for **functional reactive programming (FRP)**, which fundamentally shifts the paradigm for building web applications from imperative to reactive.

The RxJS library provides `operators` that allow you to subscribe to and query these data streams. Moreover, you can easily mix and combine them, as we will see in this chapter. Another advantage of Observables is that it is easy to cancel or unsubscribe from them, making it possible to seamlessly handle errors inline.

While it is still possible to use promises, the default method in Angular uses Observables. We will also cover promises in this chapter.

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 previous chapter, the initial set of workouts and exercises was hardcoded in the `workoutService` implementation. Let's see how to load this data from the server first.

Loading exercise and workout data

Earlier in this chapter, we seeded our database with a data form, the `seed.js` file. We now need to render this data in our views. The MongoLab Data API is going to help us here.



The MongoLab Data API uses an API key to authenticate access requests. 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 chapter. Remember that the key is always provided to a user and associated with their 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 auto-generated.
- `/<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 Data API interface, visit the MongoLab Data API documentation at <http://docs.mlab.com/data-api>.

Now we are in a position to start implementing exercise/workout list pages.



The code that we are starting with in this chapter is `checkpoint 4.6 (folder: trainer)` in the GitHub repository for this book. It is available on GitHub (<https://github.com/chandermani/angular6byexample>). Checkpoints are implemented as branches in GitHub. If you are not using Git, download the snapshot of checkpoint 4.6 (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/tree/checkpoint4.6>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

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`. But before we can do that, we have to set up our service to work with Angular's `HttpClient` module.

Adding the HttpClient module and RxJS to our project

The Angular HttpClient module is included in the Angular bundles that you have already installed. To use it, we need to import it into `app.module.ts`, like so (make sure that the import follows `BrowserModule`):

```
import { HttpClientModule } from '@angular/common/http';
. . .

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule,
    . . .
  ]
})
```

We also need an external third-party library: **Reactive Extensions for JavaScript (RxJS)**. RxJS implements the Observable pattern and it is used by Angular with the HttpClient module. It is included in the Angular bundles that are already part of our project.

Updating workout-service to use the HttpClient module and RxJS

Open `workout.service.ts` from `trainer/src/app/core`. In order to use the `HttpClient` and RxJS within the `WorkoutService`, we need to add the following imports to that file:

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { catchError } from 'rxjs/operators';
```

We are importing the `HttpClient` module along with `Observable` from RxJS and one additional RxJS operator: `catchError`. We'll see how this operator is used as we work through the code in this section.

In the class definition, add the following properties, which include a `workout` property and ones that set the URL for the collections in our Mongo database and the key to that database as well as another property: `params`, which sets up the API key as a query string for API access:

```
workout: WorkoutPlan;
collectionsUrl = "https://api.mongolab.com/api/1/ databases/<dbname>/collections";
apiKey = <key>
params = '?apiKey=' + this._apiKey;
```

Replace the `<dbname>` and `<key>` tokens with the database name and API key of the database that we provisioned earlier in the chapter.

Next, inject the `HttpClient` module into the `WorkoutService` constructor using the following line of code:

```
constructor(public http: HttpClient) {  
}
```

Then change the `getExercises()` method to the following:

```
getExercises() {
    return this.http.get<ExercisePlan>(this.collectionsUrl + '/exercises' +
    this.params)
        .pipe(catchError(WorkoutService.handleError));
}
```

If you are used to working with promises for asynchronous data operations, what you see here will look different. Instead of a promise that has a call to `then()` chained to it, what happens here is that the `http.get` method returns an Observable from the RxJS library. Notice that we are also setting the response to be of type `<ExercisePlan>` to make explicit to our upstream callers what type of Observable is being returned from our HTTP GET call.



Returning an Observable is the default response when using the `HttpClient` module's `get` method. The Observable can, however, be converted to a promise. And, as we will see later in this chapter, the option to return JSONP also exists.

Before we move on, there is one more thing to touch upon in this code. Notice that we are using a pipe method to add a `catchError` operator. This operator accepts a method, `handleError`, for handling a failed response. The `handleError` method takes the failed response as a parameter. We log the error to the console and use `observable.throw` to return the error to the consumer:

```
| static handleError (error: Response) {  
|   console.error(error);  
|   return Observable.throw(error || 'Server error');  
| }
```

To be clear, this is not production code, but it will give us the opportunity to show how to write code upstream to handle errors that are generated as part of data access.



It is important to understand that at this stage no data is flowing through the Observable until there is a subscription to it. This can bring about a gotcha moment for things such as adds and updates if you are not careful to add subscriptions to your Observables.

Modifying `getWorkouts()` to use the `HttpClient` module

The change in the code for retrieving workouts is almost identical to that for the exercises:

```
getWorkouts() {
    return this.http.get<WorkoutPlan[]>(this.collectionsUrl + '/workouts' +
  this.params)
    .pipe(catchError(WorkoutService.handleError));
}
```

Again we are specifying the type of Observable—in this case `<WorkoutPlan[]>`—that will be returned by our HTTP GET call and using `pipe` to add a `catchError` operator.

Now that the `getExercises` and `getWorkouts` methods are updated, we need to make sure that they work with the upstream callers.

Updating the workout/exercise list pages

The exercise and workout list pages (as well as `LeftNavExercises`) call either the `getExercises` OR `getWorkouts` method in `model.ts`. In order to get these working with the remote calls that are now being made using the `HttpClient` module, we need to modify those calls to subscribe to the Observable that is being returned by the `HttpClient` module. So, update the code in the `ngOnInit` method in `exercises.component.ts` to the following:

```
ngOnInit() {
  this.workoutService.getExercises()
    .subscribe(
      exercises => this.exerciseList = exercises,
      (err: any) => console.error
    );
}
```

Our method now subscribes to the Observable that is being returned by the `getExercises` method; at the point when the response arrives, it assigns the results to `exerciseList`. If there is an error, it assigns it to a `console.error` call that displays the error in the console. All of this is now being handled asynchronously using the `HttpClient` module with RxJS.

Go ahead and make similar changes to the `ngOnInit` methods in `workouts.component.ts` and `left-nav-exercises.component.ts`.

Refresh the workout/exercise list page and the workout and exercise data will be loaded from the database server.

 *Look at the complete implementation in checkpoint 5.1 in the GitHub repository if you are having difficulty in retrieving/showing data. Note that in this checkpoint, we have disabled navigation links to the workout and exercise screens because we still have to add the Observable implementation to them. We'll do that in the next section. Also remember to replace the database name and API key before you run the code from checkpoint 5.1. If you are not using Git, download the snapshot of checkpoint 5.1 (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/tree/checkpoint5.1>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.*

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 at how these calculations were implemented. The `workoutPlan` service (in `model.ts`) defines a `totalWorkoutDuration` method that does the math for this.

The difference is in the workout array that is bound to the view. In the previous chapter, 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 to our model class objects and returning them 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, we will see that they do match and hence mapping becomes unnecessary.

Mapping a server response to the model data becomes imperative if:

- Our model defines any methods
- A stored model 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)

The `WorkoutPlan` service 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 the model and server data are as follows:

- The model defines the `totalWorkoutDuration` method.

- The `exercises` array representation also differs. The `exercises` array of the model contains the full `Exercise` object, while the server data stores just the exercise identifier or name.

This clearly means that loading and saving a workout **requires** model mapping.

The way we will do this is by adding another operator to transform the Observable response object. So far, we have only returned a plain JavaScript object as the response. The nice thing is that the pipe method that we used to add error handling also allows us to add additional operators that we can use to transform the JavaScript object into the `WorkoutPlan` type in our model.

Let's rewrite the `getWorkouts` method in the `workout-service.ts` file to the following:

```
getWorkouts(): Observable<WorkoutPlan[]> {
    return this.http.get<WorkoutPlan[]>(this.collectionsUrl + '/workouts' +
    this.params)
        .pipe(
            map((workouts: Array<any>) => {
                const result: Array<WorkoutPlan> = [];
                if (workouts) {
                    workouts.forEach((workout) => {
                        result.push(
                            new WorkoutPlan(
                                workout.name,
                                workout.title,
                                workout.restBetweenExercise,
                                workout.exercises,
                                workout.description
                            ));
                    });
                }
                return result;
            }),
            catchError(this.handleError<WorkoutPlan[]>('getWorkouts', []))
        );
}
```

We have added a `map` operator that transforms this Observable into one made up of `WorkoutPlan` objects. Each `WorkoutPlan` object (we have only one at the moment) will then have the `totalWorkoutDuration` method that we need.

Looking at the code you can see that we operate on the JSON results `HttpClient` response, which is why we are using the `<any>` type. And then we create a typed array of `WorkoutPlans` and iterate through the first array using a fat arrow `forEach` function, assigning each JavaScript object to a `WorkoutPlan` object.

We return the results of these mappings to the callers that subscribe to them,

`workouts.component.ts` in this case. We have also updated the `catchError` operator with a new `handleError` method which you can find in `checkpoint 5.2`. The callers do not need to make any changes to the code they use to subscribe to our workouts Observable. Instead, the model mapping can take place at one spot in the application and then be used throughout it.

If you rerun the application, you will see that the total number of seconds now displays properly:



Checkpoint 5.2 in the GitHub repository contains the working implementation for what we have covered so far. The GitHub branch is `checkpoints5.2` (folder: `trainer`).

Loading exercise and workout data from the server

Just as we fixed the `getWorkouts` implementation in `WorkoutService` earlier, we can implement other get operations for exercise- and workout-related stuff. Copy the service implementation for the `getExercise` and `getWorkout` methods of `WorkoutService` from `workout.service.ts` in the `trainer/src/app/core` folder in checkpoint 5.2.



The `getWorkout` and `getExercise` methods 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` objects, we use the name of the exercise for unique identification. Hence, the `name` and `_id` property of each object always match.

At this point, we will need to add one more import to `workout.service.ts`:

```
| import { forkJoin } from 'rxjs/observable/forkJoin';
```

This import brings in the `forkJoin` operator, which we will be discussing shortly.

Pay special attention to the implementation for the `getWorkout` method because there is a decent amount of data transformation happening due to the model and data storage format mismatch. This is how the `getWorkout` method now looks:

```
getWorkout(workoutName: string): Observable<WorkoutPlan> {
    return forkJoin(
        this.http.get(this.collectionsUrl + '/exercises' + this.params),
        this.http.get(this.collectionsUrl + '/workouts/' + workoutName +
this.params))
    .pipe(
        map(
            (data: any) => {
                const allExercises = data[0];
                const workout = new WorkoutPlan(
                    data[1].name,
                    data[1].title,
                    data[1].restBetweenExercise,
                    data[1].exercises,
                    data[1].description
                );
                workout.exercises.forEach(
                    (exercisePlan: any) => exercisePlan.exercise =
allExercises.find(
                        (x: any) => x.name === exercisePlan.name
                    )
                );
            }
        )
    );
}
```

```
|           }
|     ),
|     catchError(this.handleError<WorkoutPlan>(`getworkout id=${workoutName}`))
| );
| }
```

There is a lot happening inside `getworkout` that we need to understand.

The `getworkout` method uses `Observable` and its `forkJoin` operator to return two `Observable` objects: one that retrieves the `workout` and another that retrieves a list of all the `Exercises`. What is interesting about the `forkJoin` operator is that not only does it allow us to return multiple `Observable` streams, but it also waits until both Observable streams have retrieved their data before further processing the results. In other words, it enables us to stream the responses from multiple concurrent HTTP requests and then operate on the combined results.

Once we have the `workout` details and the complete list of exercises, we then `pipe` the results to the `map` operator (which we saw previously with the code for the `workouts` list), which we use to change the `exercises` array of the `workout` to the correct `Exercise` class object. We do this by searching the `allExercises` `Observable` for the name of the exercise in the `workout.exercises` array returned from the server, and then assigning the matching exercise to the `workout.services` array. The end result is that we have a complete `WorkoutPlan` object with the `exercises` array set up correctly.

These `WorkoutService` changes warrant fixes in upstream callers too. We have already fixed the lists of exercises in the `LeftNavExercises` and `Exercises` components and the `workouts` in the `workouts` component. Now let's fix the `Workout` and `Exercise` components along similar lines. The `getWorkout` and `getExercise` methods in the `workout services` are not directly called by these components, but by builder services. So we'll have to fix the builder services together with the `Workout` and `Exercise` components and the two resolvers—`WorkoutResolver` and `ExerciseResolver`—that we have added to the routes for these components.

Fixing the builder services

Now that we have `WorkoutService` set up to retrieve a workout from our remote data store, we have to modify `WorkoutBuilderService` to be able to retrieve that workout as an Observable. The method that pulls the `Workout` details is `startBuilding`. In order to do that, we will break the current `startBuilding` method into two methods, one for new workouts and one for existing workouts that we have retrieved from the server. Here is the code for new workouts:

```
startBuildingNew() {
  const exerciseArray: ExercisePlan[] = [];
  this.buildingWorkout = new WorkoutPlan('', '', 30, exerciseArray);
  this.newWorkout = true;
  return this.buildingWorkout;
}
```

For existing workouts, we add the following code:

```
startBuildingExisting(name: string) {
  this.newWorkout = false;
  return this.workoutService.getWorkout(name);
}
```

We'll let you make the same fixes in `ExerciseBuilderService`.

Updating the resolvers

As we move on to using Observable types with our data access, we are going to have to make some adjustments to the resolvers that we have created for the routes leading to workout and exercise screens. We start with the `workoutResolver` in `workout-resolver.ts` that can be found in the `workout` folder.

First add the following imports from RxJs:

```
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';
import { map, catchError } from 'rxjs/operators';
```

Next update the `resolve` method as follows:

```
resolve(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Observable<WorkoutPlan> {
  const workoutName = route.paramMap.get('id');

  if (!workoutName) {
    return this.workoutBuilderService.startBuildingNew();
  } else {
    return this.workoutBuilderService.startBuildingExisting(workoutName)
      .pipe(
        map(workout => {
          if (workout) {
            this.workoutBuilderService.buildingWorkout = workout;
            return workout;
          } else {
            this.router.navigate(['/builder/workouts']);
            return null;
          }
        }),
        catchError(error => {
          console.log('An error occurred!');
          this.router.navigate(['/builder/workouts']);
          return of(null);
        })
      );
  }
}
```

As you can see, we have split out the behavior for a new workout (one where there is no workout name being passed as a parameter in the URL) and that for an existing workout. In the former case, we call

`workoutBuilderService.startBuildingExisting`, which will return a new `WorkoutPlan`. In the latter case, we call `workoutBuilderService.startBuildingExisting` and pipe the results and then map them to return the `workout` unless it is not found, in which case we

route the user back to the `workouts` screen.

Fixing the Workout and Exercise components

Once we have fixed the `WorkoutBuilderService` and the `WorkoutResolver`, there are actually no further fixes needed in the `workoutComponent`. All the work to handle the Observables has been done further downstream and all we need to do at this stage is subscribe to the route data and retrieve the workout as we have already been doing:

```
ngOnInit() {
  this.sub = this.route.data
    .subscribe(
      (data: { workout: WorkoutPlan }) => {
        this.workout = data.workout;
      }
    );
}
```

To test the implementation, uncomment the following highlighted code contained in the `onSelect` method within `workouts.component.ts`:

```
onSelect(workout: WorkoutPlan) {
  this.router.navigate( ['./builder/workout', workout.name] );
}
```

Then click on any existing workout, such as *7 Minute Workout*, from the list of workouts displayed at `/builder/workouts/`. The workout data should load successfully.

The `ExerciseBuilderService` and `ExerciseResolver` also need fixing. Checkpoint 5.2 contains those fixes. You can copy those files or do it yourself and compare the implementation. And don't forget to uncomment the code in the `onSelect` method in `exercises.component.ts`.



Checkpoint 5.2 in the GitHub repository contains the working implementation for what we have covered thus far. If you are not using Git, download the snapshot of Checkpoint 5.2 (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/tree/checkpoint5>. 2. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

~~It is now time to fix, create, and update scenarios for the exercises and workouts.~~

Performing CRUD on exercises/workouts

When it comes to create, read, update, and delete (CRUD) operations, all save, update, and delete methods need to be converted to the Observable pattern.

Earlier in the chapter, we detailed the endpoint access pattern for CRUD operations in a MongoLab collection. Head back to the *Loading exercise and workout data* section and revisit the access patterns. We need this now as we plan to create/update workouts.

Before we start the implementation, it is important to understand how MongoLab identifies 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 that 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.

Creating a new workout

Taking the bottom-up approach, the first thing that needs to be fixed is `workoutService`. Update the `addWorkout` method as shown in the following code:

```
addWorkout(workout: WorkoutPlan) {
    const workoutExercises: any = [];
    workout.exercises.forEach(
        (exercisePlan: any) => {
            workoutExercises.push({name: exercisePlan.exercise.name, duration:
exercisePlan.duration});
        }
    );
    const body = {
        '_id': workout.name,
        'exercises': workoutExercises,
        'name': workout.name,
        'title': workout.title,
        'description': workout.description,
        'restBetweenExercise': workout.restBetweenExercise
    };
    return this.http.post(this.collectionsUrl + '/workouts' + this.params, body)
        .pipe(
            catchError(this.handleError<WorkoutPlan>())
        );
}
```

In `getWorkout`, we had to map data from the server model to our client model; the reverse has to be done here. First, we create a new array for the exercises, `workoutExercises`, and then add to that array a version of the exercises that is more compact for server storage. We only want to store the exercise name and duration in the exercises array on the server (this array is of type `any` because in its compact format it does not conform to the `ExercisePlan` type).

Next, we set up the body of our post by mapping these changes into a JSON object. Note that as part of constructing this object, we set the `_id` property as the name of the workout to uniquely identify it in the database of the `workouts` collection.

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 accessed simultaneously 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 that 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



I 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 auto generation of the `id` value is preferable. In standard cases where we are creating entities, unique ID generation is done on the server (mostly by the database). The response to when an entity is created then contains the generated ID. In such a case, we need to update the model object before we return data to the calling code.

Lastly, we call the `post` method of the `HttpClient` module, passing the URL to connect to, an extra query string parameter (`apiKey`), and the data we are sending.

The last return statement should look familiar, as we use Observables to return the `workout` object as part of the Observable resolution. You need to be sure you add `.subscribe` to the Observable chain in order to make it work. We'll do that shortly by adding a subscription to the `save` method to `WorkoutComponent`.

Updating a workout

Why not try to implement the update operation? The `updateWorkout` method can be fixed in the same manner, the only difference being that the `HTTPClient` module's `put` method is required:

```
updateWorkout(workout: WorkoutPlan) {
    const workoutExercises: any = [];
    workout.exercises.forEach(
        (exercisePlan: any) => {
            workoutExercises.push({name: exercisePlan.exercise.name, duration:
exercisePlan.duration});
        }
    );

    const body = {
        '_id': workout.name,
        'exercises': workoutExercises,
        'name': workout.name,
        'title': workout.title,
        'description': workout.description,
        'restBetweenExercise': workout.restBetweenExercise
    };

    return this.http.put(this.collectionsUrl + '/workouts/' + workout.name +
this.params, body)
    .pipe(
        catchError(this.handleError<WorkoutPlan>())
    );
}
```

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 the document is not found in the collection. While making the `PUT` request, make sure that the original record exists. We can do this by making a `CET` request for the same document first and confirming that we get a document before we update it. We'll leave that for you to implement.

Deleting a workout

The last operation that needs to be fixed is deleting the workout. Here is a simple implementation where we call the `HTTPClient` module's `delete` method to delete the workout referenced by a specific URL:

```
deleteWorkout(workoutName: string) {
    return this.http.delete(this.collectionsUrl + '/workouts/' + workoutName +
this.params)
    .pipe(
        catchError(this.handleError<WorkoutPlan>())
    );
}
```

Fixing the upstream code

With that, it's now time to fix the `workoutBuilderService` and `workout` components. The `save` method of `workoutBuilderService` now looks as follows:

```
save() {
  const workout = this.newWorkout ?
    this.workoutService.addWorkout(this.buildingWorkout) :
    this.workoutService.updateWorkout(this.buildingWorkout);
  this.newWorkout = false;
  return workout;
}
```

Most of it looks the same as it was earlier because it is the same! We did not have to update this code because we effectively isolated the interaction with the external server in our `workoutService` component.

Finally, the `save` code for the `workout` component is shown here:

```
save(formWorkout: any) {
  this.submitted = true;
  if (!formWorkout.valid) { return; }
  this.workoutBuilderService.save().subscribe(
    success => this.router.navigate(['/builder/workouts']),
    err => console.error(err)
  );
}
```

Here we have made a change so that we now subscribe to the `save`. As you may recall from our previous discussions, subscribe makes an Observable live so that we can complete the save.

And that's it! We can now create new workouts and update existing workouts (we'll leave completion of deleting workouts to you). 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 out `checkpoint 5.3` for an up-to-date implementation if you are having issues running your local copy. If you are not using Git, download the snapshot of Checkpoint 5.3 (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/tree/checkpoint5.3>.

Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Something interesting happens on the network side while we make `POST` or `PUT`

requests save data. Open the browser's network log console (*F12*) and see the requests being made. The log looks something like the following:

Name	Method	Status
7minworkout?apiKey=9xf7...t11KnaqzV9E_8vCzo5ksjexx	OPTIONS	200
7minworkout?apiKey=9xf7...t11KnaqzV9E_8vCzo5ksjexx	PUT	200
workouts?apiKey=9xf7...t11KnaqzV9E_8vCzo5ksjexx	GET	200

The network log

An `OPTIONS` request is made to the same endpoint before the actual `POST` or `PUT` is done. The behavior that we witness here is termed as a **preflight request**. This happens because we are making a cross-domain request to `api.mongolab.com`.

Using promises for HTTP requests

The bulk of this chapter has focused on how the Angular `HttpClient` uses **Observables as the default for XMLHttpRequests**. This represents a significant change from the way things used to work. Many developers are familiar with using promises for asynchronous HTTP requests. With that being the case, Angular continues to support promises, but just not as the default choice. A developer has to opt for promises in an `XMLHttpRequest` in order to be able to use them.

For example, if we want to use promises with the `getExercises` method in `workoutService`, we will have to restructure the command as follows:

```
getExercises(): Promise<Exercise[]> {
  return this.http.get<Exercise[]>(this.collectionsUrl + '/exercises' +
    this.params)
      .toPromise()
      .then(res => res)
      .catch(err => {
        return Promise.reject(this.handleError('getExercises', []));
      });
}
```

In order to convert this method to use promises, all we have to do is add `.toPromise()` to the method chain, a success parameter, `then`, for the promise, and `catch` with a `Promise.reject` pointing to the existing `handleError` method.

For upstream components, we just have to switch to handling the return value as a promise rather than an Observable. So, to use promises in this case, we would have to change the code in `Exercises.component.ts` and `LeftNavExercises.component.ts` to first add a new property for the error message (we'll leave it to you as to how the error message is displayed on the screen):

```
|errorMessage: any;
```

Then change the `ngOnInit` method that is calling `workoutService` to the following:

```
ngOnInit() {
  this.workoutService.getExercises()
    .then(exerciseList => this.exerciseList = exerciseList,
          error => this.errorMessage = <any>error
    );
}
```

Of course, the ease with which we can substitute promises for Observables in this simple example does not indicate that they are essentially the same. A then promise returns another promise, which means that you can create successively chained promises. In the case of an Observable, a subscription is essentially the end of the line and cannot be mapped or subscribed to beyond that point.

If you're familiar with promises, it may be tempting at this stage to stick with them and not give Observables a try. After all, much of what we have done with Observables in this chapter can be done with promises as well. For example, the mapping of two streams of Observables that we did with `getWorkouts` using the Observable's `forkJoin` operator can also be done with the promise's `q.all` function.

However, you would be selling yourself short if you took that approach. Observables open up an exciting new way of doing web development using what is called functional reactive programming. They involve a fundamental shift in thinking that treats an application's data as a constant stream of information to which the application reacts and responds. This shift allows applications to be built with a different architecture that makes them faster and more resilient. Observables are at the core of Angular in such things as event emitters and the new version of NgModel.

While promises are a useful tool to have in your toolkit, we encourage you to investigate Observables as you get into developing with Angular. They are part of the forward-looking philosophy of Angular and will be useful in future-proofing both your applications and your skill set.



Check out the `checkpoint 5.3` file for an up-to-date implementation that includes the promises-related code that we covered previously. If you are not using Git, download the snapshot of Checkpoint 5.3 (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular-6byexample/tree/checkpoint5.3>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time. Be aware that in the next section, we will be reverting to the use of Observables for this code. This code can be found in the `checkpoint 5.4` file.

The `async` pipe

As we have seen with many of the data operations covered in this chapter, there is a fairly common pattern being repeated over and over again. When an Observable is returned from an HTTP request, we convert the response to JSON and subscribe to it. The subscription then binds the Observable output to a UI element. Wouldn't it be nice if we could eliminate this repetitive coding and replace it with a simpler way to accomplish what we are wanting to do?

Not surprisingly, Angular provides us with just the right way to do that. It's called the **async pipe**, and it can be used like any other pipe for binding to an element on the screen. However, the async pipe is a much more powerful mechanism than other pipes. It takes an Observable or a promise as an input and subscribes to it automatically. It also handles the teardown of the subscription for an Observable without necessitating any further lines of code.

Let's look at an example of this in our application. Let's go back to the `LeftNavExercises` component that we were just looking at in the previous section in connection with promises. Note that we have converted this component and the `Exercises` component from promises back to using Observables.



Check out the `checkpoint 5.4` file for an up-to-date implementation that includes the conversion of this code to use Observables once again. If you are not using Git, download the snapshot of Checkpoint 5.4 (a ZIP file) from the following GitHub location: <https://github.com/chandermani/angular6byexample/tree/checkpoint5.4>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Then make the following changes in `LeftNavExercises`. First, import Observable from RxJs:

```
| import { Observable } from 'rxjs/Observable';
```

Then change `exerciseList` from an array of exercises to an Observable of the same type:

```
| public exerciseList:Observable<Exercise[]>;
```

Next modify the call to `workoutService` that gets the exercises to eliminate the

subscription:

```
| this.exerciseList = this.workoutService.getExercises();
```

Finally, open `left-nav-exercises.component.html` and add the `async` pipe to the `*ngFor` loop, as follows:

```
|<div *ngFor="let exercise of exerciseList|async|orderBy:'title'">
```

Refresh the page and you will still see the Exercise list displaying. But this time, we have used the `async` pipe to eliminate the need to set up the subscription to the Observable. Pretty cool! This is a nice convenience that Angular has added, and since we have been spending time in this chapter understanding how Observables work with subscriptions, we have a clear idea of what the `async` pipe is now handling for us under the hood.

We'll leave it to you to implement the same change in the `Exercises component`.

~~It is important to understand the cross-domain behavior of the HTTP request and the constructs that Angular provides to make cross-domain requests.~~

Cross-domain access and Angular

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 called *same-origin policy* restrictions. Such a 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. They 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 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 the response data gives this technique the name JSONP).

The Angular JSONP service hides this complexity and provides an easy API to make JSONP requests. The StackBlitz link, <https://stackblitz.com/edit/angular-nxeuxo>, highlights how JSONP requests are made. It uses the *IEX Free Stock API* (<https://iextrading.com/developer/>) to get quotes for any stock symbol.



The Angular JSONP service only supports HTTP GET requests. Using any other HTTP request, such as POST or PUT, will generate an error.

If you look at the StackBlitz project, you will see the familiar pattern for component creation that we have followed throughout this book. We will not go over this pattern again, but will highlight a few details that are relevant to using the Angular JSONP service.

First, along with the imports for `FormsModule` and `HttpClientModule`, you will need to import `HttpClientJsonpModule` into `app.module.ts` as follows:

```
import { HttpClientModule, HttpClientJsonpModule } from '@angular/common/http';
import { FormsModule } from '@angular/forms';
...
@NgModule({
  ...
})
```

```
imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule,
  HttpClientJsonpModule
],
. .
})
```

Next, we need to add the following imports to `get-quote.component.ts`:

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { map } from 'rxjs/operators';
```

We are importing `HttpClient`, which contains the `JSONP` methods we will be using, as well as the RxJS `Observable` and the `map` operator. These imports will look familiar to you from what we have been building in this chapter.



As you work with Angular JSONP, it is important to understand that by default, it returns Observables using RxJS. This means that we will have to follow the pattern for subscribing to those Observables and use the RxJS operators to manipulate the results. We can also use the `async` pipe to streamline these operations.

Then we inject `HttpClient` into the constructor:

```
|constructor(public http: HttpClient) {}
```

Next we add several variables that we will be using in our JSONP call:

```
symbol: string;
quote: Observable<string>;
url: string = 'https://api.iextrading.com/1.0/stock/';
```

The `symbol` variable will hold the search string provided by the user. The `quote` variable will be used in our template to display the returned value from the JSONP call. And the `url` variable is the base URL for the call we will be making to the service.

Now we have everything in place for our `getQuote` method. Let's take a look at it:

```
getQuote (){
  let searchUrl = `${this.url}${this.symbol}/quote`;
  this.quote = this.http.jsonp(searchUrl, 'callback')
    .pipe(
      map( (res: string) => res)
    );
};
```

We first construct our `searchurl` by concatenating the `url` with the `symbol` and adding `/quote`. The last part `quote` is what we need to pass to the quote service to return a stock quote.

We then use the `HttpClient's jsonp` method to execute the remote call to the quote service. We pass the `searchurl` as the first parameter of that method and a string `'callback'` as our second parameter. The latter parameter is used by Angular to augment the `searchurl` with an extra query string parameter, `callback`. Internally, the Angular JSONP service then generates a dynamic script tag and a callback function and makes the remote request.

Open StackBlitz and enter symbols such as `GOOG`, `MSFT`, or `FB` to see the stock quote service in action. The browser network log for requests looks as follows:

```
| https://api.iextrading.com/1.0/stock/MSFT/quote?callback=ng_jsonp_callback_0
```

Here, `ng_jsonp_callback_0` is the dynamically generated function. And the response looks as follows:

```
| typeof ng_jsonp_callback_0 === 'function' && ng_jsonp_callback_0({ "quote": : :  
| { "symbol": "MSFT" ..});
```

The response is wrapped in the `callback` function. Angular parses and evaluates this response, which results in the invocation of the `_ng_jsonp_.req1` callback function. Then, this function internally routes the data to our function `callback`.

We hope this explains how JSONP works and what the underlying mechanism of a JSONP request is. However, JSONP has its limitations:

- First, we can make only `GET` requests (which is obvious as these requests originate due to script tags)
- Second, the server also needs to implement the part of the solution that involves wrapping the response in a function callback
- Third, there is always a security risk involved, as JSONP depends on dynamic script generation and injection
- Fourth, error handling is not reliable too because it is not easy to determine why a script load failed

Ultimately, we must recognize that JSONP is more of a workaround than a solution. As we move towards Web 2.0, where mashups become commonplace

and more and more service providers decide to expose their API over the web, a far better solution/standard has 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, a consumer application (such as *Personal Trainer*) is allowed to make some types of requests, termed **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 **complex requests**.

For complex requests, CORS mandates that the request should be preceded by an 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 the calling code and our Angular `HTTPClient` calls work without a hitch.

CORS requires configurations to be made on the server, and the MongoLab servers have already been configured to allow cross-domain requests. So the preceding `POST` and `PUT` requests that we made to the MongoLab to add and update `Exercise` and `Workout` documents all caused the preflight `OPTIONS` request.

Handling workouts not found

You might recall that in [Chapter 4, Personal Trainer](#), we created the `WorkoutResolver` to not only retrieve a workout prior to navigation to the `WorkoutComponent`, but also prevent navigation to that component if a non-existent workout was in the route parameters. Now we would like to augment this functionality by displaying an error message on the workouts screen, indicating that the workout was not found.

In order to do this, we are going to modify `WorkoutResolver` so that it reroutes to the workouts screen if a workout is not found. To start, add the following child route to `WorkoutBuilderRoutingModule` (making sure it precedes the existing workouts route):

```
children: [
  {path: '', pathMatch: 'full', redirectTo: 'workouts'},
  {path: 'workouts/workout-not-found', component: WorkoutsComponent},
  {path: 'workouts', component: 'WorkoutsComponent'},
  *** other child routes ***
],
]
```

Next, modify the `resolve` method in the `WorkoutResolver` to redirect to this route in the event that a workout is not found:

```
resolve(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Observable<WorkoutPlan> {
  const workoutName = route.paramMap.get('id');

  if (!workoutName) {
    return this.workoutBuilderService.startBuildingNew();
  } else {
    this.isExistingWorkout = true;
    return this.workoutBuilderService.startBuildingExisting(workoutName)
      .pipe(
        map(workout => {
          if (workout) {
            this.workoutBuilderService.buildingWorkout = workout;
            return workout;
          } else {
            this.router.navigate(['/builder/workouts/workout-not-found']);
            return null;
          }
        }),
        catchError(error => {
          console.log('An error occurred!');
          this.router.navigate(['/builder/workouts']);
        })
      );
  }
}
```

```

        return of(null);
    }
}

```

Then add a `notFound` boolean set to `false` to the variables in the `Workouts` component:

```

workoutList: Array<WorkoutPlan> = [];
public notFound = false;

```

And, in the `ngOnInit` method of that component, add the following code to check for the `workout-not-found` path and set the `notFound` value to `true`:

```

ngOnInit() {
  if(this.route.snapshot.url[1] && this.route.snapshot.url[1].path ===
    'workout-not-found') this.notFound = true;
  this.subscription = this.workoutService.getWorkouts()
    .subscribe(
      workoutList => this.workoutList = workoutList,
      (err:any) => console.error(err)
    );
}

```

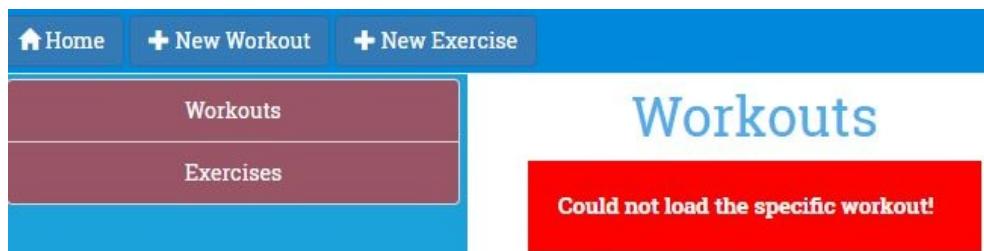
Finally in the `Workouts.component.html` template, add the following `div` tag above the workout list that will display if the `notFound` is set to `true`:

```

<div *ngIf="notFound" class="not-found-msgbox">Could not load the specific workout!
</div>

```

If we find `workout-not-found` in the path when a user is returned to the `Workouts` page, then this displays the following message on the screen:



We have fixed routing failure for the `Workout Builder` page, but the exercise builder page is still pending. Again, we will leave it to you to fix it yourself.

Another major (and pending) implementation is fixing `7 Minute Workout`, as it currently caters to only one workout routine.

Fixing the 7 Minute Workout app

As it stands now, the *7 Minute Workout* (or *Workout Runner*) app can play only one specific workout. It needs to be fixed to support the 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've got the shared model services and `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:

- Removing the hardcoded workout and exercises used in *7 Minute Workout*.
- Fixing the start page to show all available workouts and allowing users to select a workout to run.
- Fixing the workout route configuration to pass the selected workout name as the route parameter to the workout page.
- 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 is now a misnomer. I think the complete app can be called *Personal Trainer*. We can remove all references to *7 Minute Workout* from the view as well.

An excellent exercise to try out yourself! And that is why we are not going to walk you through the solution. Instead, go ahead and implement the solution. Compare your implementation with the one available at [checkpoint 5.4](#).

It's time to end the chapter and summarize your learning.

Summary

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. We bet that if we were to try this in standard jQuery or some other framework, it would require substantially more effort as compared to Angular.

We started the chapter 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.

The first Angular construct that we touched upon was the `HttpClient`, which is the primary service for connecting to any HTTP backend.

You also learned how the `HttpClient` module uses Observables. For the first time, in this chapter, we created our own Observable and explained how to create subscriptions to those Observables.

We fixed our *Personal Trainer* app so that it uses the `HttpClient` module to load and save workout data (note that data persistence for exercises is left for you to complete). 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.

We have now covered most of the building blocks of Angular, except the big one: Angular directives. We have used directives everywhere, but have not created one. The next chapter is exclusively dedicated to Angular directives. We will be creating a number of small directives, such as a remote validator, AJAX button, and a validation cues directive for the *Workout Builder* app.

Angular Directives in Depth

Directives are everywhere. They are the **fundamental building blocks** of Angular. Each extension to the application has resulted in us creating new **component directives**. These component directives have further consumed **attribute directives** (such as `ngClass` and `ngStyle`) and **structural directives** (such as `ngIf` and `ngFor`) to extend their behavior.

While we have built a number of component directives and a lone attribute directive, there are still some concepts of directive building that are worth exploring. This is especially true for attribute and structural directives, which we are yet to cover in detail.

The topics we will cover in this chapter include the following:

- **Building directives:** We build multiple directives and learn where directives are useful, how they differ from components, and how directives communicate with each other and/or their host component. We explore all directive types, including *component directives*, *attribute directives*, and *structural directives*.
- **Asynchronous validation:** Angular makes it easy to validate rules that require server interaction and hence are async in nature. We will build our first async validator in this chapter.
- **Using renderer for view manipulation:** Renderer allows view manipulation in a platform-agnostic way. We will utilize renderer for the busy indicator directive and learn about its API.
- **Host binding:** Host binding allows directives to communicate with their host element. This chapter covers how to utilize such bindings for directives.
- **Directive injection:** The Angular DI framework allows directive injection based on where in the HTML hierarchy the directives are declared. We will cover multiple scenarios pertaining to such injections.
- **Working with view children and content children:** Components have the capability to include external view templates into their own view. How to

work with the injected content is something we will cover here.

- **Understanding the NgIf platform directive:** We will look under the hood of the `NgIf` platform directive and try to comprehend the working of *structural directives* such as `NgIf`.
- **View encapsulation of Angular components:** We will learn how Angular uses concepts derived from web components to support view and style encapsulation.

Let's start the chapter by reiterating the basic classification of directives.

Classifying directives

Angular directives integrate the HTML view with the application state. Directives help us manipulate views as application state changes and respond to view updates with little or no manual interaction with the actual DOM.

Depending upon how they affect the view, these directives are further classified into three types.

Components

Component directives or components are directives with an **encapsulated view**. In Angular, when we build UI widgets, we are building components. We have already built a lot of them, such as `WorkoutRunnerComponent`, `WorkoutAudioComponent`, `videoPlayerComponent`, and many more!

An important point to realize here is that the view is bound to the component implementation and it can only work with properties and events defined on the backing component.

Attribute directives

Attribute directives, on the other hand, extend an existing component or HTML element. Consider them as behavioral extensions to these components/elements.

Since directives are behavioral extensions for predefined elements, every directive building exercise involves manipulating the state of the components/elements on which these directives are applied. The `MyAudioDirective` built in [Chapter 3, More Angular 2 – SPA, Routing, and Data Flows in Depth](#), does the same. The directive wraps the HTML5 `audio` element (`HTMLAudioElement`) for easy usage. Platform directives such as `ngStyle` and `ngClass` also function in a similar manner.

Structural directives

Structural directives, such as attribute directives, do not define their own view. Instead, they work on the *view template* (HTML fragment) provided to them as part of their usage. More often than not, the purpose of a structural directive is to show/hide or clone the template view provided to it. Platform directives such as `NgFor`, `NgIf`, and `NgSwitch` are the prime examples in this category.

I hope this quick refresher on directives is enough to get us started. We'll begin our pursuit by extending the workout builder validations and build an async validator directive.



We are starting from where we left off in chapter 5, Supporting Server Data Persistence. The Git branch `checkpoint5.4` can serve as the base for this chapter. The code is also available on GitHub (<https://github.com/chandermani/angular6byexample>) for everyone to download. Checkpoints are implemented as branches in GitHub. If you are not using Git, download the snapshot of `checkpoint5.4` (a ZIP file) from the GitHub location <http://bit.ly/ng6be-checkpoint-5-4>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time. Also, remember to update the API key in `services/workout-service.ts` with your own API key.

Building a remote validator directive

We ended [Chapter 5](#), *Supporting Server Data Persistence*, with *Workout Runner* capable of managing workouts in the MongoDB store. Since each workout should have a unique name, we need to enforce the uniqueness constraint. Therefore, while creating/editing a workout, every time the user changes the workout name, we can query MongoDB to verify that the name already exists.

As is the case with any remote invocation, this check too is asynchronous, and hence it requires a *remote validator*. We are going to build this remote validator using Angular's *async validator support*.

Async validators are similar to standard custom validators, except that instead of returning a key-value object map or null, the return value of a validation check is a **promise**. This promise is eventually resolved with the validation state being set (if there is an error), or null otherwise (on validation success).

We are going to create a validation directive that does workout name checks. There are two possible implementation approaches for such a directive:

- We can create a directive specifically for unique name validation
- We can create a generic directive that can perform any remote validation



Validation directives

While we are building a validation directive here, we could have built a standard custom validator class. The advantage of creating a directive is that it allows us to incorporate the directive in a template-driven form approach, where the directive can be embedded in the view HTML. Or, if the form has been generated using a model (reactive approach), we can directly use the validator class while creating the control objects.

At first, the requirement of checking duplicate names against a data source (the *mLab* database) seems to be too a specific requirement and 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.

The plan is to create a validator that externalizes the actual validation logic. The directive will take the validation function as input. This implies that the actual

validation logic is not a part of the validator, but a part of the component that actually needs to validate input data. The job of the directive is just to call the function and return the appropriate error keys based on the function's return value.

Let's put this theory into practice and build our remote validation directive, aptly named `RemoteValidatorDirective`.



The companion code base for the following section is Git branch `checkpoint6.1`. You can work along with us or check out the implementation available in the aforementioned folder. Or if you are not using Git, download the snapshot of `checkpoint6.1` (a ZIP file) from GitHub location <http://bit.ly/ng2be-checkpoint6-1>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Validating workout names using async validators

Like custom validators, async validators inherit from the same `validator` class too; but this time, instead of returning an object map, async validators return a `Promise`.

Let's look at the definition of the validator. Copy the definition of the validator from the GitHub (<http://bit.ly/ng6be-6-1-remote-validator-directive-ts>) folder and add it to the `shared` module folder. The validator definition looks as follows:

```
import { Directive, Input } from '@angular/core';
import { NG_ASYNC_VALIDATORS, FormControl } from '@angular/forms';

@Directive({
  selector: '[abeRemoteValidator][ngModel]',
  providers: [{ provide: NG_ASYNC_VALIDATORS, useExisting: RemoteValidatorDirective,
    multi: true }]
})
export class RemoteValidatorDirective {

  @Input() abeRemoteValidator: string;
  @Input() validateFunction: (value: string) => Promise<boolean>;

  validate(control: FormControl): { [key: string]: any } {
    const value: string = control.value;
    return this.validateFunction(value).then((result: boolean) => {
      if (result) {
        return null;
      }
      else {
        const error: any = {};
        error[this.abeRemoteValidator] = true;
        return error;
      }
    });
  }
}
```

Do remember to export this directive from the shared module, allowing us to use it in the workout builder module.

Since we are registering the validator as a directive instead of registering using a `FormControl` instance (generally used when building forms with a *reactive approach*), we need the extra provider configuration setting (added in the preceding `@Directive` metadata) by using this syntax:

```
| providers:[{ provide: NG_ASYNC_VALIDATORS, useExisting: RemoteValidatorDirective,  
| multi: true }]
```

This statement registers the validator with the existing async validators.



The strange directive selector, `selector: '[abeRemoteValidator][ngModel]`', used in the preceding code will be covered in the next section, where we will build a busy indicator directive.

Before we dig into the validator implementation, let's add it to the workout name input. This will help us correlate the behavior of the validator with its usage.

Update the workout name input (`workout.component.html`) with the validator declaration:

```
| <input type="text" name="workoutName" ...  
|   abeRemoteValidator="workoutname" [validateFunction]="validateWorkoutName">  
|   Prefixing the directive selector  
|   TIP Always prefix your directives with an identifier (abe as you just saw) that distinguishes them from framework directives and other third-party directives.
```

Note: If the `ngModelOptions`, `updateOn` is set to `submit`, change it to `blur`.

The directive implementation takes two inputs: the *validation key* through directive property `abeRemoveValidator`, used to set the *error key*, and the *validation function* (`validateFunction`), called to validate the value of the control. Both inputs are annotated with the `@Input` decorator.



The input parameter `@Input("validateFunction") validateFunction: (value: string) => Promise<boolean>;`, binds to a function, not a standard component property. We are allowed to treat the function as a property due to the nature of the underlying language, TypeScript (as well as JavaScript).

When the async validation fires (on a change of `input`), Angular invokes the function, passing in the underlying `control`. As the first step, we pull the current input value and then invoke the `validateFunction` function with this input. The `validateFunction` returns a promise, which should eventually resolve to `true` or `false`:

- If the promise resolves to `true`, the validation is successful, the promise callback function returns `null`.
- If it is `false`, the validation has failed, and an error key-value map is returned. The *key* here is the string literal that we set when using the validator (`abeRemoteValidator="workoutname"`).

This *key* comes in handy when there are multiple validators declared on the

input, allowing us to identify validations that have failed.

To the workout component next add a validation message for this failure too. Add this label declaration after the existing validation `label` for *workout name*:

```
|<label *ngIf="name.control.hasError('workoutname')" class="alert alert-danger">  
|  validation-message>A workout with this name already exists.</label>
```

And then wrap these two labels inside a `div`, as we do for *workout title* error labels.

The `hasError` function checks whether the `'workoutname'` validation key is present.

The last missing piece of this implementation is the actual validation function we assigned when applying the directive (`[validateFunction] = "validateWorkoutName"`), but never implemented.

Add the `validateWorkoutName` function to `workout.component.ts`:

```
validateWorkoutName = (name: string): Promise<boolean> => {  
  if (this.workoutName === name) { return Promise.resolve(true); }  
  return this.workoutService.getWorkout(name).toPromise()  
    .then((workout: WorkoutPlan) => {  
      return !workout;  
    }, error => {  
      return true;  
    });  
}
```

Before we explore what the preceding function does, we need to do some more fixes on the `WorkoutComponent` class. The `validateWorkoutName` function is dependent on `workoutService` to get a workout with a specific name. Let's inject the service in the constructor and add the necessary import in the imports section:

```
import { WorkoutService } from "../../core/workout.service";  
...  
constructor(... , private workoutService: WorkoutService) {
```

Then declare variables `workoutName` and `queryParamsSub`:

```
private workoutName: string;  
queryParamsSub: Subscription
```

And add this statement to `ngOnInit`:

```
this.queryParamsSub = this.route.params.subscribe(params => this.workoutName =  
  params['id']);
```

The preceding statement set the current workout name by watching (subscribing) over the observable `route.params` service. `workoutName` is used to skip workout name validation for an existing workout if the original workout name is used.

The subscription created previously needs to be clear to avoid memory leak, hence add this line to the `ngDestroy` function:

```
| this.queryParamsSub.unsubscribe();
```

The reason for defining the `validateWorkoutName` function as an instance function (the use of the *arrow operator*) instead of defining it as a standard function (which declares the function on the *prototype*) is the '`this`' scoping issue.

Look at the validator function invocation inside `RemoteValidatorDirective` (declared using `@Input("validateFunction") validateFunction;`):

```
| return this.validationFunction(value).then((result: boolean) => { ... });
```

When the function (named `validateFunction`) is invoked, the `this` reference is bound to `RemoteValidatorDirective` instead of the `WorkoutComponent`. Since `execute` is referencing the `validateWorkoutName` function in the preceding setup, any access to `this` inside `validateWorkoutName` is problematic.

This causes the `if (this.workoutName === name)` statement inside `validateWorkoutName` to fail, as `RemoteValidatorDirective` does not have a `workoutName` instance member. By defining `validateWorkoutName` as an instance function, the *TypeScript compiler creates a closure around the value of this when the function is defined.*

With the new declaration, `this` inside `validateWorkoutName` always points to the `WorkoutComponent` irrespective of how the function gets invoked.

We can also look at the compiled JavaScript for `WorkoutComponent` to know how the closure works with respect to `validateWorkoutName`. The parts of the generated code that interest us are as follows:

```
function WorkoutComponent(...) {
  var _this = this;
  ...
  this.validateWorkoutName = function (name) {
    if (_this.workoutName === name)
      return Promise.resolve(true);
```

If we look at the validation function implementation, we see that it involves querying `mLab` for a specific workout name. The `validateWorkoutName` function returns `true` when a workout with the same name is not found and `false` when a workout with the same name is found (actually a *promise* is returned).



The `getWorkout` function on `WorkoutService` returns an observable, but we convert it into a promise by calling the `toPromise` function on the observable.

The validation directive can now be tested. Create a new workout and enter an existing workout name such as `7minworkout`. See how the validation error message shows up eventually:

Name:

A workout with this name already exists.

Excellent! It looks great, but there is still something missing. The user is not informed that we are validating the workout name. We can improve this experience.

Building a busy indicator directive

While the workout name is being validated remotely, we want the user to be aware of the activity in the background. A visual clue around the input box while the remote validation happens should serve the purpose.

Think carefully; there is an input box with an asynchronous validator (which does remote validation) and we want to adorn the input box with a visual clue during validation. Seems like a common pattern to solve? Indeed it is, so let's create another directive!

But before we start the implementation, it is imperative to understand that we are not in it alone. The busy indicator directive requires the help of another directive, `NgModel`. We have already used the `NgModel` directive on `input` elements in [chapter 4, Building Personal Trainer](#). `NgModel` helps us track the input element state. The following example is taken from [chapter 4, Building Personal Trainer](#), and it highlights how `NgModel` helps us validate inputs:

```
<input type="text" name="workoutName" #name="ngModel" class="form-control" id="workout-name" ... [(ngModel)]="workout.name" required> ... <label *ngIf="name.control.hasError('required') && (name.touched || submitted)" class="alert alert-danger">Name is required</label>
```

Even the unique workout name validation done in the previous section employs the same technique of using `NgModel` to check the validation state.

Let's begin with defining the outline of the directive. Create a `busy-indicator.directive.ts` file using the CLI generator in the `src/app/shared` folder:

```
| ng generate directive busy-indicator
```

Also, export it by adding the directive to the `exports` array in the shared module file `shared.module.ts`.

Next, update the directive's constructor with `NgModel` injection and import the `NgModel` reference from `@angular/forms`:

```
| constructor(private model: NgModel) { }
```

This instructs Angular to inject the `NgModel` instance of the element on which the directive is declared. Remember that the `NgModel` directive is already present on `input (workoutname)`: `<input... name="workoutName" #name="ngModel" [(ngModel)]="workout.name" ...>`

This is enough to integrate our new directive in the workout view, so let's do it quickly.

Open `workout.component.html` from `workout-builder` and add the busy indicator directive to the workout name `input`:

```
| <input type="text" name="workoutName" ... abeBusyIndicator>
```

Create a new workout or open an existing one to see whether the `BusyIndicatorDirective` is loaded and the `NgModel` injection worked fine. This can be easily verified by putting a breakpoint inside the `BusyIndicatorDirective` constructor.

~~Angular injects the same `NgModel` instance into `BusyIndicatorDirective` that it created when it encountered `ngModel` on the input HTML.~~

You may be wondering what happens if we apply this directive on an input element that does not have the `ngModel` attribute, or as a matter of fact on any HTML element/component, such as this: `<div abeBusyIndicator></div> <input type="text" abeBusyIndicator>`

Will the injection work?

Of course not! We can try it in the create workout view. Open `workout.component.html` and add the following `input` above the workout name `input`. Refresh the app: `<input type="text" name="workoutName1" a2beBusyIndicator>`

Angular throws an exception, as follows:

```
| EXCEPTION: No provider for NgModel! (BusyIndicatorDirective -> NgModel)
```

How to avoid this? Well, Angular's DI can rescue us here as it allows us to declare an optional dependency.





Remove the `input` control that you just added before proceeding further.

Injecting optional dependencies with the `@Optional` decorator

Angular has an `@Optional` decorator, which when applied to a constructor argument instructs the Angular *injector* to inject `null` if the dependency is not found.

Hence, the busy indicator constructor can be written as follows:

```
| constructor(@Optional() private model: NgModel) { }
```

Problem solved? Not really; as stated previously, we require the `NgModel` directive for `BusyIndicatorDirective` to work. So, while we have learned something new, it is not very useful in the current scenario.



Before proceeding further, remember to revert the `workoutname` input to its original state, with `abeBusyIndicator` applied.

`BusyIndicatorDirective` should only be applied if there is an `NgModel` directive already present on the element.

The `selector` directive is going to save our day this time. Update the `BusyIndicatorDirective` selector to this:

```
| selector: `[abeBusyIndicator][ngModel]`
```

This `selector` creates the `BusyIndicatorDirective` only if the combination of `a2beBusyIndicator` with the `ngModel` attribute is present on the element. Problem solved!

It's now time to add the actual implementation.

Implementation one – using renderer

For `BusyIndicatorDirective` to work, it needs to know when the async validation on the `input` fires and when it is over. This information is only available with the `NgModel` directive. `NgModel` has a property, `control`, which is an instance of the `Control` class. It is this `control` class that tracks the current state of the input, including the following:

- Currently assigned validators (sync and async)
- The current value
- The input element state, such as `pristine`, `dirty`, and `touched`
- The input validation state, which could be any one of `valid`, `invalid`, or `pending` in the case of validation being performed asynchronously
- Events that track when the value changes or the validation state changes

`control` seems to be a useful class, and it's the `pending` state that interests us!

Let's add our first implement for the `BusyIndicatorDirective` class. Update the class with this code:

```
private subscriptions: Array<any> = [];
ngAfterViewInit(): void {
  this.subscriptions.push(
    this.model.control.statusChanges.subscribe((status: any) => {
      if (this.model.control.pending) {
        this.renderer.setStyle(this.element.nativeElement, 'border-width',
'3px');
        this.renderer.setStyle(this.element.nativeElement, 'border-color',
'gray');
      }
      else {
        this.renderer.setStyle(this.element.nativeElement, 'border-width',
null);
        this.renderer.setStyle(this.element.nativeElement, 'border-color',
null);
      }
    }));
}
```

Two new dependencies need to be added to the constructor, as we use them in the `ngAfterViewInit` function. Update the `BusyIndicatorDirective` constructor to look as follows:

```
| constructor(private model: NgModel,
```

```
|   private element: ElementRef, private renderer: Renderer) { }
```

And also add imports for `ElementRef` and `Renderer` in '`@angular/core`'.

`ElementRef` is a wrapper object over the underlying HTML element (`input` in this case). The `MyAudioDirective` directive built in [Chapter 3, More Angular 2 – SPA, Routing, and Data Flows in Depth](#), used `ElementRef` to get hold of the underlying `Audio` element.

The `Renderer` injection deserves a bit of attention. Calling `setElementStyle` is a dead giveaway that `Renderer` is responsible for managing the DOM. But before we delve more deeply into the role of `Renderer`, let's try to understand what the preceding code is doing.

In the preceding code, the `control` property on the model (the `NgModel` instance) defines an event (an Observable), `statusChanges`, which we can subscribe to in order to know when the control validation state changes. The available validation states are `valid`, `invalid`, and `pending`.

The subscription checks whether the control state is `pending` or not, and accordingly adorns the underlying element using the `Renderer` API function, `setElementStyle`. We set the `border-width` and `border-color` of the input.

The preceding implementation is added to the `ngAfterViewInit` directive lifecycle hook, which is called after the view has initialized.

Let's try it out. Open the create workout page or the existing *7 Minute Workout*. As soon as we leave workout name input, the `input` style changes and reverts once the remote validation of the workout name is complete. Nice!



Before moving forward, also add the un-subscription code to the `BusyIndicatorDirective` to avoid a memory leak. Add this function (life cycle hook) to `BusyIndicatorDirective`:

```
| ngOnDestroy() {
|   this.subscriptions.forEach((s) => s.unsubscribe());
| }
```



Always unsubscribe from observables

Always remember to unsubscribe from any observable/EventEmitter subscription done in the code to avoid memory leaks.

The implementation looks good. The `Renderer` is doing its job. But there are some unanswered questions.

Why not just get hold of the underlying DOM object and use the standard DOM API to manipulate the input styles? Why do we need the `renderer`?

Angular renderer, the translation layer

One of the primary design goals of Angular 2 was to make it run across environments, frameworks, and devices. Angular enabled this by dividing the core framework implementation into an **application layer** and a **rendering layer**. The application layer has the API we interact with, whereas the rendering layer provides an abstraction that the application layer can use without worrying about how and where the actual view is being rendered.

By separating the rendering layer, Angular can theoretically run in various setups. These include (but are not limited to):

- Browser
- Browser main thread and web worker thread, for obvious performance reasons
- Server-side rendering
- Native app frameworks; efforts are underway to integrate Angular with NativeScript with ReactNative
- Testing, allowing us to test the app UI outside the web browser



The Renderer implementation that Angular uses inside our browser is `DOMRenderer`. It is responsible for translating our API calls into browser DOM updates. In fact, we can verify the renderer type by adding a breakpoint in the `BusyIndicatorDirective`'s constructor and seeing the value of `renderer`.

For this precise reason, we avoid direct manipulation of DOM elements inside `BusyIndicatorDirective`. You never know where the code will end up running. We could have easily done this:

```
| this.element.nativeElement.style.borderWidth="3px";
```

Instead, we used the `Renderer` to do the same in a platform-agnostic way.

Look at the `Renderer` API function, `setElementStyle`:

```
| this.renderer.setStyle(  
|   this.element.nativeElement, "border-width", "3px");
```

It takes the element on which the style has to be set, the style property to update, and the value to set. The `element` references the `input` element injected into `BusyIndicatorDirective`.



Resetting styles

Styles set by calling `setElementStyle` can be reset by passing a `null` value in the third argument. Check out the `else` condition in the preceding code.

The `Renderer` API has a number of other methods that can be used to set attributes, set properties, listen to events, and even create new views. Whenever you build a new directive, remember to evaluate the `Renderer` API for DOM manipulation.



A more detailed explanation of `Renderer` and its application is available as part of Angular's design documents here: <http://bit.ly/ng2-render>

We are not done yet! With Angular's awesomeness, we can improve the implementation. Angular allows us to do *host binding* in directive implementation, helping us avoid a lot of boilerplate code.

Host binding in directives

In the Angular realm, the component/element that a directive gets attached to is termed the **host element**: a container that hosts our directive/component. For the `BusyIndicatorDirective`, the `input` element is the *host*.

While we can use the `Renderer` to manipulate the host (and we did too), the Angular data binding infrastructure can reduce the code further. It provides a declarative way to manage directive-host interaction. Using the host binding concepts, we can manipulate an element's *properties* and *attributes* and subscribe to its *events*.

Let's understand each of the host binding capabilities, and at the end, we will fix our `BusyIndicatorDirective` implementation.

Property binding using @HostBinding

Use **host property binding** to bind a *directive property* to a *host element property*. Any changes to the directive property will be synced with the linked host property during the change detection phase.

We just need to use the `@HostBinding` decorator on the directive property that we want to sync with. For example, consider this binding:

```
@HostBinding("readOnly") get busy() {return this.isbusy};
```

When applied to `input`, it will set the `input.readonly` property to `true` when the `isbusy` directive property is `true`.



Note that `readonly` is also an attribute on `input`. What we are referring to here is the `input` property `readonly`.

Attribute binding

Attribute binding binds a directive property to a host component attribute. For example, consider a directive with binding like the following:

```
@HostBinding("attr.disabled") get canEdit(): string { return !this.isAdmin ? "disabled" : null };
```

If applied to input, it will add the disabled attribute on input when the isAdmin flag is false, and clear it otherwise. We follow the same attribute binding notation used in the HTML template here too. The attribute name is prefixed with string literal attr.

We can do something similar with *class* and *style binding* too. Consider the following line: `@HostBinding('class.valid') get valid { return this.control.valid; }`

This line sets up a class binding, and the following line creates a style binding:

```
| @HostBinding("style.borderWidth")
|   get focus(): string { return this.focus?"3px": "1px"};
```

Event binding

Lastly, **event binding** is used to subscribe to the events raised by the host component/element. Consider this example:

```
@Directive({ selector: 'button, div, span, input' })
class ClickTracker {
  @HostListener('click', ['$event.target'])
  onClick(element: any) {
    console.log("button", element, "was clicked");
  }
}
```

This sets up a listener on the host event `click`. Angular will instantiate the preceding directive for every `button`, `div`, `span`, and `input` on the view and set up the host binding with the `onClick` function. The `$event` variable contains the event data for the event raised, and `target` refers to the element/component that was clicked on.

Event bindings work for components too. Consider the following example:

```
@Directive({ selector: 'workout-runner' })
class WorkoutTracker {
  @HostListener('workoutStarted', ['$event'])
  onWorkoutStarted(workout: any) {
    console.log("Workout has started!");
  }
}
```

With this directive, we track the `workoutStarted` event defined on the `workoutRunner` component. The `onWorkoutStarted` function is called when the workout starts, with the details of the started workout.

Now that we understand how these bindings work, we can improve our `BusyIndicatorDirective` implementation.

Implementation two - BusyIndicatorDirective with host bindings

You may have already guessed it! We will use *host property binding* instead of `Renderer` to set styles. Want to give it a try? Go ahead! Clear the existing implementation and try to set up a host binding for the `borderWidth` and `borderColor` style attributes without looking at the following implementation.

This is how the directive will look after the host binding implementation:

```
import {Directive, HostBinding} from '@angular/core';
import {NgModel} from '@angular/forms';

@Directive({ selector: `'[abeBusyIndicator][ngModel]'`})
export class BusyIndicatorDirective {
    private get validating(): boolean {
        return this.model.control != null && this.model.control.pending;
    }
    @HostBinding('style.borderWidth') get controlBorderWidth():
        string { return this.validating ? '3px' : null; }
    @HostBinding('style.borderColor') get controlBorderColor():
        string { return this.validating ? 'gray' : null; }

    constructor(private model: NgModel) { }
}
```

We have moved the `pending` state check into a directive property called `validating` and then used the `controlBorderWidth` and `controlBorderColor` properties for style binding. This is definitely more succinct than our earlier approach! Go test it out.

And if we tell you that this can be done without the need for a custom directive, don't be surprised! This is how we do it, just by using style bindings on the workout name `input`:

```
<input type="text" name="workoutName" ...
[style.borderColor]="name.control.pending ? 'gray' : null"
[style.borderWidth]="name.control.pending ? '3px' : null">
```

We get the same effect!

No, our effort did not go to waste. We did learn about **renderer** and **host binding**. These concepts will come in handy while building directives that provide complex behavior extension instead of just setting element styles.



If you are having a problem with running the code, look at the Git branch `checkpoint6.1` for a working version of what we have done thus far. Or if you are not using Git, download the snapshot of `checkpoint6.1` (a ZIP file) from <http://bit.ly/ng6be-checkpoint-6-1>. Refer to the `README.ma` file in the `trainer` folder when setting up the snapshot for the first time.

The next topic that we are going to take up is, *directive injection*.

Directive injection

Go back a few pages and look at the `BusyIndicatorDirective` implementation that uses the `renderer`, specifically the constructor:

```
| constructor(private model: NgModel ...) { }
```

Angular automatically locates the `NgModel` directive created for the directive element and injects it into `BusyIndicatorDirective`. This is possible because both directives are declared on the same *host element*.

The good news is that we can influence this behavior. Directives created on a parent HTML tree or child tree can also be injected. The next few sections talk about how to inject directives across the component tree, a very handy feature that allows cross-directive communication for directives that have a *common lineage* (in a view).

We will use StackBlitz (<https://stackblitz.com/edit/angular-pzljm3>) to demonstrate these concepts. SlackBlitz is an online IDE to run Angular applications!

To start with, look at the file `app.component.ts`. It has three directives: `Relation`, `Acquaintance`, and `consumer` and this view hierarchy is defined: `<div relation="grand-parent" acquaintance="jack"> <div relation="parent"> <div relation="me" consumer> <div relation="child-1"> <div relation="grandchild-1"></div> </div> <div relation="child-2"></div> </div> </div> </div>`

In the next few sections, we will describe the various ways in which we can inject the different `relation` and `Acquaintance` directives into the `consumer` directive. Check out the browser console for the injected dependencies that we log during the `ngAfterViewInit` life cycle hook.

Injecting directives defined on the same element

Constructor injection by default supports injecting directives defined on the same element. The constructor function just needs to declare the directive type variable that we want to inject: variable:DirectiveType

The `NgModel` injection that we did in `BusyIndicatorDirective` falls under this category. If the directive is not found on the current element, the Angular DI will throw an error, unless we mark the dependency as `@optional`.



Optional dependency

The `@optional` decorator is not limited to directive injection. It's there to mark any type of dependency optional.

From the plunk example, the first injection (in `consumer` directive implementation) injects the `Relation` directive with the `me` attribute (`relation="me"`) into the consumer directive: `constructor(private me:Relation ...)`

Injecting directive dependency from the parent

Prefixing a constructor argument with the `@Host` decorator instructs Angular to search for the dependency on the current element, its parent, or its parents until it reaches the component boundaries (a component with the directive present somewhere in its view hierarchy). Check the second `consumer` injection:

```
|constructor(..., @Host() private myAcquaintance:Acquaintance
```

This statement injects the `Acquaintance` directive instance declared two levels up the hierarchy.



Like the `@option` decorator described previously, the usage of `@Host()` is not limited to directives too. Angular service injection also follows the same pattern. If a service is marked with `@Host`, the search stops at the host component. It does not continue further up the component tree.

The `@SkipSelf` decorator can be used to skip the current element for a directive search.

From the StackBlitz example, this injection injects the `Relation` directive with the `relation` attribute value `parent` (`relation="parent"`) into `consumer`:

```
|@SkipSelf() private myParent:Relation
```

Injecting a child directive (or directives)

If there is a need to inject directive(s) defined on nested HTML into a parent **directive/component**, there are four decorators that can help us:

- `@ViewChild/@ViewChildren`
- `@ContentChild/@ContentChildren`

As these naming conventions suggest, there are decorators to inject a single child directive or multiple children directives:

To understand the significance of `@ViewChild/@ViewChildren` versus `@ContentChild/@ContentChildren`, we need to look at what view and content children are, a topic that we will take up soon. But for now, it's enough to understand that view children are part of a component's own view and content children are **external** HTML injected into the component's view.

Look how, in the StackBlitz example, the `ContentChildren` decorator is used to inject the child `Relation` directive into `consumer`:

```
| @ContentChildren(Relation) private children:QueryList<Relation>;
```

Surprisingly, the data type of the variable `children` is not an array, but a custom class—`QueryList`. The `QueryList` class is not a typical array, but a collection that is kept up to date by Angular whenever dependencies are added or removed. This can happen if the DOM tree is created/destroyed when using structural directives such as `NgIf` or `NgFor`. We will also talk more about `QueryList` in the coming sections.

You may have observed that the preceding injection is not a constructor injection as were the earlier two examples. This is for a reason. The injected directive(s) will not be available until the underlying component/element's content has initialized. For this precise reason, we have the `console.log` statements inside the `ngAfterViewInit` life cycle hook. We should only access the content children post

this life cycle hook execution.

The preceding sample code injects in all three child `relation` objects into the consumer **directive**.

Injecting descendant directive(s)

The standard `@ContentChildren` decorator (or as a matter of fact `@ViewChildren` too) only injects the immediate children of a directive/component and not its descendants. To include all its descendants, we need to provide an argument to `Query`:

```
| @ContentChildren(Relation, {descendants: true}) private  
|   allDescendents:QueryList<Relation>;
```

Passing the `descendants: true` parameter will instruct Angular to search for all descendants.

If you look at the console log, the preceding statement injects in all four descendants.

The Angular DI, while it seems simple to use, packs a lot of functionality. It manages our services, components, and directives and provides us with the right stuff in the right place at the right time. Directive injection in components and other directives provides a mechanism for directives to communicate with each other. Such injections allow one directive to access the public API (public functions/properties) of another directive.

It's now time to explore something new. We are going to build an Ajax button component that allows us to inject an external view into the component, a process also known as **content transclusion**.

Building an Ajax button component

When we save/update an exercise or workout, there is always the possibility of duplicate submission (or duplicate `POST` requests). The current implementation does not provide any feedback as to when the save/update operation started and when it is completed. The user of an app can knowingly or unknowingly click on the Save button multiple times due to the lack of visual clues.

Let's try to solve this problem by creating a specialized button—an *Ajax button* that gives some visual clues when clicked on and also stops duplicate Ajax submissions.

The button component will work on these lines. It takes a function as input. This input function (input parameter) should return a promise pertaining to the remote request. On clicking on the button, the button internally makes the remote call (using the input function), tracks the underlying promise, waits for it to complete, and shows some busy clues during this activity. Also, the button remains disabled until the remote invocation completes to avoid duplicate submission.



The companion code base for the following section is Git branch `checkpoint6.2`. You can work along with us, or check out the implementation available in the branch. Or if you are not using Git, download the snapshot of `checkpoint6.2` (a ZIP file) from the GitHub location <http://bit.ly/ng6be-checkpoint-6-2>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Let's create the component outline to make things clearer. Use the following command to create an `ajax-button` component under the application's shared module (`src/app/shared`) and **then export the component** from the `SharedModule`:

```
| ng generate component ajax-button -is
```

Update the component definition too and import them from `@angular/core`:

```
export class AjaxButtonComponent implements OnInit {
  busy: boolean = null;
  @Input() execute: any;
  @Input() parameter: any;
}
```

And add the following HTML template to `ajax-button.component.html`:

```
<button [attr.disabled]="busy" class="btn btn-primary">
  <span [hidden]="!busy">
    <div class="ion-md-cloud-upload spin"></div>
  </span>
  <span>Save</span>
</button>
```

The component (`AjaxButtonComponent`) takes two property bindings, `execute` and `parameter`. The `execute` property points to the function that is invoked on the Ajax button click. The `parameter` is the data that can be passed to this function.

Look at the usage of the `busy` flag in the view. We disable the button and show the spinner when the `busy` flag is set. Let's add the implementation that makes everything work. Add this code to the `AjaxButtonComponent` class:

```
@HostListener('click', ['$event'])
onClick(event: any) {
  const result: any = this.execute(this.parameter);
  if (result instanceof Promise) {
    this.busy = true;
    result.then(
      () => { this.busy = null; },
      (error: any) => { this.busy = null; });
  }
}
```

We set up a *host event binding* to the click event on `AjaxButtonComponent`. Anytime the `AjaxButtonComponent` component is clicked on, the `onClick` function is invoked.

The `HostListener` import needs to be added to the '`@angular/core`' module.

The `onClick` implementation calls the `input` function with a lone parameter as `parameter`. The result of the invocation is stored in the `result` variable.

The `if` condition checks whether the `result` is a `Promise` object. If yes, the `busy` indicator is set to `true`. The button then waits for the promise to get resolved, using the `then` function. Irrespective of whether the promise is resolved with `success` or `error`, the `busy` flag is set to `null`.

 The reason the `busy` flag is set to `null` and not `false` is due to this attribute binding `[attr.disabled]="busy"`. The `disabled` attribute will not be removed unless `busy` is `null`. Remember that in HTML, `disabled="false"` does not enable the button. The attribute needs to be removed before the button becomes clickable again.

If we are confused about this line:

```
| const result: any = this.execute(this.parameter);
```

Then you need to look at how the component is used. Open `workout.component.html` and replace the `save` button HTML with the following:

```
| <abe-ajax-button [execute]="save" [parameter]="f"></abe-ajax-button>
```

The `workout.save` function binds to `execute`, and `parameter` takes the `FormControl` object `f`.

We need to change the `save` function in the `Workout` class to return a promise for `AjaxButtonComponent` to work. Change the `save` function implementation to the following:

```
save = (formWorkout: any): Promise<Object | WorkoutPlan> => {
  this.submitted = true;
  if (!formWorkout.valid) { return; }
  const savePromise = this.workoutBuilderService.save().toPromise();

  savePromise.then(
    result => this.router.navigate(['/builder/workouts']),
    err => console.error(err)
  );
  return savePromise;
}
```

The `save` function now returns a *promise* that we build by calling the `toPromise` function on the *observable* returned from the call to `workoutBuilderService.save()`.



Make note of how we define the `save` function as an instance function (with the use of the arrow operator) to create a closure over this. It's something we did earlier while building the remote validator directive.

Time to test our implementation! Refresh the application and open the create/edit workout view. Click on the Save button and see the Ajax button in action:



The preceding animation may be short-lived as we navigate back to the workout list page post save. We can temporarily disable the navigation to see the new changes.

We started this section with the aim of highlighting how external elements/components can be transcluded into a component. Let's do it now!

Transcluding external components/elements into a component

From the very start, we need to understand what **transclusion** means. And the best way to understand this concept would be to look at an example.

No component that we have built thus far has borrowed content from outside. Not sure what this means?

Consider the preceding `AjaxButtonComponent` example in `workout.component.html`: `<ajax-button [execute]="save" [parameter]="f"></ajax-button>`

What if we change the `ajax-button` usage to the following?

```
| <ajax-button [execute]="save" [parameter]="f">Save Me!</ajax-button>
```

Will the `Save Me!` text show up on the button? It will not try it!

The `AjaxButtonComponent` component already has a template, and it rejects the content we provide in the preceding declaration. What if we can somehow make the content (`Save Me!` in the preceding example) load inside the `AjaxButtonComponent`? This act **of injecting** an external view fragment into the component's view is what we call **transclusion**, and the framework provides the necessary constructs to enable transclusions.

It's time to introduce two new concepts, *content children* and *view children*.

Content children and view children

To define it succinctly, the HTML structure that a component defines internally (using `template` or `templateUrl`) is the **view children** of the component. However, the HTML view provided as part of the component usage added to the host element (such as `<ajax-button>Save Me!</ajax-button>`), defines the **content children** of the component.

By default, Angular does not allow *content children* to be embedded as we saw before. The `Save Me!` text was never emitted. We need to explicitly tell Angular where to emit the *content children* inside the *component view template*. To understand this concept, let's fix the `AjaxButtonComponent` view. Open `ajax-button.component.ts` and update the view template definition to the following:

```
<button [attr.disabled]="busy" class="btn btn-primary">
  <span [hidden]="!busy">
    <ng-content select="[data-animator]"></ng-content>
  </span>
  <ng-content select="[data-content]"></ng-content>
</button>
```

The two `ng-content` elements in the preceding view define the *content injection locations*, where the content children can be injected/transcluded. The `selector` property defines the *CSS selector* that should be used to locate the content children when injected into the main host.

It starts to make more sense as soon as we fix the `AjaxButtonComponent` usage in `workout.component.html`. Change it to the following:

```
<ajax-button [execute]="save" [parameter]="f">
  <div class="ion-md-cloud-upload spin" data-animator></div>
  <span data-content>Save</span>
</ajax-button>
```

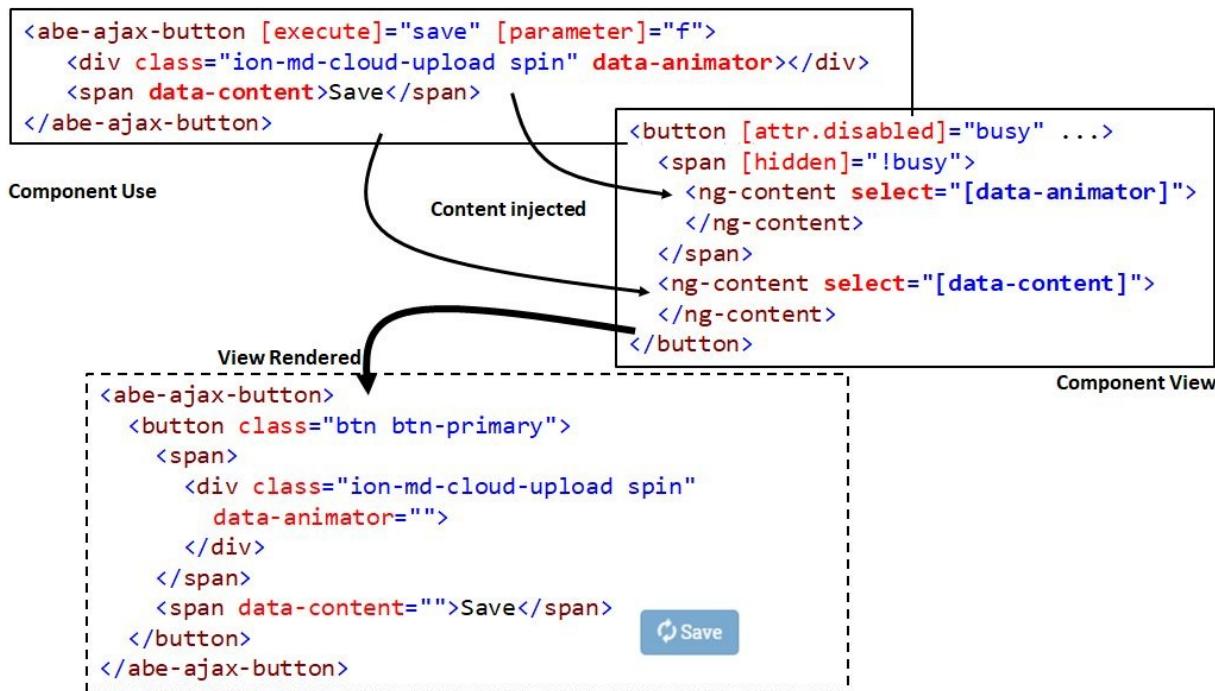
The `span` with `data-animator` is injected into the `ng-content` with the `select=[data-animator]` property and the other `span` (with the `data-content` attribute) is injected into the second `ng-content` declaration.

Refresh the application again and try to save a workout. While the end result is the same, the resultant view is a combination of multiple view fragments: one

part for component definition (*view children*) and another part for component usage (*content children*).

The following diagram highlights this difference for the rendered

AjaxButtonComponent:



The `ng-content` can be declared without the `selector` attribute. In such a scenario, the complete content defined inside the component tag is injected.

Content injection into an existing component view is a very powerful concept. It allows the component developer to provide extension points that the component consumer can readily consume and customize the behavior of the component, that too in a controlled manner.

The content injections that we defined for the `AjaxButtonComponent` allow the consumer to change the busy indicator animation and the button content, keeping the behavior of the button intact.

Angular's advantages do not end here. It has the capability to inject *content children* and *view children* into the *component code/implementation*. This allows the component to interact with its content/view children and control their behavior too.

Injecting view children using @ViewChild and @ViewChildren

In [Chapter 3](#), *More Angular 2 – SPA, Routing, and Data Flows in Depth*, we made use of something similar, *view children injection*. To recall what we did, let's look at the relevant parts of the `WorkoutAudioComponent` implementation.

The view definition looked as follows:

```
| <audio #ticks="MyAudio" loop src="/assets/audio/tick10s.mp3"></audio>
| <audio #nextUp="MyAudio" src="/assets/audio/nextup.mp3"></audio>
| <audio #nextUpExercise="MyAudio" [src]="/assets/audio/' + nextupSound"></audio>
| // Some other audio elements
```

And the injection looked as follows:

```
| @ViewChild('ticks') private _ticks: MyAudioDirective;
| @ViewChild('nextUp') private _nextUp: MyAudioDirective;
| @ViewChild('nextUpExercise') private _nextUpExercise: MyAudioDirective;
```

The directive (`MyAudioDirective`) associated with the `audio` tag was injected into the `WorkoutAudio` implementation using the `@ViewChild` decorator. The parameters passed to `@ViewChild` are the *template variable* names (such as `tick`) used to locate the element in the view definition. The `WorkoutAudio` component then used these audio directives to control the audio playback for *7 Minute Workout*.

While the preceding implementation injects `MyAudioDirective`, even child components can be injected. For example, instead of using `MyAudioDirective`, suppose we build a `MyAudioComponent`, something like the following:

```
| @Component({
|   selector: 'my-audio',
|   template: '<audio ...></audio>',
| })
| export class MyAudioComponent {
|   ...
| }
```

We can then use it instead of the `audio` tag:

```
| <my-audio #ticks loop
|   src="/static/audio/tick10s.mp3"></my-audio>
```

The injection would still work.

What happens if there is more than one directive/component of the same type defined on the component view? Use the `@ViewChildren` decorator. It allows you to query injections of one type. The syntax for the use of `@ViewChildren` is as follows:

```
| @ViewChildren(directiveType) children: QueryList<directiveType>;
```

This injects all the view children of type `directiveType`. For the `WorkoutAudio` component example stated previously, we can use the following statement to get hold of all `MyAudioDirective`:

```
| @ViewChildren(MyAudioDirectives) private all: QueryList<MyAudioDirectives>;
```

The `ViewChildren` decorator can also take a list of comma-separated selectors (*template variable names*) instead of type. For example, to select multiple `MyAudioDirective` instances in the `workoutAudio` component, we can use the following:

```
| @ViewChildren('ticks, nextUp, nextUpExercise, halfway, aboutToComplete') private all:  
| QueryList<MyAudioDirective>;
```

The `QueryList` class is a special class provided by Angular. We introduced `QueryList` in the *Injecting descendant directive(s)* section earlier in the chapter. Let's explore `QueryList` further.

Tracking injected dependencies with QueryList

For components that require multiple components/directives to be injected (using either `@ViewChildren` or `@ContentChildren`), the dependency injected is a `QueryList` object.

The `QueryList` class is a *read-only collection* of injected components/directives. Angular keeps this collection in sync based on the current state of the user interface.

Consider, for example, the `workoutAudio` directive view. It has five instances of `MyAudioDirective`. Hence, for the following collection, we will have five elements:

```
| @ViewChildren(MyAudioDirective) private all: QueryList<MyAudioDirective>;
```

While the preceding example does not highlight the syncing part, Angular can track components/directives being added or removed from the view. This happens when we use content generation directives such as `ngFor`.

Take this hypothetical template for example:

```
| <div *ngFor="let audioData of allAudios">
|   <audio [src]="audioData.url"></audio>
| </div>
```

The number of `MyAudioDirective` directives injected here equals the size of the `allAudios` array. During the program's execution, if elements are added to or removed from the `allAudios` array, the directive collection is also kept in sync by the framework.

While the `QueryList` class is not an array, it can be iterated over (as it implements the **ES6 iterable interface**) using the `for (var item in queryListObject)` syntax. It also has some other useful properties, such as `length`, `first`, and `last`, which can come in handy. Check out the framework documentation (<http://bit.ly/ng2-querylist-class>) for more details.

From the preceding discussion, we can conclude that `QueryList` saves the component developer a lot of boilerplate code that would be required if tracking had to be done manually.



View children access timing

View children injections are not available when the component/directive initializes. Angular makes sure that the view children injections are available to the component no later than the `ngAfterViewInit` life cycle event. Make sure you access the injected components/directives only when (or after) the `ngAfterViewInit` event has fired.

Let's now look at content children injection, which is almost similar, except for a few minor differences.

Injecting content children using @ContentChild and @ContentChildren

Angular allows us to inject *content children* too, using a parallel set of attributes: `@ContentChild` to inject a specific content child and `@ContentChildren` to inject content children of a specific type.

If we look back at the usage of `AjaxButtonComponent`, its content children spans can be injected into `AjaxButtonComponent` implementation by doing this:

```
| @ContentChild('spinner') spinner:ElementRef;  
| @ContentChild('text') text:ElementRef;
```

And adding template variables onto the corresponding spans in `workout.component.html`:

```
| <div class="ion-md-cloud-upload spin" data-animator #spinner></div>  
| <span data-content #text>Save</span>
```

While the preceding injection is `ElementRef`, it could have been a component too. Had we defined a component for spinner, such as:

```
| <ajax-button>  
|   <busy-spinner></busy-spinner>  
|   ...  
| </ajax-button>
```

We could have injected it too using the following:

```
| @ContentChild(BusySpinner) spinner: BusySpinner;
```

The same holds true for directives too. Any directive declared on `AjaxButtonComponent` can be injected into the `AjaxButtonComponent` implementation. For the preceding case, since the transcluded elements are standard HTML elements, we injected `ElementRef`, a wrapper that Angular creates for any HTML element.



Like view children, Angular makes sure that the content children references are bound to the variables injected before the `ngAfterContentInit` life cycle event.

While we are on the subject of injecting dependencies, let's talk about some variations around *injecting services into components*.

Dependency injection using viewProvider

We are already familiar with the mechanism of DI registration in Angular, where we register a dependency at the global level by adding it to any module declaration.

Or we can do it at a component level using the `providers` property on the `@Component` decorator:

```
| providers:[WorkoutHistoryTracker, LocalStorage]
```

 Just to avoid confusion, we are now talking about injecting dependencies other than directive/component objects. Directives/components are registered in the declarations array of a module before they can be injected using decorator hints such as `@Query`, `@ViewChild`, `@ViewChildren`, and a few others.

Dependencies registered at the component level are available for its `view children` and `content children` and their descendants.



Before we proceed, we hope that the distinction between `view` and `content children` is crystal clear to everyone. If in doubt, refer to the [Content children and view children](#) section again.

Let's take an example from [Chapter 4](#), *Building Personal Trainer*. The `WorkoutBuilderService` service was registered at the app level in the workout builder module (`WorkoutBuilderModule`):

```
| providers: [ExerciseBuilderService, ...  
|   WorkoutBuilderService]);
```

This allows us to inject `WorkoutBuilderService` across the app in order to build workouts and while running workouts. Instead, we could have registered the service at the `WorkoutBuilderComponent` level since it is the parent of all `workout/exercise creation components`, something like the following:

```
@Component({  
  template: `...`  
  providers:[ WorkoutBuilderService ]  
})  
export class WorkoutBuilderComponent {
```

This change would disallow injecting `WorkoutBuilderService` in `WorkoutRunner` or any component related to workout execution.



*What if the `WorkoutBuilderService` service is registered at the app level as well as at the component level (as shown in the preceding example)? How does the injection happen? From our experience, we know that Angular will inject a different instance of the `WorkoutBuilderService` service into `WorkoutBuilderComponent` (and its descendants), while other parts of the application (`Workout runner`) will get the global dependency. Remember **hierarchical injectors!***

Angular does not stop here. It provides some further scoping of dependencies using the `viewProviders` property. The `viewProviders` property, available on the `@component` decorator, allows the registering of dependencies that can be injected only in the view children.

Let's consider the `AjaxButtonComponent` example again, and a simple directive implementation called `MyDirective`, to elaborate on our discussion:

```
 @Directive({
  selector: '[myDirective]',
})
export class MyDirective {
  constructor(service:MyService) { }
  ...
}
```

The `MyDirective` class depends upon a service, `MyService`.

To apply this directive to the `button` element in the `AjaxButtonComponent` template, we need to register the `MyService` dependency too (assuming that `MyService` has not been registered globally):

```
 @Component({
  selector: 'ajax-button',
  template: `<button [attr.disabled]="busy" ...>
    myDirective
    ...
    <button>`,
  providers:[MyService],
  ...
})
```

Since `MyService` is registered with `AjaxButtonComponent`, `MyDirective` can be added to its content children too. Hence the `myDirective` application on *spinner HTML* will also work (the code in `workout.component.html`):

```
| <div class="ion-md-cloud-upload spin" data-animator myDirective></div>
```

But changing the `providers` property to `viewProviders`:

```
| viewProviders:[MyService]
```

Will fail the `MyService` injection for the `AjaxButtonComponent`'s content children (the `div` in the preceding code), with a DI error in the console.



Dependencies registered with `viewProviders` are invisible to its content children.

This dependency scoping for *the view and content children* may not seem useful at first sight, but it does have its benefits. Imagine we are building a reusable component that we want to package and deliver to developers for consumption. If the component has a service dependency that it prepackages too, we need to be extra cautious. If such a component allows content injection (content children), the dependent service is widely exposed if *provider-based* registration is used on the component. Any content children can get hold of the service dependency and use it, leading to undesirable consequences. By registering the dependency using `viewProvider`, only the component implementation and its child views have access to the dependency, providing the necessary layer of encapsulation.

Yet again, we are amazed by the flexibility and level of customization that the DI framework provides. While it may be intimidating for starters, once we start building more and more components/directives with Angular, we will always find areas where these concepts make our implementation simpler.

Let's shift our focus to the third classification of directives: *structural directives*.

Understanding structural directives

While we will often be using structural directives, such as `NgIf` and `NgFor`, there is seldom a need to creating a structural directive. Think carefully. If we need a new view, we create a component. If we need to extend an existing element/component, we use a directive. Whereas the most common use of structural directives is to clone a piece of a view (also called a *template view*) and then, based on some conditions:

- Either inject/destroy these templates (`NgIf` and `NgSwitch`)
- Or duplicate these templates (`NgFor`)

Any behavior implemented using structure directives will inadvertently fall into either of these two categories.

Given this fact, instead of building our own structural directive, let's look at the source code of the `NgIf` implementation.

The following is an excerpt from the `NgIf` directive that is of interest to us. We have ignored the `ngIfElse` parts from the excerpt intentionally:

```
@Directive({selector: '[ngIf]'})
export class NgIf {
  constructor(private _viewContainer: ViewContainerRef, templateRef:
TemplateRef<NgIfContext>) {
    this._thenTemplateRef = templateRef;
  }

  @Input()
  set ngIf(condition: any) {
    this._context.$implicit = this._context.ngIf = condition;
    this._updateView();
  }
  private _updateView() {
    if (this._context.$implicit) {
      if (!this._thenViewRef) {
        this._viewContainer.clear();
        this._elseViewRef = null;
        if (this._thenTemplateRef) {
          this._thenViewRef =
            this._viewContainer.createEmbeddedView(this._thenTemplateRef,
this._context);
        }
      }
    }
  ...
}
```

No magic here, just a simple structural directive that checks a Boolean condition (`this._context.$implicit`) to create/destroy the view!

The first if condition above check, if the condition `this._context.$implicit` is true. The next condition makes sure that the view is already not rendered by checking the variable `_thenViewRef`. We only want to flip the view if `this._context.$implicit` translates from `false` to `true`. If both if's conditions are true the existing view is cleared (`this._viewContainer.clear()`) and the reference to the else view is cleared. The innermost if condition makes sure that the if's template reference is available. Finally, the code calls `_viewContainer.createEmbeddedView` to render (or re-render) the view.

It's not difficult to understand how the directive works. What needs to be detailed are the two new injections, `ViewContainerRef` (`_viewContainer`) and `TemplateRef` (`_templateRef`).

TemplateRef

The `TemplateRef` class (`_templateRef`) stores the reference to the template that the structural directive is referring to. Remember the discussion on structural directives from [chapter 2, Building Our First App - 7 Minute Workout?](#) All structural directives take a template HTML that they work on. When we use a directive such as `NgIf`:

```
| <h3 *ngIf="currentExercise.exercise.name=='rest'">
|   ...
| </h3>
```

Angular internally **translates** this declaration to the following:

```
| <ng-template [ngIf]="currentExercise.exercise.name=='rest'">
|   <h3> ... </h3>
| </ng-template>
```

This is the template that structural directives work with, and `_templateRef` points to this template.

The other injection is `viewContainerRef`.

ViewContainerRef

The `viewContainerRef` class points to the container where templates are rendered. This class has a number of handy methods for managing views. The two functions that `NgIf` implementation uses, `createEmbeddedView` and `clear`, are there to add and remove the template HTML.

The `createEmbeddedView` function takes the template reference (again injected into the directive) and renders the view.

The `clear` function destroys the element/component already injected and clears the view container. Since every component and its children referenced inside the template (`TemplateRef`) are destroyed, all the associated bindings also cease to exist.

Structural directives have a very specific area of application. Still, we can do a lot of nifty tricks using the `TemplateRef` and `viewContainerRef` classes.

We can implement a structural directive that, depending on the user role, shows/hides the view template.

Consider this example of a hypothetical structural directive, `forRoles`:

```
| <button *forRoles="admin">Admin Save</button>
```

The `forRoles` directive will not render the button if the user does not belong to the `admin` role. The core logic would look something like the following:

```
| if(this.loggedInUser.roles.indexOf(this.forRole) >=0){  
|     this.viewContainer.createEmbeddedView(this.templateRef);  
| }  
| else {  
|     this.viewContainer.clear();  
| }
```

The directive implementation will need some sort of service that returns the logged-in user's details. We will leave the implementation for such a directive to the readers.

What the `forRoles` directive does can also be done using `NgIf`:

```
| <button *ngIf="loggedInUser.roles.indexOf('admin')>=0">Admin Save</button>
```

But the `forRoles` directive just adds to the template's readability with clear intentions.

A fun application of structural directives may involve creating a directive that just duplicates the template passed to it. It would be quite easy to build one; we just need to call `createEmbeddedView` twice:

```
ngOnInit() {  
  this.viewContainer.createEmbeddedView(this._templateRef);  
  this.viewContainer.createEmbeddedView(this._templateRef);  
}
```

Another fun exercise!

The `viewContainerRef` class also has some other functions that allow us to inject components, get the number of embedded views, reorder the view, and so on and so forth. Look at the framework documentation for `viewContainerRef` (<http://bit.ly/view-container-ref>) for more details.

That completes our discussion on structural directives and it's time to start something new!

The components that we have built thus far derive their styles (CSS) from the common *bootstrap style sheet* and some custom styles defined in `app.css`. Angular has much more to offer in this area. A truly reusable component should be completely self-contained, in terms of both behavior and user interface.

Component styling and view encapsulation

A longstanding problem with web app development is the lack of encapsulation when it comes to DOM element behavior and styles. We cannot segregate one part of the application HTML from another through any mechanism.

In fact, we have too much power at our disposal. With libraries such as jQuery and powerful *CSS selectors*, we can get hold of any DOM element and change its behavior. There is no distinction between our code and any external library code in terms of what it can access. Every single piece of code can manipulate any part of the rendered DOM. Hence, the encapsulation layer is broken. A badly written library can cause some nasty issues that are hard to debug.

The same holds true for CSS styling too. Any UI library implementation can override global styles if the library implementation wants to do so.

These are genuine challenges that any library developer faces when building reusable libraries. Some emerging web standards have tried to address this issue by coming up with concepts such as **web components**.

Web components, in simple terms, are reusable user interface widgets that encapsulate their state, style, user interface, and behavior. Functionality is exposed through well-defined APIs, and the user interface parts are encapsulated too.

The *web component* concept is enabled by four standards:

- HTML templates
- Shadow DOM
- Custom elements
- HTML imports

For this discussion, the technology standard we are interested in is **Shadow DOM**.

Overview of Shadow DOM

Shadow DOM is like a parallel DOM tree hosted inside a component (*an HTML element, not to be confused with Angular components*), hidden away from the main DOM tree. No part of the application has access to this shadow DOM other than the component itself.

It is the implementation of the Shadow DOM standard that allows view, style, and behavior encapsulation. The best way to understand Shadow DOM is to look at HTML5 `video` and `audio` tags.

Have you ever wondered how this `audio` declaration:

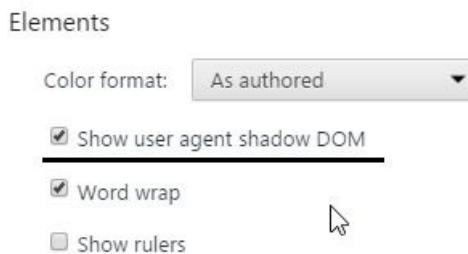
```
|<audio src="/static/audio/nextup.mp3" controls></audio>
```

Produces the following?



It is the **browser** that generates the underlying Shadow DOM to render the audio player. Surprisingly, we can even look at the generated DOM! Here is how we do it:

- Take the preceding HTML, create a dummy HTML page, and open it in Chrome.
- Then open the Developer tools window (*F12*). Click on the Setting icon on the upper-left corner.
- On the General settings, click on the checkbox, as highlighted in the following screenshot, to enable the inspection of Shadow DOM:



Refresh the page, and if we now inspect the generated `audio` HTML, the Shadow DOM shows up:

```
▼<audio controls src="/static/audio/nextup.mp3">
  ▼#shadow-root (user-agent)
    ▼<div pseudo="-webkit-media-controls">
      ▼<div pseudo="-webkit-media-controls-overlay-enclosure">
        ▶<input type="button" style="display: none;">...
      </div>
    ▼<div pseudo="-webkit-media-controls-enclosure">
      ▼<div pseudo="-webkit-media-controls-panel">
        ▶<input type="button" pseudo="-webkit-media-controls-play-button">...
        ▶<input type="range" step="any" pseudo="-webkit-media-controls-timeline" max="0.972">...
        <div pseudo="-webkit-media-controls-current-time-display" style="display: none;">0:00</div>
        <div pseudo="-webkit-media-controls-time-remaining-display" style="display: none;">0:00</div>
        ▶<input type="button" pseudo="-webkit-media-controls-mute-button">...
        ▶<input type="range" step="any" max="1" pseudo="-webkit-media-controls-volume-slider">...
        ▶<input type="button" pseudo="-webkit-media-controls-toggle-closed-captions-button" style="display: none;">...
        ▶<input type="button" style="display: none;">...
        ▶<input type="button" pseudo="-webkit-media-controls-fullscreen-button" style="display: none;">...
      </div>
    </div>
  </div>
</audio>
```

Under `shadow-root`, there is a whole new world that the other part of the page and script do not have access to.



*In the Shadow DOM realm, `shadow-root` (`#shadow-root` in the preceding code) is the root node for the generated DOM, hosted inside a **shadow host** (in this case the `audio` tag). When the browser renders this element/component, what gets rendered is the content from the shadow root and not the shadow host.*

From this discussion, we can conclude that Shadow DOM is a parallel DOM created by the browser that encapsulates the *markup, style, and behavior* (DOM manipulation) of an HTML element.



This was a gentle introduction to Shadow DOM. To learn more about how Shadow DOM works, we recommend this series by Rob Dodson: <http://bit.ly/shadow-dom-intro>

But what has all this got to do with Angular? As it turns out, Angular components support some sort of view encapsulation too! This allows us to isolate styles for Angular components too.

Shadow DOM and Angular components

To understand how Angular employs the concept of Shadow DOM, we will first have to learn about styling Angular components.

When it comes to styling the apps built as part of this book, we have taken a conservative approach. Be it *Workout Builder* or the *Workout Runner (7 Minute Workout)* app, all the components that we built derive their style from *bootstrap* CSS and from custom styles defined in `app.css`. No component has defined its own style.

While this adheres to the standard practices of web app development, sometimes we do need to deviate. This is especially true when we are building self-contained, packaged, and reusable components.

Angular allows us to define styles specific to a component by using the `style` (for inline style) and `styleUrl` (external style sheet) properties on the `@Component` decorator. Let's play around with the `style` property and see what Angular does.

We will use the `AjaxButtonComponent` implementation as our playground for the next exercise. But before doing that, let's look at the `AjaxButtonComponent` HTML as it stands now. The HTML tree for `AjaxButtonComponent` looks as follows:

```
<abe-ajax-button>
  <button class="btn btn-primary">
    <span hidden>
      <div class="ion-md-cloud-upload spin"
           data-animator="">
      </div>
    </span>
    <span data-content="">Save</span>
  </button>
</abe-ajax-button>
```

Let's override some styles using the `styles` property:

```
@Component({
  ...
  styles:[`
```

```
    background: green;  
  }]  
})
```

The preceding *CSS selector* sets the `background` property to `green` for all HTML buttons. Save the preceding style and refresh the work builder page. The button style has been updated. No surprises here? Not true, there are some! Look at the generated HTML:

```
▼<abe-ajax-button _ngcontent-c1 ng-reflect-execute="function (formWorkout) {  
  " ng-reflect-parameter="[object Object]">  
  ▼<button _ngcontent-c1 class="btn btn-primary">  
    ►<span _ngcontent-c1 hidden>...</span>  
    <span data-content>Save</span>  
  </button>  
</abe-ajax-button>
```

There are some new attributes added to a number of HTML elements. And where have the recently defined styles landed? At the very top, inside the `head` tag:

```
<style>button[_ngcontent-c1] {  
  background: green;  
}</style>
```

The style defined in the `head` section has an extra scope with the `_ngcontent-c1` attribute (the attribute name may differ in your case). This scoping allows us to style `AjaxButtonComponent` independently and it cannot override any global styles.



Angular does the same even if we use the `styleUrls` property. Suppose we had embedded the same CSS in an external CSS file and used this: `styleUrls: ['static/css/ajax-button.css']` Angular would have still in-lined the styles into the `head` section, by fetching the CSS, parsing it, and then injecting it.

The styles that by definition, should have affected the appearance of all the buttons in the application, have had no effect. Angular has scoped these styles.



This scoping makes sure that the component styles do not mess with the already defined style, but the reverse is not true. Global styles will still affect the component unless overridden in the component itself.

This scoped style is the result of Angular trying to emulate the Shadow DOM paradigm. The styles defined on the component never leak into the global styles. All this awesomeness without any effort!





If you are building components that define their own styles and want a degree of isolation, use the component's `style/styleurl` property instead of using the old-school approach of having a common CSS file for all styles.

We can further control this behavior by using a `@Component` decorator property called **encapsulation**. The API documentation for this property mentions:

encapsulation: `viewEncapsulation` Specify how the template and the styles should be encapsulated. The default is `viewEncapsulation.Emulated` if the view has styles, otherwise `viewEncapsulation.None`.

As we can see, as soon as we set the style on the component, the encapsulation effect is `Emulated`. Otherwise, it is `None`.



If we explicitly set `encapsulation` to `viewEncapsulation.None`, the scoping attributes are removed and the styles are embedded in the head section as normal styles.

And then there is a third option, `viewEncapsulation.Native`, in which Angular actually creates Shadow DOM for the components view. Set the `encapsulation` property on the `AjaxButtonComponent` implementation to `viewEncapsulation.Native`, and now look at the rendered DOM:

```
▼<abe-ajax-button ng-reflect-execute="function (formWorkout) {  
  " ng-reflect-parameter="[object Object]">  
▼#shadow-root (open)  
  <style></style>  
  <style>  
    button {  
      background: green;  
    }</style>  
  ▼<button class="btn btn-primary">  
    ▼<span hidden>  
      <div class="ion-md-cloud-upload spin" data-animator></div>  
    </span>  
    <span data-content>Save</span>  
  </button>  
</abe-ajax-button>
```

AjaxButtonComponent now has a shadow DOM! This also implies that the complete styling of the button is lost (style derived from bootstrap CSS) and the button needs to now define its own style.

Angular goes to great lengths to make sure that the components we develop can work independently and are reusable. Each component already has its own template and behavior. In addition to that, we can also encapsulate component

styles, allowing us to create robust, standalone components.

This brings us to the end of the chapter, and it's time to wrap up the chapter with what we've learned.

Summary

As we conclude this chapter, we now have a better understanding of how directives work and how to use them effectively.

We started the chapter by building a `RemoteValidatorDirective`, and learned a lot about Angular's support for *asynchronous validations*.

Next in line was `BusyIndicatorDirective`, again an excellent learning ground. We explored the **renderer** service, which allows component view manipulation in a platform-agnostic way. We also learned about **host bindings**, which let us bind to a host element's events, attributes, and properties.

Angular allows directives declared across the view lineage to be injected into the lineage. We dedicated a few sections to understanding this behavior.

The third directive (component) that we created was `AjaxButtonComponent`. It helped us understand the critical difference between **content children** and **view children** for a component.

We also touched upon structural directives, where we explored the `ngIf` platform directive.

Lastly, we looked at Angular's capabilities in terms of view encapsulation. We explored the basics of Shadow DOM and learned how the framework employs the Shadow DOM paradigm to provide view plus style encapsulation.

The next chapter is all about testing Angular apps, a critical piece in the complete framework offering. The Angular framework was built with testability in mind. The framework constructs and the tooling support make automated testing in Angular easy. More about this in the next chapter....

Testing Personal Trainer

Unless you are a superhero who codes perfectly, you need to test what you build. Also, unless you have loads of free time to test your application again and again, you need some test automation.

When we say Angular was built with testability in mind, we really mean it. It has a strong **Dependency Injection (DI)** framework, some good mock constructs, and awesome tools that make testing in an Angular app a fruitful endeavor.

This chapter is all about testing and is dedicated to testing what we have built over the course of this book. We test everything from components to pipes, services, and our app directives.

The topics we cover in this chapter include:

- **Understanding the big picture:** We will try to understand how testing fits into the overall context of Angular app development. We will also discuss the types of testing Angular supports, including unit and **end-to-end (E2E)** testing.
- **Overview of tools and frameworks:** We will cover the tools and frameworks that help in both unit and end-to-end testing with Angular. These include **Karma** and **Protractor**.
- **Writing unit tests:** You will learn how to do unit testing with Angular using **Jasmine** and **Karma** inside a browser. We will unit test what we have built in the last few chapters. This section also teaches us how to unit-test various Angular constructs, including pipes, components, services, and directives.
- **Creating end-to-end tests:** Automated end-to-end tests work by mimicking the behavior of the actual user through browser automation. You will learn how to use Protractor combined with WebDriver to perform end-to-end testing.

Let the testing begin!

As you start reading this chapter, we suggest that you download the code for `checkpoint 7.1`. It is available on GitHub (<https://github.com/chandermani/angular6byexample>) for everyone to download.

 Checkpoints are implemented as branches in GitHub. If you are not using Git, download the snapshot of `checkpoint7.1` (a ZIP file) from this GitHub location: <https://github.com/chandermani/angular2-byexample/archive/checkpoint7.1.zip>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

This checkpoint contains the tests that were generated by the Angular CLI as we created components, services, pipes, and directives in the earlier chapters. We have made minor changes in these tests so that they all pass. For the most part these are basic "Hello World" tests that confirm the creation of a component or other Angular construct. We will not be covering these tests in this chapter, but encourage you to review them.

The need for automation

The size and complexity of apps being built for the web are growing with each passing day. The plethora of options that we now have to build web apps is just mind-boggling. Add to this the fact that the release cycles for products/apps have shrunk drastically from months to days, or even multiple releases per day! This puts a lot of burden on software testing. There is too much to be tested. Multiple browsers, multiple clients and screen sizes (desktop and mobile), multiple resolution, and so on.

To be effective in such a diverse landscape, automation is the key. *Automate everything that can be automated* should be our mantra.

Testing in Angular

The Angular team realized the importance of testability and hence created a framework that allowed easy testing (automated) for apps built on it. The design choice of using DI constructs to inject dependencies everywhere has helped. This will become clear as the chapter progresses and we build a number of tests for our apps. However, before that, let's understand the types of testing that we target when building apps on this platform.

Types of testing

There are broadly two forms of testing that we do for a typical Angular app:

- **Unit testing:** Unit testing is all about testing a component in isolation to verify the correctness of its behavior. Most of the dependencies of the component under test need to be replaced with mock implementations to make sure that the unit tests do not fail due to failure in a dependent component.
- **End-to-end testing:** This type of testing is all about executing the application like a real end user and verifying the behavior of the application. Unlike unit testing, components are not tested in isolation. Tests are done against a running system in real browsers, and assertions are done based on the state of the user interface and the content displayed.

Unit testing is the first line of defense against bugs, and we should be able to iron out most issues with code during unit testing. But unless E2E is done, we cannot confirm that the software is working correctly. Only when all the components within a system interact in the desired manner can we confirm that the software works; hence, E2E testing becomes a necessity.

You can view these two types of testing like a pyramid with E2E testing on the top and unit testing on the bottom. The pyramid indicates that the number of unit tests you write should substantially exceed the number of E2E tests. The reason is that with unit tests you are breaking your application down into small testable units, whereas with integration tests you are spanning multiple components from the UI through to the backend. Also setting up E2E tests tends to be more complicated than unit tests.

Who writes unit and E2E tests and when are they written are important questions to answer.

Testing – who does it and when?

Traditionally, E2E testing was done by the **Quality Assurance (QA)** team and developers were responsible for unit-testing their code before submitting. Developers did some amount of E2E testing too, but overall the E2E testing process was manual.

With the changing landscape, modern testing tools, especially on the web front, have allowed developers to write automated E2E tests themselves and execute them against any deployment setup (such as development/stage/production). Tools such as Selenium, together with WebDrivers, allow easy browser automation, thus making it easy to write and execute E2E tests against real web browsers.

A good time to write E2E scenario tests is when the development is complete and ready to be deployed.

When it comes to unit testing, there are different schools of thought around when a test should be written. A *Test Driven Developer* writes tests before the functionality is implemented. Others write tests when the implementation is complete to confirm the behavior. Some write while developing the component itself. Choose a style that suits you, keeping in mind that the earlier you write your tests, the better.

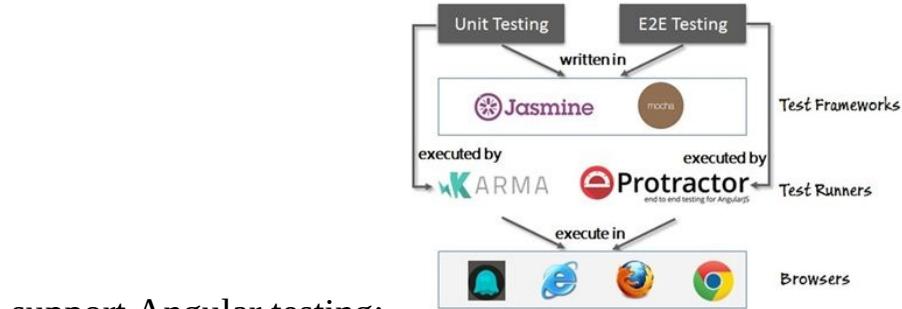


We are not going to give any recommendations, nor are we going to get into an argument over which one is better. Any amount of unit tests is better than nothing. Our personal preference is to use the middle approach. With TDD, we feel that the test creation effort at times is lost as the specifications/requirements change. Tests written at the start are prone to constant fixes as the requirement changes. The problem with writing unit tests at the end is that our target is to create tests that pass according to the current implementation. The tests that are written are retrofitted to test the implementation where they should test the specifications. Adding tests somewhere in the middle works best for us.

Let's now try to understand the tooling and technology landscape available for Angular testing.

The Angular testing ecosystem

Look at the following diagram to understand the tools and frameworks that



support Angular testing:

The tools and frameworks that support Angular testing

As we can see, we write our tests using unit testing libraries such as **Jasmine** or **Mocha**.



At the moment, the Angular testing library works by default with Jasmine. However, the Angular team has indicated that they have made the framework more generic so that you can use other testing libraries such as Mocha with it. The Angular documentation has not yet been updated to include how to do this. For a discussion of using Mocha with the Angular CLI testing commands see <https://github.com/angular/angular-cli/issues/4071>.

These tests are executed by either Karma or Protractor depending on whether we are writing unit or integration tests. These test runners in turn run our tests in a browser such as Chrome, Firefox, IE, or headless browsers such as PhantomJS. It is important to highlight that not only E2E, but also unit tests are executed in a real browser.



All the tests in this chapter are written using Jasmine (both unit and integration tests). Karma will be our test runner for unit tests and Protractor for E2E tests.

Getting started with unit testing

The ultimate aim of unit testing is to test a specific piece of code/component in isolation to make sure that the components work according to the specification. This reduces the chances of failures/bugs in the component when integrated with other parts of the software. Before we start writing tests, there are some guidelines that can help us write good and maintainable tests:

- One unit should test one behavior. For obvious reasons, testing one behavior per unit test makes sense. A failing unit test should clearly highlight the problem area. If multiple behaviors are tested together, a failed test requires more probing to assert what behavior was violated.
- Dependencies in a unit test should be mocked away using test doubles such as fakes, mocks, or st. Unit testing, as the name suggests, should test the unit and not its dependencies.
- Unit tests should not change the state of the component being tested permanently. If it does happen, other tests may get affected.
- The order of execution of unit tests should be immaterial. One unit test should not be dependent on another unit test to execute before it. This is a sign of a brittle unit test. It may also mean that the dependencies are not mocked.
- Unit tests should be fast. If they are not fast enough, developers will not run them. This is a good reason to mock all dependencies such as database access, remote web service call, and others in a unit test.
- Unit tests should try to cover all code paths. Code coverage is a metric that can help us assess the effectiveness of unit tests. If we have covered all positive and negative scenarios during testing, the coverage will indeed be higher. A word of caution here: high code coverage does not imply that the code is bug-free, but low coverage clearly highlights a lack of areas covered in unit tests.
- Unit tests should test both positive and negative scenarios. Just don't concentrate on positive test cases; all software can fail, and hence unit testing failure scenarios are as important to test as success scenarios.

These guidelines are not framework-specific, but give us enough ammunition for writing good tests. Let's begin the process of unit testing by setting up the components required for it.

Setting up Karma and Jasmine for unit testing

When we created our project using the Angular CLI, the CLI configured the setup for unit testing our code with Karma and Jasmine. It did so by adding several Karma and Jasmine modules to our project. It also added a Karma configuration file—`karma.config.js`—to the root directory of our application—`trainer/`—and a file called `tests.ts` in the `trainer/src` directory. The CLI makes use of these files at runtime to create the configuration for executing our tests. This means that we can run our tests by simply using the following command:

```
| ng test
```

And the CLI will also watch our tests for changes and automatically rerun them.



We will not be covering the configuration files in detail here. The out-of-the-box settings will be fine for our purposes. Refer to the Karma documentation (<http://karma-runner.github.io/1.0/config/configuration-file.html>) to understand more about the various Karma configuration options.

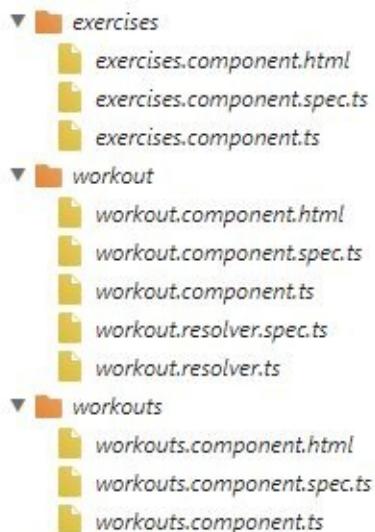
Organization and naming of our test files

To unit test our app, we should have one test (such as `workout-runner.spec.ts`) file for each TypeScript file that we plan to test in our project. And this is what the Angular CLI does for us. When we create a component, service, pipe, or directive using the CLI, the CLI will generate a corresponding test and place it in the same file directory.



Naming the test files with the name of the file under test plus `.spec` is a convention that is used by developers who test with Jasmine. It is also used to facilitate the mapping of files to tests in the configuration steps that we outlined previously.

This test file contains the unit test specification for the corresponding component, as shown in the following screenshot (taken in the Karma debugger when running our unit tests):



Unit-testing Angular applications

Over the course of this book, we have built components that cover every construct available in Angular. We have built components, pipes, a few services, and finally some directives too. All of these are testable in unit tests.



The code for the rest of this chapter can be found in `checkpoint 7.2`. It is available on GitHub (<https://github.com/chandermanni/angular6byexample>) for everyone to download. Checkpoints are implemented as branches in GitHub. If you are not using Git, download the snapshot of `checkpoint7.2` (a ZIP file) from this GitHub location: <https://github.com/chandermanni/angular2byexample/archive/checkpoint7.2.zip>. Refer to the `README.md` file in the `trainer` folder when setting up the snapshot for the first time.

Just to get the hang of unit testing with Jasmine, let's test the smallest and easiest component first: the pipe.

Unit-testing pipes

Pipes are the easiest to test as they have minimum or zero dependencies on other constructs. The `secondsToTimePipe` that we created for *Workout Runner* (the *7 Minute Workout* app) has no dependencies and can be easily unit-tested.

Look at the Jasmine framework documentation to understand how to write unit tests using Jasmine. The CLI is using Jasmine 2.6 for our unit tests (<http://jasmine.github.io/2.6/introduction.html>). Jasmine has some of the best documentations available and the overall framework is very intuitive to use. We strongly recommend that you head over to the [Jasmine site](#) and get yourself familiar with the framework before you proceed.

Open the `seconds-to-time.pipe.spec.ts` file in the `trainer/src/app/shared` folder and update the unit test in there as follows:

```
import { SecondsToTimePipe } from './seconds-to-time.pipe';
describe('SecondsToTimePipe', () => {
  const pipe = new SecondsToTimePipe();
  it('should convert integer to time format', () => {
    expect(pipe.transform(5)).toEqual('00:00:05');
    expect(pipe.transform(65)).toEqual('00:01:05');
    expect(pipe.transform(3610)).toEqual('01:00:10');
  });
});
```

Let's take a look at what we are doing here in our test file.

Not surprisingly, we import `SecondsToTimePipe`, which we are going to test. This is just like the imports we have used elsewhere in our TypeScript classes. Notice that we use a relative path to the file in which it is located `'./seconds-to-time.pipe'`. In Angular, this means to look for the component to test in the same directory as the test itself. As you recall, this is the way we set up our file structure: putting our tests in the same directory as the file under test.

In the next line, we start using Jasmine syntax. First, we wrap the test in a `describe` function that identifies the test. The first parameter of this function is a user-friendly description of the test; in this case, it is `SecondsToTimePipe`. For the second parameter, we pass a lambda (fat arrow) function that will contain our test. After setting up a local variable to hold the pipe, we call Jasmine's `beforeEach` function and use this to inject an instance of our pipe.



Since the `beforeEach` function runs before every test that is in our `describe` function, we can use it for common code that will run in each of our tests. In this case, it is not strictly necessary since there is only one test in our `describe` function. But it is a good idea to get into the habit of using it for common setup scenarios, as we will see going forward.

Next, we call Jasmine's `it` function and pass it a title, along with three calls to Jasmine's `expect` function (Jasmine's name for assertions). These are all self-explanatory.



It is not necessary to explicitly import these Jasmine functions in our test.

Running our test files

Now it's time to run our tests using the following command:

```
| ng test
```

The Angular CLI will transpile our TypeScript files to JavaScript and then watch for changes in these files.

We should then see this output in the Terminal window (the total number of tests may be different for you):

```
10% building modules 1/1 modules 0 active 10 05 2018 14:02:30.612:WARN [karma]: No captured browser, open http://localhost:9876/
10 05 2018 14:02:30.625:INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/
10 05 2018 14:02:30.625:INFO [launcher]: Launching browser Chrome with unlimited concurrency
10 05 2018 14:02:30.631:INFO [launcher]: Starting browser Chrome 10 05 2018 14:02:43.312:WARN [karma]: No captured browser, open http://localhost:9876/
10 05 2018 14:02:44.384:INFO [Chrome 66.0.3359 (Windows 10.0.0)]: Connected on socket A7GxjvNLPKdcZ-wLAAA with id 45967919
Chrome 66.0.3359 (Windows 10.0.0): Executed 31 of 48 SUCCESS (0 secs / 1.195 secs)
Chrome 66.0.3359 (Windows 10.0.0): Executed 44 of 48 SUCCESS (0 secs / 1.807 secs)
Chrome 66.0.3359 (Windows 10.0.0): Executed 48 of 48 SUCCESS (2.06 secs / 2.014 secs)
```

The last line shows that our test passed successfully (along with all our other tests).

You can also view the test results in the browser window that Karma will launch



when it runs our tests:

You'll notice here that Karma displays the `describe` statement

(SecondsToTimePipe) that is used for our pipe tests and nests under it under the `it` statement (should convert integer to time format) for the test we have created in order to show us the expected results for our test. Reading the results as they are displayed makes it very easy to understand the outcome of our test.

To make sure that it is reporting the correct pass/fail results, let's make a change in the test to cause one of the expectations to fail. Change the time in the first expectation to six seconds rather than five, like so: `expect(pipe.transform(5, [])).toEqual('00:00:06');`

We get the following error message:

```
Chrome 66.0.3359 (Windows 10.0.0): Executed 24 of 48 (1 FAILED) (0 secs / 1.376 secs)
Chrome 66.0.3359 (Windows 10.0.0) SecondsToTimePipe should convert integer to time format FAILED
  Expected '00:00:05' to equal '00:00:06'.
```

What's nice about this error message is that it combines the `describe` and `it` descriptions into a complete sentence that provides a clear summary of the error. This shows how Jasmine allows us to write readable tests so that someone who is new to our code can quickly understand any problems that may arise in it. The next line shows us which expectation was not met, what was expected, and what the actual results were that did not meet this expectation.

We also get a stack trace below this message and a final line that shows the overall results of our tests:

```
Chrome 66.0.3359 (Windows 10.0.0): Executed 24 of 48 (1 FAILED) (0 secs / 1.376 secs)
```

And in the browser, we see the following:



One thing you'll notice is that when we make the change to our test, we do not have to rerun Karma. Instead, it watches for any changes in our files and related tests and immediately reports success or failure whenever we make a change.

Pretty cool! Let's undo the last change that we made and put the test back into a passing state.

Unit-testing components

Testing Angular components is more complicated than testing simple pipes or services. That is because Angular components are associated with views and also usually have more dependencies than services, filters, or directives.

Angular testing utilities

Because of their complexity, Angular has introduced utilities that enable us to test our components more easily. These testing utilities include the `TestBed` class (which we previously used to initialize our tests) and several helper functions in `@angular/core/testing`.

`TestBed` has a `createComponent` method that returns a `ComponentFixture` containing several members and methods, including:

- `debugElement`: For debugging a component
- `componentInstance`: For accessing the component properties and methods
- `nativeElement`: For accessing the view's markup and other DOM elements
- `detectChanges`: For triggering the component's change detection cycle

`ComponentFixture` also contains methods for overriding the view, directives, bindings, and providers of a component. Going forward, we will be using `TestBed` throughout the rest of our tests.

`TestBed` has a method called `configureTestingModule` that we can use to set up our testing as its own module. This means we can bypass the initial bootstrap process and compile our components under test within our test files. We can also use `TestBed` to specify additional dependencies and identify the providers that we will need.



According to the Angular documentation (<https://angular.io/guide/testing#testbed-class-summary>), it is important to call `TestBed` methods within a `beforeEach` to ensure a fresh start before each individual test.

Managing dependencies in our tests

Components in Angular integrate the view with everything else. Due to this, components normally have more dependencies compared to any of the services, filters, or directives.

Notwithstanding the fact that our unit tests focus on the code within the component itself, we still need to account for these dependencies in our tests or else the tests will fail (we skipped the dependency setup for pipe testing as it did not have external dependencies).

Two approaches exist for handling these dependencies: inject them into our component or create a mock or fake for them that we can use in our tests. If a dependency is simple enough, we can just inject an instance of it into our test class. However, if the dependency is significantly complicated, especially if it has dependencies of its own and/or makes remote server calls, then we should be mocking it. The Angular testing library provides the tools for us to do that.

The component that we plan to test in this section is the `WorkoutRunner` component. Located inside `trainer/src/components/workout-runner/`, this is the component that runs a specific workout.

Unit-testing WorkoutRunnerComponent

With this background, let's get started with unit testing `WorkoutRunnerComponent`.

First, open `workout-runner-component.spec.ts` and update the imports to the following:

```
import { inject, fakeAsync, async, tick, TestBed, discardPeriodicTasks } from '@angular/core/testing';
import { NO_ERRORS_SCHEMA } from '@angular/core';
import { Router } from '@angular/router';
import { of } from 'rxjs/observable/of';

import { WorkoutPlan, ExercisePlan, Exercise } from './core/model';
import { WorkoutRunnerComponent } from './workout-runner.component';
import { SecondsToTimePipe } from './shared/seconds-to-time.pipe';
import { WorkoutService } from './core/workout.service';
import { WorkoutHistoryTrackerService } from './core/workout-history-tracker.service';
```

These imports identify the test utilities (and things such as `Router` and `of` from `RxJS`) that we will be using in our tests along with the types and dependencies that our component requires. We'll discuss these dependencies in a moment. One import that looks different from the others is the one that imports `NO_ERRORS_SCHEMA` from `@angular/core`. We will use this import to ignore elements in the component that we will not be testing. Again, we will discuss that further in a moment.

One more thing to note with the imports is that `@angular/core/testing` is a part of the core module and not in a separate testing module. This is a common pattern with imports for Angular testing. For example, when we get to testing HTTP, you will see that we are importing from `@angular/http/testing`.

Setting up component dependencies

Next, we need to establish our component's dependencies and determine whether we need to inject or mock them. If we look at the code for the `WorkoutRunner` component, we see that there are three dependencies being injected into our component:

- `WorkoutHistoryTracker`: This is a component that has some behavior attached to it. So we definitely want to mock it.
- `Router`: We'll have to mock this too in order to isolate `WorkoutRunner` from the rest of the application and prevent our test from trying to navigate away from the `WorkoutRunner` view.
- `WorkoutService`: This is a service that we will use to make an HTTP call to retrieve our workouts. We will mock this service as well since we don't want to be making a call to an external system within our test.

Mocking dependencies - workout history tracker

Angular allows us to mock our dependencies in a straightforward manner using simple classes. Let's start with mocking `WorkoutHistoryTracker`. To do that, add the following class just after the imports:

```
class MockWorkoutHistoryTracker {  
  startTracking() {}  
  endTracking() {}  
  exerciseComplete() {}  
}
```

We do not need to mock the entire `WorkoutHistoryTracker` class, but only the methods that `WorkoutRunner` will be calling. In this case, those methods are `startTracking()`, `endTracking()`, and `exerciseComplete()`. We have made these methods empty because we do not need anything returned from them in order to test `WorkoutRunner`. Now we can inject this dummy implementation into `WorkoutRunner` wherever it is looking for `WorkoutHistoryTracker`.

Mocking dependencies – workout service

In [Chapter 5](#), *Supporting Server Data Persistence*, we extended the workout service to make a remote call to retrieve the data that populates a workout. For unit testing the workout runner, we will want to replace that call with a mock implementation that returns some static data that we can use to run the test. So we will add a third mock class, as follows:

```
class MockWorkoutService {  
    sampleWorkout = new WorkoutPlan(  
        'testworkout',  
        'Test Workout',  
        40,  
        [  
            new ExercisePlan(new Exercise( 'exercise1', 'Exercise 1', 'Exercise 1  
description',  
                                         '/image1/path', 'audio1/path'), 50),  
            new ExercisePlan(new Exercise( 'exercise1', 'Exercise 2', 'Exercise 2  
description',  
                                         '/image2/path', 'audio2/path'), 30),  
            new ExercisePlan(new Exercise( 'exercise1', 'Exercise 3', 'Exercise 3  
description',  
                                         '/image3/path', 'audio3/path'), 20)  
        ],  
        'This is a test workout'  
    );  
  
    getWorkout(name: string) {  
        return of(this.sampleWorkout);  
    }  
    totalWorkoutDuration() {  
        return 180;  
    }  
}
```

Notice that the `getWorkout` method is returning an `Observable`, as indicated by the use of the `of` operator. Otherwise the class is self-explanatory.

Mocking dependencies - router

As with `WorkoutHistoryTracker` and `WorkoutService`, we also will be using mocking to handle the dependency that we have on the Angular router. But here we will be taking a slightly different approach. We will assign a Jasmine spy to `a navigate method` on our mock:

```
| export class MockRouter {  
|   navigate = jasmine.createSpy('navigate');  
| }
```

This will be sufficient for our purposes because we only want to make sure that the router's `navigate` method is being called with the appropriate route (`finished`) as a parameter. The Jasmine spy will allow us to do that as we will see later.

Configuring our test using TestBed

Now that we have our imports and dependencies out of the way, let's get started with the tests themselves. We begin by adding a Jasmine `describe` function that will wrap our tests, followed by setting two local variables using `let`: one for `fixture` and the other for `runner`:

```
describe('Workout Runner', () =>{
  let fixture:any;
  let runner:any;
```

Next we'll add a `beforeEach` function that sets up our test configuration:

```
beforeEach( async(() =>{
  TestBed
    .configureTestingModule({
      declarations: [ WorkoutRunnerComponent, SecondsToTimePipe ],
      providers: [
        {provide: Router, useClass: MockRouter},
        {provide: WorkoutHistoryTracker ,useClass:
          MockWorkoutHistoryTracker},
        {provide: WorkoutService ,useClass: MockWorkoutService}
      ],
      schemas: [ NO_ERRORS_SCHEMA ]
    })
    .compileComponents()
    .then(() => {
      fixture = TestBed.createComponent(WorkoutRunnerComponent);
      runner = fixture.componentInstance;
    });
}));
```

The `beforeEach` method executes before each test, which means that we will only have to set this up once in our test file. Inside `beforeEach`, we add an `async` call. This is required because of the asynchronous `compileComponents` method that we are calling.



The Angular documentation indicates that the `async` function `arranges` for the tester's code to run in a `special async test zone` that hides the mechanics of asynchronous execution, just as it does when passed to an `it` test. For more information refer to

<https://angular.io/docs/ts/latest/guide/testing.html#!#async-in-before-each>. We'll discuss this in more detail shortly.

Let's go through each method call in the order that they are executed. The first method, `configureTestingModule`, allows us to build on the base configuration of the testing module and add things such as imports, declarations (of the components,

directives, and pipes we will be using in our test), and providers. In the case of our test, we are first adding declarations for the workout runner, our component under test, and the `SecondsToTimePipe`:

```
| declarations: [ WorkoutRunnerComponent, SecondsToTimePipe ],
```

Then we add three providers for our `Router`, `WorkoutHistoryTracker`, and `WorkoutService`:

```
| providers: [
  {provide: Router, useClass: MockRouter},
  {provide: WorkoutHistoryTracker, useClass: MockWorkoutHistoryTracker},
  {provide: WorkoutService, useClass: MockWorkoutService}
],
```

For each of these providers, we set the `useClass` property to our mocks instead of the actual components. Now, anywhere in our test, when the `WorkoutRunner` requires any of these components, the mock will be used instead.

The next configuration may seem a bit mysterious:

```
| schemas: [ NO_ERRORS_SCHEMA ]
```

This setting allows us to bypass the errors we would otherwise get regarding the custom elements associated with two components that we are using in the component's template: `ExerciseDescriptionComponent` and `VideoPlayerComponent`. At this point, we don't want to be testing these components within the test for the `WorkoutRunnerComponent`. Instead, we should be testing them separately. One thing to be aware of, however, when you use this setting is that it will suppress all schema errors related to elements and attributes in the template of the component under test; so it may hide other errors that you do want to see.

When you set up a test using `NO_ERRORS_SCHEMA`, you are creating what is called a shallow test, one that does not go deeper than the component you are testing. Shallow tests allow you to reduce complexities in the templates within the component you are testing and reduce the need for mocking dependencies.

The final steps in the configuration of our test are to compile and instantiate our components:

```
| .compileComponents()
| .then(() => {
|   fixture = TestBed.createComponent(WorkoutRunnerComponent);
|   runner = fixture.componentInstance;
|});
```

As mentioned previously, we are using an `async` function in our `beforeEach` method because this is required when we call the `compileComponents` method. This method call is asynchronous and we need to use it here because our component has an external template that is specified in a `templateUrl`. This method compiles that external template and then inlines it so that it can be used by the `createComponent` method (which is synchronous) to create our component fixture. This component fixture in turn contains a `componentInstance=WorkoutRunner`. We then assign both the fixture and the `componentInstance` to local variables.

As mentioned previously, the `async` function we are using creates a special `async test zone` in which our tests will run. You'll notice that this function is simplified from normal `async` programming and lets us do things such as using the `.then` operator without returning a promise.



You can also compile and instantiate test components inside individual test methods. But the `beforeEach` method allows us to do it once for all our tests.

Now that we have configured our test, let's move on to unit-testing `WorkoutRunner`.

Starting unit testing

Starting from the loading of workout data to transitioning of exercises, pausing workouts, and running exercise videos, there are a number of aspects of the `WorkoutRunner` that we can test. The `workout.spec.ts` file (available in the `components/workout-runner` folder under `trainer/src`) contains a number of unit tests that cover the preceding scenarios. We will pick up some of those tests and work through them.

To start with, let's add a test case that verifies that the workout starts running once the component is loaded:

```
| it('should start the workout', () => {
|   expect(runner.workoutTimeRemaining).toEqual(runner.workoutPlan.totalWorkoutDuration());
|   expect(runner.workoutPaused).toBeFalsy();
|});
```

This test asserts that the total duration of the workout is correct and the workout is in the running state (that is, not paused).

So let's execute the test. It fails (check the Karma console). Strange! All the dependencies have been set up correctly, but still the second `expect` function of the `it` block fails as it is `undefined`.

We need to debug this test.

Debugging unit tests in Karma

Debugging unit tests in Karma is easy as the tests are run in the browser. We debug tests as we debug the standard JavaScript code. And since our Karma configuration has added mappings from our TypeScript files to our JavaScript files, we can debug directly in TypeScript.

When Karma starts, it opens a specific browser window to run the tests. To debug any test in Karma, we just need to click on the Debug button available at the top of the browser window.



There is one window opened by Karma and one when we click on Debug; we can use the original window for testing too, but the original window is connected to Karma and does a live reload. Also, the script files in the original window are timestamped, which changes whenever we update the test and hence requires us to put in a breakpoint again to test.

Once we click on Debug, a new tab/window opens with all the tests and other app scripts loaded for testing. These are scripts that were defined during the Karma configuration setup in the `karma.conf.js` files section.

To debug the preceding failure, we need to add breakpoints at two locations. One should be added inside the test itself and the second one inside `workoutComponent`, where it loads the workout and assigns the data to the appropriate local variables.

Perform the following steps to add a breakpoint in Google Chrome:

1. Open the Karma debug window/tab by clicking on the Debug button on the window loaded by Karma when it started.
2. Press the F12 key to open the developer console.
3. Go to the Sources tab and the TypeScript files for your application will be located in the `source` folder.
4. We can now put breakpoints at the required locations just by clicking on the line number. This is the standard mechanism to debug any script. Add breakpoints at the locations highlighted here:

```

workout-runner
  |- exercise-description
  |- video-player
  |- workout-audio
  |- workout-container
  |- workout-runner.component.html
  |- workout-runner.component.spec.ts
  |- workout-runner.component.ts
  |- app.component.html
  |- workout-container
  |- workout-runner.component.html
  |- workout-runner.component.spec.ts
  |- workout-runner.component.ts
  |- app.component.html
  |- app.component.spec.ts
  |- app.component.ts
  |- $lazy_route_resource_lazy
  |- polyfills.ts
  |- test.ts

```

```

64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277

```

- We refresh the Debug page (the one we opened when we clicked on the Debug button). The breakpoint in `workout-runner.ts` is never hit, causing the test to fail.

What we overlooked is that the code that we were trying to reach is within the `start` method of `workout-runner`, and the `start` method is not being called in the constructor. Instead it is called in `ngDoCheck` after the data for the workout has been loaded through a call to the `getWorkout` method in `ngOnInit`. Add calls to `ngOnInit` and `ngDoCheck` in your test, like so:

```

it('should start the workout', () => {
  runner.ngOnInit();
  runner.ngDoCheck();
  expect(runner.workoutTimeRemaining).toEqual(
    runner.workoutPlan.totalWorkoutDuration());
  expect(runner.workoutPaused).toBeFalsy();
});

```

- Save the change and Karma will run the test again. This time it will pass.



As the number of tests grows, unit testing may require us to concentrate on a specific test or a specific suite of tests. Karma allows us to target one or more tests by prepending `f` to the existing `it` block; that is, `it` becomes `f`. If Karma finds tests with `f`, it only executes those tests. Similarly, a specific test suite can be targeted by prepending `f` to the existing `describe` block: `fdescribe`. Also, if you prepend `x` to an `it` block, making it `xit`, then that block will be skipped.

Let's continue unit-testing the component!

Unit-testing WorkoutRunner continued...

What other interesting things can we test? We can test whether the first exercise has started. We add this test to `workout.spec.ts` after the one we just added:

```
it('should start the first exercise', () => {
```

```
    spyOn(runner, 'startExercise').and.callThrough(); runner.ngOnInit();
```

```
    runner.ngDoCheck();
```

```
    expect(runner.currentExerciseIndex).toEqual(0);
```

```
    expect(runner.startExercise).toHaveBeenCalledWith(
```

```
        runner.workoutPlan.exercises[runner.currentExerciseIndex]);
```

```
    expect(runner.currentExercise).toEqual(
```

```
        runner.workoutPlan.exercises[0]);
```

```
});
```

The second `expect` function in this test is interesting. It uses a Jasmine feature: Spies. Spies can be used to verify method invocations and dependencies.

Using Jasmine spies to verify method invocations

A spy is an object that intercepts every call to the function it is spying on. Once the call is intercepted, it can either return fixed data or pass the call to the actual function being invoked. It also records the call invocation details that can be used later in `expect` as we did in the preceding test.



Spies are very powerful and can be used in a number of ways during unit testing. Look at the documentation on spies at <http://jasmine.github.io/2.0/introduction.html#section-Spies> to learn more about them.

The second `expect` function verifies that the `startExercise` method was called when the workout started (`toHaveBeenCalledWith`). It is also asserting the correctness of the parameters passed to the function. The second `expect` statement asserts the behavior using a spy, but we first need to set up the spy to make this assert work.

In this case, we are using the spy to mock a call to the `startExercise` method. We can use the spy to determine whether the method has been called and with what parameters, using Jasmine's `toHaveBeenCalledWith` function.



Look at the Jasmine documentation for the `toHaveBeenCalled` and `toHaveBeenCalledWith` functions to learn more about these assert functions.

Here, the method is being called with the current `Exercise` as a parameter. Since the previous `expect` confirms that this is the first exercise, this `expect` confirms that a call to start that first exercise was executed.

There are a couple of things to note here. First, you have to be careful to put the setup for `spyOn` prior to calling `ngOnInit`. Otherwise, the spy will not be *spying* when the `startExercise` method is called and the method invocation will not be captured.

Second, since the spy is a mock, we will normally not be able to verify anything within the `startExercise` method. This is because the method itself is being mocked. This means that we cannot actually verify that the `currentExercise`

property has been set, since that is being done inside the mocked method. However, Jasmine allows us to chain the spy with `and.callThrough`, which will mean that in addition to tracking the calls to the method, it will delegate to the actual implementation. This then allows us to test that the `currentExercise` has also been set correctly inside the `startExercise` method.

Using Jasmine spies to verify dependencies

While we just used a spy to verify the call to a method within our class, Jasmine spies are also useful in mocking calls to external dependencies. But why test calls to our external dependencies at all? After all, we are trying to limit our testing to the component itself!

The answer is that we mock a dependency to make sure that the dependency does not adversely affect the component under test. From a unit testing perspective, we still need to make sure that these dependencies are called by the component being tested at the right time with the correct input. In the Jasmine world, spies help us assert whether dependencies were invoked correctly.

If we look at the `WorkoutRunner` implementation, we emit a message with the details of the workout whenever the workout starts. An external dependency, `WorkoutHistoryTracker`, subscribes to this message/event. So let's create a spy and confirm that `WorkoutHistoryTracker` started when the workout started.

Add this `it` block after the preceding one:

```
| it("should start history tracking", inject([WorkoutHistoryTracker], (tracker:  
|   WorkoutHistoryTracker) => {  
|     spyOn(tracker, 'startTracking');  
|     runner.ngOnInit();  
|     runner.ngDoCheck();  
|     expect(tracker.startTracking).toHaveBeenCalled();  
|   }));
```

Within the `it` block, we add a spy on the `tracker`, a local instance of the `WorkoutHistoryTracker`. Then we use the spy to verify that the `startTracking` method of that dependency has been called. Simple and expressive!

You may recall that we are using `MockHistoryWorkoutTracker` here; it contains a mock, a `startTracking` method that is empty and returns nothing. That is fine because we are not testing the `WorkoutHistoryTracker` itself, but just the method invocation on it being made by the `WorkoutRunner`. This test shows how useful it is to be able to combine mocks with spies to fully test the inner workings of the `WorkoutRunner`,

separately and apart from its dependencies.

Testing event emitters

Examining the code for the `WorkoutRunner`, we see that it sets up several event emitters that look like the following one for `workoutStarted`:

```
| @Output() workoutStarted: EventEmitter<WorkoutPlan> = new EventEmitter<WorkoutPlan>();
```

The Angular documentation describes an event emitter as an output property that fires events to which we can subscribe with an event binding. In [Chapter 2](#), *Building Our First App - 7 Minute Workout*, we described in detail how event emitters are used in Workout Runner. So we have a good understanding of what they do. But how do we unit-test our event emitters and determine that they are firing events in the way we expect?

It's actually pretty easy to do. If we remember that an **event emitter is an Observable Subject** to which we can subscribe, we realize that we can simply subscribe to it in our unit test. Let's revisit our test that verifies that a workout is starting and add the highlighted code to it:

```
it('should start the workout', () => {
  runner.workoutStarted.subscribe((w: any) => {
    expect(w).toEqual(runner.workoutPlan);      });
  runner.ngOnInit();
  runner.ngDoCheck();
  expect(runner.workoutTimeRemaining).toEqual(
    runner.workoutPlan.totalWorkoutDuration());
  expect(runner.workoutPaused).toBeFalsy();
});
```

We injected the `workoutService` and added a subscription to the `workoutStarted` event emitter and an expectation that checks to see whether the property is emitting a `workoutPlan` when the event is triggered. The subscription is placed before `ngOnInit` because that is the method that results in the `workoutStarted` event being triggered, and we need to have our subscription in place before that happens.

Testing interval and timeout implementations

One of the interesting challenges for us is to verify that the workout progresses as time elapses. The `Workout` component uses `setInterval` to move things forward with time. How can we simulate time without actually waiting?

The answer is the Angular testing library's `fakeAsync` function, which allows us to run otherwise asynchronous code synchronously. It does this by wrapping the function to be executed in a `fakeAsync` zone. It then supports using synchronous timers within that zone and also allows us to simulate the asynchronous passage of time with `tick()`.



For more information about `fakeAsync`, see the Angular documentation at <https://angular.io/guide/testing#async-test-with-fakeasync>.

Let's see how we can use the `fakeAsync` function to test the timeout and interval implementations in our code. Add the following test to `workout-runner.spec.ts`:

```
it('should increase current exercise duration with time', fakeAsync(() => {
  runner.ngOnInit();
  runner.ngDoCheck();
  expect(runner.exerciseRunningDuration).toBe(0);
  tick(1000);
  expect(runner.exerciseRunningDuration).toBe(1);
  tick(1000);
  expect(runner.exerciseRunningDuration).toBe(2);
  tick(8000);
  expect(runner.exerciseRunningDuration).toBe(10);
  discardPeriodicTasks();
}));
```

In addition to injecting `WorkoutRunner`, we first wrap the test in `fakeAsync`. Then we add a call to the `WorkoutRunner`'s `ngOnInit()` method. This kicks off the timers for the exercises within `WorkoutRunner`. Then within the test, we use the `tick()` function set at various durations to test the operation of the timer for an exercise, and make sure that it continues running for the duration that we expected it to run. Using `tick()` allows us to fast forward through the code and avoid having to wait for the exercise to complete over several seconds as it would if we ran the code asynchronously.

At the end, we call `discardPeriodicTasks()`. This method is one of the Angular testing utilities and it can be used with `fakeAsync` to clear any pending timers that may be in the task queue.



More information about these and other Angular testing utilities can be found at <https://angular.io/guide/testing#testing-utility-apis>.

Let's try another similar test. We want to make sure that the `workoutRunner` is correctly transitioning from one exercise to the next. Add the following test to `workout-runner.ts`:

```
| it("should transition to next exercise on one exercise complete", fakeAsync(() => {
|   runner.ngOnInit();
|   runner.ngDoCheck();
|   let exerciseDuration = runner.workoutPlan.exercises[0].duration;
|   TestHelper.advanceWorkout(exerciseDuration);
|   expect(runner.currentExercise.exercise.name).toBe('rest');
|   expect(runner.currentExercise.duration).toBe(
|     runner.workoutPlan.restBetweenExercise);
|   discardPeriodicTasks();
| }));
|});
```

Again we wrap the test in `fakeAsync` and call `runner.ngOnInit` to start the timer. Then we grab the duration of the first exercise and use the `tick()` function within the following `TestHelper` method to advance the timer one second beyond the duration of that exercise:

```
| class TestHelper {
|   static advanceWorkout(duration: number) {
|     for (let i = 0; i <= duration; i++) {tick(1000);
|   }
| };
```

Next, we test the expectation that we are now in the `rest` exercise and thus have transitioned from the first exercise.

Testing workout pause and resume

When we pause a workout, it should stop and the time counter should not lapse. To check this, add the following time test: it("should not update workoutTimeRemaining for paused workout on interval lapse", fakeAsync(() => {

```
runner.ngOnInit();
runner.ngDoCheck();
expect(runner.workoutPaused).toBeFalsy(); tick(1000);
expect(runner.workoutTimeRemaining).toBe(
  runner.workoutPlan.totalWorkoutDuration() - 1); runner.pause();
expect(runner.workoutPaused).toBe(true); tick(1000);
expect(runner.workoutTimeRemaining).toBe(
  runner.workoutPlan.totalWorkoutDuration() - 1); discardPeriodicTasks(); }));
```

The test starts with verifying the state of the workout as not paused, advances the time for one second, pauses it, and then verifies that the time of `workoutTimeRemaining` does not change after the pause.

Unit-testing services

Unit testing of services is not much different from unit testing components. Once we get the hang of how to set up a component and its dependencies (mostly using mocks), it becomes a routine affair to apply that learning to testing services. More often than not, the challenge is to set up the dependencies for the services so that testing can be done effectively.

Things are a little different for services that make remote requests (using either `http` or `jsonp`). There is some setup required before we can test such services in isolation.

We will target `workoutService` and write some unit tests for it. Since this service makes remote requests to load workout data, we will explore how to test such a service with a mock HTTP backend. Angular provides us with the `HttpTestingController` for doing that.

Mocking HTTP request/response with HttpTestingController

When testing services (or, as a matter of fact, any other Angular construct) that make remote requests, we obviously do not want to make actual requests to a backend to check the behavior. That does not even qualify for a unit test. The backend interaction just needs to be mocked away. Angular provides for precisely that. Using `HttpTestingController`, we intercept HTTP requests, mock actual responses from the server, and assert endpoints invocation too.

Open `workout-service.spec.ts` and add the following import statements at the top of the file:

```
import { TestBed, inject, async, fakeAsync } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController } from
  '@angular/common/http/testing';
import { HttpClient, HttpResponse } from '@angular/common/http';

import { WorkoutService } from './workout.service';
import { WorkoutPlan, Exercise } from './model';
```

In addition to the imports from the `core/testing` module, we are importing both `HttpClientTestingModule` and `HttpTestingController` from the `http/testing` module. We are also importing `WorkoutService` and `WorkoutPlan` that we will be testing.

Once we have the imports in place, we will begin creating the test with the Jasmine `describe` statement that wraps our tests, and will set several local variables:

```
describe('Workout Service', () => {
  const collectionUrl = '...[mongo connection url]...';
  const apiKey = '...[mongo key]...';
  const params = '?apiKey=' + apiKey;
  let httpClient: HttpClient;
  let httpTestingController: HttpTestingController;
  let workoutService: WorkoutService;
```

In addition to creating local variables for `HttpClient`, `HttpTestingController`, and `workoutService`, you'll also notice that we are setting local variables for our Mongo connection. To be clear, we are not setting these variables in order to make a remote call to Mongo, but instead to test that the connection properties are being

set properly.

The next step is set up the providers and dependency injection for our tests. To handle the providers, add the following to the test file:

```
beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
    providers: [WorkoutService],
  });
  httpClient = TestBed.get(HttpClient);
  httpTestingController = TestBed.get(HttpTestingController);
  workoutService = TestBed.get(WorkoutService);
});
```

First, we call `TestBed.configureTestingModule` to import the `HttpClientTestingModule` and add the `WorkoutService`. According to the Angular documentation (<https://angular.io/api/common/http/testing/HttpClientTestingModule>), `HttpClientTestingModule` sets up `HttpClient` to use `HttpClientTestingBackend` as `HttpBackend`. The nice thing here is that this setup is completely hidden from our test setup, so we don't have to write code to wire this up.

Next we use the `TestBed.get` method to populate our local variables—`httpClient`, `httpTestingController`, and `workoutService`. We'll also add the following `afterEach` method to make sure that after every test completes there are no more pending requests:

```
afterEach(() => {
  httpTestingController.verify();
});
```

With this setup in place, we are now in a position to create tests for `workoutService` that avoid us making a remote call. We'll start with a simple test that makes sure that `workoutService` loads:

```
it('should be created', inject([WorkoutService], (service: WorkoutService) => {
  expect(service).toBeTruthy();
}));
```

While this test may seem trivial, it is important to place it here because it acts as a check to make sure that we have set up our configuration correctly.

Next, we'll add the following test to make sure that we are able to inject `HttpClient` when we instantiate `WorkoutService`:

```
| it('can instantiate service with "new"', inject([HttpClient], (http: HttpClient) => {
|   expect(http).not.toBeNull('http should be provided');
|   const service = new WorkoutService(http);
|   expect(service instanceof WorkoutService).toBe(true, 'new service should be ok');
| }));
|});
```

Now, we'll move to testing several of the methods in the workout-service. First, we will make sure that it returns all workouts when the `getWorkouts` method is called. To do that, add the following test:

```
it('should return all workout plans', () => {
  let expectedWorkouts: WorkoutPlan[];
  let actualWorkouts: WorkoutPlan[];

  expectedWorkouts = [
    { name: 'Workout1', title: 'workout1' },
    { name: 'Workout2', title: 'workout2' },
    { name: 'Workout3', title: 'workout3' },
    { name: 'Workout4', title: 'workout4' }
  ] as WorkoutPlan[];

  workoutService.getWorkouts().subscribe(
    workouts => actualWorkouts = workouts,
    fail
  );
  const req = httpTestingController.expectOne(workoutService.collectionsUrl +
  '/workouts' + params);
  expect(req.request.method).toEqual('GET');
  req.flush(expectedWorkouts);
  expect(actualWorkouts === expectedWorkouts);
});
```

We'll start by declaring two arrays of `WorkoutPlans`—`expectedWorkouts` and `actualWorkouts`. We'll then populate `expectedWorkouts` with four `WorkoutPlans`. Because we are testing retrieval of the `WorkoutPlans` and not their content, we have made these minimal workouts.

Since the `Http` module returns RxJS Observables, we next use the pattern of subscribing to those Observables. You should be used to seeing this pattern from our coverage of Observables in [Chapter 5, Supporting Server Data Persistence](#). Notice that we use `fail` as the second parameter, which will cause the test to fail if there is an issue with subscribing to the Observable.

Next we call a method on the `HttpTestingController` named `expectOne` passing our request URL. According to the Angular documentation (<https://angular.io/api/common/http/testing/HttpTestingController#expectone>), this method does two things: it expects that a request has been made that matches the URL provided in the method call and it returns a mock request. In the next line we make sure that the mock request is an HTTP GET. Finally, we flush the request with the

`expectedWorkouts` and confirm that the `actualWorkouts` equal the `expectedWorkouts`.

We'll follow the same pattern to build additional tests that confirm that we are able to do the following:

- Return a `workout` plan with a specific name
- Map `exercises` correctly within the `getWorkout` method

You can review these tests in the code for [checkpoint 7.2](#). But one thing to note is that in both these tests we are testing two HTTP calls. For example, here is the code in the second of these two tests:

```
const req1 = httpTestingController.expectOne(workoutService.collectionsUrl +  
  '/exercises' + params);  
expect(req1.request.method).toEqual('GET');  
req1.flush(allExercises);  
  
const req2 = httpTestingController.expectOne(workoutService.collectionsUrl +  
  '/workouts/Workout1' + params);  
expect(req2.request.method).toEqual('GET');  
req2.flush(expectedWorkout);
```

This may seem a little confusing at first until we realize that with the `getWorkout` method we are actually making two `HTTP` calls: one to retrieve a `workout` and one to retrieve all `exercises`. As you recall from [Chapter 5, Supporting Server Data Persistence](#), we are doing that in order to create a fuller description of each exercise that is included within `workout`.

With that, we are finished with testing our service.

Next, we need to learn how to test directives. The next section is dedicated to understanding the challenges in directive testing and how to overcome them.

Unit-testing directives

No other Angular constructs that we have tested so far do not involve any UI interaction. But directives, as we know, are a different beast. ~~Directives are all about enhancing a component's view and extending the behavior of HTML elements. While testing directives, we cannot ignore the UI connections, and hence directive testing may not strictly qualify as unit testing.~~

The good thing about directive testing is that its setup process is not as elaborate as that for services or components. The pattern to follow while unit-testing directives is as follows:

1. Take an HTML fragment containing the directive markup
2. Compile and link it to a mock component
3. Verify that the generated HTML has the required attributes
4. Verify if the changes that the directive created changes the state

The TestBed class

As mentioned previously, Angular provides the `TestBed` class to facilitate this kind of UI testing. We can use it to dig into the markup in a component's view and check for DOM changes that are triggered by events. Armed with this tool, let's get started with the testing of our directives. In this section, we are going to test `remoteValidator`.



This will be a good time to revisit the directives that we built in the previous chapter. Also, keep the code handy for the tests that we will create in the following sections.

Testing remote validator

Let's start with unit-testing `remotevalidatorDirective`. Just to refresh our memory, `remotevalidatorDirective` validates an input against remote rules. It does so by calling a component method that returns a promise. If the promise is resolved with success, the validation passes; otherwise, the validation fails. The `[validateFunction]` attribute provides the link between the DOM and the component's method that checks for the duplication.

Similar to our other test files, we have a `remote-validator.directive.spec.ts` file in the shared folder. Refer to the file in [checkpoint 7.2](#) for the imports, which we will not cover at this point.

Just below the import statements, add the following component definition:

```
@Component({
  template: `
    <form>
      <input type="text" name="workoutName"
        id="workout-name" [(ngModel)]="workoutName"
        abeRemoteValidator="workoutname" [validateFunction]="validateWorkoutName">
    </form>
  `})
export class TestComponent {
  workoutName: string;

  constructor() {
    this.workoutName = '7MinWorkout';
  }
  validateWorkoutName = (name: string): Promise<boolean> => {
    return Promise.resolve(false);
}
}
```

This component looks a lot like the components that we set up in our other tests to mock dependencies. Here, however, it is serving a slightly different purpose; it is acting as a host container for the directive that we will be testing. Using this minimal component, lets us avoid having to load the actual host for this directive, which is the `workout` component.

One thing to notice here is that we have set up a method for `validateWorkoutName` that will be called by our directive. It is essentially a stub that just returns a resolved `Promise` of `false`. Remember that we are not concerned with how this

method handles its validation, but with verifying that the directive calls it and returns the correct result, either `true` or `false`.

Next, we set up the `describe` statement for our test suite by adding the following code, which injects `RemoteValidatorDirective` into our tests:

```
describe('Remotevalidator', () => {
  let fixture: any;
  let comp: any;
  let debug: any;
  let input: any;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [ FormsModule ],
      declarations: [ TestComponent, RemoteValidatorDirective ]
    });
    fixture = TestBed.createComponent(TestComponent);
    comp = fixture.componentInstance;
    debug = fixture.debugElement;
    input = debug.query(By.css('[name=workoutName]'));
  }));
});
```

As you can see, we are setting up local variables for `fixture`, its `componentInstance`, and `debugElement`. We are also using `by.css` (which we will see more of in our end-to-end tests) along with the `query` method on `debugElement` to extract the `workoutName` input from our component. We'll be using these to delve into the rendered HTML in our directive.

Now we are ready to write our individual tests. First, we'll write a test to confirm that we have been able to load `RemoteValidatorDirective`. So add the following code:

```
it("should load the directive without error", fakeAsync(() => {
  expect(input.attributes.a2beRemoteValidator).toBe('workoutname', 'remote validator directive should be loaded.')
}));
```

What's interesting about this test is that using the `debugElement`, we have been able to drill-down into the attributes of the `input` tag in our host component and find our validator, confirming that it has indeed been loaded. Also notice the use of `fakeAsync`, which we discussed in connection with unit testing. Using it makes it possible for us to write our tests in a synchronous fashion and avoid the complications that would otherwise exist with trying to manage the asynchronous rendering of our host component. Next, we'll write two tests to confirm that our validator is working properly. The first test will make sure that an error is created if remote validation fails (that is, a workout with the same

name as the one we are using is found). Add the following code for that test:

```
it('should create error if remote validation fails', fakeAsync(() => {
  spyOn(comp, 'validateWorkoutName').and.callThrough();
  fixture.detectChanges();
  input.nativeElement.value = '6MinWorkout';
  tick();

  const form: NgForm = debug.children[0].injector.get(NgForm);
  const control = form.control.get('workoutName');

  expect(comp.validateWorkoutName).toHaveBeenCalled();
  expect(control.hasError('workoutname')).toBe(true);
  expect(control.valid).toBeFalsy();
  expect(form.valid).toEqual(false);
  expect(form.control.valid).toEqual(false);
  expect(form.control.hasError('workoutname', ['workoutName'])).toEqual(true);
});
```

Again, we are using `fakeAsync` to eliminate the challenges that we would otherwise have with the async behavior associated with the rendering and execution of our `remoteValidatorDirective`. Next, we add a spy to track the invocation of the `validateWorkoutName` method. We also set the spy to call through to our method, because in this case, we are expecting it to return false. The spy is being used to verify that our method has indeed been invoked. Next, we set `fixture.detectChanges`, which triggers a change detection cycle. We then set the value of our input and call `tick`, which will, we hope, trigger the response we are expecting from our remote validator. We then grab the form encapsulating our input tag using the injector that is available from the child element array of the `debug` element. From there, we extract the form control for our input box. Then we run several expectations that confirm that an error has been added both to our control and to the form and that both are now in an invalid state. The next test is the mirror opposite of this test and it checks for a positive:

```
it('should not create error if remote validation succeeds', fakeAsync(() => {
  spyOn(comp, 'validateWorkoutName').and.returnValue(Promise.resolve(true));
  fixture.detectChanges();
  input.nativeElement.value = '6MinWorkout';
  tick();

  const form: NgForm = debug.children[0].injector.get(NgForm);
  const control = form.control.get('workoutName');

  expect(comp.validateWorkoutName).toHaveBeenCalled();
  expect(control.hasError('workoutname')).toBeFalsy();
  expect(control.valid).toBeTruthy();
  expect(form.control.valid).toEqual(true);
  expect(form.valid).toEqual(true);
  expect(form.control.hasError('workoutname', ['workoutName'])).toEqual(false);
});
```

Other than changing the expectations, the only change we are making from the previous test is setting up our spy to return a value of `true`. Unit-testing our `remoteValidatorDirective` shows how powerful the `TestBed` utilities are in testing our UI and the elements and behaviors associated with it.

i believe, E2E testing not used as i know.. let me skip at all.

Getting started with E2E testing

Automated **E2E** testing is an invaluable asset if the underlying framework supports it. As the size of an app grows, automated E2E testing can save a lot of manual effort.

Without automation, it's just a never-ending battle to make sure that the app is functional. However, remember that in an E2E setup, not everything can be automated; automation may require a lot of effort. With due diligence, we can offload a sizable amount of manual effort, but not everything.

The process of E2E testing of a web-based application is about running the application in a real browser and asserting the behavior of the application based on the user interface state. This is how an actual user does testing.

Browser automation holds the key here, and modern browsers have become smarter and more capable in terms of supporting automation. Selenium tools for browser automation are the most popular option out there. Selenium has the WebDriver (<https://www.w3.org/TR/webdriver/>) API that allows us to control the browser through the automation API that modern browsers natively support.

The reason behind bringing up Selenium WebDriver is that the Angular E2E testing framework/runner **Protractor** also uses WebDriverJS, which is a JavaScript binding of WebDriver on Node. These language bindings (like the preceding JavaScript binding) allow us to use the automation API in the language of our choice.

Let's discuss Protractor before we start writing some integration tests for our app.

Introducing Protractor

Protractor is the de facto test runner for E2E testing in Angular. Protractor **uses** Selenium WebDriver to control a browser and simulate user actions.

A typical Protractor setup has the following components:

- A test runner (Protractor)
- A Selenium server
- A browser

We write our test in Jasmine and use some objects exposed by Protractors (which is a wrapper over WebDriverJS) to control the browser.

When these tests run, Protractor sends commands to the Selenium server. This interaction happens mostly over HTTP.

The Selenium server, in turn, communicates with the browser using the WebDriver Wire Protocol, and internally the browser interprets the action commands using the browser driver (such as ChromeDriver in the case of Chrome).

~~It is not that important to understand the technicalities of this communication, but we should be aware of the E2E testing setup. Check out the article from the Protractor documentation at <http://angular.github.io/protractor/#/infrastructure> to learn more about this flow.~~

Another important thing to realize when using Protractor is that the overall interaction with the browser or the browser control flow is asynchronous in nature and promise-based. Any HTML element action, whether `sendKeys`, `getText`, `click`, `submit`, or any other, does not execute at the time of invocation; instead the action is queued up in a control flow queue. For this precise reason, the return value of every action statement is a promise that gets resolved when the action completes.

To handle this asynchronicity in Jasmine tests, Protractor patches Jasmine, and

therefore assertions like these work:

```
| expect(element(by.id("start")).getText()).toBe("Select Workout");
```

They work despite the `getText` function returning a promise and not the element content.

With this basic understanding of how Protractor works, let's set up Protractor for end-to-end testing.

Setting up Protractor for E2E testing

The Angular CLI has already set up our project to allow us to use Protractor. The configurations for that setup can be found in the `protractor.config.js` file in the `trainer` folder. For the most part, you should be able to use those configurations without change to run your end-to-end tests. We did, however, make one change in that configuration file. We did extend the `defaultTimeoutInterval` in that file to 60000 milliseconds in order to give our tests that are running the workouts more time to finish: `const { SpecReporter } = require('jasmine-spec-reporter');`

```
exports.config = {
  ...
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 60000,
    print: function() {}
  },
  ...
};
```



The configuration file documentation on the Protractor website (<https://github.com/angular/protractor/blob/master/lib/config.ts>) contains details on other supported configurations.

That is enough to start testing with Protractor. In order to run our tests we simply execute the following command in the `trainer` folder: `ng e2e`

Now, let's get started with writing some end-to-end tests.

Writing E2E tests for the app

Let's start in a simple manner and test our app start page (`#/start`). This page has some static content, a workout listing section with search capabilities, and the ability to start a workout by clicking on any workout file.



All our E2E tests will be added to the `e2e` folder under `trainer`.

Open the file called `app.e2e-spec.ts` to the `e2e` folder under `trainer`, which contains the following code:

```
import { AppPage } from './app.po';

describe('Personal Trainer App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  it('should display welcome message', () => {
    page.navigateTo();
    expect(page.getParagraphText()).toEqual('Ready for a Workout?');
  });
})
```

Let's walk through this simple test.

The first interesting piece is the import at the top of the page—`import { AppPage } from './app.po'`. This is referring to a file in the same directory that contains what is called a page object. This page object contains the following:

```
import { browser, by, element } from 'protractor';

export class AppPage {
  navigateTo() {
    return browser.get('/');
  }

  getParagraphText() {
    return element(by.css('abe-root h1')).getText();
  }
}
```

The use of the page object allows us to simplify the code in our test so that it is more readable. So instead of directly calling `browser.get ('/')` in our test, we call

the `navigateTo()` method from our page object.



The browser object referred to in our page object is a global object exposed by Protractor and it is used to control the browser-level actions. Underneath, it is just a wrapper around WebDriver. The `browser.get("")` method navigates the browser to start the app page, every time, before the start of the test.

The same for the `getParagraphText()` method—it allows us to call that method in our test and look for some text on the screen without having to identify the exact location on the page where that will appear. We'll discuss page objects in more detail as we get into more complicated end-to-end testing. The `getParagraphText()` in our page object also employs two new globals, `element` and `by`, which are made available by Protractor:

- `element`: This function returns an `ElementFinder` object. The primary job of `ElementFinder` is to interact with the selected element. We will be using the `element` function to select `ElementFinder` extensively in our tests.



Refer to the documentation at <http://www.protractortest.org/#/locators#actions> to learn more about element manipulation API support. Functions such as `getText()` are actually defined on `WebElement`, but are always accessed using `ElementFinder`. As the documentation suggests, `ElementFinder` can be treated as `webElement` for most purposes. For more information, you can refer to <http://www.protractortest.org/#/locators#behind-the-scenes-elementfinders-versus-webelements>.

- `by`: This object is there to locate elements. It has functions that create locators. In the preceding test, a locator is created to search for elements with a css tag equal to `abe-root h1`. If you are familiar with CSS selectors, you will know that this identifies what we are looking for as an `h1` tag inside the custom element `abe-root`. There are a number of locators that can be used to search for a specific element. These include by class, by ID, and by css. Refer to the Protractor documentation on locators at <http://angular.github.io/protractor/#/locators> to learn about the supported locators.



Just to reiterate what we discussed earlier, `getText()` in the page object does not return the actual text, but a Promise; we can still assert on the text value.

Getting back to the actual test, it uses the methods in the page object to verify that some content ("Ready for a Workout?") is present on the page.

This simple test highlights another salient feature of Protractor. It automatically detects when the Angular app is loaded and when data is available for testing.

There are no ugly hacks to delay testing (using `timeouts`) that may otherwise be required in standard E2E testing scenarios.

Remember, this is an SPA; full-page browser refresh does not happen, so it is not that simple to determine when the page is loaded and when the data that is rendered for AJAX calls is available. Protractor makes it all possible.



Protractor may still time out while trying to assess whether the page is available for testing. If you are hitting timeout errors with Protractor, this article from the Protractor documentation can be really helpful (<http://www.protractortest.org/#/timeouts>) for debugging such issues.

Setting up backend data for E2E testing

Setting up backend data for E2E testing is a challenge, irrespective of the E2E framework that we employ for testing. The ultimate aim is to assert the behavior of an application against some data, and unless the data is fixed, we cannot verify the behavior that involves getting or setting data.

One approach to setting up data for E2E tests is to create a test data store specifically for E2E tests with some seed data. Once the E2E tests are over, the data store can be reset to its original state for future testing. For *Personal Trainer*, we can create a new database in MongoLab dedicated exclusively to E2E testing.

This may seem a lot of effort, but it is necessary. Who said E2E testing is easy! In fact, this challenge is there even if we do manual testing. For a real app, we always have to set up data stores/databases for every environment, whether *dev*, *test*, or *production*.

In this case, we will continue to use our existing backend, but go ahead and add another workout that we will use for testing. Name this workout `1minworkout` and give it a title of `1 Minute Workout`. Add two exercises to the workout: Jumping Jacks and Wall Sit. Set the duration of each exercise to 15 seconds and the rest time to one second.



We have deliberately kept our new workout short so that we can complete our end-to-end testing of this workout within the normal timeouts provided by Protractor.

More E2E tests

Let's get back to testing the workout search features on the start page. With the addition of *1 Minute Workout*, we now have two workouts and we can assert search behaviors against these.



If you have added other workouts to the backend, just adjust the numbers in this test accordingly.

Add this test after the existing test in `workout-runner.spec.ts`:

```
it('should search workout with specific name.', () => {
  const filteredWorkouts = element.all(by.css('.workout.tile'));
  expect(filteredWorkouts.count()).toEqual(5);

  const searchInput = element(by.css('.form-control'));
  searchInput.sendKeys('1 Minute Workout');

  expect(filteredWorkouts.count()).toEqual(1);
  expect(filteredWorkouts.first().element(by.css('.title')).getText()).toBe('1 Minute
Workout');
});
```

The test uses `ElementFinder` and `Locator API` to look for elements on the page. Check the second line of the test. We are using the `element.all` function together with the `by.css` locator to do a multi-element match on all elements on the screen that are using the `.workout.tile` CSS class. This gives us a list of workouts against which the next line asserts the element count of 3.

The test then gets hold of the search input using the `element` function along with the `by.css` locator to do a single element match for an element using the `.form-control` CSS class. We then use the `sendKeys` function to simulate data entry in the search input.

The last two expect operations check for the count of elements in our list, which after the search should be 1. Also, they check whether the correct workout is filtered based on a `div` tag using the `title` CSS class that is a child of the element that contains our workout. This last expect statement highlights how we can chain element filtering and get hold of child elements in HTML.

There is one more test associated with the start page that we should add. It tests

```
the navigation from the start page to the workout runner screen. Add this code  
for that test: it('should navigate to workout runner.', () => {  
  const filteredWorkouts = element.all(by.css('.workout.tile'));  
  filteredWorkouts.first().click();  
  expect(browser.getCurrentUrl()).toContain('/workout/1minworkout');  
})
```

This test uses the `click` function to simulate clicking on a workout tile, and then we use the `browser.getCurrentUrl` function to confirm that the navigation is correct.

Run the test again (`protractor tests/protractor.conf.js`) and once again observe the magic of browser automation as the tests run one after another.

Can we automate E2E testing for *Workout Runner*? Well, we can try.

Testing Workout Runner

One of the major challenges with testing Workout Runner is that everything is time-dependent. With unit testing, at least we were able to mock the interval, but not anymore. Testing exercise transitions and workout completion is definitely difficult.

However, before we tackle this problem or try to find an acceptable workaround, let's digress and learn about an important technique to manage E2E testing: page objects!

Using page objects to manage E2E testing

We touched on page objects earlier. The concept of page objects is simple. We encapsulate the representation of page elements into an object so that we do not have to litter our E2E test code with `ElementFinder` and `locators`. If any page element moves, we just need to fix the page object.

Here is how we can represent our Workout Runner page:

```
import { browser, by, element } from 'protractor';

export class WorkoutRunnerPage {
  pauseResume: any;
  playButton: any;
  pauseButton: any;
  exerciseTitle: any;
  exerciseDescription: any;
  exerciseTimeRemaining: any;

  constructor() {
    this.pauseResume = element(by.id('pause-overlay'));
    this.playButton = element.all(by.css('.ion-md-play'));
    this.pauseButton = element.all(by.css('.ion-md-pause'));
    this.exerciseTitle = element(by.id('exercise-pane')).element(by.tagName('h1')).getText();
    this.exerciseDescription = element.all(by.className('card-text')).first().getText();
    this.exerciseTimeRemaining = element(by.id('exercise-pane')).all(by.tagName('h4')).first().getText();
  }
}
```

This page object now encapsulates many of the elements that we want to test. By organizing the element selection code in one place, we increase the readability and hence the maintainability of E2E tests.

Now add the Workout Runner page object to the top of the test file. We'll use it in a test for the workout runner. Add the following new describe block containing the first of our workout runner tests:

```
describe('Workout Runner page', () => {
  beforeEach(() => {
    browser.get('/workout/1minworkout');
  });
}
```

```

it('should load workout data', () => {
  browser.waitForAngularEnabled(false);
  const page = new WorkoutRunnerPage();
  page.pauseResume.click();
  expect(page.exerciseTitle).toBe('Jumping Jacks');
  expect(page.exerciseDescription)
    .toBe('A jumping jack or star jump, also called side-straddle hop is a
physical jumping exercise.');
});

```

The test verifies that the workout is loaded and the correct data is shown. We make full use of the page object that we defined earlier. Run the test and verify that it passes.

Let's get back to the challenge of testing code based on `interval` or `timeout`. Let's add a test that confirms a click event on the screen, when the pause button is pushed:

```

it('should pause workout when paused button clicked', () => {
  const page = new WorkoutRunnerPage();
  let timeRemaining;
  browser.waitForAngularEnabled(false);

  page.pauseResume.click();
  expect(page.playButton.count()).toBe(1);
  expect(page.pauseButton.count()).toBe(0);

  page.exerciseTimeRemaining.then((time) => {
    timeRemaining = time;
    browser.sleep(3000);
  });
  page.exerciseTimeRemaining.then((time) => {
    expect(page.exerciseTimeRemaining).toBe(timeRemaining);
  });
});

```

What is interesting here is that we use the `browser.sleep` function within a promise to verify that the exercise time remaining is the same before and after the button is clicked. We are again using our `WorkoutRunner` page object to make the test more readable and understandable.

It's now time to wrap up the chapter and summarize our learning.

Summary

We do not need to reiterate how important unit- and E2E-testing are for any application. The way the Angular framework has been designed makes testing the Angular app easy. In this chapter, we covered how to write unit tests and E2E tests using libraries and frameworks that target Angular.

For unit testing, we used Jasmine to write our tests and Karma to execute them. We tested pipes, components, services, and directives from *Personal Trainer*. In the process, we learned about the challenges and the techniques used to effectively test these types.

For E2E testing, the framework of choice was Protractor. We still wrote out tests in Jasmine, but the test runner this time was Protractor. We learned how Protractor automates E2E testing using Selenium WebDriver, as we did some scenario testing for the *Start* and *Workout Runner* pages.

If you have reached this point, you are getting closer to becoming a proficient Angular developer. The next chapter reinforces this with more practical scenarios and implementations built using Angular. We will touch upon important concepts in the last chapter of this book; these include multilingual support, authentication and authorization, communication patterns, performance optimizations, and a few others. You certainly do not want to miss them!

25-Oct-2018

Some Practical Scenarios

With seven chapters under our belt, you should feel pretty good. What you have learned thus far is a direct consequence of the apps we have built in the last few chapters. We believe you should now have an adequate understanding of the framework, how it works, and what it supports. Armed with this knowledge, as soon as we start to build some decent-sized apps, there are some common problems/patterns that invariably surface, such as these:

- How do we authenticate the user and control his/her access (authorize)?
- How do we make sure that the app is performing well enough?
- My app requires localized content. What do I do?
- What tools can I use to expedite app development?
- I have an AngularJS app. How do I migrate it?

And some more!

In this chapter, we will try to address such common scenarios and provide some working solutions and/or prescriptive guidance to handle such use cases.

The topics we will cover in this chapter include:

- **Angular seed projects:** You will learn how some seed projects in Angular can help us when starting a new engagement.
- **Authenticating Angular applications:** This is a common requirement. We look at how to support cookie- and token-based authentication in Angular.
- **Angular performance:** A customary performance section is a must as we try to detail what makes Angular performant and things you can do to make your apps faster.
- **Migrating AngularJS apps to Angular:** AngularJS and Angular are altogether different beasts. In this chapter, you will learn how to gradually migrate an AngularJS app to Angular.

Let's start at the beginning!

Building a new app

Imagine a scenario here: we are building a new application and given the super awesomeness of the Angular framework, we have unanimously decided to use Angular. Great! What next? Next is the mundane process of setting up the project.

Although a mundane activity, it's still a critical part of any engagement. Setting up a new project typically involves:

- Creating a standard folder structure. This is at times influenced by the server framework (such as *RoR*, *ASP.Net*, *Node.js*, and others).
- Adding standard assets to specific folders.
- Setting up the build, which in the case that, we are developing an Angular-based web application, includes:
 - Compiling/transpiling content if using TypeScript
 - Configuring the Module loader
 - Dependency management in terms of framework and third-party components
 - Setting up unit/E2E testing
 - Configuring builds for different environments such as dev, test, and production. Again, this is influenced by the server technology involved.
 - Code bundling and minification.

There is a lot of stuff to do.

What if we can short-circuit the overall setup process? This is indeed possible; we just need a **seed project** or a **starter site**.

Seed projects

Angular CLI as a built-in scaffolding tool is awesome! But, it is not the only option out there. There are a number of *seed projects/starter sites* that can get us up and running in no time. Some seed projects integrate the framework with a specific backend and some only dictate/provide Angular-specific content. Some come preconfigured with vendor-specific libraries/frameworks (such as *LESS*, *SASS*, *Bootstrap*, and others), whereas others just provide a plain vanilla setup.

Some of the notable seed projects worth exploring are as follows:

- **Angular Starter** (<http://bit.ly/ng-starter>): This seed repo serves as an Angular starter for anyone looking to get up and running with Angular and TypeScript fast. It uses **Webpack** (module bundler) to build our files and assist with boilerplate. It's a complete build system with a substantial number of integrations.
- **Angular Seed** (<http://bit.ly/ng-starter-seed>): Another seed project similar to Angular Starter. This seed project uses **gulp** for build automation, and the module bundler system is not as advanced as Webpack.

These projects along with Angular CLI provide a head start when building with Angular.

If the app is tied to a specific backend stack, we have two choices, which are as follows:

- Use one of these seed projects and integrate it with the backend manually.
- Find a seed project/implementation that does it for us. There is a good chance you will find such seed projects.

Angular performance

Angular has been designed with performance in mind. Every part of the framework, starting from the framework footprint, initial load time, memory utilization, change detection plus data binding, and DOM rendering, has been tweaked or is being tweaked for better performance.

The next few sections are dedicated to understanding how performant Angular is and the tricks it uses to achieve some impressive performance gains.

Byte size

The Byte size of the framework is a good starting point for performance optimization. While the world is moving towards high-speed internet, a sizable population among us is on a slow connection and are using their mobile to connect to the web. We may not think too much about a few KB here or there, but it does matter!

While the byte size of Angular out of the box is bigger than AngularJS, there are techniques that can drastically reduce the size of an Angular bundle.

To start with, the standard techniques of *minification* and *gzipping* can reduce this gap substantially. And with Angular, we can do some nifty tricks with *module bundler/loaders* to reduce the Angular bundle size even more.

Tree shaking may be a quirky name for a process, but it literally does what it says! As we build apps using TypeScript (or ES2015), containing *modules* and *exports*, a module bundler such as *Rollup* (<http://rollupjs.org>) can perform static code analysis on such code, determine what parts of the code are never used, and remove them before bundling the release bits. Such module bundlers, when added to the app's build process, can analyze the framework bit, any third-party library, and the app code to remove any dead code before creating bundles. *Tree shaking can result in enormous size reduction as you don't bundle framework bits that you don't use.*

One of the biggest framework pieces that can be removed from the framework bundle is the *compiler*. Yes, you read right, it's the compiler!



For curious readers, the compiler is the single biggest framework piece, contributing roughly 50% in size to the Angular bundle (In Angular v2.0.0).

Using tree shaking together with **Ahead-of-Time (AoT)** compilation, we can just get rid of the Angular compiler (in the browser) altogether.

With AoT compilation, the view templates (HTML) are compiled beforehand on the server side. This compilation again is done as part of the app's build process

where a server version of the Angular compiler (a node package) compiles every view in the application.

With all the templates compiled, there is no need to send the Angular compiler bits to the client side at all. Tree shaking can now just get rid of the compiler and create a far slimmer framework package. Angular CLI supports AoT compilation and can be used for production builds.



Read more about AoT in the framework documentation available at <http://bit.ly/ngx-aot>.

Initial load time and memory utilization

The initial load time for any web app with a full-fledged JavaScript framework is typically slow. This effect is more pronounced on mobile devices, where the JavaScript engine may not be as powerful as a desktop client. For a better user experience, it becomes imperative that the framework initial load time is optimized, especially for mobile devices.

Out of the box, **Angular 2 was five times faster than AngularJS** when it came to initial load time and re-rendering the view. These numbers are getting better as the Angular team evolves the framework.

Further, AoT compilation too can improve the initial load time of the application as a time-consuming activity (view compilation) is not required.

The same holds true for memory utilization. Angular fares better here too, and things will get even better with future releases.

If you are planning to switch to Angular, this is something that you should look forward to: a performant framework built for the future.

The next three performance improvements that we are going to talk about have been made possible because of a single architectural decision: *the creation of a separate renderer layer*.

The Angular rendering engine

The biggest disadvantage of AngularJS was that the framework was tied to the browser DOM. The directives, the binding, and the interpolations all worked against the DOM.

With Angular, the biggest architectural change that came in was a separate rendering layer. Now, an Angular app has two layers:

- **Application layer:** This is the layer our code resides in. It uses an abstraction build over the renderer layer to interact with it. The Renderer class we saw in [Chapter 6](#), *Angular Directives in Depth*, is the interface that we use to interact with the rendering layer.
- **Rendering layer:** This layer is responsible for translating requests from the application layer into rendered components, and reacting to user input and view updates.

The default renderer implementation for the renderer is `DomRenderer`, which runs inside the browser. But there are other rendering abstractions too and we will discuss them in the following section.

Server-side rendering

Pre-rendering on the server side is yet another technique for improving the initial load time of an Angular app. This technique is really helpful on mobile devices, as it improves the perceived load time considerably.

Server-side rendering takes care of the initial page load before client-side rendering kicks in (and handles view rendering henceforth).

In such a scenario, when the user requests for a view/page, a piece of software on the server generates a fully materialized HTML page with data pre-bound to the view and sends it to the client along with a small script. The app view is therefore immediately rendered, ready for interaction. While the framework loads in the background, the small script that was sent along the first time captures all user inputs and makes them available to the framework, allowing it to replay the interactions once it is loaded.

Angular Universal, as it is touted, allows rendering and sharing of the view both on the server and the client side.

Server-side rendering is only made possible because of separation of the rendering layer described previously. The initial view is generated by a renderer implementation on the server, named `serverDomRenderer`. There is a Node.js plugin (<http://bit.ly/ng-universal-node>) that can be used in a number of node web frameworks such as *Express*, *Hapi*, *Sail*, and others.



Look at the Angular design docs for Angular Universal (<http://bit.ly/ng-universal-design>) and the embedded YouTube videos at the top of the design doc to learn more about server-side rendering.

Performance is not the only benefit with server-side rendering. As it turns out, search indexers too like pre-rendered HTML content. Server-side rendering is really useful in areas such as **search engine optimization (SEO)** and deep linking, which allows easy content sharing.

Offloading work to a web worker

Offloading work to a **web worker** is a neat idea, again made possible due to the separation of the rendering layer from the application layer.

Web workers provide a mechanism for running scripts in background threads. These threads can execute work that does not involve the browser DOM. Be it a CPU-intensive task or a remote XHR invocation, all can be delegated to **web workers**.

In today's world, CPUs with multiple cores are the norm, but **JavaScript execution is still single-threaded**. There is a need for a standard/mechanism to utilize these idle cores for our apps. Web workers fit the bill perfectly, and since most modern browsers support them, we all should be writing code that utilizes web workers.

Sadly, that's not happening. Web workers are still not mainstream, and there are good reasons for that. Web workers impose a good number of restrictions on what is allowed and what is not. These limitations include:

- **No direct access to DOM:** Web workers cannot directly manipulate the DOM. In fact, web workers do not have access to multiple globals such as `window` and `document`, and others are not available on the web worker thread. This severely limits the number of use cases where a web worker can be utilized.
- **Browser support:** Web workers are only available for modern/evergreen browsers (IE 10+).
- **Inter-process communication:** Web workers do not share the memory with your main browser process, and hence need to communicate with the main thread (UI thread) only through *message passing* (serialized data). Adding to that, the message passing mechanism is asynchronous in nature, adding another layer of complexity to the communication model.

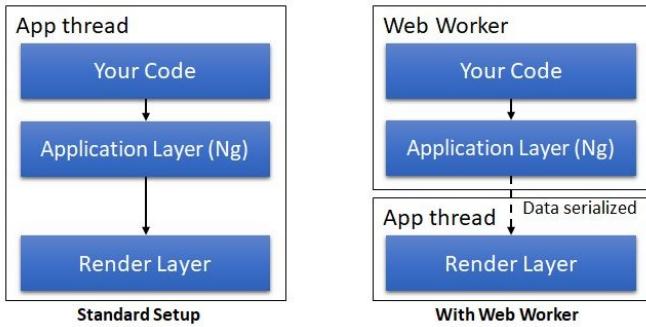
Clearly, web workers are hard to use.

Angular tries to alleviate these limitations by integrating the web worker usage

into the framework itself. It does that by running the complete application in the web worker thread, except the rendering part.

The framework takes care of the communication between the application code running inside the web worker and the renderer running inside the main UI thread. From a developer's perspective, there are no visible differences.

This is again made possible due to the separation of the renderer layer in Angular. The following diagram shows the layers that run on the app main thread and what runs inside the web worker:



Go through this talk (<http://bit.ly/yt-ng-web-worker>) from Jason Teplitz to learn about what web workers have to offer.

Performant mobile experience

Angular's rendering abstraction again opens up a host of integration avenues, especially on the mobile platform. Rather than running an app on a mobile browser, Angular renderers can be created that can tap the device's native UI capabilities.

Two notable projects in this domain are renderers for platforms:

- **ReactNative** (<http://bit.ly/rnative>): A renderer for ReactNative (<http://bit.ly/ng-rnative>). It allows the writing of Angular apps using ReactNative's rendering capabilities.
- **NativeScript** (<https://www.nativescript.org/>): Angular and NativeScript teams have collaborated to create a renderer for NativeScript (<http://bit.ly/ng-native-script>).

App platforms such as *ReactNative* and *NativeScript* already do a superb job of providing JavaScript-based APIs for the native mobile platforms (iOS and Android), allowing us to utilize a single code base with a familiar language. Angular renderers take things a step further. With Angular integration, a good amount of code can be shared across browsers and mobile devices. Things may only differ in terms of view templates and view-related services such as dialogs, popups, and others.

Look at the documentation for the respective renderers to understand how they work and the features they support.

Next up on the line, we have framework improvements in terms of *change detection*.

Change detection improvements

One of the major performance improvements in Angular over AngularJS is in how *change detection* works in Angular. Out of the box, Angular change detection is insanely fast, and it can be tweaked even further for better results.

The next few sections talk about Angular change detection in depth. It's an important topic to understand when building anything at scale. It also helps us debug scenarios where it may seem that change detection is not working as advertised.

Let's start the discussion by understanding what change detection is and why it is important.

Change detection

Angular's *data binding engine* does a great job of binding the view with the model data (component data). These are live bindings where Angular keeps the view in sync with model changes. Any time the model changes, the binding engine re-renders parts of the view that are dependent on the model. To manage this view-model synchronization, Angular needs to know when the model changed and what changed exactly. This is what **change detection** is all about. During app execution, Angular frequently does what we call **change detection runs** to determine what changed.



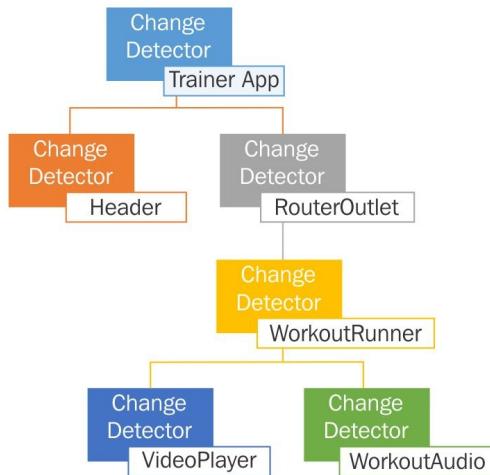
If you are from AngularJS, a change detection run is roughly equivalent to **digest cycles**, except that in Angular there are **no cycles**.

While this problem of keeping the model and view in sync may sound simple, it's a tough nut to crack. Unlike the component tree, the interconnection between multiple models can be complex. Changes in one component model can trigger changes in multiple component models. Furthermore, these interconnections may have cycles. A single model property could be bound to multiple views. All these complex scenarios need to be managed using a robust change detection infrastructure.

In the next few sections, we explore how the Angular change detection infrastructure works, when change detection triggers, and how can we influence change detection behavior in Angular.

Change detection setup

It all starts with Angular setting up change detectors for every component rendered on the view. Since every Angular app is a hierarchy of components, these change detectors are also set up in the same hierarchy. The following diagram highlights the **change detector hierarchy** of the *Workout Builder* app at a point in time:



A change detector attached to a component has the responsibility of detecting changes in the component. It does that by parsing the binding on the component's template HTML and setting up the necessary change detection watches.

Remember, the detector only sets up watches on model properties used in the template, not on all component properties.

Another important point worth highlighting here is that *change detection is set up one way, from model to view*. Angular does not have the concept of two-way data binding and hence the preceding figure is a directed tree without cycles. This also makes change detection more predictable. Interleaving model and view updates are disallowed.

When does change detection kick in?

Does Angular constantly check for changes in the model data? Considering the fact that the component properties we bind the view to do not inherit from any special class, Angular has no way of knowing which property changed. The only way out for Angular is to constantly query each data-bound property to know its current value and compare it against its old value for changes. Highly inefficient to say the least!

Angular does better than that, as change detection runs get executed only at specific times during app execution. Think carefully about any web application; what causes a view to update?

A view can get updated due to:

- **User input/browser events:** We click on a button, enter some text, or scroll the content. Each of these actions can update the view (and the underlying model).
- **Remote XHR requests:** This is another common reason for view updates. Getting data from a remote server to show on the grid and getting user data to render a view are examples of this.
- **setTimeout and setInterval timers:** As it turns out, we can use `setTimeout` and `setInterval` to execute some code asynchronously and at specific intervals. Such code can also update the model. For example, a `setInterval` timer may check for stock quotes at regular intervals and update the stock price on the UI.

For obvious reasons, Angular change detection too kicks in only when any of these conditions occur.

The interesting part here is not when Angular's change detection kicks in but how Angular is able to intercept all *browser events*, *XHR requests*, and `setTimeout` and `setInterval` functions.

This feat in Angular is performed by a library called `zone.js`. As the documentation describes:

"A Zone is an **execution context** that **persists** across **async tasks**."

One of the basic abilities of this library is that it can hook into a piece of code and trigger callbacks when code execution starts and when it ends. The code being monitored could be a sequence of calls that are both synchronous and asynchronous in nature. Consider this example, which highlights the usage:

```
let zone = new NgZone({ enableLongStackTrace: false });
let dowork = function () {
  console.log('Working');
};

zone.onMicrotaskEmpty.subscribe((data:any) => {
  console.log("Done!");
});

zone.run(() => {
  dowork();
  setTimeout(() => {
    console.log('Hard');
    dowork();
  }, 200);
  dowork();
});
```

like AOP ?

We wrap a piece of code inside a call to the `zone.run` call. This code calls the `dowork` function synchronously twice, interleaved with a `setTimeout` call that invokes the same function after a lapse of 200 milliseconds.

By wrapping this sequence inside `zone.run`, we can know when the call execution is complete. In zone terminology, these are **turns**. The code before `zone.run` sets up a subscriber that gets called when execution is complete, using the `zone.onMicrotaskEmpty` function:

If we execute the preceding code, the logs look as follows:

```
Working // sync call
Working // sync call
Done! // main execution complete
Hard    // timeout callback
Working // async call
Done! // async execution complete
```

The `onMicrotaskEmpty` subscription is executed twice, once after the sequential execution completes (defined inside `run` callback) and one after the asynchronous `setTimeout` execution is complete.

Angular change detection uses the same technique to execute our code within

zones. This code could be an *event handler*, which internally makes more synchronous and asynchronous calls before completing, or it could be a `setTimeout/setInterval` operation that may again require a UI update.

The Angular change detection framework subscribes to the `onMicrotaskEmpty` observable for the executing zone, and kicks in change detection whenever a turn is complete. The following diagram highlights what happens when code similar to the one just described is run on a button click:



During the execution of the code block, if the zone library determines that the call is asynchronous in nature, it spawns a new micro task that has its own life cycle. It is the completion of these micro tasks that also triggers `onMicrotaskEmpty`.

If you want to know how the change detection trigger looks inside Angular, here is an excerpt from the Angular source code (simplified further):

```
class ApplicationRef {
  constructor(private zone: NgZone) {
    this._zone.onMicrotaskEmpty.subscribe(
      {next: () => { this._zone.run(() => { this.tick(); }); }});
  }
  tick() {
    this._views.forEach((view) => view.detectChanges());
  }
}
```

The `ApplicationRef` class tracks all the change detectors attached throughout the app and triggers a change detection cycle when the application-level zone object fires the `onMicrotaskEmpty` event. We will shortly touch upon what happens during this change detection.

`Zone.js` gets the ability to track execution context across any asynchronous call because it overrides the default browser API. The override, also termed **monkey patching**, overrides the *event subscription, XHR requests, and*

`setTimeout/setInterval` API. In the example highlighted previously, the `setTimeout` we invoke is a monkey-patched version of the original browser API.

Now that we know how change detectors are set up and when this activity kicks in, we can look at how it works.

How does change detection work?

Once the change detectors are set up and the browser API is monkey-patched to trigger change detection, the real change detection kicks in. This is quite a simple process.

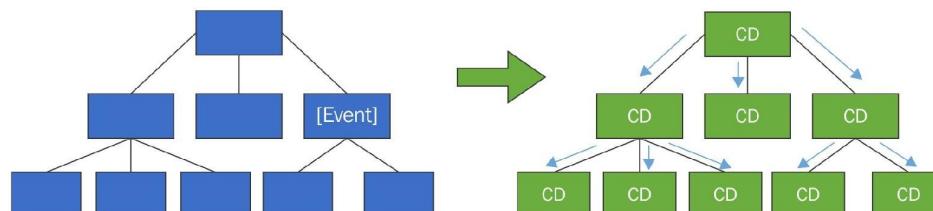
As soon as any of the asynchronous event callbacks are triggered (the execution of an event handler is also an async activity), Angular first executes the application code we attached to the callback. This code execution may result in some model updates. After the execution of the callback, Angular needs to respond to the changes by triggering a *change detection run*.

In a change detection run, starting from the top of the component tree, every change detector evaluates its respective component's template bindings to see if the value of the binding expression has changed.

There are some things that we need to highlight regarding this execution:

- Angular does a strict equality check (using `==`) to detect changes. Since it's not a deep comparison, for a binding that refers to an object, Angular will only update the view when the object reference changes.
- The change detection flow is unidirectional (starting from root), from parent to child in a top-down fashion. The detectors on the parent component run before the child detectors.

By default, the change detection algorithm navigates the complete tree, irrespective of where the change was triggered in the tree. This implies all binding is evaluated on every change detection run:



A binding evaluation on every run may seem inefficient, but it is not. Angular employs some advance optimizations to make this check super fast. Still, if we want to tweak this behavior, we do have some switches that can reduce the number of checks performed. We will touch

upon this topic soon.

- Change detectors only track properties that are part of template bindings, not the complete object/component properties.
- To detect changes in the bound value, the change detectors need to track the previous value of the expression evaluated during the last change detection run. Clearly, some amount of bookkeeping is required for every template binding we use.

The obvious next question would be what happens when a change is detected (by a change detector)?

Since all the hard work of setting up change detection and identifying changes has already been done, this step just involves updating the component state and synchronizing the component DOM.

There are a few more observations worth highlighting here:

- First and foremost, Angular separates the model update step from the DOM update. Consider this code snippet, which is invoked when someone clicks on a button:

```
dowork() {  
    this.firstName="David";  
    this.lastName="Ruiz";  
}
```

Assuming that both `firstName` and `lastName` are bound to the component view, a change to `firstName` does not update the DOM binding immediately. Instead, Angular waits for the `dowork` function to complete before triggering a change detection run and DOM update.

- Secondly, a change detection run does not (and should not) update the model state. This avoids any cycles and cascading updates. A change detection run is only responsible for evaluating the bindings and updating the view. This also means that we should not update the model state during change detection. If we update the model during change detection, Angular throws an error.

Let's see an example of this behavior:

1. Open `start.component.html` from the 7 Minute Workout app and update the last `div` to:

```
<div class="col-sm-3">  
    Change detection done {{changeDetectionDone()}}  
</div>
```

2. And, add a `changeDetectionDone` function to the component implementation (`start.component.ts`), which looks like:

```
times: number = 0;  
changeDetectionDone(): number {  
    this.times++;  
    return this.times;  
}
```

3. Run the app, load the start page, and then look at the browser console. Angular has logged a number of errors that look like the following:

```
EXCEPTION: Expression has changed after it was checked.  
Previous value: 'Change  
detection done 1'. Current value: 'Change detection done 2' ...
```

We are changing the state of the component when calling the `changeDetectionDone` function (inside an interpolation), and Angular throws an error because it does not expect the component state to update.

This change detection behavior is enabled only when **production mode** in Angular is disabled. Production mode can be enabled by calling the `enableProdMode()` function before bootstrapping the application ([in bootstrap.ts](#)). When enabled, Angular behaves a bit differently. It turns off assertions and other checks within the framework. Production mode also affects the change detection behavior. In non-production mode, Angular traverses the component tree twice to detect changes. If on the second pass any binding expression has changed, it throws an error. In contrast, when in production mode, change detection tree traversal is done only once. The change detection error that we saw on the [console will not show up if we enable production mode](#). This can lead to an inconsistency between the model and view state. Something we should be aware of! [The bottom line is that we cannot alter the state of a component when change detection is in progress](#). A direct corollary: if we are using a function inside the binding expression, function executions should be stateless, without any side effects.

- Lastly, this change detection traversal from root to leaf node executes only once during the change detection run.

i A pleasant surprise for folks with an AngularJS background! The digest cycle count in Angular is 1. Angular developers will never face "the digest iterations exceeded exception!" A far more performant change detection system!

Change detection performance

Let's talk about change detection performance. If you think checking the complete component tree every time for change is inefficient, you will be surprised to know how fast it is. Due to some optimization around how expressions are evaluated and compared, Angular can perform thousands of checks in a couple of milliseconds.

Under the hood, for every expression involved in the view binding, Angular generates a change detection function that specifically targets a particular binding. While it may seem counter intuitive at first, Angular does not have a common function for determining whether an expression has changed. Instead, it's like writing our own change detection function for every property that we bind to. This allows the JavaScript VM to optimize the code, resulting in improved performance.



Want to learn more about it? Check out this video by Victor Savkin: <https://youtu.be/jvKGQSFQf10>.

In spite of all this optimization, there may still be cases where traversing the complete component tree may not be performant enough. This is especially true when we have to render a large dataset on the view, keeping the bindings intact. The good news is that the Angular change detection mechanism can be tweaked.

The reason Angular needs to do the complete tree walk is that model changes in one place may trigger model changes at other places. In other words, a model change may have a cascading effect, where interconnected model objects are also updated. Since Angular has no way to know what exactly changed, it checks the complete component tree and associated model.

If we can help Angular determine what parts of the application state are updated, Angular can be pretty smart about what part of the component tree it traverses to detect changes. We do this by storing the app data in some special data structures that help Angular decide what components need to be checked for changes.

There are three ways in which we can make Angular change detection smarter.

Using immutable data structures

Immutable objects/collections are objects that cannot be changed once created. Any property change results in a new object being created. This is what `immutable.js`, a popular library for immutable data structures, has to say:

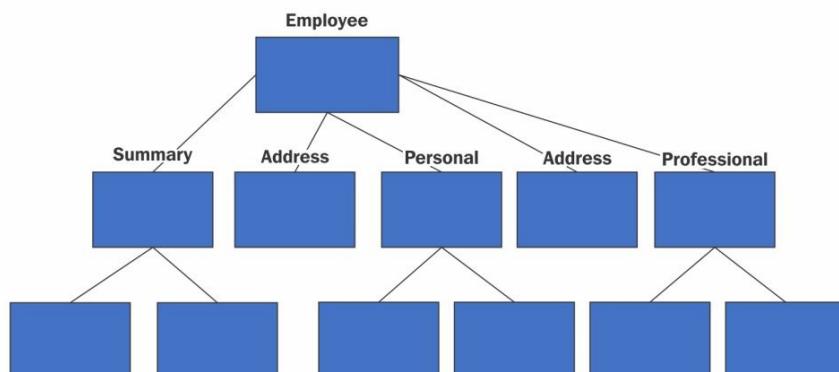
Immutable data cannot be changed once created, leading to much simpler application development, no defensive copying, and enabling advanced memoization and change detection techniques with simple logic.

Let's try to understand how immutable data structures help in the Angular context with an example.

Imagine we are building a set of components to collect employee information for a **Human Resource (HR)** software. The employee component view looks something like this:

```
<Employee>
  <summary [model]="employee"></employee>
  <personal [model]="employee.personal"></personal>
  <professional [model]="employee.professional"></professional>
  <address [model]="employee.home"></address>
  <address [model]="employee.work"></address>
</Employee>
```

It has sections for taking personal, professional, and address information. The `summary` components provide a read-only view of the employee data being entered. Each of the components has a property called `model`, highlighting what part of employee data these components manipulate. Each of these components' summary, professional, personal, and address internally may have other child components. This is how the component tree looks:



What happens when we update an employee's personal information? With standard objects (mutable), Angular cannot make any assumption about the shape of data and what has changed; hence, it does the complete tree walk.

How does immutability help here? When using an immutable data structure, any change to an object's properties results in a new object being created. For example, if we create an immutable object using a popular library, `Immutable.js`:

```
| personalInfo = Immutable.Map({ name: 'David', 'age': '40' });
```

Changes to either the `name` or `age` property of `personalInfo` create a new object:

```
| newPersonalInfo = personalInfo.set('name', 'Dan');
```

This immutability comes in handy if each of the employee model properties (`personal`, `professional`, `home`, and `work`) is immutable.

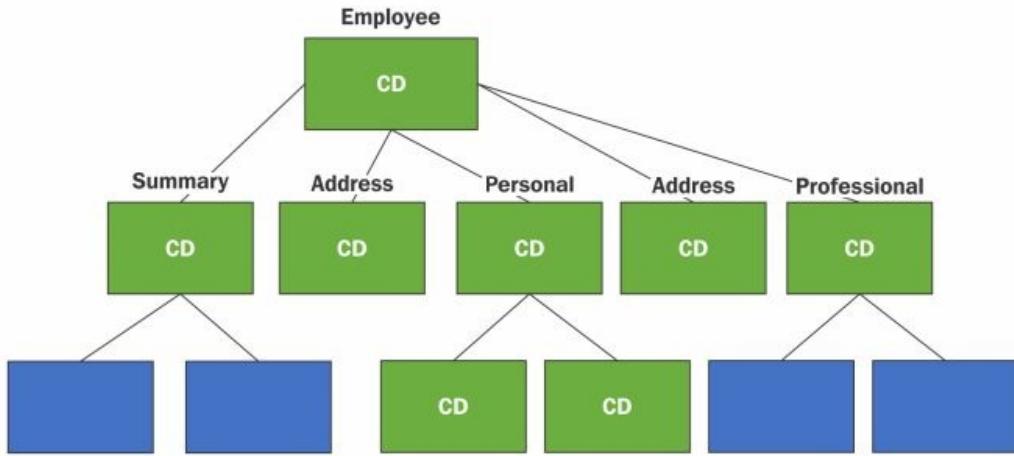
Take, for instance, the `PersonalInfo` component definition that binds to personal info data:

```
@Component({
  selector: 'personal',
  template: `
    <h2>{{model.name}}</h2>
    <span>{{model.age}}</span>`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
class PersonalInfo {
  @Input() model;
}
```

Since the only thing `PersonalInfo` depends upon is the `model` property, and the `model` property binds to an immutable object, Angular only needs to do a check for changes if the `model` reference changes. Otherwise, the complete `PersonalInfo` component subtree can be skipped.

By setting the `PersonalInfo` component property `changeDetection` to `ChangeDetectionStrategy.OnPush`, we instruct Angular to trigger change detection only when the component's inputs change.

If we change the change detection strategy to `onPush` for each of the Employee component children and update the employee's personal info, only the `PersonalInfo` component subtree is checked for changes:



For a large component tree, such an optimization will improve the app/view performance many times over.



When set to `onPush`, Angular triggers change detection only when the component's input property changes or there is an event raised inside the component or its children.

Developing applications using immutable data structures departs from the standard development paradigm where the application state is totally mutable. What we have highlighted in this section is how Angular takes advantage of immutable data structures to optimize the change detection process.

Observables are another kind of data structure that can help us optimize Angular change detection.

Using Observables

Observables are data structures that trigger events when their internal state changes. The Angular *eventing infrastructure* extensively uses *Observables* to communicate the components' internal state to the outside world.

While we have used Observable output properties (the `EventEmitter` class) to raise events, input properties too can take Observables. Such observable inputs can help Angular optimize change detection.

When using observables, the change detection switch still remains `changeDetectionStrategy.OnPush`. But this time, only if a component input triggers an event (as they are observables) will Angular perform the dirty check. When the input triggers an event, the complete component tree path, starting from the affected component to the root, is marked for verification.

When performing the view update, Angular will only sync the affected path and ignore the rest of the tree.

Manual change detection

We can actually disable change detection on a component completely and trigger manual change detection when required. To disable change detection, we just need to inject the component-specific change detector (the `ChangeDetectorRef` class instance) into the component and call the `detach` function:

```
constructor(private ref: ChangeDetectorRef) {  
    ref.detach();  
}
```

Now, the onus is on us to inform Angular when the component should be checked for changes.



We can reattach the component to the change detection tree by using the `reattach` function on `ChangeDetectorRef`.

We seldom need to disable the standard change detector setup, unless there are situations where standard change detection becomes an expensive affair.

Take, for example, a public chatroom app, which is receiving messages from thousands of people connected to it. If we constantly keep pulling the messages and refreshing the DOM, the app may become unresponsive. In such a scenario, we can disable change detection on parts of the chat app component tree and manually trigger change detection to update the UI at specific intervals.

While we have seen three ways to tweak change detection behavior, the good thing is that these are not exclusive. Parts of the component tree can use immutable data structures, parts can use Observables, parts can employ manual change detection, and the rest can still use the default change detection. And Angular will happily oblige!

Enough on change detection for now. We may never need it unless we are building some large views with a chatty UI. Such scenarios require us to squeeze every bit of performance out of the change detection system, and the system is ready for it.

Next, we will have a look at another common requirement that most apps

invariably have: authenticating their users.

Handling authentication and authorization

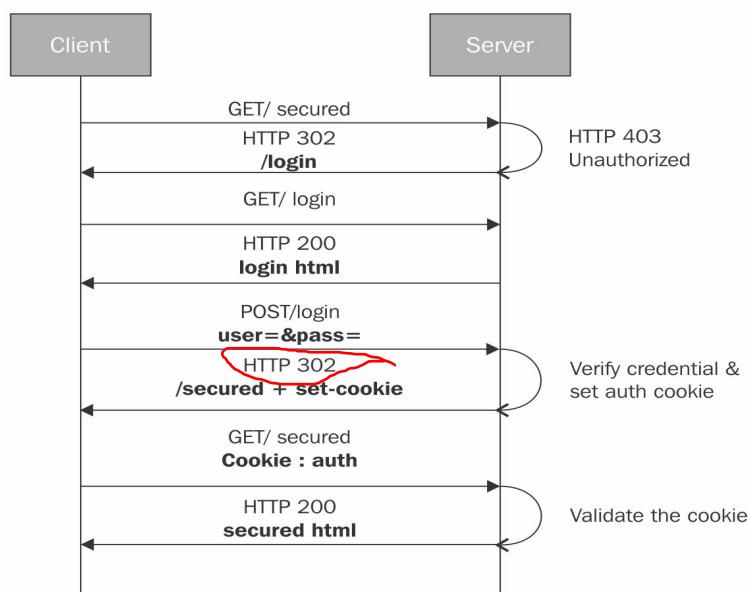
Most, if not all, apps have a requirement to authenticate/authorize their users. We may argue that authentication and authorization are more of a server concern than a client one, and that is correct. Still, the client side needs to adapt and integrate with the authentication and authorization requirement imposed by the server.

In a typical app execution workflow, the app first loads some partial views then makes calls to pull data from the server, and finally binds data to its view. Clearly, *the views* and *the remote data API* are two assets that need to be secured.

To guard these resources, we need to understand how a typical application is secured on the server. There are primarily **two broad approaches to securing any web applications**: *cookie-based authentication* and *token-based authentication*. Each of them requires different handling on the client part. The next two sections describe how we can integrate with either of these approaches.

Cookie-based authentication

This authentication mechanism is the easiest to implement if the server stack supports it. It's non-intrusive and may require bare minimum changes to the Angular application. **Cookie-based authentication** involves setting the browser cookie to track the user authentication session. The following sequence diagram explains a typical cookie-based authentication workflow:



Here is how a typical authentication workflow works:

- When trying to access a secured resource from the browser, if the user is not authenticated, the server sends an HTTP 401 Unauthorized status code. A user request is an unauthorized request if there is no cookie attached to the request or the cookie is expired/invalid.
- This unauthorized response is intercepted by the server or, at times, by the client framework (Angular in our case) and it typically results in a 302 redirect (if intercepted by the server). The redirect location is the URL to the login page (the login page allows anonymous access).
- The user then enters the username and password on the login page and does a POST to the login endpoint.
- The server validates the credentials, sets a browser cookie, and redirects the user to the originally requested resource.

- Henceforth, the authentication cookie is a part of every request (added by the browser automatically), and the server uses this cookie to confirm his identity and whether the user is authenticated.

As we can see, with this approach, the Angular infrastructure is not involved, or the involvement is minimal. Even the login page can be a standard HTML page that just sends data to the login endpoint for authentication. If the user lands on the Angular app, it implicitly means that the user has been authenticated.



The cookie-based authentication flow may vary depending on the server framework, but the general pattern of setting a cookie and attaching a cookie with every subsequent request remains the same.

In a cookie-based application authentication, if the application wants to get the user context, a server endpoint (such as `/user/details`) is exposed that returns the logged in user's specific data. The client application can then implement a service such as `userService` that loads and caches the user profile data.

The scenario described here assumes that the API server (the server that returns data) and the site where the application is hosted are in a single domain. That may not always be the case. Even for *Personal Trainer*, the data resides on the *MongoLab* servers and the application resides on a different server (even if it is local). And we already know that this is a cross-domain access and it comes with its own set of challenges.

In such a setup, even if the API server is able to authenticate the request and send a cookie back to the client, the client application still does not send the authentication cookie on a subsequent request.

To fix this, we need to set a variable, `withCredentials`, to `true` on each XHR request. This can be enabled at the global level by overriding `BaseRequestOptions` (the `withCredentials` property). The framework uses the `BaseRequestOptions` class to set the default values for every HTTP request.

This can also be enabled on a per-request level by passing in the `withCredentials:true` flag in each HTTP request method as the last parameter:

```
|this.httpService.get(url,{withCredentials:true});
```

The last parameter to every HTTP function, including `get`, `post`, and `put`, is an

options object. This allows us to override the options for the request being made.

Once this flag is enabled, the client browser will start attaching the authentication cookie for the cross-domain requests.

The server too needs to have **cross-origin resource sharing (CORS)** **enabled** and needs to respond in a specific manner for the request to succeed. It should set the **access-control-allow-credentials** header to **true** and the **access-control-allow-origin** header to the host site making the request.



Check out the MDN documentation (<http://bit.ly/http-cors>) to learn about this scenario in detail.

Cookie-based authentication is definitely less work on the client side, but there are times when you have to revert to token-based access. This could be because:

- Cookies and cross-domain requests do not play nicely across browsers. Specifically, IE8 and IE9 do not support them.
- The server may not support generating cookies, or the server only exposes token-based authentication.
- Token-based solutions are easy **to integrate with a native mobile application and desktop clients**.
- Tokens are not susceptible to cross-site request forgery (CSRF) attacks.

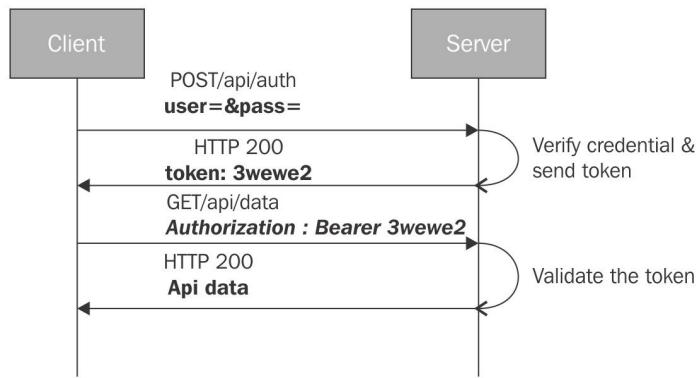


To know more about CSRF, look at the CSRF Prevention cheat sheet at <http://bit.ly/csrf-cs>.

The next section talks about supporting token-based authentication.

Token-based authentication

Token-based access is all about sending a token (typically in HTTP headers) with each request instead of a cookie. A simplified token-based workflow looks something like this:



Many public APIs (such as *Facebook* and *Twitter*) use token-based authentication. The format of the token, where it goes, and how it is generated depends on the protocol used and the server implementation. Popular services that use token-based authentication implement the **OAuth 2.0** protocol for token generation and exchange.

In a typical token-based authentication setup, the views are available publicly but the API is secured. If the application tries to pull data through API calls without attaching the appropriate token to the outgoing request, the server returns an *HTTP 401 Unauthorized* status code.

Enabling token authentication is a good amount of work in Angular. It involves setting up a login page/component, guard conditions, authentication service, and authentication context shared across the application. If you embark on this journey, make sure to look for Angular libraries/module that makes this integration easier. The `ngx-auth` library (<http://bit.ly/ngx-auth>) can be a good starting point.

That's all about authentication, but what about authorization? Once the user context is established, we still need to make sure that the user is only able to

access parts that he/she is allowed to. *Authorization* is still missing.

Handling authorization

Like authentication, authorization support too needs to be implemented on both the server and client side, more so on the server than the client. Remember, anyone can hack into the JavaScript code and circumvent the complete authentication/authorization setup. So, always tighten your server infrastructure irrespective of whether the client has the necessary checks in place or not.

This still does not mean that we do not do any authorization checks on the client. For standard users, this is the first line of defense against unwarranted access.

When working on an authorization requirement for any application, there are three essential elements that are part of the setup:

- The resources that need to be secured/authorized
- A list of roles and users that are part of these roles
- A mapping between the resources and the roles that define who can access what

From an Angular app perspective, the resources are the pages and, sometimes, sections of pages that need to be restricted to specific roles. If the user is in a specific role, depending upon the role-resource mapping, they get access to some pages; otherwise, they are denied access.

While authorization in an Angular application can be implemented in a number of ways, we will outline a generic implementation that can be further customized to suit your needs in the future.

Adding authorization support

To enable authorization, the first thing that we need to do is expose the logged in user data, including his/her roles, throughout the application.

Sharing user authentication context

User context can be shared using an Angular service, which can then be injected into components that require the authorization context. Look at this service interface:

```
class SessionContext { currentUser():User { ... };  
isUserInRole(roles:Array<string>):boolean { ... }; isAuthenticated:boolean; }
```

The `SessionContext` service tracks the user login session and provides details such as:

- The logged in user (`currentUser`)
- Whether the user is authenticated (`isAuthenticated`)
- The `isUserInRole` function, which returns `true` or `false` based on whether the user is part of any of the roles passed into the `roles` parameter

With such a service in place, we can add authorization for routes, thereby restricting access to some routes to specific roles only.

Restricting routes

Like authentication, the `canActivate` guard check can also be used for authorization. Implement a class with the `CanActivate` interface and inject the `SessionContext` service into the constructor; then, check whether the user belongs to a specific role in the `canActivate` function using the `SessionContext` service. Check out the following code snippet:

```
export class AuthGuard implements CanActivate {  
  
  constructor(private session:SessionContext) { }  
  
  canActivate() {  
  
    return this.session.isAuthenticated &&  
  
      session.isUserInRole(['Contributor', 'Admin']);  
  
  }  
}
```

Only users with roles of *Contributor* and *Admin* now have access to routes that have this guard condition.

But what happens when a page has view elements that are rendered based on the user's role?

Conditionally rendering content based on roles

Conditionally rendering content is easy to implement. We just need to show/hide HTML elements based on the user role. We can build a *structural directive* such as `ng-if` that can verify that the user belongs to a role before rendering the content. The directive's usage looks like:

```
<div id='header'>
  <div> Welcome, {{userName}}</div>
  <div><a href='/setting/my'>Settings</a></div>
  <div *a2beRolesAllowed='["admin"]'>
    <a href='/setting/site'>Site Settings</a>
  </div>
</div>
```

The preceding code checks whether the user is in an admin role before rendering a Site Setting hyperlink.

The directive implementation mimics how `ng-if` works, except that our show/hide logic depends upon the `SessionContext` service. Here is a sample implementation for the `a2beRolesAllowed` directive:

```
@Directive({ selector: '[a2beRolesAllowed]' })
export class RolesAllowedDirective {
  private _prevCondition: boolean = null;
  constructor(private _viewContainer: ViewContainerRef,
    private _templateRef: TemplateRef, private SessionContext _session) { }

  @Input() set a2beRolesAllowed(roles: Array<string>) {
    if (this._session.isUserInRole(roles)) {
      this._viewContainer
        .createEmbeddedView(this._templateRef);
    } else {
      this._viewContainer.clear();
    }
  }
}
```

This is a trivial implementation that uses `SessionContext` and the roles passed as input (`a2beRolesAllowed`) to show/hide a fragment.

This brings us to the end of authentication and authorization implementation.

The reference implementation walkthrough should help us build authentication and authorization into our apps. With this basic understanding in place, any setup can be tweaked to handle other custom authentication/authorization scenarios.

It's now time to address the elephant in the room: migrating from *AngularJS* to *Angular*. If you are starting afresh on Angular, you can **very well skip the next section.**



From the Angular's migration guide "Angular is the name for the Angular of today and tomorrow. AngularJS is the name for all v1.x versions of Angular."

26-Oct-2018

i do skip this topic at all.

Migrating AngularJS apps

If you have been working extensively on AngularJS, Angular poses some pertinent questions:

- Should I migrate my old AngularJS apps to the latest Angular version?
- When should the migration happen?
- Is the migration one-shot or can it be done in an incremental fashion?
- What is the effort involved?
- Can I do something today that helps with the migration in the future?
- I am starting a new AngularJS app today. What should I do to make the migration seamless in the future?

Every such query needs to be addressed to make sure the transition is as smooth as possible. No one likes surprises later in the game! In the coming sections, we will try to answer a number of such questions. As part of the learning, we will also walk you through migrating the AngularJS version of the *Trainer* app (developed for the first version of this book) to Angular. This will help everyone make some informed decisions on when and how to migrate to Angular.

"Should I migrate or not" is something that we will address first.

Should I migrate?

Just because Angular is here doesn't mean AngularJS is gone. AngularJS is still being developed, albeit not at the same pace as Angular. Google is still committed to supporting AngularJS for a good amount of time. The AngularJS team is working on version 1.7 currently and plan to release it before June 2018. After that, 1.7 will enter the **Long Term Support (LTS)** period, where the focus will only be on bug fixes. We can learn more about it from their blog post at <http://bit.ly/ng1-lte-support>.

Clearly, AngularJS will continue to be supported and hence should not be a major concern for migration. The move to Angular can be decided based on what Angular has to offer over its predecessor.

Advantages of Angular

Angular is designed for the future and overcomes a number of shortcomings in its predecessor. In this section, we emphasize what makes Angular a better framework than AngularJS.

Things you should be aware of while making a decision to move to Angular:

- **Better behavioral encapsulation:** Admittedly, while AngularJS *scopes* seem to be god sent when we started learning AngularJS, we have now realized how difficult it is to manage the hierarchical nature of scopes. Component-based development in Angular provides a better encapsulation in terms of the state of the application. A component manages its own state, takes input, and raises events; a clear demarcation of responsibilities that are easy to reason with!
- **Less of the framework in app code:** You don't need special objects such as a *scope*. DI works with annotation (in TypeScript). You don't set up watches. All in all, when reading a component code, you will not find framework-level constructs in it.
- **Smaller framework API to explore:** AngularJS had a host of directives that one had to be aware of. With Angular template syntax, directives related to browser events are gone. This reduces the number of directives that one needs to be aware of.
- **Performance:** Angular is faster compared to its predecessor. A complete section of this book was dedicated to understanding what makes Angular a high-performance framework.
- **Mobile-friendly:** Angular tries to optimize the user's mobile experience by utilizing technologies such as server-side rendering and web workers. Angular applications on mobile are more performant than those of its predecessor.
- **Cross-platform:** Angular targets running on most devices and across platforms. You can use Angular to build applications for web and mobile. As we learned earlier, the separation of the rendering layer has open up a great number of possibilities in terms of where Angular can be utilized.

In a true sense, Angular supersedes its predecessor, and in a perfect world, everyone should be working on a better framework/technology. But migration is never easy, especially in this case, where the two frameworks differ a lot.

What we recommend is even if you do not plan to migrate to Angular today, build your AngularJS apps in ways that allow easy migration to Angular in future.

The next section talks about the principles and practices to follow for AngularJS today, allowing easy migration in the future.

Developing AngularJS apps today for easy migration

The new Angular is a paradigm shift and the way we develop components in Angular is quite different from AngularJS. For easy migration, Angular too should embrace component-based development. This can be achieved if we follow some guidelines/principles while building AngularJS apps. The next few sections detail these guidelines.



The advice listed here is highly recommended even if you do not plan to migrate to Angular. These recommendation bits will make AngularJS code more modular, organized, and testable.

One component per file

This can be anything: an AngularJS *controller*, *directive*, *filter*, or *service*. One component per file allows better organization of code and easy migration, allowing us to clearly identify how much progress has been made.

Avoiding inline anonymous functions

Use named functions instead to declare controllers, directives, filters, and services. A declaration such as this: `angular.module('7minWorkout').controller('WorkoutController',[...])`

```
angular.module('app')  
.directive('remoteValidator', [...])
```

```
angular.module('7minWorkout')  
.filter('secondsToTime', function () { ... })
```

```
angular.module('7minWorkout')  
.factory('workoutHistoryTracker', [...])
```

Should be converted to this: `function WorkoutController($scope, ...) { ... }`

```
WorkoutController.$inject = ['$scope', ...];
```

```
function remoteValidator($parse) {...}
```

```
remoteValidator.$inject=[ $parse];
```

```
function secondsToTime() {...}
```

```
function workoutHistoryTracker($rootScope, ...) { ...}
```

```
workoutHistoryTracker.$inject = ['$rootScope', ...];
```

The advantages of using named functions are ease of debugging and ease of migration to TypeScript. Using named functions also requires that the dependencies be registered using the `$inject` function property.



`$inject`-based dependency declaration safeguards against minification and adds to the readability of the functions.

To avoid exposing global name functions with this approach, it is advisable to wrap the function in an **Immediately Invoked Function Expression (IIFE)**:

```
(function() {
```

```
    function WorkoutController($scope, ...) { ... }
```

```
    WorkoutController.$inject = ['$scope', ...];  
    angular.module('7minWorkout')
```

```
        .controller('WorkoutController', WorkoutController);
```

```
})();
```

Avoiding \$scope!

Yes, you read it right; avoid the `$scope/$rootScope` object or using scopes directly!

The biggest problem with AngularJS scopes is their hierarchical nature. Accessing the parent scope from the child scope gives us tremendous flexibility, but it comes at a cost. This can unknowingly create unwarranted dependencies that make the app really hard to debug and, of course, migrate. In contrast, in Angular, a view is bound to its component implementation and cannot access data outside its boundary implicitly. Therefore, if you plan to migrate to Angular, *avoid scopes at all costs.*

There are a number of techniques that can be used to remove the `$scope` object dependency. The next few subsections elaborate on some techniques that can help us avoid AngularJS scopes.

Using controller as (controller aliasing) syntax everywhere

AngularJS 1.3+ has the *controller as* syntax for *controllers*, *directives*, and *routes*. *controller as* syntax allows AngularJS data binding expressions to work against a controller's instance properties instead of the current *scope*'s object properties. With the controller as paradigm in place, we never need to interact with the scope directly, and hence future migration becomes easy.



While controller aliasing gets rid of scope access, scopes are still there in AngularJS. The complete AngularJS data binding infrastructure depends upon scopes. Controller aliasing just puts an indirection between our code and scope access.

Consider the following syntax for *controller as* in views:

```
<div ng-controller="WorkoutListController as workoutList">
  <a ng-repeat="workout in workoutList.workouts" href="#/workout/{{workout.name}}">
</div>
```

And the corresponding controller implementation:

```
function WorkoutListController($scope, ...) {
  this.workouts=[];
```

`WorkoutListController as workoutList` creates an alias `workoutList` for `WorkoutListController` on the current scope, hence allowing us to bind to the `workouts` property defined on the controller.

Route definition too allows controller aliasing using the `controllerAs` property in a *route definition object*:

```
$routeProvider.when('/builder/workouts', {
  ...
  controller: 'WorkoutListController',
  controllerAs: 'workoutList'
});
```

Finally, directives too can use `controllerAs`, and together with the `bindToController` property on the *directive definition object*, we can get rid of any direct scope

access.

 *Look at the Angular documentation on controllers, routes, and directives to get a basic understanding of the controller as syntax. Also, look at the following posts for some more detailed samples on this topic: <http://bit.ly/ng1-controller-as> and <http://bit.ly/ng1-bind-to>.*

Avoiding ng-controller

If scopes can be avoided, so can controllers!

This may again seem counterintuitive, but the approach has real benefits. What we ideally want to do is emulate component behavior in AngularJS. Since the closest thing to components in AngularJS is *element directives* (with `restrict='E'`), we should utilize *element directives* everywhere.

An AngularJS element directive with its own template and isolated scope can very well behave like an Angular component and only be dependent on its internal state for its view binding. We just don't need `ng-controller`.

Consider the use of `ng-controller` for audio tracking from the AngularJS version of the *Personal Trainer* app:

```
<div id="exercise-pane" class="col-sm-7">
  ...
  <span ng-controller="WorkoutAudioController">
    <audio media-player="ticksAudio" loop autoplay src="content/tick10s.mp3"></audio>
    <audio media-player="nextUpAudio"  src="content/nextup.mp3"></audio>
  ...
</span>
```

Instead of using `WorkoutAudioController`, an element directive can encapsulate the workout audio's view and behavior. Such a directive can then replace the complete `ng-controller` declaration and its view:

```
<div id="exercise-pane" class="col-sm-7">
  ...
  <workout-audio-component></workout-audio-component>
```

When replacing `ng-controller` with an element directive, the scope variables that the controller depends upon should be passed to the directive using the `bindToController` property on the *directive definition object*—something like this:

```
bindToController: {
  name: '=',
  title: '&amp;',
}
```



This topic has been extensively covered in these two blogs posts by Tero: <http://bit.ly/ng2-no-controllers> and <http://bit.ly/ng2-refactor-to-component>. Must-read posts with a wealth of information!

Building using the AngularJS 1.5+ component API

AngularJS 1.5+ has a **component API** that allows us to create directives that can be easily migrated to Angular. The component API is preconfigured with sensible defaults, hence incorporating the best practices when it comes to building truly isolated and reusable directives.

Look at the component API at <http://bit.ly/ng1-dev-guide-components> and this informative post by Tod Motto at <http://bit.ly/1MahwNs> to learn about the component API.

To reiterate what has been emphasized earlier, these steps are not just targeted towards easy Angular migration but also towards making AngularJS code better. Component-based UI development is a better paradigm than what we are used to with AngularJS.



We highly recommend that you go through the AngularJS style guide (<http://bit.ly/ng1-style-guide>). This guide contains a wealth of tips/patterns that allow us to build better AngularJS apps, and is in sync with the guidelines provided previously for easy Angular migration.

Finally, if we have decided to migrate, it's time to decide what to migrate.

What to migrate?

For an app in maintenance mode, where most of the development activity revolves around bug fixes and some enhancements, it would be prudent to stick to AngularJS. Remember the old saying *if it ain't broke, don't fix it*.

If the app is being actively developed and has a clear long-term roadmap, migrating to Angular is worth considering. As we dig deeper into the intricacies of migration, we will realize the time and effort involved in the process. While the Angular team has worked really hard to make this migration smooth, by no stretch of the imagination is this a trivial job. It is going to take a good amount of time and effort to perform the actual migration.

The silver lining here is that we do not need to migrate everything at once. We can work slowly towards migrating parts of the AngularJS code base to Angular. Both the frameworks can coexist and can depend on each other too. This also allows us to develop new parts of applications in Angular. How cool is that?

But again, this flexibility comes at a cost—the cost of bytes. As both frameworks are downloaded, the page bytes do increase, something that we should be aware of.

Also, while the coexistence of both the frameworks allows us to migrate without much disruption, we cannot make it a perpetual activity. Eventually, AngularJS has to go, and the sooner it does the better.

During migration, the best thing that can be done is to carve out new SPAs within the existing application. For example, we can build the Admin area of an app entirely using Angular, with a separate host page, but still share the common infrastructure of style sheets, images, and even AngularJS services if we refactor the code a bit. As we will learn later, migrating services to Angular is the easiest.

Breaking an application into multiple smaller ones introduces full-page refreshes, but this is a cleaner approach when it comes to migration.

Taking all of this into consideration, if we have decided to migrate and identified

areas of migration, you need to do the prep work for migration.

Preparing for Angular migration

Welcome to the big brave world of Angular migration! A successful migration strategy involves making sure that we do the groundwork beforehand, avoiding any late surprises.

As prep work, the first step is to analyze the application from a third-party library dependency perspective.

Identifying third-party dependencies

Any third-party library that an AngularJS app uses needs a migration strategy too. These could be either jQuery-based libraries or AngularJS libraries.

jQuery libraries

jQuery libraries in AngularJS were consumed by creating a directive wrapper over them. We will have to migrate such directives to Angular.

AngularJS libraries

Migrating AngularJS libraries is a bit of a tricky affair. AngularJS has a massive ecosystem, and Angular too has been around for some time and now has a healthy community. When migrating, each AngularJS library needs to be substituted with an Angular alternative.

If we do not find a perfect upgrade path for a specific library, we can either:

- Customize a similar component/library available out there
- Take the more radical approach of building our own library from the ground up in Angular

Each of these choices has trade-offs in terms of time and complexity.

Another choice that needs to be made is the development language. Should we use TypeScript, ES2015, or plain old JavaScript (ES5)?

Choice of language

We would definitely recommend TypeScript. It's a super awesome language that integrates very well with Angular and vastly reduces the verbosity of Angular declarations. Also, given that it can coexist with JavaScript, it makes our lives easier. Even without Angular, TypeScript is one language that we should embrace for the web platform.

In the coming sections, we will migrate the AngularJS *Personal Trainer* app to Angular. The app is currently available on *GitHub* at <http://bit.ly/a1begit>. This app was part of the first version of this book, *AngularJS by Example*, and was built using JavaScript.



We are again going to follow the checkpoint-based approach for this migration. The checkpoints that we highlight during the migration have been implemented as GitHub branches. Since we will be interacting with a Git repository for v1 code and using Node.js tools for the build, please set up Git and Node.js on your development box before proceeding further.

Migrating AngularJS's Personal Trainer

Before we even begin the migration process, we need to set up the v1 *Personal Trainer* locally.

The code for the migrated app can be downloaded from the GitHub site at <https://github.com/chandermani/angularjsbyexample>. Since we migrate in chunks, we have created multiple checkpoints that map to **GitHub branches** dedicated to migration. Branches such as `ng6-checkpoint8.1`, `ng6-checkpoint8.2`, and so on highlight this progression. During the narration, we will highlight the branch for reference. These branches will contain the work done on the app up to that point in time.



The 7 Minute Workout code is available inside the repository folder named `trainer`.

So, let's get started!

Setting up AngularJS's Personal Trainer locally

Follow these steps and you will be up and running in no time:

1. From the command line, clone the v1 GitHub repository:

```
| git clone https://github.com/chandermani/angularjsbyexample.git
```

2. Navigate to the new Git repo and check out the `ng6-base` branch to get started:

```
| cd angularjsbyexample  
|   git checkout ng6-base
```

3. Since the app loads its workout data from **MongoDB** hosted in **mLab** (<https://mlab.com/>), you need an mLab account to host workout-related data. Set up an mLab account by signing up with them. Once you have an mLab account, you need to retrieve your API key from mLab's management portal. Follow the instructions provided in the API documentation (<http://bit.ly/mlab-docs>) to get your API key.
4. Once you have the API key, update this line in `app/js/config.js` with your API key:

```
| ApiKeyAppenderInterceptorProvider.setApiKey("<yourapikey>");
```

5. And add some seed workout data into your mLab instance. The instructions to add the seed data into mLab are available in the source code file, `app/js/seed.js`.

6. Next, install the necessary *npm packages* required for v1 *Personal Trainer*:

```
| cd trainer/app  
| npm install
```

7. Install `http-server`; it will act as a development server for our v1 app:

```
|   npm i http-server -g
```

Verify that the setup is complete by starting the `http-server` from the `app` folder:
http-server -c-1

And open the browser location `http://localhost:8080`.

The v1 *Personal Trainer* start page should show up. Play around with the app to verify that the app is working fine. Now, the migration can begin.

Identifying dependencies

The first step before we begin migrating v1 *Personal Trainer* is to identify the external libraries that we are using in the AngularJS version of Personal Trainer.

The external libraries that we are using in v1 are:

- angular-media-player
- angular-local-storage
- angular-translate
- angular-ui-bootstrap
- owl.carousel

Libraries such as `angular-media-player` and `angular-local-storage` are easy to migrate/replace. We have already done this in earlier chapters of this book.

The `angular-translate` library can be replaced with `ngx-translate`, and as we will see in the coming sections, it is not a very challenging task.

We use `angular-ui-bootstrap` for **modal dialogs** in *Personal Trainer v1*. We replace it with `ngx-modal` (<http://bit.ly/ngx-modal>) as the only control we were using from `angular-ui-bootstrap` was the dialog control.

Now that we have sorted out the external dependencies, let's decide the language to use.

While the existing code base is JavaScript, we love TypeScript. Its type safety, its terse syntax, and how well it plays with Angular makes it our language of choice. Hence, it's going to be TypeScript all the way.

Another thing that tilts the decision in favor of TypeScript is that we do not need to migrate the existing code base to TypeScript. Anything we migrate/build new, we build it in TypeScript. Legacy code still remains in JavaScript.

Let's start. As a first migration task, we need to set up a module loader for our v1 Personal Trainer.

Setting up the module loader

Since we are going to create a number of new Angular components spread across numerous small files, adding direct script reference is going to be tedious and error-prone. Angular CLI also is not of much help here as it cannot manage the existing codebase implemented in JavaScript.

We need a **module loader**. A module loader (*ES6 modules* and not Angular) can help us with:

- Creating isolated/reusable modules based on some common module formats
- Managing the script loading order based on dependencies
- Allowing bundling/packaging of a module and on-demand loading for dev/production deployments

We use the **SystemJS** module loader for this migration.

Install SystemJS from the command line using:

```
| npm i systemjs --save
```



All the commands need to be executed from the `trainer/app` folder.

We open `index.html` and remove all the script references of our app scripts. All script references with the source as `src='js/*.*'` should be removed, except `angular-media-player.js` and `angular-local-storage.js`, as they are external libraries.



Note: We are not removing script references for third-party libraries, but only app files.

Add SystemJS configurations after all third-party script references:

```
<script src="js/vendor/angular-local-storage.js"></script>
<script src="node_modules/systemjs/dist/system.src.js">
</script>
<script>
  System.config({ packages: {'js': {defaultExtension: 'js'}}});
  System.import('js/app.js');
</script>
```

Remove the `ng-app` attribute on the `body` tag, keeping the `ng-controller` declaration

intact:

```
| <body ng-controller="RootController">
```

The `ng-app` way of bootstrapping has to go as we switch to the `angular.bootstrap` function for manual bootstrapping. Manual bootstrapping helps when we bring Angular into the mix.

The preceding `SystemJS.import` call loads the application by loading the first app module (JavaScript) defined in `js/app.js`. We are going to define this JavaScript module shortly.

Create a new file called `app.module.js` in the same folder as `app.js` and copy the complete contents of `app.js` into `app.module.js`.



Remember to get rid of the `use strict` statement. The TypeScript compiler does not like it.

All the app module definitions are now in `app.module.js`.

Next, clear `app.js` and add the following imports and bootstrap code:

```
import './app.module';
import './config.js';
import './root.js';
import './shared/directives.js';
import './shared/model.js';
import './shared/services.js';
import './7MinWorkout/services.js';
import './7MinWorkout/directives.js';
import './7MinWorkout/filters.js';
import './7MinWorkout/workout.js';
import './7MinWorkout/workoutvideos.js';
import './WorkoutBuilder/services.js';
import './WorkoutBuilder/directives.js';
import './WorkoutBuilder/exercise.js';
import './WorkoutBuilder/workout.js';

import * as angular from "angular";

angular.element(document).ready(function() {
  angular.bootstrap(document.body, ['app'],
  { strictDi: true });
});
```

We have added *ES6 import statements* to `app.js`. These are the same scripts that were earlier referenced in `index.html`. SystemJS now loads these script files when loading `app.js`.

Moving all of the AngularJS module declarations into a new file, `app.module.js`, and importing it first into `app.js` makes sure that the AngularJS modules are defined before any of the `import` statements are executed.



Do not confuse ES6 modules and AngularJS modules defined/accessed using `angular.module('name')`. These two are altogether different concepts.

The last few lines bootstrap the AngularJS application using the `angular.bootstrap` function.

Module loading is enabled now; let's enable TypeScript too.

Enabling TypeScript

To enable TypeScript, install the TypeScript compiler using `npm`:

```
| npm i typescript -g
```

Next, open `package.json` and add these lines inside the script configuration:

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1"  
  "tsc": "tsc -p . -w"  
}
```



The new script's properties that we just added provide a shortcut for commonly executed commands.

To enable IDE IntelliSense for JavaScript libraries, we need to install their type definitions. **Type definitions** or **typings** are files that define the public interface for a TypeScript/JavaScript library. These type definitions help IDEs provide IntelliSense around the library functions. typings are available for most of the popular JavaScript libraries and for frameworks/libraries written in TypeScript.

Let's install typings for the libraries we use. From the command line, execute:

```
| npm install @types/angular @types/angular-route @types/angular-resource @types/angular-mocks --save-dev
```

Next, copy `overrides.d.ts` from `checkpoint8.1` to the local `app` folder. This helps TypeScript compiler to handle the global `angular` object used in `app.js` and other places.

We now need to set up some configurations for the TypeScript compiler. Create a file called `tsconfig.json` (in the `trainer/app` folder) and copy the configuration from the `ng6-checkpoint8.1` repo branch (also available remotely at <http://bit.ly/ng6be-8-1-tsconfig>). Run the compiler using:

```
| npm run tsc
```

This should start the TypeScript compiler, and there should be no errors reported.





Keep this command running in a separate console window at all times during development. The compiler will continuously watch for changes to the TypeScript file and rebuild the code if changes are detected.

Change the extension of the `app.js` and `app.module.js` files to `app.ts` and `app.module.ts`. The TypeScript compiler detects these changes and compiles the TypeScript files. After compilation, the compiler produces two files for each TypeScript file. One is the compiled JavaScript file (such as `app.js`) and the other is a map file (`app.js.map`) for debugging purposes.



We have not set up an elaborate build for this exercise as our primary focus is on migration. For your own apps, the initial setup steps may vary depending upon how the build is already set up.

Before we test our new changes, `config.js` needs to be fixed because we have enabled strict DI check-in AngularJS through:

```
| angular.bootstrap(document.body, ['app'], { strictDi: true });
```

Replace the `config.js` content with updated content available in `ng6-checkpoint8.1` or at <http://bit.ly/ng6be-8-1-configjs> (and remember to set the API key again). The update fixes the `config` function and makes it minification-friendly. Time to test the app!

Make sure the TypeScript compiler is running in one console; run `http-server -c-1` in a new console window.

Navigate to `http://localhost:8080` the app start page should load.

Commit/save your local changes.



If things work fine, you can even commit your local changes to your git repo. This will help you track what has changed over time as we migrate the app piece by piece. The implementation till this point is available in the `ng6-checkpoint8.1` GitHub branch. If you are facing problems, compare the `ng6-base` and `ng6-checkpoint8.1` branches to understand the changes made. Since the code is hosted in GitHub, we can use the GitHub compare interface to compare commits in a single branch. See the documentation on how to do it here: <http://bit.ly/github-compare>. The link at <http://bit.ly/ng6be-compare-base-8-1> shows a comparison between `ng6-base` and `ng6-checkpoint8.1`. You can ignore the diff view for `app.js` and `app.module.js`, generated as part of the TypeScript compilation.

Time to introduce Angular!

Adding Angular

We start by installing Angular and dependent *npm modules* for our app. We will update the `package.json` file with the necessary packages first.

Copy the updated package file from <http://bit.ly/ng6be-8-2-package-json> into your local installation.

`package.json` now references some new packages related to Angular. Install the referenced packages by calling:

| `npm install`

 *If you are having trouble with installing packages with `npm install`, delete the `node_modules` folder and run `npm install` again.*

Then, add a few library references that Angular is dependent upon (and are not loaded using SystemJS) in `index.html` before the `system.src.js` script reference (two in total):

```
|<script src="/node_modules/core-js/client/shim.min.js"></script>
|<script src="/node_modules/zone.js/dist/zone.js"></script> <script
|src="/node_modules/systemjs/dist/system.src.js"></script>
```

As it stands now, the SystemJS configuration has been set up in the `index.html` file itself. Since Angular requires some decent amount of configuration, we are going to create a separate *SystemJS configuration file* instead, and reference that in `index.html`.

Add this script reference after the `system.src.js` reference:

```
|<script src="systemjs.config.js"></script>
```

Now, clear the `script` section containing the call to the `system.config` function and replace it with the following:

```
|<script>System.import('app');</script>
```

Copy the `systemjs.config.js` from <http://bit.ly/ng6be-8-2-system-js-config> and place it in the same folder as `package.json`.

Also, update `tsconfig.json` and add a new property called `moduleResolution` to `compilerOptions`:

```
| "removeComments": false,  
|   "moduleResolution": "node"
```

This instructs TypeScript to look for type definitions in the `node_modules` folder. Remember, Angular typings are bundled as part of the Angular library itself, and hence a separate type definition import is not required.

Now that the Angular-specific references have been added, we need to modify the existing bootstrap process to also load Angular.

The ngUpgrade Library

To support gradual migration from AngularJS to Angular, the Angular team has released a library, `ngUpgrade`. The library contains a set of services that allow AngularJS and Angular to be loaded in tandem and play well together. This library has services that can help:

- Bootstrap an app with both the AngularJS and Angular frameworks loaded. This is the first thing we are going to do.
- Incorporate an Angular component in an AngularJS view.
- Incorporate an AngularJS component in an Angular view, albeit with some limitations.
- Share dependencies across the frameworks.

The primary tool in this library is `upgradeModule`. As the platform documentation summarizes: "This is a module that contains utilities for bootstrapping and managing hybrid applications that support both Angular and AngularJS code."

As we make progress with our migration efforts, the role of `upgradeModule` becomes clearer.

Let's learn how to bootstrap the hybrid AngularJS and Angular app using `upgradeModule`.

Bootstrapping the hybrid app

To bootstrap a hybrid application, we must bootstrap both the Angular and AngularJS parts of the application. The sequence involves the process is as follows:

1. Bootstrap the Angular app
2. Then, using `upgradeModule`, bootstrap the AngularJS app

Since Angular has just been added, we need to define the root app module for Angular.

Create a new file, `app-ng1.module.js`, and copy the complete content of `app.module.ts` to the new file. We will use the `app.module.ts` file to define an Angular module and hence the existing AngularJS module has been shifted to a new file.

Also, remember to update the `import` statement in `app.ts` in line with the changes:
`import './app-ng1.module.js';`

Let's now add the Angular root module definition to `app.module.ts`.

Replace the content of `app.module.ts` with the Angular module definition. Copy the new definition from `ng6-checkpoint8.2` (GitHub location: <http://bit.ly/ng6be-8-2-app-module-ts>).

```
The AppModule implementation defines a function, ngDoBootstrap: constructor(private upgrade: UpgradeModule) { }
ngDoBootstrap() {
    this.upgrade.bootstrap(document.documentElement, ['app']);
}
```

The Angular framework invokes this function as part of the application bootstrap. This function internally uses the `bootstrap` function of `upgradeModule` to bootstrap the AngularJS app. This function takes the same argument as the `angular.bootstrap` function takes.

While we have defined the root module for Angular, we still have not defined the entry point for the Angular application. Create a new file, `main.ts`, in the `app` folder and add the following code:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
import './app';
platformBrowserDynamic().bootstrapModule(AppModule);
```

This code, when executed, instructs the Angular framework to bootstrap the application with `AppModule`. The statement `import './app'` helps in loading all the script references in `app.ts`. Before proceeding, do remember to remove the AngularJS application bootstrapping code from `app.ts` (the complete `angular.element` block).

Refresh your app and make sure it works as before. Do watch out for errors in the TypeScript compiler console window or browser console log.

Congratulations! We now have a hybrid app up and running. Both frameworks are now working in tandem.



Look at the `ng6-checkpoint8.2` branch if you are facing issues upgrading to Angular. Again, you can also compare the git branches `ng6-checkpoint8.1` and `ng6-checkpoint8.2` to understand what has changed (<http://bit.ly/ng6be-compare-8-1-8-2>).

The migration process can start now. We can start by migrating a part of an AngularJS view/directive to Angular.

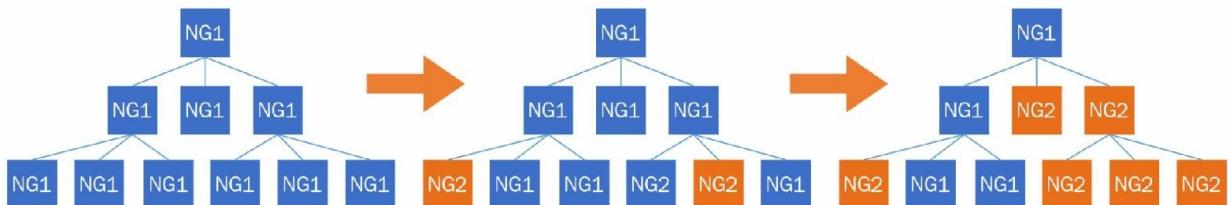
Injecting Angular components into AngularJS views

The most common migration pattern involves migrating lower-level AngularJS directives/views to Angular components. If we visualize the AngularJS HTML view structure as a tree of directives, we start at the leaf. We migrate parts of a directive/view to an Angular component and then embed the component inside the AngularJS view template. This Angular component is injected into the AngularJS view as an *element directive*.



*The closest thing to **Angular components** that AngularJS has is **element directives**. During migration, we are either migrating element directives or controller-view pairs.*

This is a bottom-up approach to migrating view/directives to Angular components. The following diagram highlights how the AngularJS view hierarchy gradually transforms into an Angular component tree:



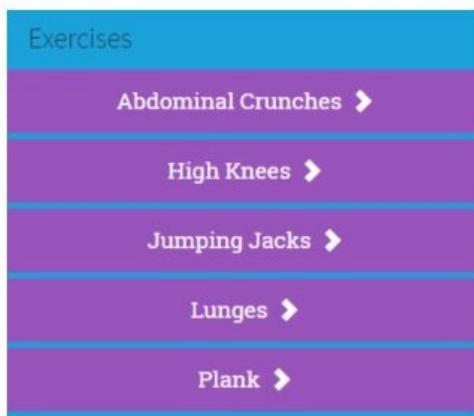
Let's migrate something small and get a feel for how things work. `ExerciseNavController` and its corresponding view fit the bill.

Migrating our first view to Angular component

`ExerciseNavController` is part of *Workout Builder* and is located inside `trainer/app/js/WorkoutBuilder/exercise.js`. The corresponding view is served from `trainer/app/partials/workoutbuilder/left-nav-exercises.html`.

The primary purpose of this controller-view pair is to show the list of available exercises when building a workout (available user path

`http://localhost:8080/#/builder/workouts/new`):



Clicking on any of these exercise names adds the exercise to the workout being constructed.

Let's start with creating a component for the preceding view.



*Before starting on the new component, add a new *Workout Builder* module (`WorkoutBuilderModule`) to the application. Copy the module definition from `ng6-checkpoint8.3` in the *WorkoutBuilder* folder (GitHub location: <http://bit.ly/ng6be-8-3-workout-builder-module-ts>). Also, import the newly created module in `app.module.ts`.*

Instead of inlining the complete code here, we suggest copying the `exercise-nav.component.ts` file from GitHub branch `ng6-checkpoint8.3` (<http://bit.ly/ng6be-8-3-exercise-nav-component-ts>) and adding it to the `WorkoutBuilder` folder locally. The file has already been referenced in `WorkoutBuilderModule`. We also add `ExerciseNavController` to

`entryComponents` array as the component will be used from AngularJS module directly.

From an implementation standpoint, let's look at some relevant parts.



Contrasting template syntax

The Angular team has published an excellent reference (<http://bit.ly/ng2-a1-a2-quickref>) that details the common view syntaxes in AngularJS and their equivalents in Angular. Highly recommended when migrating an AngularJS app!

To start with, if you look at the `exercise-nav.component.ts` file, the component template is similar to `left-nav-exercises.html` used in AngularJS, except there is no `ng-controller` and the template bindings are Angular-based:

```
template: `<div id="left-nav-exercises">
  <h4>Exercises</h4>
  <div *ngFor="let exercise of exercises" class="row">
    ...
  </div>`
```

If we focus on the component implementation (`ExercisesNavComponent`), the first striking thing is the component's dependencies:

```
constructor(
  @Inject('WorkoutService') private workoutService: any,
  @Inject('WorkoutBuilderService') private workoutBuilderService: any)
```

`WorkoutService` and `WorkoutBuilderService` are AngularJS services injected into Angular components.

Nice! If that is your initial reaction, we can't blame you. The ease with which we can inject AngularJS service into Angular is pretty cool! But the story is still incomplete. There is no magic happening here. Angular cannot access AngularJS services unless it is told where to look. To help Angular, we need to create *factory provider* wrappers for the AngularJS service.

Injecting AngularJS dependencies into Angular

When we allow an AngularJS service to be used in Angular, we are *upgrading* the service. The way it is done is by creating a **factory provider wrapper** over the existing AngularJS service and registering the wrapper with an Angular module or component.

Let's create wrappers for the two AngularJS services, `workoutService` and `workoutBuilderService`, that we have used in the last section.

A factory provider for an AngularJS service can be created using Angular's dependency injection API. Here is an example of a factory provider for `WorkoutService`:

```
export function workoutServiceFactory(injector: any) {
  return injector.get('WorkoutService');
}

export const workoutServiceProvider = {
  provide: 'WorkoutService',
  useFactory: workoutServiceFactory,
  deps: ['$injector']
};
```

In this code, `$injector` is the AngularJS *injector service* and we are referencing the injector in the Angular execution context. The call to `injector.get('WorkoutService');` in the previous factory function retrieves the service from AngularJS realm.

The provider created can then be registered with the app module:

```
| providers:[workoutServiceProvider]
```

The shortcoming of this approach is that we have to define the factory function (`workoutServiceFactory`) and provider (`workoutServiceProvider`) for each service. That is too much code!

Instead, we can create a generic factory provider and factory function

implementation, which can take any AngularJS service and register it with the same name (*string token*) in Angular. We have already done the hard work and created a new class, `UpgradeHelperService`. Download it from the codebase for `ng6-checkpoint8.3` (<http://bit.ly/ng6be-upgrade-helper-service-ts>).

The implementation exposes a function, `upgradeService`, which takes one argument, the name of the AngularJS service, and returns a factory provider instance. The factory provider implementation internally uses a *string token* to register the dependency. To create a factory provider for an AngularJS service, we just need to call:

```
| UpgradeHelperService.upgradeService('WorkoutService')
```

Service dependencies at times have other dependencies, and hence it's better if we bring in all the service dependencies from AngularJS in one go. Register all the existing AngularJS dependencies as factory providers in `app.module.ts`:

```
providers: [
  UpgradeHelperService.upgradeService('ExercisePlan'),
  UpgradeHelperService.upgradeService('WorkoutPlan'),
  UpgradeHelperService.upgradeService('WorkoutService'),
  UpgradeHelperService.upgradeService('WorkoutBuilderService'),
  UpgradeHelperService.upgradeService('ExerciseBuilderService'),
  UpgradeHelperService.upgradeService('ApiKeyAppenderInterceptor'),
  UpgradeHelperService.upgradeService('appEvents'),
  UpgradeHelperService.upgradeService('workoutHistoryTracker'),
]
```

Back to component integration! As the new `ExercisesNavComponent` is rendered inside an AngularJS view, it needs to be registered as an *AngularJS directive*.

Registering Angular components as directives

`ExercisesNavComponent` can be converted into an AngularJS directive using `ngUpgrade` library function `downgradeComponent`. As the function name suggests, we are downgrading an Angular component into an AngularJS element directive.

Open `app.ts` and add the highlighted lines:

```
import {ExercisesNavComponent} from './WorkoutBuilder/exercise-nav-component'
import { downgradeComponent } from '@angular/upgrade/static';
...
angular.module('WorkoutBuilder')
  .directive('exerciseNav', downgradeComponent({ component: ExercisesNavComponent })
  as angular.IDirectiveFactory);
```

The `downgradeComponent` function returns a *factory function* containing the *directive definition object*. We register the component as an AngularJS directive, `exerciseNav`.



Every Angular component is registered as an element directive when used in AngularJS.

The component implementation is complete. We now need to clean up the old code and inject the new directive into the view.

Delete the definition of `ExercisesNavController` from `exercise.js`.

Replace the content of `left-nav-exercises.html` (located in the `partials` folder) with:
`<exercise-nav></exercise-nav>`

And we are good to go.

Note that we do not get rid of `left-nav-exercises.html` as AngularJS still loads `left-nav-exercises.html` as part of the route transition, but the view inside is an Angular component.

Go ahead and try out the new implementation. Create a new workout and try to

add exercises from the left nav. The functionality should work as before.



Look at `ng6-checkpoint8.3` in case you are facing issues upgrading to Angular. You can compare the git branches `ng6-checkpoint8.2` and `ng6-checkpoint8.3` to understand what has changed (<http://bit.ly/ng6be-compare-8-2-8-3>).

While we have only migrated a trivial component, this exercise highlights how easy it is to convert/downgrade an Angular component to the AngularJS directive and use it in an AngularJS view. The overall encapsulation of an Angular component makes this chore easy.

This downgraded component can even take an input from the parent scope using the familiar Angular property binding syntax: `<exercise-nav [exercises]='vm.exercises'></exercise-nav>`

Add to that, the event raised by the component can be subscribed by the AngularJS container scope too:

```
| <exercise-nav (onExerciseClicked)='vm.add(exercise)'></exercise-nav>
```

We now have an Angular component running inside AngularJS using services initially designed for AngularJS. A promising start to our migration journey!

Before we move any further, it's time to highlight how this collaboration works and the rules of engagement.

Rules of engagement

The migration story from AngularJS to Angular is only possible because these frameworks can coexist, and possibly share data. There are some touch points where the boundaries can be crossed. To have a better sense of how a hybrid application works and what is achievable in such a setup, we need to understand the areas of collaboration between the two frameworks.

There are three areas that need discussion:

- Template interleaving in the DOM
- Dependency injection
- Change detection

Since Angular components and AngularJS directives can coexist in a DOM, the question we need to answer is who owns what parts of the DOM?

AngularJS directives and Angular components

When it comes to ownership of a DOM element, the golden rule is:

Every DOM element is owned/managed by exactly one of the Angular frameworks.

Take our previous migration example. The view that is part of `ExercisesNavComponent` is managed by Angular, whereas the container view (`left-nav-exercises.html`) is managed by AngularJS.

Things get a bit tricky at the boundaries of these directives and components. Consider the declaration inside `left-nav-exercises.html`:

```
|<exercise-nav></exercise-nav>
```

Who owns this? The short answer is AngularJS.

While this is an Angular component, the host element is owned by AngularJS. This means all AngularJS template syntax works:

```
|<exercise-nav ng-if='showNav'></exercise-nav>
|<exercise-nav ng-repeat='item in items'></exercise-nav>
```

As these components and directives coexist in the same view, they often need to communicate. There are two ways to manage this communication:

- Using the templating capabilities of AngularJS and Angular:
 - An Angular component embedded inside an AngularJS view can take inputs from the parent scope using event and property binding
 - In a similar fashion, if a directive is injected into an Angular component view, it too can get inputs from the parent component and call the parent component function (through its isolated scope)
- Using shared services; we saw an example of this previously as we injected the `WorkoutService` and `WorkoutBuilderService` AngularJS services into `ExercisesNavComponent`



Injecting AngularJS directives into Angular is a bit tricky. To be able to inject an AngularJS directive into an Angular template, the directive needs to abide by some rules. We will talk about these rules in the coming sections.

Sharing functionality using services is far more flexible compared to sharing through view templates. Injecting services across framework boundaries requires us to register the service across both frameworks and let Angular take care of the rest. Let's learn how dependency injection works across boundaries.

Resource sharing and dependency injection

How dependencies are registered in a hybrid app is driven by how DI works in these two frameworks. For AngularJS, there is only one global injector, whereas Angular has a concept of hierarchical injectors. In a hybrid environment, the least common denominator is the global injector that both the frameworks support.

Sharing an AngularJS service

As we saw in the factory provider example earlier, AngularJS services can be registered with Angular by creating a wrapper factory provider.

Since dependency injection in AngularJS is string token-based, the corresponding providers too use string tokens to locate dependencies in Angular.

Looking back at the example of dependency registration share earlier, the dependency was registered with a help of a helper class:
UpgradeHelperService.upgradeService('WorkoutService')

And it was injected using the `Inject` decorator (with a string token):

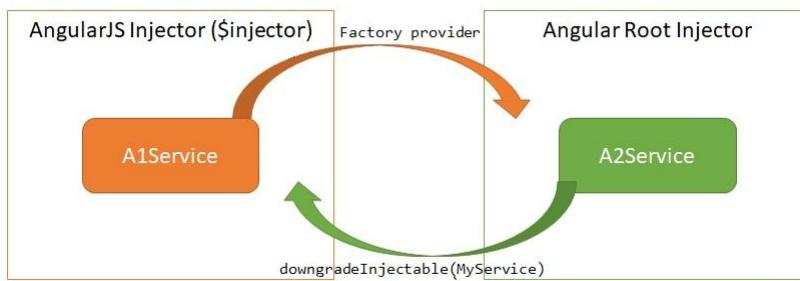
```
| constructor(@Inject('WorkoutService') private workoutService: any...)
```

Sharing an Angular service

Services from Angular too can be injected into AngularJS. Since AngularJS only has a global injector, the dependency is registered at the global level. The `ngUpgrade` library has a `downgradeInjectable` function for this. The `downgradeInjectable` function creates a factory function than can be consumed by the AngularJS module's factory API: `angular.module('app').factory('MyService', downgradeInjectable(MyService))`

`MyService` can now be injected across the AngularJS app, like any other service.

Look at the following diagram; it summarizes what we have discussed:



One last topic of this discussion is change detection.

Change detection

In a hybrid application, change detection is managed by Angular. If we are used to calling `$scope.$apply()` in our code, we don't need to do it in a hybrid application.

We have already discussed how Angular change detection works. The Angular framework takes care of triggering AngularJS change detection by internally calling `$rootScope.$apply()` on standard triggering points.

Now that we understand the rules of engagement, it is easier to comprehend how things work, what's feasible, and what's not.

Let's set some bigger/meatier targets and migrate the start and finish pages of the v1 app.

Migrating the start and finish pages

The finish page migration is easy to do, and I suggest you do it yourself. Create a folder called `finish` inside the `js` folder, and create two files, `finish.component.ts` and `finish.component.html`. Implement the component by looking at the existing implementation (or copy it from `ng6-checkpoint8.4`).

Add the component to the `declarations` and `entryComponents` array on `AppModule` (`app.module.ts`). And then, fix the route for the `finish` to load the `finish` component.

Remember to also downgrade `FinishComponent` using the `downgradeComponent` function (check `app.ts`) and to fix the AngularJS *finish route* to use the new directive:

```
| $routeProvider.when('/finish', { template: '<finish></finish>' });
```

Lastly, remember to delete the `finish` HTML template from the `partials/workout` folder.



If you are stuck in migrating the `finish` page, compare the `ng6-checkpoint8.3` and `ng6-checkpoint8.4` git branches to understand what has changed in the `8.4` branch (<http://bit.ly/ng6be-compare-8-3-8-4>).

The `finish` page was easy, the `start` page is not! While the `start` page seems to be an easy target, there are some challenges that require some head-scratching.

Look at the `start` page template (`partials/workout/start.html`); the biggest issue with the `start` page is that it uses a third-party library, `angular-translate`, to localize the content of the page. Since we are migrating the complete page/view to Angular, we need a mechanism to handle these AngularJS library dependencies.

`angular-translate` comes with a *filter* (*pipe* in the Angular world) and a directive, both named `translate`. Their job is to translate string tokens into localized string literals.

If the `start` page becomes an Angular component, we need to convert the filter into an Angular pipe and, in some way, make the `translate` directive work in Angular.

We have at least these two choices to handle this migration scenario:

- Create a new filter, also upgrade the v1 `translate` directive using `UpgradeModule`
- Find a suitable replacement for `angular-translate` in the Angular world

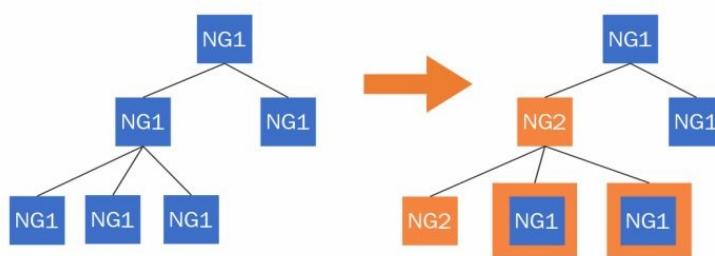
Although the first choice seems to be the easiest, it has some serious limitations. Angular imposes some stringent requirements around how a directive can be upgraded to Angular.



Upgrading an AngularJS directive does not mean the directive has been migrated. Angular instead allows us to use an AngularJS element directive as is inside Angular component views.

AngularJS directive upgrade

At times, the parts of an application may be migrated in a top-down fashion; a higher-order view is converted into a component. In such a case, instead of migrating all the custom directives that are part of the AngularJS view, we just upgrade them to Angular components using some interfaces and functions defined in the `upgradeModule`. The following diagram illustrates this migration path:



The Angular framework puts some restrictions on what can be upgraded to an Angular component. Here is an excerpt from the Angular migration guide.

To be Angular compatible, an AngularJS component directive should configure these attributes:

- `restrict: 'E'`. Components are usually used as elements.
- `scope: {}` - an isolate scope. In Angular, components are always isolated from their surroundings, and you should do this in AngularJS too.
- `bindToController: {}`. Component inputs and outputs should be bound to the controller instead of using the `$scope`.
- `controller` and `controllerAs`. Components have their own controllers.
- `template` or `templateUrl`. Components have their own templates.

Component directives may also use the following attributes:

- `transclude: true/{}}, if the component needs to transclude content from elsewhere`
- `require, if the component needs to communicate with some parent component's controller`

Component directives should not use the following attributes:

- `compile`. This will not be supported in Angular.
- `replace`: True. Angular never replaces a component element with the component template. This attribute is also deprecated in AngularJS.
- `priority` and `terminal`. While AngularJS components may use these, they are not used in Angular and it is better not to write code that relies on them.

The only AngularJS directives that can be upgraded to Angular are element directives, given that all other conditions are met.

With this sizeable laundry list, upgrading an AngularJS directive to Angular is difficult when compared to an Angular component downgrade. More often than not, we have to do an actual code migration of an AngularJS directive if the parent view has been migrated to Angular.

Looking at the `angular-translate` source code, we realize it uses the `$compile` service; therefore, the upgrade option is ruled out. We need to find an alternative library.

We do have an internationalization library for Angular, `ngx-translate` (<http://www.ngx-translate.com/>).

Replacing angular-translate with ngx-translate

ngx-translate is an internationalization library that targets Angular. This library can replace v1 *angular-translate*.

Install the npm package for `ngx-translate` and `ngx-translate/http-loader`:

```
| npm install @ngx-translate/core --save  
| npm install @ngx-translate/http-loader --save
```

The *http-loader* loads the localization files from the server.

Update `systemjs.config.js` to include the *ngx-translate* library. Add entries to the `map` property:

```
var map = {...  
  '@ngx-translate/core': 'npm:@ngx-translate/core/bundles/core.umd.js',  
  '@ngx-translate/http-loader': 'npm:@ngx-translate/http-loader/bundles/http-  
loader.umd.js'  
}
```

As described in the `ngx-translate` documentation, we need to configure the translate module and HTTP loader. Open `app.module.ts` and add the highlighted code:

```
export function HttpLoaderFactory(http: HttpClient) {  
  return new TranslateHttpLoader(http, '/i18n/');  
}  
  
@NgModule({  
  imports: [...  
    HttpClientModule,  
    FormsModule,  
    TranslateModule.forRoot({  
      loader: {  
        provide: TranslateLoader,  
        useFactory: HttpLoaderFactory,  
        deps: [HttpClient]  
      }  
    })  
  ],...
```

The preceding provider declaration sets up a loader that loads the translation files (`.json`) from the `i18n` folder. The `HttpClientModule` import is required for the

translate library to load translations from the server. Remember to copy the translation files (*.json) from git branch `ng6-checkpoint8.4` (<http://bit.ly/ng6be-8-4-i18n>).

Add these import statements to `app.module.ts` to keep the TypeScript compiler happy:

```
import { HttpClient, HttpClientModule } from '@angular/common/http';
import { TranslateModule, TranslateLoader, TranslateService } from '@ngx-translate/core';
import { TranslateHttpLoader } from '@ngx-translate/http-loader';
```

The `ngx-translate` library is now ready to be used. The first thing that we are going to do is set the default translation language as soon as the application bootstraps.

Using ngDoBootstrap for initialization

With Angular, luckily the `AppModule` already defines a function, `ngDoBootstrap`, that the framework calls when bootstrapping an AngularJS application—a perfect place to do `ngx-translate` initialization. Update the `ngDoBootstrap` function in `app.module.ts` with the following code snippet:

```
ngDoBootstrap() {
    this.upgrade.bootstrap(document.documentElement, ['app']);

    var translateService = this.upgrade.injector.get(TranslateService);
    // determine the current locale.
    var userLang = navigator.language.split('-')[0];
    userLang = /(fr|en)/gi.test(userLang) ? userLang : 'en';

    translateService.setDefaultLang('en');
    translateService.use(userLang);
});
```

The code tries to determine the current browser language and sets the current language for translations accordingly. Make note of how we get hold of `TranslateService`. The `upgradeModule` object holds the reference to Angular's *root injector*, which in turn loads `TranslateService` from `ngx-translate`.

With the background work done for the start component, copy the start page implementation from the `ng6-checkpoint8.4` branch (<http://bit.ly/ng6be-8-4-start>) into a new folder `app/js/start`.

Then, add the start component declaration to `app.module.ts`.

Like other components registered as an AngularJS directive before use, add this statement to `app.ts`:

```
import {StartComponent} from './start/start-component';
angular.module('start').directive('start',
  upgradeAdapter.downgradeng6Component(StartComponent) as angular.IDirectiveFactory);
```

The start template file now uses the `translate` pipe (the name of the pipe is the same as the AngularJS filter `translate`).

The start page also has three pipes, `search`, `orderBy`, and `secondsToTime`:

```
|<a *ngFor="let workout of workouts|search:'name':searchContent|orderBy:'name'" ...>
|...
|{{workout.totalWorkoutDuration()|secondsToTime}}
```

Copy the implementation for these pipes together with the definition of the shared module (`shared.module.ts`) from `ng6-checkpoint8.4` (<http://bit.ly/ng6be-8-4-shared>) and add it to the `js/shared` folder locally. Also, remember to import the shared module (`shared.module.ts`) into the app module (`app.module.ts`). We will not dwell on any of the pipe implementation here, as we have done that already in earlier chapters.

Start and finish component implementation is complete. Let's integrate them into the app.

Integrating the start and finish pages

Start/finish views are loaded as part of route change, so we need to fix the route definition in `config.js`. Update the start and finish route definitions to:

```
| $routeProvider.when('/start', { template: '<start></start>' });
| $routeProvider.when('/finish', { template: '<finish></finish>' });
```

The route template HTML is a part of the AngularJS view. Since we have registered both `startComponent` and `FinishComponent` as AngularJS directives, the route loads the correct components.



If you have already migrated the finish page, you do not need to redo the route setup for finish as described.

A few more fixes are pending before we can test the implementation.

Remember to copy the translation files `de.json` and `en.json` from the `ng6-checkpoint8.4` folder `i18n` (<http://bit.ly/ng6-8-4-i18n>). Now, we are ready to test what we have developed.

If not started, start the TypeScript compiler and HTTP-server, and then launch the browser. The start and finish pages should load just fine. But the translations do not work! Clicking on the language translation links on the top nav has no effect. Content always renders in English.



We now are at checkpoint `ng6-checkpoint8.4`. If you are stuck, compare the git branches `ng6-checkpoint8.3` and `ng6-checkpoint8.4` to understand what changed (<http://bit.ly/ng6be-compare-8-3-8-4>).

The translations still do not work because the top nav code (`root.js`) that enables translation is still using the older library. We need to get rid of angular-translate (the v1 library) altogether. Having two libraries doing the same work is not something we want, but removing it is also not that simple.

Getting rid of angular-translate

To get rid of the angular-translate (v1) library, we need to:

- Remove the angular-translate's directive/filter references from all AngularJS views
- Get rid of any code that uses this library

Getting rid of the v1 directive/filter altogether is a difficult task. No can we use the v2 `ngx-translate` pipe in the AngularJS view. Also, migrating every view using the v1 translate directive/filter to Angular at one shot is feasible. There has to be a better solution out there? And there is!

Why not write a new AngularJS filter that uses ngx-translate's translation service (`TranslateService`) for translations and then use the new filter everywhere?

Problems solved!

Let's call this filter `ngxTranslate`. We replace all references to the `translate` filter in the v1 view with `ngxTranslate`. All v1 `translate` directive references too are replaced with an `ngxTranslate` filter.

Here is how the filter implementation looks:

```
import { TranslateService } from '@ngx-translate/core';

export function ngxTranslate(ngxTranslateService: TranslateService) {
  function translate(input) {
    if (input && ngxTranslateService.currentLang) {
      return ngxTranslateService.instant(input);
    }
  }
  translate['$stateful'] = true;
  return translate;
}

ngxTranslate.$inject = ['TranslateService'];
angular.module('app').filter("ngxTranslate", ngxTranslate);
```

Create a file called `ngx-translate.filter.ts` in the `shared` folder and add the preceding implementation. The filter uses `TranslateService` to localized content. To make the service discoverable in AngularJS, it needs to be downgraded using the `ngUpgrade` module method `downgradeInjectable`. Open `app.ts` and add the following

lines:

```
import './shared/ngx-translate.filter';
...
import { downgradeComponent, downgradeInjectable } from '@angular/upgrade/static';
...
import { TranslateService } from '@ngx-translate/core';
...
angular.module('app')
  .factory('TranslateService', downgradeInjectable(TranslateService));
```

This code registers `TranslateService` using a string token, `'TranslateService'`, in AngularJS. The first import statement also loads the new filter at runtime.

To test this implementation, there are a few more steps needed.

To start with, replace all references to `translate` (directive and filter) across the AngularJS view with `ngxTranslate`. There are references in these files: `description-panel.html`, `video-panel.html`, `workout.html` (in the folder `partials/workout`), and `index.html`. Replacing the filter in the interpolation is a simple exercise. For the `translate` directive, replace it with interpolation. For example, in `partials/workout/description-panel.html`, the line of code is as follows:

```
| <h3 class="panel-title" translate>RUNNER.STEPS</h3>
```

It then becomes the following:

```
| <h3 class="panel-title">{{ 'RUNNER.STEPS' | ngxTranslate }}</h3>
```

Remember to quote the string token (`'RUNNER.STEPS'`) inside the interpolation.

Finally, copy the updated `root.js` from <http://bit.ly/ng6be-8-5-root-js>. We have replaced all references to the `$translate` service with `TranslateService` and refactored the code to use the new service. `root.js` contains the implementation for the v1 `RootController`.

We are good to go now. Try out the new implementation; the app should load translation using the *ngx-translate* library.

We can now delete all references to *angular-translate*. There are references in `index.html`, `app-ng1.module.ts`, and `config.js`.

The migration of the start and finish pages is complete.



Compare the branches `ng6-checkpoint8.4` and `ng6-checkpoint8.5` to understand the new changes in `ng6-checkpoint8.5` (<http://bit.ly/ng6be-compare-8-4-8-5>).

We will stop here and direct you to the other GitHub branches pertaining to migration. All branches starting with `ng6-checkpoint*` are the migration branches. Try to migrate the pending views and compare them with the GitHub branch changes. Remember, a working version of the app has already been developed in Angular, and hence there is a good reference point. Look at the `README.md` file for each branch to know what part of the application was migrated to Angular.

Meanwhile, let's summarize our learnings from the migration that we did.

Learning

We hope this migration exercise has provided enough insight into the process. You can now gauge the complexity, the time, and the effort required to migrate elements from AngularJS to Angular. Let's highlight what we have learned as part of this process:

- **Migration is time-consuming:** Migration is by no stretch of the imagination a trivial exercise. Each page/view presents its own challenges that we need to overcome. Some elements are easy to migrate and some are not. The best thing you can do today if you are developing in AngularJS would be to follow the advice from the *Developing AngularJS apps today for easy migration* section.
- **Migrate third-party libraries first:** Migrating third-party libraries can be quite challenging. The reasons are manifold:
 - Such libraries are used across pages
 - They may not be upgradable to Angular (using `upgradeAdapter`)
 - Migrating each view that uses such a library may not be feasible when the library is extensively used

It's better to identify all third-party dependencies in your app and find a suitable alternative for them in the Angular world. If possible, develop some **proof of concept (POC)** with the new library to understand how different the new library is from the existing implementation.

- **Libraries with overlap may exist:** While migrating, there could be scenarios where both AngularJS and Angular versions of a library coexist. Minimize this time period and migrate to the newer version as soon as possible.
- **It is easier to integrate Angular components into AngularJS than the other way round:** While migrating, migrate the complete view to Angular. Due to the restriction imposed by Angular, it becomes very difficult to have a parent Angular component with embedded AngularJS element directives. With such limitations, a bottom-up approach to migrating works better than

a top-down approach.

- **Anything non-UI-related is easy to migrate:** For *Personal Trainer*, we migrate the services last as they can be easily migrated.
- **Feature parity better AngularJS and Angular:** Angular may not have every feature that AngularJS supports. In such a case, we need workarounds to achieve the desired behavior.

That completes our migration story. With this, it's time to conclude the chapter and summarize our lessons from it.

Summary

In this chapter, we gained some useful insight into a number of practical issues surrounding Angular development. These tips/guidelines can be extremely handy when building real-life applications using the framework.

We started the chapter by exploring the concept of *seed projects* and how these projects can get us up and running in no time. We looked at some popular seed projects that can serve as a base for any new Angular app development.

In spite of being a server-side concern, authentication and authorization do affect the client implementation. The section on authentication/authorization covered how to handle authentication in both cookie- and token-based setups.

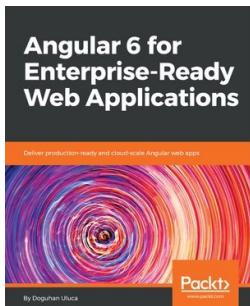
We looked at the ever-so-important topic of performance, where you learned ways to optimize an Angular app's performance.

Finally, we migrated the v1 *Personal Trainer* to Angular. The gradual migration process taught us the intricacies of migration, the challenges faced, and the workaround done.

The book is coming to a close, but for everyone reading it, the journey has just begun. It's time to put theories into practice, hone our newly acquired skills, build something useful with Angular, and share it with the world. The more you invest in Angular, the more rewarding the framework is. Let's get started!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

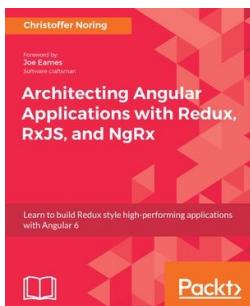


Angular 6 for Enterprise-Ready Web Applications

Doguhan Uluca

ISBN: 9781786462909

- Create full-stack web applications using Angular and RESTful APIs
- Master Angular fundamentals, RxJS, CLI tools, unit testing, GitHub, and Docker
- Design and architect responsive, secure and scalable apps to deploy on AWS
- Adopt a minimalist, value-first approach to delivering your app with Kanban
- Get introduced to automated testing with continuous integration on CircleCI
- Optimize Nginx and Node.js web servers with load testing tools



Architecting Angular Applications with Redux, RxJS, and NgRx

Christoffer Noring

ISBN: 9781787122406

- Understand the one-way data flow and Flux pattern
- Work with functional programming and asynchronous data streams
- Figure out how RxJS can help us address the flaws in promises
- Set up different versions of cascading calls
- Explore advanced operators
- Get familiar with the Redux pattern and its principles
- Test and debug different features of your application
- Build your own lightweight app using Flux, Redux, and NgRx

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!