

Angular 2

Succinctly[®]

by Joseph D. Booth

Angular 2 Succinctly

By
Joseph D. Booth

Foreword by Daniel Jebaraj



Copyright © 2017 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the Succinctly Series of Books	12
About the Author	14
Chapter 1 Introduction.....	15
CSS styling.....	15
JavaScript coding	15
Third-party libraries	16
The good	16
The bad	16
Why AngularJS?.....	16
Angular 2.....	17
ECMAScript 6	17
TypeScript	17
Dependency injection	18
Web components	18
Without web components.....	18
Web components.....	19
Summary.....	22
Chapter 2 Dev Environment	23
Language	23
Editor.....	23
Node Package Manager (NPM).....	24
Installing NPM	24
Confirming NPM	25

Installing packages	25
Summary	25
Chapter 3 Angular CLI	26
Getting Angular CLI	26
Creating a new project	27
Project root	27
ng serve	28
ng build	28
Environments folder	29
Summary	29
Chapter 4 Files and Folders	30
Folder structure	30
GitHub files	31
.gitignore	31
README.md	31
.editorconfig	31
angular-cli.json	32
package.json	32
Package information	32
Scripts	33
License	33
Dependencies	33
devDependencies	34
tsLint.json	35
src folder	35
tsConfig.json	35

typings.json.....	36
Summary.....	38
Chapter 5 Customization	39
Adding libraries.....	39
angular-cli.json	39
Font Awesome.....	40
Assets	40
Environments	40
Summary.....	41
Chapter 6 Your Environment.....	42
Your folder structure	42
Summary.....	42
Chapter 7 Exploring Hello World	43
Start Angular CLI.....	43
Import	43
@Component	44
Export statement.....	44
Modules	44
Imports array	45
Declarations array.....	45
Bootstrap array	45
Our main program	45
Index.html.....	46
Style sheets	47
Body	47
Styles.css	48

Summary.....	49
Chapter 8 Tweaking It a Bit	50
ng serve	50
Summary.....	50
Chapter 9 Components.....	51
Component files	51
Import	51
Metadata.....	52
Class code.....	54
Summary.....	58
Chapter 10 Templates.....	59
Template declaration	59
HTML	59
Interpolation	60
Expressions.....	60
Pipe operator	60
Custom pipes.....	61
Template statements.....	63
Displaying data in templates.....	64
Arrays	64
Interfaces.....	65
Classes.....	66
Conditions.....	66
Switch statement	67
Looping.....	67
Summary.....	68

Chapter 11 Modules	69
Basic module options	69
declarations	69
imports	70
providers	70
exports	71
bootstrap	71
app.module.ts	71
main.ts	72
Summary	72
Chapter 12 Our Application	73
Screen mockups	73
Standings page	73
Scorekeeper's page	74
Summary	74
Chapter 13 Menu Navigation	75
Base href	75
App component	76
Views folder	76
Main menu	76
Placeholder components	77
Route definitions	78
path	78
Component	78
app.route.ts	78
App module	80

Page not found	80
Navigation	81
Summary.....	82
Chapter 14 Services and Interfaces	83
Standings page	83
Data model.....	84
Database design.....	84
Service design.....	85
Interfaces.....	85
Service code.....	86
Getting the data	86
Injectable.....	87
Consuming the service	89
Importing the service	89
Adding the provider metadata	90
Update the constructor.....	90
The modified.....	92
Summary.....	93
Chapter 15 Standings	94
Standings component.....	94
Template page.....	95
Class code.....	96
Standings display	100
Summary.....	101
Chapter 16 Editing Data.....	102
Data binding	102

Property binding.....	102
Attribute binding.....	102
Event binding.....	103
Class binding.....	103
Style binding.....	104
One-way binding summary.....	104
Two-way binding.....	105
ngModel.....	105
Summary.....	106
Chapter 17 Scoring.....	107
Scoring component.....	107
Scoring template.....	108
Class code.....	109
Summary.....	112
Chapter 18 Getting HTTP Data.....	113
Web services.....	113
JSON test website.....	113
JSON format.....	113
Web commands.....	114
Angular HTTP.....	115
Root module.....	115
Web service.....	115
Creating the web service.....	116
Using the service.....	117
Passing parameters.....	118
Post method.....	118

Error handling	119
Summary.....	120
Chapter 19 Summary	121
Appendix 1 Component Metadata.....	122
Appendix 2 Template Syntax	123

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Joseph D. Booth has been programming since 1981 in a variety of languages, including BASIC, Clipper, FoxPro, Delphi, Classic ASP, Visual Basic, Visual C#, and the .NET framework. He has also worked in various database platforms, including DBASE, Paradox, Oracle, and SQL Server.

He is the author of *GitHub Succinctly*, *Accounting Succinctly*, *Regular Expressions Succinctly*, and *Visual Studio Add-Ins Succinctly* from Syncfusion, as well as six books on Clipper and FoxPro programming, network programming, and client/server development with Delphi. He has also written several third-party developer tools, including CLIPWKS, which allows developers to programmatically create and read native Lotus and Excel spreadsheet files from Clipper applications.

Joe has worked for a number of companies including Sperry Univac, MCI-WorldCom, Ronin, Harris Interactive, Thomas Jefferson University, People Metrics, and Investor Force. He is one of the primary authors of Results for Research (market research software), PEPSys (industrial distribution software), and a key contributor to AccuBuild (accounting software for the construction industry).

He has a background in accounting, having worked as a controller for several years in the industrial distribution field, but his real passion is computer programming.

In his spare time, Joe is an avid tennis player and a crazy soccer player (he plays goalie). He also practices yoga and martial arts, and holds a brown belt in Judo. You can visit his website [here](#).

Chapter 1 Introduction

Imagine, if you will, doing web development in an environment where you can control every aspect of the page you are building, and customize it to your heart's content. Don't like the white background on your page? You can change it easily using CSS. Want to react when your user moves into a field? You can hook into that event. It is a powerful environment—you develop your content, and can leave its presentation as is, or customize it as much as desired.

CSS styling

For example, the following line of CSS sets your document to an **antique white** background with a **navy** font color.

Code Listing 1

```
body {  
    background-color: antiquewhite;  
    color: navy;  
}
```

It's powerful, quick, and easy!

JavaScript coding

You can hook into the event system just as easily by attaching event handlers to your elements.

Code Listing 2

```
<button id="btn" onclick="ShowBtn();">Hello World!</button>
```

Then, you add code to perform whatever function you would like.

Code Listing 3

```
function ShowBtn() {  
    theBtn = document.getElementById("btn").textContent;  
    alert(theBtn);  
}
```

Third-party libraries

Once you start working in this environment, you begin to realize there are subtle differences between CSS and JavaScript, depending on the browser the user is running. One approach is to attempt to code around these differences, but that adds more complexity to this environment. Fortunately, there are third-party libraries that hide that complexity from you by handling the browser differences internally.

So, to keep focused on your application, rather than browser differences, you come to rely on third-party software libraries like jQuery, Knockout, and Kendo UI.

The good

These third-party libraries hide this complexity and provide great functionality to make development much easier. For example, we can use jQuery's `on()` function to hide the browser differences in event handling.

The bad

However, these third-party libraries have access to the very same elements on the page that you do. So your nice page with the antique white background might be changed to a dark color background because a library replaced your CSS class name with its own. Oh, and that JavaScript code you wrote when the button gets clicked? Well, it is possible a third-party library likes the function name `ShowBtn()`, so its method gets called instead of yours.

There are workarounds that most web developers are familiar with, such as loading libraries in a particular order, adding qualifier to all your class names and function calls, loading your CSS and JavaScript last, etc. Some even go as far as not relying on third-party libraries.

Why AngularJS?

AngularJS is an open-source JavaScript framework developed and maintained by Google, Inc. and several open-source contributors. It was originally released in 2009, primarily aimed at making the HTML syntax more capable for application development. It included concepts such as data binding and HTML templates. In its simplest form, an Angular application would consist of an HTML page (with imbedded “variables”) and a JavaScript object (called a controller) with properties and methods. The developer would manipulate the controller properties, and Angular would update the DOM (HTML page) to reflect the changing values.

Angular 2 was released in 2014 and rewritten to take advantage of new features that allow the development of components. It is a different approach to front-end development; instead of building a page and hoping your third-party libraries don't conflict with your own code, you build components that will work the way you'd expect, and then use Angular to display the components to your user.

As we work through the book, we will develop components and use existing components to build our webpages. As Angular 2 continues to develop, you will see many third-party components that can be used in an Angular application. This will allow us to do front-end web development much like other environments, selecting the components we want to use and tying them together into an integrated product.

Angular 2

Angular 2 is a complete rewrite of the Angular library, and is not backwards compatible with Angular 1 applications. This caused some concern among developers, but the Angular team wanted to take advantage of many new features that were not around in 2009. Angular 2 is about making the most of the new browser developments to move forward and create better applications.

Some of the new web features that Angular 2 embraces are:

ECMAScript 6

ECMAScript is a scripting language specification standardized by ECMA International. JavaScript is one of the most popular implementations of ECMAScript for client-side web applications. It was first published in 1997, and has grown over the years. The most recent version (ES6) adds substantial syntax improvements for writing complex scripting applications. While not every browser supports all the new features, ECMAScript 6 is the future of JavaScript.



Note: You can read about ECMA6 in Matthew Duffield's book [ECMAScript 6 Succinctly](#), available from Syncfusion.

TypeScript

TypeScript is a superset of JavaScript developed and maintained by Microsoft. It adds features to JavaScript, most notably data types for variables. It also adds many of the features in ECMAScript 2015 (the scripting language most current browsers support). Angular 2 itself is written in TypeScript.

TypeScript files (**.ts** extension) are transpiled (meaning source code is converted to another source language) into JavaScript files (**.js** extension). This allows developers to use the features of TypeScript and still have browsers support and run the script.

We will use TypeScript in this book's examples, but it's okay if you are not familiar with the language. Knowing JavaScript and any object-oriented language (such as C#) will allow you to feel right at home with TypeScript. (Plus, Syncfusion has [a book in the Succinctly series](#) to help you learn it if needed.)

Dependency injection

Dependency injection is a software design pattern that attempts to reduce tightly coupled components by changing how component dependencies are handled. For example, if a logging component needs to notify users when something is amiss, it might be tempting to access another component (**NotifyUsersByEmail**, perhaps) within the logging component. While coding is a bit simpler, it creates a dependent relationship between the logging and the **NotifyUsersByEmail** component.

These dependencies make it difficult to test and debug components individually. For example, if you expect an event to be logged and users to be notified but it doesn't happen, you need to determine if the mail component failed or the log component failed. Also, if the mail component changes, then the logging component must be updated to accommodate the changes.

A common approach to solving this is to write an interface that describes how the logging component plans to interact with the notification tasks it needs. Namely, it defines how to specify users, the subject and message, and how to invoke the **send** method. The logging component doesn't know the details, just that it can access another object that matches the interface. The object itself gets passed into the component (often from the constructor) for the base component to use. The logging object doesn't care about the particulars of the object, only that it has the agreed upon fields and methods.

Such an approach makes testing easier, and accommodates changes as well. For example, if we wrote a **NotifyUsersBySMS** object, as long as it offers the same methods as the **Logging** component expects, we can change the notification behavior simply by passing in a different component to the constructor.

Web components

Web components were first introduced in 2011, although components were a part of software development for many years prior to that. The standards necessary to implement web components are being worked on by the W3C, and they represent the future of web application development.

Without web components

When you develop front-end code, you are generally using some JavaScript framework, and possibly some CSS, sprinkling it through your HTML, and hoping that some subsequent CSS or JavaScript files don't come along and change all the code you've written.

Recently, while working on a website, I found a JavaScript library that looked like it provided a solution to an issue. I added the library and the CSS files to my layout page, and got the functionality I wanted, with one nasty little side effect: the CSS changed all my `<p>` tags to be **text-align: center**.

Front-end development becomes a matter of carefully combining JavaScript and CSS libraries that hopefully don't conflict with each other and produce the functionality we are trying to implement.

Web components

Web components are essentially fully encapsulated HTML elements that the browser knows how to display. Since the HTML and CSS are encapsulated in the component, the component will always display the way it was designed, even if some later-loaded CSS style sheet changes the presentation rules on HTML elements.

When you create an HTML page, you are creating the Document Object Model (DOM). The DOM is a representation of your HTML (or XML, etc.) source code as a nested tree structure. Browsers use various layout engines (such as WebKit or Gecko) to handle parsing the HTML into a DOM. Once the DOM is built, JavaScript and CSS can manipulate the DOM. If you've worked with jQuery or CSS, you've certainly seen selectors such as # (ID) or . (class) to get particular DOM elements.

Shadow DOM

The Shadow DOM is an encapsulated DOM object that can be created from any existing DOM element. The DOM element that creates the Shadow DOM is known as the **ShadowHost**. The new element is referred to as the **ShadowRoot**. The following JavaScript fragment shows how to create a Shadow DOM element.

Code Listing 4

```
<script>
  var ShadowHost = document.querySelector('button');
  var ShadowRoot = ShadowHost.createShadowRoot();
  ShadowRoot.innerHTML="Hello from Angular 2";
</script>
```

Markup in the Shadow Root is not visible to the scripts outside of the Shadow DOM. The purpose of the Shadow DOM is to provide an encapsulated snippet, safe from prying eyes.

If you explore the following example in Code Listing 5, you see that the HTML element contains **Hello, world**. However, the script creates a **ShadowRoot** (if it does not yet exist) and sets the **innerHTML** to "Hello from Angular 2". When the browser executes this code, the button's inner content gets replaced with whatever content is in the **ShadowRoot**.

Code Listing 5

```
<body>
  <button id="btn" onclick="ShowBtn();" >Hello, world!</button>
  <script>
    var ShadowHost = document.querySelector('button');
    if (ShadowHost.shadowRoot==null)
    {
      var ShadowRoot = ShadowHost.createShadowRoot();
      ShadowRoot.innerHTML="Hello from Angular 2";
    }
  </script>
```

```

    <script>
      function ShowBtn() {
        theBtn = document.getElementById("btn").innerHTML;
        alert(theBtn);
      }
    </script>
  </body>

```

However, when you click the button, the **ShowBtn()** function shows the content **Hello, world**, not the content that the browser displayed from the **ShadowRoot**. This is an example of the encapsulation and scoping necessary to build web components.



Note: Some browsers expose parts of the Shadow DOM to CSS through special pseudo-selectors. Also, component authors can choose to expose some content, particularly theming and styling, if they want to.

Template tag

Another piece of the component puzzle is the HTML **<template>** tag. This tag allows us to build HTML fragments for later use. Content inside a template tag will not display, and is not active (images won't be downloaded, scripts won't run, etc.).

Templates can contain HTML, CSS style sheets, and even JavaScript. They can be placed anywhere with the DOM, and stay inactive until you need them. The following is a simple template that draws a border and shadow around an **<h3>** element and adds some text.

Code Listing 6

```

<template>
  <style>
    h3 {
      color: darkblue;
      border: 2px solid gray;
      box-shadow: 10px 10px 5px #0f0f0f;
      width: 20%;
      margin-left: 20px;
      padding-left: 10px;
    }
  </style>
  <h3>From the Shadows... </h3>
</template>

```

Now, when we create our Shadow Root, rather than rely on manipulating the **innerHTML** property, we will plug the template code in. And the **<h3>** style within our template, since it is in the Shadow Root, will not impact any other **<h3>** styling the page might be using.

Code Listing 7

```
var ShadowHost = document.getElementById('HostDiv');
if (ShadowHost.shadowRoot==null) // See if the element has a shadow root?
{
    var ShadowRoot = ShadowHost.createShadowRoot();
    // Get the template
    var tpl1 = document.querySelector('template');
    ShadowRoot.appendChild(document.importNode(tpl1.content, true));
}
```

We grab the template element (we can use `getElementById()` if we have a number of different templates) and append it into the Shadow Root element. Our screen will now show the following in the **HostDiv** element.

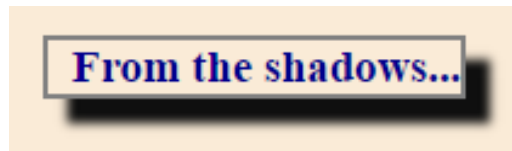


Figure 1: Shadow Root

Getting content from the host

Often, our template code will want to get its content from the host element, rather than hard-coding the content in the template code. For example, if our Shadow Host element looked like the following:

Code Listing 8

```
<div id="HostDiv"><span>The DIV</span></div>
```

We could replace our template's HTML line:

Code Listing 9

```
<h3>From the Shadows... </h3>
```

with the following fragment:

Code Listing 10

```
<h3>Courtesy of the shadow host <content select="span"></content></h3>
```

When the template is inserted, it will grab the content of the Shadow Host's `` tag and use it instead.



Courtesy of the shadow host The DIV

Figure 2: Content from Shadow Host

Taken together, the Shadow DOM and the template system open the door to allow components within the front-end development of webpages. And that is one of the biggest benefits Angular 2 provides.

Summary

Angular 2 takes advantage of advantage of web components and the Shadow DOM to support component-driven development. In the rest of the book, we will use the Angular 2 framework to create a component-driven application, and perhaps along the way, create some reusable components for other applications.



Note: The code samples in this book are available for download at <https://bitbucket.org/syncfusiontech/angular-2-succinctly>.

Chapter 2 Dev Environment

To work with Angular 2 and start developing your own component-driven websites, you will need to make some decisions and create an environment for building, compiling, and running your application. In this chapter, we will set that environment up and get ready to build our first application.

Language

Angular 2 lets us build client applications in HTML and CSS, and either JavaScript or a script language that compiles to JavaScript. We are going to use **TypeScript** rather than JavaScript for working with Angular. Don't worry if you are not familiar with TypeScript, if you have used JavaScript and object-oriented programming before, you should be able to pick up TypeScript very quickly.



Note: You can download the book [*TypeScript Succinctly*](#) by Steve Fenton from the [Synconfusion website](#).

You can use JavaScript for Angular 2; however, Angular 2 itself is written in TypeScript, and most examples will use the TypeScript language. Since TypeScript is “strongly-typed JavaScript” (and more), any editor that supports TypeScript will be able to use IntelliSense features to make your development work much easier.

Editor

You can use any editor you would like to develop Angular 2 applications, but your programming will work much easier if you choose an editor that supports TypeScript. For the purposes of this book, we are going to use Microsoft's Visual Studio Code. You can download Visual Studio Code [here](#) (it's free).

If you are not familiar with Visual Studio Code, it is an editor and development tool that operates similarly to Visual Studio, but is more geared toward managing files and folders than solutions and projects. An Angular 2 application works in folders and files, so Visual Studio Code can work very well with Angular 2 applications.



Note: You can download the book [*Visual Studio Code Succinctly*](#) by Alessandro Del Sole from the [Synconfusion website](#).

Node Package Manager (NPM)

Angular 2 is a JavaScript framework that will add several files and folders into your application folder. The best way to manage these files and folders is to let another application do it for you. The Node Package Manager is that program. It is a prerequisite to using Angular 2 for development.

Installing NPM

To install the package manager, visit the [Node.js download page](#).

You will see the following page (although its appearance may change over time).

node

HOME | ABOUT | DOWNLOADS | DOCS | FOUNDATION | GET INVOLVED | SECURITY | NEWS

Downloads

Latest LTS Version: **v6.9.2** (includes npm 3.10.9)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS
Recommended For Most Users
Current
Latest Features

Windows Installer
node-v6.9.2-x64.msi

Macintosh Installer
node-v6.9.2.pkg

Source Code
node-v6.9.2.tar.gz

Windows Installer (.msi)	32-bit	64-bit	
Windows Binary (.exe)	32-bit	64-bit	
macOS Installer (.pkg)	64-bit		
macOS Binaries (.tar.gz)	64-bit		
Linux Binaries (x86/x64)	32-bit	64-bit	
Linux Binaries (ARM)	ARMv6	ARMv7	ARMv8
Source Code	node-v6.9.2.tar.gz		

Figure 3: NPM Install Page

Download and install the version that is right for your machine.

NPM is a console-based program that we will use to install the Angular 2 framework, as well as other useful features. Visual Studio Code allows you run a command window (use the **View > Integrated Terminal** menu) from within Visual Code, or you can open your own console window and run the necessary commands from there.

You can always update to the most current version of NPM by using the following command at a command prompt window.

```
npm install npm@latest -g
```

This will install the most current version for use globally on your computer.



Note: There is much more to Node and NPM than what we need for Angular 2. JavaScript packages can be managed using NPM, and at the time of this writing, there are over 300,000 packages accessible through NPM. Be sure to visit [the NPM website](#) if you want to learn more.

Confirming NPM

You should open a console window and confirm that NPM was installed properly. You can do this by checking the version, as shown in Figure 4 (your version may be different).

```
E:\Joe\SD>npm -v
4.0.3
E:\Joe\SD>_
```

Figure 4: NPM Version

Installing packages

When we create our Angular 2 application folders, one of the files will be a JSON configuration file that NPM will use to manage our package dependencies. Here is a sample.

```
1. "dependencies": {
2.   "@angular/common": "~2.3.0",
3.   "@angular/compiler": "~2.3.0",
4.   "@angular/core": "~2.3.0",
5.   "@angular/forms": "~2.3.0",
6.   "@angular/http": "~2.3.0",
7.   "@angular/platform-browser": "~2.3.0",
8.   "@angular/platform-browser-dynamic": "~2.3.0",
9.   "@angular/router": "~3.31.0",
10.  "@angular/upgrade": "~2.3.0",
```

We will cover this more in the next chapter. One key point is that the version number is specified for the dependencies, allowing you to keep your production environment safely at one version, while possibly exploring new features in a later version.

Summary

Now that your development environment is built, you are ready to put together your first Angular 2 application.

Chapter 3 Angular CLI

Angular CLI (command line interface) is a new tool that was developed to make it easier to create and deploy Angular applications. Much of the setup and configuration work you need to do can be done through the [Angular CLI](#). Deploying applications is much simpler using the CLI tool.



Note: At the time this book was written, Angular CLI was still in beta status, but it represents an exciting time saving tool for the Angular environment. Be sure to visit the website to keep track of the application.

Getting Angular CLI

To use Angular CLI, you need to have the latest versions of node and NPM. You can check the versions at the command line with the following lines:

- `npm --version`
- `node -v`

NPM must be version 3 or higher, and Node must be version 4 or higher. If you don't have these versions, you can update NPM using NPM itself. The command is:

Code Listing 11

```
npm install npm@latest -g
```

If you need to make an update, the easiest and safest approach is to visit the [Node.js website](#) and download the latest version. Once you have these versions, to install Angular CLI, simply run the following command from a command prompt.

Code Listing 12

```
npm install -g angular-cli
```

This will install **angular-cli**, and you can use the **ng** command to run it.

Code Listing 13

```
ng help
```

This will provide a list of all the commands and their parameters. For this chapter, we are focusing on the **new** and **build** commands.



Note: If you want to update Angular CLI to the latest version, you will need to **uninstall the current version and get the latest version, as shown by the following commands.**

```
npm uninstall -g angular-cli  
npm cache clean  
npm install -g angular-cli@latest
```

Creating a new project

Once you've installed Angular CLI, you can create a new project using the **new** command. Its syntax is:

Code Listing 14

```
ng new <projectName>
```

This will create a folder with the project name and place the needed files into the folder. The **package.json** file will be generated automatically for you, with the latest version of the Angular modules. The installation process will be performed as part of the creation, so the **node_modules** folder will be populated for you.



Note: The **ng new** command performs two tasks: it first creates a folder and then creates the project files. The **ng init** command is similar, except it creates the project files in an existing folder.

Once this step is complete, you have a working application, including the needed configuration files. The application itself will reside with the **src** folder, with the familiar **app** folder within in.

Project root

The project root folder will hold several configuration files. The **package.json** file (we will cover this in the next chapter), a configuration file for Angular CLI (**angular-cli.json**), and one for the lint checker (**tslint.json**). You'll also find a GitHub readme file (**README.md**).

src

In the **src** folder, you'll find the **index.html** file and a CSS file, as well as some other files for testing, a favicon file, and more. The configuration file for TypeScript (**tsconfig.json**) will be here as well.

There is a folder within the **src** called **environments**. This folder contains a pair of TypeScript components that export a production flag, so you can do debug and production builds.

src\app

Within the **src\app** folder, you'll find the base module and component. You will likely be adjusting **app.module.ts** and **app.component.ts** to be your application starting points. The base ones are simply an Angular version of **Hello World**.

ng serve

You can run the **ng serve** command to start a web server on local host, port 4200. If you run **ng serve** and open a browser to <http://localhost:4200>, you will see the basic application display the message “app works!”

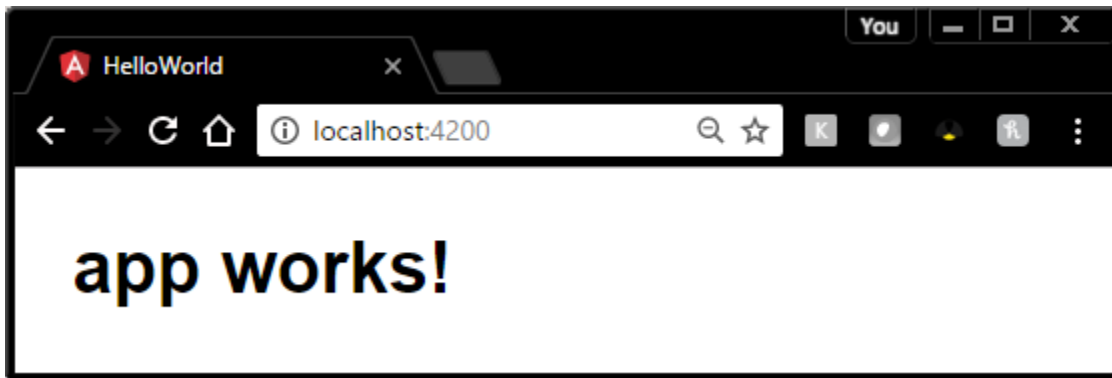


Figure 5: Angular CLI Hello World



Tip: The scripts and styles are bundled and can be found in the **dist** folder.

ng build

The **ng build** command is used to create a set of distribution files in the output folder (usually **dist**). You can do a dev build (the default) by simply running the following command.

Code Listing 15

```
ng build
```

The **dist** folder will contain the bundles and any required assets for running the application. To do a production build (which will result in substantially smaller bundle files), you can use the following command.

Code Listing 16

```
ng build -prod
```

Environments folder

The **environments** folder contains Angular modules that can be loaded, depending on the type of build.

Code Listing 17: Environment.prod.ts

```
export const environment = {  
  production: true  
};
```

You can import this file into your component with the following code.

Code Listing 18

```
import { environment } from '../environments/environment';
```

This will give you access to the properties in the environment files, which will allow you to adjust your code if need be, based on whether it is a development or production build.

Summary

Angular CLI is still a beta production, but it represents a new and very helpful application to use in your Angular development. We touched upon some of the basics, but the CLI is powerful and worth exploring more. It includes support for unit tests, and it can generate starting components of any Angular type.

You should periodically visit the [official Angular CLI website](https://cli.angular.io/) to see the growth and enhancements the tools will provide. It is not necessary to use the Angular CLI, but as you get familiar with Angular 2, you will come to appreciate the CLI to make your overall Angular development work easier.

Chapter 4 Files and Folders

When you set up an Angular 2 application using Angular CLI, it populates a folder for your development work. This folder includes the **node_modules** (libraries your application might need), the source code to your application, and some testing tools. In this chapter, we explore these folders and files.

Folder structure

After project creation, Angular 2 applications will have a root folder (for configuration files) and several other common folders:

- **src**: The main folder for your application's source code.
- **node_modules**: The folder where libraries are stored.
- **e2e**: The end-to-end testing folder.

Here is a look at the structure within Visual Studio Code.

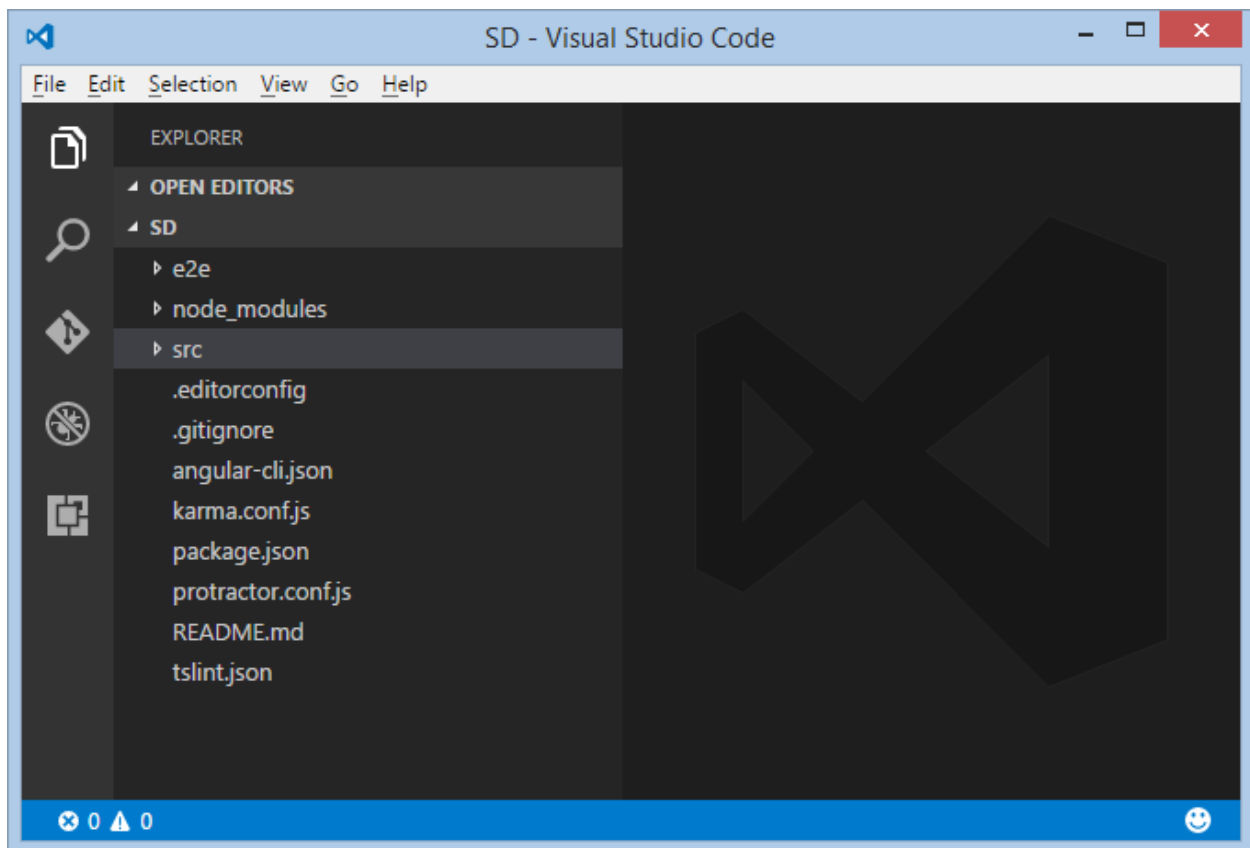


Figure 6: Folder Structure

GitHub files

The default application contains a couple files needed by Git and GitHub. These are the **.gitignore** and **README.md** files. If you are interested in learning more about GitHub, you can download my book [GitHub Succinctly](#) from the Syncfusion website.

.gitignore

The **.gitignore** file is the standard file instructing GitHub and Git which files in a folder should be excluded (or ignored) from Git version control. You can specify files or folders not to check into the repository, such as:

- **/dist**: Ignore files in the distribution folder.
- **.map**: Don't commit any map files.
- **/node_modules**: Don't commit the various Angular or other libraries.

Lines beginning with a **#** are comment lines.

README.md

GitHub projects contain a **README.md** file that is displayed as part of the GitHub listing. The Angular CLI will include a readme file for you when creating a new project.

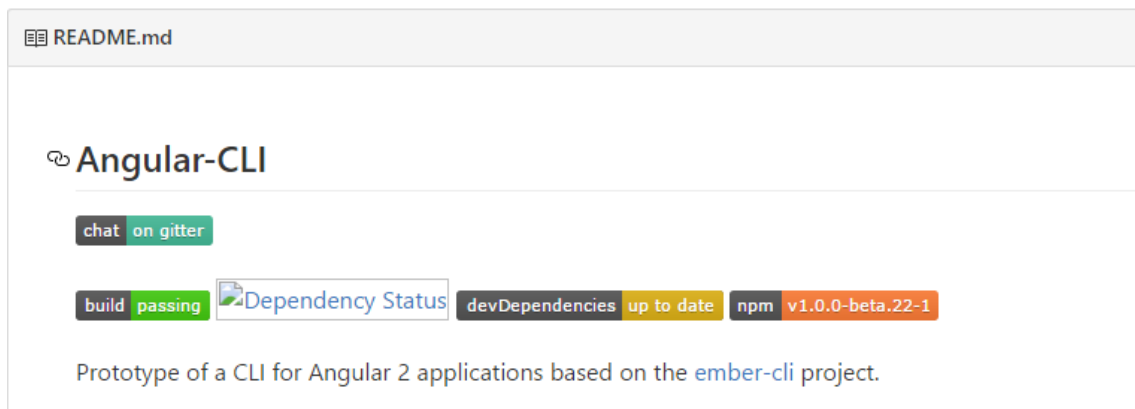


Figure 7: Sample GitHub Readme

If you are not using GitHub, you can ignore these files, but having them generated for you is a nice time-saver.

.editorconfig

This file allows you to provide configuration settings for your editor. You can set various properties, such as **indent_style** (tab or space) and **indent_size** (number). Visual Studio Code doesn't natively support **.editorconfig**, but you can install a plugin that supports it by pressing **Ctrl+Shift+P** and entering the following command.

```
ext install .EditorConfig
```



Tip: If you try to create a file beginning with a period, Windows will view it as a file extension and ask for a file name. However, if you put a period at the end as well, Windows will create the file.

angular-cli.json

This file allows you to configure the behavior of the Angular CLI tool. There are various sections, such as project, apps, and defaults. The apps group contains most of the options you might need to customize. Here are a few of the settings you might use:

- **apps/root:** Folder where the root application is, using **src**.
- **apps/outDir:** Folder to hold the distribution, typically **dist**.
- **apps/assets:** Files and folders that should be included with distribution.
- **apps/styles:** Collection of style sheets to bundle into the application.
- **apps/scripts:** Collection of JavaScript files to bundle.

package.json

This file tells NPM which dependencies are needed for your application. While you might not need them all, there is no harm in adding more packages than you use. Your client application will only get the packages you actually use in your code, not the entire list from this file. Since this is a JSON file, the elements described here will be between { } in the actual file.

Package information

This section contains information about the package, such as its name and version.

Code Listing 20

```
"name": "helloworld",
"version": "1.0.0",
"description": "First Angular-2 application",
"author": "Joe Booth",
```

The **name** attribute is required, and must be lowercase. The **version**, **description**, and **author** attributes are optional, but should be provided in case another developer wants to find out information about your application.

Scripts

The **scripts** section is used to provide commands to the NPM command-line application.

Code Listing 21

```
"scripts": {  
  "start": "ng serve",  
  "lint": "tslint \"src/**/*.ts\"",  
  "test": "ng test",  
  "pree2e": "webdriver-manager update ",  
  "e2e": "protractor"  
},
```

When you run the NPM **start** command to launch your application, you are actually running the Angular 2 **serve** command, which *trans-compiles* the TypeScript files and opens the server on the local host. You can add additional scripts, perhaps to compress your JavaScript files, for example.

License

The purpose of the **package.json** file is to provide information about your application, and if the package is open source, you can specify the license on the package, such as MIT. You can also set the license to **UNLICENSED** and add the **private** attribute (set to **true**). However, once you've completed your package, you can publish it to the Node repository for others if you would like. For now, we will keep our project private with the following attributes.

Code Listing 22

```
"license": "UNLICENSED",  
"private": true,
```



Note: There are a number of additional attributes available for describing the package and adding keywords to make it available as open source to other developers. If you decide to make your package available to the public, explore attributes like *repository* (location where source is), *keywords* (to make searching for it easier), and *bugs* (email address where to report any bugs).

Dependencies

The **dependencies** section lists all the modules that your package relies on, and will be automatically added when someone installs your package. The list of dependencies you might need for an Angular 2 application is shown in the following code listing.

Code Listing 23

```
"dependencies": {
```

```

"@angular/common": "2.2.3",           // Common service, pipes, directives
"@angular/compiler": "2.2.3",         // Template compiler
"@angular/core": "2.2.3",             // Critical runtime parts
"@angular/forms": "2.2.3",
"@angular/http": "2.2.3",             // Angular HTTP client
"@angular/platform-browser": "2.2.3",
"@angular/platform-browser-dynamic": "2.2.3",
"@angular/router": "3.2.3",           // Component router

"core-js": "^2.4.1",
"rxjs": "5.0.0-beta.12",
"zone.js": "0.6.23",
"ts-helpers": "1.1.1"
},

```

When we run the `install` command, these dependencies will be added to your application folder from the Node repository. You can use the list in Code Listing 23 as a good starting point, but you might start adjusting this as you work more with Angular 2 applications. Comments are actually not supported in a JSON file; the comments in this example are just for indicating what the various modules are for. The configuration files available on the [SynCFusion Bitbucket page](#) do not have comments.



Note: *At the time of this writing, Angular 2 was on version 2.2.3. You might need to update your dependencies as future versions of Angular are released.*

devDependencies

This section lists the dependencies that are only needed for development and testing your application. When doing a production release, these dependencies will not be added.

Code Listing 24

```

"devDependencies":
{
  "@angular/compiler-cli": "2.2.3",
  "@types/jasmine": "2.5.38",
  "@types/node": "^6.0.42",
  "angular-cli": "1.0.0-beta.22-1",
  "codelyzer": "~2.0.0-beta.1",
  "jasmine-core": "2.5.2",
  "jasmine-spec-reporter": "2.5.0",
  "karma": "1.2.0",
  "karma-chrome-launcher": "^2.0.0",
  "karma-cli": "^1.0.1",
  "karma-jasmine": "^1.0.2",
  "karma-remap-istanbul": "^0.2.1",

```

```
"protractor": "4.0.9",  
"ts-node": "1.2.1",  
"tslint": "^4.0.2",  
"typescript": "~2.0.3",  
"webdriver-manager": "10.2.5"  
  
}
```

The **package.json** file interacts with NPM to install dependencies. Your version numbers might be different from the examples in Code Listing 24, but by keeping this configuration file with the project, you can ensure current production projects stay stable on the same versions, while possibly exploring updated versions for later applications.

tsLint.json

This configuration file lets you set up the options for the **codelyzer** lint checker. A lint checker is a nice additional tool (installed as part of Angular CLI), but sometimes your coding style could cause unnecessary warnings. These warnings can clutter your lint checking, and make it hard to see the beneficial warnings. By customizing the file, you can adapt the lint checker to work with your coding style.

For example, the **any** type in TypeScript is allowed, but somewhat defeats the purpose of a strongly typed language. You can control whether to report usage of **any** with the following entry in the configuration file.

Code Listing 25

```
"no-any": true
```



Tip: *Lint checkers are handy tools for reducing the likelihood of certain errors that will compile, but they may cause problems at run-time. Every rule checked generally has a good reason for it, so be sure to decide whether the rule is actually a nuisance or if you should rethink your coding style.*

src folder

The **src** folder contains the configuration file for the TypeScript compiler and the type specifications.

tsConfig.json

The **tsConfig.json** file contains the compiler options for TypeScript. It allows us to specify how we want the **.ts** files to be transpiled into JavaScript files. The options we use for our default are shown in the following code listing.

Code Listing 26

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  }
}
```

You do not have to have a **tsConfig.json** file, but it is included to get the exact behavior we want, including when we want ES5 (ECMAScript 5) rather than the default ECMAScript 3. We've also chosen to keep comments in the trans-compiled JavaScript code. You can read [Steve Fenton's book on TypeScript](#) if you want more details on the compiler options.

typings.json

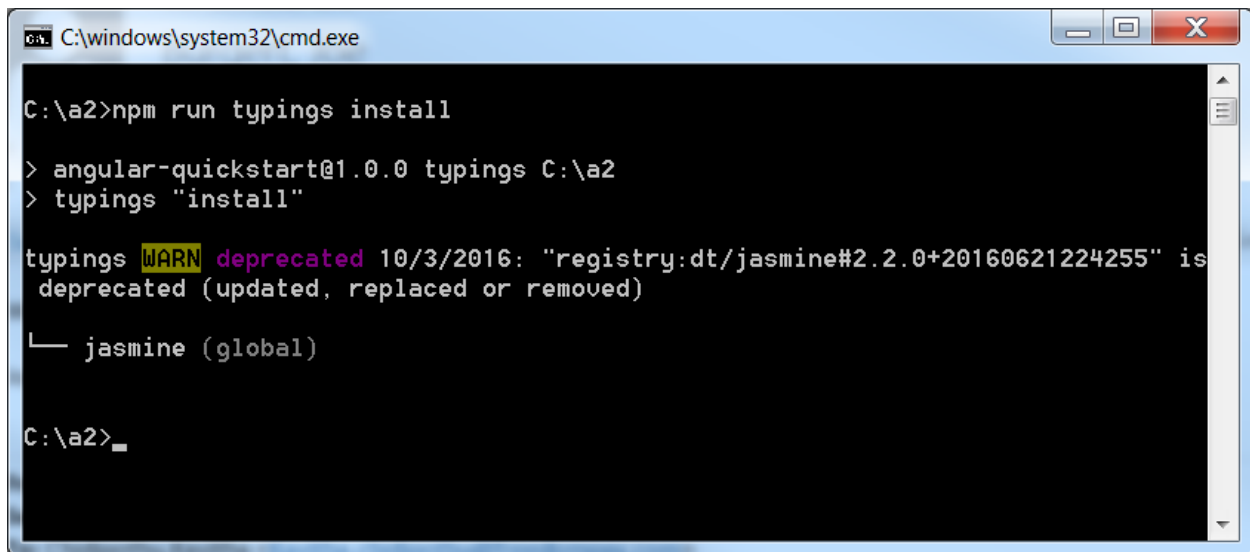
Some JavaScript libraries (such as jQuery) add features and syntax to the JavaScript environment that the TypeScript compiler doesn't know how to handle. When the compiler doesn't know how to handle the syntax, it throws an error.

Library developers might create their own type declaration files (- **d.ts** extension). These would provide the compiler with information about features in the libraries. The **typings.json** file tells the compiler where to find the definition files for libraries we might use. The following lines of code provide typing location information for three features we might need during our application development.

Code Listing 27

```
{
  "globalDependencies": {
    "core-js": "registry:dt/core-js#0.0.0+20160914114559",
    "jasmine": "registry:dt/jasmine#2.5.0+20161003201800",
    "node": "registry:dt/node#6.0.0+20161014191813"
  }
}
```

These lines indicate that these particular libraries will be used, so the NPM program should find them and install them as part of building this application. However, while these libraries are not needed for Hello World (although useful for other applications), you will get a warning message about them).

A screenshot of a Windows command prompt window titled "C:\windows\system32\cmd.exe". The window has a black background with white text. The user has entered the command "npm run typings install". The output shows the command being executed: "> angular-quickstart@1.0.0 typings C:\a2" and "> typings 'install'". A warning message is displayed: "typings WARN deprecated 10/3/2016: 'registry:dt/jasmine#2.2.0+20160621224255' is deprecated (updated, replaced or removed)". Below the warning, it shows "└─ jasmine (global)". The prompt "C:\a2>" is visible at the bottom.

```
C:\windows\system32\cmd.exe

C:\a2>npm run typings install

> angular-quickstart@1.0.0 typings C:\a2
> typings "install"

typings WARN deprecated 10/3/2016: "registry:dt/jasmine#2.2.0+20160621224255" is
deprecated (updated, replaced or removed)

└─ jasmine (global)

C:\a2>
```

Figure 8: TypeScript Warning Message

To fix the warning message, you will need to update the library using the **typings install** feature. The syntax is as follows.

Code Listing 28

```
npm run typings -- install dt~jasmine --save --global
```

This command runs the **typings** application and passes everything after the double dash to the **typings** program, so it becomes:

Code Listing 29

```
typings install dt~jsamine --save -global
```

You will notice that the **typings d.ts** file has been updated (look in **typings\globals\jasmine**), and the **typings.json** file has also been updated with the latest version number.

Code Listing 30

```
"jasmine": "registry:dt/jasmine#2.5.0+20161003201800"
```

You can also run this command.

Code Listing 31

```
npm run typings list
```

This will show you a list of all the **typings** files (**d.ts**) that have been installed for the application.



Note: *You should not need to edit the `typings.json` file manually, but rather use the `typings` tool to update the dependencies and version numbers.*

Summary

There are many configuration files with the project, which provide great deal of flexibility in how you develop your Angular 2 applications. The most common ones you'll need in development are the **package.json** (to get libraries and new versions) and **angular-cli.json** (for controlling the code generation and execution in the application) files.

Chapter 5 Customization

You can edit the configuration files to customize many aspects of your development environment. In this chapter, we are going to cover how to customize your application by integrating some third-party libraries you might want to use.

Adding libraries

While the `ng new` command will generate a **package.json** file for you, it does not know about the various libraries you might need in your application. For example, if we want to add Bootstrap and Font Awesome to our application, we would need to first install them using NPM.

Code Listing 32

```
npm install bootstrap@next
npm install font-awesome
```

After running this command, you will see the library files in your **node_modules** folder.

angular-cli.json

The configuration file **angular-cli.json** can be found in the `\angular-cli\blueprints\ng2\files` subdirectory, and contains various configuration options for running the Angular CLI. Within the **apps** section, you will find a few options that are useful to know:

- **root**: The folder of the actual application, usually **src**.
- **outDir**: The folder where distribution files should be written, usually **dist**.
- **assets**: An array of additional files that should be included in the distribution.
- **styles**: An array of style sheets that should be included.
 - We can add the Bootstrap styles sheets here, using:
 - `../node_modules/bootstrap/dist/css/bootstrap.css`
- **scripts**: An array of JavaScript files that should be included.
 - We can add Bootstrap and the dependencies by adding the following lines:
 - `../node_modules/jquery/dist/jquery.js`
 - `../node_modules/tether/dist/js/tether.js`
 - `../node_modules/bootstrap/dist/js/bootstrap.js`

Once these additions are made, Bootstrap styles and JavaScript files will be bundled and included in your application when you run it and when you build the distribution.

Font Awesome

The process is basically the same, except that we need to import the actual font files. For this, we use the **addons** collection in the **angular-cli.json** file.

Code Listing 33

```
styles: [ ..., "../node_modules/font-awesome/css/font-awesome.css" ]
addons: [ "../node_modules/font-
          awesome/fonts/*.+(otf|eot|svg|ttf|woff|woff2)" ]
```

The **addons** collection is used for files your application needs, other than scripts and style sheets.

Assets

The **assets** collection contains various elements (files and folders) that you want copied to the distribution folder when the application is built. For the default build:

Code Listing 34

```
assets: ["assets", "favicon.ico"]
```

This tells the build process to copy the contents of the **assets** folder and the **favicon.ico** file to the distribution folder (typically **dist**).

Environments

The **Environments** folder contains TypeScript files that will be loaded depending on the environment the build is being made for. For example, the **environment.prod.ts** file located in the **angular-cli\blueprints\ng2\files__path__\environments** subdirectory contains the following code.

Code Listing 35

```
Export const environment = {
  Production: true;
}
```


You can import the **environment** module into your application and query the **Production** property in case your application needs to behave differently depending on the environment.

Summary

Although you can manually add scripts and style sheets to your HTML templates, by using Angular CLI, you gain the benefits of bundling to reduce page load times. You can also use the assets to move additional content to the distribution. Finally, the **environments** folder lets you provide properties your application can use to distinguish whether it is running in development or production.

Chapter 6 Your Environment

As we saw in [Chapter 4](#), Angular CLI creates a default structure for project. The **src/app** folder holds all the TypeScript source files, and the TypeScript compiler configuration file. You will also note that the default installation places the template HTML and CSS files in the same folder. This is okay for small projects, but you are not limited to that approach.

Your folder structure

Within the **app** folder, you can structure your folders any way you'd like. I use an approach somewhat like Microsoft MVC applications.

- **Classes:** Class libraries
- **Interfaces:** Interface modules
- **Services:** Service layers
- **Views:** HTML template code

If you use a different structure, you will need to adapt some of your pathing references to your structure. If you have multiple Angular 2 applications, I would recommend using the same folder system for each application.

Another approach might be to put files related to application functionality in the same folder, such as:

- **Payroll:** Payroll/employee information
- **Ordering:** Customers and orders
- **Inventory:** Inventory and work in process

Summary

Decide what overall structure works best for you, and stay with it—particularly in a team environment where multiple people might be working on the application.

Chapter 7 Exploring Hello World

In this chapter, we are going to look into the default files created by Angular CLI to see how their version of **Hello World** works.

Start Angular CLI

Let's begin by creating an application using the Angular CLI tool.

Code Listing 36

```
ng new HelloWorld
```

Once the application has been created, open the **src\app** folder, and let's look at the default component (**app.component.ts**). The component will be very simple, but will get us started. It should have the following (or similar) code, depending on Angular updates.

Code Listing 37

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

Every Angular 2 application needs at least one root component, and by convention, it is named **AppComponent**. This component is responsible for populating a portion of the screen. (If you remember the Shadow DOM and **template** tag from our introduction chapter, this will make sense as to what exactly Angular is doing.)

Although simple, this code illustrates a basic component inside Angular 2. Let's take a look.

Import

The **import** statement is used to reference the modules we need (similar to the **using** statements from a C# program). For our simple program, we only need the **Component** module found in the **angular/core** library. For actual applications, you will likely have more than one **import** statement.



Tip: When entering an `import` statement, enter the file name first, so IntelliSense can show you the component names it finds in the file.

@Component

The `@Component` statement is a function that takes an array of metadata and associates it with the component we are defining. For this component, we specify the CSS selector and the template we want to use. When the component is instantiated, the tasks should sound familiar.

- Create a Shadow Root from the selector (Shadow Host).
- Load the selected template into the Shadow Root.

Export statement

The final line of code, the `export` statement, tells the name of the component that this module will make available to others. The code between the braces `{ }` is where we can store properties and methods of the component class, although we will not need any class details yet for the simple `HelloWorld` component.

This `export` statement allows other components and modules to import it, which we will see in our next module.

Modules

The `App.module.ts` file is the root module that describes how the application elements work together. Every application will have at least one Angular module. The root module for our `Hello World` application is shown here.

Code Listing 38

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

import { AppComponent }  from './app.component';
@NgModule({
  declarations: [ AppComponent ],
  imports:      [ BrowserModule, FormsModule, HttpClientModule ],
```

```
providers:    [],
bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Like our component, we import the components from Angular (**NgModule**, **BrowserModule**, **FormsModule**, and **HttpModule**) as well as our application component (**AppComponent**).



***Note:** The default module contains the imports from the Angular library that most applications are likely to need. Your applications might not need these modules, but are just as likely to need additional modules to be imported.*

The **@NgModule** decorator takes some metadata about how to compile and run the application.

Imports array

The **imports** array lists all modules that the application will need. Every application that runs in a browser will require the **BrowserModule**. The default module also imports the **Forms** and **Http** modules.

Declarations array

This array contains the components our application might need. In this case, only the app component we created earlier is needed. In other applications, you are likely to have multiple components specified in the array.

Bootstrap array

The **bootstrap** array contains the name of the component that Angular will create and insert into the index.html host webpage. In this case, it is also our app component.

Our main program

Now that we've seen the module and component, we need another program to start the environment (called bootstrapping) to load our component. In our **src** folder (not the **app** folder), there is a file called **main.ts**.

Code Listing 39

```
import './polyfills.ts';
```

```
import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';
import { enableProdMode }          from '@angular/core';
import { environment }              from '../environments/environment';
import { AppModule }               from '../app/';

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

The first line imports the **polyfills** library, which is used to allow older browsers to use certain TypeScript features that the browser doesn't support (not a graphics library, which the name might suggest). This allows you as the developer to write code using the new browser features you need, even if the browser doesn't natively support the features.

The next four lines allow us to load components. The platform browser module allows us to run the application within a browser. However, Angular 2 applications might run on other platforms, so this approach allows the same code to be a bootstrap for the browser, or possibly another platform.

The next statement checks to see whether we are running in production or development mode. Since we are importing the **environment** module, we can access the **production** flag and optionally add some additional code (in this example, the **enableProdMode()** function). If **ProdMode** is enabled, some development mode features (such as assertion checking) are turned off.

The last line of code calls the **bootstrapModule** method and passes it our module. We could have made a single file to do the component and bootstrapping, but remember that we do not want to write tightly coupled code. Displaying the view (**AppComponent.ts**) and bootstrapping the application (**main.ts**) are two distinct tasks, so having them in separate components is a nice example of loosely coupled component design.

Index.html

The final piece we need is to create the **index.html** file, which should be very familiar to web developers. Index.html is your starting point, and contains very common statements. You'll declare the **<html>**, the title, and some meta tags, as shown in Code Listing 40.

Code Listing 40

```
<!doctype html>
<html>
```

```
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>

...
```

You can add any tags or other header elements as part of your HTML.

Style sheets

If you want any style sheets, you can load them here as well. Remember that these style sheets will only affect the main DOM and not the components, which live in their own encapsulated world.

Code Listing 41

```
<link rel="stylesheet" href="styles.css">
...
```

Although you can add style sheets directly in the index file, I would recommend adding them to **angular-cli.json** and letting them get bundled together as part of the Angular build process.

Body

We can now specify our body content, which for this application is simply going to create a placeholder for the component to be written.

Code Listing 42

```
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

The **Loading...** is a placeholder that will appear on the website while the initialization process is completed. If you've added Font Awesome to your project, you can display a spinner icon during the load process, as shown in Code Listing 43.

Code Listing 43

```
<app-root><i class='fa fa-spinner fa-spin'></i>Loading...</app-root>
```

Styles.css

You will also find a style sheet (**styles.css**) that will reside in the same project folder as the **index.html** file. The default one is empty, but you can add any style rules you want applied to your application.

Code Listing 44

```
h1 {  
  color: #369;  
  font-family: Arial, Helvetica, sans-serif;  
  font-size: 250%;  
}  
body {  
  margin: 2em;  
  background-color: lightgrey;  
}
```

Finally, let's see what this looks like.

Open a command window, change to the application directory, and enter the following.

Code Listing 45

```
npm start
```

You will see a lot of compilation messages scroll by, and eventually see a message indicating the “**bundle is now VALID.**” When this message appears, open a browser and navigate to <http://localhost:4200>. If all goes well, you will see our very impressive **Hello World** application running in the browser window.

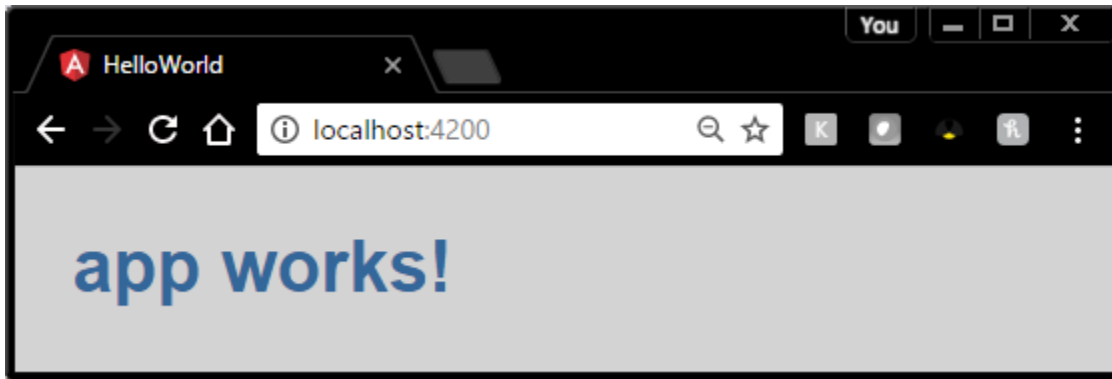


Figure 9: Hello World

Summary

It might seem like a lot of files and work for a simple Hello World application, but fortunately, Angular CLI does most of the work for you. We now have a component-based application running in the browser. While it might not sound like much, as we explore Angular in more detail, you should come to appreciate how powerful components are in the world of front-end development.

Chapter 8 Tweaking It a Bit

Now that Hello World is running, we want to look at a feature that NPM scripts provide to make development easier.

ng serve

ng serve is the Angular CLI web server which runs locally (**localhost**) in your browser. When you run **npm start** (the script that runs **ng serve**), you'll see information about the build processor, and hopefully an indication that the build is now valid.

```
Hash: a9ca33ae05db94d7f206
Time: 2719ms
chunk    <0> main.bundle.js, main.bundle.map <main> 4.65 kB <2> [initial]
chunk    <1> styles.bundle.js, styles.bundle.map <styles> 10.1 kB <3> [initial]
[rendered]
chunk    <2> vendor.bundle.js, vendor.bundle.map <vendor> 2.22 MB [initial]
chunk    <3> inline.bundle.js, inline.bundle.map <inline> 0 bytes [entry]
webpack: bundle is now VALID.
```

Figure 10: Start Up

ng serve is also looking at the folder and detecting changes. If you keep the browser open and change your source files, you'll see that the program detects the change, recompiles the code, and refreshes the browser on the fly.

```
webpack: bundle is now INVALID.
10% building modules 0/1 modules 1 active ...x.js!E:\Joe\HelloWorld\src\styles.
40% building modules 1/2 modules 1 active ...x.js!E:\Joe\HelloWorld\src\styles.
Hash: 3da667b5b7a89eb79417
Time: 2540ms
chunk    <0> main.bundle.js, main.bundle.map <main> 4.65 kB <2> [initial]
chunk    <1> styles.bundle.js, styles.bundle.map <styles> 10.1 kB <3> [initial]
[rendered]
chunk    <2> vendor.bundle.js, vendor.bundle.map <vendor> 2.22 MB [initial]
chunk    <3> inline.bundle.js, inline.bundle.map <inline> 0 bytes [entry]
webpack: bundle is now VALID.
```

Figure 11: Detecting Changes

This watching for changes and incrementally compiling them makes it easier to make code updates and immediately see the impact of the change. It will show you which file was changed, and then update your browser with the changes—a nice feature during your development cycle.

Summary

Once your application is started, the tools provided give you a nice environment to explore and see what is happening behind the scenes.

Chapter 9 Components

Now that the environment is up and running and we've seen some of the tools we can use, it is time to dig into the actual features and language syntax of Angular 2. We will start with the most basic element: the component.

The component is an encapsulated piece of code that is generally responsible for handling a piece of the user's screen. We define our base page with a collection of CSS selectable elements and then ask Angular to call the proper component to populate the selected element. As the user navigates through our site, Angular creates the components necessary to populate the page elements.

Component files

Each component should reside in its own .ts file. This file will typically contain three sections: the import section, which specifies the other components we want to use (import) in our component; the metadata section, which provides details about the component to Angular; and finally, the actual component code itself.

Import

The basic syntax is:

Code Listing 46

```
import { Component_name } from 'module location'
```

At minimum, you will need to import **Component** from the Angular core. This is necessary to provide metadata about the component to Angular.



Tip: When adding the `import` to your code, specify the `from` name first; this will allow Visual Studio Code to use IntelliSense to validate the component name automatically.

Code Listing 47

```
import { Component } from '@angular/core';
```

There are many components available in the Angular libraries. Some of the possible component classes you are likely to need are:

- **Http** in `@angular/http`: Basic HTTP operations like **GET**, **POST**, etc.
- **Router** in `@angular/router`: Maps URLs to components.
- **Location** in `@angular/common`: Allows interaction with the browser's URL.

As you start working with Angular 2, you will explore the various libraries available both in the base libraries provided by Angular and possibly any third-party libraries that provide components to help your front-end web development work.

Metadata

The metadata is a set of properties that describe the component to Angular. There several options available, but the most commonly used are the **selector** property and the **template** property. For components, the **selector** and a **template** method are both required.

selector

The **selector** value is a string that refers to the element on the page where the component should render its content. The selector typically is an element name, such as `<main-app>`. Other options include:

- Class: `<div class="main-app"> </div>` Selector: `.main-app`
- Attribute: `<div data-name="main-app"> </div>` Selector: `[data-name]`

template

The template is the HTML code that will be inserted into the selected element when the component is rendered. The template can be simply HTML code, but it is much more powerful than that. For example, any values enclosed in double braces `{{ }}` will get replaced with a value from the component class. Consider the following template code.

Code Listing 48

```
<h1>Company: {{CompanyName}} </h1>
```

This code would have the `{{CompanyName}}` replaced with the value of the **CompanyName** property in the component's class code. This process is referred to as interpolation, and is just the tip of the iceberg of template capabilities.

In the next chapter, we will explore the template capabilities in more detail.

templateUrl

If the template code is too large, or you just want to separate the HTML view from the component, you can use the **templateUrl** instead of the **template**, and provide a file location where the template content can be found.



Note: You can only have one or the other (*template* or *templateUrl*).

styles

The **styles** attribute allows you to define styles for elements within the template. The element is a collection of styles (delimited by brackets []). Within the collection, you can add classes and styles to be used by your component. For example, the following styles will set the **<p>** tags within your component to an antique white background and the **<h1>** tags to a navy blue text color.

Code Listing 49

```
[ `
  p: { background-color: antiquewhite; }
  h1: { color: navy; }
` ]
```

Notice the use of the backtick (```) character to create multiline styles.

styleUrls

Similar to the **templateUrl** property, the **styleUrls** property can be used to specify the styles via a style sheet (or collection of style sheets). The following syntax shows a sample property value that will load both the **style.css** and **customer.css** files.

Code Listing 50

```
styleUrls: [ 'app/styles.css', 'app/customer/css' ]
```

In this case, the style sheets are found in the **app** folder. The URL is relative to the application root (typically where the **index.html** file resides).

encapsulation

The **encapsulation** attribute is a feature that allows Angular 2 to work with old browsers (those which don't support the Shadow DOM). There are three possible values in the enumeration:

- **Emulated**
- **Native**
- **None**

The default value, **Emulated**, creates a surrogate ID for the host selector and pre-processes the styles to include this ID. This allows older browsers to have CSS unique to the component.

Native (for new browsers) takes advantage of the Shadow DOM and creates a Shadow Root for the component. This is the preferred approach to keep your component styles isolated if you know your target browsers are new, and you don't need to support older browsers.

None doesn't do anything, but leaves the styles as they are in the DOM.

To include the enumeration, you need to import the **ViewEncapsulation** enum values from the **angular/core** library, as shown in Code Listing 51.

Code Listing 51

```
import { ViewEncapsulation } from '@angular/core';
```

These are the basic component properties to get started, but there are few more that we will cover in later chapters.

Class code

The actual code that makes up the component is the final piece of the component. We use the **export class** statement to define the name of the class and its content. Here is an example class in the **AppComponent**.

Code Listing 52

```
import { Component } from '@angular/core';
@Component({
  selector: 'main-app',
  template: `<h1>{{ClubTitle}}</h1>
             <h2>{{LeagueTitle}} </h2> `
})
export class AppComponent {
  // Properties
  ClubTitle: string
  LeagueTitle: string;
  // Constructor
  public constructor() {
    this.ClubTitle = "The 422 Sportsplex";
    this.LeagueTitle="Adult Over 30 Leagues";
  }
  // Class methods
  public ChangeLeagues( newLeague: string) {
```

```
        this.LeagueTitle = newLeague;
    }
}
```

The properties are TypeScript variables that we plan to use in the class. In this example, we are simply defining two strings to hold the club and league title. The constructor is the code that gets invoked when the component is first instantiated. You can only have one constructor in your component.

You can also create class methods, such as the **ChangeLeagues()** method. This method simply changes the **LeagueTitle** property, which will then be updated in the template display.

The component class code is based on TypeScript classes, and allows you to create properties and define their scope, as well as create class methods. By default, all properties and methods in the class are considered public.



Tip: *In my opinion, explicitly using the `public` keyword is good programming style.*

Properties

To define properties within the class, you just need to declare the variable, and optionally, provide it with a data type. The basic syntax is:

Code Listing 53

```
[ public | private | protected ]
    Property Name: data Type
    [ = initial Value ]
```

By default, the property is **public**, meaning its value can be seen outside the class. **Private** means the variable is only used within the class, and it will not be defined if you try to access it externally. **Protected** means it acts like private, except that derived classes can use the property as well.

Property Name follows general rules for defining a variable: it can contain alphanumeric values and an underscore character. It cannot begin with a number.

data Type indicates the kind of data you expect to put in a property. It is technically optional, since TypeScript can infer data type by its usage, but it is a good habit to specify the type. Valid data types include:

- **number**
- **string**
- **boolean**

You can also assign an initial value as part of the declaration by adding an equal sign and the initial value (that matches the property's data type). You can declare the initial value by referencing another property with the class (including ones defined later in the class). To reference a class property, you will need to qualify the variable name with the **this** prefix.

You can also use **null** and **undefined** for initial values when declaring properties. These behave the same way they do in JavaScript.

Here are some example property definitions, showing the various features you can use when creating properties in the class code.

Code Listing 54

```
private LastClub: string = this.PriorOwner;
protected _PriorOwner: string = "YSC Sports";
private CallCtr: number = 0;
private IsAdmin: boolean = false;
ClubTitle: string = this._PriorOwner;
LeagueTitle: string;
private _CurrentOwner: string;
```

You can initialize the properties in-line or later in a class method, such as the constructor.

Accessors

Sometimes, providing direct access to a property can be risky, so you might want to control the access to the variable through program code. This is done by creating accessor functions that act just like properties, but call a function to get or set the property values behind the scenes.

To create an accessor function, you need to declare the keyword **GET** or **SET**, a function name, and a data type. The syntax is shown in Code Listing 55.

Code Listing 55

```
get| set    function_name() : Data Type
```

Get accessor

The **GET** function is used to return the value of the “property.” It could simply be a private field in the class, or a more complex business rule. For example, if we want our **ClubOwner** property to return the current owner of the club, the function would look like this.

Code Listing 56

```
get ClubOwner() : string
{
    return this._CurrentOwner;
}
```

In this example, the private function **ClubOwner** simply returns the value of the private variable **_CurrentOwner**.

Set accessor

The **set** accessor function is used to update the property, and can be omitted to create a read-only property. You can add business logic to accept or deny the update, or you might need to do additional work when the property is updated. Here is an example of a **SET** function that does two tasks. First, it confirms that you have admin permission to make the update, and second, it puts the original owner value into the **_PriorOwner** private variable.

Code Listing 57

```
set ClubOwner(newOwner: string) {
    // Check permissions
    if (this.isAdmin) {
        this._PriorOwner = this._CurrentOwner;
        this._CurrentOwner = newOwner;
    }
}
```

Assuming you've retrieved the **isAdmin** flag from some database call, you could easily control who has access to various properties in the class.

Constructors

The **constructor** is a public method (it cannot be private or protected) that is called whenever the class is instantiated. If you don't specify a constructor method, a default constructor is used. A class can only have one constructor.

The syntax is just like any void TypeScript function, using the name **constructor**. It is often used to initialize variables, or perhaps make a database call to set various properties of the object.

Code Listing 58

```
// Constructor
public constructor() {
    this.ClubTitle = "The 422 Sportsplex";
}
```

```
this.LeagueTitle="Adult Over 30 Leagues";  
this.SeasonStart = new Date();  
};
```

Class methods

A **class** method is a TypeScript function, and can be public, private, or protected, just like the properties. The syntax for declaring the method is:

Code Listing 59

```
[ public | private | protected ]  
  Method Name ( optional parameters )  
: optional return type  
{  
  body of method  
}
```

public is the default, and if no return type is specified, it will be undefined. For example, in our example, the following method could be used to change the current league being displayed.

Code Listing 60

```
// Class methods  
public ChangeLeagues( newLeague: string) {  
  this.LeagueTitle = newLeague;  
  // Retrieve league details  
}
```

Remember that any class properties need to use the qualifier **this**. You can create variables internally in the method, scoped to adjust the method as needed.

Summary

Once you've provided the module with info about how it will be used within your application (the CSS selector and template), you can work with the TypeScript language to provide whatever business logic your application needs. As your logic updates properties in the class, those updates will appear on the page via the data-binding (interpolation) system built into Angular. We'll explore this more in the next chapter.

Chapter 10 Templates

The templates you create as part of your component are the powerful UI pieces that let your component present data to your application user. In this chapter, we explore how to use templates to render any data your application needs to present. The template compiler is found in the **@angular/compiler** directory.

Template declaration

The template can be included in the metadata of the component (a good solution if the template is small) or stored in a separate file and referenced via the **templateUrl** property. The **templateUrl** property looks in the root level (same level as index.html). You can use the **moduleId** metadata property (set to the value **module.id**). The following code fragment tells the component to look in the **app/Views** folder for the HTML.

Code Listing 61

```
@Component({
  selector: 'main-app',
  moduleId: module.id,
  templateUrl: 'app/Views/League.html',
```

It is your preference as to where you would like to store your template. If you choose to store it in the metadata, you can use backticks (```) as delimiters to create a multiline template.

Now, let's look at the content we can place in the template.

HTML

The templates are HTML code, with some enhancements to let them work with Angular components. Most anything that goes in HTML can be used in a template, the notable exception being the **<script>** tag, which will be ignored. Remember, the HTML code is meant to be imbedded within a CSS selector, so tags like **<html>**, **<header>**, and **<body>** won't be of much use.

Angular includes the concept of data binding, which is basically extending the HTML syntax to bring component properties and expressions into the generated template.

Interpolation

Interpolation is the simplest example of data binding. If you include a property name from your component within `{{ }}`, the property value will be inserted into the generated HTML. If the component changes the value of the property, the HTML snippet from that component's template will be updated on the screen.

Expressions

You are not limited to putting property names between `{{ }}`; you can also put expressions. Angular will evaluate the expression (which must return a value) and convert the result to a string, which is displayed. Note that any expressions that update variables (like the increment and decrement operators) or create variables (like assignments, etc.) are not supported.

Note also that the expressions can reference component properties, but cannot reference global variables or objects. You could, however, declare the object value you want to use in your component, as shown in the following code.

Code Listing 62

```
get BrowserUserAgent() : string
{
    return navigator.userAgent
}
```

Be sure to keep your template expressions simple and free of side effects. In general, reading a property should never cause some other property or value to change.

Pipe operator

The pipe operator (`|`) allows you to apply a modifier that can control how a property appears on the screen. The basic syntax is:

Code Listing 63

```
{{ property | pipe_name }}
```

There are a number of built-in pipes included with Angular. These include:

- **uppercase:** Converts property value to uppercase.
- **lowercase:** Converts property value to lowercase.
- **percent:** Expresses the decimal value as a percentage with % sign.
- **currency:** Converts to specified currency.
- **date:** Displays property as a date string.

Some pipes (such as currency and date) can optionally take a parameter to further customize the data's appearance. The parameter is separated from the pipe name by the colon (:) character. If multiple parameters are used, they are all separated by the colon.

For example, if a date property is simply coded as `{{ SeasonStart }}`, it will appear as:

Thu Aug 11 2016 18:50:35 GMT-0400 (Eastern Daylight Time)

If we want something a little bit less wordy, we could use the date pipe with an additional format parameter.

Code Listing 64

```
<h4>New season starts on {{ SeasonStart | date:"fullDate" }}</h4>
```

This will display the date in a more readable format:

Thursday, August 11, 2016

Other date formats include:

- **medium** (Aug 25, 2016, 12:59:08 PM)
- **short** (8/25/2016, 12:59 PM)
- **fullDate** (Thursday, August 25, 2016)
- **longDate** (August 25, 2016)
- **mediumDate** (Aug 25, 2016)
- **shortDate** (8/25/2016)
- **mediumTime** (12:59:08 PM)
- **shortTime** (12:59 PM)

You can chain pipes together, so the following example will show the season start date in **medium** format and uppercase.

Code Listing 65

```
<h4>Starts on {{ SeasonStart | date:"medium" | uppercase }}</h4>
```

Custom pipes

You can create your own pipes for data transformation as well. For example, many web applications show dates and times in a friendlier fashion, such as “a few minutes ago...”, “tomorrow,” or “last week.” We can create a pipe that works on a date property and returns a string value to display on the site.

Creating the pipe class

You will need to create a component class, decorate it with the **@Pipe** keyword, and then implement the **PipeTransform** method from the interface.

Code Listing 66

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({name: 'FriendlyDate'})
export class FriendlyDate implements PipeTransform {}
```

The class code must implement the **transform()** method. This method takes a value parameter (of the input data type) and returns a string. You can have additional parameters if your pipe supports parameters.

Code Listing 67

```
transform(value: Date): string {
    let CurrentDay: Date = new Date();
    let diff: number = Math.floor(( Date.parse(value.toString()) -
                                    Date.parse(CurrentDay.toString()) ) / 86400000);
    // Check for today, tomorrow, and yesterday
    if (diff==0)      { return "today" };
    if (diff==1)      { return "tomorrow" };
    if (diff==--1)    { return "yesterday "};
    // Determine day of week
    let weekdays = new Array(7);
    weekdays[0] = "Sunday";
    weekdays[1] = "Monday";
    weekdays[2] = "Tuesday";
    weekdays[3] = "Wednesday";
    weekdays[4] = "Thursday";
    weekdays[5] = "Friday";
    weekdays[6] = "Saturday";

    let CurWeekDay:number = CurrentDay.getDay();
    let valueDay:number = value.getDay();
    if (valueDay>CurWeekDay && diff<7 ) {
        return weekdays[valueDay];
    }
    return value.toDateString();
}
```

Our class code takes a date parameter and compares it to the current day. It returns **yesterday**, **today**, or **tomorrow** if appropriate. It then checks to see if the date is less than 7 days out, and past the current weekday. If this happens, it simply displays the day name. Finally, if it cannot come up with a simple expression, it defaults to showing the date as a formatted string.

Using the custom pipe

To use the custom pipe, you will need to include it in your module declarations, as shown in the following code.

Code Listing 68: app.module.ts Excerpt

```
import { FriendlyDate } from './app.friendlydate';
@NgModule({
  declarations: [ AppComponent, AppStandings, AppScoring, AppAdmin,
    FriendlyDate ],
```

Once it is declared, you can use it just as you would any other pipe.

Code Listing 69

```
<p>Season starts: {{ SeasonStart | FriendlyDate }} </p>
```



Note: This example pipe is only to illustrate the basics of creating custom pipes. If you want to use custom pipes, be sure to download the complete version from [Bitbucket](#).

Template statements

You can bind to various events using the template statement feature of Angular. A template statement is an expression that does something, which can be an assignment or a call to a method within your component. Template statements are how data gets updated in an Angular application. A simple assignment statement might be:

Code Listing 70

```
ClubTitle = 'JB Soccer'
```

We could add a button in our template, and when the button is clicked (or another event occurs), we could execute the statement.

Code Listing 71

```
<button (click)="ClubTitle='JB Soccer'"> Take over club</button>
```

This tells Angular when the **click** event of the button occurs, assign the new owner (**JB Soccer**) to the component property **ClubTitle**. If club title was in the template, it would be updated on the screen.

You can also call methods in the component to toggle between leagues on the screen, as the following example shows.

Code Listing 72

```
<button (click)="onToggle()"> Toggle </button>
```

Be sure to keep your template statements simple, such as assignments or method calls. Complicated business logic belongs in the component methods, not in the template statement.

Displaying data in templates

We've looked at a few simple expressions and statements to get data from our component on to the application webpage. Let's expand our class a bit to show how we can display arrays, objects, etc., and not just the basic string and numeric data types.

Arrays

An array is a collection of TypeScript objects that can be simple base types or objects. Each element in the collection is assigned an index value, starting at zero. You can create an array using the following syntax options.

Code Listing 73

```
VarName: datatype [] = [ list of initial values]
```

For example, to declare all the leagues at a soccer club, we might use the following.

Code Listing 74

```
public Leagues: string[] = ['Co-ed competitive',  
                             'Co-ed recreational',  
                             'Over 30 mens']
```

This syntax would create an array of strings called **Leagues** and provide three initial values.

You can also create an array using the following syntax.

Code Listing 75

```
VarName: Array<datatype> = [ list of initial values]
```

The data type does not have to only be a base type, but could also be an object or interface implementation method. For example, if we have a **Team** class, we could use the following syntax to create an array of **Teams**.

Code Listing 76

```
public Teams: Team [] = [ new Team(1,'Nomads'),  
                           new Team(2,'Old guys') ];
```

Once the array is defined, all of the expected TypeScript and JavaScript methods, such as **length**, **push**, **slice**, etc., can be used on the array.

Interfaces

TypeScript allows you to create interfaces, which are definitions of what a class that implements the interface should look like. The interface consists of properties and methods, but no actual code. The interface serves as the blueprint for classes that implement that interface.

By convention, interface names begin with the letter **i**, although it's not a requirement. The following code example shows a sample interface for a soccer team.

Code Listing 77: interfaces/team.ts

```
/**  
 * Team interface  
 */  
export interface iTeam{  
    id: number,  
    name: string,  
    type?: string    // Co-ed, Over-30, Open  
}
```

This interface requires an **id** number and a string **name**. It also supports an optional **type** property, which is a string. You can also specify methods in the interface, which a class would then be expected to provide the code for.



Note: The question mark after a property name means that the property is optional.

Classes

A **class** variable can also be used by itself or as an array data type. For example, we might create a **Team** class that implements the **iTeam** interface, as shown in the following code.

Code Listing 78: classes/teams.ts

```
import { iTeam } from '../interfaces/Teams';
class Team implements iTeam {
  public id: number;
  public name: string;
  constructor( id: number, name: string) {
    this.id = id;
    this.name = name;
  }
}
```

In this example, the **Team** class implements the **iTeam** interface and provides a constructor that takes the **id** and team name and assigns them to the **public** variables. We can then initialize our array as shown in the following code listing.

Code Listing 79

```
public Teams: Team[] = [ new Team(1,"Nomads"), new Team(2,"Old guys")];
```

Once you've created your variables and arrays, the template syntax provides methods and directives for you to manipulate them.

Conditions

You can use the ***ngIf** syntax in your template to conditionally build part of the template-based conditions in your component. For example, if we were to declare a Boolean value called **IsAdmin** in our component, the following construct could hide or show the final menu option, based on the **IsAdmin** value.

Code Listing 80

```
<ul *ngIf="IsAdmin" class="nav navbar-nav navbar-right">
  <li>
    <a routerLink="/Admin" routerLinkActive="active" >Admin</a>
  </li>
</ul>
```

You can test expressions or simple variables, so **Teams.length>0** will work just as well as a Boolean variable. However, complex business conditions should be kept in the component to keep the display code as simple as possible.

Switch statement

The **switch** statement in Angular can be used when you have a single property or expression to be compared against multiple different values. There are two basic parts to the statement: the first provides the property or expression you wish to evaluate, and the second part is the list of values to compare the expression to.

Code Listing 81

```
[ngSwitch]="expression"  
*ngSwitchCase="value"  
*ngSwitchCase="value"  
*ngSwitchDefault
```

You can use it to include some output in your HTML, as shown in the following example.

Code Listing 82

```
<div class="container" [ngSwitch]="CurrentRole">  
  <div *ngSwitchCase=10>Header Referee</div>  
  <div *ngSwitchCase=20>Referee</div>  
  <div *ngSwitchCase=30>Scorekeeper</div>  
  <div *ngSwitchCase=40>Admin</div>  
  <div *ngSwitchDefault>End User</div>  
</div>
```

This code sample assumes the component has set the **CurrentRole** property to a numeric value. It then tests the property value, and displays a different text, depending upon the role. The final piece, the ***ngSwitchDefault**, provides the value to display if none of the prior conditions are met.

Looping

The ***ngFor** syntax is used to loop through arrays or other types of collections. It allows you to iterate each item in the collection, one at a time, and display it on the template page. The basic syntax is:

Code Listing 83

```
*ngFor = "let oneItem of collection"
  {{ oneItem }}
```

OneItem will contain whatever data type or object type was placed in the collection. For example, if we wanted to list items on the template, we could use the following code sample.

Code Listing 84

```
<ul>
  <li *ngFor="let oneTeam of Teams">
    {{ oneTeam }}
  </li>
</ul>
```

We can also extract elements from the object. For example, if our **Teams** object had both a **team id** property and a **TeamName** property, we could use the following code example to build a select list, allowing the user to pick a team.

Code Listing 85

```
<select (change)="onTeamChange($event.target.value)">
  <option *ngFor="let oneTeam of Teams" value={{oneTeam.id}}>
    {{ oneTeam.TeamName }}
  </option>
</select>
```

The **(change)** syntax tells Angular to call a component method (**onTeamChange**) whenever the value in the select box is changed. The event target value will contain the value (**team ID**) of the selected element.

Summary

The Angular template system allows you to bind data from your component to display in the browser, as well as directives to control what gets displayed and loop over collections. We will use these features in our example applications in the next few chapters.

Chapter 11 Modules

Angular uses modules as a method of tying various components, directives, pipes, and services together in a single, cohesive unit. The module lets you make some features public for component templates, as well as make services available at the application level.

Basic module options

To create a module, you need to define a class and decorate it with the **@NgModule** keyword. You will need to begin by importing two modules from Angular.

Code Listing 86

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

The **NgModule** provides the decorator for setting up the module. The **BrowserModule** provides important services that the entire application can use (such as **NgFor** and **NgIf**). Your module should always import these two, and you can import additional modules from Angular or third-party components as your application grows.

You will also need to import your application components.

Code Listing 87

```
// Get your application components
import { AppComponent }  from './app.component';
import { AppStandings } from './app.standings';
```

declarations

The **declarations** section declares an array of all the components, directives, and custom pipes we want our module to include.

Code Listing 88

```
declarations: [
  AppComponent,
  AppStandings
```

```
],
```

For your main application, you will typically declare the component that produces the menu and the components used to present various data views. These declarations are usable within the module, but are not visible outside the module unless they are explicitly exported.

imports

If you want to provide components or directives to the components that are referenced within the module, you will need to include them in the **imports** array. For example, the following listing allows the app component to use the standard directives (such as ***NgIf**) and provides access to the routing components (see the next chapter for details about routing).

Code Listing 89

```
imports:      [ BrowserModule, FormsModule, HttpModule ]
```

providers

The **providers** keyword contains an array of services that will be made available to the entire application. For example, the following code sample makes the **appRoutingProviders** from Angular and our own internal **SoccerService** available the entire application.

Code Listing 90

```
providers: [ appRoutingProviders,  
             SoccerService ],
```

Any services in the main application module can be injected into any component. For example, if a component wanted to use our **SoccerService**, you would normally need to include a **providers** section in the component itself.

Code Listing 91

```
providers: [SoccerService]
```

However, since the **SoccerService** is specified in the main module definition, you do not need to specify it in the component. Angular will know how to resolve the provider and inject it into the component. The component constructor will still request the service, and Angular will handle it.

Code Listing 92

```
public constructor(private _soccerService: SoccerService )
```

exports

The **exports** keyword provides an array of components that the module exports (allowing importing modules to use them). Declarations within the module are private; you will need to specifically export components, directives, etc., that you want to share with other modules.

Code Listing 93

```
exports:      [ AppComponent ]
```

bootstrap

The **bootstrap** keyword provides an array of all components that can be bootstrapped. Typically, this is the app component only. If a component can be bootstrapped, it will have its own selector, template, etc., and allow Angular to add it to a website.

Code Listing 94

```
bootstrap:    [ AppComponent ]
```

app.module.ts

When you create an Angular application, you should define the main application module, named **app.module.ts**. The simple module code (that exports the **AppComponent**) is shown in the following example.

Code Listing 95: app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

This module only provides **AppComponent** to the main application for the bootstrapping operation. You can skip this and directly bootstrap the **AppComponent**, but as your application grows, the module provides a convenient way to add more functionality in one location.

main.ts

If you use a module, rather than a single component, you will need to adapt your main application to bootstrap the module, rather than the component. The code to bootstrap the module is shown in the following code.

Code Listing 96

```
// Get the bootstrap component from Angular
import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';

// And get our application module
import { AppModule } from './app.module';

// Then launch the application.
platformBrowserDynamic().bootstrapModule(AppModule);
```

Summary

As your application grows, the ability to bundle features into a module will help build cohesive units of functionality. Some of the many benefits that Angular offers are components and bundles—ways to build loosely coupled applications—which are easier to debug and maintain.

Chapter 12 Our Application

In this chapter, we are going to put together a simple example application called the Soccer Dashboard to tie everything together in Angular 2 application development. The application is used to display standings for a sports league, and to allow real-time updates as the score keepers post the scores.

In one of my soccer leagues, many of the players wanted to know the standings and results right after the games were finished. By allowing the referees to record the scores on their mobile phones, even as they are leaving the field, it is very possible that the website showing the standings could be up-to-date before the players leave the fields for their trip home.



Note: The code samples in this book are available for download at <https://bitbucket.org/syncfusiontech/angular-2-succinctly>.

Screen mockups

The following are mockups (generated with Balsamiq) that illustrate what the application should look like. One benefit of mockups is that you can avoid the “iceberg effect,” where clients think the application is further along because they see the screens (the tip of the iceberg) without realizing the amount of code that still needs to be completed.

Standings page

The Standings page shows the teams sorted by points (assuming three points for a win and one for a tie). The secondary sort order is total goals. Keep in mind that when designing any ranking system, you will need multiple sort criteria to handle ties. Also, express the ranking in the most common style for the appropriate sport. Soccer uses points, while baseball uses games behind.

	Games	Points	Goals For	Goals against
Old Men United	6	16	32	12
422 Nomads	6	12	25	16
FC Dauntless	6	9	19	21
Kellie's Kickers	6	9	18	18
Blue Devils	6	7	14	24
Torn Achilles	6	3	10	27

Figure 12: Standings Page



Note: If you were to run a tournament, you could show the standings on a large monitor with a JavaScript timer to re-poll the service and get updated scores every few minutes.

Scorekeeper's page

This page is designed to be as simple as possible, so the referees or scorekeepers can quickly update the game results (feeding the Standings page). They select the game from the drop-down list, and then add the scores.

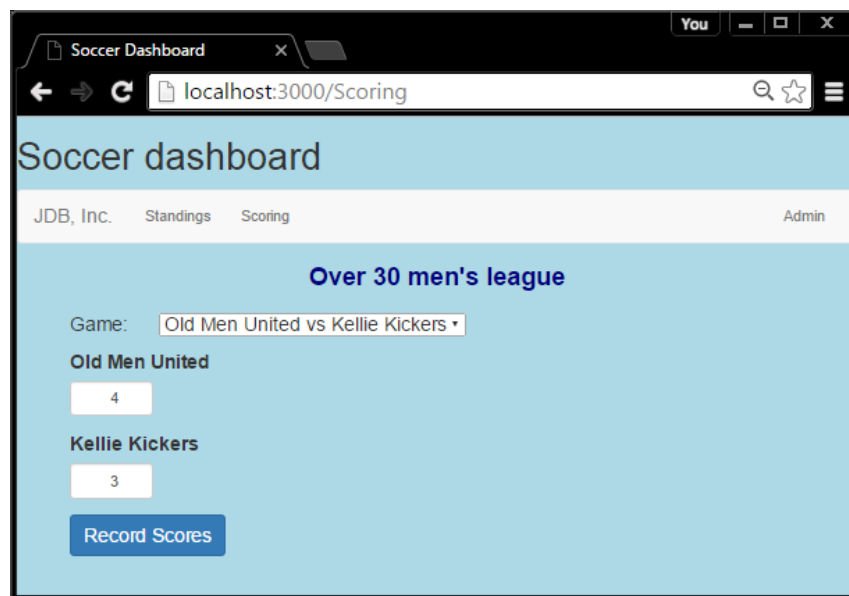


Figure 13: Scorekeeper's page

The component code will update the scores in the database, and the Standings page will be updated when it is next refreshed.



Note: The module to record the scores would generally require some sort of authentication method, such as a login password or OAuth.

Summary

This application should provide a basic walk-through of how to create modules, menus, services, etc., in an Angular 2 application. The focus is on the front-end, and the data is stored in SQL database tables. You can manually update those tables, or expand the application to create some additional Angular components to allow an admin user to update the base data tables.

Chapter 13 Menu Navigation

Typical applications have multiple views and some sort of navigation method to move between them. The **Router** module is used to handle the navigation between views. For our Soccer Dashboard, we will need three components: the Standings page, the Scoring page, and an administration page.

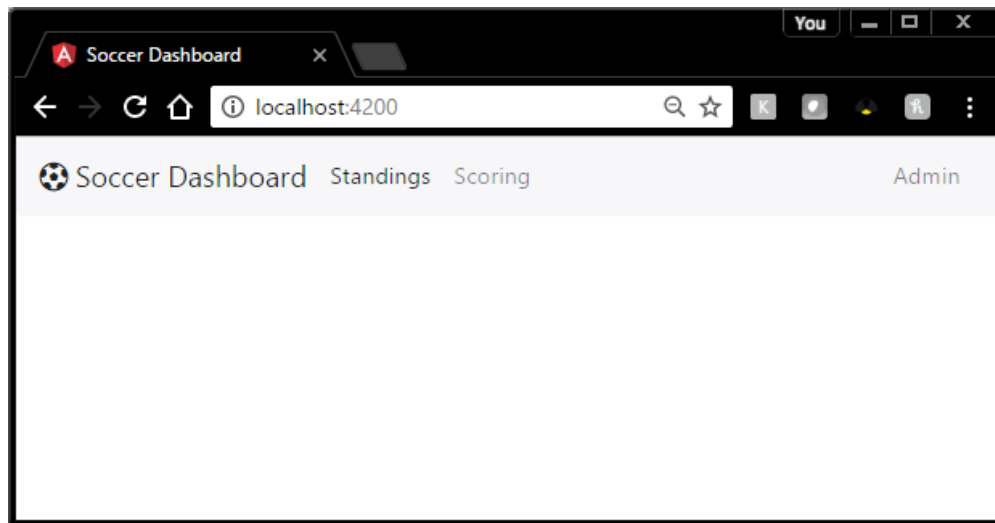


Figure 14: Sample Menu

We are going to use Twitter Bootstrap as our primary UI system and Font Awesome for some icons. Be sure to include these in your Angular CLI configuration file (see [Chapter 5](#) for details).

Base href

HTML 5 browsers provide support for browser history modification through new methods like **pushState()** and **replaceState()**. These methods allow Angular 2 to create URLs for in-app navigation. In order to support this navigation, there are a few steps you need to follow. The first is to declare a base reference in your main page (**index.html**). The following line should be added as the first line in the **head** section.

Code Listing 97

```
<base href="/">           <!-- Used to compose navigation URLs -->
```



Note: The Angular CLI application will already include the href line for you.

App component

The app component we've created in prior pages will be adapted to become a navigation component now, so we need a new component with a template focused on menus or buttons to navigate to different components.

Views folder

You can design your folder structure in any fashion you'd like. I place my template views into a folder called **Views**. It is a preference, not a requirement, but it makes my separation of business logic and display easier to manage. Here is our main application component for the Soccer Dashboard:

Code Listing 98: app.component.ts

```
<!--  
    Main application component  
-->  
import { Component } from '@angular/core';  
@Component({  
    selector: 'app-root',  
    templateUrl: './views/Main.html'  
})  
export class AppComponent {  
    title = 'Soccer Dashboard';  
}
```

Main menu

Our main menu view code, shown in Code Listing 99, is a Twitter Bootstrap navbar setup. The systems use `<a>` tags to indicate the menu links. However, rather than the standard `href` property, we will use the Angular 2 property called **routerLink**. This property will serve to tell the navigation page which “link” to go to when a menu item is selected. We will define these links later in this chapter.

Code Listing 99: app/views/main.html

```
/*  
    Main menu view  
*/
```

```

<nav class="navbar navbar-light bg-faded">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">
      <i class="fa fa-futbol-o">&nbsp;</i>{{ title }}
    </a>
    <ul class="nav navbar-nav">
      <li class="nav-item">
        <a class="nav-link active" routerLink="/Standings"
          routerLinkActive="active" >Standings</a>
      </li>
      <li>
        <a class="nav-link" routerLink="/Scoring" >Scoring</a>
      </li>
      <li class="nav-item float-xs-right ">
        <a class="nav-link" routerLink="/Admin">Admin</a>
      </li>
    </ul>
  </div>
</nav>
<router-outlet></router-outlet>

```

The final line, `<router-outlet>`, is a placeholder for where the router output will be written.

Placeholder components

To create and test the menu, we will create simple placeholder components using the same code.

Code Listing 100: app.standings.ts

```

/*
  Standings component
*/
import { Component } from '@angular/core';           // Component metadata
import { ViewEncapsulation } from '@angular/core';  // Encapsulation enum

```

```
@Component({
  encapsulation: ViewEncapsulation.Native,           // Use Shadow DOM
  template: '<h3>Standings</h3>'
})
export class AppStandings {}
```

Notice that there is no selector specified in this component, since the router system will determine where to put the generated output when this component is used. Create a similar component for the **AppScoring** and **AppAdmin** menu items.

Route definitions

To configure the routing, you will first need to define a series of route definitions. The definition contains at least two elements:

- **path**: The name users see in the address bar.
- **component**: The component that is called when we navigate to this route.

path

The path is the name of the “URL” that the router system will use to link to our component. There are no slashes in the path name; the router will handle the name resolution.

Component

The component itself is the name of the existing component that should be used when the select path is chosen.

For example, we might use the following construct for the link to the Standings page.

Code Listing 101

```
{ path: "Standings",component: AppStandings }
```

app.route.ts

The **app.route.ts** file is created to build and configure your **Routes** to various application components.

Code Listing 102: app.route.ts

```
import { Routes }           from '@angular/router';
import { AppStandings }     from './app.standings';
import { AppScoring }       from './app.scoring';
import { AppAdmin }         from './app.admin';
```

The first section of the code imports the routing module from Angular and all of the components you will need in your application. Next is the construction of the application routes.

Code Listing 103

```
export const appRoutes: Routes = [
  { path: "Standings", component: AppStandings },
  { path: "Scoring", component: AppScoring },
  { path: "Admin", component: AppAdmin },
];
```

The final section creates a public constant of the array of routes we build. We will access this array in our **app.module.ts** file.

A couple things about the routes to keep in mind:

- You can pass a parameter to the routes using the **:parameter** syntax, as shown in the following code.

Code Listing 104

```
{ path: 'game/:id', component: GameDetails }
```

- You can use the **'**'** as a wild card path name, the default component (such as a 404 error page) to use if the router cannot match any path.

Code Listing 105

```
{ path: '**', component: PageNotFound }
```



Note: One thing to keep in mind is that the order of the array is important, since the router searches the array until it finds the first match. If you put a wild card element in your first array, it would never reach any of the other paths in the route.

App module

The **app.module.ts** module will provide the link between the routing we defined in our application.

Code Listing 106: app.module.ts

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

import { appRoutes }      from './app.routes';  // Routes defined earlier
import { AppComponent }  from './app.component';
import { AppStandings }  from './app.standings';
import { AppScoring }    from './app.scoring';
import { AppAdmin }      from './app.admin';

@NgModule({
  imports:      [ BrowserModule,RouterModule.forRoot(appRoutes) ],
  declarations: [ AppComponent,AppStandings,AppScoring,AppAdmin ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

We import our various modules (both the Angular code modules and the application modules), and then we set up the declarations to organize our application. We import our routes from the **app.routes** file we created and assign them to the **RouterModule** using the **forRoot** method.

Page not found

You can add a route using the following syntax that will be called if the page is not found.

Code Listing 107

```
{ path: '**', component: NotFoundComponent }
```

This allows you to display an error message if the user enters an invalid URL. An example page **NotFoundComponent** is shown in the following code.


```
// Angular 2 modules
import { Component } from '@angular/core';           // Component metadata
import { ViewEncapsulation } from '@angular/core';  // Encapsulation Enum
// Component metadata, defining the template code
@Component({
  encapsulation: ViewEncapsulation.Native,           // Shadow DOM
  template: '<h3>Page not found</h3>'
})
export class NotFoundComponent {
}
```

You could add code to the component to log the URL failures to see if a user might be trying to hack into the website, or if a link on the site might be invalid.



Tip: *The page not found condition is not an actual condition, but simply a default of what to do if all the routes are exhausted. Be sure to add the route definition to the end of your `Routes` collection.*

Navigation

With routing defined and integrated into our application, the main menu will now load our components into the `<router-output>` section, as shown in Figure 15. Notice that the URL also changes to reflect our route name.

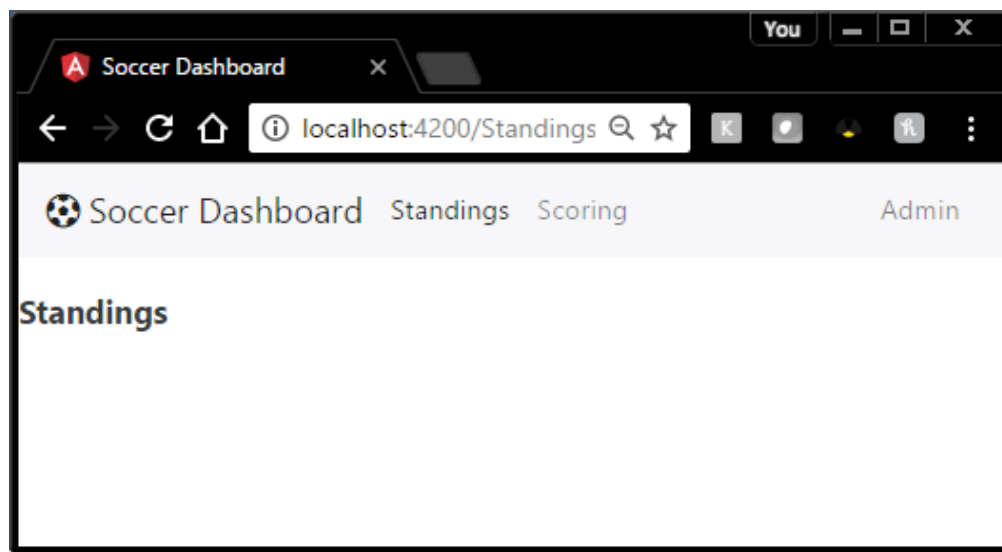


Figure 15: Routing in Action

Summary

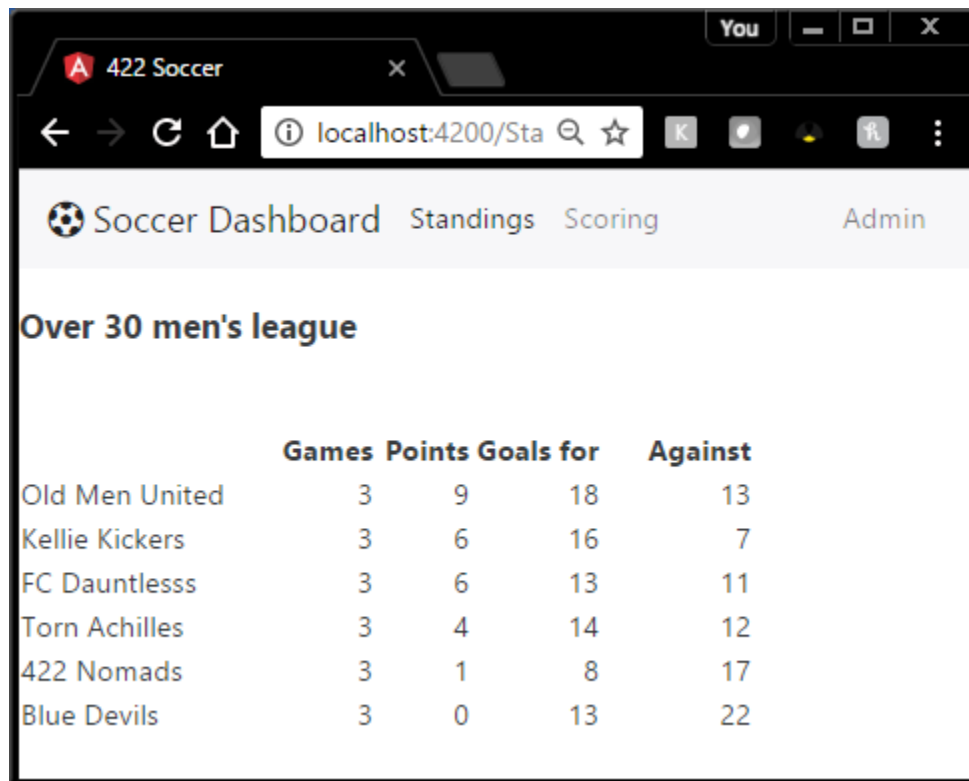
By setting up the router system and routes, we can now provide the navigation links to move between our application's views. In the next chapter, we will begin to add some actual code to the views, instead of our simple placeholder components.

Chapter 14 Services and Interfaces

Often when developing applications, you will need some code to provide functionality or data to multiple components. For that type of operation, Angular 2 allows you to create reusable services and pass (inject) them into your component for use. Shared data or functionality should immediately shout “create a service!”

Standings page

The Standings page is going to be broken into two pieces: the display and the data component. Separation of presentation and data is almost always a preferred approach to building applications. We will provide a service to get the data needed and a presentation layer to show the data.



	Games	Points	Goals for	Against
Old Men United	3	9	18	13
Kellie Kickers	3	6	16	7
FC Dauntless	3	6	13	11
Torn Achilles	3	4	14	12
422 Nomads	3	1	8	17
Blue Devils	3	0	13	22

Figure 16: Standings Page

As the scores are updated in the scoring component, the standings are updated, and when the browser revisits the page, the updated standings will appear.

Data model

The basic data model for our Soccer Dashboard is the schedule data. The schedule data includes both games already played (for the standings module) and games yet to be played for scoring module. Since the data model will serve both components, we are going to create a service to handle it.

Database design

We could use a Microsoft SQL Server database to hold our tables. Angular 2 will rely on some sort of web service to communicate with the data, so any database system (or even text files) could be used; it is a black box to Angular.

Teams table

Each soccer team has a row in the teams table. The structure of the table is:

Column	Data Type	Notes
Id	int	primary key
Name	varchar(32)	not null

You'd have more fields, such as start date, team type (co-ed, over-30, etc.), but we are only showing the basic structure. Feel free to enhance as you'd like.

Refs table

The refs table keeps track of the referees that the club uses. Each referee will have a user ID so they can log in to see the game they need to record the scores for.

Column	Data Type	Notes
Id	int	primary key
RefName	varchar(32)	not null
UserId	varchar(20)	not null

Schedule table

The schedule table contains the games that are scheduled and those that have already been played. Any game not yet played will contain a -1 in the scoring columns. This allows the referee to find games that have likely been played (on a date in the past), have not yet been scored (where scores are -1), and were refereed by the user.

Column	Data Type	Notes
Id	int	primary key
PlayingDate	date	not null
HomeID	integer	foreign key to teams
AwayID	integer	foreign key to teams
HomeScore	int	defaults to -1
AwayScore	int	defaults to -1
RefID	integer	foreign key to refs
notes	varchar(max)	

Service design

The service design process consists of two steps. First, we design the interfaces to put and get the data from the database. Then we design the service to make the data available to the various components. Within the **app** folder, create **interfaces** and **services** folders.

Interfaces

Even though the database design has three tables, we only need one interface to read and update the **games** table. We will also create an interface to hold the rankings, even though this will be computed in the code, rather than stored in the database.

Code Listing 109: Schedule Interface

```
/** Schedule interface */
export interface ISchedule{
  id: number,
  PlayingDate: Date,
  HomeTeam: string,
  AwayTeam: string,
  HomeScore: number,
  AwayScore: number,
  RefName: string,
  notes?: string }

```

Note that even though the physical table relies on foreign keys to link the content together, our interface has the actual team and referee names. Most likely, the database has a view to create the populated schedule from the three physical tables.

Code Listing 110: Ranking Interface

```
/** Rankings interface */
export interface iRanking {
  TeamName: string,
  GamesPlayed: number,
  Wins: number,
  Ties: number,
  GoalsFor: number,
  GoalsAgainst: number
}
```

Service code

With our interfaces built, we can put together a service that provides two basic functions. One is to return the entire schedule as a collection of schedule objects. The second function is to update the scores and notes for a particular schedule ID number.

Getting the data

In this chapter, we are going to create internal data for mockup and testing purposes. In a later chapter, we will get the data from an HTTP web service call. For now, this file will be stored in the **services** folder.

Schedule-data.ts

We will create a TypeScript file called **Schedule-data.ts**. The purpose of the file is to provide data to the service.

Code Listing 111

```
import {iSchedule} from "../interfaces/schedule";
import {iTeam} from "../interfaces/teams";

export const SEASON_SCHEDULE: iSchedule[] =
  [
```

```

    {id:1,PlayingDate:new Date(2016,8,23),
      HomeTeam:'Old Men United',AwayTeam:'Kellie Kickers',
      HomeScore:4,AwayScore:3,RefName:'Joanne',notes:'Overtime
game'},
    {id:2,PlayingDate:new Date(2016,8,26),
      HomeTeam:'Torn Achilles',AwayTeam:'422 Nomads',
      HomeScore:7,AwayScore:2,RefName:'Colin',notes:''},
    {id:3,PlayingDate:new Date(2016,8,28),
      HomeTeam:'Blue Devils',AwayTeam:'FC Dauntless',
      HomeScore:4,AwayScore:6,RefName:'Gene',notes:''},
    ...
  ]
export
  const TEAMS: iTeam[] =
  [
    { id:1,name:"Old Menu United",type:"Over 30"},
    { id:2,name:"422 Nomads",type:"Over 30"},
    { id:3,name:"FC Dauntless",type:"Over 30"},
    { id:4,name:"Kellie's Kickers",type:"Over 30"},
    { id:5,name:"Blue Devils",type:"Over 30"},
    { id:6,name:"Torn Achilles",type:"Over 30"}
  ]

```

The first line imports the schedule interface into the component. The component itself simply exports a collection of **schedule** objects. We now have the data elements (**interface** and **collection**) that we need to provide to our service.

Injectable

We now need to make a class that will be used as a service to other components. The first step is the use the **@Injectable** method, which means the code can be injected into other components. We will need to import both the **Injectable** module from Angular and the module to provide the data that we just wrote.

Code Listing 112

```
import {Injectable} from 'angular2/core';
```

```
import {SEASON_SCHEDULE, TEAMS } from './schedule-data';
@Injectable()
```

We follow this with the class code, which will contain the various methods that the service will offer.

Code Listing 113

```
export class SoccerService{  method calls }
```

We can have both private and public (default) methods in the **service** class; it follows the rules of a TypeScript class.

Code Listing 114: SoccerService.ts

```
/**
 * SoccerService
 * Joe Booth:  Angular 2 Succinctly
 */
import { Injectable }    from '@angular/core';
import { SEASON_SCHEDULE, TEAMS } from './schedule-data';

@Injectable()
export class SoccerService {
  getSchedule() : any {
    return Promise.resolve(SEASON_SCHEDULE);
  }
  getTeams() : any {
    return Promise.resolve(TEAMS);
  }
  private ComputeRankings() {
    // To compute rankings from the schedule
  }
}
```


In our service, we are offering two methods: the `getSchedule()` and the `getTeams()` methods. They promise to return a data collection (in this case, from the `schedule-data` class). Since the service simply returns the data, it doesn't care how the data is generated—it expects the `schedule-data` class to take care of that. In this example, we might need a private method that computes the rankings from the schedule data. (We will do this in a later chapter.)



Note: *Promise is a TypeScript/JavaScript command that allows asynchronous operations to be performed. The `.resolve` keyword provides the source of the data that will return the object.*

Not all browsers support asynchronous operations, so you can make standard calls by returning the data directly, rather than using a promise. For example:

Code Listing 115

```
getTeamsAsync() : any {  
    return Promise.resolve(TEAMS);  
}  
getTeams() : any {  
    return TEAMS;  
}
```

I would recommend providing both methods in your services so that as browsers begin to support ECMAScript 6, it should be an easy update to the service to start taking advantage of the asynchronous operation support.

Consuming the service

Now that our service is created, we want to use it in our components, in this case, to provide the data the component needs to display.

Importing the service

The first step is to tell our components where they can find the service, using the `import` statement.

Code Listing 116

```
import { SoccerService } from './services/soccerService';  
import { Title } from '@angular/platform-browser';
```

I've put the services and interfaces in separate folders, off the **app** folder. Be sure to adjust your file locations if you use a different structure. You will need to import this service to any component that uses the service. I've also decided to import the **Title** service from Angular (allowing me to set the browser's title bar).

Code Listing 117

```
// Our interfaces
import { Team } from './interfaces/Teams';
import { Title } from '@angular/platform-browser';
import { Ranking } from './interfaces/rankings';
import { Schedule } from './interfaces/schedule';
import { SoccerService } from './services/soccerService';
```

Adding the provider metadata

The next step is to tell the component about service providers it will use. This is a new component directive called **providers**. It takes a list of the services and other injectable modules our component might need.

Code Listing 118

```
providers: [Title, SoccerService]
```

In this example, we are injecting the **Title** module from **angular/platform-browser**, and our **soccerService** module. You can add as many injectable modules as needed for your component.

Update the constructor

The constructor of your component will need to be updated to receive the injected modules. For example, our **soccer** component is going to update the browser title and get its data from the **soccerService**. The following code shows the modified constructor.

Code Listing 119

```
public constructor( private _titleService: Title,
                   private _soccerService: SoccerService ) {
    this._titleService.setTitle("422 Soccer");
    this.getTeams();
}
```

Although you can name the injected modules any way you choose, I prefer the underscore character to distinguish them from other variables within your component. In Code Listing 119, we are using the **Title** service from Angular to update the browser's title, and we will add our own method, **GetTeams()**, to copy the team data from the service into a collection variable within our component.

Using the service

Our service provides different methods for returning data. We've added a method to get the teams both synchronously and asynchronously. I've added a Boolean flag called **UsingAsync** to my component (and defaulted it to **false** for now). The **GetTeams** method will use this flag to determine how to update the internal **Teams** collection object. Remember that if the method is running asynchronously, it is possible that your code will continue to run before the method completes.

Code Listing 120

```
getTeams() {  
  if (this.UsingAsync) {  
    let xx = this._soccerService.getAllTeamsAsync();  
    xx.then((Teams:Team[])=> this.MyTeams =Teams );  
  }  
  else  
  {  
    this.MyTeams = this._soccerService.getAllTeams();  
  }  
}
```

In this example, we are coding both methods to get the team data, but only using the synchronous version for now.

The **UsingAsync** Boolean property (when **true**) uses the **xxxAsync()** methods. These methods call the service and get a promise back.

Code Listing 121

```
let xx = this._soccerService.getTeamsAsync();
```

In the second portion, the **then()** tells the method what to do once the service is back. In this case, we are simply taking the service results and updating our **MyTeams** variable with them.

Code Listing 122

```
xx.then((Teams:Team[])=> this.MyTeams =Teams
```

You can learn more about promises and asynchronous programming in TypeScript and JavaScript by downloading any of the books mentioned previously from the Syncfusion library.

App.standings

The modified **app.standings** component code is shown in Code Listing 123.

Code Listing 123

```
import { Component } from '@angular/core';
import { Title } from '@angular/platform-browser';
// Our interfaces
import { Team } from './interfaces/Teams';
import { SoccerService } from './services/soccerService';
@Component({
  template: '<h3>Standings</h3>',
  providers: [Title, SoccerService]
})
export class AppStandings {
  public UsingAsync: boolean = false;
  public MyTeams: Team[];

  public constructor(private _titleService: Title,
                     private _soccerService: SoccerService ) {
    this._titleService.setTitle("422 Sportsplex");
    this.getTeams();
  }

  getTeams() {
    if (this.UsingAsync) {
      let xx = this._soccerService.getAllTeamsAsync();
      xx.then((Teams:Team[])=> this.MyTeams =Teams );
    }
    else
    {
      this.MyTeams = this._soccerService.getAllTeams();
    }
  }
}
```

```
}  
}
```

Summary

A service is a TypeScript class that provides data (or other functionality) to components as needed. In this chapter, we looked at a simple service that returns data to the components and showed how to create the service and inject it into the component. We also covered how to use the service methods within the component.

When you are designing systems, you should always consider writing any data provider as a service. Separation of data and presentation is part of Angular and makes development much more robust. The template handles the view, the components class, and the logic specific to the component, and the service provides data or globally needed business functionality.

Chapter 15 Standings

In this chapter, we are going to use the service we created in the previous chapter to display the standings on the website.

Standings component

The **standings** component is responsible for showing the standings on a website. A club might use a large video monitor to let the players see the website between games.

Code Listing 124: App.standings.ts

```
// Import various modules we might need,
// module name and what file/library to find them
import { Component } from '@angular/core';
import { ViewEncapsulation } from '@angular/core';
// Our interfaces
import { Team } from './interfaces/Teams';
import { Ranking } from './interfaces/rankings';
import { Schedule } from './interfaces/schedule';
import { SoccerService } from './services/soccerService';
```

We need to import some modules from Angular, as well as a few of our own interfaces and services.

Code Listing 125: App.component.ts Continued

```
// Component metadata, defining the template code
@Component({
  templateUrl: './app/views/Standings.html',    // HTML template name
  encapsulation: ViewEncapsulation.Native,      // Use Shadow DOM
  // Set styles for template
  styles: [`
    h3 {text-align:center;color:navy;font-size:x-
large;margin:0px;}
    table {
      width:92%;margin:1em auto;font-size:large;
```

```

        font-family:"Comic Sans MS", cursive, sans-serif; }
    th { text-decoration:underline;}
    ` ],
    providers: [SoccerService ]
  })

```

Note that in the `templateUrl` property, the `views` folder is relative to the `app` folder. Be sure to adjust your path if you choose a different folder structure.

Template page

I generally place my templates in a folder called **Views** under the main **app** folder. This is strictly a style choice—you can place them anywhere and reference their locations properly. However, I prefer keeping views, interfaces, and app code in their own folders.

Code Listing 126: Views/standings.html

```

<h3>{{LeagueName}} </h3>
<br />
<table >
  <thead>
    <tr>
      <th style="width:35%;"></th>
      <th style="width:13%;text-align:right;">Games</th>
      <th style="width:13%;text-align:right;">Points</th>
      <th style="width:18%;text-align:right;">Goals for</th>
      <th style="width:21%;text-align:right;">Against</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let currentRow of Standings">
      <td>{{ currentRow.TeamName }}</td>
      <td style="text-align:right;">{{ currentRow.GamesPlayed
}}</td>
      <td style="text-align:right;">
        {{ currentRow.Wins*3 + currentRow.Ties }}</td>

```

```

        <td style="text-align:right;">{{ currentRow.GoalsFor }}</td>
        <td style="text-align:right;">{{ currentRow.GoalsAgainst
    }}</td>
    </tr>
</tbody>
</table>

```

The template code is composed mostly of HTML and interpolation variables, such as the **LeagueName** at the top of the page. We also use the ***ngFor** directive to loop through the **Standings** collection in the component. Most of the information is straight from the properties, although we use the interpolated expression **currentRow.Wins* 3 + currentRow.Ties** to compute the **points** column. In general, you should let the component perform such calculations, but I wanted to provide a small example of how you can do calculations within the template as well.

Class code

Our class code is going to declare an empty **Ranking** array, then ask the service for the schedule data. Once the schedule data is returned, a public method will walk through the schedule and compute the rankings, updating the **Ranking** array.

Code Listing 127

```

export class AppStandings {
    // public properties (default is public)
    public LeagueName: string;
    public UsingAsync: boolean = false;
    public MySchedule: Schedule[];
    public Standings: Ranking[];
}

```

The constructor code:

Code Listing 128

```

public constructor(private _soccerService: SoccerService ) {
    this.LeagueName = "Over 30 men's league";
    this.getSchedule();
    this.ComputeRankings();
}

```


The reason **ComputeRankings** is public is that we might want to wrap the call in a timer function, in case the club is running a tournament and wants standings updates in real time.

getSchedule()

This code gets the schedule data from the service and places it in the **MySchedule** array.

Code Listing 129

```
private getSchedule() {
  if (this.UsingAsync) {
    let xx = this._soccerService.getScheduleAsnyc();
    xx.then((Schedules:iSchedule[])=> this.MySchedule =Schedules
);
  }
  else
  {
    this.MySchedule = this._soccerService.getSchedule();
  }
}
```

Once the schedule data is available, we can call **ComputeRankings** to update the **Rankings** collection (which is what the template displays).

ComputeRankings

ComputeRankings is an example of business logic in a component. Rather than have the service determine the rankings (a function only needed by the component), we will rely on the service to provide the schedule data, and let the component determine the ranking.

In general, soccer rankings award three points for a win and one point for a tie. There are often tie-breaker rules, such as goals scored, head-to-head, etc. So our component needs to read the schedule and sum up the wins, ties, goals for, and goals against. This data is placed into the ranking collection, which is then sorted to display the Standings page.

Code Listing 130

```
public ComputeRankings() {
  var curDate: Date = new Date();
  var TeamAt: number;
  this.Standings = [];
  this.MySchedule.forEach(element => {
    // Empty the array
```

```

// If game has already been played
if (element.PlayingDate < curDate && element.HomeScore>=0) {
    TeamAt = this.FindTeam(element.HomeTeam);
    if (TeamAt<0)
    {
        this.Standings.push(
            { TeamName: element.HomeTeam,
              GamesPlayed:0,Wins:0,Ties:0,
              GoalsFor:0,GoalsAgainst:0 } )
        TeamAt = this.Standings.length-1;
    }
    this.UpdCurrentRow(element,TeamAt,"H");
    TeamAt = this.FindTeam(element.AwayTeam);
    if (TeamAt<0)
    {
        this.Standings.push(
            { TeamName: element.AwayTeam,
              GamesPlayed:0,Wins:0,Ties:0,
              GoalsFor:0,GoalsAgainst:0 } )
        TeamAt = this.Standings.length-1;
    }
    this.UpdCurrentRow(element,TeamAt,"A");
}
});

```

The code reads through the schedule, and for any game in the past that has already been played (a **HomeScore** of -1 means not yet played), it adds the statistics about the game to the **Standings** collection for the matching team (using the **UpdCurrentRow** private method).

Once the collection is built, we use the TypeScript code to sort the collection, based on total points, and then **GoalsFor**.

Code Listing 131

```

// Sort standings
this.Standings.sort((left, right): number =>

```

```

{
    if (left.Wins*3+left.Ties<right.Wins*3+right.Ties) return 1;
    if (left.Wins*3+left.Ties>right.Wins*3+right.Ties) return -1;
    // Else, then are tied, typically we'd add addition logic to break
Ties
    if (left.GoalsFor<right.GoalsFor) return 1;
    if (left.GoalsFor>right.GoalsFor) return -1;
    // Finally, return zero if still tied.
    return 0;
})
};

```

The **sort** function compares two objects (so we can access the properties) and returns **1** if the right side is higher, **-1** if the left side is higher, and **0** for a tie.



Note: We could add additional tie-break rules if needed, particularly early in a soccer season when only a few games have been played.

There are a couple private routines used to support the **Standings** methods. The **UpdCurrentRow** updates the **Standings** collection based on the outcome of the selected game from the schedule.

Code Listing 132

```

private UpdCurrentRow(element:Schedule,TeamAt:number,HomeAway:string) {
    this.Standings[TeamAt].GamesPlayed ++;
    if (HomeAway=="H") {
        this.Standings[TeamAt].GoalsFor += element.HomeScore;
        this.Standings[TeamAt].GoalsAgainst += element.AwayScore;
        // Win
        if (element.HomeScore>element.AwayScore)
        {
            this.Standings[TeamAt].Wins++;
        }
    }
    if (HomeAway=="A") {

```

```

        this.Standings[TeamAt].GoalsFor += element.AwayScore;
        this.Standings[TeamAt].GoalsAgainst += element.HomeScore;
        if (element.AwayScore>element.HomeScore)
        {
            this.Standings[TeamAt].Wins++;
        }
    }
    if (element.HomeScore==element.AwayScore)
    {
        this.Standings[TeamAt].Ties++;
    }
}

```

The **FindTeam** method searches the **Standings** collection for the team by name.

Code Listing 133

```

// Find the team or -1
private FindTeam(TeamName:string) : number {
    var FoundAt: number = -1;
    for (var _x=0;_x < this.Standings.length;_x++)
    {
        if (this.Standings[_x].TeamName==TeamName) {
            return _x;
        }
    }
    return FoundAt;
}

```

Standings display

When our component is completed and run, the following screen display appears.

	Games	Points	Goals for	Against
Old Men United	3	9	18	13
Kellie Kickers	3	6	16	7
FC Dauntless	3	6	13	11
Torn Achilles	3	4	14	12
422 Nomads	3	1	8	17
Blue Devils	3	0	13	22

Figure 17: Standings Page

Summary

This chapter illustrates how to combine a service (to provide schedule data) with a component to compute and display rankings.

Chapter 16 Editing Data

In the soccer example from the last chapter, the **Standings** component only displayed the data; it did not allow you to edit or update the data at all. For scoring, we need to look up data from a drop-down list, and then allow a user to edit the scores. Finally, the new scores need to be written back to the component. For this, we need to dig deeper into data binding.

Data binding

Data binding is the linking of properties from your component to your generated template HTML. You've seen one example with interpolation, where the property values between `{{ }}` are bound to display elements within the HTML.

Property binding

Another method of binding is to assign a component property to an HTML property. For example, if you wanted to bind the name of the club into an input text, you could use the following syntax.

Code Listing 134

```
[HTML property]="Component property"
```

Code Listing 135

```
<input type="text" [value]="ClubOwner" >
```

You are not limited to text boxes; you can bind to styles, properties, classes, and more. For example, you might have an `` tag that represents an avatar or photo for a person. The `[src]="personAvatar"` could be used to display the person's avatar or photo in an HR system.

Attribute binding

Attributes in HTML are used to initialize various elements' properties. They are assigned when the HTML is written. When the DOM is built, it converts many of the attributes to properties for manipulation in your client-side code. However, attributes set in HTML are always strings, while the DOM property that corresponds to the attribute can be any data type.

You can use the `attr.name` syntax to update the string value of the attribute, although sometimes that might not give you the intended effect. For example, the disabled attribute must be removed, not simply set to an empty string. In general, the property binding should be used to manipulate the DOM behavior.

Code Listing 136

```
<button [attr.disabled]="true"> Discard changes </button>
```

If you want to enable the button, you would have to remove the **disabled** attribute, not simply set the **attr.disabled** to **false** (which will have no effect). Fortunately, there is also a property called **disabled**, which can be manipulated using the property binding.

Code Listing 137

```
<button [disabled]="FoundChanges"> Discard changes </button>
```

Event binding

If you want to bind a statement to an event of an element, you enclose the event name in parentheses. This is how you would create an association between any event and your component code. For example, if you have a save button, you could use the following syntax to call your component's **save** method when the button is clicked.

Code Listing 138

```
<button (click) = "SaveData()" > Save changes </button>
```

Common events you might want to bind include:

- **blur**: User leaves an input field.
- **focus**: User enters an input field.
- **submit**: User clicks a button that submits the form.
- **change**: User changes value of an input element.

There are other events, such as key up and key down, window resize, etc.

When the **event** method is called, a parameter called **\$event** is available to the method. The contents of the **\$event** object will vary based on the DOM element and the event called. For example, the **input** event has a property called **target.value**, which contains the content of the input field.

Class binding

Class binding allows your component to specify a class name. For example, imagine we have two classes, one for current accounts and one for past due accounts. Inside your component, you have a method called **InvoiceClass()** that returns a string, either **Current** or **PastDue**.

The following syntax would let the component set the color for each invoice in the template code.

Code Listing 139

```
<span [class]="InvoiceClass" > {{ InvoiceNumber }} </span>
```

You can also bind a class name to a Boolean component property or function. For example, we might show the top three scorers with a class name called **Top3**. Once the fourth-highest scorer is found, we want to remove the class.

Code Listing 140

```
<span class="Top3" [class.Top3]="InTopThree"> {{ PlayerName }} </span>
```

Once the **InTopThree** method returns **false**, the **Top3** class will no longer be applied to the player name span.

Style binding

The style binding allows you to use a function or expression to apply a style to an element. For example, imagine we had a list of find results, but any results more than 30 days old should appear a bit faded.

Code Listing 141

```
<span [style.background-color]="isOver30 ? 'antiquewhite','white'>
{{article}} </span>
```

In this example, the Boolean component public property, **isOver30**, is evaluated. If **true**, then the style for the background color is set to antique white; otherwise, it is set to white. The general syntax is:

Code Listing 142

```
[style.style property name] = "expression" | function | style value
```

One-way binding summary

Here are some examples of one-way binding, all following the same general syntax:

- **[target]** (Can be property, class, style, etc.)
- **(target)** (Event name)
- **=** "expression or statement"

The one-way binding specifies the target and the expression, function, or inline statement that should be tied to the target object. The key takeaway is that the component provides the target value, either a function or a property value. However, the target value is not explicitly returned or updated in the component.

Our next section discusses two-way binding techniques, which get data back AND forth from the template page to the component.



Note: When writing component functions that return values for the target, be sure that the data type matches the expected value. In addition, the function should be simple and not impact other component variables that might be referenced in other bindings.

Interpolation and property binding can perform the exact same functionality; behind the scenes, Angular converts interpolations to property bindings. There is no reason to choose one method or the other; however, I would suggest that you lean toward readability and be consistent with your usage.

Two-way binding

Two-way binding is used to both display a component property and update the property when the user makes changes while in the template. Two-way binding is a combination of a property binding and an event binding. For example, the following syntax would display the value of **FirstName** in the input element's **value** field, and would also respond to the input event by updating the property with the value the user entered.

Code Listing 143

```
<input id="FirstName" [value]="FirstName"
      (input)="FirstName = $event.target.value" />
```

While this will work properly, it requires a lot of extra work, and you duplicate the property name. Fortunately, Angular 2 provides a syntax that makes the binding a bit easier.

ngModel

The **ngModel** directive provides a simpler syntax for two-way data binding. It is found in the **Angular/forms** library, so you will need to add the import to your app module file, and the import metadata in the **@ngModule** declaration.

Code Listing 144

```
import { FormsModule } from '@angular/forms';
```

Code Listing 145

```
imports:      [ BrowserModule, routing, FormsModule ],
```

These two changes will allow components to use the **ngModel** directive to create data binding. The basic syntax is:

Code Listing 146

```
[(ngModel)] = "component property"
```

This shorthand directive performs the property and event binding (note the brackets and parentheses) for your template. You will also need to make sure that the **input** element has a **name** property as well, so Angular can properly map the component to the HTML element.

Code Listing 147

```
<input type="number"  
      [(ngModel)]="AwayScore" name="AwayScoreVal" />
```

The HTML **name** element does not have to match the component name; Angular just needs the **name** element to perform the binding.

Summary

The binding techniques let you enhance your basic HTML in your template code to produce displays and input of your component data. In the next chapter, we will combine these techniques to create a component to record scores for our soccer club application.

Chapter 17 Scoring

The scoring component allows a person to pick a game from the schedule, update the home and away scores, and then save the updated scores back to the component. The scoring screen is shown in the following figure.

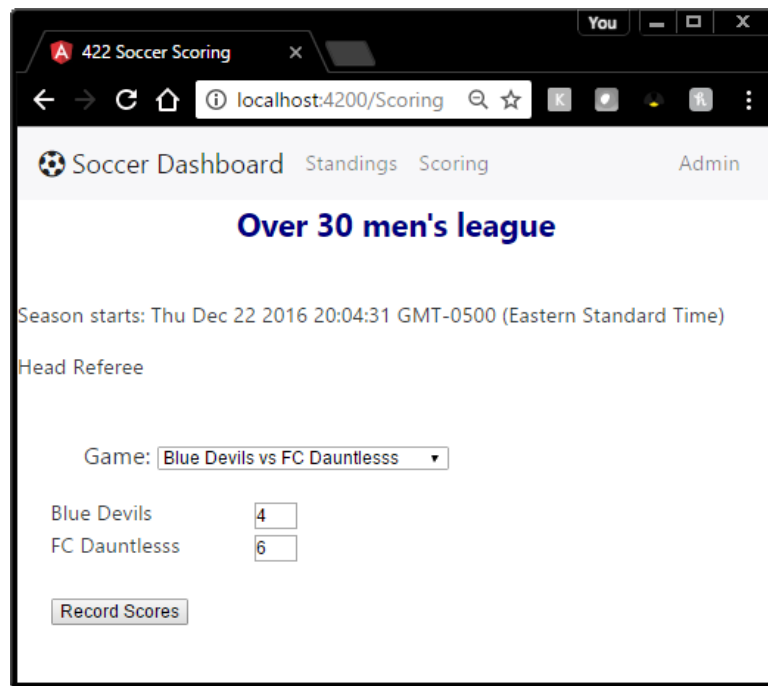


Figure 18: Scoring Screen

The user selects a game, the screen updates both team names as labels, and the team scores as edit boxes. The user can then change the scores and click the **Record Scores** button to save the updated scores (and possibly update the Standings view).

Scoring component

Code Listing 148

```
/* Scoring component */
import { Component } from '@angular/core';
import { Title } from '@angular/platform-browser';
import { ViewEncapsulation } from '@angular/core';
// Our application services and interfaces
```

```
import { Schedule } from '../interfaces/schedule';
import { SoccerService } from '../services/soccerService';
@Component({
  templateUrl: './app/views/scoring.html',          // HTML template name
  styles: [` h3 {text-align:center;color:navy;
              font-size:x-large;margin:0px;font-weight:bold;}
            select { font-size:large;margin-left:25px;} `],
  providers: [Title,SoccerService]
})
```

Scoring template

The scoring template is a simple HTML page where the scores can be recorded by the referee.

Code Listing 149

```
<h3>{{LeagueName}} </h3>
<br />
<form>
  <p>Season starts:  {{ SeasonStart }} &nbsp;</p>
  <div class="container" [ngSwitch]="CurrentRole*10">
    <div *ngSwitchCase=10>Head Referee</div>
    <div *ngSwitchCase=20>Referee</div>
    <div *ngSwitchCase=30>Scorekeeper</div>
    <div *ngSwitchCase=40>Admin</div>
    <div *ngSwitchDefault>End User</div>
  </div>
<br/>
```

This code simply changes the header based on the current role, so the type of user running the page is displayed. You could use a similar approach to have certain features or options visible or not, based on the user admin level.

Code Listing 150

```
<p style="font-size:large;margin-left:50px;">Game:
  <select (change)="onSchedChange($event.target.value)">
    <option *ngFor="let currentRow of MySchedule"
      value={{currentRow.id}}>
      {{ currentRow.HomeTeam + " vs " + currentRow.AwayTeam }}
    </option>
  </select>
```

```
</p>
```

This code uses the ***ngFor** directive to populate the drop-down box by iterating over an array. It also binds the **change** event of the **select** element to an event inside the class code.

Code Listing 151

```
<div class="form-group">
  <label for="HomeTeam">{{HomeTeam}}</label>
  <input type="number" (input)="HomeScore = $event.target.value"
    class="form-control" id="HomeGoals" [value]="HomeScore">
</div>
<div class="form-group" >
  <label for="AwayTeam">{{AwayTeam}}</label>
  <input type="number" [(ngModel)]="AwayScore" name="AwayScore"
    class="form-control" id="AwayGoals" [value]="AwayScore">
</div>
```

We then have code to display the home and away team names using interpolation and property binding. This links the edit box value with the variable in the component.

Our final part of the template is to link the **button click** event to a method (which must be public) inside the component.

Code Listing 152

```
<button type="button" (click)="onRecordScores()"
  class="btn btn-primary">Record Scores</button>
</form>
```

There are several data-binding elements within the HTML template to collect data and trigger events within the component code. The **(click)** binding will call the **onRecordScores()** method inside the component.

You will also notice some simple interpolations, such as **HomeTeam** and **AwayTeam**, for displaying the selected team names as part of the screen display.

Class code

The class code for the scoring component performs several actions: it populates the schedule array, detects when a user changes the drop-down box, and fills the score based on the schedule information. It also has code to handle the record scores option when the user clicks the button.

Variables

We declare several variables inside the component that we will use both for the component itself and public variables for the template page.

Code Listing 153

```
private UsingAsync: boolean = false;
    private CurGame: number = 0;
    public MySchedule: Schedule[];
    public LeagueName: string;
    public HomeTeam : string;
    public AwayTeam : string;
    public HomeScore : number =0;
    public AwayScore : number =0;
    public SeasonStart: Date = new Date();
    public CurrentRole: number=1;
```

Constructor

The constructor code updates some variables and specifies the service to be used throughout the component.

Code Listing 154

```
public constructor(private _soccerService: SoccerService ) {
    this.LeagueName = "Over 30 men's league";
    this.getSchedule();
    this.SeasonStart.setTime( this.SeasonStart.getTime() +4 * 86400000 );
    if (this.MySchedule.length>0) {
        this.UpdVariables(0);    // Default to first game
        this.CurGame = 1;
    }
}
```

Public methods

Any method that the template code references must be declared as **public**. We need two public methods: one for the select box change, and another for the user clicking the button to record the scores.

Code Listing 155

```
public onSchedChange(GameValue:number) {
    if (GameValue>0)
    {
        this.UpdVariables(GameValue);
        this.CurGame = GameValue;
    }
}

// Get the score from the form and update it
public onRecordScores() {
    this.MySchedule[this.CurGame-1].AwayScore =
    Number(this.AwayScore);

    this.MySchedule[this.CurGame-1].HomeScore =
    Number(this.HomeScore);
}
```

Private methods

The **private** methods are generally methods needed to support the component, but do not need to be used outside the component body. We have a couple private methods in this component.

Code Listing 156

```
// Update screen variable based on current game
private UpdVariables(GameID: number) {
    // Need to search Schedule array, looking for game ID
    var x : number = 0;
    if (GameID >0) {
        x = GameID-1;
    }
    this.HomeTeam = this.MySchedule[x].HomeTeam;
    this.AwayTeam = this.MySchedule[x].AwayTeam;
    this.HomeScore = this.MySchedule[x].HomeScore;
    this.AwayScore = this.MySchedule[x].AwayScore;
}

// Get the schedule (only showing games not yet scored)
```

```
private getSchedule() {  
    if (this.UsingAsync) {  
        let xx = this._soccerService.getScheduleAsnyc();  
        xx.then((Schedules:Schedule[])=> this.MySchedule =Schedules  
    );  
    }  
    else {  
        this.MySchedule = this._soccerService.getSchedule();  
    }  
}
```

If you update the scores, and then go back to the Standings page, you will see the changes reflected in the standings, goals for, and goals against. For an actual application, you would most likely write a web service and use the HTTP module to persist the changes to the back-end database.

Summary

In this chapter, we looked at a simple example of getting data back and forth from the form to the component, and responding to form events and button clicks.

Chapter 18 Getting HTTP Data

Many applications use a database to persist the information the application has collected. It could be Microsoft SQL Server, Oracle, or another database. In this chapter, we will cover how to consume a web service to collect some data and possibly write it back.

Web services

The HTTP protocol was designed to allow requests to be sent to web servers and responses to be retrieved from the servers. Browsers are designed to make requests of webpages, and typically bring back complete webpages to the browser. When you enter a URL in a browser address bar, such as www.syncfusion.com, the browser looks up the IP address for the website and sends a request to that website, which returns with a full HTML webpage to display to the user.

A web service operates very similarly, except rather than return a full webpage, it typically returns a small amount of formatted data that an application can then use as needed. The data is generally formatted as XML documents or a JSON string. Angular 2 works with JSON data very well, and we will look at how to communicate using JSON -oriented web services.

JSON test website

The [JSON test website](http://ip.jsontest.com/) provides a few sample web service calls that return JSON data. For example, if you enter the URL <http://ip.jsontest.com/> in your browser, you will not get a full website back, but rather a JSON string with your IP, as shown in the following code.

Code Listing 157

```
{"ip": "127.0.0.1"}
```

Angular provides the functionality to make the API request and gets the returned data into your application. There are thousands of APIs available. Many are free, but require you get an access key to use them.

JSON format

JSON (JavaScript Object Notation) is an open-source format for transmitting data objects. It is designed to be both compact and readable by humans at the same time. Although JSON is language independent, you can see its JavaScript roots.

JSON elements consist of a property name and a value, such as the IP example in Code Listing 157. A JSON object may have multiple elements. The JSON object is delimited by { }, with the collection of elements between the brackets. For example, you might represent a person with a JSON object of:

Code Listing 158

```
{ "name": "Jonathan" , "phone": "555-1212", "email":  
jonathan@bitbucket.com }
```

The object is readable by both humans and computer applications. You've also seen some JSON files in earlier chapters as you've set up the Angular 2 environments.

The property values can be numeric, string, Boolean, or date. You can also have a collection of objects by nesting the objects with an outer set of { } characters.

You can make a collection of objects by providing an object name and the [] delimiters. Within the brackets could be multiple objects. For example, the JSON in the following code represents various phone numbers a person might have.

Code Listing 159

```
"PhoneNumbers" : [  
    { "type": "home","areaCode": "215","number": "555-1212" },  
    { "type": "mobile","areaCode": "610","number": "867-5309" },  
]
```

The JSON structure provides flexibility to be as simple or complex as your application needs it to be.

Web commands

Angular 2 provides the HTTP module that allows you to perform the following basic web methods:

- **GET**: Retrieve data from a URL; query parameters can be used.
- **POST**: Like **GET**, except parameters are passed in the request body.
- **HEAD**: Like **GET**, but the server only returns the HTTP headers.
- **PUT**: Upload a file directly to the web server.
- **DELETE**: Delete a resource from the web server.

We are only going to focus on **GET** and **POST**, which are the most commonly used HTTP web requests. If you've designed your web service, you can support the **PUT** and **DELETE** verbs, but by their nature, they pose security risks.

Angular HTTP

Angular includes a base library called **@angular/http**, which contains the methods needed to make web calls. You will need to import the **Http** and **Response** modules from this library, as shown in the following code.

Code Listing 160

```
import {Http,Response} from '@angular/http';
```

Next, modify your component's constructor to save the **Http** module, such as:

Code Listing 161

```
constructor(private _http: Http) {  
  }  
}
```

This is the basic setup you will need for components that will make HTTP requests.

Root module

Since we want the HTTP services to be available throughout the application, we can add the **Http** module to our **app.module.ts** file, as shown in the following listing.

Code Listing 162

```
import { HttpClientModule } from '@angular/http';  
...  
@NgModule({  
  imports: [ BrowserModule, HttpClientModule],  
})
```

Be sure to update your root module with these additions if they are not already there.

Web service

You should generally wrap the web calls in a service so they can be used by different components as needed. For example, we are going to create a **WebService** class to handle all the HTTP calls. You could import the **WebService** as a provider and save it to a private variable during the **constructor** code.

Code Listing 163

```
import { WebService } from '../services/WebService';  
@Component({  
  providers: [ WebService ],  
})
```

```

    providers: [SoccerService,WebService]
  })
  export class AppScoring {
    public constructor( private _web: WebService ) { }
  }

```

At this point, the component has access to the service and will be able to use any of the methods the web service chooses to provide. For our example, we are simply going to call the IP method from the JSON test website and display the IP address in our template. With a few tweaks, you could easily use this service to restrict access to certain sections of the template to a whitelist of acceptable IP addresses.

Creating the web service

The web service will handle the actual interaction with the sites or API you wish to use. You should start with the appropriate imports from the Angular libraries and from the Reactive Extensions (RxJS) library.



Note: The Angular 2 HTTP services rely on JavaScript observables, which is what the RxJS libraries are designed to handle. In general, an observable is a way to allow JavaScript to run asynchronously.

Web service imports

The `WebService` class will need the following imports to set up HTTP and the observables.

Code Listing 164

```

import { Injectable }    from '@angular/core';
import { Http,Response}  from '@angular/http';
import { Observable }    from 'rxjs/Rx';
import 'rxjs/Rx';

```

Since it is a service, we will need to make it injectable and save the `Http` module during the constructor.

Code Listing 165

```

@Injectable()
export class WebService {
  constructor(private _http: Http) {
  }
}

```

With this basic setup, we can start adding the various methods to make HTTP calls. Our first method will call the **ip.jsontest** web service to get the IP address of the site visitor. This could be useful for logging purposes or to whitelist access to certain areas of the website.

Code Listing 166

```
private _IPURL: string = "http://ip.jsontest.com";
public getIP (): Observable<string> {
    return this._http.get(this._IPURL)
        .map(this.extractIP)
}
```

The **Observable** and **map** functions come from the **RxJS** library. An error message of **Map undefined** would indicate that the **RxJS** library is not loaded. You could also create private a method to extract the data elements from the returned JSON object. In our example, the private method (**ExtractIP**) gets the **IP string** field from the JSON object the call returns.

Code Listing 167

```
private extractIP(res: Response) :string {
    let body = res.json();
    return body.ip || { };
}
```

You will likely need different methods to extract object properties, depending on the structure of the JSON data that is being returned.

Using the service

Assuming your component refers to the service as **_web**, we need to perform two steps. First is to declare a component property to hold the results from the service.

Code Listing 168

```
public IPAddr: string;
```

The second step is to subscribe to the service method. Keep in mind that the web service is called asynchronously, so your TypeScript code will keep going after the call to the **getIP()** method. The **subscribe()** method is basically telling the method: go off and do your thing, but I want you to do something as soon as you are done—in this case, simply put the results into my variable.

Code Listing 169

```
_web.getIP().subscribe( IP => this.IPAddr = IP);
```

Once the `getIP()` method completes (and returns the IP address), we tell the method to take the results and put it into our `IPAddr` variable. In this, we are getting a simple string. We could also define a class that matches the web service JSON structure and have the service copy the entire JSON structure into an object created from our class.

Passing parameters

There are two ways to pass parameters to a web service. You can pass them on the **GET** query string if the parameters are part of the URL. For example, the **jsontest** site has a service that will compute an MD5 from a given text string. If we pass the following URL:

Code Listing 170

```
md5.jsontest.com?text=Angular2Rocks
```

The site will return a JSON string, as shown in the following code.

Code Listing 171

```
{
  "md5": "c2690756861ba21dc367ab72b0621906",
  "original": "Angular2Rocks"
}
```

The only caveat to keep in mind is that the parameters passed on the query string must be URL-encoded. Fortunately, TypeScript includes a function that allows us to easily do that.

Code Listing 172

```
public getMD5( str: string) : Observable<string> {
  let finalUrl = this._MD5URL + encodeURI(str);
  return this._http.get(finalUrl)
    .map(this.extractMD5);
}
```

The rest of the code to use the **GET** method and a query string parameter is the same. Don't forget the custom extraction method to get the proper JSON parameter.

Post method

A second technique to pass parameters to a web service is via the **POST** method. When you use **POST**, it hides the parameters from casual users (since they are not visible on the query string), and it allows larger parameter sizes. (Query string length is generally limited to 2048 bytes, although this varies by browser.)

When using a **POST** method, you will need to include some additional modules for building the request.

Code Listing 173

```
import { Headers, RequestOptions } from '@angular/http';
```

The **Headers** are used to tell the web service the type of data we are sending, in this case, JSON data. The **RequestOptions** are used to construct the data request to be sent to the web service.

To build the request, you will need to create both the headers and the request options, as shown in the following sample. The content type of **application/json** tells the service to expect JSON data.

Code Listing 174

```
FindZipCity (zipCode: string): Observable<string> {  
  let headers = new Headers({ 'Content-Type': 'application/json' });  
  let options = new RequestOptions({ headers: headers });
```

The actual call is like a **GET** method, except for the extra parameters of the JSON data and the request options.

Code Listing 175

```
return this.http.post(this.heroesUrl, { zipCode }, options)  
  .map(this.extractCity)
```

You would need to write your own **extractCity** method, or perhaps create a class to get all the information returned by the web service.

Error handling

No matter which method you are using, **GET** or **POST**, you should add code to check for any errors that might occur. The error handler is a private method in the web service. You only need one, and can attach it to any of the calls.

Code Listing 176

```
private handleError (error: Response | any) {  
  let errMsg: string;  
  if (error instanceof Response) {  
    const body = error.json() || '';  
    const err = body.error || JSON.stringify(body);
```

```

    errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
  } else {
    errMsg = error.message ? error.message : error.toString();
  }
  return Observable.throw(errMsg);
}

```

To attach the error handler to the method calls, you chain a **catch** method after the **map** call, and pass the error handling method. For example:

Code Listing 177

```

public getMD5( str: string) : Observable<string> {
  let finalUrl = this._MD5URL + encodeURI(str);
  return this._http.get(finalUrl)
    .map(this.extractMD5)
    .catch(this.handleError);
}

```

Your component code can access the error by checking for it during the subscribe call, as shown in the following code example.

Code Listing 178

```

_web.getIP().subscribe( IP => this.IPAddr = IP,
                        Error => this.ErrMsg = Error);

```

This will copy the error message to the component string property called **ErrMsg**. You could optionally display it in your template using the ***ngIf** directive, but the error message might not be meaningful to the end user.

Summary

The HTTP module makes web service and API calls simple—just remember that the calls are asynchronous, so you need to subscribe to get the results. This will allow your application to run quickly, even as it reaches out to other sites for JSON information.

Chapter 19 Summary

Angular 2 is a powerful framework for developing applications. In this book, we explored enough of the framework to create a simple application, but I hope it whets your appetite for what is possible. I suggest keeping the following websites bookmarked and visiting them periodically to stay on top of the new features of both Angular 2 and Angular CLI.

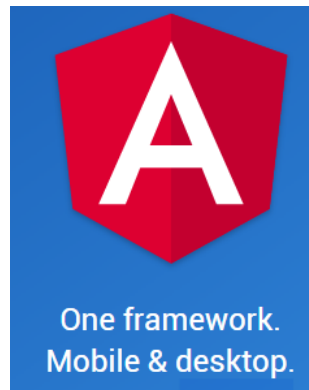


Figure 19: Angular Home Page at <https://angular.io>

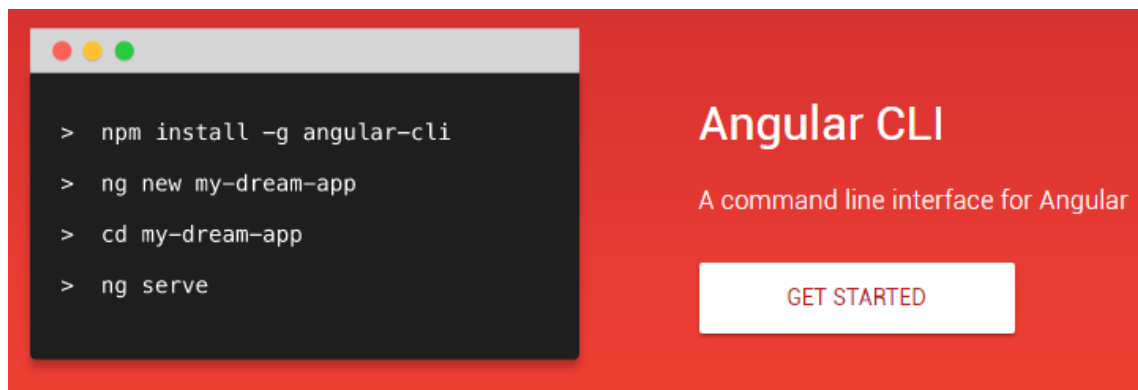


Figure 20: Angular CLI Home Page at <https://cli.angular.io>

And of course, keep the [SynCFusion website](#) bookmarked to find a large, free library of e-books to stay on top of the ever-changing world of application development.

Enjoy!

Appendix 1 Component Metadata

Each Angular 2 component has metadata associated with it, providing at minimum, two pieces of information. The location (**selector**) on the HTML page and the **template** to add to the page. Some of the key component metadata options are shown in the following table.

Directive	Purpose
selector	CSS selector that identifies where the component template code should be added.
providers	Array of the various dependency injection providers for this component.
template	String contains template code to inject, can use backticks to insert multiline templates.
templateUrl	A reference to a template stored in an external HTML page.
styles	An array of style elements to be added inline to the template.
styleUrls	An array of CSS style sheets that should be injected into the component's template.
encapsulation	<p>You can set the encapsulation method for the component using one of the following options:</p> <p>Native: Use Shadow DOM, supported by new browsers.</p> <p>Emulated: Emulate Shadow DOM by renaming CSS elements to scope the CSS to the component.</p> <p>None: No encapsulation, CSS is added directly to HTML stream (necessary for older browsers).</p> <p>You will need to import the View Encapsulation module from Angular core to access the enum of encapsulation types.</p> <pre>import {ViewEncapsulation} from '@angular/core';</pre>

Appendix 2 Template Syntax

The template of HTML and enhanced Angular 2 features will be the main source for your application's UI features. This appendix has a summary of the template options. A template generally consists of HTML and multiple bindings.

Syntax	Purpose
<code>{{component_variable}}</code>	Replaces expression between the <code>{{ }}</code> with content of the named component variable.
<code>(event_name)=method</code>	Binds event (click , change , focus , etc.) to the named component method. Optionally add \$event for event parameters.
<code>{{expression}}</code>	Replaces content between <code>{{ }}</code> with the string result of evaluating the expression
<code>[value]="variable"</code>	Binds the property value (text elements) to the component variables.
<code>[attr.name]="string"</code>	Binds the HTML attribute name to the indicate value.
<code>[class.special]="Boolean"</code>	Adds the special class to element if Boolean value is true .
Directives	
<code>*ngIf="expression"</code>	Adds or removes DOM content based on the Boolean value of the expression.
<code>*ngFor="Let x of list"</code>	Loops through the list, assigning each element to x .
<code>[(ngModel)]="property"</code>	Performs two-way binding with property name in component. Requires the forms module.