

LEARN ANGULAR 4 FAST

Over 70 Example Projects

Mark Clow

1 Acknowledgements and Revisions

1.1 Acknowledgements

1.2 Revisions

2 Introduction

2.1 How the Book Started

2.2 About the Book

2.3 Angular

2.4 Angular and Naming

2.5 My Opinion as a Developer

2.6 So Why Is Angular the Answer?

2.7 Sounds Good, Doesn't It?

3 Web Applications and AJAX

3.1 Introduction

3.2 Introducing the Client & Server

3.3 Server (Web Server)

3.4 Client (Web Browser)

3.5 Communication (Usually Performed with HTTP)

3.6 Server-Side Web Application

3.7 Client-Side Web Application (or Single Page Web Application)

3.8 The Balancing Act

3.9 AJAX

3.10 Callbacks

3.11 Promises

3.12 Encoding

3.13 HATEOAS & HAL

3.14 Debugging Tools

3.15 Analyzing JSON

4 AngularJS vs Angular (Old vs New)

4.1 Introduction

4.2 What is AngularJS?

4.3 Why Was AngularJS so Popular?

4.4 When Were AngularJS and Angular born?

4.5 Browsers

4.6 TypeScript

4.7 Modules

4.8 Controllers and Components

4.9 Dependency Injection

4.10 Scope

4.11 Forms

4.12 Templates

4.13 Conclusion

5 JavaScript

5.1 JavaScript History

5.2 JavaScript Shortcomings (< ES6)

5.3 JavaScript Strict Mode

[5.4 JavaScript \(>= ES6\)](#)

[5.5 Modules](#)

[5.6 Deployment of ES6 Code](#)

[5.7 TypeScript](#)

6 TypeScript

[6.1 Introduction](#)

[6.2 What is TypeScript?](#)

[6.3 Can Browsers Run TypeScript?](#)

[6.4 The Microsoft TypeScript Website](#)

[6.5 The Main Differences between the JavaScript and TypeScript Languages](#)

7 Editors

[7.1 Introduction](#)

[7.2 Visual Studio Code](#)

[7.3 Website](#)

[7.4 Opening Your Project in Visual Studio Code.](#)

[7.5 How to Open Files](#)

[7.6 How to See the Available Commands and Hot Keys](#)

[7.7 How to Configure the Build](#)

[7.8 How to Build](#)

[7.9 To View Build Errors](#)

[7.10 Sidebar Modes](#)

[7.11 Extensions](#)

[7.12 Notes](#)

8 Node

[8.1 Introduction](#)

[8.2 Installing Node](#)

[8.3 Setting Up Node in Your Project Folder](#)

[8.4 Running Code with the Node Command](#)

[8.5 Node Modules and Dependencies](#)

[8.6 Node Package Manager](#)

[8.7 Node Module Installation Levels](#)

[8.8 'package.json' File](#)

[8.9 Folder 'node_modules'](#)

[8.10 Npm - Installing Modules into Node](#)

[8.11 Updating Node Modules](#)

[8.12 Checking Your Node Version](#)

[8.13 Uninstalling Node Modules](#)

[8.14 How to Install the Latest Angular](#)

9 Start Coding With the CLI

[9.1 Introduction](#)

[9.2 Angular CLI Gets You Going Fast](#)

[9.3 This Chapter](#)

[9.4 CLI = Command Line Interface](#)

[9.5 You Need Node to Install CLI](#)

[9.6 Create a Start Project](#)

[9.7 Open the Start Project](#)

- [9.8 Modify the Start Project](#)
- [9.9 Start Project - Compile Errors](#)
- [9.10 Start Project - Runtime Errors](#)
- [9.11 Watcher & Web Server](#)
- [9.12 CLI Version](#)
- [9.13 How the Start Project Bootstraps](#)
- [9.14 Conclusion](#)

[10 Introducing Components](#)

- [10.1 Introduction](#)
- [10.2 What Does a Component Consist Of?](#)
- [10.3 Component Annotation](#)
- [10.4 Component Templates](#)
- [10.5 Component Styles](#)
- [10.6 Component Class](#)
- [10.7 Model View Controller](#)
- [10.8 Introduction to Data Binding](#)
- [10.9 One Way Data Binding with '{{' and '}}'](#)
- [10.10 One Way Data Binding - Example Code](#)
- [10.11 One Way Data Binding with '\[' and '\]' \(or *\)](#)
- [10.12 One Way Data Binding – Example Code #1](#)
- [10.13 Html Element Attributes](#)
- [10.14 One Way Data Binding – Example Code #2](#)
- [10.15 Two Way Data Binding with '\[\(and '\)\]'](#)
- [10.16 Two Way Data Binding – Example Code](#)
- [10.17 Event Handling](#)
- [10.18 Event Handling – Example Code](#)

[11 Introducing Modules](#)

- [11.1 Introduction](#)
- [11.2 Why Modules?](#)
- [11.3 Different Types of Modules](#)
- [11.4 AngularJS Module System](#)
- [11.5 JavaScript Modules](#)
- [11.6 Angular Module System](#)
- [11.7 Angular Module System – Example](#)
- [11.8 Deployment - Separate Modules](#)
- [11.9 Deployment – Node](#)

[12 Introducing Webpack](#)

- [12.1 Introduction](#)
- [12.2 Webpack and the Angular CLI](#)
- [12.3 What Does Webpack Do?](#)
- [12.4 What about Your Modules and Dependencies?](#)
- [12.5 Benefits](#)
- [12.6 Chunks](#)
- [12.7 Development Process](#)
- [12.8 Install Webpack](#)
- [12.9 Configuration](#)

13 Introducing Directives

13.1 Introduction

13.2 Components May Use Directives

13.3 Types of Directives

13.4 NgIf

13.5 NgIf – Example

13.6 NgFor

13.7 NgFor – Example

13.8 NgSwitch, NgSwitchWhen, NgSwitchDefault

13.9 NgSwitch – Example

13.10 NgClass

13.11 NgClass - Example

13.12 NgStyle

13.13 NgStyle - Example

13.14 Other Directives

13.15 Creating Directives

13.16 Accessing the DOM in Directives

13.17 Creating Simple Directive - Example

13.18 Accessing the DOM Events in Directives

13.19 Accessing the DOM Properties in Directives

13.20 Creating Directive with Events – Example

14 More Components

14.1 Introduction

14.2 Components and Child Components

14.3 Composition

14.4 Composition & Data Location

14.5 Data Flowing Downwards

14.6 Events Flowing Upwards

14.7 Emitting Output through @Output()

14.8 Composition Example

14.9 Template Reference Variables

14.10 Component Class Lifecycle

15 Dependency Injection

15.1 Introduction

15.2 Advantages of Dependency Injection

15.3 Angular Provided Services

15.4 Other Services

15.5 Services You May Want to Write Specifically for Your Own Project

15.6 Writing Service Classes

15.7 Providers

15.8 Injectors

15.9 Resolving Dependencies

15.10 Creating Service – Example

15.11 Why Do Multiple Instances of the Same Service Get Created?

15.12 How Do We Ensure a Single Instance of the Service Gets Created?

15.13 What Do We Do If We Want to Share a Singleton of the Service?

[15.14 Convert App to Share One Instance of Service – Example #1](#)
[15.15 Convert App to Share One Instance of Service – Example #2](#)
[15.16 Class Providers](#)
[15.17 Class Providers – Example](#)
[15.18 Factory Providers](#)
[15.19 Factory Providers – Example #1](#)
[15.20 Factory Providers – Example #2](#)
[15.21 Value Providers](#)
[15.22 Value Providers – Example](#)
[15.23 Injector API](#)

16 Angular 4 and UI Widgets

[16.1 Introduction](#)
[16.2 How to Use a UI Widget Library with Angular](#)
[16.3 Pre-Angular Way](#)
[16.4 The Angular Way](#)
[16.5 Pre-Angular \(Html & Css & JavaScript\) vs Angular Module](#)
[16.6 Bootstrap](#)
[16.7 How to Start Using Ng Bootstrap](#)
[16.8 Bootstrap - Example Code](#)
[16.9 Material](#)
[16.10 How to Start Using Angular Material](#)
[16.11 Material – Example Code](#)
[16.12 Conclusion](#)

17 Routes and Navigation

[17.1 Introduction](#)
[17.2 Router](#)
[17.3 Router Routes on The Client Side](#)
[17.4 Router Module](#)
[17.5 Router Module – Objects Included](#)
[17.6 Simple Routing - Example](#)
[17.7 Nested Routing](#)
[17.8 Nested RoutingUrls](#)
[17.9 Nested Routing – Example](#)
[17.10 Route Configuration](#)
[17.11 Route Configuration – Data](#)
[17.12 Route Configuration – Data – Example](#)
[17.13 Route Path Parameters](#)
[17.14 Route Path Parameters – Example](#)
[17.15 Route Query Parameters](#)
[17.16 Route Query Parameters – Example](#)
[17.17 Router - Imperative Navigation](#)
[17.18 Router - Imperative Navigation Methods](#)
[17.19 Router - Imperative Navigation – Example](#)
[17.20 Router – Extracting Data](#)
[17.21 Dynamically Changing Route Configuration](#)
[17.22 Route Guards](#)

17.23 Route Guards – Example

18 Observers, Reactive Programming & RxJS

18.1 Introduction

18.2 Reactive Extensions

18.3 RxJS Libraries

18.4 What Are Asynchronous Data Streams?

18.5 Examples of Asynchronous Data Streams

18.6 Observable Sequences (Observables)

18.7 Observers

18.8 Observers - Example Code

18.9 Subscriptions

18.10 Observables, Observers and Future JavaScript ES7

18.11 Operators

18.12 Operators - Example Code

18.13 Operators that Create Observables

18.14 Operator ‘From’

18.15 Operator ‘Interval’

18.16 Operator ‘Of (Was ‘Just’)

18.17 Operator ‘Range’

18.18 Operator ‘Repeat’

18.19 Operator ‘Timer’

18.20 Operators that Transform Items Emitted by Observables

18.21 Operator ‘Buffer’

18.22 Operator ‘Map’

18.23 Operator ‘Scan’

18.24 Operators that Filter Items Emitted By Observables

18.25 Operator ‘Debounce’

18.26 Operator ‘Debounce’ - Example Code

18.27 Operator ‘Distinct’

18.28 Operator ‘Filter’

18.29 Operator ‘Take’

18.30 Operators that Combine Other Observables

18.31 Share Operator

18.32 Share Operator – Example Code

19 RxJS with Angular

19.1 Introduction

19.2 AngularJS

19.3 Angular Uses Observables Instead of Promises

19.4 Observables and Angular

19.5 Observables and DOM Events

19.6 Observables and DOM Events – Example Code

19.7 Observables and Http Services

20 Http

20.1 Introduction

20.2 Http & Http Methods

20.3 Http Headers

[20.4 Http Body](#)

[20.5 Http – Passing Information](#)

[20.6 REST](#)

[20.7 JSON](#)

[20.8 Http Client](#)

[20.9 Using the Http Client](#)

[20.10 Asynchronous Operations](#)

[20.11 Request Options](#)

[20.12 Http Method ‘Get’](#)

[20.13 Http Method ‘Get’ – Example](#)

[20.14 Http Method ‘Get’ Using Parameters](#)

[20.15 Http Method ‘Get’ Using Parameters – Example](#)

[20.16 Http Method ‘Get’ Using Path Parameters – Example](#)

[20.17 Http Method ‘Post’](#)

[20.18 Http Method ‘Post’ – Example](#)

[20.19 Http Method ‘Put’ Using Path Parameters](#)

[20.20 Http Method ‘Patch’ Using Path Parameters](#)

[20.21 Http Method ‘Delete’ Using Path Parameters](#)

[20.22 Modifying the Server Response](#)

[20.23 Modifying the Server Response - Example](#)

[20.24 Handling an Error Server Response](#)

[20.25 Handling an Error Server Response - Example](#)

[20.26 Asynchronous Pipes](#)

[20.27 Asynchronous Pipes – Example](#)

21 Forms

[21.1 Introduction](#)

[21.2 Two Methods of Writing Forms](#)

[21.3 Form Module](#)

[21.4 Form Model Objects](#)

[21.5 Forms and CSS](#)

[21.6 Template Forms](#)

[21.7 Template Forms – Module Setup](#)

[21.8 Template Forms - Example](#)

[21.9 Template Variables and Data Binding](#)

[21.10 Template Forms and Data Binding - Example](#)

[21.11 Template Forms and CSS – Example](#)

[21.12 Reactive Forms](#)

[21.13 Reactive Forms – Module Setup](#)

[21.14 Reactive Forms – Bind Template to Model](#)

[21.15 Reactive Forms - Example](#)

[21.16 Reactive Forms – Form Builder](#)

[21.17 Reactive Forms - Form Group Nesting](#)

[21.18 Reactive Forms - Form Group Nesting – Example](#)

[21.19 Validators](#)

[21.20 Combining Multiple Validators](#)

[21.21 Custom Validation](#)

21.22 Custom Validation – Example Code

22 Pipes

[22.1 Introduction](#)

[22.2 Angular Pipes](#)

[22.3 Angular Pipes – Example](#)

[22.4 Custom Pipes](#)

[22.5 Generate Custom Pipe using CLI](#)

[22.6 Custom Pipes - Example Code](#)

23 Zones & Change Detection

[23.1 Introduction](#)

[23.2 Change Detection](#)

[23.3 NgZone is Zone.js for Angular](#)

[23.4 How Does NgZone / Zone.js Notice Events?](#)

[23.5 NgZone Is Used by Angular to Notice Events](#)

[23.6 How Does Angular Perform Change Detection?](#)

[23.7 Mutable Objects vs Immutable Objects](#)

[23.8 Using NgZone](#)

[23.9 Example Code – Running Asynchronous Code within Angular Zone.](#)

[23.10 Example Code – Running Asynchronous Code Outside Angular Zone.](#)

24 Testing

[24.1 Introduction](#)

[24.2 Unit Testing](#)

[24.3 Continual Integration](#)

[24.4 Automated Unit Tests](#)

[24.5 Integration Testing](#)

[24.6 Mocks](#)

[24.7 Karma](#)

[24.8 Jasmine](#)

[24.9 Jasmine – Setup and Teardown](#)

[24.10 CLI](#)

[24.11 CLI - Running Unit Tests](#)

[24.12 CLI – Unit Test Files](#)

[24.13 CLI – Dependency Injection](#)

[24.14 Angular Testing Objects](#)

[24.15 Component Fixture](#)

[24.16 CLI Unit Test – Example #1](#)

[24.17 CLI Unit Test – Example #2](#)

[24.18 CLI Unit Test – Example #3](#)

[24.19 Angular Http Testing Objects](#)

[24.20 Angular Http Testing Objects - Example](#)

25 More Advanced Topics

[25.1 View Encapsulation](#)

[25.2 Why is View Encapsulation Required?](#)

[25.3 Shadow DOMs](#)

[25.4 Component Encapsulation](#)

[25.5 ViewEncapsulation.Emulated – Example](#)

[25.6 ViewEncapsulation.Native – Example](#)

[25.7 ViewEncapsulation.None – Example](#)

[25.8 View Encapsulation - Conclusion](#)

[25.9 Styling Content Children](#)

26 Resources

[26.1 Introduction](#)

[26.2 ‘Official Website’](#)

[26.3 GitHub](#)

[26.4 Angular 4-Related Blogs](#)

[26.5 Angular Air](#)

27 Conclusion

1 Acknowledgements and Revisions

1.1 Acknowledgements

First and foremost, thanks to my wife Jill and her patience. I hope she is enjoying herself doing her favorite things like Paddle boarding, Kayaking and being at one with nature. I hope she never reads this book because it would bore her.

Thanks go out to the people publishing blogs and articles to the web, without you I would never have been able to perform as much research as I did.

Even more thanks go out to the people working on Angular, especially those updating the Angular.io website with useful information. It was invaluable to me.

1.2 Revisions

Date	Notes
July 17	Initial version.

2 *Introduction*

2.1 How the Book Started

I never wanted to write a book, I am far too lazy for that. I realized that as a nearly 50 year old developer I was going to go through some pain to learn this new technology. So, I figured that I would write it up for myself in a series of PowerPoints so that it would make it simpler. Thirty BIG PowerPoints later I had a lot of content, so I tried to self-publish. If this book does well I will spend considerable time keeping it up to date. If the book doesn't sell my spare time will probably go somewhere else.

2.2 About the Book

● Disclaimer

Let's get this over with as quickly as possible. I need to mention two things: some of this information in this book may be incorrect (I am a human being that makes mistakes) and that this publication is somewhat opinionated. I am trying my best to be as technically accurate as possible but I am still learning a lot and have much to learn about Angular 4. I do have some strong opinions but please don't take them seriously. I do not intend to harm anything or anyone, I am not smart enough for that.

● Scope

This scope of this book is to help developers get started in Angular 4. You are not going to read and learn ALL there is to know about Angular 4. Knowing everything about Angular 4 is not the purpose of the book, getting up to speed as a developer is. In my opinion 'getting up to speed' means having a good overall knowledge, sufficient to start working.

● Approach

This book takes the approach of chapters with small code examples built with the Angular CLI. You will be able to 'try out' code without being burdened with setting up a large project. I did it this way because I found this format easier to understand than a large project.

● Example Code

The example code will be available here:

<https://github.com/markclow/learn-angular4-fast>

Remember that you will need to do a 'npm install' on each project to install the dependencies and get it working.

Sometimes you may also need to re-install the CLI using the following two commands:

```
npm uninstall --save-dev angular-cli
```

```
npm install --save-dev @angular/cli@latest
```

Sometimes you may get the 'Environment configuration does not contain environmentSource entry.' error. This is fixed by editing the file 'angular-cli.json' and changing the setting from:

```
"environments": {  
  "source": "environments/environment.ts", "dev": "environments/environment.ts", "prod": "environments/environment.prod.ts"  
}
```

to:

```
"environmentSource": "environments/environment.ts", "environments": {  
  "dev": "environments/environment.ts", "prod": "environments/environment.prod.ts"  
}
```

2.3 Angular

The purpose of AngularJS and Angular 4 is to create Single Page Applications. We will soon cover how web applications have evolved from Server-Side Applications to Single Page Applications.

2.4 Angular and Naming

So, Angular gives us a way of writing Single Page Applications. However, there is now more than one version of Angular. As of the time of writing this book we have three versions: The original Angular, which runs on JavaScript.

The second Angular, Angular 2, which runs on TypeScript.

The third Angular, Angular 4, which runs on TypeScript.

Now we have three Angulars, developers have rallied around a newer naming convention, **which I am going to use in this book**: The original Angular should be called AngularJS, as it runs on JavaScript and is very different to the other Angulars.

Angular2, Angular4, Angular 5 and so forth are going to be called just Angular.

2.5 My Opinion as a Developer

I am a developer who is used to using a typed, comprehensive language (Java, .NET C#, VB) on the server, who enjoys the benefits of a compiler. However, I am also someone who has to do

```
package com.mkyong.test.core;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD) //can use in method only.
public @interface Test {

    //should ignore this test?
    public boolean enabled() default true;

}
```

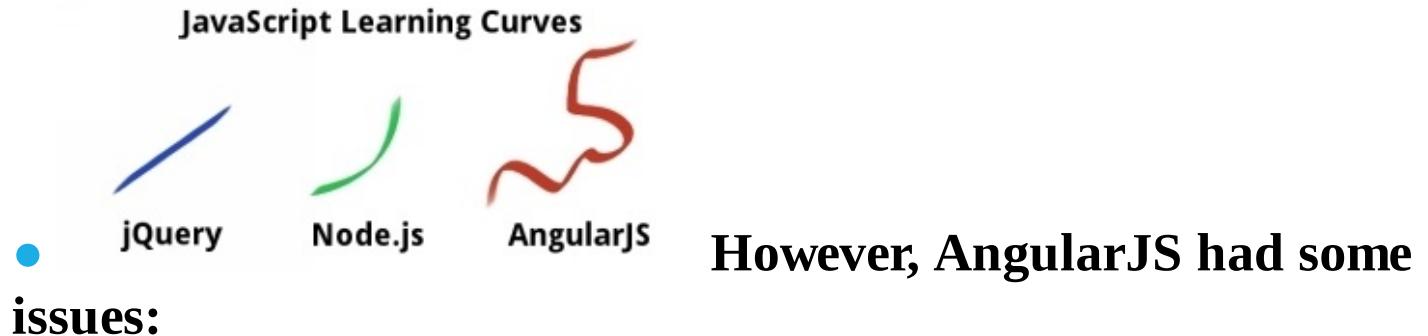
client-side

coding, being a ‘full stack developer’. I

don’t like JavaScript much. I have a long complaint list and you can see it in the JavaScript chapter later on.

However, I have to use JavaScript because that’s what runs on the browsers. I have little choice.

I just need code that runs on the current browsers and I wish there was a structured way of doing it with a proper language. I also liked the original Angular (AngularJS) as you could get stuff running FAST.



It had some strange syntax, (mostly because it used JavaScript, for example IFFE's) and patterns. This caused the learning curve for AngularJS to be somewhat inconsistent. Some parts are easy to learn, others are more difficult.

Most AngularJS developers wanted the Google people to take a step back and re-architect Angular to make it simpler, more logical, and more (acceptable) to developers in general, not just UI guys.

I think they did a tremendous job when they converted AngularJS to Angular.

2.6 So Why Is Angular the Answer?

- **You can use it with typed languages.**

Angular can be used with languages like TypeScript, CoffeeScript. Transpilation (more of that later) is nothing new and has been around before Angular 4 came out.

You can also write Angular code in JavaScript (I wouldn't though). However, I believe it's *easier* to write Angular code in TypeScript because Angular was *written in* TypeScript. TypeScript (like other transpiled languages) gives you the ability to write Angular code in a language similar to Java, .NET C#, VB on the server. You have classes, interfaces, casting etc.! This will catch on because there are hordes of developers who are used to writing code in those languages who can transition over to Angular 4.

- **If you use TypeScript you can use annotations.**

Annotations, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

Having done a lot of Java Spring development with JPA, I am used to using annotations and I am totally at ease with them. All you are doing is basically adding more information about your code. This information is also conveniently located inline inside your code so you can see it.

Annotations have a number of uses, among them:

Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.

Compile-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.

Runtime processing — Some annotations are available to be examined at runtime.

If you decide to stay with JavaScript for your Angular 4 coding, you can say goodbye to annotations.

- **You use dependency injection with dependencies injected through constructors.**

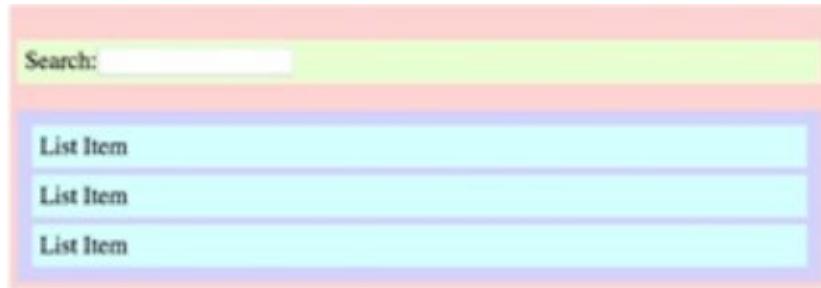
Being a Java Spring guy, I love dependency injection, it makes life simpler. I found the Angular 4 dependency injection similar and a breeze to use. We will go into dependency injection more in Chapter ‘

Dependency Injection'.

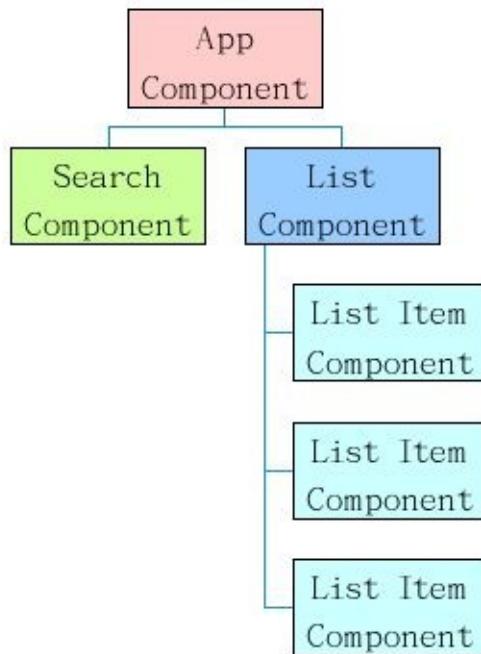
- **You can develop a well-structured, logical user interface.**

Angular user interfaces consist of *Components*. A Component can contain other Components, which can contain other Components. This is known as composition and this can form a complex hierarchy of components. Components can also talk between themselves.

Here is a search user interface. Enter search at top, list items go underneath.



In Angular you could implement this with the following hierarchy of components.



- **You use instance variables bound to the UI (no \$scope).**

The strength of Angular is its binding and that remains. In AngularJS the developer used to bind visual components to variables contained within the scope. Now the developer binds to variables contained within the class. This is similar to how you code a UI in Java or .NET on the server.

2.7 Sounds Good, Doesn't It?

There has to be a catch, correct?

Well there is good news and there is bad news.

● Good News

1. The good news is Angular seems to be the answer for mainstream developers like you and me (i.e. non-UI gurus). It may make our life easier in the long term.
2. Some of the Angular things won't take much learning because you already know them from the server side.
3. The Angular CLI makes code generation a snap.
4. There is plenty of Angular sample code online.

● Bad News

1. There are a lot of new technologies and concepts to learn.

TypeScript Transpilation of TypeScript to JavaScript.

Editing TypeScript.

Creating and Consuming JavaScript Modules.

Deploying Code, including JavaScript Modules.

Angular Components.

Angular Dependency Injection.

Angular and UI Widget Libraries.

Angular Router.

Reactive Extensions.

And More Luckily these things above are covered in upcoming chapters!

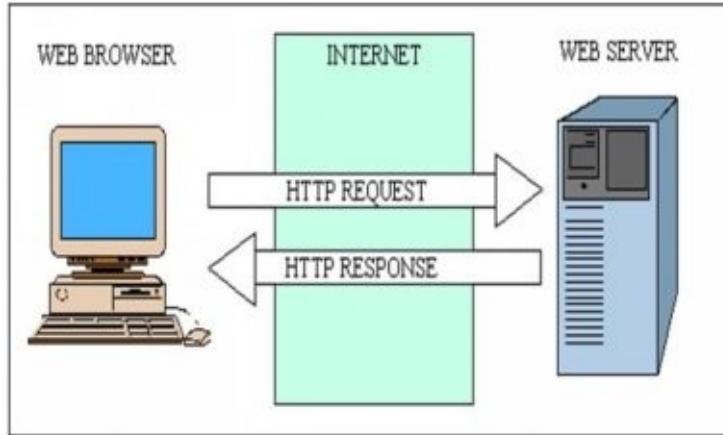
3 *Web Applications and AJAX*

3.1 *Introduction*

Before we dive into Angular, it is better for us to introduce some of the basic concepts of web development. I am sure many of you can just skip over this chapter.

3.2 Introducing the Client & Server

Web applications basically involve two computers communicating with each other: a server and a client.



3.3 Server (Web Server)

The server sits in the company office or data center, listens to http requests and responds back with answers. This server will also access the data (stored in a database) that is used by the web application.

3.4 Client (Web Browser)

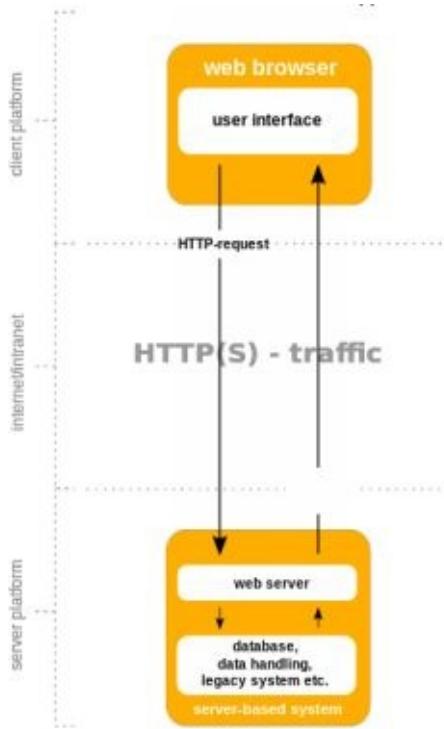
The user sits on their own computer, using the web browser to interact with the web application. This computer communicates with the server, sending http requests and receiving answers when needed. Client computers can be a variety of machines, from iwatches to cell-phones to tablets to computers!

3.5 Communication (Usually Performed with HTTP)

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers. HTTP works as a request-response protocol between a client and server. We will cover this in more detail in this chapter: [HTTP](#).

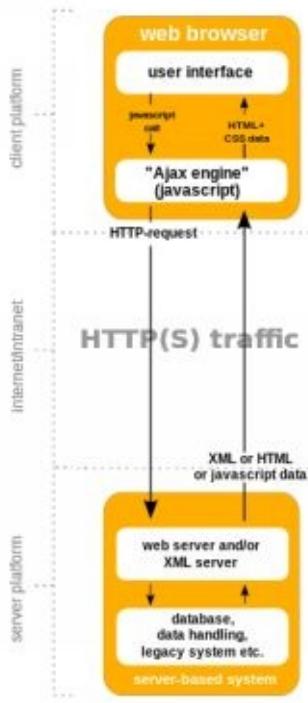
3.6 Server-Side Web Application

A server-side web application is one where more of the application executes on the server and the client is only used to display html pages one at a time. When the user performs an action in the Web Application, the client sends a request to the server, which does something and returns a brand-new html page to be displayed on the client as a response. The Web Page is regenerated every time and sent back to be displayed on the Client's web browser.



3.7 Client-Side Web Application (or Single Page Web Application)

Client-side web applications (also known as Single Page Apps) are more modern and the computing industry is moving more towards this model. In this model, a lot of the application executes on the server but also some code executes on the client (the web browser), to avoid the frequent regeneration of pages. When the user performs an action in the client, it sends a request to the server, which does something and returns information about the result, not an entirely new html page. The client-side code listens for an answer from the server and itself decides what to do as a response without generating a new page. Client-side web applications tend to be more interactive and flexible because they can respond more quickly to user interactions. They can respond more quickly to interactions because they don't have to wait on the server to send back as much data. They only need to wait for the server to respond back with a result, rather than a whole html page.



3.8 The Balancing Act

So, you basically have two types of web applications: server-side and client side (SPA): black and white.

The reality is that your web application should be in the middle, in the ‘grey’ area.

The server-side should remain the repository for the ‘clever stuff’. The business rules, data storage, settings should remain on the server and be invoked or retrieved from the client-side when required.

The client-side (browser) should use the modern client-side technology to avoid full-page refreshes. However, it should not be ‘too smart’ or ‘too bloated’. It should know enough to do its job of interacting with the user and nothing more. It should invoke code on the server-side to do smart things or perform business processes. It should not have too much ‘hardcoded’ information also – that is better managed on the server.

You must avoid throwing ‘everything but the kitchen sink’ into the client.

3.9 AJAX

● Introduction

AJAX stands for Asynchronous JavaScript and XML. AJAX is a new technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and JavaScript.

When a client-side web application needs to communicate with the server, it uses AJAX to send something out and waits for the result to come back. Remember it gets back a result that only contains data, not an entirely new web page. Also the client-side code does not stop running while it is waiting, as it still has to display the user interface and respond to the user. This is the asynchronous part of AJAX.

Client-side web applications use JavaScript to invoke the AJAX request and to respond to it. This is the JavaScript part of AJAX.

AJAX requests used to use the XML language as the data format for the request and result data going forth between the client and the server. Both XML and JSON are commonly-used formats for transferring data in text form. Nowadays AJAX tends to use JSON as the data format instead of XML. This is because JSON is much more compact and maps more directly onto the data structures used in modern programming languages.

● Metaphor for AJAX Communication

● Scenario

You call your wife to ask her to do something for you. Her phone is busy and you leave her a message asking her to stop at the supermarket and buy you a case of beer. In the meantime, you keep watching TV.

● Outcomes

Success: She calls you back and tells you the beer is on its way.

Failure: She calls you back and tells you the store was closed.

3.10 Callbacks

We introduced the asynchronous nature of AJAX above, how the client-side code does not stop running while it is waiting for a response from the server. Typically, when you make an AJAX call you have to tell it what to do when the server response is received. This is the code that the AJAX system code should fire when the response is received. This is often known as the ‘callback’.

- **Two Types of Callbacks**

When you perform AJAX operations you invoke the AJAX code with parameters and one or two functions – the callbacks.

- **Success**

One of the callbacks is invoked if the server responds successfully and the client receives the answer without error (the ‘done’ or ‘success’ callback).

- **Failure**

Another of the callbacks is optional and is invoked if the server responds back with an error (or the AJAX call cannot communicate with the server). This is also known as the ‘fail’ or ‘error’ callback.

3.11 Promises

Sometimes you invoke AJAX code and it returns what's known as a 'Promise' or a 'Deferred'. This is an object that is created that is a 'promise of response' from an AJAX operation. When you receive such a 'Promise', you can register your 'Success' or 'Failure' callbacks with the Promise. This enables the Promise to invoke the callback once a success or failure occurs.

3.12 Encoding

When you work with AJAX (or even normal communication between Client and Server), you need to ensure that the information is sent in a form in which is suitable for transmission. If you don't use encoding, it is quite possible that some the information is not received exactly as it was sent. This is especially true for some of the special character information being sent, for example spaces, quotes etc.

Here are three main methods to encode information:

Method	Notes
encodeURI	<p>This is useful for encoding entire urls into UTF-8 with escape sequences for special characters. It encodes the string in the same manner as 'encodeURIComponent', except that it doesn't touch characters that make up the url path (such as slashes).</p> <p>Example: <i>http://www.cnn.com gets converted to http://www.cnn.com%0A</i></p>
encodeURIComponent	<p>This is useful for encoding parameters. It is not suitable for encoding entire URLs because it can replace important URL path information with escape sequences.</p> <p>Example: <i>http://www.cnn.com gets converted to http%3A%2F%2Fwww.cnn.com%0A</i></p>
escape	<p>This returns a string value (in Unicode format) that contains the contents of [the argument]. Take care as servers do not expect to receive data in Unicode format by default.</p> <p>Example: <i>http://www.cnn.com gets converted to http%3A//www.cnn.com%0A</i></p>

To test these methods, head over to this webpage:

HTTP://PRESSBIN.COM/TOOLS/URLENCODE_URLDECODE/

**URL-encode and URL-decode text strings
as you type using PHP and Javascript functions**

hi there, how ya doin'?

URL-encode

URL-decode

urlencode()

hi+there%2C+how+ya+doin%27%3F

encodeURIComponent()

hi%20there%2C%20how%20ya%20doin%3F

encodeURI()

hi%20there,%20how%20ya%20doin%3F

escape()

hi%20there%2C%20how%20ya%20doin%27%3F

3.13 HATEOAS & HAL

To talk with the server, the client needs to know to which URLs that the server is available on. This information should not be hardcoded on the client. The server should tell the client what URLs to use to get information. There are various standards as to the format of sending this information back to the client, including HAL.

For example, if the client sends an AJAX request to the server to retrieve a list of customers, the information returned should include the URLs for the AJAX requests for each customer. This avoids hardcoding the customer AJAX request URL on the client.

Here are some articles on this subject:

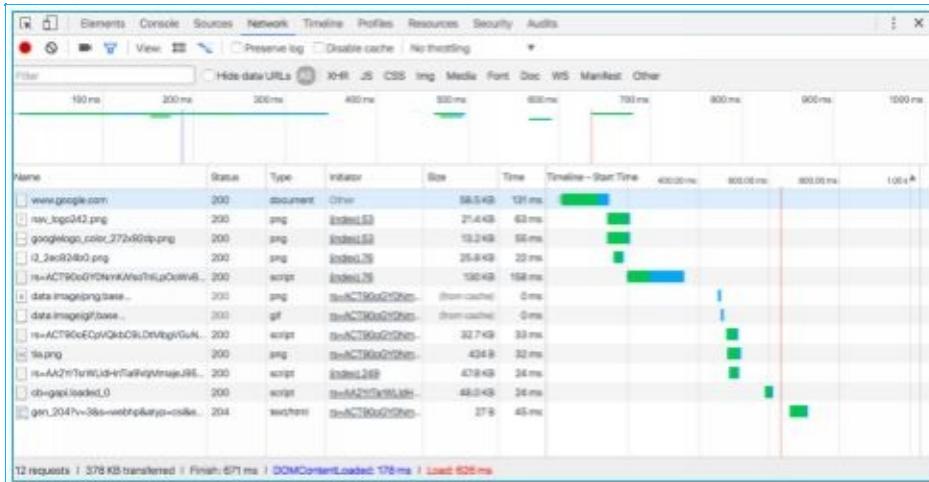
<https://martinfowler.com/articles/richardsonMaturityModel.html>

<https://en.wikipedia.org/wiki/HATEOAS>

3.14 Debugging Tools

● Developer Tools

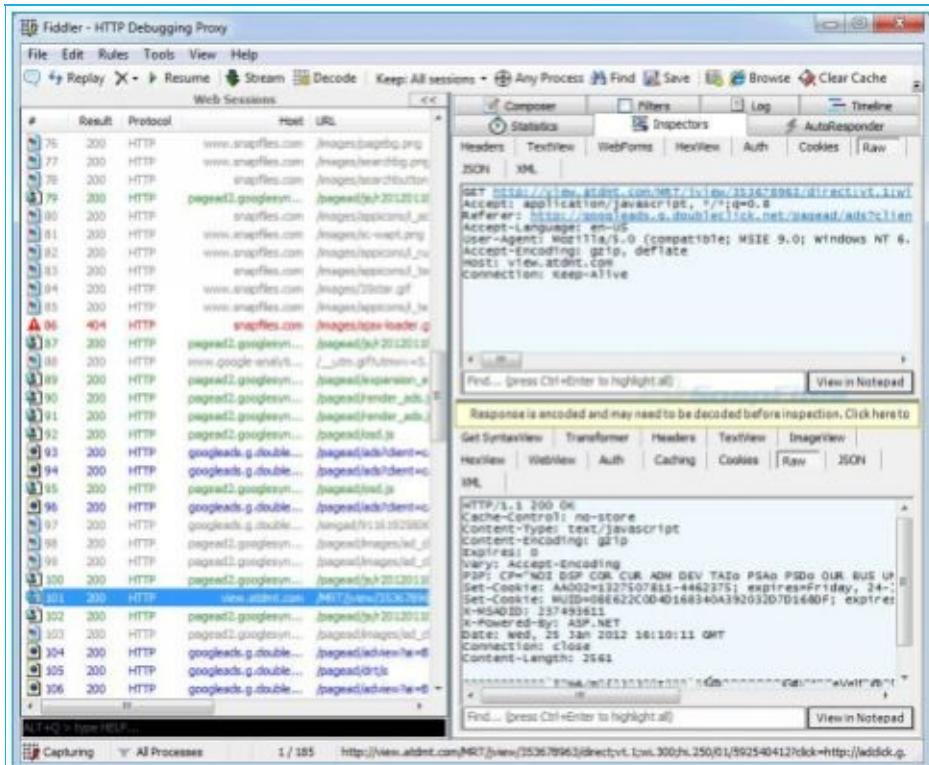
Your web browser has developer tools built in. One of them is the ‘network’ tool that allows you to monitor data traffic between the client and the server. This data traffic is presented as a list with a timeline. You can select an item on the list to view it in more detail to see exactly what data was sent to the server and what data came back.



● Fiddler

Fiddler is similar to the network tab in your developer tools in your web browser. However, it has some extra capabilities, such as creating your own AJAX requests and running scripts.

[HTTP://WWW.TELERIK.COM/FIDDLER](http://www.telerik.com/fiddler)



3.15 Analyzing JSON

You will often receive long JSON responses from the server and you will often need to traverse this response data to extract just the data you need. Your response data will normally be passed to your AJAX ‘success callback’ as an argument. Here are some tips on examining this data.

• Convert it to a String

You can call the ‘`JSON.stringify`’ function to convert this data into a string. This will enable you to output it to the console in your ‘success callback’:

```
function success(data){  
  console.log('success - data:' + JSON.stringify(data)); //  
  // do something with data  
  //  
}
```

- Copy the JSON Data Out of the Console

To copy this JSON data into your clipboard, do the following:

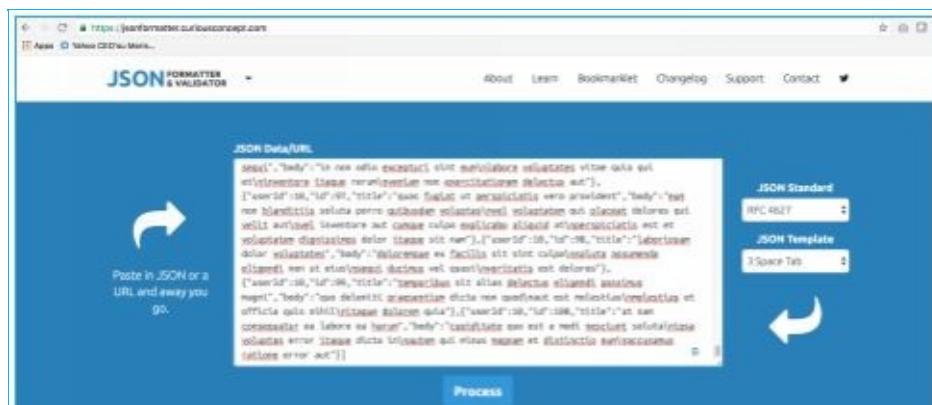
1. Select your browser.
 2. Go to ‘developer tools’.
 3. Select ‘console’.
 4. Select JSON text.
 5. Right-click and select ‘copy’.

- Format the JSON Data to Make It More Readable

Now you have the JSON data in your clipboard, you can copy and paste it into a website to make it more readable:

1. Select your browser.
 2. Go to website below (or similar, there are lots!).

<https://jsonformatter.curiousconcept.com/>



- Paste the JSON into the big textbox.
 - Click on the ‘Process’ button.

5. The website will show you the JSON data in a validated, formatted, easy to read webpage. You can even view the JSON full-screen!

The screenshot shows a browser window for the JSON Formatter & Validator website. At the top, there's a navigation bar with links for About, Team, Bookmarklet, Changelog, Support, and Contact. Below the navigation is a banner for a free O'Reilly book titled "Graph Databases". The main content area has a header "#1 September 5th 2016, 13:11:47 pm". A red bar at the top of the content area says "INVALID JSON (RFC 4627)". Below this, under "Validator Output", there are three error messages:

- Error: Expecting object or array, not string. [Line 1, column 1]
- Error: Strings should be wrapped in double quotes. [Line 17, column 5]
- Error: Strings should be wrapped in double quotes. [Line 17, column 6]

Under "Formatted JSON Data", the JSON code is displayed with syntax highlighting. The code consists of two arrays, each containing two objects. The objects have properties: "userId", "id", "title", and "body". The "body" property contains placeholder Latin text.

```
["user":1,"id":1,"title":"eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat."], [{"user":2,"id":2,"title":"dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat."}]
```


4 AngularJS vs Angular (Old vs New)

4.1 Introduction

Before we learn Angular, we need to introduce the original Angular (AngularJS) and talk about the most important differences between the first Angular and the later ones (i.e. Angular2, Angular 4 etc. The convention is simply to call these Angular).

4.2 What is AngularJS?

It's the original Angular.

It's a JavaScript framework for dynamic web applications, no page reloads required. Also known as SPA: Single Page Application.

It's popular for creating web pages with widgets that work fast on any browser.

It allows users to extend HTML to add domain-specific tags.

E.g. <CAR> It allows users to bind data from the model to the HTML / domain-specific tags.

4.3 Why Was AngularJS so Popular?

It took off like wildfire because it was a great tool for prototyping applications **quickly**. It was also flexible in that you could use the html for a page and build quickly on it, turning it from static html into a moving, responsive sexy app.

- **Converting HTML into a Working Application in AngularJS**

1. Take an html template and modify some of the html code elements to add data binding.

Binding.

In case you don't know, data binding allows a visual control (text box, select box etc) to have its value synchronized with a variable. For example, you could have a 'city' variable that is bound to a 'city' text box. If the user types into the 'city' text box, the value of the 'city' variable is updated. If the code changes the value of the 'city' variable, the 'city' text box is updated to match.

2. Add a JavaScript Angular controller containing:

1. Add the variables for the html markup to be bound to.
2. Add behavioral JavaScript code (button clicks etc).

Done!

You could (obviously) do a lot more but the point is that developers could get *quick* results turning raw Html into a working, responsive application.

4.4 When Were AngularJS and Angular born?

AngularJS was released in 2009.

Google announced development of Angular 4 on September 2014 and it went into Beta January 2015.

In between 2009 and 2014, many things have changed in the world of web development and this has affected the development of Angular 2.

Angular 4 was released in March 2017.

4.5 Browsers

AngularJS runs on web browsers and web browsers run JavaScript. So, JavaScript is the platform for AngularJS and Angular.

● Browsers Now Upgrade Themselves

The term "evergreen browser" refers to browsers that are automatically upgraded to future versions, rather than being updated by distribution of new versions from the manufacturer, as was the case with older browsers. The term is a reflection on how the design and delivery of browsers have changed quickly over the last few years. Now all of the widely-used browsers are evergreen and update themselves.

● Browsers Run JavaScript Using JavaScript Engines

We used to think of a web browser and its ability to run JavaScript as the same thing. Since Node (which uses the JavaScript engine from Google Chrome to run programs away from the browser), this has changed and you can run these engines standalone away from the browser.

A JavaScript engine is a program or interpreter which executes JavaScript code. A JavaScript engine may be a traditional interpreter, or it may utilize just-in-time compilation to bytecode in some manner. Although there are several uses for a JavaScript engine, it is most commonly used in Web browsers.

Since AngularJS, JavaScript engines have steadily improved with new versions of ECMA JavaScript. This 'ECMA' stuff means 'version'. AngularJS ran on web browsers running version of JavaScript called ECMA5. Now most browsers are running a later version. You could write a book about how ECMA has evolved since version 5. With ECMA6 (also known as ECMA 2016), JavaScript took a giant leap towards becoming a structured, typed language like Java or .NET. The two more important changes are the new syntaxes for creating classes and modules. Super-important and relevant for this book.

● Shims and Polyfills

Shims and Polyfills are software components designed to allow older browsers to run more modern code. A shim is a piece of code that intercepts existing API calls on a browser and implements different behavior. This is for standardizing APIs across different environments. So, if two browsers implement the same API differently, you could use a shim to intercept the API calls in one of those browsers and make its behavior align with the other browser. A polyfill is a piece of JavaScript that can 'implant' missing APIs into an older browser. For example, shims and polyfills enable older ECMA5 browsers to run ECMA6 code.

● The World of ECMA JavaScript Changes Quickly

As you may know, the world of client-side JavaScript changes quickly. It is better for this book to pass you to a web page that is regularly updated with the latest information:

<https://en.wikipedia.org/wiki/ECMAScript>

4.6 TypeScript

In between AngularJS and Angular, JavaScript has improved and has become more of a structured language. However, you can take things even further for and use the TypeScript language, which is structured and even more like languages like Java & .NET C#. In fact, TypeScript was developed by Microsoft to be an improved successor to JavaScript so you can see where the similarities came from.

TypeScript = ECMAScript + Types + Annotations

Google developed Angular using TypeScript. Angular and the TypeScript language are a great combination so this is what we are going to learn in this book.

• Transpilation

So how does TypeScript run on a web browser?

It doesn't (not at the moment). It gets converted back to compatible JavaScript using a process called Transpilation. A transpile is a piece of software that converts the source code of one language into the source code of another. For example: TypeScript, CoffeeScript, Caffeine, Kaffeine and more than two dozen other languages are transpiled into JavaScript.

If you want to see transpilation first-hand, go over to the web page

<https://www.typescriptlang.org/play/> and view some of the examples. If you select 'Using Classes' you can see how a modern TypeScript class is converted down to compatible JavaScript:

TypeScript Class

This would be the code you write.

```
class Greeter {  
    greeting: string;  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}
```

Transpiled to Browser-Compatible JavaScript

This would be the code the browsers run when you deploy.

```
var Greeter = (function () {  
    function Greeter(message) {  
        this.greeting = message;  
    }  
    Greeter.prototype.greet = function () {  
        return "Hello, " + this.greeting;  
    };  
    return Greeter;  
}());
```

● Debugging and Map Files

So, you are writing code one way and deploying it in another, correct? That must be a nightmare to debug.

Yes, it would be a nightmare to debug if you didn't have Map files. They are automatically generated by your transpiler and give the browser the information it needs to map the original (TypeScript) code to the deployed (JavaScript) code. That means the JavaScript debugger can let you debug your source code as if the browser was running it. How cool is that! And if you enable '.map' files on your browser it will automatically look for them, pick them up and use them. I use '.map' files all the time when I am debugging in Chrome.

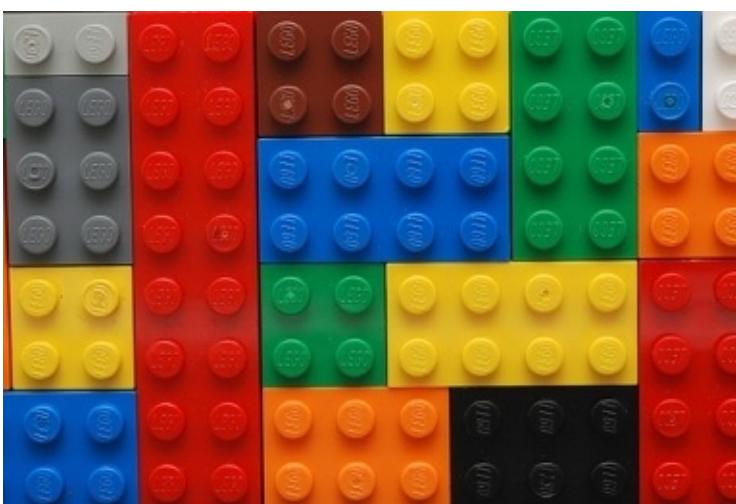
Map files:

1. Map a combined/minified/transpiled file back to an unbuilt state.
2. Map JavaScript lines of code in the browser back to the TypeScript lines of code.
3. Enable browsers / debuggers to show the original code that you wrote in TypeScript and debug it.

● Transpilation and Your Project Build

There are many ways of setting up your project to transpile your TypeScript code into browser-friendly JavaScript. It all depends on your project setup. You have many options in this regard and it can get complicated and confusing.

I recommend that you get started using the Angular CLI tool. It can very simply generate ready-made projects that have a simple build process setup, including transpilation. It can also be of use in larger projects.



4.7

Modules

The word module refers to small units of independent, reusable software code, for example code to perform animations. I think of modules like Lego blocks. Each block has its own purpose but it is plugged into a larger structure (the application).

AngularJS had its own module system which was simple to use. At that time JavaScript did not have its own system of modularizing code.

Angular has its own module system to package Angular code into modules, as well as modern JavaScript modules.

Modules are like software lego blocks.

This will all be covered later, don't worry!

4.8 Controllers and Components

AngularJS used Controllers to represent a widget in the user interface on the Html page.

Angular 4 replaces Controllers with the Component object.

Components can have their own tag. E.g. <Component1>.

Components have a class which contains data and code.

We will cover components in greater detail (see Chapter ‘**Error! Reference source not found.**’) as they are literally the building blocks of Angular 4 applications.

4.9 Dependency Injection

As I mentioned earlier, being a Java Spring guy, I love dependency injection, it makes life simpler. We could spend pages and pages on this subject but I am sure that you already know what this term means, and the benefits that dependency injection provides.

AngularJS provided dependency injection.

Angular 4 also provides dependency injection also. Because your components have classes, dependencies are now usually injected via the constructor, using the Constructor Injection pattern.

● Constructor Injection

This software pattern is known as *constructor injection* and it is another server-side technology that is now being used on client-side. In case you don't know here is my explanation.

Here is an example of Java Spring using Constructor Injection.

The configuration below specifies a constructor argument, a string message 'Spring is fun' (in red).

```
<?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans.xsd">
    <bean id="message"
        class="org.springframeworkdi.xml.ConstructorMessage"> <constructor-arg value="Spring is fun." /> </bean>

</beans>
```

The bean class below expects to receive the message in the constructor.

```
public class ConstructorMessage {

    private String message = null;

    /**
     * Constructor
     */
    public ConstructorMessage(String message) {
        this.message = message;
    }

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }
}
```

What's so great about this?

In this case, it is a simple example of a string. However, it shows how a software object (in this case a string) is ‘plugged into’ another software object using the constructor.

For example, in Angular you could create a software object that handles communication with your server. Then you can pass it into every object (class) that needs it by passing it in through the constructor. Then in the class you have a ready-made way of talking to the server.

Write a service once, use it many times in many places.

4.10 Scope

In AngularJS, Scope (\$scope) used to be the ‘data container’ for the controller. Your variables would be contained in the \$scope object. For example, if you had a controller for an input form for an address, each line of the address would probably be a variable inside the \$scope for the Controller.

In Angular you no longer have Controllers you have Components. Components have a class, similar to those in Java or .NET. The class is the ‘data container’ and contains your variables. This is far more like conventional server-side coding. For example, if you have a Component for an input form with an address, each line of the address would probably be a variable inside Component’s class, similar to a Java Swing (or Windows Form) class.

4.11 Forms

Writing code that handles data input on forms is obviously important. It was easy to write AngularJS code that worked with forms, data input and validation. However Angular has new forms modules that make it easier to:

- Create forms dynamically.

- Validate input with common validators (required).

- Validate input with custom validators.

- Test forms.

4.12 Templates

AngularJS and Angular both use html templates. The html in the template is bound to the data variables and the code in order to make a working application.

Unfortunately, the template syntax has diverged. We will cover the new syntax in detail in Chapter ‘More Component’.

1	<input ng-model="thing.item" type="text">
2	<button ng-click="thing.submit(item)" type="submit">

AngularJS Template

1	<input #item type="text">
2	<button (click)="submit(item)" type="submit">

Angular Template

4.13 Conclusion

AngularJS took off like wildfire because it was a quick way to write cross-browser applications. However, it had some inconsistencies and needed updating to use capabilities offered by the updated browsers.

The more modern Angular is like AngularJS, only with a more straightforward development environment, backed up with the ability to work with newer JavaScript and TypeScript.

You develop the user interface in building blocks called Components.

Components use classes to contain their variables and application code.

Classes have instance variables, constructors and methods.

You can inject dependencies into your classes using the constructor.

Instance variables can be bound to the template to create a responsive user interface.

5 *JavaScript*

5.1 *JavaScript History*

When Netscape hired Brendan Eich in April 1995, he was told that he had 10 days to create and produce a working prototype of a programming language that would run in Netscape's browser.

10 days!!!! I would say he did a pretty good job considering the amount of time he was given.

JavaScript is continually evolving. Currently most web browsers support JavaScript ES5 but ES6 will become the norm within the next year or two.

5.2 JavaScript Shortcomings (< ES6)

This is a list of the currently-perceived shortcomings in JavaScript up to and including the current version ES5. Many of these shortcomings have been addressed in ES6, which will be covered shortly.

• Types

In programming, data types is an important concept. To be able to operate on variables, it is important to know something about the type. Without data types, a computer cannot safely solve this: `var foo = 123 + "Mark";`

What's the answer?

123Mark?

Or should it throw an error because 123 is a number and “Mark” is a string?

JavaScript supports only 6 types: Undefined, Null, Boolean, String, Number and Object. Only 1 Number type!!! There are so many types of numbers: integers, decimals, etc. I don't think I am stretching things when I write that in terms of Types, JavaScript does not ‘cut it’.

• Fail Fast Behavior

Code should either work accurately or it should fail immediately (fast). Because JavaScript has fewer types and rules it quite often continues rather than fails, with strange side effects. Things you don't think that are going to work DO work.

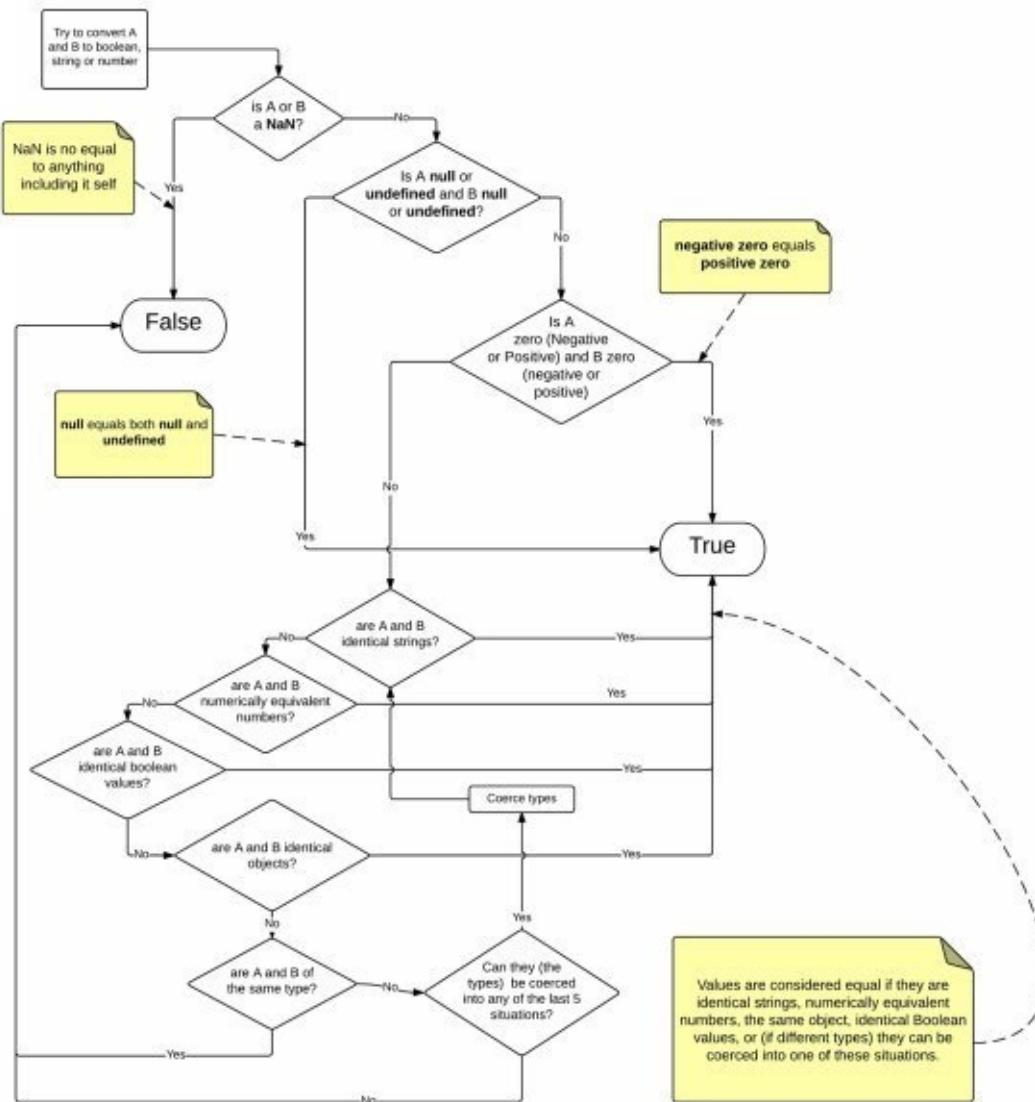
For example, the code below *doesn't* fail: `alert(([![]+[]])[+[]]+([![]+[]])[+!+[[]]+([![]]+[])[[[[]]])[+!+[[]]+[+[]]]+([![]+[]])[!+[]]+![[]]);`

• Value/Object Comparison

When you compare two variables in Java or a .NET language, you don't need to be a rocket-science to figure out how it is going to compare them. You implement a ‘`.equals()`’ method. However, because JavaScript has few types it has this complicated logic used to compare values or objects. To see how JavaScript compares variables, take a look at the equality algorithm below. Remember to take a Tylenol first!

Equality Operator '=='

for example `A == B`



The Abstract Equality Comparison Algorithm

The comparison `x == y`, where `x` and `y` are values, produces true or false. Such a comparison is performed as follows:

1. If `Type(x)` is the same as `Type(y)`, then
 1. If `Type(x)` is `Undefined`, return true.
 2. If `Type(x)` is `Null`, return true.
 3. If `Type(x)` is `Number`, then
 1. If `x` is `NaN`, return false.
 2. If `y` is `NaN`, return false.
 3. If `x` is the same `Number` value as `y`, return true.
 4. If `x` is `+0` and `y` is `-0`, return true.
 5. If `x` is `-0` and `y` is `+0`, return true.
 6. Return false.
 4. If `Type(x)` is `String`, then return true if `x` and `y` are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, [return false](#).
 5. If `Type(x)` is `Boolean`, return true if `x` and `y` are both true or both false. Otherwise, return false.
 6. Return true if `x` and `y` refer to the same object. Otherwise, return false.
2. If `x` is `null` and `y` is `undefined`, return true.
3. If `x` is `undefined` and `y` is `null`, return true.
4. If `Type(x)` is `Number` and `Type(y)` is `String`, return the result of the comparison `x == ToNumber(y)`.
5. If `Type(x)` is `String` and `Type(y)` is `Number`, return the result of the comparison `ToNumber(x) == y`.
6. If `Type(x)` is `Boolean`, return the result of the comparison `ToNumber(x) == y`.
7. If `Type(y)` is `Boolean`, return the result of the comparison `x == ToNumber(y)`.
8. If `Type(x)` is either `String` or `Number` and `Type(y)` is `Object`, return the result of the comparison `x == ToPrimitive(y)`.
9. If `Type(x)` is `Object` and `Type(y)` is either `String` or `Number`, return the result of the comparison `ToPrimitive(x) == y`.
10. Return false.



Abstract Equality Comparison Algorithm 1

Scoping

```
function a(){
  foo1 = 'hello';
}

function b(){
  var foo2 = 'there';
}

a();
b();

alert(foo1);
alert(foo2);
```

In JavaScript, undeclared variables are promoted implicitly to global variables. To me that seems illogical and dangerous (just my opinion), surely to have a global variable you should declare it such?

In the code to the side, variable foo1 is a global variable, variable foo2 is not.

When this code runs you only see one alert box “hello”. You don’t see the second one because foo2 is not set (as it went out of scope and is not a global variable).

5.3 JavaScript Strict Mode

● Introduction

JavaScript Strict Mode was released in ES5. It does not affect old code, in other words using the strict mode command will not break JavaScript code if run in ES4. This mode is intended to prevent unexpected error by enforcing better programming practices.

● Invocation

The "use strict" directive is only recognized at the beginning of a script or a function. This mode can run in two different scopes: file and function. If you place this directive at the beginning of your script file, all of the code in that file will be run in that mode. If you place this directive at the beginning of your function, all of the code in the function will be run in that mode.

We are not going to cover every aspect of Strict Mode but we are going to cover the main ones below.

● Assigning to an Undeclared Variable or Object

Strict Mode throws an error when the user assigns a value to an unassigned variable or object, preventing the creation of an unintended global variable. There is more of this subject later on in this chapter.

Example Code That Throws an Error in Strict Mode

```
"use strict";
pie = 3.14;

"use strict";
obj = {str:10, zip:30350};
```

● Deleting Variables or Objects

Strict Mode does not allow you to use the 'delete' keyword to delete variables or objects.

Example Code That Throws an Error in Strict Mode

```
"use strict";
var pie = 3.14;
delete pie;
```

● Duplicating Function Arguments

Strict Mode does not allow a function to have more than one argument of the same name in a function.

Example Code That Throws an Error in Strict Mode

```
"use strict";
function concat(word1, word1) {};
```

● Duplicating Object Properties

Strict Mode does not allow a function to have more than one property of the same name in an object.

Example Code That Throws an Error in Strict Mode

```
"use strict";
var obj = {
prop1 : 0,
prop2 : 1,
prop1 : 2
};
```

● Read Only Properties

In ES5, users can define object properties using the function ‘Object.defineProperties’. This function allows the developer to define some properties as non-writeable (i.e. read-only). In normal mode, the code does not throw an error when the code attempts to write to a read-only property. In Strict Mode, the code throws an error in this circumstance.

Example Code That Throws an Error in Strict Mode

```
var obj = Object.defineProperties({}, {
  prop1 : {
    value : 1,
    writable : false
  }
});

obj.prop1 = 2;
```

● Non-Extensible Variables or Objects

In ES5, users can use the function ‘Object.preventExtensions’ to prevent objects from being extended.). In normal mode, the code does not throw an error when the code attempts to extend an object. In Strict Mode, the code throws an error in this circumstance.

Example Code That Throws an Error in Strict Mode

```
"use strict";
var obj = {prop1 : 1};
Object.preventExtensions(obj);
obj.prop2 = 2;
```

● Keywords

Strict Mode has introduced these additional reserved keywords that cannot be used in your code in this mode.

implements interface let package private protected public static yield

5.4 JavaScript (>= ES6)

● Introduction

JavaScript ES6 is much improved over ES5. We are not going to cover all of the improvements between ES5 and ES6, just the major ones. Covering all the improvements would take several chapters! Note that if you want to play around with ES6 and you are not sure what to do, visit www.es6fiddle.net and it out.

● Constants

These are for variables that cannot be re-assigned new values.

```
const TAX = 0.06;
```

● Block Scoped Variables and Functions

Before ES6, JavaScript had two big pitfalls with variables.

1. Assigning to an Undeclared Variable Makes It Global

In JavaScript, undeclared variables are promoted implicitly to global variables. As I mentioned before: to me that seems illogical and dangerous (just my opinion). The ‘strict mode’ in JavaScript throws an error if the script attempts to assign to an undeclared variable, for example.

```
"use strict";
mark = true; // no 'var mark' to be found anywhere....
```

2. Declaring a Variable Using the ‘var’ Statement Made It Scoped to the Nearest Whole Function

When you declare variables with the ‘var’ statement, this scopes the variables to the nearest whole function. The example below has two ‘x’ variables assigned, one inside the function but outside the ‘if’ block, another inside the function and inside the ‘if’ block. Notice how the code runs as if there is only one ‘x’ variable. This is because it is scoped to the entire function. It retains the same value even if it leaves the scope of the ‘if’ statement.

```
function varTest() {
var x = 31;
if (true) {
    var x = 71; // same variable!
    console.log(x); // 71
}
console.log(x); // 71
}
```

Now ES6 Allows Developers to Declare Variables and Functions within Block Scope

ES6 has a new ‘let’ statement that is used to declare variables. It is similar to the ‘var’ statement except that the variable is scoped to the nearest enclosing block i.e. ‘{’ and ‘}’.

The example below shows how the *inner* variable ‘x’ is scoped to the nearest block in the ‘if’ statement. When the code exits the ‘if’ statement the *inner* ‘x’ variable goes out of scope. Thus

when the console log is printed in the statement below the ‘if’, it shows the value of the *outer* ‘x’ variable instead.

```
function letTest() {  
let x = 31;  
if (true) {  
    let x = 71; // different variable  
    console.log(x); // 71  
}  
console.log(x); // 31  
}
```

ES6 also allows us to define functions that are scoped within a block. These functions go out of scope immediately when the block terminates. For example, the code below works fine on Plunker with ES5 but throws ‘Uncaught ReferenceError: log is not defined’ when run on Es6fiddle.net.

```
if (1 == 1){  
    function log(){  
        console.log("logging");  
    }  
    log();  
}  
log();
```

● Arrow Functions

Arrow functions are a new ES6 syntax for writing JavaScript functions. An arrow function is an anonymous function that you can write inline in your source code (usually to pass in to another function). You don’t need to declare arrow functions by using the ‘function’ keyword. One very important thing about arrow functions is that the value of the ‘this’ variable is preserved inside the function.

```
// ES5  
var multiply = function(x, y) {  
    return x * y;  
};  
  
// ES6  
var multiply = (x, y) => { return x * y };|
```

● Functions Arguments Can Now Have Default Values

You can specify default values in case some of the arguments are undefined.

Example

```
function multiply(a = 10, b = 20){  
return a * b;  
}  
console.log(multiply(1,2));  
console.log(multiply(1));  
console.log(multiply());
```

Output

• Functions Now Accept Rest Parameters

This parameter syntax enables us to represent an indefinite number of arguments as an array.

Example

```
function multiply(...a){  
var result = 1;  
for (let arg in a){  
    result = result * a[arg];  
}  
return result;  
}  
console.log(multiply(5,6));  
console.log(multiply(5,6,2));
```

Output

```
30  
60
```

• String Interpolation

This enables variables to be data-bound into text-strings. Please note that the interpolation only works with the new quote character ` used for template literals. Template literals allow the user to use multi-line strings and string interpolation. String interpolation does not work with strings enclosed in the existing quotes " and '.

Example

```
var person = {name: "julie", city: "atlanta"}; console.log(person.name);  
// works  
console.log(` ${person.name} lives in ${person.city}`); // doesnt work  
console.log(`${person.name} lives in ${person.city}`); console.log('${person.name} lives in ${person.city}'');
```

Output

```
julie  
julie lives in atlanta  
${person.name} lives in ${person.city}  
${person.name} lives in ${person.city}
```

5.5 Modules

● Definition

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

Currently most web browsers run JavaScript version ECMA 5. ECMA5 was not written to work with modular programming. However, ECMA6 is designed to work with modules and its specification was agreed on June 2015.

● Lego Blocks

ES6 JavaScript lets you write your code into modules and use them like Lego blocks.

For example, you could have an Internationalization Utility module that contains a lot of code for internationalization, including code to load resource bundles for different locales etc. However you only need other code to access one method of this code, a method called ‘getI18N(locale, key)’, which would return text for a locale and a key. JavaScript modules give you the ability to do that, letting you code a ‘black box’ of code that is only accessible through public interfaces (in this case an exported function).

● One File

In ES6, you write each module in its own JavaScript file. One ‘.js’ file. There is exactly one module per file and one file per module. You have two ways of exporting things from a module to make them usable from the outside. Both ways of exporting use the ‘export’ keyword. You can mix the two ways of exporting things in the same module but it is simpler if you don’t – just pick one. Once you have exported code, you can use it elsewhere by importing it.

● Exporting Method #1 – Named Exports

If you want your module to export more than one thing (for example constant, function, object...), use named imports. These enable you to export code with names.

Module ‘mymath.js’

```
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}
```

Importing and using module code:

```
import { square, diag } from 'mymath'; console.log(square(11));
console.log(diag(4, 3));
```

Note how the export doesn't need a semi-colon. Note the names must match the original names in the module.

● Exporting Method #2 – Default Exports

Each module can only have one default export. This is useful if you only want your module to export one thing.

Module ‘mymath.js’

```
export default function square(x) {  
    return x * x;  
}
```

Importing and using module code:

```
import sq from 'mymath';  
sq();
```

Note how the export doesn't need a semi-colon. Note how the name ‘sq’ does not match the function in the module. Using default exports allows you to use ‘nicknames’ (as it knows the object it is going to use as there is only one).

5.6 Deployment of ES6 Code

- **Question**

We need to write modern code (ES6 or above) that can be deployed onto browsers running ES5.
What Can We Do?

- **Answer:**

Use transpilation to convert the code.

5.7 TypeScript

● Question

Does ES6 give you everything you need? What if we need something even better than ES6?

● Answer

ES6 is a big step up from ES5 but it is still missing some pieces that are provided by modern structured languages like Java and C#. For example: strong typing, decorators, enumerations etc.

However, don't worry. There is already something out there that takes ES6, builds on it and takes it a step further. It's called TypeScript. We will write modern code in ES6 and Typescript and use transpilation to convert it into compatible code to be deployed on the major web browsers.

Typescript was written by Microsoft and it is a very modern, structured language similar to Java and C#.

Google works in partnership with Microsoft on Typescript and used it to write Angular itself.

Therefore, using Typescript with Angular is a very good idea!

6 *TypeScript*

6.1 **Introduction**

TypeScript is a superset of JavaScript (written by Microsoft) which primarily provides optional static typing, classes and interfaces.

It is open source and is being developed on GitHub. The compiler is implemented in TypeScript and can work on any JavaScript host.

TypeScript is a strict superset of JavaScript. As such, a JavaScript program is also a valid TypeScript program, and a TypeScript program can seamlessly consume JavaScript. TypeScript compiles to compatible JavaScript.

It's quite similar to Java/.NET but there are sometimes differences, for example Constructors / Interfaces.

6.2 What is TypeScript?

In a nutshell:

TypeScript = JavaScript + Types + Classes + Modules + More

The most important of these is Types, which we will discuss below. This enables IDEs to provide a richer environment for spotting common errors as you type the code.

6.3 Can Browsers Run TypeScript?

No, not yet! TypeScript code is compiled down to JavaScript.

6.4 The Microsoft TypeScript Website

Microsoft has kindly provided the following website for us to learn TypeScript: typescriptlang.org.

The screenshot shows the main homepage of the TypeScript website. At the top, there's a navigation bar with links for 'learn', 'play', 'download', and 'interact'. Below the navigation is a search bar containing 'npm i -g typescript' and buttons for 'VS2013', 'VS2015', and 'the source'. A large blue header banner features the word 'TypeScript' in white. Below the banner, a sub-header says 'TypeScript lets you write JavaScript the way you really want to.' followed by 'TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any host. Any OS. Open Source.' To the right of this text is a blue button with the text 'Get TypeScript Now' and a right-pointing arrow.

Scalable

TypeScript offers classes, modules, and interfaces to help you build robust components.

These features are available at development time for high-confidence application development, but are compiled into simple JavaScript.

TypeScript types let you define interfaces between software components and to gain insight into the behavior of existing JavaScript libraries.

The screenshot shows the TypeScript playground interface. At the top, there's a navigation bar with links for 'learn', 'play', 'download', and 'interact', with 'play' being the active tab. Below the navigation is a sub-navigation bar with links for 'tutorial', 'handbook', 'samples', and 'language spec'. The main area contains two code editors side-by-side. The left editor is labeled 'TypeScript' and contains the following TypeScript code:

```
1 class Greeter {
2     greeting: string;
3     constructor(message: string) {
4         this.greeting = message;
5     }
6     greet() {
7         return "Hello, " + this.greeting;
8     }
9 }
10 var greeter = new Greeter("world");
11
12 var button = document.createElement('button');
13 button.textContent = "Say Hello";
14 button.onclick = function() {
15     alert(greeter.greet());
16 }
17
18 document.body.appendChild(button);
19
20
```

The right editor is labeled 'JavaScript' and contains the corresponding JavaScript code:

```
1 var Greeter = (function () {
2     function Greeter(message) {
3         this.greeting = message;
4     }
5     Greeter.prototype.greet = function () {
6         return "Hello, " + this.greeting;
7     };
8     return Greeter;
9 })();
10 var greeter = new Greeter("world");
11 var button = document.createElement('button');
12 button.textContent = "Say Hello";
13 button.onclick = function () {
14     alert(greeter.greet());
15 };
16 document.body.appendChild(button);
17
```

At the bottom of the playground interface, the word 'Playground' is centered.

6.5 The Main Differences between the JavaScript and TypeScript Languages

• Strong Typing

TypeScript gives us strong typing. It allows us to discover flaws in our code before they're pushed into production, if your code isn't valid, it won't compile.

Equality comparison is easier than with ECMA5 JavaScript because you can easily detect if the two items compared are of the same type. If they are not an error can be produced. After type checking is completed, equality checking is easier because both items are of the same type.

Having types in your code also allows IDE's more information to work with. For example if an IDE knows a variable is a string it can with narrowing down the autocomplete selection to strings only.

TypeScript offers the following basic types: Boolean, Number, String, Array, Enum, Any and Void.

• Classes

Classes are not in ECMAScript 5 but they are in TypeScript and ECMAScript 6.

Classes have constructors in the following format.

```
class Animal {  
    private name:string; constructor(theName: string) { this.name = theName; }  
}
```

Classes can extend other classes.

```
class Animal {  
    name:string;  
    constructor(theName: string) { this.name = theName; }  
    move(meters: number = 0) {  
        alert(this.name + " moved " + meters + "m.");  
    }  
}
```

```
class Snake extends Animal {  
    constructor(name: string) { super(name); }  
    move(meters = 5) {  
        alert("Slithering..."); super.move(meters);  
    }  
}
```

```
class Horse extends Animal {  
    constructor(name: string) { super(name); }  
    move(meters = 45) {  
        alert("Galloping..."); super.move(meters);  
    }  
}
```

Classes Can Implement Interfaces (see interfaces below for an example).

Classes can use public / private modifiers for member variables or methods. If you don't specify public or private for a variable or a method, the compiler assumes that the member is public.

● Interfaces

TypeScript interfaces can apply to functions.

```
interface SearchFunc {  
  (source: string, subString: string): boolean;  
}  
  
var mySearch: SearchFunc;  
mySearch = function(source: string, subString: string) {  
  var result = source.search(subString);  
  if (result == -1) {  
    return false;  
  }  
  else {  
    return true;  
  }  
}
```

TypeScript interfaces Can Apply to Properties.

Interfaces can enforce properties but can also have optional properties (for example color below).

```
interface LabelledClothing {  
  label: string;  
  size: number;  
  color?: string;  
}  
  
function printLabel(labelled: LabelledClothing) {  
  console.log(labelled.label + " " + labelled.size); }  
  
var myObj = {size: 10, label: "Dress"};  
printLabel(myObj);
```

TypeScript Interfaces Can Apply to Arrays

```
interface StringArray {  
  [index: number]: string;  
}  
  
var myArray: StringArray; myArray = ["Bob", "Fred"];
```

Classes Can Implement Interfaces.

```
interface ClockInterface {  
  currentTime: Date;  
  setTime(d: Date);  
}  
  
class Clock implements ClockInterface {  
  currentTime: Date;  
  setTime(d: Date) {  
    this.currentTime = d;  
  }  
  constructor(h: number, m: number) { }  
}
```

And you can have interfaces that extend other Interfaces.

```
interface Shape {
```

```
    color: string;  
}  
  
interface Square extends Shape {  
    sideLength: number;  
}  
  
var square = <Square>{};  
square.color = "blue"; square.sideLength = 10;
```

● Modules

Modules are not in ECMAScript 5 but they are in TypeScript and ECMAScript 6.

The ‘export’ keyword allows you to export your TypeScript objects in a module so that they can be used elsewhere.

There are two main types of TypeScript Modules: internal modules and external modules. In Angular 4 most of the time you will be working with *external* modules.

Internal Modules

Internal modules are Type Script’s own approach to modularize your code. You use the ‘Module’ keyword to create a module. Internal modules can span across multiple files, effectively creating a namespace. There is no runtime module loading mechanism, so – in a browser environment – you have to load the modules using `<script>` tags on your own. Alternatively, you can compile all TypeScript files into one big JavaScript file that you include using a single `<script>` tag.

Declaring Internal Modules

```
module mymod {  
  
    export function doSomething() {  
        // this function can be accessed from outside the module }  
  
    export class ExportedClass {  
        // this class can be accessed from outside the module }  
  
    class AnotherClass {  
        // this class can only be accessed from inside the module }  
}
```

Consuming Internal Modules

Address them using their fully qualified name.

```
var exportedClassInstance = new mymod.ExportedClass(); Or import them.
```

```
import ExportedClass = mymod.ExportedClass; var exportedClassInstance = new ExportedClass();
```

External Modules

These are the types of modules most commonly-used when developing in Angular 4. External modules leverage a runtime module loading mechanism. We will go into module loading

mechanisms in Chapter ‘**Error! Reference source not found.**’.

Declaring External Modules

If you want to use external modules, you have to decide which module system (AMD or CommonJS) to use and then compile your sources using the –module compiler flag. Possible values are ‘amd’ or ‘commonjs’.

In computing, a namespace is a set of symbols that are used to organize objects of various kinds. With external modules, your file’s name and path will create the namespace, which identifies the item.

Example file ‘projectdir/ExportedClass.ts’.

```
class ExportedClass {  
// code ....  
}  
export = ExportedClass;
```

Consuming External Modules

```
import ExportedClass = require("projectdir/ExportedClass"); var exportedClassInstance = new ExportedClass();
```

● Enumerations

These are familiar to Java and .NET developers.

```
enum Color {Red, Green, Blue}; var c: Color = Color.Green;
```

● Generics

Again, familiar to Java and .NET developers.

```
interface LabelledClothing {  
label: string;  
size: number;  
}  
var arr: Array<LabelledClothing> = new Array<LabelledClothing>();
```

● Constructors

TypeScript uses the ‘Constructor’ keyword to declare constructors, rather than the class name. Another difference is that it automatically assigns constructor arguments as properties. You don’t need to assign instance variables in your constructor, this is already done for you.

This:

```
class Person {  
constructor(private firstName: string, private lastName: string) {  
}  
}
```

Equals:

```

class Person {
    private firstName: string;
    private lastName: string;

    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

```

● Functions

Arrow Functions

Arrow functions are not in ECMAScript 5 but they are in TypeScript and ECMAScript 6. An arrow function is a function that you can write inline in your source code (usually to pass in to another function).

```

var calculateInterest = function (amount, interestRate, duration) {
    return amount * interestRate * duration / 12;
}

```

Could Be Written As:

```

var calculateInterest2 = (amount, interestRate, duration) => {
    return amount * interestRate * duration / 12;
}

```

Or in A Shortened Format As:

```

var calculateInterest3 = (amount, interestRate, duration) => amount *
interestRate * duration / 12;

```

Arrow Functions The syntax is not the main reason why developer use Arrow Functions in TypeScript. The main reason is that the value of the 'this' variable is **preserved** inside Arrow Functions.

This can be of great benefits to the developer as with regular JavaScript functions there is a mechanism called 'boxing' which wraps or change the 'this' object before entering the context of the called function. Inside an anonymous function, the 'this' object represents the global window. In other functions, it represents something else.

So many developers use arrow functions when they absolutely want to ensure that the 'this' variable is what they expect.

Example Regular Function

```

function Person(age) {
    this.age = age
    this.growOld = function(){
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function(){ console.log(person.age); },2000); // 1, should have been 2

```

After running this code, `person.age` has value 1. **It should have value 2.**

That's because the 'this' variable inside the Person function does not actually represent the Person function.

Example Arrow Function

```

function Person(age) {
    this.age = age
    this.growOld = () => {
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function(){ console.log(person.age); },2000); // 2

```

After running this code, `person.age` has value 2, which is correct.

That's because the 'this' variable inside the Person function represents the Person function as expected.

Ellipsis Operator

This operator (denoted by '...') allows a method to accept a list of arguments as an array.

```

function sum(...numbers: number[]) {
var aggregateNumber = 0;
for (var i = 0; i < numbers.length; i++)
aggregateNumber += numbers[i];
return aggregateNumber;
}

console.log(sum(1, 5, 10, 15, 20));

```

● Getters and Setters

If you are targeting browsers with ECMAScript5, this scripting version supports the ‘Object.defineProperty()’ feature. If you use TypeScript getters and setters then you can define and directly access properties with the ‘.’ notation. If you are used to C# then you are already quite used to this.

```
class foo {  
private _bar:boolean = false;  
get bar():boolean {  
    return this._bar; }  
set bar(theBar:boolean) {  
    this._bar = theBar; }  
}  
  
...  
  
var myBar = myFoo.bar;  
myFoo.bar = true;
```

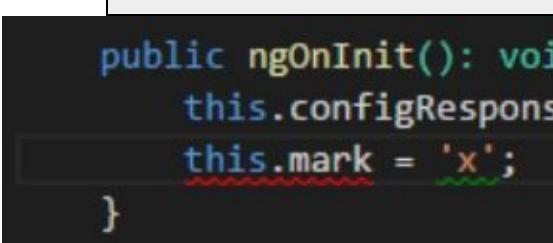
● Types

You can have variable types in TypeScript but it is optional. The reason it is optional is so that TypeScript is backwards-compatible so that TypeScript can run all your JavaScript code. When you declare a variable in Typescript you can specify the variable type by adding ‘: [type]’ after its name.

Example

We declare a mark variable of the type ‘number’.

var mark: number = 123; So, if we edit the code to assign a string to this variable in the code we get the syntax error highlighting below. Note that this doesn’t happen in Plunker, only in an editor like Visual Studio Code.



```
public ngOnInit(): void  
{  
    this.configResponses();  
    this.mark = 'x';  
}
```

If we save this (bad) code and we compile the TypeScript then we get the following error: Type ‘string’ is not assignable to type ‘number’.

Primitive Types

TypeScript offers the following primitive types: any, void, number, string and boolean. Primitive types are not inherited from the ‘Object’ class and they are not extendable (you cannot subclass them). Primitive types are typically named with a lower case first letter: e.g. number.

Object Types

Object types are types that are not primitives. They are inherited from the ‘Object’ class and they are extendable. Object types are typically named with an upper case first letter: e.g. Number.

Objects of this type have access to their ‘prototype’ so that you can add additional functionality to the object.

```
String.prototype.Foo = function() {  
    // DO THIS...  
}
```

Object Types also let you use the ‘instanceof’ to check class.

```
myString instanceof String
```

Union Types

Sometimes you want a variable to be one of multiple types e.g. a string or a number. You can use the union (‘|’) type for this. The variable below can be a string or a number. The code below is valid.

```
var name: string|number;  
  
...  
  
constructor(){  
    this.name = 'abc';  
    this.name = 22;  
}
```

Another example:

```
var action = ActionNew | ActionSave | ActionDelete ; ...  
  
if (action instanceof ActionNew){  
    ...do something...  
}
```

Union Types can also apply to function arguments and results.

```
function format(value: string, padding: string | number) { // ... }  
  
function getFormatted(anyValue:any): string | number { // ... }
```

Alias Types

You can also use the ‘type’ keyword to define type aliases:

```
type Location = string|number;  
var loc: Location;
```

Tuple Types

A tuple is a finite ordered list of elements, for example: name, address, numeric zip code. TypeScript allows you to access this data using variables using classes or tuples. The tuple type allows you to define a variable as a sequence of types.

```
var contactInfo: [string, string, number];  
contactInfo = ['Mark', '12 Welton Road', 30122];
```


7 Editors

7.1 Introduction

There are a wide range of editors available that will work with TypeScript, including: Visual Studio, Visual Studio Code, WebStorm, WebEssentials, Eclipse.

7.2 Visual Studio Code

I chose Visual Studio Code. It is an open source source code editor developed by Microsoft for Windows, Linux and OS X. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring. The reasons I chose it are: It was free.

It ran on both my pc and my mac.

It was written by the same people who wrote TypeScript, so we know it would work well with it.

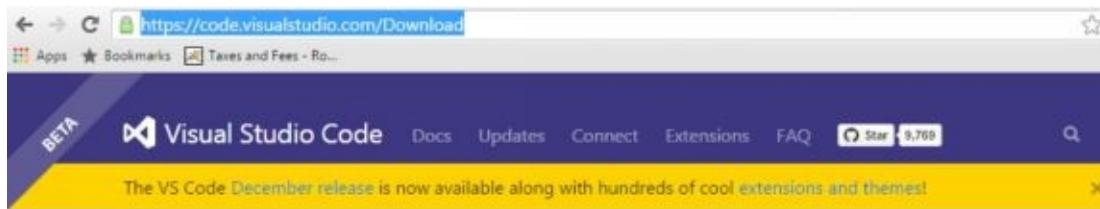
It was relatively compact.

It also works well with JavaScript, PHP etc.

Like the other editors mentioned above, it also has the code completion (control & space) and syntax highlighting.

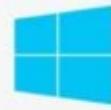
7.3 Website

code.visualstudio.com



Code focused development. Redefined.

Build and debug modern web and cloud applications. Code is free and available on your favorite platform - Linux, Mac OSX, and Windows.



[Download for Windows](#)



[Download for Linux x64](#)



[Download for OS X](#)

[32bit version](#)

7.4 Opening Your Project in Visual Studio Code.

● Shell

Double-click on Visual Studio Code to open it.

Go to ‘File’ menu.

Select ‘Open Folder’.

Select your project’s root folder.

● Command Line

Navigate to the root folder of your project.

Enter the command ‘code .’ (code space period).

7.5 How to Open Files

Control P

This lists the files at the top underneath the text box at the top. When you type it filters the list:

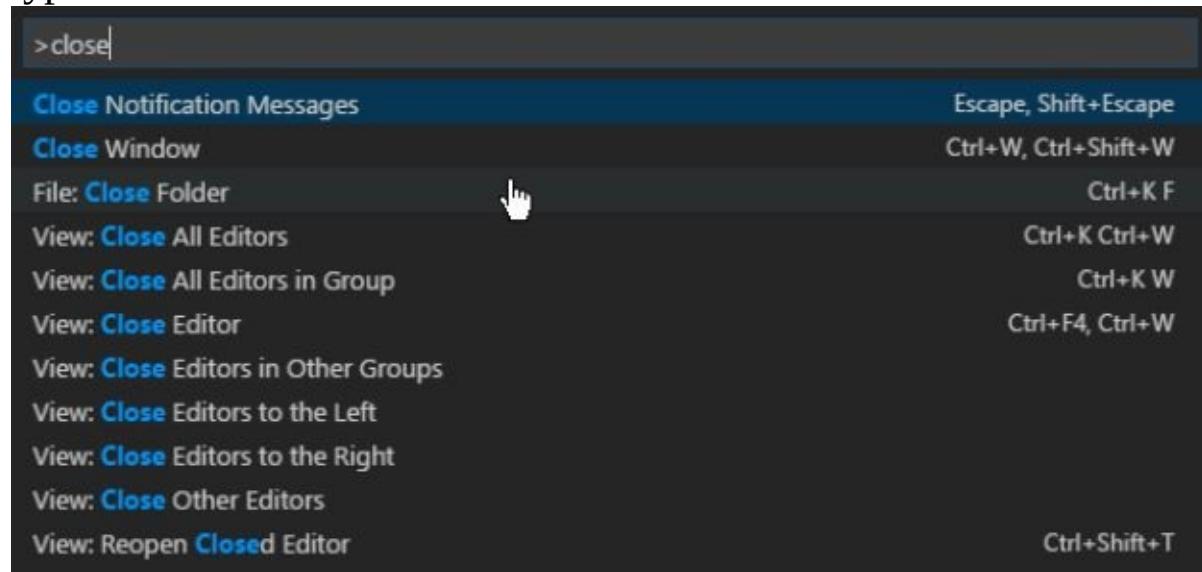
The screenshot shows a search interface with a text input field containing 'dashboard.js'. Below the input field is a list of file results. The first result, 'dashboard.controller.js /src/client/app/dashboard', is highlighted in blue and has a 'recently opened (2)' label next to it. The other results are listed below in a standard black font.

File Path	Count
dashboard.controller.js /src/client/app/dashboard	recently opened (2)
dashboard.controller.js.map /src/client/app/dashboard	
dashboard.route.js.map /src/client/app/dashboard	file and symbol results (10)
dashboard.module.js.map /src/client/app/dashboard	
dashboard.route.js.html /report/coverage/report-lcov/lcov-report/app/dashboard	
dashboard.route.js.html /report/coverage/report-html/app/dashboard	
dashboard.route.spec.js /src/client/app/dashboard	
dashboard.module.js.html /report/coverage/report-html/app/dashboard	
dashboard.module.js.html /report/coverage/report-lcov/lcov-report/app/dashboard	
dashboard.controller.js.html /report/coverage/report-html/app/dashboard	
dashboard.controller.js.html /report/coverage/report-lcov/lcov-report/app/dashboard	
dashboard.controller.spec.js /src/client/app/dashboard	

7.6 How to See the Available Commands and Hot Keys

Control – Shift – P

This lists the commands at the top underneath the text box at the top. When you type it filters the list:



7.7 How to Configure the Build

Edit file ‘tasks.json’ (in the root folder of your project).

This is configuration file specifies the build command we are going to use in the example project.

It runs ‘npm run build’ on the command line to invoke the build.

See Chapter ‘Introducing Webpack’ for more information on Webpack and the build process.

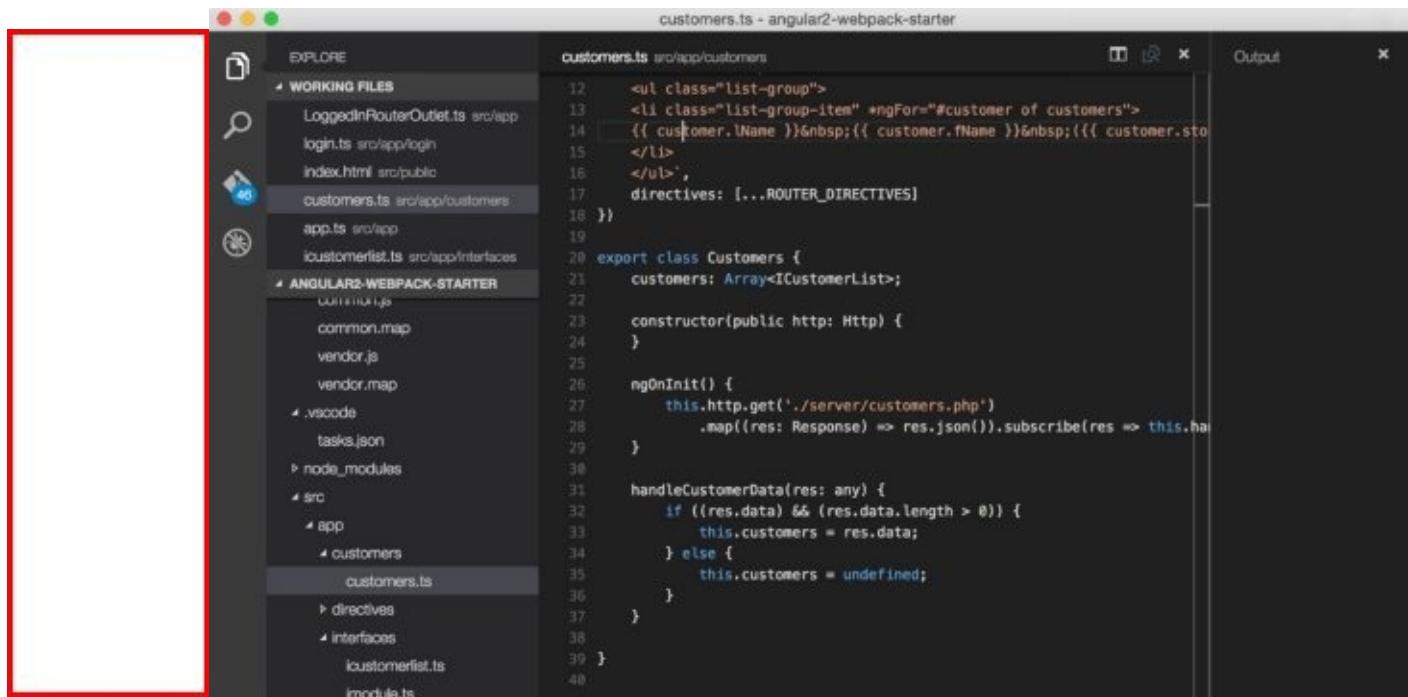
```
tasks.json .vscode
1 {
2   "version": "0.1.0",
3   "command": "npm",
4   "isShellCommand": true,
5   "args": ["run", "build"]
6 }
```

7.8 How to Build

Control – Shift – B

Build output will be displayed in the Output window.

It normally takes between 10 – 30 seconds to run.



The screenshot shows the Visual Studio Code interface with a red box highlighting the left sidebar (Exploratory View) and the main code editor area. The sidebar displays the project structure under 'WORKING FILES' and 'ANGULAR2-WEBPACK-STARTER'. The code editor shows the file 'customers.ts' with the following content:

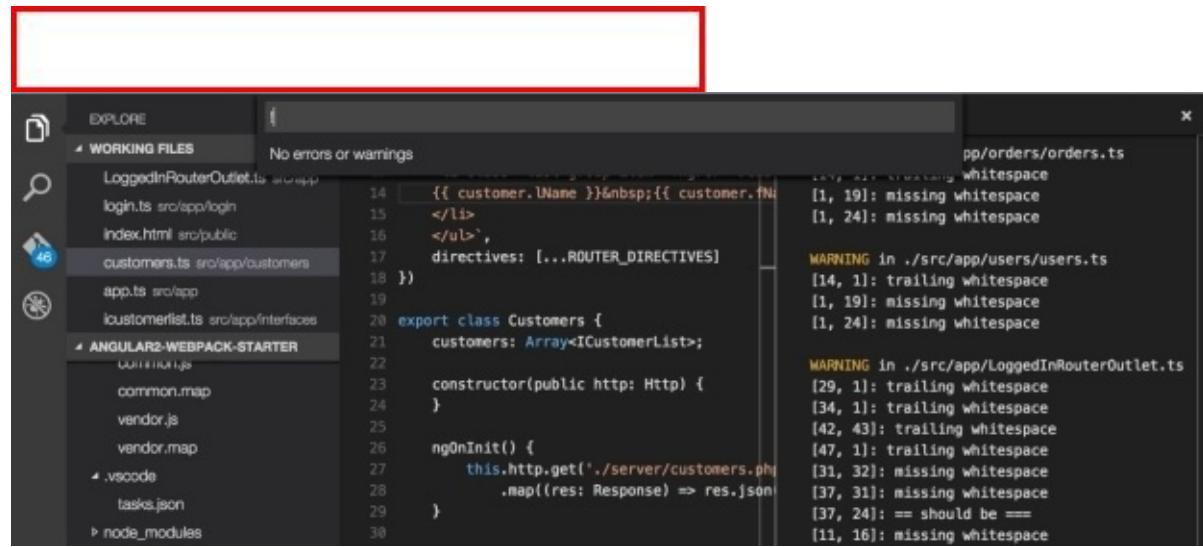
```
customers.ts - angular2-webpack-starter
customers.ts src/app/customers
12  <ul class="list-group">
13    <li class="list-group-item" *ngFor="#customer of customers">
14      {{ customer.lName }}&nbsp;{{ customer.fName }}&nbsp;{{ customer.sto
15    </li>
16  </ul>;
17  directives: [...ROUTER_DIRECTIVES]
18 }
19
20 export class Customers {
21   customers: Array<ICustomerList>;
22
23   constructor(public http: Http) {
24   }
25
26   ngOnInit() {
27     this.http.get('./server/customers.php')
28       .map(res: Response) => res.json().subscribe(res => this.ha
29   }
30
31   handleCustomerData(res: any) {
32     if ((res.data) && (res.data.length > 0)) {
33       this.customers = res.data;
34     } else {
35       this.customers = undefined;
36     }
37   }
38
39 }
40
41
42
43
44
45
46
47
48
```

7.9 To View Build Errors

Control – Shift – M

Lists errors at top of the screen.

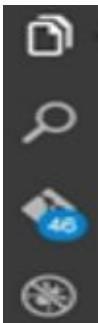
Click on an error to navigate to the source of the error.



The screenshot shows the Visual Studio Code interface with a red box highlighting the terminal window. The terminal displays build errors for the file `pp/orders/orders.ts`. The errors are:

```
pp/orders/orders.ts
[1, 19]: missing whitespace
[1, 24]: missing whitespace
WARNING in ./src/app/users/users.ts
[14, 1]: trailing whitespace
[1, 19]: missing whitespace
[1, 24]: missing whitespace
WARNING in ./src/app/LoggedInRouterOutlet.ts
[29, 1]: trailing whitespace
[34, 1]: trailing whitespace
[42, 43]: trailing whitespace
[47, 1]: trailing whitespace
[31, 32]: missing whitespace
[37, 31]: missing whitespace
[37, 24]: == should be ===
[11, 16]: missing whitespace
```

The left sidebar shows the project structure with files like `LoggedInRouterOutlet.ts`, `login.ts`, `index.html`, `customers.ts`, `app.ts`, `ICustomerList.ts`, and `common.map`.



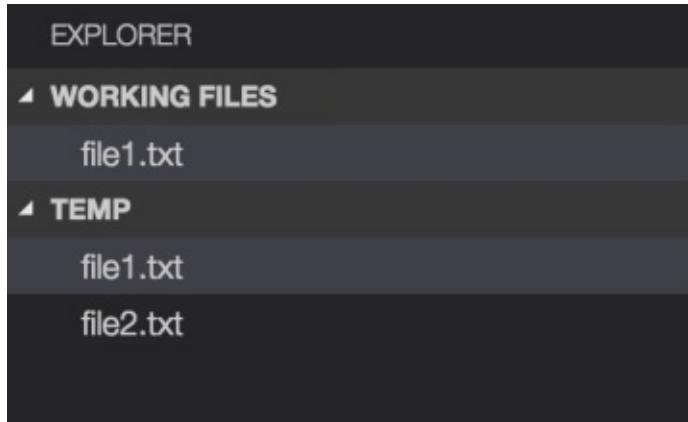
7.10 Sidebar Modes

Visual Studio Code shows a sidebar to the left and an editing area to the right. You can show and hide the sidebar using the Control - B keyboard shortcut. You can easily switch between four main sidebar modes: Explorer, Search, Git and Debug. There are different ways to switch mode: Click on the big icons on the left side.

Use the options in the ‘View’ menu.

Use the hotkeys (see below).

● Explorer



This is the file explorer window (on the left side by default). This is split into two sections: Working Files (above) and Project Files (below). Click on a file in the file list to display it on the right side for editing.

To activate / focus this window use:

Files icon on left.

‘View’ menu option ‘Explorer’.

Control – Shift – E

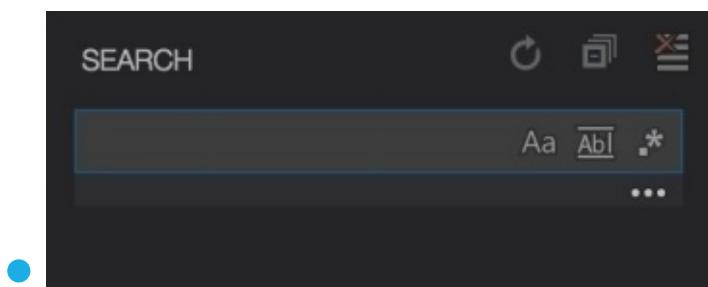


● Working Files

When you edit files, they appear in working files. If you are only editing a few of the project files at once, it is handy having these files listed at the top. When you hover over the ‘Working Files’ heading, it shows an ‘X’ to allow you to clear this list.

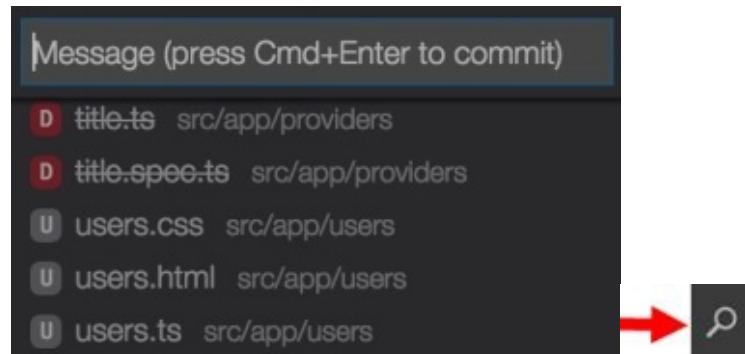
● Project Files (in this case TEMP)

This is a list of all the files in the project, as well as the folders.



To activate / focus this window use:

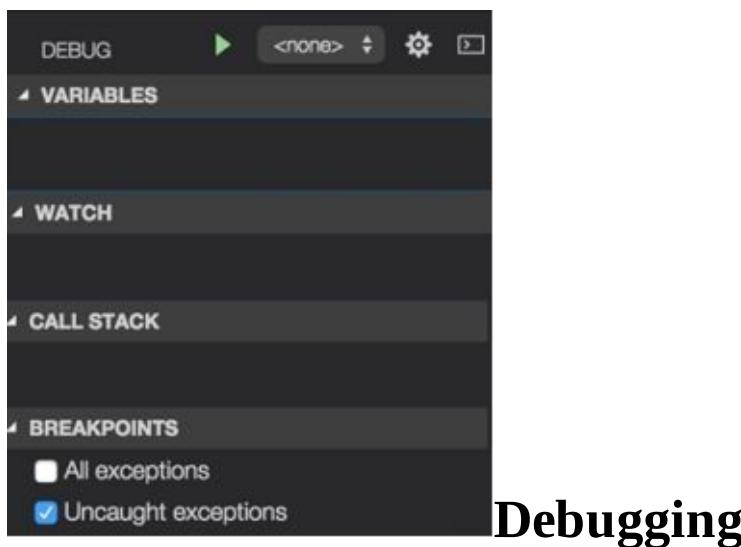
- Magnifier icon on left.
- 'View' menu option 'Search'.
- Control – Shift – F



● GIT

To activate / focus this window use:

- Git icon on left.
- 'View' menu option 'Git'.
- Control – Shift – G



To activate / focus this window use:

- Bug icon on left.

‘View’ menu option ‘Debug’.

Control – Shift – D



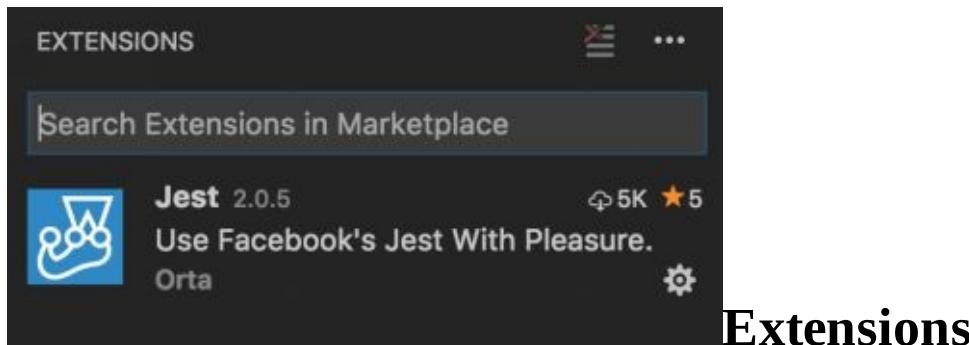
The debugging is more useful for debugging server-side code than browser-side code.

You can debug browser code using this debugger if you enable remote debugging on your browser and attach to it but it is probably easier just to use the available (and excellent) browser debuggers.

To debug your server-side code you must first setup a debug launch task. This enables you to setup your debugging launch configuration, which you use to start the server-side code and start debugging it.

To do this:

1. Click on the bug icon on the left or use another option (see below).
2. Click on the ‘gear’ icon and this opens the debug configuration settings (in .settings/launch.json).
3. Pick your debugging configuration (next to the gear icon) and hit play to launch it.



To activate / focus this window use:

Extension icon on left.

‘View’ menu option ‘Extensions’.

Control – Shift – X

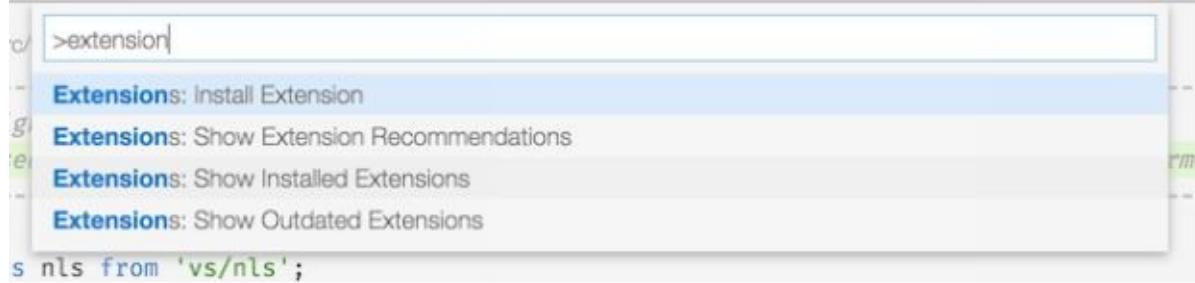


7.11 Extensions

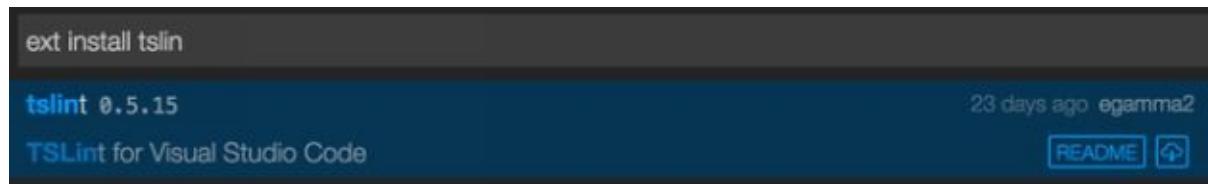
It is very easy to install extensions into Code. The Angular 4 project I work on has a build process that includes ‘Linting’. This checks code to ensure that it follows style guidelines. Often the build will fail if the user adds too much whitespace. This becomes annoying and it is a good idea to install the ‘linter’ extension into Code so that it highlights ‘linting’ issues as they occur (with a warning at the bottom left).

To view commands to do with extensions, enter the following:

extensions This displays the following list of available commands:



To install an extension into Code, enter the following command and follow instructions: ext install To setup the TypeScript linter in Code, enter the following command and follow instructions: ext install tslint



7.12 Notes

● IntelliSense

Being a rich editing environment, Code offers IntelliSense code selection and completion. If a language service knows possible completions, the IntelliSense suggestions will pop up as you type. You can always manually trigger it with control & space.

● Saving

This is obvious but the normal File menu commands apply, as well as the Ctrl – S shortcut.

● Going Back

Code allows you to navigate in and out of code freely, you can ‘Control ‘ & Left Mouse click to ‘drill into’ code, such as a method. This is great when you need to look at something in detail. However you need to be able to go back to where you were. This is where the navigate backwards comes in. You can also navigate forwards also.

Keyboard: CTRL + -; CTRL + SHIFT + -

Menu: View -> Navigate Backward; View -> Navigate Forward

8 Node

8.1 Introduction

We need to get coding soon. However, to get coding we are going to need a project. To create a project, we are going to need to use the CLI (more on that later) and the CLI needs Node to work. So we need to cover Node before coding!

Node is a JavaScript runtime that you install on your computer. It is a platform for development tools and servers (or anything else). It is straightforward to use and there are hundreds of modules already written for it, lots of code you can reuse.

Node uses the V8 JavaScript engine code written by Google for the Chrome browser, in combination with additional modules to do file I/O and other useful stuff not done in a browser. Node does nothing by itself but it is a platform on which many JavaScript code modules can be run. Useful modules like web servers, transpilers etc. To download Node, go to nodejs.org/download. You will need it!

8.2 Installing Node

To install Node, go to the website ‘nodejs.org’ to download and install the core Node software for your computer. When you go to that webpage you have the option of downloading and installing the most recommended release, or the latest release. Obviously, the former is more stable and thus recommended.

8.3 Setting Up Node in Your Project Folder

The following command sets up node in your project, asking you some questions and then generating the ‘package.json’ file. Please run this command in the root folder of your project.

```
npm init
```

8.4 Running Code with the Node Command

Once you have node installed you will have command-line access to a command ‘node’. Entering this command without arguments will allow you to type in JavaScript and hit enter to run it.

```
$ node  
> console.log('Hello World');  
Hello World
```

The more useful way to use the command ‘node’ is to enter this command plus a file name as an argument. This will execute the contents of the file as JavaScript. In this case we will create a file ‘hello.js’:

```
setTimeout(function() {  
    console.log('Hello World!');  
}, 2000);
```

Now we can run it:

```
node hello.js The program waits 2 seconds then writes 'Hello World' to the console.
```

8.5 Node Modules and Dependencies

Now we know how to run JavaScript code through Node, we need to know about how to install these useful modules. You would think that this would be simple but it is not because many node modules depend on other node modules to work. So when you install a node module, node needs to ensure that any node modules that are dependencies also need to be installed. That's why the 'Node Package Manager' was invented for you. To add, update and delete node modules to your project and also manage these interdependencies!

8.6 Node Package Manager

For this purpose, Node provides a command-line access to a command ‘npm’. This means ‘node package manager’ and has many different arguments allowing you to install modules, update them or uninstall them.

The website docs.npmjs.com is a great resource for detailed documentation on node package manager. Also www.npmjs.com is a great resource for available node packages.

8.7 Node Module Installation Levels

There are two levels of node module installation.

- **Global**

If you're installing something that you want to use on the command line, install it globally. To install a module globally, add ‘-g’ to the npm install on the command line.

```
npm install -g typescript
```

- **Local**

If you're installing something that you want to use *in* your program (not from the command line) use this level. To install a module locally, leave out the ‘-g’ from the npm install on the command line.

```
npm install express
```

8.8 ‘package.json’ File

Node is designed to be run from the command line within a project folder. It allows developers to store information pertinent to their project in a ‘package.json’ file, which should reside in the root folder of your project. This file specifies many useful things about your project : The name and version of your project.

What node modules your project depends on (and what versions of these node modules you need).

What node modules are required for your project in production.

What node modules are required for your project in development (i.e. not needed for production).

● Updating this File

You can update this ‘packages.json’ file in two ways:

1. [By using node commands \(on the command-line\) that install/update/delete node modules and update this file.](#)
2. [Edit this file yourself. Then you run node commands to install/update/delete node modules to match this file.](#)

● Version Numbers

Note that the ‘package.json’ file allows the developers to specify node modules that the project requires. Note that when you specify the dependencies in this file, you also specify the versions of these dependencies, for example 1.0.1. Node allows you to be flexible and specify the version number you require in many different ways:

1.2.1	Must match version 1.2.1.
>1.2.1	Must be later than version 1.2.1.
>=1.2.1	Must be version 1.2.1 or later.
<1.2.1	Must be before version 1.2.1.
<=1.2.1	Must be before or equal to version 1.2.1.
~1.2.1	Must be approximately equivalent to version 1.2.1.
^1.2.1	Must be compatible with version 1.2.1.
1.2.x	Must be any version starting with ‘1.2.’.
*	Any version.

8.9 Folder ‘node_modules’

When you install a node module it is downloaded and placed into the folder ‘node_modules’ within your project folder. Often you get a lot more than you bargained for, because the node module you installed has many dependencies! So you end up with a huge ‘node_modules’ folder with dozens of module subdirectories inside. Sometimes it takes a long time for npm to download and install the project node modules.

8.10 Npm - Installing Modules into Node

There are two different ways of installing modules into Node. You can run the command ‘npm install’ specifying the module (to install it) or you can edit the ‘package.json’ file and then run ‘npm install’

● Run ‘npm install [Module Name]’ To Install the Module

This works great if you are doing something simple, like adding a single additional node module to your project. For example, one of the most useful node modules is Express, a capable web server. To install Express we could enter the following on the command-line: `npm install express`

Note

This will not update your node dependency file ‘package.json’. If you need this module to be saved as a project dependency, please add the ‘--save’ or ‘--save-dev’ argument to the command.

Save Argument ‘—save’

This adds the node module that you are about to install as a node modules which is required for your project in production.

```
npm install express --save
```

Save Argument ‘—save-dev’

This adds the node module that you are about to install as a node modules which is required for your project in development only (i.e. not needed for production).

```
npm install express --save-dev
```

● Edit the ‘package.json’ File Then Run ‘npm Install’

Manually editing your ‘package.json’ file is the best way to install multiple modules when your project depends on multiple modules. First of all you have to setup a ‘package.json’ file in the root folder of your project. This file contains an overview of your application. There are a lot of available fields, but in the file below you can see the minimum. The dependencies section describes the name and version of the modules you’d like to install. In this case we will also depend on the Express module.

Example package.json File.

```
{
  "name": "MyStaticServer",
  "version": "0.0.1",
  "dependencies": {
    "express": "3.3.x"
  }
}
```

To install the dependencies outlined in the package.json file we enter this on the command-line in the root folder of our project.

```
npm install
```

8.11 Updating Node Modules

Sometimes your dependencies change. You want to add an additional module but adding that module requires that others are of a later version number. Note that node provides the following command to check to see if your modules are outdated: `npm outdated` There are two different ways of updating modules in Node.

You can run the command ‘`npm update`’ specifying the module to be updated. Also add the ‘`--save`’ option if you want your ‘`package.json`’ file updated with the later version. If the `-g` flag is specified, this command will update globally installed packages.

You can edit the ‘`package.json`’ file, update the module dependency and then run ‘`npm update`’. This will update your modules to match the specifications in this file.

8.12 Checking Your Node Version

If you already have node installed, you can check its version by running the following command:

```
npm -v
```

8.13 Uninstalling Node Modules

You can run the command ‘npm uninstall’ specifying the module to be uninstalled. Also add the ‘--save’ option if you want your ‘package.json’ file updated with the module removed from the dependency list. If the -g flag is specified, this command will remove globally installed packages.

8.14 How to Install the Latest Angular

Go to your project folder and issue the following command:

```
npm install @angular/{common,compiler,compiler-cli,core,forms/http,platform-browser,platform-browser-dynamic,platform-server,router/animations}@latest typescript@latest --save
```


9 Start Coding With the CLI

9.1 Introduction

When I first started developing in Angular 2, I found that it was a sharp learning curve at first. The reason was that it was very hard to get your project going as there was no standard ‘Angular 2’ project blueprint that would simply take care of building and running your project. You had to setup your dependencies in Node (more on that later), set up your build process, setup to deployment process. This made Angular 2 tough at first, as you had to learn the concepts and the syntax at the same time!

9.2 Angular CLI Gets You Going Fast

The Angular CLI was developed to allow developers to get going with Angular fast. And it's great as it can generate projects that are well-structured and designed. I cannot tell you what a great tool it has turned out to be how and how quickly it has been adopted The Angular CLI is an open source project and you can look at its code here: <https://github.com/angular/angular-cli/>

The official Angular CLI documentation is here:

<https://cli.angular.io/>

<https://angular.io/docs/ts/latest/cli-quickstart.html>

9.3 This Chapter

The purpose of this chapter is to get you creating a project using the CLI. It is not going to go into great detail on the CLI yet as it not yet required. There will be more information on the CLI in later chapters. However, if you want lots of information right now, may I recommend this excellent article: <https://www.sitepoint.com/ultimate-angular-cli-reference/>

9.4 CLI = Command Line Interface

The Angular CLI uses a command line interface. You use the ‘ng’ command in the terminal and it does things. You may think that this reminds you of the ‘bad old days’ when you had to remember these commands, but when you look at what it does that becomes of small importance: Let’s you create new Angular applications.

Let’s you run a development server with live reloading of changes.

Let’s you add more code to your Angular application Runs your application’s tests.

Builds your application for deployment.

Deploy your application.

9.5 You Need Node to Install CLI

To get the CLI running, you first need to install Node.js version 4.0.0 or greater. To install the cli, enter the following command in a terminal (this will kick off all kind of Node downloads): `npm install -g angular-cli` Note that the ‘-g’ parameter installs angular cli as a global package. This will put the ‘ng’ command on the path, making it usable in any directory.

You can check your version of CLI by running the following command:

```
ng version
```

9.6 Create a Start Project

Finally, we are going to do some coding. Well not really! Let's just create the basic project and run it. Follow the steps below:

1. Open a terminal window.
2. Navigate to a suitable folder, for example your Documents.
3. Enter the command below. This will create a new Angular app in a folder called 'Start'. It will also spew out lots of files that it creates.

```
ng new start
```

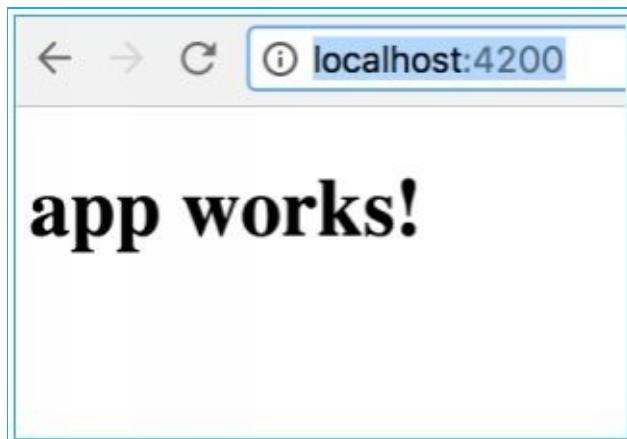
4. Navigate into the 'start folder'.

```
cd start
```

5. Enter the following command to start the app:

```
ng serve
```

6. Open your web browser and browse to localhost:4200. You should see the text 'app works'. That means that your project is running.



9.7 Open the Start Project

Now let's take a look at this project and what's in it. Open Visual Studio code and open the folder 'start'. Here's what's inside and how it is structured.

• What's in the Root Folder?

e2e	Folder for testing files. More on testing, Karma and Protractor in a later chapter.
node_modules	Folder for project node dependencies.
src	Folder for project source code.
.editorConfig	Editor configuration file.
.gitignore	Git ignore file.
angular-cli.json	CLI configuration file. Change your CLI options in this file.
karma-conf.json	Karma configuration file. More on testing, Karma and Protractor in a later chapter.
package.json	Node dependencies configuration file.
protractor-conf.js	Protractor configuration file. More on testing, Karma and Protractor in a later chapter.
README.md	Readme informational file. Useful as it contains information on the CLI commands.
tslint.json	Lint configuration file.

• Source Code

This is the really important stuff. The source code that was generated by CLI for your project. Here is the starting point for your coding!

app	Folder for your application source code files. Currently contains source code for an application component (more on this later).
assets	Folder for your application image and css files.
environments	Folder for configuration files for environments. For example configurations for development and production.

favicon.ico	Application icon.
index.html	The html page for the Angular Single Page Application.
main.ts	Code to start the application. More on this later.
styles.css	Global style definitions.
test.ts	Code to run the application tests.
tsconfig.json	Typescript / compiler configuration file.

9.8 Modify the Start Project

Let's modify the start project and see what happens. Follow the steps below:

1. Open a terminal window. Navigate to the 'start' folder and ensure the 'ng start' command is running and that navigating to 'localhost:8080' produces the webpage as expected ('app works!'). Leave the 'ng start' command running.
2. Edit the file 'src/app/app.component.ts' and change it to the following:

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

3. Go back to your web browser. It should now display this:



app works! and has been modified....

Note how the app automatically re-compiled and reloaded as soon as you hit 'Save' in your editor. That's because the CLI project includes 'Watchman', which watches for changed files and rebuilds & reloads your app when you change it.

9.9 Start Project - Compile Errors

Let's introduce a compile error into the project and see what happens.

Edit the file 'src/app/app.component.ts' and change it to the following (remove the quotes from the text 'app works!....':

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

Note how the app doesn't change or reload. Note how you get the error messages in the terminal window. Also note that you get error messages in the browser console. In Chrome, you view the browser console by selecting 'More Tools' then 'Developer Tools' in the menu.

9.10 Start Project - Runtime Errors

Let's introduce a runtime error into the project and see what happens.

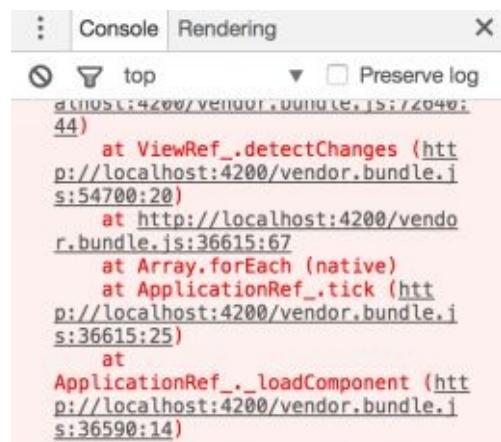
1. Edit the file 'src/app/app.component.ts' and change it back to the original code:

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

2. Edit the file 'src/app/app.component.html' and change it to the following (to create an error):

```
<h1>
{{title.test.test}}
</h1>
```

Note how the app goes blank. If you check the terminal it says 'webpack: Compiled successfully'. So, the compile worked. However, the page did not load because we (purposefully) introduced a runtime error (i.e. one that only occurs when the app runs). To find the error, go to the browser console. In Chrome, you view the browser console by selecting 'More Tools' then 'Developer Tools' in the menu.



9.11 Watcher & Web Server

As we mentioned earlier, if you leave ‘ng serve’ running, this watches our files (performing a compile and redeploy when necessary) and runs a local web server on localhost:4200. When you change something and hit save, the watcher does the following:

- A Webpack build, including transpilation to compatible JavaScript and bundling code. More on Webpack in a later chapter.

- Generates a new index.html file, adding script references as required to reference the JavaScript files bundled by Webpack.

- Performs a new deployment onto the local web server.

- Refreshes the web page.

9.12 CLI Version

Use the following command to check which version of the CLI you are running: `ng --version` To update the CLI version, you should uninstall it and reinstall it with the following commands: `npm uninstall -g angular-cli`
`npm cache clean`
`npm install -g angular-cli`

9.13 How the Start Project Bootstraps

• Bootstrapping

Bootstrapping usually refers to a self-starting process that is supposed to proceed without external input. In this case, it refers to how an Angular application starts up. In this chapter, we are going to take a look at how the starter project (from the previous chapter) bootstraps.

• Loading the Start Project Application

When we go to localhost:4200 the following happens:

1. The web browser opens the file ‘index.html’ by default.
2. The browser loads the script files on the end. This includes ‘main.bundle.js’, which is a transpiled version of the typescript file ‘main.ts’. This is our ‘main’ app entry point.
3. Main.bundle.js loads some modules then calls the angular system code below:
platformBrowserSpecific().bootstrapModule(AppModule).
4. AppModule is loaded, it is the root Angular module that is used to bootstrap the application. This is an Angular module, not a JavaScript module – these are different things. We will cover Angular modules in a later chapter. If you look at AppModule.ts you will see that it contains the following line to tell the module to bootstrap with the AppComponent component:

```
@NgModule({  
  ...  
  bootstrap: [AppComponent]  
})
```

5. The AppModule bootstraps with the AppComponent, injecting the component into the space between the start and end tags ‘app-root’.

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

9.14 Conclusion

Hopefully this chapter introduced you to CLI. There is so much more you can do with it than just creating a starter project: Adding different types of objects to your project.

Testing your code.

Building your code.

Deploying your code.

Etc.

We will be using it in all of our coding examples, so don't worry we will be covering it much more and will be doing many more things with it.

10 Introducing Components

10.1 Introduction

An Angular Component is similar to an AngularJS Controller.

A Component is basically markup, meta-data and a class (which contains data and code), which combined together create a UI widget. They are the main tool that we use to build an interactive UI with. All Angular applications have a root component, often called the Application Component.

Angular provides ways for Components to pass data to each other and to respond to each other's events. We will go into Component Inputs and Outputs in Chapter '**Error! Reference source not found.**'.

• Components Are Like Lego UI Blocks

You can write a Component and use it as a Child Component in several other Components. They were designed to be self-contained and loosely-coupled for this purpose. Each Component contains valuable data about itself: What data it needs as input.

What events it may emit to the outside.

How to draw itself.

What its dependencies are.

• Components and Files

Normally when you develop components, you have one component in each three files because there are three parts to a component: the template, the class and the style. This is how the CLI works by default. For example when you create an app in the CLI using the command 'ng new [project name]', the CLI generates three files for the app component (more if you include .spec.ts testing files):

1. `app.component.css` - style
2. `app.component.html` - template
3. `app.component.ts` - class

However, that is not your only option. Here are more options.

1. **Include the style in the .ts class file – this is called an inline style and it saves you having to have a style file for the component. Use the CLI ‘—inline-style’ argument to generate components with inline styles.**
2. **Include the template in the .ts class file – this is called an inline template and it saves you having to have a template file for the component. Use the CLI ‘—inline-template’ argument to generate components with inline styles.**
3. **Include multiple component classes in the same file. Each component can have separate style and template files, or have them all in the same file like below:**

```

import { Component } from '@angular/core';
@Component({
selector: 'Paragraph',
template: `

<p><ng-content></ng-content></p> `,
styles: ['p { border: 1px solid #c0c0c0; padding: 10px }']
})
export class Paragraph {
}

@Component({
selector: 'app-root',
template: `

<p>
<Paragraph>Lorem ipsum dolor sit amet, consectetur adipiscing elit. </Paragraph> <Paragraph>Praesent eget ornare neque, vel consectetur eros. </Paragraph> </p>
`,
styles: ['p { border: 1px solid black }']
})
export class AppComponent {
title = 'app works!';
}

```

● Examples - Components and Files

You may find code examples with multiple components in the same file. This was done on purpose so that you could copy and paste more code into fewer files.

● Components and Modules

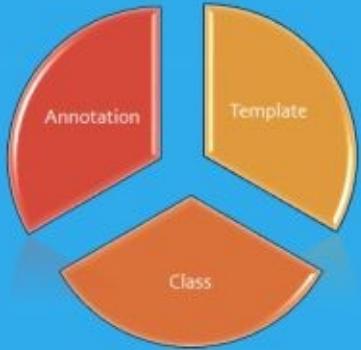
When you use components in your App, you need to ensure that each component is declared in modules. We will introduce modules in more detail in the next chapter. Below is an example of the module declaring two components: AppComponent and Paragraph.

```

import { AppComponent, Paragraph } from './app.component';
@NgModule({
declarations: [
  AppComponent,
  Paragraph
],
imports: [
  BrowserModule,
  FormsModule,
  HttpModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

An Angular2 Component Consists Of:



10.2

What Does a Component Consist Of?

- **Annotation.**

Provides meta-data to combine all the parts together into a component.

- **Template.**

Usually html markup. Used to render the component in the browser. The View in MVC. Can contain tags for nested components.

- **Class.**

Has annotations to add meta-data. Contains data (was \$scope). The Model in MVC. Contains code for behavior. This Controller in MVC.

10.3 Component Annotation

This annotation is located near the top of the class and is the most important element of it. It is a function that marks the class as a Component and accepts an object. It uses the object to provide meta-data to Angular about the Component and how to run it.

If you use the CLI to generate a project and you examine the generated component ‘app.component.ts’ then you will see the following @Component annotation:

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})
```

• Basic Elements

These are basic elements that you can add to the @Component annotation.

Annotation Element	Notes
selector	What markup tag, element this component corresponds to.
template / templateUrl	Specifies the template, which contains the markup for the Component. You have two options: <ol style="list-style-type: none">1. Use ‘template’ to specify the template inline in a block of quotes. This works great for simple templates.2. Use ‘templateUrl’ to specify the relative path to an external template file. This is better for larger or more complicated template.
styles / styleUrls	Specifies the css information for the template markup. You have two options. <ol style="list-style-type: none">1. Use ‘styles’ to specify an array of styles inline. This works great for just a couple of style definitions.2. Use ‘styleUrls’ to specify an array of relative paths to style definition files. This is better when you use a variety of styles.

• Selector Syntaxes

The selector syntax is like a JQuery selector:

Type	Example	Example of Selected Markup	Notes
Name	welcome	<welcome> </welcome>	This is the most common way of using the selector. Just make sure that this tag is unique and will never be used by html. It's often a good idea to use a common prefix for your project and all of the components therein. For example, a rewards program project could have the prefix ‘rp_’.

Id	#welcome	<div id='welcome'> </div>	
Css class	.welcome	'<div class='welcome'> </div>	

• Selectors and DSL

In Angular you can create Components and Directives that map to specific tags (or attributes). For example, if you are creating an application to sell cars, you could use tags and attributes like this: <CarSearch></CarSearch>, <CarList></CarList>, <CarDetail></CarDetail> etc. In effect, with Angular Components and Directives we are creating a DSL for our application. A domain-specific language (DSL) is a computer language specialized to a particular application domain. DSL's are very powerful because they allow the code to be specific to the domain of the application (i.e. its use) and represent in language form the business entities that they represent.

• Other Elements

These are other, more advanced elements that you can add to the `@Component` annotation. We will go into detail on many of these later on.

Annotation Element	Notes
animations	list of animations of this component
changeDetection	change detection strategy used by this component
encapsulation	style encapsulation strategy used by this component
entryComponents	list of components that are dynamically inserted into the view of this component
exportAs	name under which the component instance is exported in a template
hosts	map of class property to host element bindings for events, properties and attributes
Inputs	list of class property names to data-bind as component inputs
interpolation	custom interpolation markers used in this component's template
moduleId	ES/CommonJS module id of the file in which this component is defined
outputs	list of class property names that expose output events that others can subscribe to
providers	list of providers available to this component and its children

queries	configure queries that can be injected into the component
viewProviders	list of providers available to this component and its view children

10.4 Component Template s

● Introduction

The template contains the markup code to display the component in a web browser. Information about component's template is provided by the annotation.

● Template Location

The template markup can be included in the same file as the Component Class, or it can be in a separate file:

Template Markup Included Inline in Component Annotation

```
@Component({
  selector: 'app-root',
  template: `
    <div class='app'>
      [app]
      <app-customer-list>
        </app-customer-list>
    </div>
  `,
  styles: ['.app {background-color:#d5f4e6;margin:10px;padding:10px;}']
})
```

Template Markup Contained in Separate File

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

● Script Tags

The `<script>` tag is not allowed in a Component Template. It is forbidden, eliminating the risk of script injection attacks. In practice, `<script>` is ignored and a warning appears in the browser console.

In other words, never to this:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1>
      {{title}}
    </h1>
    <script>
      alert('app works');
    </script>
  `,
  styles: []
})
export class AppComponent {
  title = 'app works!';
```

● Elvis Operator

Angular also has issues with null values, especially with Template Expressions. For example, if we have object 'x' which is null and we have the following code: Total {{x.totalAmt}}

This will cause JavaScript and Zone issues (more on Zone later) and your Component will suddenly not render! I wish I had a dollar for every single time this happened to me!

Luckily for us the Elvis Operator helps us. Simply put a question mark in the template expression next to the variable that may be null. As soon as that variable is found to be null, the Elvis Operator tells the code to exit, leaving a blank. This stops the evaluation of the property and bypasses the JavaScript issue.

```
Total {{x?.totalAmt}}
```

Sometimes you need multiple Elvis Operators in a template expression.

```
Total {{x?.amt?.total}}
```

10.5 Component Styles

● Introduction

The styles contain the css rules required to change the component's style. Information about component's template is provided by the style annotation. You can specify the component's style in the component or in an external file. When you create an Angular CLI project, its style files are specified in the ‘.angular-cli.json’ file.

● Style Location

The styles can be included in the same file as the Component Class, or they can be in a separate file:

Style Markup Included Inline in Component Annotation

```
@Component({
  selector: 'app-root',
  template: `
    <div class='app'>
      [app]
      <app-customer-list>
      </app-customer-list>
    </div>
  `,
  styles: ['.app {background-color:#d5f4e6;margin:10px;padding:10px;}']
})
```

Style Markup Contained in Separate File

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

10.6 Component Class

This TypeScript class contains both the data and the code for the Component.

The data is contained in variables, which can be bound to the HTML markup in the template.

The code is can respond to user events (like clicking or on a button) or can invoke itself to start doing things.

(Yet again) this is only an introduction, we will go into Component Classes in more detail in Chapter ‘More Component’.

10.7 Model View Controller

MVC is a way of writing mostly for implementing [user interfaces](#) on computers. It divides a given software application into three interconnected parts: the Model (the data), the View (what the user sees) and Controller (the commands to update the model). So in this context of Angular, it could be said that the Model is the data inside the Component Class, the View is the Component Template and the Controller could be code in the Component Class.

10.8 Introduction to Data Binding

Data binding is what made Angular so popular. Binding is the synchronization of elements of the component UI widget to the data in your component classes, and the reverse also. This is managed by Angular for you. You setup variables in your component classes to store data and edit the HTML in your component template to add binding to that data. Now your HTML is no longer static, it changes with your data! If you want your Components to interact with the User, you must use Data Binding in them.

In terms of MVC, at runtime Angular uses Change Detection to ensure that the Components View always reflects the Components Model. With Data Binding you can control the user interface of your components by changing variables, and accept user input, allowing them to change the value of some variables. Data binding can control every aspect of the user interface: hiding things, closing things, showing results, accepting user input etc. It is incredibly powerful and easy to use.

• Example: Data Binding in a Login Component

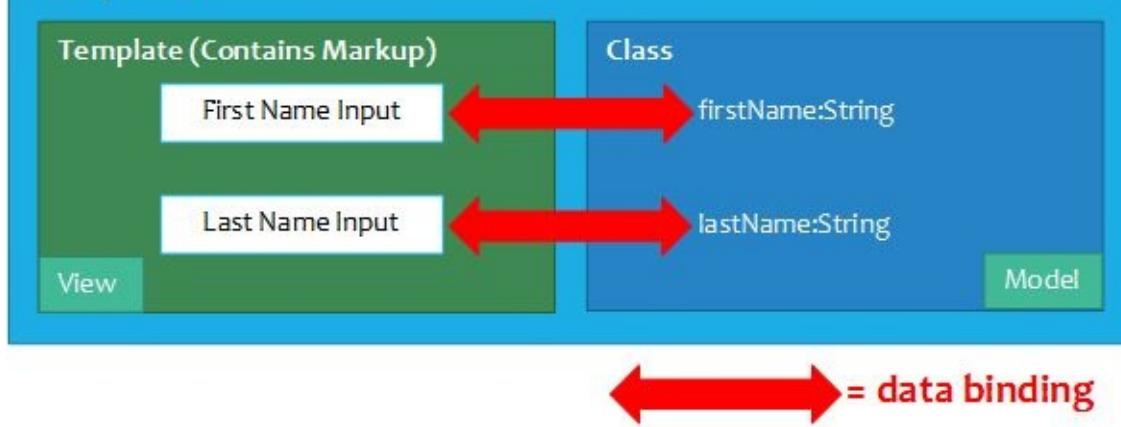
The form consists of a title 'Login' and two input fields. The first field is labeled 'Username' with a placeholder 'Username' and a red 'Required' validation message. The second field is labeled 'Password' with a placeholder 'Password' and a red 'Required' validation message. A 'Submit' button is at the bottom.

You could have a login form with two fields. Each field has a textbox in the html in the Component Template. Each field has a corresponding instance variable in the Component Class. The textboxes and the instance variables are bound to each other. If someone enters a username, the username instance variable is updated with the new value. When the developer codes the 'Submit' button, he or she gets the username and password from the instance variables, rather than having to extract them from the html.

• Example: Data Binding and Customer Data Input

You could have a form that enables you to input customer information using fields. Each field has a textbox in the html in the Component Template. Each field has a corresponding instance variable in the Component Class. Angular Data Binding enables you to have the instance variables updated automatically when the user inputs information into html fields. It also enables you to have the html fields updated automatically when you change the value of the instance variables, as well as having the instance variables updated automatically when the user types into the textboxes. When the developer wishes to default the value of the input fields, all he or she needs to do is set the instance variable values. The html textboxes will update automatically.

Component



● Types of Data Binding

There are two main types of databinding; one-way and two-way.

1. One way data-binding can occur when the template (the view) is automatically kept up to date with the latest values in the class instance variables (the model). Updates flow in only one direction.
2. One way data-binding can occur when the class instance variables (the model) is automatically kept up to date with values input from the template (the view). Updates flow in only one direction.
3. Two-way data binding is when the class instance variables (the model) and the template (the view) keep each other up to date. Updates flow in both directions.

10.9 One Way Data Binding with ‘{{’ and ‘}}’

● Introduction

Also known as ‘moustaches’ or ‘interpolation’.

The double curly braces are used for one way binding a template expression, making a calculation from available data in the Model and including it in the View. The expression produces a value and it is included in the View (the markup from the Component Template). The Model (i.e. data in the Component Class) is never updated.

● Template Expression

A template expression is usually a simple JavaScript expression.

Usually the Template Expression is just the name of a property in the model (i.e. an instance variable in the Component’s Class). Angular replaces that property name with the string value of the property (the string value of the instance variable).

Sometimes the Template Expression gets more complicated. Angular attempts to evaluate that expression (which can contain math, property names, method calls etc.) and converts the evaluation into a string. Then it replaces the contents and the curly braces with the result.

Examples of curly braces and template expressions:

`{{2+2}}`

`{{firstName}}`

`{{1 + 1 + getVal()}}`

10.10 One Way Data Binding - Example Code

This will be example ‘ch10-ex100’.



● Step 1 – Build the App using the CLI

Enter the command below. This will create a new Angular app in a folder called ‘Start’. It will also spew out lots of files that it creates.

```
ng new ch10-ex100 --inline-template --inline-style
```

● Step 2 – Start Ng Serve

```
cd ch10-ex100  
ng serve
```

● Step 3 – Open App

Open your web browser and browse to localhost:4200. You should see the text ‘app works’. That means that your project is running.

● Step 4 – Edit Component

Edit the file ‘src/app/app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `
<h1>
  {{title}}
</h1>
<p>
  Length: {{title.length}}
</p>
<p>
  Reversed: {{getReversed(title)}}
</p>
`,
styles: []
})
export class AppComponent {
title = 'app works!';

getReversed(str: string){
  let reversed = "";
  for(let i=str.length-1;i>=0;i--){
    reversed += str[i];
  }
  return reversed;
}}
```

```
reversed += str.substring(i,i+1);
}
return reversed;
}
}
```

● Exercise Complete

Your app should be working at localhost:4200. Note how the Template uses two expressions: one to show the length of the title and another to reverse the title using a method in the Class.

10.11 One Way Data Binding with '[' and ']' (or *)

● Introduction

The square braces can be used for one way binding. With these you can bind a template expression, making a calculation from available data in the Model and including it in the Data Binding Target.

You could also use the prefix '*' instead of the double square braces.

● Format:

[Data Binding Target] = "Template Expression"

Or:

*Data Binding Target = "Template Expression"

● Data Binding Target

This is something in the DOM (including Element Properties, Component Properties and Directive Properties) which can be bound to the result of the expression to the right side of the target.

Markup	Description
	Sets image source to property 'imageUrl' in Model.
<div [ngClass] = "{selected: isSelected}"></div>	Sets CSS class according to property 'isSelected' in Model.
<car-detail [car]="selectedCar"></car-detail>	Sets the 'car' attribute of the 'car detail' to property 'selectedCar' in the Model. The car-detail could be a component and this would pass information from the current template to that component using the 'car' attribute.
<button [style.color] = "isSpecial ? 'red' : 'green'">	Sets the button color according to property 'isSpecial' in the model.

● Template Expression

This is used to calculate a value from available data in the Model.

10.12 One Way Data Binding – Example Code #1

Doesnt work:



Works:



This will be example ‘ch10-ex200’.

- **Step 1 – Build the App using the CLI**

```
ng new ch10-ex200 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch10-ex200  
ng serve
```

- **Step 3 – Open App**

Open your web browser and browse to localhost:4200. You should see the text ‘app works’. That means that your project is running.

- **Step 4 – Edit Component**

Edit the file ‘src/app/app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1>Doesnt work:</h1>
    
    <h1>Works:</h1>
    <img [src]="starUrl">
  `,
  styles: []
})
export class AppComponent {
  starUrl = 'https://developer.mozilla.org/samples/cssref/images/starsolid.gif';
}
```

- **Exercise Complete**

Your app should be working at localhost:4200. Note the following:

The first image tag fails because it doesn’t wrap ‘src’. It takes the ‘startUrl’ literally rather

than calculating it as an expression from an instance variable.

The second image tag works because it wraps ‘src’ in square brackets, meaning that this is an expression that requires calculating the value of the ‘starUrl’ instance variable.

10.13 Html Element Attributes

Sometimes you need to dynamically create attributes in the html elements generated by your template. Example:

```
<li  
id="12345"  
data-make="bmw"  
data-model="m3"  
data-parent="cars">  
...  
</li>
```

In this case the ‘id’ tag is used to identify the element (very useful for JavaScript and CSS) and there are various data elements that store information.

In Angular you can use the ‘[attr.***name***]’ syntax to set attributes in the generated html.

10.14 One Way Data Binding – Example Code #2

- 2002 bmw m3 [View](#)
- 2017 acura nsx [View](#)
- 2016 chevy camaro [View](#)

This component will list some cars and let you click on the ‘View’ button to view an article about the car. The interesting thing about the component is that it stores attribute data in each element.

This will be example ‘ch10-ex250’.

● Step 1 – Build the App using the CLI

```
ng new ch10-ex250 --inline-template --inline-style
```

● Step 2 – Start Ng Serve

```
cd ch10-ex250  
ng serve
```

● Step 3 – Open App

Open your web browser and browse to localhost:4200. You should see the text ‘app works’. That means that your project is running.

● Step 4 – Edit Component

Edit the file ‘src/app/app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { Car } from './car';
@Component({
selector: 'app-root',
template: `<ul>
<li *ngFor="let car of _cars"> <span [attr.id]="car.id" [attr.data-desc]="car.make + ' ' + car.model" [attr.data-article]="car.article">
{{car.year}}&nbsp;{{car.make}}&nbsp;{{car.model}}&nbsp;<button (click)="showCar($event)">View</button></span> </li>
</ul>
`,
styles: []
})
export class AppComponent {
_cars = [
  new Car('car1', 2002, 'bmw', 'm3', 'https://en.wikipedia.org/wiki/BMW_M3'), new Car('car2', 2017, 'acura', 'nsx',
'https://en.wikipedia.org/wiki/Honda_NSX'), new Car('car3', 2016, 'chevy', 'camaro', 'https://en.wikipedia.org/wiki/Chevrolet_Camaro') ];

showCar(event){
  const desc = event.target.parentElement.dataset.desc; if (window.confirm('If you click "ok" you would be redirected to an article about
the ' +
desc + '. Cancel will load this website ')) {
```

```
window.location.href=event.target.parentElement.dataset.article; };
```

```
}
```

```
}
```

● Step 5 – Edit Class

Edit the file ‘src/app/car.ts’ and change it to the following:

```
export class Car {  
    constructor(  
        private _id: string,  
        private _year: number,  
        private _make: string,  
        private _model: string,  
        private _article: string){  
    }  
  
    public get id(): string {  
        return this._id;  
    }  
  
    public get year(): number {  
        return this._year;  
    }  
  
    public get make(): string {  
        return this._make;  
    }  
  
    public get model(): string {  
        return this._model;  
    }  
  
    public get article(): string {  
        return this._article;  
    }  
}
```

● Exercise Complete

Note how the ‘desc’ data attribute is generated: [attr.data-desc] = "car.make + ' ' + car.model"

Note how the JavaScript is used to get the ‘desc’ data attribute: const desc = event.target.parentElement.dataset.desc;

10.15 Two Way Data Binding with '[() and ')']'

● Introduction

'[()]' is also known as 'banana in a box'.

We have already seen this. The '[() and ')']' format is used for two way binding a Property, in other words reading it and writing it from the Model.

Format: [(Data Binding Target)] = "Property"

● Data Binding Target

This is something in the DOM (including Component and Directive tags) which can be bound to the property of the expression to the right side of the target. For the input box, the data binding target is 'ngModel', which corresponds to the text in the input box.

● Property

This is a Property in the Model (an Instance Variable in the Component Class).

10.16 Two Way Data Binding – Example Code



This is a component that changes foreground and background colors when the user changes their input. This will be example ‘ch10-ex300’.

- **Step 1 – Build the App using the CLI**

```
ng new ch10-ex300 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch10-ex300  
ng serve
```

- **Step 3 – Open App**

Open your web browser and browse to localhost:4200. You should see the text ‘app works’. That means that your project is running.

- **Step 4 – Edit Component**

Edit the file ‘src/app/app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `

<p>
Foreground: <input [(ngModel)]="fg" /> </p>
<p>
Background: <input [(ngModel)]="bg" /> </p>
<div [ngStyle]="{{'color': fg, 'background-color': bg, 'padding': '5px'}}> Test
</div>
` ,
styles: []
})
export class AppComponent {
fg = "#ffffff";
bg = "#000000";
}
```

- **Exercise Complete**

Your app should be working at localhost:4200. When the user changes the color value, this updates the model, which then updates the template’s html.

Binding occurs from the input field to the model (when the user changes the color values). When the input field changes, the model updates to match.

Binding occurs from the model to the template's html. When the model updates, the template's html updates to match.

10.17 Event Handling

● Introduction

A user interface needs to respond to user input. That is why we have event handling in our Component Templates. We specify a target event and what statement to happen when that event occurs.

Format: (Target Event) = "Template Statement"

Target Event

This is the name of the event in between curly brackets. Remember that the round brackets are used to imply event observation.

Template Statement

This is an instruction of what to do when the Target Event occurs. Normally this is a call to a method in the Component Class that does something, normally modifying instance variables that are bound to the template, causing a change in the UI. The event information is available in the \$event variable, which may or may not be utilized. For example, if you are watching for input on a textbox then you could pass the value of the text in the textbox to the method using information from the \$event. You will see this in the example

10.18 Event Handling – Example Code

hello Mark

Upper-Case: HELLO MARK
Lower-Case: hello mark

It accepts input in a textbox, captures the input event and displays the input in both upper and lower-case. This will be example ‘ch10-ex400’.

• Step 1 – Build the App using the CLI

```
ng new ch10-ex400 --inline-template --inline-style
```

• Step 2 – Start Ng Serve

```
cd ch10-ex400  
ng serve
```

• Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

• Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, AfterViewInit, ViewChild } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  template: `  
    <input #input type="text" (input)="textInput($event)" value="" /> <hr>  
    Upper-Case: {{upperCase}}  
    <br/>  
    Lower-Case: {{lowerCase}}  
    `,  
  styles: []  
})  
export class AppComponent implements AfterViewInit{  
  upperCase: string = "";  
  lowerCase: string = "";  
  @ViewChild('input') inputBox;  
  
  textInput(event){  
    this.upperCase = event.target.value.toUpperCase(); this.lowerCase = event.target.value.toLowerCase(); }  
  
  ngAfterViewInit() {  
    this.inputBox.nativeElement.focus()  
  }  
}
```

• Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. A template variable ‘#input’ and viewChild are used to get a reference to the input box. After the view is initialized (lifecycle method ‘ngAfterViewInit’ is fired), focus is set to the input box.
2. The template uses the following code to listen for the ‘input’ event, firing the method ‘textInput’ (passing in the event object) when it occurs.

```
(input)="textInput($event)"
```

3. The class has a method ‘textInput’ which is fired by the ‘input event’. It calculates the upper-case and lower-case versions of the user’s input, which it sets to instance variables which are bound (one way) from the class to the template.

11 Introducing Modules

11.1 Introduction

The word **module** refers to small units of independent, reusable code. A typical module is a cohesive block of code dedicated to a single purpose. A module **exports** something of value in that code, typically one thing such as an object.

This chapter is all about introducing the concepts of the different modules. It will not include many coding examples as you will be coding modules later on.

11.2 Why Modules?

JavaScript gives you freedom to do many things very badly – you are under no obligation to write re-usable code. You can scatter your code anywhere.

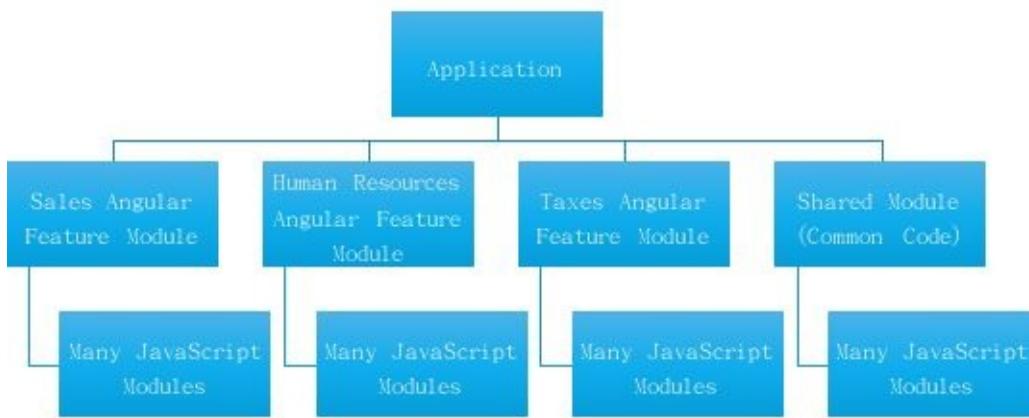
This has to change now that JavaScript and its environment is maturing. You need to make objects simpler by concealing their internal workings and just leaving public interfaces available from the outside. You need to be able to package code up into re-usable blocks which can be packaged and deployed separately from each other. You also need the ability to load them on demand, rather than loading everything up (slowly) when the app starts.

11.3 Different Types of Modules

This chapter is going to introduce the three types of ways AngularJS, Angular and JavaScript have modularized code:

1. The AngularJS Module System that was included in the original version of Angular. This enabled you to modularize your code at a course-grained level.
2. JavaScript modules are now available in ES6 & Typescript. This enables you to modularize your code at a fine-grained level. Remember there is 1 module per source code file (.ts or .js).
3. The Angular Module System that is now included in Angular. This enables you to modularize your code at a course-grained level. You can bundle units of Angular code into modules. For example, if you are writing a system in Angular that contains an application for Sales, an application for Human Resources and an application for Taxes, then you could split down these three applications into separate feature modules and a shared module for sharing common code.

• Angular Applications Are Made Up of Angular Modules & JavaScript Modules



11.4 AngularJS Module System

Remember this is for the original Angular. Disregard it if you think this information is irrelevant. AngularJS had its own module system which was simple. You had an angular module, which could contain angular controllers, directives etc.

```
angular.module('xxx', [ //services.config,
    'ngCookies',
    'ngRoute',
    'ngResource',
    'ngSanitize',
    'angularSpinner',
    'ui.bootstrap.demo',
    'ui.bootstrap',
    'ui.select',
    'wj',
    'angularModalService'

])
```

Above we are declaring module ‘xxx’ which depends on many other modules: ngCookies, ngRoute, ngResource, ngSanitize, angularSpinner, ui.bootstrap.demo, ui.bootstrap, ui.select, wj and angularModalService.

Then underneath the code below we would declare the items within this module ‘xxx’.

11.5 JavaScript Modules

● JavaScript Libraries

JavaScript used to work with libraries and these libraries were very useful for helping a developer in an area of development. For example, JQuery used to help developers with UI development. However, these libraries were very well-written but not implemented as modules. They were implemented as JavaScript scripts (as in ‘.js’ script files) that would create JavaScript objects to do things. They did not use the JavaScript module system because it was not around then.

● Introduction

Now ES6 and later supports modules. In Javascript Modules, every file is one module. You can code your own modules or you can use other people’s modules. You can use node pull dependent modules into your project (into the ‘node_modules’ folder).

● Importing and Exporting

When we code in Angular 4 in TypeScript we use the two JavaScript Module keywords below:

Export	Export module code.
Import	Import module code.

● Exporting Code

We write our application as a collection of small modules. Our code exports objects from the module to the outside world using the ‘export’ keyword.

For example, the code below is used to tell TypeScript that we are exporting class ‘App’ for use elsewhere.

```
export class App {...}
```

Export a Default Object from a Module

```
module "foo" {
export default function() { console.log("hello!") }
}
```

● Importing Code

We have import statements to tell TypeScript to go get module code from somewhere. The ‘somewhere’ can be from someone else’s module or from local code in the same project.

Importing Code from Someone Else’s Module

When we use import statements to go get code from someone else’s module we specify the module

name and the name of the item we wish to import. You specify the module name after the ‘from’.

For example, we are importing the Component from Angular:

```
import { Component } from '@angular/core'; For example, we are importing the Date Picker from  
ngx-bootstrap:
```

```
import { DatepickerModule } from 'ngx-bootstrap-datepicker';
```

Importing Your Project Code

When we import code from local code in the same project, we specify a relative path to that code. In the example below we specify a relative path (the ‘./’ below). This tells TypeScript that the code is in the same folder as the code that is going to use that module.

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

● More Import Syntaxes

Import All

```
import * as myModule from 'my-module';
```

Named Import

This name needs to **exactly** match the name of an object exported in the module.

```
import { myMember } from 'my-module';
```

Multiple Named Imports from a Module

These names need to **exactly** match the names of objects exported in the module.

```
import { foo, bar } from 'my-module';
```

Default Import from a Module

The name does not need to match any object exported in the module. It can be an alias. It knows it has to import the default object from the module.

```
import myDefault from 'my-module';
```

11.6 Angular Module System

● Introduction

This is the way Angular bundles code into re-usable modules. The Angular system code itself is modularized using this module system. Many third-parties provide additional functionality to Angular using modules, which you can easily include into your application.

● Why Does Angular Have Its Own Modules?

So why does Angular not just use JavaScript modules. Why does it force developers into using its own module system?

Well for a start it uses standard JavaScript modules but they don't go far enough – they don't make it easy for Angular to declare logical blocks of Angular code that consist of disparate objects tied together – for example components, services, pipes. In the earlier days of Angular 2, developers did not have the option of Angular modules and developers used module loaders to load and start applications (System.js). To me, it did not work well in practice. It was hard to learn, easy to break and too complicated.

I worked on Angular2 when it was in beta and liked the product, hated how complicated it was with module loading. I came back to it later to find how you could quickly and simply use the Angular CLI to build a modularized application that used Webpack for deployment. I welcome the Angular Module system and think it works elegantly with the CLI and Webpack.

● Start Project

We have already used the Angular Module System, even if you don't know it. If you open the 'Start' project that you created using the CLI you will see that you already have a file 'app.module.ts'. Note that modules should have '.module' in the name. Let's open it up and take a look.

```
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 { }
```

● @NgModule Annotation

This annotation is the most important part of this class. It is a function that accepts an object and uses the object to provide meta-data to Angular about the module: how to compile it and how to run it. So the @NgModule is Angular's declarative way for you to tell Angular how to put the pieces together. Note that the @NgModule itself needs to be imported from Angular core at the top.

Declarations

This should be an array of the Angular Components, Directives and Pipes used by your module and nothing else, no ES6 classes or anything else. When you add a Component using the CLI Command ‘ng generate component’, it imports the Component and adds it to this list of declarations. If you add a Component and use it without declaring it here, you will receive an error message in the browser console.

Imports

This should be an array of Angular modules required by the application here. These modules must be defined using @NgModule. Angular itself has many system modules that you will find useful and the CLI includes several of these for you by default, including the browser module, the forms module and the http module.

Providers

This should be an array of Angular provider objects required by the application. These provider objects are services classes and values that are injected into your classes for you using Dependency Injection. If you had a common service object used by the Components to talk to the server, you would add it here as a provider.

Bootstrap

You can use modules to contain the code for your application. To run, your application needs to know how to start and with what Component it should start (the root). This is where you specify the root component, which will be created and mounted into the html when the application starts. This root component is often called AppComponent.

● Root Module

Your Angular application can contain multiple modules. However, it always has a starting point, a module that it uses to bootstrap itself. This is the root module and it is often called AppModule.

● Routing Module

We will go into routing later on but routing is very important to an Angular application. It allows the user to map components to URLs and navigate the User Interface. When we use the CLI to build an Angular application, it builds a separate module for your application’s routing, usually in the file ‘app-routing.ts’. This may seem superfluous but it very neatly packages the Angular routing objects together with your app’s routing setup together into one module, which handles all routing for your app.

● Feature Modules

Bounded Contexts

Domain-driven design (DDD) is an approach to software development for complex needs by connecting the implementation to an evolving model. Domain Driven Design often has to deal with modelling very large complex business requirements and its approach to this is to break these requirements into Contexts. Bounded Contexts are areas of business requirements that can be logically separated. For an example please refer to the ‘shamelessly copied’ diagram below:



As you can see you have two contexts: Sales and Support.

Each one could be a separate part of your Angular application. In fact, each one could be contained in its own separate module, a Feature Module. Each module can contain specific code to meet specific requirements not required anywhere else. For example, the Sales module could contain an Angular UI to manage the sales pipeline and this would not be used anywhere else. So, feature modules often contain code that is not intended to be used outside that module.

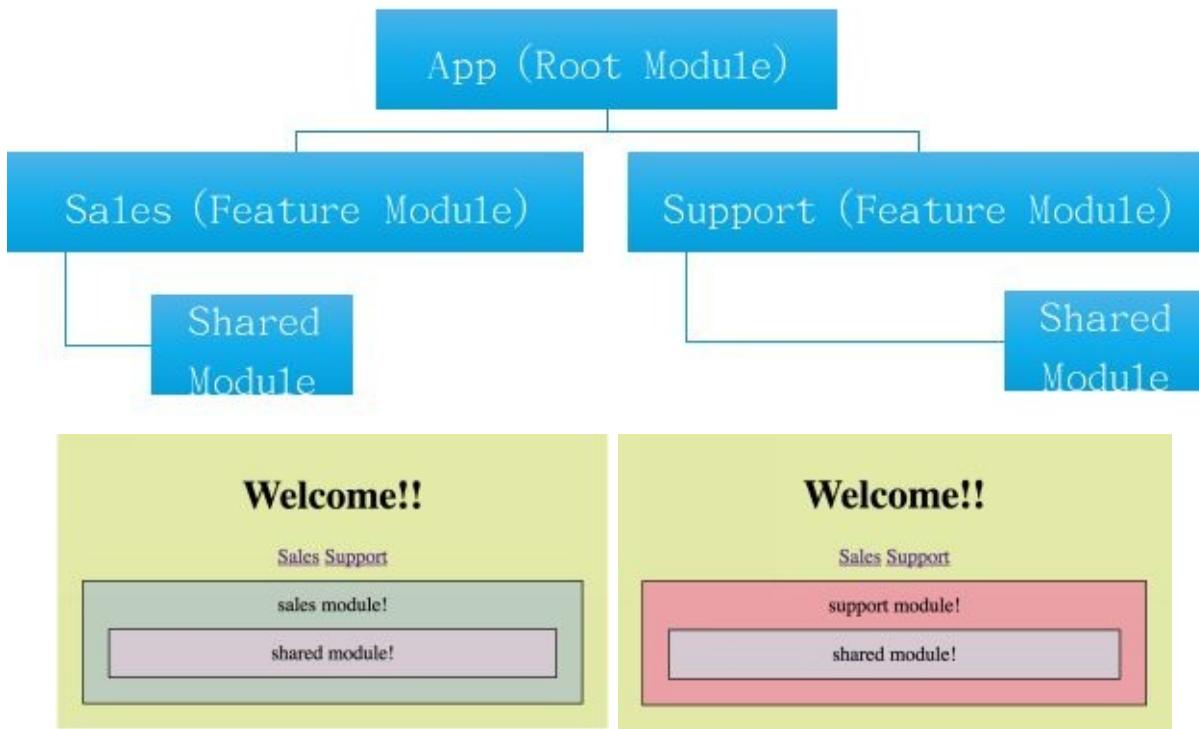
When required, the Root Module can include as many Feature Modules as required. The Feature Module can even be loaded on demand when the user (for example) clicks on the ‘Sales’ menu.

● Shared Modules

You can think of Feature Modules as blocks of code that are not shared. Shared modules are the opposite – they contain the most commonly-used code that is modularized so that it can be re-used as much as possible. When required, the Root Module can include as many Shared Modules as required.

11.7 Angular Module System – Example

This example is a very basic exercise of how you can use root modules, feature modules and shared modules together. This example has a root module ‘App’ and two feature modules: ‘Sales’ and ‘Support’. Both use a component from a shared module ‘Shared’.



- **Step 1 – Build the App using the CLI**

```
ng new ch11-ex100
```

- **Step 2 – Start Ng Serve**

```
cd ch11-ex100  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Generate Modules**

Let’s use the CLI to generate additional modules.

```
ng generate module shared  
ng generate module routing --routing  
ng generate module sales  
ng generate module support
```

- **Step 5 – Generate Components**

Let’s use the CLI to generate additional components.

```
ng generate component sales  
ng generate component support  
ng generate component shared
```

● Step 6 - Edit Component Styles

Edit the file ‘sales.component.css’ and change it to the following:

```
div {  
  background-color: #bdcebe;  
  border: 1px solid #000000;  
  padding: 10px;  
  margin: 10px;  
}
```

Edit the file ‘support.component.css’ and change it to the following:

```
div {  
  background-color: #eca1a6;  
  border: 1px solid #000000;  
  padding: 10px;  
  margin: 10px;  
}
```

Edit the file ‘shared.component.css’ and change it to the following:

```
div {  
  background-color: #d6cbd3;  
  border: 1px solid #000000;  
  padding: 10px;  
  margin: 10px;  
}
```

Edit the file ‘app.component.css’ and change it to the following:

```
div {  
  background-color: #e3eaa7;  
  border: 10px;  
  padding: 10px;  
}
```

● Step 7 - Edit Component Templates

Edit the file ‘sales.component.html’ and change it to the following:

```
<div>  
sales module!  
<app-shared></app-shared>  
</div>
```

Edit the file ‘support.component.html’ and change it to the following:

```
<div>  
support module!  
<app-shared></app-shared>  
</div>
```

Edit the file ‘shared.component.html’ and change it to the following:

```
<div>
shared module!
</div>
```

Edit the file ‘app.component.html’ and change it to the following:

```
<div style="text-align:center"> <h1>
    Welcome!!
</h1>
<a [routerLink]="'[sales']">Sales</a> <a [routerLink]="'[support']">Support</a> <router-outlet></router-outlet> </div>
```

● Step 8 - Edit Routing Module

Edit the file ‘routing.module.ts’ and change it to the following:

```
import { NgModule } from '@angular/core'; import { CommonModule } from '@angular/common';
import { Routes, RouterModule } from '@angular/router'; import { SalesComponent } from './sales/sales.component'; import {
  SupportComponent } from './support/support.component';
const routes: Routes = [
{
  path: 'sales',
  component: SalesComponent
},
{
  path: 'support',
  component: SupportComponent
},
{
  path: '**',
  component: SalesComponent
}
];

@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule],
providers: []
})
export classRoutingModule { }
```

● Step 9 - Edit Sales Module

Edit the file ‘sales.module.ts’ and change it to the following:

```
import { NgModule } from '@angular/core'; import { CommonModule } from '@angular/common';
import { SalesComponent } from './sales.component';

@NgModule({
imports: [
  CommonModule
],
declarations: [SalesComponent]
})
export class SalesModule { }
```

● Step 10 - Edit Shared Module

Edit the file ‘shared.module.ts’ and change it to the following:

```
import { NgModule } from '@angular/core'; import { CommonModule } from '@angular/common';
import { SharedComponent } from './shared.component';

@NgModule({
  imports: [
    CommonModule
  ],
  exports: [
    SharedComponent
  ],
  declarations: [SharedComponent]
})
export class SharedModule { }
```

● Step 11 - Edit Support Module

Edit the file ‘support.module.ts’ and change it to the following:

```
import { NgModule } from '@angular/core'; import { CommonModule } from '@angular/common';
import { SupportComponent } from './support.component';
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [SupportComponent]
})
export class SupportModule { }
```

● Step 12 - Edit App Module

Edit the file ‘app.module.ts’ and change it to the following:

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

import { AppComponent } from './app.component';
import {RoutingModule} from './routing/routing.module'; import { SalesComponent } from './sales/sales.component'; import {
  SupportComponent } from './support/support.component'; import { SharedModule } from './shared/shared.module';
@NgModule({
  declarations: [
    AppComponent,
    SalesComponent,
    SupportComponent
  ],
  imports: [
    BrowserModule,
    RouterModule,
    SharedModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

The Routing Module provides the code for the app routing. The root app imports this module and all the routing code is ready and usable.

The Shared Module provides the shared component. The root app only has to import this module to get access to its component.

The Sales and Support modules don't have to import the Shared Module or the Shared Component, even though it is used in the Sales and Support components.

The Sales and Support modules don't import anything other than the Angular Common Module. This Common Module is nothing to do with our code. This is Angular's way of providing the code for basic Angular directives like NgIf, NgFor etc.

11.8 Deployment - Separate Modules

The ability to have feature modules, shared modules etc accessible from one root module sounds great but the problem is that you might need to update the feature modules separately, especially if you have a separate team working on each feature. For example, the ‘sales’ people may have different release dates from the ‘support’ people. Unfortunately, the example above is deployed in one group of Webpack modules.

If you want ‘Sales’ to be deployable separately from ‘Support’ then each one should be its own single page application in its own folder. That makes life much easier when it is time to deploy.

11.9 Deployment – Node

• Using Node to Manage Dependencies on Common Code

Another issue with deployment is the common code. ‘Sales’ may need a different version of the common code than ‘Support’. One may use new common objects and the other may not. That makes it a good time to consider using node to manage each project’s dependency on common code.

You can create Node modules from Angular projects. Remember Angular itself has modules that are deployed through Node modules. This is a little beyond the scope of this book but I have done this myself, thanks to the (superb) article below: <https://medium.com/@cyrilletuzi/how-to-build-and-publish-an-angular-module-7ad19c0b4464>

You will need to setup some code of public code repository for this to work, like github.

Here is a (very simple) example that I put up on github:

<https://github.com/markclowisg/sharedcomponents>

• Useful Node Commands

When working with Angular and Node together, you might also want to consider using the node package manager commands ‘npm link’ and ‘npm scope’.

Npm Link

This is very useful when it comes time to build your node modules. It lets you setup a link so that dependent projects can use your node code without it having to be continually rebuilt and redeployed to your repository.

Npm Scope

This is useful when you have several npm ‘common code’ projects and you want to group them under a name prefix. Angular does this with its ‘@angular’ npm package prefix. You may want to consider this. If you work for company ‘abc’ and you have two common npm packages for components and services, you may want to use scopes so they can be ‘@abc/components’ and ‘@abc/services’.

12 Introducing Webpack

12.1 Introduction

Nowadays you can do a lot more stuff in modern browsers, and this is going to increase even more in the future! Thanks to technologies like Angular 4 there will be fewer page reloads and more JavaScript code in each page, a lot of code on the client-side. So, you need a way to deploy all of this code efficiently so that it loads quickly.

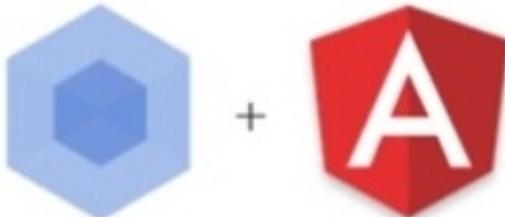
Your complex client-side application may contain modules, some of which may load synchronously, some asynchronously. So how do we package it all and deploy it most efficiently – we use Webpack!

12.2 Webpack and the Angular CLI

The Angular CLI uses Webpack to transpile, compile and deploy project code. However later on in this chapter you will see we mention Webpack configuration and the ‘webpack.config.js’. Then you will look for it in your project and notice that it is missing. This is on purpose because the people who wrote the Angular CLI wanted to hide as many configuration details as possible to make things simpler. And this included the Webpack configuration.

There is an Angular CLI command to make the Webpack configuration file available: `ng eject`. However, I would take care with this command as there may be some unexpected side-effects. More information here: <https://github.com/angular/angular-cli/wiki/eject>

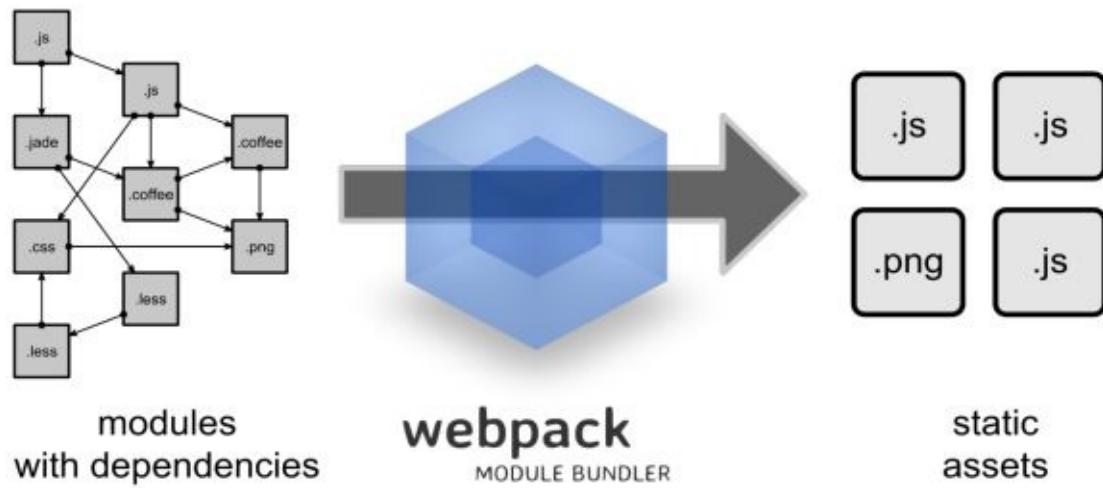
<http://stackoverflow.com/questions/39187556/angular-cli-where-is-webpack-config-js-file-new->



2017-feb-ng-eject

12.3 What Does Webpack Do?

Webpack is a module bundler. It takes modules with dependencies and generates static assets representing those modules.



12.4 What about Your Modules and Dependencies?

If you use Node for your development then Webpack will read your Node configuration file ‘`packages.json`’ and automatically include your dependencies as static assets in the build.

This takes away the pain of configuring module loading and deployment. You don’t have to figure anything out. I have used Webpack on every Angular 4 project I have worked on because it makes life easier.

```
},  
  "dependencies": {  
    "angular2": "2.0.0-beta.0",  
    "es6-promise": "^3.0.2",  
    "es6-shim": "^0.33.3",  
    "es7-reflect-metadata": "^1.2.0",  
    "rxjs": "5.0.0-beta.0",  
    "zone.js": "0.5.10",  
    "ng2-bootstrap": "1.0.0-beta.1"  
  },  
  12.5 }
```

Benefits

Webpack is good for large projects because it allows for development and production modes. Development mode can utilize non-minimized assets like JavaScript, enabling your application to be debugged in this mode. Production mode can use minimized assets so it has a lighter footprint.

12.6 Chunks

Your code base can be split into multiple chunks and those chunks can be loaded on demand reducing the initial loading time of your application. Quicker loading times. As a developer, you also have control over configuring these chunks (see later).

12.7 Development Process

1. Code your project.
2. Run Webpack as part of your build process (or have it run for you by the CLI).
3. After the build then you have all your static assets ready to deploy on the server.

12.8 Install Webpack

You don't need to install Webpack if you are using the CLI. Webpack runs under Node.

However, if you want to experiment with Webpack separately, you can use the following command line to install it (remember to be in the root folder of your project).

```
npm install webpack -g
```

12.9 Configuration

If you run the ‘ng eject’ command mentioned earlier in this chapter, your Webpack options will be contained in the ‘webpack.config.js’ file in the root folder of your project. In this file:

● Output Path

You can specify where the bundled assets are put, the output path.

● Entry Points

Your app can start in different places using different code.

Webpack will pack the code for deployment so that it can start with these different codes but share common packaged chunks.

● Loaders

A loader is a node function takes a type of file and converts files of this type into a new source for bundling. Loaders are basically node packages used by Webpack.

```
loaders: [
  // Support for .ts files.
  {
    test: /\.ts$/,
    loader: 'ts-loader',
    query: {
      'ignoreDiagnostics': [
        2403, // 2403 -> Subsequent variable declarations
        2300, // 2300 -> Duplicate identifier
        2374, // 2374 -> Duplicate number index signature
        2375 // 2375 -> Duplicate string index signature
      ],
      exclude: [ /\.spec|e2e\.\ts$/], /node_modules\/(?!ng2-.+)/
    },
  },
]
```

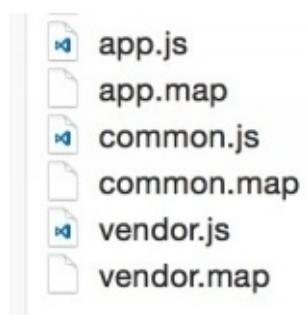
● Plugins

We use the ‘CommonsChunk’ plugin In the book’s example project to split our code into deployable chunks which can be loaded separately.

The job of the CommonsChunk plugin is to determine which modules (or chunks) of code you use the most, and pull them out into a separate file. That way you can have a common file that contains both CSS and JavaScript that every page in your application needs.

```
plugins: [
  new CommonsChunkPlugin({ name: 'vendor', filename: 'vendor.js', minChunks: Infinity }),
  new CommonsChunkPlugin({ name: 'common', filename: 'common.js', minChunks: 2, chunks: ['app', 'vendor'] })
  // include uglify in production
],
```

Is used to create:



13 Introducing Directives

13.1 Introduction

Now it's the time to introduce Directives. Directives are markers on a DOM element (such as an attribute) that tell Angular to attach a specified behavior to an existing element.

Directives have been around since AngularJS, but they are quite complex to use. They have got a lot easier to use in Angular, especially in the area of passing data into the Directives. They used to be the main way of creating custom tags in an AngularJS application. Now they have been replaced by Directives and Components.

Angular itself provides many directives to help you in your coding. You can also code your own.

13.2 Components May Use Directives

As we mentioned earlier we can think of Components having three main elements:

1. The Annotation, which provides Angular with meta-data to combine all the parts together into a component.
2. The Template, which contains markup (usually html), which is used to render the component in the browser.
3. The Class, which contains the data and code for the component. The code implements the desired behavior of the Component.

● Directives are Used by The Template

As you can see, the template is used to generate the markup for the display of the component. This markup may include the tags (or other selectors) for other Angular Components, thus allowing the Composition of Components from other Components. However, this markup may also include Directives to implement certain behaviors.

For example, you may have a component that displays the promotion details of a promotion request. However, the person viewing the promotion request may not have the rights to view the information, thus some elements should be hidden. You could use the Angular ‘ngIf’ directive to evaluate the user’s rights and hide or show elements based on them.

13.3 Types of Directives

So, we now know that Directives are used by the Component templates. However, they may affect the output of the template in different ways.

Some Directives may completely change the structure of the output of the template. These directives can change the DOM layout by adding and removing view DOM elements. Let's call these Structural Directives.

Some Directives may simply change the appearance of items output by the template. Let's call these Non-Structural Directives.

• **Structural Directives**

Angular includes several structural directives for you to use in the template: NgIf NgFor NgSwitch, NgSwitch, NgSwitchWhen, NgSwitchDefault

• **Non-Structural Directives**

Angular includes several non-structural directives for use in the template: NgClass NgStyle NgControlName NgModel

13.4 NgIf

This is a directive that you add to an element in the markup, usually to a container element like a ‘div’.

If the template expression for the ‘ngIf’ is true, then the content inside the element is included in the view DOM after the bindings have been completed.

If the template expression for the ‘ngIf’ is false, then the content inside the element is excluded from the view DOM after the bindings have been completed.

So, this directive is used to include or exclude an element of the UI, including the elements child elements.

13.5 NgIf – Example

Let's use ngIf to hide and show elements. In this example, we toggle between showing a name and an address. This will be example 'ch13-ex100'.

Name: Mark

Toggle

Address: Atlanta

Toggle

- **Step 1 – Build the App using the CLI**

```
ng new ch13-ex100
```

- **Step 2 – Start Ng Serve**

```
cd ch13-ex100  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- **Step 4 – Edit Class**

Edit 'app.component.ts' and change it to the following:

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styles: ['div.box { width: 200px; padding: 20px; margin: 20px; border: 1px solid black; color: white; background-color: green }']
})
export class AppComponent {
showName: boolean = true;

toggle(){
  this.showName = !this.showName;
}
}
```

- **Step 5 – Edit Template**

Edit ‘app.component.html’ and change it to the following:

```
<div *ngIf="this.showName" class="box"> Name: Mark
</div>
<div *ngIf="!this.showName" class="box"> Address: Atlanta
</div>
<button (click)="this.toggle()">Toggle</button>
```

13.6 NgFor

This is a directive for processing each item of an iterable object, outputting a markup for each one.

This is known as a Structural directive because it can change the DOM layout by adding and removing view DOM elements.

This directive is useful for generating repeating content, such as a list of customers, elements of a dropdown etc.

Each item processed of the iterable has the following variables available in its template context:

Variable	Description
Item itself	Example: ngFor="#name of names". In this case the item has the variable 'name'.
Index	Current loop iteration for each template context.
last	Boolean value indicating whether the item is the last one in the iteration.
even	Boolean value indicating whether this item has an even index.
odd	Boolean value indicating whether this item has an odd index.

13.7 NgFor – Example

Let's use ngFor to show a list. This will be example 'ch13-ex200'.

0: Peter Falk
1: Mary-Ann Blige
2: Eminem

• Step 1 – Build the App using the CLI

```
ng new ch13-ex200
```

• Step 2 – Start Ng Serve

```
cd ch13-ex200  
ng serve
```

• Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see 'app works!'.

• Step 4 – Edit Class

Edit 'app.component.ts' and change it to the following:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  names = [
    'Peter Falk', 'Mary-Ann Blige', 'Eminem'];
}
```

• Step 5 – Edit Template

Edit 'app.component.html' and change it to the following:

```
<div *ngFor="let name of names; let i = index;"> <div>{{i}}: {{name}}</div>
</div>
```

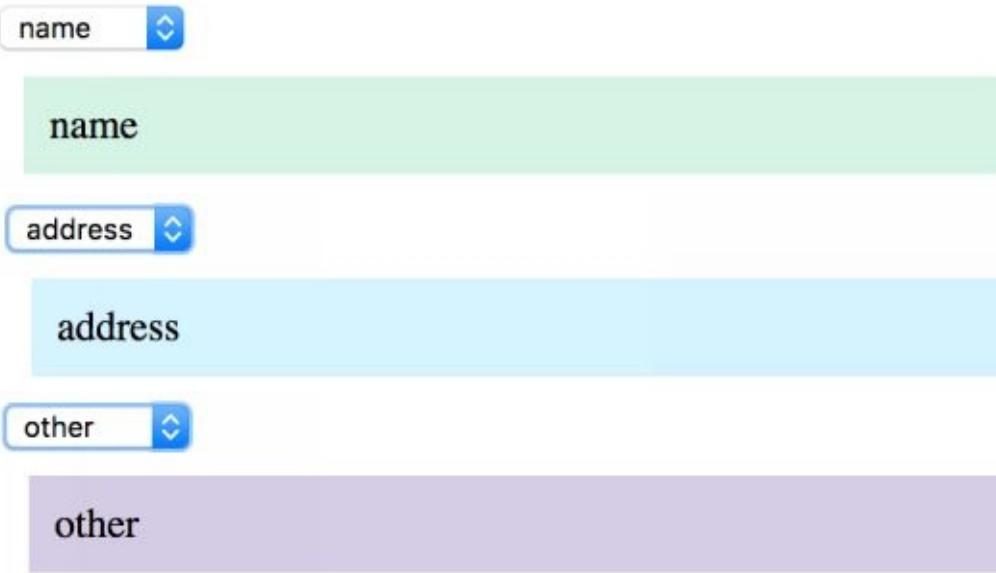
13.8 NgSwitch, NgSwitchWhen, NgSwitchDefault

This is a directive for adding or removing DOM elements when they match switch expressions.

This is known as a Structural directive because it can change the DOM layout by adding and removing view DOM elements.

13.9 NgSwitch – Example

Let's use ngSwitch to hide and show elements according to your selection. This will be example 'ch13-ex300'.



- **Step 1 – Build the App using the CLI**

```
ng new ch13-ex300
```

- **Step 2 – Start Ng Serve**

```
cd ch13-ex300  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- **Step 4 – Edit Class**

Edit 'app.component.ts' and change it to the following:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: ['.block1 {background-color:#d5f4e6;margin:10px;padding:10px;}', '.block2 {background-color:#d5f4ff;margin:10px;padding:10px;}', '.block3 {background-color:#d5cce6;margin:10px;padding:10px;}']
})
export class AppComponent {
  selection = 'name';
  options = ['name','address','other'];
}
```

- **Step 5 – Edit Template**

Edit 'app.component.html' and change it to the following:

```
<select [(ngModel)]="selection"> <option *ngFor="let option of options">{ {option} }</option> </select>
<div [ngSwitch]="selection">
<div class="block1" *ngSwitchCase="options[0]">name</div> <div class="block2" *ngSwitchCase="options[1]">address</div> <div
class="block3" *ngSwitchDefault>other</div> </div>
```

13.10 NgClass

We can change the appearance of DOM elements by adding or removing classes using this directive. Its argument is an object that contains pairs of the following: a css class name an expression. The css class name is added to the target DOM element if the expression is true, otherwise it is omitted. It is not useful for just simply setting a css class. It's probably easier to use something like the code below: <div [class]="classNames">Customer {{name}}.</div>

13.11 NgClass - Example

It lets the user click on an animal in an animal list to select it. The selected animal is highlighted in red. This will be example ‘ch13-ex400’.

cat

dog

zebra

giraffe

● Step 1 – Build the App using the CLI

```
ng new ch13-ex400
```

● Step 2 – Start Ng Serve

```
cd ch13-ex400  
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; @Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styles: [  
    '.selected { color: white; background-color:red; padding: 10px; margin: 10px }', '.unselected { background-color: white; padding: 10px;  
margin: 10px}'  
  ]  
)  
export class AppComponent {  
  selectedAnimal = 'cat';  
  animals = ['cat', 'dog', 'zebra', 'giraffe'];  
  
  onAnimalClicked(event:Event){  
    const clickedAnimal = event.srcElement.innerHTML.trim(); this.selectedAnimal = clickedAnimal; }  
}
```

● Step 5 – Edit Template

Edit ‘app.component.html’ and change it to the following:

```
<div *ngFor="let animal of animals"> <div [ngClass]="{{'selected': animal === selectedAnimal, 'unselected' : animal !== selectedAnimal}}"
```

```
(click)="onAnimalClicked($event)">{ {animal} }</div> </div>
```

• Exercise Complete

Your app should be working at localhost:4200.

13.12 NgStyle

This is a directive for setting the css styles of an element. If you only want to set one style, it's probably easier to use something like the code below:

```
<div [style.fontSize]="selected ? 'x-large' : 'smaller'"> Some text.  
</div>
```

However, if you want to set multiple styles, this directive is the way to go. This directive expects an expression that evaluates to an object containing style properties. This expression can be inline code like this: `[ngStyle]="{{'color': 'blue', 'font-size': '24px', 'font-weight': 'bold'}}"`

Or a function call like this:

```
[ngStyle]="setStyles(animal)"  
  
... later on in the class ...  
  
setStyles(animal:String){  
  let styles = {  
    'width' : '50px'  
  }  
  return styles;  
}
```

13.13 NgStyle - Example

It lets the user click on an animal in an animal list to select it. The selected animal is highlighted in red. This will be example ‘ch13-ex500’.



● Step 1 – Build the App using the CLI

```
ng new ch13-ex500
```

● Step 2 – Start Ng Serve

```
cd ch13-ex500  
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; @Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html'  
})  
export class AppComponent {  
  selectedAnimal = 'cat';  
  animals = ['cat', 'dog', 'zebra', 'giraffe'];  
  
  onAnimalClicked(event:Event){  
    const clickedAnimal = event.srcElement.innerHTML.trim(); this.selectedAnimal = clickedAnimal; }  
  
  getAnimalStyle(animal){  
    const isSelected = (animal === this.selectedAnimal); return {  
      'padding' : '10px',  
      'margin' : '10px',  
      'color' : isSelected ? '#ffffff' : '#000000', 'background-color' : isSelected ? '#ff0000' : '#ffffff', }  
  }  
}
```

● Step 5 – Edit Template

Edit ‘app.component.html’ and change it to the following:

```
<div *ngFor="let animal of animals"> <div [ngStyle]="getAnimalStyle(animal)" (click)="onAnimalClicked($event)">{{animal}}</div> </div>
```

- **Exercise Complete**

Your app should be working at localhost:4200.

13.14 Other Directives

Angular also uses other directives for form handling. We will cover those in a later chapter.

13.15 Creating Directives

• Directives and Components Are Similar

They are both Angular objects that correspond to elements in the markup and can modify the resulting user interface. They both have selectors. The selector is used to identify the Component or Directive that is associated with markup in the web page or a template. For Components, you usually use the tag name, for example ‘CustomerList’. For Directives, you usually use a tag attribute name, which utilizes square brackets, for example ‘[tooltip]’.

They both have annotations. Directives have the `@Directive` annotation, Components have the `@Component` annotation.

They both have classes and the classes can use Dependency Injection in the same manner through the Constructor.

• Directives and Components Aren’t Completely the Same

A component requires a view whereas a directive does not.

Directives do not have a template. There is no bundled HTML markup used to render the element.

Directives add behavior to an existing DOM element. For example, you could add a directive for a tooltip. You create the directive, you add the directive selector to the html or the templates that use it and it delivers the functionality (you will need to add imports as well).

• Steps to Creating a Directive

Creating a Directive is similar to creating a Component.

1. Import the Directive decorator.
2. Add the `@Directive` annotation, including a CSS attribute selector (in square brackets) that identifies our directive as an attribute. You can also add other elements to the `@Directive` annotation, including input properties and host mappings.
3. Specify the name of the public input property for binding (if required).
4. Write the Directive class. This class will use Constructor Injection and will probably manipulate the injected element and renderer.
5. Apply the decorator to the Components or Directives which are going to use it.

13.16 Accessing the DOM in Directives

As we mentioned earlier, Directives are markers on a DOM element (such as an attribute) that tell Angular to attach a specified behavior to an existing element. So, we obviously need a way to access the DOM element that the Directive is being applied to, as well as ways to modify the DOM element.

Angular provides us with two very useful objects: the ElementRef and the Renderer.

1. The ElementRef object gives you direct access to the DOM element for the directive through the ‘nativeElement’ property. Be cautious with the use of the ElementRef object. Permitting direct access to the DOM can make your application more vulnerable to XSS attacks.
2. The Renderer object gives us many helper methods to enable us to modify the DOM element.

We can inject both into our class. The code below accepts the element reference (which lets you access the DOM element using its ‘nativeElement’ property) and the renderer through the constructor and makes each one a private instance variable:

```
constructor(private element: ElementRef, private renderer: Renderer) {  
}
```

13.17 Creating Simple Directive - Example

This is a simple directive that changes the size of the html element to which it is added. This will be example ‘ch13-ex600’.

• Step 1 – Build the App using the CLI

```
ng new ch13-ex600
```

• Step 2 – Navigate to Directory

```
cd ch13-ex600
```

• Step 3 – Create Directive using CLI

Use the cli to create the files and also modify the module ‘app.module.ts’.

```
ng generate directive sizer
```

This will generate some files including ‘sizer.directive.ts’. Now edit ‘sizer.directive.ts’ and change it to the following:

```
import { Directive, Input, Component, ElementRef, Renderer, OnInit } from '@angular/core';
@Directive({
  selector: '[sizer]'
})
export class SizerDirective implements OnInit {
  @Input() sizer: string;

  constructor(private element: ElementRef, private renderer: Renderer) {}

  ngOnInit() {
    this.renderer.setStyle(this.element.nativeElement, 'font-size', this.sizer);
  }
}
```

Note how the directive does its work in the ‘ngOnInit’ method that is fired after the directive has initialized.

If you were to move the ‘setStyle’ code to the constructor then this would work because the ‘sizer’ input variable does not have its value immediately set, it is set when the app component initializes.

• Step 5 – Edit Template

Edit ‘app.component.html’ and change it to the following:

```
<div sizer="72px">
  {{title}}
</div>
```

• Step 6 – View Application

Open web browser and navigate to localhost:4200. It should display ‘app works’ in large text.

- **Exercise Complete**

Your app should be working at localhost:4200. Note how you can use the renderer to update the style to change the size.

13.18 Accessing the DOM Events in Directives

We may also need a way to access the DOM events for the element linked to the directive. Angular provides us with different ways to access these events.

• Using the Directive Element ‘Host’

This can be used to specify the events, actions, properties and attributes related to the host element. It can be used to bind events to code in the class.

```
@Directive({
  selector: 'input',
  host: {
    '(change)': 'onChange($event)',
    '(window:resize)': 'onResize($event)'
  }
})
class InputDirective {
  onChange(event:Event) {
    // invoked when the input element fires the 'change' event
  }
  onResize(event:Event) {
    // invoked when the window fires the 'resize' event
  }
}
```

• HostListeners

Angular host listeners are annotations that allow you to bind a method in your class to a DOM event.

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}

private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```

13.19 Accessing the DOM Properties in Directives

You may want to modify the properties for the element linked to the directive. You can obviously do this using the element ref. However, there is another way. You can use the @HostBinding directive to bind a DOM property of the element to an instance variable in your Angular directive. Then you can update the value of the variable and the DOM property will automatically be updated to match.

For example in the code below you could control the background color of the element by modifying the value of the 'backgroundColor' instance variable.

```
@Directive({
  selector: '[myHighlight]',
})
class MyDirective {
@HostBinding('style.background-color') backgroundColor:string = 'yellow'; }
```

13.20 Creating Directive with Events – Example

This is an example directive that works with host events. Host events map to DOM events in the host element. They are useful when you need a directive that responds to things happening on the DOM. This will be example ‘ch13-ex700’.

- **Step 1 – Build the App using the CLI**

```
ng new ch13-ex700
```

- **Step 2 – Navigate to Directory**

```
cd ch13-ex700
```

- **Step 3 – Create Directive using CLI**

Use the cli to create the files and also modify the module ‘app.module.ts’.

```
ng generate directive hoverer
```

This will generate some files including ‘hoverer.directive.ts’. Now edit ‘hoverer.directive.ts’ and change it to the following:

```
import { Directive, Input, ElementRef, Renderer } from '@angular/core';
@Directive({
  selector: '[hoverer]',
  host: {
    '(mouseenter)': 'onMouseEnter()',
    '(mouseleave)': 'onMouseLeave()'
  }
})

export class HovererDirective {
  @Input() hoverer;

  constructor(
    private elementRef: ElementRef,
    private renderer: Renderer) { }

  onMouseEnter(){
    this.renderer.setStyle(
      this.elementRef.nativeElement, 'color', this.hoverer);
  }

  onMouseLeave(){
    this.renderer.setStyle(
      this.elementRef.nativeElement, 'color', 'black');
  }
}
```

- **Step 4 – Edit Template**

Edit ‘app.component.html’ and change it to the following:

```
<h1 hoverer="red">{{title}}</h1>
```

● Step 5 – View Application

Open web browser and navigate to localhost:4200. It should turn red when you hover over ‘app works!’.

● Exercise Complete

Your app should be working at localhost:4200.

14 More Components

14.1 Introduction

The purpose of this chapter is to enhance your knowledge of Components further with more advanced topics.

14.2 Components and Child Components

So, you know a Component is a building block in a user interface.

An Angular application always has an Application (or root) Component. This Component (like other components) has a tag in the html and Angular bootstraps into that component. This Application Component (like other components) can contain other (Child) Components.

So, Components can contain other Components. This is known as Composition.

14.3 Composition

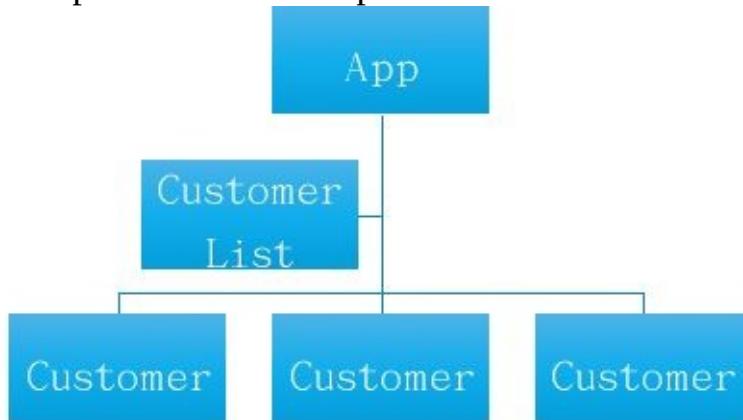
As we mentioned earlier, components are like Lego bricks for the UI. Composition is the art of composing an application using these Lego bricks together.

We are going to introduce Composition with an example below.

14.4 Composition & Data Location

● Introduction

When you write a single page application, the convention is that you have a hierarchy of components, a Composition. For example:



When you are coding with a hierarchy of compositions, you must take great care in **storing the data (known as state) in the correct place so it is never repeated** (i.e. stored twice). Pete Hunt at Facebook wrote a superb article here: <https://facebook.github.io/react/docs/thinking-in-react.html>. The article is about React but the same rules apply to Angular.

14.5 Data Flowing Downwards

● Introduction

Data should flow downwards from higher-level components to lower-level components.

When you create a component that receives data from outside, you must explicitly tell Angular to expect that data as input, using the `@Input` decorator. You place the `@Input` decorator next to the instance variable to which the data will be injected from outside.

When you pass data into a component from the outside, you pass that data into the component using input properties.

● Aliases

Sometimes you may want the name of the input property to be different from the name of the instance variable to which it will be injected. That is when you need to use an ‘alias’, which allows you to specify the input property name. The alias may be specified inside parentheses in the `@Input` decorator.

● Example Code

```
bmw : m3
porsche : 911
bmw : m3
```

This component will pass data from the application to car components. This will be example ‘ch14-ex100’.

Step 1 – Build the App using the CLI

```
ng new ch14-ex100 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex100
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit App Class

Edit ‘app.component.ts’ and change it to the following:

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

@Component({
  selector: 'app-root',
  template: `
<car *ngFor="let car of cars" [theCar]="car"></car> `,
  styles: []
})
```

```
export class AppComponent {
  cars:Array<ICar> = [
    {make: 'bmw', model: 'm3'},
    {make: 'porsche', model: '911'},
    {make: 'bmw', model: 'm3'}
];
}
```

Step 5 – Create ICar Interface

```
ng generate interface ICar
```

Step 6 – Edit ICar Interface

Edit ‘icar.ts’ and change it to the following:

```
Edit 'icar.ts' and change it to the following: export interface ICar {
  make: string,
  model: string
}
```

Step 7 - Create Car Class

```
ng generate component Car --inline-template --inline-style --flat
```

Step 8 – Edit Car Class

Edit ‘car.component.ts’ and change it to the following:

```
import { Component, Input } from '@angular/core'; import { ICar } from './icar';

@Component({
  selector: 'car',
  template: `
    <p>
      {{car.make}} : {{car.model}}
    </p>
  `,
  styles: []
})
export class CarComponent {
  @Input('theCar') car: ICar;
}
```

Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The Application Component has a list of three Cars. We use the ‘NgFor’ directive to iterate over the list of cars, generating a Car Component for each one. We use the ‘theCar’ input property to pass the car to the Car component
2. We have a Car Component to display each car. In the Car component we use the ‘theCar’ aliased input property to accept the car instance variable from the outside.

● Warning!!

You can pass objects that contain fields through the `@Input()` properties and through the ‘inputs’ element of `@Component` Annotation. For example, you could perform an HTTP request to get a customer object which contains a name & address then pass it into a Child Component through an attribute to display it. This works well but you need to bear in mind that the attribute you are passing in may be null until the server returns the response. So, the Child Component may attempt to display elements like the name and address of a null object, which can cause Angular to throw Exceptions and NOT display the data when it has come back from the server. This has caught me out several times.

The solution to this is to use the Elvis Operator.

14.6 Events Flowing Upwards

● Introduction

Sometimes you need to compose parent Components that contain child Components and control them. Parent Components need to have code that responds to things happening (events) on child Components. Events should flow upwards, emitted upwards from lower-level components and responded to by higher-level components.

● Setting Up Child Component to Pass Custom Events Up to Parent Components

1. Import the EventEmitter class.
2. Specify the custom events that your component will emit by using the ‘events’ element of the ‘@Component’ directive. You must remember to do this!!
3. Create an event emitter in your class as an instance variable.
4. Call the event emitter method ‘emit’ when you want to emit an event.

● Setting Up Parent Component to Receive Custom Events from Child Components

1. Add the Component with the Custom Events to your other Component. Remember to import it and specify it in the ‘directive’ element of the ‘@Component’ annotation.
2. Add the Component with the Custom Event to the markup in the template of the other Component. Edit the markup in the template to respond to the Custom Event using the event name in round brackets and the template statement it will fire, for example: `(wordInput)="wordInputEvent($event)"`. Notice that this uses the same syntax as non-custom events.

14.7 Emitting Output through @Output()

You create an '@Output()' instance variable of type EventEmitter in the Child Component/Directive. You modify the Child Component/Directive to use this instance variable to emit events when required. You modify the Parent Component to bind an event attribute of the same name in its Template with a Template Statement. Angular will emit the event from the Child Component/Directive to the Parent and will invoke the Template Statement.

● Example Code



This code shows how events can flow upward from one component to another. This will be example 'ch14-ex200'.

Step 1 – Build the App using the CLI

```
ng new ch14-ex200 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex200  
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see 'app works!'.

Step 4 – Edit App Class

Edit 'app.component.ts' and change it to the following:

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

@Component({
  selector: 'app-root',
  template: `
    <car *ngFor="let car of cars" (carDelete)="deleteCar(car)" [theCar]="car"> </car>
  `,
  styles: []
})
```

```

export class AppComponent {
  cars:Array<ICar> = [
    {make: 'bmw', model: 'm3'},
    {make: 'porsche', model: '911'},
    {make: 'ford', model: 'mustang'}
  ];

  deleteCar(car: ICar){
    alert('Deleting car:' + JSON.stringify(car));
  }
}

```

Step 5 – Create ICar Interface

ng generate interface ICar

Step 6 – Edit ICar Interface

Edit ‘icar.ts’ and change it to the following:

```

Edit 'icar.ts' and change it to the following: export interface ICar {
  make: string,
  model: string
}

```

Step 7 - Create Car Class

ng generate component Car --inline-template --inline-style --flat

Step 8 – Edit Car Class

Edit ‘car.component.ts’ and change it to the following:

```

import { Component, Input, Output, EventEmitter } from '@angular/core'; import { ICar } from './icar';

@Component({
  selector: 'car',
  template: `
    <p>
      {{car.make}} : {{car.model}}
    <button (click)="delete(car)">Delete</button> </p>
  `,
  styles: []
})
export class CarComponent {
  @Input('theCar') car: ICar;
  @Output() carDelete = new EventEmitter();

  delete(car: ICar){
    this.carDelete.emit(car);
  }
}

```

Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The Application Component has a list of three Cars. It listens for the ‘carDelete’ event,

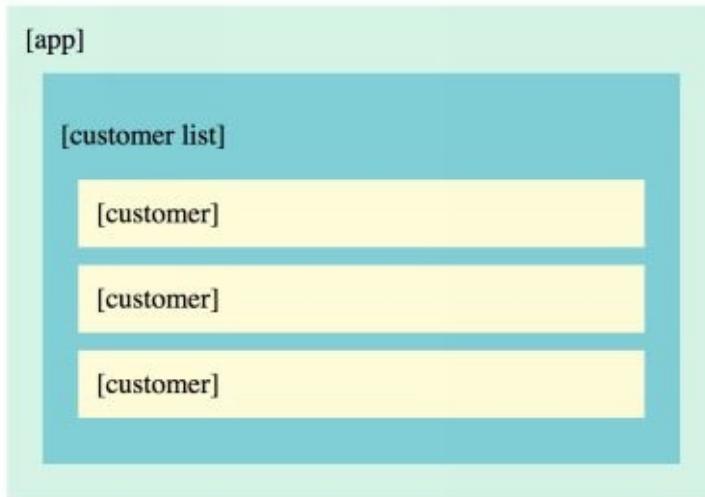
firing the ‘deleteCar’ method when it occurs.

2. We have a Car Component to display each car. It contains a delete button and it emits a ‘carDelete’ event when the user clicks on it.

14.8 Composition Example

● Create Components

Let's use the Angular CLI to create a crude example of a component that contains other components, in the structure below: We are going to write an app that contains a customer list which contains three customers. This is example 'ch14-ex300'.



Step 1 – Build the App using the CLI

```
ng new ch14-ex300 --inline-template --inline-style
```

Notice the '--inline-template' and '--inline-style' arguments. These tell the CLI to combine the template and style into the component's class, making the component definition 1 file instead of 3. Much easier when you have small templates with few styles. When you get to code much larger components you may want to rethink this.

When you use this command (and the commands below), you can add the ---spec argument to tell the CLI not to create a '.spec.ts' files for the app and the components. I just left the spec file generation alone.

Step 2 – Start Ng Serve

```
cd ch14-ex300
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see 'app works!'.

Step 4 – Create Customer List Component

```
ng generate component customer-list --flat --inline-template --inline-style
```

Note again how we are using the '--inline-template' and 'inline-style' arguments again to combine the component into one file. Note that we are using the argument '--spec false' to tell the CLI not to generate a '.spec.ts' testing file.

Step 5 – Create Customer Component

```
ng generate component customer --flat --inline-template --inline-style
```

Note again how we are using the '--

‘inline-template’ and ‘inline-style’ arguments again to combine the component into one file. Note again that we are using the argument ‘--spec false’ to tell the CLI not to generate a ‘.spec.ts’ testing file.

Step 6 – Edit the App Component

Copy and paste the code below into ‘app.component.ts’.

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `
<div class='app'>
[app]
<app-customer-list>
</app-customer-list>
</div>
`,
styles: ['.app {background-color:#d5f4e6;margin:10px;padding:10px;}']
})

export class AppComponent { }
```

Step 6 – Edit the Customer List Component

Copy and paste the code below into ‘customer-list.component.ts’.

```
import { Component, OnInit } from '@angular/core';
@Component({
selector: 'app-customer-list',
template: `
<div class='customerList'>
<p>
[customer list]
</p>
<app-customer>
</app-customer>
<app-customer>
</app-customer>
<app-customer>
</app-customer>
<app-customer>
</app-customer>
</div>
`,
styles: ['.customerList {background-color:#80ced6;margin:10px;padding:10px;}']
})
export class CustomerListComponent implements OnInit {

constructor() { }

ngOnInit() {
}

}
```

Step 7 – Edit the Customer Component

Copy and paste the code below into ‘customer.component.ts’.

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

```

@Component({
  selector: 'app-customer',
  template: `
    <div class='customer'>
      [customer]
    </div>
  `,
  styles: ['.customer {background-color:#fefbd8;margin:10px;padding:10px}']
})
export class CustomerComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}

```

Exercise Complete

Your app should be working at localhost:4200. You have Composed an app consisting of different components. Note the following: Each Component has a '@Component' directive at the top that specifies the selector. For example, the Customer Component:

```

@Component({
  selector: 'app-customer',

```

When you need to include that component in another component, you use the selector as a tag. For example, the Customer List Component uses the Customer Component's tag to include it in the template three times.

```

<app-customer>
</app-customer>
<app-customer>
</app-customer>
<app-customer>
</app-customer>

```

The file 'app.module.ts' was modified by the CLI. Each Component was added as a declaration in the module. More on modules later!

● Data Flowing Downwards

Let's modify the example 'ch14-ex300' to pass data down from the Customer List Component to the Customer Components. This will be example 'ch14-ex400'.

[app]

[customer list]

Brian | Atlanta

Peter | San Francisco

Janet | Colorado

Edit the Customer Component

We edit the Customer Component to accept input data from the outside.

Modify the imports to include the Input class from the angular core.

```
import { Component, OnInit, Input } from '@angular/core'; Change the template to include string  
interpolation of the instance variable 'customer'. {{customer.name}} and  
{{customer.city}}. This will output the contents of the 'name' and 'city' properties of the  
'customer' instance variable.
```

```
template: `  
  <div class='customer'>  
    {{customer.name}} | {{customer.city}}  
  </div>  
`;
```

Declare instance variable 'customer' as an input variable.

```
@Input() customer;
```

Edit the Customer List Component

We edit the Customer List Component to pass data to the Customer Component using one-way data binding (more on this later).

Replace the tags below:

```
<app-customer>  
</app-customer>  
<app-customer>  
</app-customer>  
<app-customer>  
</app-customer>
```

with the following:

```
<app-customer *ngFor="let customer of customerList" [customer]="customer"> </app-customer>
```

Declare the instance variable 'customerList' and populate it with data. Add the code after 'export' and before 'constructor'.

```
private customerList = [
```

```
{ name: 'Brian', city: 'Atlanta' },
{ name: 'Peter', city: 'San Francisco' }, { name: 'Janet', city: 'Colorado' },
];
```

Exercise Complete

Your app should be working at localhost:4200. You modified the Component List to contain customer list data and you passed this data down to the Customer using one-way (downwards) data binding. Note the following: The Customer List Component sets up the customer list data as an instance variable and the template refers to this variable.

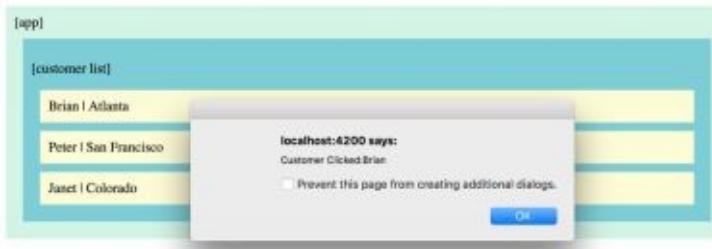
The Customer List Component uses an ‘ngFor’ in the template. This allows the template to iterate over the customer list, creating a ‘customer’ variable for each customer and passing it down to the Customer Component via a bound attribute.

The Customer Component declares an instance variable called ‘customer’. It uses an annotation ‘@Input()’ to tell Angular to have its value automatically set from the outside. Note that the ‘Input’ class has to be imported at the top of the Customer Component’s class.

The Customer Component uses ‘{{customer.name}}’ and ‘{{customer.city}}’ in the template to output the ‘name’ and ‘city’ properties of the instance variable called ‘customer’.

Events Flowing Upwards

Let’s modify the example above to fire events from the Customer Component to the Customer List Component. This will be example ‘ex300’ and it is based off example ‘ch14-ex500’.



Edit the Customer Component

We edit the Customer Component to output a ‘clicked’ event when the user clicks on a customer.

Modify the imports to include the output and event emitter classes from the angular core.

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core'; Modify the template to call the method 'onClicked' when the user clicks on a Customer:
```

```
template: `<div class='customer'(click)="onClicked()"> {{customer.name}} | {{customer.city}}</div>`;
```

Add an instance variable for the event emitter to the typescript class: @Output() clicked: EventEmitter<String> = new EventEmitter<String>(); Add the ‘onClicked’ method to the typescript class:

```
onClicked(){this.clicked.emit(this.customer.name);}
```

Edit the Customer List Component

We edit the Customer List Component to respond to the ‘clicked’ event emitted by the Customer Component.

Modify the template to bind to the ‘clicked’ event, calling the ‘onCustomerClicked’ method when the event occurs: <app-customer *ngFor="let customer of customerList" [customer]="customer" (clicked)="onCustomerClicked(\$event)"> Add the method ‘onCustomerClicked’ to receive the event data and display an alert box with the customer name.

```
onCustomerClicked(customerName:String){  
  alert('Customer Clicked:' + customerName);  
}
```

Exercise Complete

Your app should be working at localhost:4200. The app should now display an alert box when you click on a Customer. Note how the events are flowing up from multiple Customer Components to a single Customer List Component.

Note the following:

The Customer Component sets up an instance variable for an event emitter that outputs an event with string data. It uses an annotation ‘@Output()’ to tell Angular that other Components should be able to bind to this event.

The Customer Component template includes the Angular directive (click) to listen and respond to the user clicking on the ‘div’. It fires a method which uses the event emitter to output the event.

The Customer List Component includes an event handler to listen and respond to the custom ‘clicked’ event in the Customer Component.

The Customer List Component contains code in the ‘onCustomerClicked’ method to receive the data from the event and display an alert box.

14.9 Template Reference Variables

● Introduction

A Template Reference Variable is a reference to one or more elements within a template. You can use the 'ref-' prefix instead of '#'.

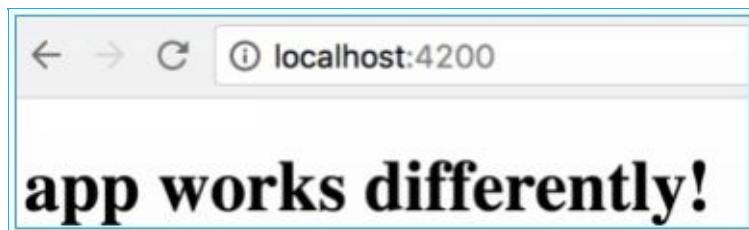
Once you have declared a template reference variable, you can use it in either the template or in the code. However, you need to know that this variable is not set by Angular until the ngAfterViewInit lifecycle method has completed. We will be covering the Angular lifecycles in a later chapter.

● ViewChild

Declares a reference to a child element in the component. When you declare your instance variable you specify a selector in parentheses, which is used to bind the child element to the instance variable.

● ViewChild – Example

This example shows text and it looks similar to the CLI default application. This is example ‘ch14-ex600’.



However, if you examine the code you will see that it uses a template variable to refer to the ‘h1’ element: <h1 #title></h1> and has code to set its inner html after the Component has finished loading the view:

```
ngAfterViewInit(){
  this.title.nativeElement.innerHTML = 'app works differently!'
}
```

Also note that the template variable is referred to by some interpolation in the template: The title is {{title.innerHTML}}

Step 1 – Build the App using the CLI

ng new ch14-ex600 --inline-template Remember that the ‘--inline-template’ tells the CLI to use inline templates when generating the new application.

Step 2 – Start Ng Serve

```
cd ch14-ex600
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, ElementRef, ViewChild, AfterViewInit } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1 #title></h1>
    The title is {{title.innerHTML}}
  `,
  styleUrls: ['./app.component.css']
})
export class AppComponent implements AfterViewInit {
  @ViewChild('title') title: ElementRef;

  ngAfterViewInit() {
    this.title.nativeElement.innerHTML = 'app works differently!'
  }
}
```

Step 5 – View App

You should see ‘app works differently!’.

• ViewChildren

Declares a reference to multiple child element in the component. When you declare your instance variable you specify a selector in parentheses, which is used to bind the child elements to the instance variable.

This selector can be the Child Type (i.e. the class of the Child angular element) or the Template Reference(s) (i.e. #name).

- **View Children – Example**

In the example below use `ViewChildren` to access a list of paragraphs. We use `ViewChildren` with a list of child reference names, separated by commas. This is example ‘ch14-ex700’.

*Et atque amplexum datur etiam deinde
ad hunc modum: quodcumque sibi
admodum placuerit, et quodcumque
admodum possit, et quodcumque
admodum videat.*

At vero eos et accusam et justo duo dolores et ea rebum.
Stet clita kasd gubergren, no sea takimata sanctus est
Lorem ipsum dolor sit amet. Lorem ipsum dolor sit
amet, consetetur sadipscing elitr, sed diam nonumy
eirmod tempor invidunt ut labore et dolore magna
aliquyam erat, sed diam voluptua.

Number of Paragraphs:2

Step 1 – Build the App using the CLI

Step 2 – Start Ng Serve

```
cd ch14-ex700
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, ViewChildren, AfterViewInit } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <p #paragraph1>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. </p> <p #paragraph2>At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.</p> <p *ngIf="note">{{note}}</p>
  `,
  styles: ['p { background-color: #FFE5CC; padding: 15px; text-align: center }']
})
export class AppComponent implements AfterViewInit{
  @ViewChildren('paragraph1, paragraph2') paragraphs;
  note: string = "";

  ngAfterViewInit(){
    setTimeout(_ => this.note = 'Number of Paragraphs: ' + this.paragraphs.length); }
}
```

Exercise Complete

Your app should be working at localhost:4200.

You should two paragraphs of text, with a paragraph count below.

● NgContent and Transclusion

Transclusion is the inclusion and transference of content from the area inside the Component’s tags into the Component’s template. The NgContent tag is used for transclusion. It even has a selector that allows you to select which content to include. If you use a '[' in the selector (e.g. '[test'] then this can be used to select content with this attribute (e.g. ‘<div test>hejwejgwegrhj</div>’).

● NgContent and Transclusion – Example

The example below is very simple and does not use selectors. It simply includes the text between the Component tags. This is example ‘ch14-ex800’.

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Praesent eget ornare neque, vel consectetur eros.

Step 1 – Build the App using the CLI

```
ng new ch14-ex800 --inline-template
```

Step 2 – Start Ng Serve

```
cd ch14-ex800  
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Classes

This time we are going to add two Component classes into the same file.

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core';
@Component({
selector: 'Paragraph',
template: `

<p><ng-content></ng-content></p> `,
styles: ['p { border: 1px solid #c0c0c0; padding: 10px }']
})
export class Paragraph { }

@Component({
selector: 'app-root',
template: `

<p>
<Paragraph>Lorem ipsum dolor sit amet, consectetur adipiscing elit. </Paragraph> <Paragraph>Praesent eget ornare neque, vel consectetur eros. </Paragraph> </p>
`,
styleUrls: ['./app.component.css']
})
export class AppComponent {
title = 'app works!';
}
```

Step 5 – Edit Module

We have two Component classes in the same file. However, we need to ensure that both Components are declared in the module definition. Otherwise the Components won’t be useable.

Edit ‘app.module.ts’ and change it to the following:

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

```
import { NgModule } from '@angular/http';
import { AppComponent, Paragraph } from './app.component';
@NgModule({
declarations: [
  AppComponent,
  Paragraph
],
imports: [
  BrowserModule,
  FormsModule,
  HttpModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Exercise Complete

You should two paragraphs of text.

● ContentChild

So, you can use ‘ngContent’ to transclude additional content. You can use ContentChild to declares a reference to a child element in the transcluded additional content.

● ContentChild – Example

The example below is like the previous example only it uses ContentChild to get a reference to the ‘title’ element inside the transcluded content. It then includes the inner html from that element. This is example ‘ch14-ex900’.

Paragraph 1

```
    Paragraph 1
    Paragraph 2
    Step 1 – Build the App using the CLI
    Step 2 – Start Ng Serve
    Step 3 – Open App
    Open web browser and navigate to localhost:4200. You should see ‘app works!’.
    Step 4 – Edit Classes
    Edit ‘app.component.ts’ and change it to the following:
```

```
    Paragraph 1
    Paragraph 2
    Step 1 – Build the App using the CLI
    Step 2 – Start Ng Serve
    Step 3 – Open App
    Open web browser and navigate to localhost:4200. You should see ‘app works!’.
    Step 4 – Edit Classes
    Edit ‘app.component.ts’ and change it to the following:
```

```
    Paragraph 1
    Paragraph 2
    Step 1 – Build the App using the CLI
    Step 2 – Start Ng Serve
    Step 3 – Open App
    Open web browser and navigate to localhost:4200. You should see ‘app works!’.
    Step 4 – Edit Classes
    Edit ‘app.component.ts’ and change it to the following:
```

```
ng new ch14-ex900 --inline-template
```

Step 2 – Start Ng Serve

```
cd ch14-ex900
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Classes

Edit ‘app.component.ts’ and change it to the following:

```

import { Component, ContentChild } from '@angular/core';
@Component({
selector: 'Paragraph',
template: `
<div>
<b>{ {title.nativeElement.innerHTML} }</b> <p><ng-content></ng-content></p> </div>
`,
styles: ['p { border: 1px solid #c0c0c0 }']
})
export class Paragraph {
@ContentChild('title') title;
}

@Component({
selector: 'app-root',
template: `
<p>
<Paragraph><title #title>Paragraph 1</title>Lorem ipsum dolor sit amet, consectetur adipiscing elit. In pulvinar egestas massa sit amet scelerisque.</Paragraph> <Paragraph><title #title>Paragraph 2</title>Praesent eget ornare neque, vel consectetur eros. Morbi gravida finibus arcu, vel mattis justo dictum a.</Paragraph> </p>
`,
styleUrls: ['./app.component.css']
})
export class AppComponent {
title = 'app works!';
}

```

Step 5 – Edit Module

We have two Component classes in the same file. However, we need to ensure that both Components are declared in the module definition. Otherwise the Components won't be useable.

Edit ‘app.module.ts’ and change it to the following:

```

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

import { AppComponent, Paragraph } from './app.component';
@NgModule({
declarations: [
  AppComponent,
  Paragraph
],
imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

Exercise Complete

You should see two paragraphs of text.

• ContentChildren

You can use ContentChildren to declares a reference to multiple child elements in the transcluded additional content.

● ContentChildren – Example

The example below is like the previous example only it uses ContentChild to get a reference to the ‘title’ element inside the transcluded content. It then includes the inner html from that element. This is example ‘ch14-ex1000’.

 Lorem ipsum dolor sit amet, consectetur adipiscing
 elit.

- Albertus Falx
- Godefridus Turpilius
- Demipho Renatus

 Number of people: 3

 Praesent eget ornare neque, vel consectetur eros.

- Hanno Grumio
- Lycus Auxilius

 Number of people: 2

Step 1 – Build the App using the CLI

```
ng new ch14-ex1000 --inline-template
```

Step 2 – Start Ng Serve

```
cd ch14-ex1000  
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Classes

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, ContentChildren } from '@angular/core';  
@Component({  
  selector: 'Person',  
  template: `<div>&nbsp;-&nbsp;<ng-content></ng-content></div>`,  
  styles: [""]  
})  
export class Person {  
  
  @Component({  
    selector: 'Paragraph',  
    template: `<div>  
      <ng-content></ng-content>  
      <p *ngIf="people">Number of people: {{people.length}}</p>  
    </div>`  
  })  
  people: Person[] = []  
}
```

```

styles: ['div { border: 1px solid #c0c0c0; margin:10px; padding:10px }', 'p { margin: 5px 0 }']
})
export class Paragraph {
@ContentChildren(Person) people;
}

@Component({
selector: 'app-root',
template: `
<Paragraph>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    <Person>Albertus Falx</Person> <Person>Godefridus Turpilius</Person> <Person>Demipho Renatus</Person> </Paragraph>
<Paragraph>Praesent eget ornare neque, vel consectetur eros.
    <Person>Hanno Grumio</Person> <Person>Lycus Auxilius</Person> </Paragraph>
    `,
styleUrls: ['./app.component.css']
})
export class AppComponent {
title = 'app works!';
}

```

Step 5 – Edit Module

We have three Component classes in the same file. However, we need to ensure that both Components are declared in the module definition. Otherwise the Components won't be useable.

Edit ‘app.module.ts’ and change it to the following:

```

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

import { AppComponent, Paragraph, Person } from './app.component';
@NgModule({
declarations: [
  AppComponent,
  Paragraph,
  Person
],
imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

Exercise Complete

You should two paragraphs of text. Each paragraph should have a list of people and a people count at the bottom.

14.10 Component Class Lifecycle

Like AngularJS, Angular manages the components for you – when it creates them, when it updates them, when it destroys them etc. Each component has what is known as lifecycle events – its birth, life events like changes and death. Sometimes you need to add extra code that is fired for you by Angular when these events occur.

● Constructor vs OnInit

Sometimes you need to setup your component and initialize it. You have two choices here. You can use the constructor or the ‘OnInit’ lifecycle method. The OnInit lifecycle method is fired when the component is first initialized.

It’s up to you which one you should use but many people are of the following opinion: *Mostly we use ngOnInit for all the initialization/declaration and avoid stuff to work in the constructor. The constructor should only be used to initialize class members but shouldn't do actual "work".*

● Example

You may want to add some code to do something once the component has loaded and is visible. For example, place the input focus on the first field so the user can just start typing away. You would think that you could add this code to the component class constructor but that would be incorrect as the constructor is fired before the component is visible. In fact, you would probably add this code to ngAfterViewInit – after the view has been initialized.

● Interfaces

To hook into a lifecycle method, your component’s class should implement the required interface. The interface will then force you to implement the corresponding method.

For example, to implement a method fired after the view has initialized you should implement the interface ‘AfterViewInit’, which requires the method ‘ngAfterViewInit’. For more details, refer to the table below.

Interface	Method	Description
OnChanges	ngOnChanges	Called when an input or output binding value changes
OnInit	ngOnInit	After the first ngOnChanges
DoCheck	ngDoCheck	Developer's custom change detection
AfterContentInit	ngAfterContentInit	After component content initialized
AfterContentChecked	ngAfterContentChecked	After every check of component content

AfterViewInit	ngAfterViewInit	After component's view(s) are initialized
AfterViewChecked	ngAfterViewChecked	After every check of a component's view(s)
OnDestroy	ngOnDestroy	Just before the directive is destroyed.

● NgOnChanges

This callback is invoked when the value of a bound property changes. It executes, every time the value of an input property changes. It will receive a changes map, containing the current and previous values of the binding, wrapped in a SimpleChange.

● NgOnChanges – Example

The screenshot shows a simple web page with a heading 'Change this field:' followed by a text input box containing the word 'hello'. Below the input box is a section titled 'History' which contains a list of JSON objects representing change events:

```
{"nm":{"previousValue":{},"currentValue":""}}
{"nm":{"previousValue":"","currentValue":"h"}}
{"nm":{"previousValue":"h","currentValue":"he"}}
 {"nm":{"previousValue":"he","currentValue":"hel"}}
 {"nm":{"previousValue":"hel","currentValue":"hell"}}
 {"nm":{"previousValue":"hell","currentValue":"hello"}}
```

This is a component with a textbox that lets you enter text. When you make a change, the changes are recorded below. This will be example ‘ch14-ex1100’.

Step 1 – Build the App using the CLI

```
ng new ch14-ex1100 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex1100
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Classes

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';
@Component({
```

```

selector: 'name',
template: `

<p *ngFor="let change of changes">
{{change}}
</p>
`,
styles: []
})
export class NameComponent implements OnChanges{
@Input('name') nm;
changes: Array<string> = [""];

ngOnChanges(changes: SimpleChanges){
    this.changes.push(JSON.stringify(changes));
}
}

@Component({
selector: 'app-root',
template: `

Change this field: <input [(ngModel)]="name" /> <hr/>
History
<name [name]="name"></name>
`,
styles: []
})
export class AppComponent{
name: string = "";
}

```

Step 5 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

```

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

import { AppComponent, NameComponent } from './app.component';
@NgModule({
declarations: [
    AppComponent,
    NameComponent
],
imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

Exercise Complete

Your app should be working at localhost:4200. Notice the following:

1. Both components reside in the same file but they have to be imported and declared separately in the module.

2. The app component uses 2 way binding to the ‘name’ instance variable. It passes the ‘name’ instance variable to the ‘Name’ component.
3. The ‘Name’ component uses the lifecycle method ‘ngOnChanges’ to listen for changes to input properties (in this case ‘name’). When this method is fired it uses JSON.stringify to dump out a string representation of the change to a list of the changes below.

● NgOnInit

This callback is invoked once Angular is done creating the component and has initialized it. is called directly after the constructor and after the ngOnChange is triggered for the first time.

● NgOnInit – Example

Thu Jun 01 2017 21:27:04 GMT-0400 (EDT)

Thu Jun 01 2017 21:27:05 GMT-0400 (EDT)

This is a component that displays logs. A log of when the component initializes and when the lifecycle method ‘ngOnInit’ is invoked. This will be example ‘ch14-ex2000’.

Step 1 – Build the App using the CLI

```
ng new ch14-ex1200 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex1200
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit } from '@angular/core';
@Component({
selector: 'app-root',
template:`
<p *ngFor="let log of logs">
{{log}}
</p>
`,
styles: []
})
export class AppComponent implements OnInit{
logs: Array<string> = [ new Date() + ""];

constructor(){
  for (let i=0;i<1000;i++){
    console.log(i);
  }
}
```

```
ngOnInit(){
  this.logs.push(new Date()+'');
}
}
```

Exercise Complete

Your app should be working at localhost:4200. Notice the following:

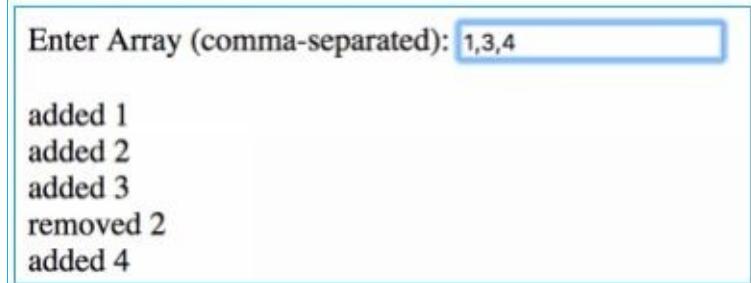
1. The log is initialized when the instance variable is defined.
2. The constructor has some code to slow down the creation of the component.
3. The log is augmented when Angular is done creating the component.

● **NgDoCheck**

This callback is invoked every time the input properties of a component or a directive are checked. We can use this lifecycle hook to extend the check with our own custom check logic.

● **NgDoCheck– Example**

This is a component that will let you create an array and will figure out what you change. This will be example ‘ch14-ex1300’.



Enter Array (comma-separated): 1,3,4

added 1
added 2
added 3
removed 2
added 4

Step 1 – Build the App using the CLI

```
ng new ch14-ex1300 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex1300
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Classes

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, Input, DoCheck, IterableDiffers } from '@angular/core';
@Component({
selector: 'numbers',
template: `
  {{numbers}}
  <br/>
<p *ngFor="let change of changes">
```

```

{{change}}
</p>
``,
styles: ['p{padding:0;margin:0}']
})
export class NumbersComponent implements DoCheck {
@Input('numbers') numbersArray: Array<string>; changes: Array<string> = [];
differ;

constructor(private differers: IterableDifferers) {
  this.differ = differers.find([]).create(null); }

ngDoCheck() {
  const differences = this.differ.diff(this.numbersArray); if (differences) {
    if (differences.forEachAddedItem) {
      differences.forEachAddedItem((item) => {
        if ((item) && (item.item)){
          this.changes.push('added ' + item.item); }
      });
    }
    if (differences.forEachRemovedItem) {
      differences.forEachRemovedItem((item) => {
        if ((item) && (item.item)){
          this.changes.push('removed ' + item.item); }
      });
    }
  }
}

@Component({
  selector: 'app-root',
  template: `
    Enter Array (comma-separated): <input [(ngModel)]="numbers" (onModelChange)="onModelChange()" /> <br/>
    <numbers [numbers]="numbers.split(',')"></numbers> `,
  styles: []
})
export class AppComponent {
  numbers = "";
}

```

Step 5 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

```

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

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

```

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

Exercise Complete

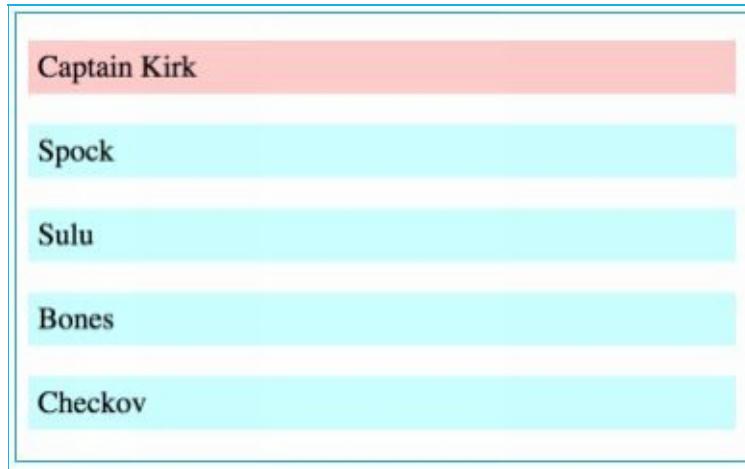
Your app should be working at localhost:4200. Note the following:

1. The App component parses the input string into an array and passes it to the Numbers component.
2. The Numbers component has an Iterable Differ injected via the constructor so that it can be an instance variable and used later.
3. When the input changes and the input property to the Numbers component change, that component uses the differ to analyze the changes and add each change to the change log.

● NgAfterContentInit

This callback is invoked after ngOnInit: when the component or directive's content has been initialized and the bindings have been checked for the first time.

● NgAfterContentInit– Example



In the example below the app component declares a crew structure with members inside its content. Later on, this lifecycle callback is used to select the first crew member on the list. This will be example 'ch14-ex1400'.

Step 1 – Build the App using the CLI

```
ng new ch14-ex1400 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex1400  
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see 'app works!'.

Step 4 – Edit Classes

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, Input, AfterContentInit, ContentChildren, QueryList } from '@angular/core';
@Component({
selector: 'member',
template: `

<p [style.backgroundColor]="getBackgroundColor()"><ng-content></ng-content></p> `,
styles: ['p{padding: 5px}']
})
export class MemberComponent {
selected = false;
getBackgroundColor(){
    return this.selected ? "#FFCCCC" : "#CCFFFF";
}

@Component({
selector: 'crew',
template: `

<p><ng-content></ng-content></p> `,
styles: []
})
export class CrewComponent implements AfterContentInit {
@ContentChildren(MemberComponent) members: QueryList<MemberComponent>;
ngAfterContentInit() {
    this.members.first.selected = true;
}
}

@Component({
selector: 'app-root',
template: `
<crew>
    <member>Captain Kirk</member> <member>Spock</member>
    <member>Sulu</member>
    <member>Bones</member>
    <member>Checkov</member> </crew>
`,
styles: []
})
export class AppComponent {}
```

Step 5 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

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

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

Exercise Complete

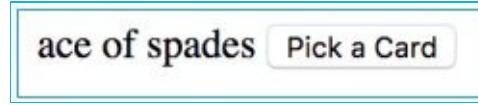
Your app should be working at localhost:4200. Note the following:

1. The App component declares a crew structure with members **inside its content**.
2. The Crew component declares the instance variable ‘members’ (which is declared using `@ContentChildren`) to map into the list of crew members inside its own ‘crew’ tag in the content. Note that this variable is of the type ‘QueryList’ so that it can be queried easier.
3. The Crew content uses the lifecycle method ‘ngAfterContentInit’ to access the instance variable of members once it has been set for you by Angular. This lifecycle method then sets the ‘selected’ instance variable of the first member so that he or she is highlighted.
4. The Member component shows the content inside the ‘member’ tag and sets the component’s background color according to the ‘selected’ instance variable.

● **NgAfterContentChecked**

This callback is performed after every check of the component or directive’s content, effectively when all the bindings of the components have been checked; even if they haven’t changed.

● **NgAfterContentChecked – Example**



ace of spades Pick a Card

This example allows you to pick a card. This will be example ‘ch14-ex1500’.

Step 1 – Build the App using the CLI

```
ng new ch14-ex1500 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex1500  
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Classes

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, ContentChild, AfterContentChecked } from '@angular/core';  
@Component({  
  selector: 'card',  
  template: `
```

```

<ng-content></ng-content>
`;
styles: []
})
export class CardComponent {
}

@Component({
selector: 'app-root',
template: `
<card>{{card}}</card>
<button (click)="pickCard($event)">Pick a Card</button> `,
styles: []
})
export class AppComponent implements AfterContentChecked {
card = CARD_ACE_OF_SPADES;

@ContentChild(CardComponent) contentChild: CardComponent;
ngAfterContentChecked() {
  console.log("content inside card has been checked: " + this.card);
}

pickCard() {
  this.card = this.card === CARD_ACE_OF_SPADES ? CARD_TEN_OF_CLUBS : CARD_ACE_OF_SPADES;
}

const CARD_ACE_OF_SPADES = 'ace of spades';
const CARD_TEN_OF_CLUBS = 'ten of clubs';

```

Step 5 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

```

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

import { AppComponent, CardComponent } from './app.component';
@NgModule({
declarations: [
  AppComponent,
  CardComponent
],
imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The App component uses a Card component to display the current card. The current card is an instance variable and the value of that instance variable is placed inside the Card

Component as inner content.

2. The App component has a button to allow you to flip to another card. It changes the value of the instance variable.
3. The App component has a 'ngAfterContentChecked' method that is fired automatically when the content inside the Card component changes. This is fired when the current card changes.

● NgAfterViewInit

This callback is invoked after a component's view, and its children's views, are created and have been initialized. It is useful for performing component initializations. Note that the @ViewChild and @ViewChildren instance variables are set and available at this point (unlike earlier in the component lifecycle).

● NgAfterViewInit – Example

First Input Field:

This example shows you how to set initial input focus. This will be example ‘ch14-ex1600’.

Step 1 – Build the App using the CLI

```
ng new ch14-ex1600 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex1600  
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, AfterViewInit, ViewChild } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  template: `  
    First Input Field: <input #firstInput /> ,  
    styles: []  
  `})  
export class AppComponent implements AfterViewInit{  
  @ViewChild('firstInput') firstInput;  
  
  ngAfterViewInit(){  
    // ViewChild variables are available in this method.  
    // Set initial focus.  
    this.firstInput.nativeElement.focus();  
  }  
}
```

Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The App component sets up the ‘firstInput’ instance variable to reference the template variable declared as ‘#firstInput’.
2. The App component has ‘ngAfterViewInit’ method that is fired once the view has been initialized and the ‘firstInput’ instance variable is available. This method sets the initial input focus.

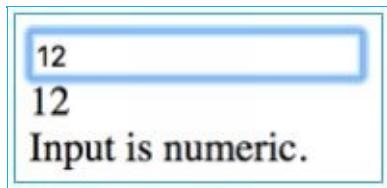
● NgAfterViewChecked

This callback is invoked after every check of the component’s view. Applies to components only. When all the bindings of the children directives have been checked; even if they haven’t changed. It can be useful if the component is waiting for something coming from its child components.

Do not set any variables bound to the template here. If you do, you will receive the ‘Expression has changed after it was checked’ error.

● NgAfterViewChecked – Example

This example allows you to input something and displays a message saying if your input is numeric or not. This will be example ‘ch14-ex1700’.



Step 1 – Build the App using the CLI

```
ng new ch14-ex1700 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex1700  
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, ViewChild, AfterViewChecked } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  template: `  
    <input [(ngModel)]="input"/>  
    <br/>  
    {{input}}  
    <br/>  
    <div #message></div>  
  `,  
  styles: []  
})
```

```

})
export class AppComponent implements AfterViewChecked {
  input: string = "";

  @ViewChild('message') message;

  ngAfterViewChecked(){
    console.log('AfterViewChecked');
    if (isNaN(parseInt(this.input))){
      this.message.nativeElement.innerHTML = "Input not numeric."; }else{
      this.message.nativeElement.innerHTML = "Input is numeric." }
  }
}

```

Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The App component sets up the ‘message’ instance variable to reference the template variable declared as ‘#message’.
2. The App component has ‘ngAfterViewChecked’ method that is fired once the view’s bindings have been checked. This is the method in which we check the input and set the ‘message’ to indicate if the user’s input is numeric or not.

● NgOnDestroy

This callback is invoked when a component, directive, pipe or service is destroyed. Add code here to destroy any references that may remain as instance variables (i.e. clean up your references).

● NgOnDestroy – Example

This example counts up using an interval timer, which it destroys when the component is destroyed. It is the perfect place to get the component ready to be disposed of: for example, to cancel background tasks. This will be example ‘ch14-ex1800’.



Step 1 – Build the App using the CLI

```
ng new ch14-ex1800 --inline-template --inline-style
```

Step 2 – Start Ng Serve

```
cd ch14-ex1800
ng serve
```

Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1>
      {{count}}
    </h1>
  `,
  styles: []
})
export class AppComponent implements OnInit, OnDestroy{
  interval;
  count = 0;

  ngOnInit(){
    this.interval = setInterval(() => {
      this.count++;
    })
  }

  ngOnDestroy(){
    clearInterval(this.interval);
    delete this.interval;
  }
}
```

Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The App component has ‘ngOnInit’ method that is fired once it has loaded. It initializes the interval, which counts up.
2. The App component has ‘ngOnDestroy’ method that is fired when it is being destroyed. It clears the interval, which stops the counting up.

15 Dependency Injection

15.1 Introduction

In software engineering, **dependency injection** is a software design pattern that implements inversion of control for resolving **dependencies**. A **dependency** is an object that can be used (a service). An **injection** is the passing of a **dependency** to a dependent object (a client) that would use it.

For example, the code below:

```
var svc = new ShippingService(new ProductLocator(), new PricingService(), new InventoryService(), new TrackingRepository(new ConfigProvider()), new Logger(new EmailLogger(new ConfigProvider())));
```

could be replaced by something like:

```
var svc = container.Resolve<IShippingService>();
```

15.2 Advantages of Dependency Injection

1. Your code is cleaner and more readable.
2. The objects are loosely coupled.
3. Dependency injection makes it possible to eliminate, or at least reduce, a components unnecessary dependencies.
4. Reducing a component's dependencies typically makes it easier to reuse in a different context.
5. Dependency injection also increases a components testability.
6. Dependency injection moves the dependencies to the interface of components. So you don't reference the dependencies explicitly, you reference them via interfaces.

15.3 Angular Provided Services

Service	Description
Http	For HTTP communication with the server.
Form	Form handler code.
Router	Page navigation code.
Animation	UI animations.
UI Library	E.g. NgBootstrap

15.4 Other Services

Remember there are others that you can get at ngmodules.org.

15.5 Services You May Want to Write Specifically for Your Own Project

You might want to write specific implementations of the following services: Server Communication.

Security.

Auditing.

Logging.

Session.

Remember that your implementations can ‘wrap’ other services. For example, your Server Communication service could itself use the Angular Http service and add more functionality, implement something differently or just have different configuration.

15.6 Writing Service Classes

When you write services, you typically write them as typescript classes, with one file ('filename.service.ts') per class. It is a good idea marking these classes as injectable using the '@Injectable()' annotation. @Injectable() marks a class as available to an injector for instantiation. Generally speaking, an injector reports an error when trying to instantiate a class that is not marked as '@Injectable()'.

15.7 Providers

Providers are used to register classes, functions or values so that they can be used by the Dependency Injection. The Injector class uses the Provider to supply information so that it can create an instance of an Object to be injected into another. So, a Provider is basically a source of information on how to create an instance of an Object. This information includes a ‘token’, the identifier of an Object that may need to be created. When you see ‘provide(‘ in your Angular code, you are seeing a call to an Angular function to register information on how to create an Object.

There are three types of Providers: Class Providers, Factory Providers and Value Providers. We will introduce the Class Providers first and show examples with them first because they are used the most.

15.8 Injectors

Each Component has its own Injector that is used *with the Providers* to create Objects for the Component. When Components have child Components, the Injector creates Child Injectors for the Child Components.

15.9 Resolving Dependencies

When the Dependency Injection needs to inject an Object into a Component, it attempts to resolve the Object in the local Injector (i.e. the one for the Component) using the ‘get’ method. If that cannot be resolved (in other words the Object does not already exist in the Injector), it attempts to resolve the Object in the Parent Component’s Injector, and so forth all the way to the Application Component. This ensures that the nearest (local) injector’s Provider is used in preference to a higher-level Provider. This is known as ‘Shadowing’ and it is similar to how local variables with the same name are used in preference to global variables of the same name.

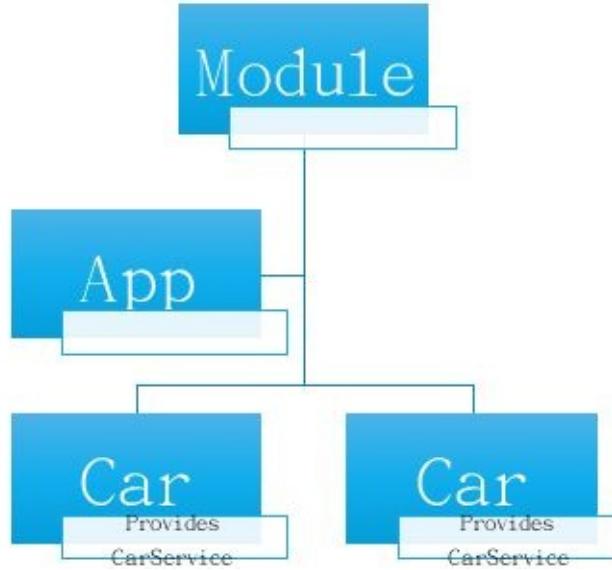
Note that normally Angular handles the resolution and creation of dependencies for you. However, the Injector class offers you methods to invoke this yourself, for example ‘resolveAndCreate’.

15.10 Creating Service – Example

Ford GT Is Supercharged: yes

Corvette Z06 Is Supercharged: no

This is a simple component that uses a service to provide information about cars. This will be example ‘ch15-ex100’. It will provide the service in the Car component.



- **Step 1 – Build the App using the CLI**

```
ng new ch15-ex100 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd exercise7  
ng ch15-ex100
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Create Service Class**

Edit ‘car.service.ts’ and change it to the following:

```
import { Injectable } from '@angular/core';  
@Injectable()  
export class CarService {  
  constructor(){  
    console.log('CarService: constructor'); }  
  // Some dummy method.  
  isSuperCharged(car: string){  
    return car === 'Ford GT' ? 'yes' : 'no'; }  
}
```

● Step 5 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit, Input } from '@angular/core'; import { CarService } from './car.service';

@Component({
  selector: 'car',
  template: `
    <h3>
      {{name}} Is Supercharged: {{supercharged}}
    </h3>
  `,
  styles: [],
  providers: [CarService]
})
export class CarComponent implements OnInit{
  @Input() name;
  supercharged: string = "";
  constructor(private service: CarService){}
  ngOnInit(){
    this.supercharged = this.service.isSuperCharged(this.name);
  }
}

@Component({
  selector: 'app-root',
  template: `
    <car name="Ford GT"></car>
    <car name="Corvette Z06"></car> `,
  styles: []
})
export class AppComponent {
  title = 'app works!';
}
```

● Step 6 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

import { AppComponent, CarComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent, CarComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

The car service outputs a log when in the constructor. This service contains a method ‘isSuperCharged’ that receives a car name as an argument and returns a ‘yes’ or ‘no’ accordingly.

The app component has a car component that is used twice. The car component specifies the car service as a provider. The car component invokes the ‘service’ method ‘isSuperCharged’ in the method ‘ngOnInit’. ‘ngOnInit’ is fired when the component has been initialized.

15.11 Why Do Multiple Instances of the Same Service Get Created?

Open the console and you will see the following:

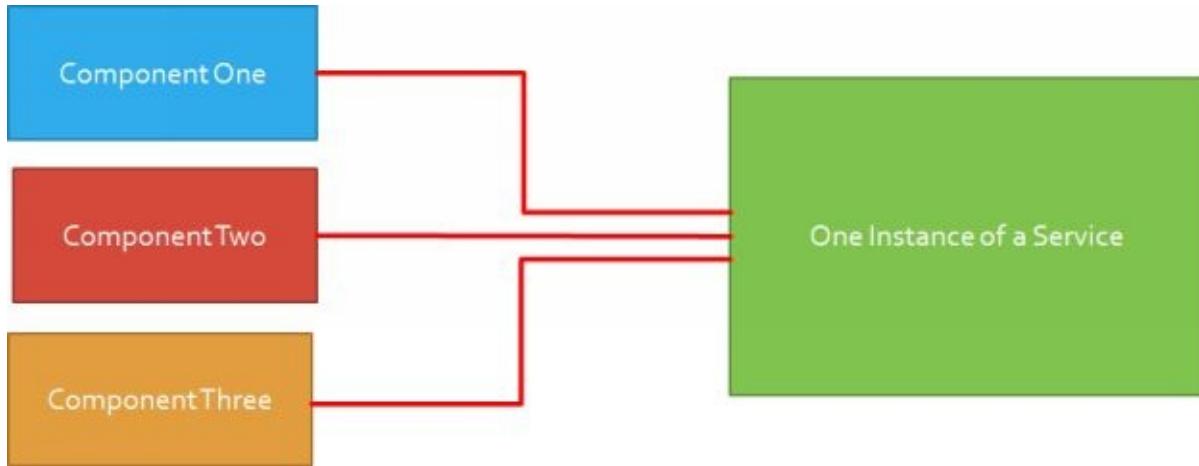
```
CarService: constructor  
CarService: constructor
```

As you can see from the console when you run this example, you can see that the constructor is invoked twice as the service is created twice. That is because the CarService is provided in the Car component and the Car component is create twice. Here is an excerpt of the Car component:

```
@Component({  
  selector: 'car',  
  ...  
  providers: [CarService]  
})  
export class CarComponent implements OnInit{  
  ...  
  constructor(private service: CarService){}  
  ...  
}
```

15.12 How Do We Ensure a Single Instance of the Service Gets Created?

What we want is this:



To ensure that a single instance of the service gets created, we simply move the provider to the application level or a class that is only being used once. In the code example, we could specify the provider in the

15.13 What Do We Do If We Want to Share a Singleton of the Service?

We don't specify that we need it on the Car object, because there are multiple Cars.

We need to specify that we need the Service somewhere else on an **application-level**.

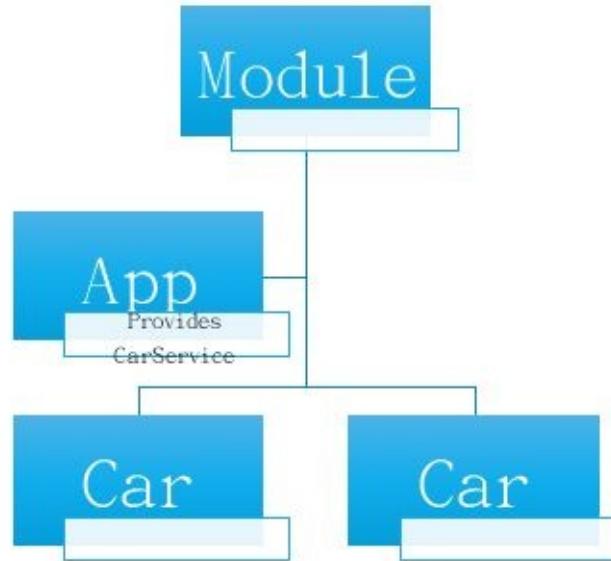
Now let's convert our App to share **one** instance of the service.

15.14 Convert App to Share One Instance of Service – Example #1

Ford GT Is Supercharged: yes

Corvette Z06 Is Supercharged: no

This is a simple component that uses a service to provide information about cars. This will be example ‘ch15-ex200’ and it will be the same as ‘ch15-ex100’ except that it provides the service in the App component so that only one instance of the Car service is created.



- Step 1 – Build the App using the CLI

```
ng new ch15-ex200 --inline-template --inline-style
```

- Step 2 – Start Ng Serve

```
cd ch15-ex200  
ng serve
```

- Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- Step 4 – Create Service Class

This is the same as the previous example. Edit ‘car.service.ts’ and change it to the following:

```
import { Injectable } from '@angular/core';  
@Injectable()  
export class CarService {  
  constructor(){  
    console.log('CarService: constructor'); }  
  // Some dummy method.  
  isSuperCharged(car: string){
```

```
    return car === 'Ford GT' ? 'yes' : 'no'; }
```

```
}
```

● Step 5 – Edit Class

This is different to the previous example. Edit ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit, Input } from '@angular/core'; import { CarService } from './car.service';

@Component({
  selector: 'car',
  template: `
    <h3>
      {{name}} Is Supercharged: {{supercharged}}
    </h3>
    `,
  styles: [],
  providers: []
})
export class CarComponent implements OnInit{
  @Input() name;
  supercharged: string = "";
  constructor(private service: CarService){}
  ngOnInit(){
    this.supercharged = this.service.isSuperCharged(this.name);
  }

@Component({
  selector: 'app-root',
  template: `
    <car name="Ford GT"></car>
    <car name="Corvette Z06"></car> `,
  styles: [],
  providers: [CarService]
})
export class AppComponent {
  title = 'app works!';
}
```

● Step 6 – Edit Module

This is the same as the previous example. Edit ‘app.module.ts’ and change it to the following:

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

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

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

• Exercise Complete

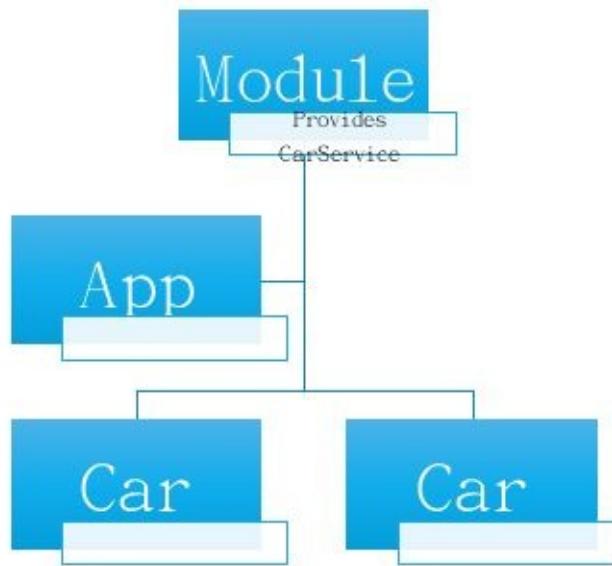
Your app should be working at localhost:4200. Note that the Car service constructor is only logged once in the console. This is because the Car service only needs to be created once in the App component and it can be used by all subcomponents.

15.15 Convert App to Share One Instance of Service – Example #2

Ford GT Is Supercharged: yes

Corvette Z06 Is Supercharged: no

This is a simple component that uses a service to provide information about cars. This will be example ‘ch15-ex300’ and it will be the same as ‘ch15-ex200’ except that it provides the service in the module so that only one instance of the Car service is created and it can be used anywhere in the app.



- **Step 1 – Build the App using the CLI**

```
ng new ch15-ex300 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch15-ex300  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Create Service Class**

This is the same as the previous example. Edit ‘car.service.ts’ and change it to the following:

```
import { Injectable } from '@angular/core';  
@Injectable()  
export class CarService {  
    constructor(){  
        console.log('CarService: constructor');  
    }  
    // Some dummy method.  
}
```

```
isSuperCharged(car: string){  
    return car === 'Ford GT' ? 'yes' : 'no'; }  
}
```

● Step 5 – Edit Class

This is different to the previous example. Edit ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit, Input } from '@angular/core'; import { CarService } from './car.service';  
  
@Component({  
    selector: 'car',  
    template: `  
        <h3>  
            {{name}} Is Supercharged: {{supercharged}}  
        </h3>  
        `,  
    styles: []  
})  
export class CarComponent implements OnInit{  
    @Input() name;  
    supercharged: string = "";  
    constructor(private service: CarService){}  
    ngOnInit(){  
        this.supercharged = this.service.isSuperCharged(this.name); }  
    }  
  
@Component({  
    selector: 'app-root',  
    template: `  
        <car name="Ford GT"></car>  
        <car name="Corvette Z06"></car> `,  
    styles: []  
})  
export class AppComponent {  
    title = 'app works!';  
}
```

● Step 6 – Edit Module

This is different to the previous example. Edit ‘app.module.ts’ and change it to the following:

```
import { BrowserModule } from '@angular/platform-browser'; import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { HttpClientModule } from '@angular/http';  
  
import { AppComponent, CarComponent } from './app.component'; import { CarService } from './car.service';  
  
@NgModule({  
    declarations: [  
        AppComponent, CarComponent  
    ],  
    imports: [  
        BrowserModule,  
        FormsModule,  
        HttpClientModule  
    ],
```

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

• Exercise Complete

Your app should be working at localhost:4200. Note that the Car service constructor is only logged once in the console. This is because the Car service only needs to be created once in the App module and it can be used by all subcomponents.

15.16 Class Providers

As mentioned earlier, there are three types of Providers: Class Providers, Factory Providers and Value Providers. Class Providers allow us to tell the Provider which class to use for a Dependency.

15.17 Class Providers – Example

```
Seiko Time:Mon Jun 12 2017  
21:29:40 GMT-0400 (EDT)
```

In the example below we have a component that relies on a Watch service. This will be example ‘ch15-ex350’.

- **Step 1 – Build the App using the CLI**

```
ng new ch15-ex350 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch15-ex350  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Edit Class**

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core';
class Watch {
getTime(): string {
    return new Date() + "";
}
}

class Seiko extends Watch {
getTime(): string{
    return "Seiko Time:" + super.getTime();
}
}

@Component({
selector: 'app-root',
template: `

<h1>
    {{watch.getTime()}}
</h1>
`,
styles: [],
providers: [
    {provide: Watch,
     useClass: Seiko
}]
})
export class AppComponent {
constructor(private watch:Watch){}
}
```

● Exercise Complete

Your app should be working at localhost:4200. Note that when we use the Provider element of the @Component annotation to create the dependency, we specify that a subclass of the Watch (a Seiko).

15.18 Factory Providers

As mentioned earlier, there are three types of Providers: Class Providers, Factory Providers and Value Providers. Factory providers use a function to provide Angular with an instance of an Object. This is useful when you need to dynamically change the Object that you want created, based on some data.

15.19 Factory Providers – Example #1

This is a simple component that uses a logging service. This will be example ‘ch15-ex400’.

• Step 1 – Build the App using the CLI

```
ng new ch15-ex400 --inline-template --inline-style
```

• Step 2 – Start Ng Serve

```
cd ch15-ex400  
ng serve
```

• Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

• Step 4 – Create Service Class

This is the same as the previous example. Edit ‘car.service.ts’ and change it to the following:

```
import { Injectable } from '@angular/core';  
@Injectable()  
export class LoggingService {  
    constructor(private dateAndTime: boolean){  
        console.log('LoggingService: constructor'); }  
    log(message){  
        console.log((this.dateAndTime ? new Date() + ':' : '') + message); }  
}
```

• Step 5 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { LoggingService } from './logging.service';  
  
@Component({  
    selector: 'app-root',  
    template: `'  
        <h1>  
            {{title}}  
        </h1>  
        ;  
    styles: [],  
    providers: [provideLoggingService()]  
})  
export class AppComponent {  
    constructor(private logging: LoggingService){  
        logging.log('test log');  
    }  
    title = 'app works!';  
}  
export const LOGGING_USE_DATE = false;  
export function provideLoggingService() {  
    return {
```

```
provide: LoggingService,  
useFactory: () => new LoggingService(LOGGING_USE_DATE) }  
}
```

• Exercise Complete

Your app should be working at localhost:4200. Note that this logging service has the option of including the logging date and time. You can set this using the constructor to the logging service. A factory provider is used to provide an instance of the logging service.

Date Included in Logging

```
LoggingService: constructor  
Sun Jun 11 2017 20:10:01 GMT-0400 (EDT): test log  
Angular is running in the development mode. Call e
```

```
export const LOGGING_USE_DATE = true; export function provideLoggingService() {  
return {  
  provide: LoggingService,  
  useFactory: () => new LoggingService(LOGGING_USE_DATE) }  
}
```

Date Not Included in Logging

```
LoggingService: constructor  
test log  
Angular is running in the development mode.  
.
```

```
export const LOGGING_USE_DATE = false; export function provideLoggingService() {  
return {  
  provide: LoggingService,  
  useFactory: () => new LoggingService(LOGGING_USE_DATE) }  
}
```

15.20 Factory Providers – Example #2

This is a simple component that displays a Playing Card. This will be example ‘ch15-ex500’.

Card is King of Diamonds

- Step 1 – Build the App using the CLI

```
ng new ch15-ex500 --inline-template --inline-style
```

- Step 2 – Start Ng Serve

```
cd ch15-ex500  
ng serve
```

- Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- Step 4 – Create Card Class

This is the same as the previous example. Edit ‘card.ts’ and change it to the following:

```
import { Injectable } from '@angular/core';  
@Injectable()  
export class Card {  
    constructor(public suite: string, public rank: string) {}  
    toString(): string {  
        return "Card is " + this.rank + " of " + this.suite;  
    }  
}
```

- Step 5 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { Card } from './card';  
@Component({  
    selector: 'app-root',  
    template: `  
        <h1>  
            {{title}}  
        </h1>  
        ;  
    styles: [],  
    providers: [{  
        provide: Card,  
        useFactory: () => {  
            const suite: number = Math.floor(Math.random() * 4); const suiteName: string =  
            suite == 0 ? "Clubs" :  
            suite == 1 ? "Diamonds" :  
            suite == 2 ? "Hearts" : "Spades"; const rank: number = Math.floor(Math.random() * 15); const rankName: string =  
            rank == 0 ? "Ace" :  
            rank == 1 ? "Joker" :  
            rank == 2 ? "King" :  
            rank == 3 ? "Queen" :  
            rank == 4 ? "Jack" :  
            rank == 5 ? "Ten" :  
            rank == 6 ? "Nine" :  
            rank == 7 ? "Eight" :  
            rank == 8 ? "Seven" :  
            rank == 9 ? "Six" :  
            rank == 10 ? "Five" :  
            rank == 11 ? "Four" :  
            rank == 12 ? "Three" :  
            rank == 13 ? "Two" :  
            rank == 14 ? "One" :  
            rank == 15 ? "King" :  
            rank == 16 ? "Queen" :  
            rank == 17 ? "Jack" :  
            rank == 18 ? "Ten" :  
            rank == 19 ? "Nine" :  
            rank == 20 ? "Eight" :  
            rank == 21 ? "Seven" :  
            rank == 22 ? "Six" :  
            rank == 23 ? "Five" :  
            rank == 24 ? "Four" :  
            rank == 25 ? "Three" :  
            rank == 26 ? "Two" :  
            rank == 27 ? "One" :  
            rank == 28 ? "King" :  
            rank == 29 ? "Queen" :  
            rank == 30 ? "Jack" :  
            rank == 31 ? "Ten" :  
            rank == 32 ? "Nine" :  
            rank == 33 ? "Eight" :  
            rank == 34 ? "Seven" :  
            rank == 35 ? "Six" :  
            rank == 36 ? "Five" :  
            rank == 37 ? "Four" :  
            rank == 38 ? "Three" :  
            rank == 39 ? "Two" :  
            rank == 40 ? "One" :  
            rank == 41 ? "King" :  
            rank == 42 ? "Queen" :  
            rank == 43 ? "Jack" :  
            rank == 44 ? "Ten" :  
            rank == 45 ? "Nine" :  
            rank == 46 ? "Eight" :  
            rank == 47 ? "Seven" :  
            rank == 48 ? "Six" :  
            rank == 49 ? "Five" :  
            rank == 50 ? "Four" :  
            rank == 51 ? "Three" :  
            rank == 52 ? "Two" :  
            rank == 53 ? "One" :  
            rank == 54 ? "King" :  
            rank == 55 ? "Queen" :  
            rank == 56 ? "Jack" :  
            rank == 57 ? "Ten" :  
            rank == 58 ? "Nine" :  
            rank == 59 ? "Eight" :  
            rank == 60 ? "Seven" :  
            rank == 61 ? "Six" :  
            rank == 62 ? "Five" :  
            rank == 63 ? "Four" :  
            rank == 64 ? "Three" :  
            rank == 65 ? "Two" :  
            rank == 66 ? "One" :  
            rank == 67 ? "King" :  
            rank == 68 ? "Queen" :  
            rank == 69 ? "Jack" :  
            rank == 70 ? "Ten" :  
            rank == 71 ? "Nine" :  
            rank == 72 ? "Eight" :  
            rank == 73 ? "Seven" :  
            rank == 74 ? "Six" :  
            rank == 75 ? "Five" :  
            rank == 76 ? "Four" :  
            rank == 77 ? "Three" :  
            rank == 78 ? "Two" :  
            rank == 79 ? "One" :  
            rank == 80 ? "King" :  
            rank == 81 ? "Queen" :  
            rank == 82 ? "Jack" :  
            rank == 83 ? "Ten" :  
            rank == 84 ? "Nine" :  
            rank == 85 ? "Eight" :  
            rank == 86 ? "Seven" :  
            rank == 87 ? "Six" :  
            rank == 88 ? "Five" :  
            rank == 89 ? "Four" :  
            rank == 90 ? "Three" :  
            rank == 91 ? "Two" :  
            rank == 92 ? "One" :  
            rank == 93 ? "King" :  
            rank == 94 ? "Queen" :  
            rank == 95 ? "Jack" :  
            rank == 96 ? "Ten" :  
            rank == 97 ? "Nine" :  
            rank == 98 ? "Eight" :  
            rank == 99 ? "Seven" :  
            rank == 100 ? "Six" :  
            rank == 101 ? "Five" :  
            rank == 102 ? "Four" :  
            rank == 103 ? "Three" :  
            rank == 104 ? "Two" :  
            rank == 105 ? "One" :  
            rank == 106 ? "King" :  
            rank == 107 ? "Queen" :  
            rank == 108 ? "Jack" :  
            rank == 109 ? "Ten" :  
            rank == 110 ? "Nine" :  
            rank == 111 ? "Eight" :  
            rank == 112 ? "Seven" :  
            rank == 113 ? "Six" :  
            rank == 114 ? "Five" :  
            rank == 115 ? "Four" :  
            rank == 116 ? "Three" :  
            rank == 117 ? "Two" :  
            rank == 118 ? "One" :  
            rank == 119 ? "King" :  
            rank == 120 ? "Queen" :  
            rank == 121 ? "Jack" :  
            rank == 122 ? "Ten" :  
            rank == 123 ? "Nine" :  
            rank == 124 ? "Eight" :  
            rank == 125 ? "Seven" :  
            rank == 126 ? "Six" :  
            rank == 127 ? "Five" :  
            rank == 128 ? "Four" :  
            rank == 129 ? "Three" :  
            rank == 130 ? "Two" :  
            rank == 131 ? "One" :  
            rank == 132 ? "King" :  
            rank == 133 ? "Queen" :  
            rank == 134 ? "Jack" :  
            rank == 135 ? "Ten" :  
            rank == 136 ? "Nine" :  
            rank == 137 ? "Eight" :  
            rank == 138 ? "Seven" :  
            rank == 139 ? "Six" :  
            rank == 140 ? "Five" :  
            rank == 141 ? "Four" :  
            rank == 142 ? "Three" :  
            rank == 143 ? "Two" :  
            rank == 144 ? "One" :  
            rank == 145 ? "King" :  
            rank == 146 ? "Queen" :  
            rank == 147 ? "Jack" :  
            rank == 148 ? "Ten" :  
            rank == 149 ? "Nine" :  
            rank == 150 ? "Eight" :  
            rank == 151 ? "Seven" :  
            rank == 152 ? "Six" :  
            rank == 153 ? "Five" :  
            rank == 154 ? "Four" :  
            rank == 155 ? "Three" :  
            rank == 156 ? "Two" :  
            rank == 157 ? "One" :  
            rank == 158 ? "King" :  
            rank == 159 ? "Queen" :  
            rank == 160 ? "Jack" :  
            rank == 161 ? "Ten" :  
            rank == 162 ? "Nine" :  
            rank == 163 ? "Eight" :  
            rank == 164 ? "Seven" :  
            rank == 165 ? "Six" :  
            rank == 166 ? "Five" :  
            rank == 167 ? "Four" :  
            rank == 168 ? "Three" :  
            rank == 169 ? "Two" :  
            rank == 170 ? "One" :  
            rank == 171 ? "King" :  
            rank == 172 ? "Queen" :  
            rank == 173 ? "Jack" :  
            rank == 174 ? "Ten" :  
            rank == 175 ? "Nine" :  
            rank == 176 ? "Eight" :  
            rank == 177 ? "Seven" :  
            rank == 178 ? "Six" :  
            rank == 179 ? "Five" :  
            rank == 180 ? "Four" :  
            rank == 181 ? "Three" :  
            rank == 182 ? "Two" :  
            rank == 183 ? "One" :  
            rank == 184 ? "King" :  
            rank == 185 ? "Queen" :  
            rank == 186 ? "Jack" :  
            rank == 187 ? "Ten" :  
            rank == 188 ? "Nine" :  
            rank == 189 ? "Eight" :  
            rank == 190 ? "Seven" :  
            rank == 191 ? "Six" :  
            rank == 192 ? "Five" :  
            rank == 193 ? "Four" :  
            rank == 194 ? "Three" :  
            rank == 195 ? "Two" :  
            rank == 196 ? "One" :  
            rank == 197 ? "King" :  
            rank == 198 ? "Queen" :  
            rank == 199 ? "Jack" :  
            rank == 200 ? "Ten" :  
            rank == 201 ? "Nine" :  
            rank == 202 ? "Eight" :  
            rank == 203 ? "Seven" :  
            rank == 204 ? "Six" :  
            rank == 205 ? "Five" :  
            rank == 206 ? "Four" :  
            rank == 207 ? "Three" :  
            rank == 208 ? "Two" :  
            rank == 209 ? "One" :  
            rank == 210 ? "King" :  
            rank == 211 ? "Queen" :  
            rank == 212 ? "Jack" :  
            rank == 213 ? "Ten" :  
            rank == 214 ? "Nine" :  
            rank == 215 ? "Eight" :  
            rank == 216 ? "Seven" :  
            rank == 217 ? "Six" :  
            rank == 218 ? "Five" :  
            rank == 219 ? "Four" :  
            rank == 220 ? "Three" :  
            rank == 221 ? "Two" :  
            rank == 222 ? "One" :  
            rank == 223 ? "King" :  
            rank == 224 ? "Queen" :  
            rank == 225 ? "Jack" :  
            rank == 226 ? "Ten" :  
            rank == 227 ? "Nine" :  
            rank == 228 ? "Eight" :  
            rank == 229 ? "Seven" :  
            rank == 230 ? "Six" :  
            rank == 231 ? "Five" :  
            rank == 232 ? "Four" :  
            rank == 233 ? "Three" :  
            rank == 234 ? "Two" :  
            rank == 235 ? "One" :  
            rank == 236 ? "King" :  
            rank == 237 ? "Queen" :  
            rank == 238 ? "Jack" :  
            rank == 239 ? "Ten" :  
            rank == 240 ? "Nine" :  
            rank == 241 ? "Eight" :  
            rank == 242 ? "Seven" :  
            rank == 243 ? "Six" :  
            rank == 244 ? "Five" :  
            rank == 245 ? "Four" :  
            rank == 246 ? "Three" :  
            rank == 247 ? "Two" :  
            rank == 248 ? "One" :  
            rank == 249 ? "King" :  
            rank == 250 ? "Queen" :  
            rank == 251 ? "Jack" :  
            rank == 252 ? "Ten" :  
            rank == 253 ? "Nine" :  
            rank == 254 ? "Eight" :  
            rank == 255 ? "Seven" :  
            rank == 256 ? "Six" :  
            rank == 257 ? "Five" :  
            rank == 258 ? "Four" :  
            rank == 259 ? "Three" :  
            rank == 260 ? "Two" :  
            rank == 261 ? "One" :  
            rank == 262 ? "King" :  
            rank == 263 ? "Queen" :  
            rank == 264 ? "Jack" :  
            rank == 265 ? "Ten" :  
            rank == 266 ? "Nine" :  
            rank == 267 ? "Eight" :  
            rank == 268 ? "Seven" :  
            rank == 269 ? "Six" :  
            rank == 270 ? "Five" :  
            rank == 271 ? "Four" :  
            rank == 272 ? "Three" :  
            rank == 273 ? "Two" :  
            rank == 274 ? "One" :  
            rank == 275 ? "King" :  
            rank == 276 ? "Queen" :  
            rank == 277 ? "Jack" :  
            rank == 278 ? "Ten" :  
            rank == 279 ? "Nine" :  
            rank == 280 ? "Eight" :  
            rank == 281 ? "Seven" :  
            rank == 282 ? "Six" :  
            rank == 283 ? "Five" :  
            rank == 284 ? "Four" :  
            rank == 285 ? "Three" :  
            rank == 286 ? "Two" :  
            rank == 287 ? "One" :  
            rank == 288 ? "King" :  
            rank == 289 ? "Queen" :  
            rank == 290 ? "Jack" :  
            rank == 291 ? "Ten" :  
            rank == 292 ? "Nine" :  
            rank == 293 ? "Eight" :  
            rank == 294 ? "Seven" :  
            rank == 295 ? "Six" :  
            rank == 296 ? "Five" :  
            rank == 297 ? "Four" :  
            rank == 298 ? "Three" :  
            rank == 299 ? "Two" :  
            rank == 300 ? "One" :  
            rank == 301 ? "King" :  
            rank == 302 ? "Queen" :  
            rank == 303 ? "Jack" :  
            rank == 304 ? "Ten" :  
            rank == 305 ? "Nine" :  
            rank == 306 ? "Eight" :  
            rank == 307 ? "Seven" :  
            rank == 308 ? "Six" :  
            rank == 309 ? "Five" :  
            rank == 310 ? "Four" :  
            rank == 311 ? "Three" :  
            rank == 312 ? "Two" :  
            rank == 313 ? "One" :  
            rank == 314 ? "King" :  
            rank == 315 ? "Queen" :  
            rank == 316 ? "Jack" :  
            rank == 317 ? "Ten" :  
            rank == 318 ? "Nine" :  
            rank == 319 ? "Eight" :  
            rank == 320 ? "Seven" :  
            rank == 321 ? "Six" :  
            rank == 322 ? "Five" :  
            rank == 323 ? "Four" :  
            rank == 324 ? "Three" :  
            rank == 325 ? "Two" :  
            rank == 326 ? "One" :  
            rank == 327 ? "King" :  
            rank == 328 ? "Queen" :  
            rank == 329 ? "Jack" :  
            rank == 330 ? "Ten" :  
            rank == 331 ? "Nine" :  
            rank == 332 ? "Eight" :  
            rank == 333 ? "Seven" :  
            rank == 334 ? "Six" :  
            rank == 335 ? "Five" :  
            rank == 336 ? "Four" :  
            rank == 337 ? "Three" :  
            rank == 338 ? "Two" :  
            rank == 339 ? "One" :  
            rank == 340 ? "King" :  
            rank == 341 ? "Queen" :  
            rank == 342 ? "Jack" :  
            rank == 343 ? "Ten" :  
            rank == 344 ? "Nine" :  
            rank == 345 ? "Eight" :  
            rank == 346 ? "Seven" :  
            rank == 347 ? "Six" :  
            rank == 348 ? "Five" :  
            rank == 349 ? "Four" :  
            rank == 350 ? "Three" :  
            rank == 351 ? "Two" :  
            rank == 352 ? "One" :  
            rank == 353 ? "King" :  
            rank == 354 ? "Queen" :  
            rank == 355 ? "Jack" :  
            rank == 356 ? "Ten" :  
            rank == 357 ? "Nine" :  
            rank == 358 ? "Eight" :  
            rank == 359 ? "Seven" :  
            rank == 360 ? "Six" :  
            rank == 361 ? "Five" :  
            rank == 362 ? "Four" :  
            rank == 363 ? "Three" :  
            rank == 364 ? "Two" :  
            rank == 365 ? "One" :  
            rank == 366 ? "King" :  
            rank == 367 ? "Queen" :  
            rank == 368 ? "Jack" :  
            rank == 369 ? "Ten" :  
            rank == 370 ? "Nine" :  
            rank == 371 ? "Eight" :  
            rank == 372 ? "Seven" :  
            rank == 373 ? "Six" :  
            rank == 374 ? "Five" :  
            rank == 375 ? "Four" :  
            rank == 376 ? "Three" :  
            rank == 377 ? "Two" :  
            rank == 378 ? "One" :  
            rank == 379 ? "King" :  
            rank == 380 ? "Queen" :  
            rank == 381 ? "Jack" :  
            rank == 382 ? "Ten" :  
            rank == 383 ? "Nine" :  
            rank == 384 ? "Eight" :  
            rank == 385 ? "Seven" :  
            rank == 386 ? "Six" :  
            rank == 387 ? "Five" :  
            rank == 388 ? "Four" :  
            rank == 389 ? "Three" :  
            rank == 390 ? "Two" :  
            rank == 391 ? "One" :  
            rank == 392 ? "King" :  
            rank == 393 ? "Queen" :  
            rank == 394 ? "Jack" :  
            rank == 395 ? "Ten" :  
            rank == 396 ? "Nine" :  
            rank == 397 ? "Eight" :  
            rank == 398 ? "Seven" :  
            rank == 399 ? "Six" :  
            rank == 400 ? "Five" :  
            rank == 401 ? "Four" :  
            rank == 402 ? "Three" :  
            rank == 403 ? "Two" :  
            rank == 404 ? "One" :  
            rank == 405 ? "King" :  
            rank == 406 ? "Queen" :  
            rank == 407 ? "Jack" :  
            rank == 408 ? "Ten" :  
            rank == 409 ? "Nine" :  
            rank == 410 ? "Eight" :  
            rank == 411 ? "Seven" :  
            rank == 412 ? "Six" :  
            rank == 413 ? "Five" :  
            rank == 414 ? "Four" :  
            rank == 415 ? "Three" :  
            rank == 416 ? "Two" :  
            rank == 417 ? "One" :  
            rank == 418 ? "King" :  
            rank == 419 ? "Queen" :  
            rank == 420 ? "Jack" :  
            rank == 421 ? "Ten" :  
            rank == 422 ? "Nine" :  
            rank == 423 ? "Eight" :  
            rank == 424 ? "Seven" :  
            rank == 425 ? "Six" :  
            rank == 426 ? "Five" :  
            rank == 427 ? "Four" :  
            rank == 428 ? "Three" :  
            rank == 429 ? "Two" :  
            rank == 430 ? "One" :  
            rank == 431 ? "King" :  
            rank == 432 ? "Queen" :  
            rank == 433 ? "Jack" :  
            rank == 434 ? "Ten" :  
            rank == 435 ? "Nine" :  
            rank == 436 ? "Eight" :  
            rank == 437 ? "Seven" :  
            rank == 438 ? "Six" :  
            rank == 439 ? "Five" :  
            rank == 440 ? "Four" :  
            rank == 441 ? "Three" :  
            rank == 442 ? "Two" :  
            rank == 443 ? "One" :  
            rank == 444 ? "King" :  
            rank == 445 ? "Queen" :  
            rank == 446 ? "Jack" :  
            rank == 447 ? "Ten" :  
            rank == 448 ? "Nine" :  
            rank == 449 ? "Eight" :  
            rank == 450 ? "Seven" :  
            rank == 451 ? "Six" :  
            rank == 452 ? "Five" :  
            rank == 453 ? "Four" :  
            rank == 454 ? "Three" :  
            rank == 455 ? "Two" :  
            rank == 456 ? "One" :  
            rank == 457 ? "King" :  
            rank == 458 ? "Queen" :  
            rank == 459 ? "Jack" :  
            rank == 460 ? "Ten" :  
            rank == 461 ? "Nine" :  
            rank == 462 ? "Eight" :  
            rank == 463 ? "Seven" :  
            rank == 464 ? "Six" :  
            rank == 465 ? "Five" :  
            rank == 466 ? "Four" :  
            rank == 467 ? "Three" :  
            rank == 468 ? "Two" :  
            rank == 469 ? "One" :  
            rank == 470 ? "King" :  
            rank == 471 ? "Queen" :  
            rank == 472 ? "Jack" :  
            rank == 473 ? "Ten" :  
            rank == 474 ? "Nine" :  
            rank == 475 ? "Eight" :  
            rank == 476 ? "Seven" :  
            rank == 477 ? "Six" :  
            rank == 478 ? "Five" :  
            rank == 479 ? "Four" :  
            rank == 480 ? "Three" :  
            rank == 481 ? "Two" :  
            rank == 482 ? "One" :  
            rank == 483 ? "King" :  
            rank == 484 ? "Queen" :  
            rank == 485 ? "Jack" :  
            rank == 486 ? "Ten" :  
            rank == 487 ? "Nine" :  
            rank == 488 ? "Eight" :  
            rank == 489 ? "Seven" :  
            rank == 490 ? "Six" :  
            rank == 491 ? "Five" :  
            rank == 492 ? "Four" :  
            rank == 493 ? "Three" :  
            rank == 494 ? "Two" :  
            rank == 495 ? "One" :  
            rank == 496 ? "King" :  
            rank == 497 ? "Queen" :  
            rank == 498 ? "Jack" :  
            rank == 499 ? "Ten" :  
            rank == 500 ? "Nine" :  
            rank == 501 ? "Eight" :  
            rank == 502 ? "Seven" :  
            rank == 503 ? "Six" :  
            rank == 504 ? "Five" :  
            rank == 505 ? "Four" :  
            rank == 506 ? "Three" :  
            rank == 507 ? "Two" :  
            rank == 508 ? "One" :  
            rank == 509 ? "King" :  
            rank == 510 ? "Queen" :  
            rank == 511 ? "Jack" :  
            rank == 512 ? "Ten" :  
            rank == 513 ? "Nine" :  
            rank == 514 ? "Eight" :  
            rank == 515 ? "Seven" :  
            rank == 516 ? "Six" :  
            rank == 517 ? "Five" :  
            rank == 518 ? "Four" :  
            rank == 519 ? "Three" :  
            rank == 520 ? "Two" :  
            rank == 521 ? "One" :  
            rank == 522 ? "King" :  
            rank == 523 ? "Queen" :  
            rank == 524 ? "Jack" :  
            rank == 525 ? "Ten" :  
            rank == 526 ? "Nine" :  
            rank == 527 ? "Eight" :  
            rank == 528 ? "Seven" :  
            rank == 529 ? "Six" :  
            rank == 530 ? "Five" :  
            rank == 531 ? "Four" :  
            rank == 532 ? "Three" :  
            rank == 533 ? "Two" :  
            rank == 534 ? "One" :  
            rank == 535 ? "King" :  
            rank == 536 ? "Queen" :  
            rank == 537 ? "Jack" :  
            rank == 538 ? "Ten" :  
            rank == 539 ? "Nine" :  
            rank == 540 ? "Eight" :  
            rank == 541 ? "Seven" :  
            rank == 542 ? "Six" :  
            rank == 543 ? "Five" :  
            rank == 544 ? "Four" :  
            rank == 545 ? "Three" :  
            rank == 546 ? "Two" :  
            rank == 547 ? "One" :  
            rank == 548 ? "King" :  
            rank == 549 ? "Queen" :  
            rank == 550 ? "Jack" :  
            rank == 551 ? "Ten" :  
            rank == 552 ? "Nine" :  
            rank == 553 ? "Eight" :  
            rank == 554 ? "Seven" :  
            rank == 555 ? "Six" :  
            rank == 556 ? "Five" :  
            rank == 557 ? "Four" :  
            rank == 558 ? "Three" :  
            rank == 559 ? "Two" :  
            rank == 560 ? "One" :  
            rank == 561 ? "King" :  
            rank == 562 ? "Queen" :  
            rank == 563 ? "Jack" :  
            rank == 564 ? "Ten" :  
            rank == 565 ? "Nine" :  
            rank == 566 ? "Eight" :  
            rank == 567 ? "Seven" :  
            rank == 568 ? "Six" :  
            rank == 569 ? "Five" :  
            rank == 570 ? "Four" :  
            rank == 571 ? "Three" :  
            rank == 572 ? "Two" :  
            rank == 573 ? "One" :  
            rank == 574 ? "King" :  
            rank == 575 ? "Queen" :  
            rank == 576 ? "Jack" :  
            rank == 577 ? "Ten" :  
            rank == 578 ? "Nine" :  
            rank == 579 ? "Eight" :  
            rank == 580 ? "Seven" :  
            rank == 581 ? "Six" :  
            rank == 582 ? "Five" :  
            rank == 583 ? "Four" :  
            rank == 584 ? "Three" :  
            rank == 585 ? "Two" :  
            rank == 586 ? "One" :  
            rank == 587 ? "King" :  
            rank == 588 ? "Queen" :  
            rank == 589 ? "Jack" :  
            rank == 590 ? "Ten" :  
            rank == 591 ? "Nine" :  
            rank == 592 ? "Eight" :  
            rank == 593 ? "Seven" :  
            rank == 594 ? "Six" :  
            rank == 595 ? "Five" :  
            rank == 596 ? "Four" :  
            rank == 597 ? "Three" :  
            rank == 598 ? "Two" :  
            rank == 599 ? "One" :  
            rank == 600 ? "King" :  
            rank == 601 ? "Queen" :  
            rank == 602 ? "Jack" :  
            rank == 603 ? "Ten" :  
            rank == 604 ? "Nine" :  
            rank == 605 ? "Eight" :  
            rank == 606 ? "Seven" :  
            rank == 607 ? "Six" :  
            rank == 608 ? "Five" :  
            rank == 609 ? "Four" :  
            rank == 610 ? "Three" :  
            rank == 611 ? "Two" :  
            rank == 612 ? "One" :  
            rank == 613 ? "King" :  
            rank == 614 ? "Queen" :  
            rank == 615 ? "Jack" :  
            rank == 616 ? "Ten" :  
            rank == 617 ? "Nine" :  
            rank == 618 ? "Eight" :  
            rank == 619 ? "Seven" :  
            rank == 620 ? "Six" :  
            rank == 621 ? "Five" :  
            rank == 622 ? "Four" :  
            rank == 623 ? "Three" :  
            rank == 624 ? "Two" :  
            rank == 625 ? "One" :  
            rank == 626 ? "King" :  
            rank == 627 ? "Queen" :  
            rank == 628 ? "Jack" :  
            rank == 629 ? "Ten" :  
            rank == 630 ? "Nine" :  
            rank == 631 ? "Eight" :  
            rank == 632 ? "Seven" :  
            rank == 633 ? "Six" :  
            rank == 634 ? "Five" :  
            rank == 635 ? "Four" :  
            rank == 636 ? "Three" :  
            rank == 637 ? "Two" :  
            rank == 638 ? "One" :  
            rank == 639 ? "King" :  
            rank == 640 ? "Queen" :  
            rank == 641 ? "Jack" :  
            rank == 642 ? "Ten" :  
            rank == 643 ? "Nine" :  
            rank == 644 ? "Eight" :  
            rank == 645 ? "Seven" :  
            rank == 646 ? "Six" :  
            rank == 647 ? "Five" :  
            rank == 648 ? "Four" :  
            rank == 649 ? "Three" :  
            rank == 650 ? "Two" :  
            rank == 651 ? "One" :  
            rank == 652 ? "King" :  
            rank == 653 ? "Queen" :  
            rank == 654 ? "Jack" :  
            rank == 655 ? "Ten" :  
            rank == 656 ? "Nine" :  
            rank == 657 ? "Eight" :  
            rank == 658 ? "Seven" :  
            rank == 659 ? "Six" :  
            rank == 660 ? "Five" :  
            rank == 661 ? "Four" :  
            rank == 662 ? "Three" :  
            rank == 663 ? "Two" :  
            rank == 664 ? "One" :  
            rank == 665 ? "King" :  
            rank == 666 ? "Queen" :  
            rank == 667 ? "Jack" :  
            rank == 668 ? "Ten" :  
            rank == 669 ? "Nine" :  
            rank == 670 ? "Eight" :  
            rank == 671 ? "Seven" :  
            rank == 672 ? "Six" :  
            rank == 673 ? "Five" :  
            rank == 674 ? "Four" :  
            rank == 675 ? "Three" :  
            rank == 676 ? "Two" :  
            rank == 677 ? "One" :  
            rank == 678 ? "King" :  
            rank == 679 ? "Queen" :  
            rank == 680 ? "Jack" :  
            rank == 681 ? "Ten" :  
            rank == 682 ? "Nine" :  
            rank == 683 ? "Eight" :  
            rank == 684 ? "Seven" :  
            rank == 685 ? "Six" :  
            rank == 686 ? "Five" :  
            rank == 687 ? "Four" :  
            rank == 688 ? "Three" :  
            rank == 689 ? "Two" :  
            rank == 690 ? "One" :  
            rank == 691 ? "King" :  
            rank == 692 ? "Queen" :  
            rank == 693 ? "Jack" :  
            rank == 694 ? "Ten" :  
            rank == 695 ? "Nine" :  
            rank == 696 ? "Eight" :  
            rank == 697 ? "Seven" :  
            rank == 698 ? "Six" :  
            rank == 699 ? "Five" :  
            rank == 700 ? "Four" :  
            rank == 701 ? "Three" :  
            rank == 702 ? "Two" :  
            rank == 703 ? "One" :  
            rank == 70
```

```
rank == 3 ? "Queen" :  
    (rank - 3).toString();  
    return new Card(suiteName, rankName); }  
}]  
})  
  
export class AppComponent {  
title = 'app works!';  
constructor(card:Card){  
    this.title = card.toString();  
}  
}
```

15.21 Value Providers

As mentioned earlier, there are three types of Providers: Class Providers, Factory Providers and Value Providers. We have provided code and examples for Class Providers and Factory Providers. Now we will cover Value Providers.

Value Providers simply provide a value of an Object.

15.22 Value Providers – Example

Language is: en

This will be example ‘ch15-ex600’.

- **Step 1 – Build the App using the CLI**

```
ng new ch15-ex600 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch15-ex600  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Edit Class**

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, Injector } from '@angular/core';
@Component({
selector: 'app-root',
template: `

<h1>
  {{title}}
</h1>
`,
styles: [],
providers: [
  {
    provide: 'language',
    useValue: 'en'
  }
]
})
export class AppComponent {
  title: string = "";
  constructor(private injector: Injector) {
    this.title = 'Language is: ' + injector.get('language');
  }
}
```

15.23 Injector API

If you want even more control over creating Dependencies, you can access the Injector object directly. The Injector is a class in the Angular core package. It is a dependency injection container used for instantiating objects and resolving dependencies.

If your classes that you are attempting to resolve and create (using the Injector) have dependencies themselves, the Injector automatically attempts to resolve and create these for you.

You can also use the additional options in the Provider class when you use it.

• Injector – Example 1

```
import { Injector } from '@angular/core';
const injector = Injector.resolveAndCreate([Car, Engine, Tires, Doors]);
const car = injector.get(Car);
```

• Injector – Example 2

```
import { Injector } from '@angular/core';
const injector = Injector.resolveAndCreate(
[
  provide(Car, useClass: Car)),
  provide(Engine, useClass: Engine)),
  provide(Tires, useClass: Tires)),
  provide(Doors, useClass: Doors))
];
);

const car = injector.get(Car);
```


16 Angular 4 and UI Widgets

16.1 Introduction

Angular is the core of many new JavaScript apps. However, you need to couple Angular 4 with a front-end UI framework, such as Bootstrap or Material. I am going to cover Bootstrap in this chapter as it is currently the more common of the two. However, don't let this deter you from Material, I am sure that it's excellent.

16.2 How to Use a UI Widget Library with Angular

You can do this in two ways:

1. The pre-angular way, use html markup and JavaScript in the normal manner.
2. The angular way, with custom markup directives. You utilize a 3rd party module of custom components and directives that generate UI Widget html markup like #1.

16.3 Pre-Angular Way

You can create components using html and JavaScript that style components in the same manner as you would in JQuery or another earlier JavaScript library.

• What's HTML Markup?

HTML is the most common document format of the web. It stands for Hyper Text Markup Language. A markup language is a system for [annotating](#) a [document](#) in a way that is [syntactically distinguishable](#) from the text. Some markup languages, such as HTML, have pre-defined [presentation semantics](#)—meaning that their specification prescribes how to display the structured data. Html Markup tells the computer how to display something. In Angular we write dynamic user interfaces. Angular components use Html Markup to tell the computer how to display things.

• Where Is the Markup?

It's in the template. The template is specified in the `@Component` annotation of the Component. It is also sometimes specified in the `@View` annotation also (more on that later).

16.4 The Angular Way

You can create components using a module of pre-built and styled Angular components and directives, delivered as a module so you can re-use them.

• What is a Module of Components and Directives?

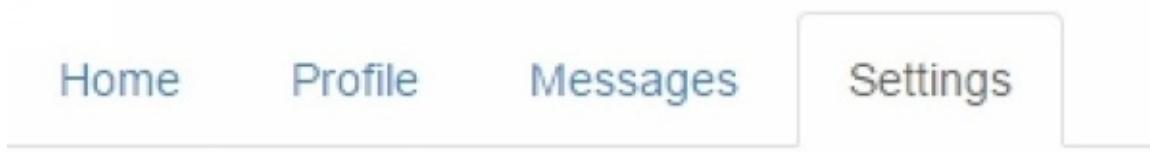
It's a module of Component objects (just like the ones you wrote earlier) and Directives that enable you to use tags to create a Bootstrap UI. They require you to use someone else's code but they save you time by providing prebuilt components and the directives with which to use them.

In the book (including the example project) I am using the ng2-bootstrap module (<http://valor-software.github.io/ng2-bootstrap/>). This is a bootstrap implementation for Angular 4.

We will go over Directives later in Chapter 'Introducing Directives'.

16.5 Pre-Angular (Html & Css & JavaScript) vs Angular Module

Here is a common UI element: a tab. We will write the HTML markup for the same tab with and without the NG-Bootstrap module.



● Pre-Angular (HTML & Css & JavaScript)

```
<div class="tabbable tabs-left" style="margin-top: 100px;"> <ul class="nav nav-tabs">
  <li class="active"><a href="#pane1" data-toggle="tab" rel="popover" id="tab">Homee</a></li> <li><a href="#pane2" data-toggle="tab"
title="blah blah" id="tab1">Profile</a></li> <li><a href="#pane3" data-toggle="tab" id="tab2">Messages</a></li> <li><a href="#pane4" data-
toggle="tab">Settings</a></li> </ul>
<div class="tab-content">
  <div id="pane1" class="tab-pane active">...</div> <div id="pane2" class="tab-pane">...</div> <div id="pane3" class="tab-pane">...</div>
<div id="pane4" class="tab-pane">...</div> </div>
</div>
```

● Angular with NG-Bootstrap Module

```
<ngb-tabset>
<ngb-tab title="Home">
  <ng-template ngbTabContent>
    ...
  </ng-template>
</ngb-tab>
<ngb-tab title="Profile">
  <ng-template ngbTabContent>
    ...
  </ng-template>
</ngb-tab>
<ngb-tab title="Messages">
  <ng-template ngbTabContent>
    ...
  </ng-template>
</ngb-tab>
<ngb-tab title="Settings">
  <ng-template ngbTabContent>
    ...
  </ng-template>
</ngb-tab>
</ngb-tabset>
```

● Conclusion

These modules can make the code smaller and have meaning, for example calling tabs ‘ngb-tabs’ instead of ‘div’s.

16.6 Bootstrap

Bootstrap is a free and open-source collection of tools for creating websites and web applications. It contains HTML- and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. It aims to ease the development of dynamic websites and web applications. Bootstrap is compatible with the latest versions of the Google Chrome, Firefox, Internet Explorer, Opera, and Safari browsers, although some of these browsers are not supported on all platforms.

Since version 2.0 it also supports responsive web design. This means the layout of web pages adjusts dynamically, taking into account the characteristics of the device used (desktop, tablet, mobile phone). Starting with version 3.0, Bootstrap adopted a mobile-first design philosophy, emphasizing responsive design by default.

To see more, go to: <http://getbootstrap.com>.

• Bootstrap Uses A Grid To Provide Responsive UIs

Bootstrap provides a grid system to allow developers (those who may lack skills in responsive-design) to write code that works on mobile, tablet and desktop.

The default Bootstrap grid system utilizes 12 columns, making for a 940px wide container without responsive features enabled.

With the responsive CSS file added, the grid adapts to be 724px and 1170px wide depending on your viewport. Below 767px viewports, the columns become fluid and stack vertically.

• A Webpage Using Bootstrap



16.7 How to Start Using Ng Bootstrap

Ng Bootstrap is an angular version of the Bootstrap library and you can use it to quickly build apps with Bootstrap widgets. The source code is here: <https://github.com/ng-bootstrap/ng-bootstrap> and the demos are here: <https://ng-bootstrap.github.io/-/components/accordion>.

1. Build the app using the CLI in the usual manner.
2. Use Node Package Manager to install both ng-bootstrap and bootstrap modules.

```
npm install --save @ng-bootstrap/ng-bootstrap bootstrap
```

3. Tell the CLI project to use the styles in the bootstrap css file. Edit ‘.angular.json’ and add the following entry under styles:

```
"./node_modules/bootstrap/dist/css/bootstrap.css",
```

4. Edit your module file (e.g. app.module.ts) and specify the NgbModule as an import. This will make the code in NgbModule available in this Angular module.

```
imports: [  
  NgbModule.forRoot(),  
  BrowserModule  
,
```

16.8 Bootstrap - Example Code

Please select your pizza:

Hawaiian Peperoni Everything

Your Selection: Hawaiian

This component allows the user to select the pizza using a group of buttons that act like a group of radio buttons. This is example 'ch16-ex100'.

- **Step 1 – Build the App using the CLI**

```
ng new ch16-ex100 --inline-template --inline-style
```

- **Step 2 – Install Ng Bootstrap & Bootstrap**

```
cd ch16-ex100  
npm install --save @ng-bootstrap/ng-bootstrap bootstrap
```

- **Step 3 – Install Bootstrap Styles into Project**

Edit '.angular.json' and add the following entry under styles:

```
"./node_modules/bootstrap/dist/css/bootstrap.css", The style block should look like this:
```

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

- **Step 4 –Start Ng Serve**

```
ng serve
```

- **Step 5 – Open App**

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- **Step 6 – Edit Module**

Edit 'app.module.ts' and change it to the following:

```
import { BrowserModule } from '@angular/platform-browser'; import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';  
import { AppComponent } from './app.component';  
  
@NgModule({  
  declarations: [  
    AppComponent
```

```
],  
imports: [  
    NgbModule.forRoot(),  
    BrowserModule,  
    FormsModule  
],  
providers: [],  
bootstrap: [AppComponent]  
})  
export class AppModule { }
```

● Step 7 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { NgbModule } from '@ng-bootstrap/ng-bootstrap'; @Component({  
selector: 'app-root',  
template: `  
<div style="padding:10px">  
    <h2>Please select your pizza:</h2> <div [(ngModel)]="model" ngbRadioGroup name="radioBasic"> <label class="btn btn-primary">  
<input type="radio" value="Hawaiian"> Hawaiian </label>  
    <label class="btn btn-primary"> <input type="radio" value="Peperoni"> Peperoni </label>  
    <label class="btn btn-primary"> <input type="radio" value="Everything"> Everything </label>  
    </div>  
    <hr>  
    Your Selection: {{model}}  
</div>  
  
`,  
styles: []  
})  
export class AppComponent {  
    model = 'Hawaiian';  
}
```

● Exercise Complete

Your app should be working at localhost:4200.

16.9 Material

Material is a design language developed by Google. Expanding upon the "card" motifs that debuted in Google Now, Material Design makes more liberal use of grid-based layouts, responsive animations and transitions, padding, and depth effects such as lighting and shadows. Designer Matías Duarte explained that, "unlike real paper, our digital material can expand and reform intelligently. Material has physical surfaces and edges. Seams and shadows provide meaning about what you can touch." Google states that their new design language is based on paper and ink.

The canonical implementation of Material Design for web application user interfaces is called Polymer. It consists of the Polymer library, a shim that provides a Web Components API for browsers that do not implement the standard natively, and an elements catalog, among which is the "paper elements collection" that features visual elements of the Material Design.

To see more, go to: <http://www.material-ui.com>.

• A Webpage Using Material

The screenshot shows a web browser displaying the 'Components' section of the Material-UI documentation. The left sidebar lists various components like Prerequisites, Installation, Usage, Server Rendering, Examples, Customization, Components, Discover More, Resources, GitHub, React, and Material Design. The main content area shows two examples: 'AppBar' and 'Buttons'. The 'AppBar' example shows a toolbar with branding, navigation, search, and actions. The 'Buttons' example shows a row with an IconButton, a title, and a FlatButton. Both examples include code snippets and descriptions.

16.10 How to Start Using Angular Material

Angular Material is an angular version of the Material library and you can use it to quickly build apps with Material components. The source code is here: <https://github.com/jelbourn/material2-app> and the example is here: <https://material2-app.firebaseioapp.com/> .

1. Build the app using the CLI in the usual manner.
2. Use Node Package Manager to install Angular Material, Angular Animation and Hammer (JavaScript library).

```
npm install --save @angular/material  
npm install --save @angular/animations  
npm install --save hammerjs
```

3. Add the icons to your project by including them in the index.html file:

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
```

4. Rename the styles file ‘styles.css’ to ‘styles.scss’ and change it to the following:

```
@import '~@angular/material/prebuilt-themes/deeppurple-amber.css';
```

5. Change the reference to the style file in ‘.angular-cli.json’:

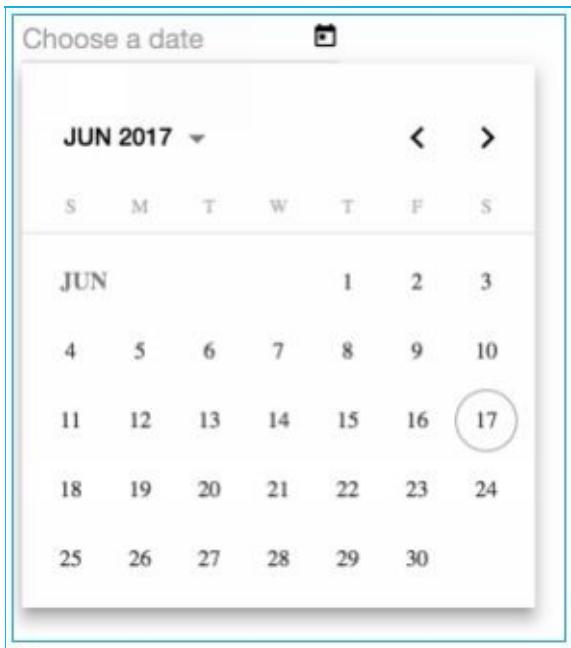
```
"styles": [  
  "styles.scss"  
],
```

6. Edit the CLI-generated module file ‘app.module.ts’ and ensure it imports the widget modules (e.g. MdButtonModule, MdCheckboxModule) , the animations module (BrowserAnimationsModule) and hammerjs:

```
import {MdButtonModule, MdCheckboxModule} from '@angular/material'; import {BrowserAnimationsModule} from '@angular/platform-browser/animations'; import { hammerjs } from 'hammerjs';

@NgModule({  
  ...  
  imports: [MdButtonModule, MdCheckboxModule], [BrowserAnimationsModule]  
  ...  
})
```

16.11 Material – Example Code



This component allows the user to select the date with a material-styled date picker popup. This is example ‘ch16-ex200’.

- **Step 1 – Build the App using the CLI**

```
ng new ch16-ex200 --inline-template --inline-style
```

- **Step 2 – Install Angular Material, Animations & Bootstrap**

```
cd ch16-ex200
npm install --save @angular/material
npm install --save @angular/animations
npm install --save hammerjs
```

- **Step 3- Add the icons to your project by including them in the index.html file:**

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
```

- **Step 4- Rename the styles file ‘styles.css’ to ‘styles.scss’ and change it to the following:**

```
@import '~@angular/material/prebuilt-themes/deeppurple-amber.css';
```

- **Step 5- Change the reference to the style file in ‘.angular-cli.json’:**

```
"styles": [
  "styles.scss"
],
```

● Step 6 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

import { MdInputModule, MdDatepickerModule, MdNativeDateModule } from '@angular/material'; import { BrowserAnimationsModule } from '@angular/platform-browser/animations'; import { hammerjs } from 'hammerjs';

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

● Step 7 – Edit Component Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
<md-input-container>
  <input mdInput [mdDatepicker]="picker" placeholder="Choose a date"> <button mdSuffix [mdDatePickerToggle]="picker"></button>
</md-input-container>
<md-datepicker #picker></md-datepicker>
`,
  styles: []
})
export class AppComponent {
  title = 'app';
}
```

● Step 4 –Start Ng Serve

```
ng serve
```

● Step 5 – Open App

Open web browser and navigate to localhost:4200. You should see the app running.

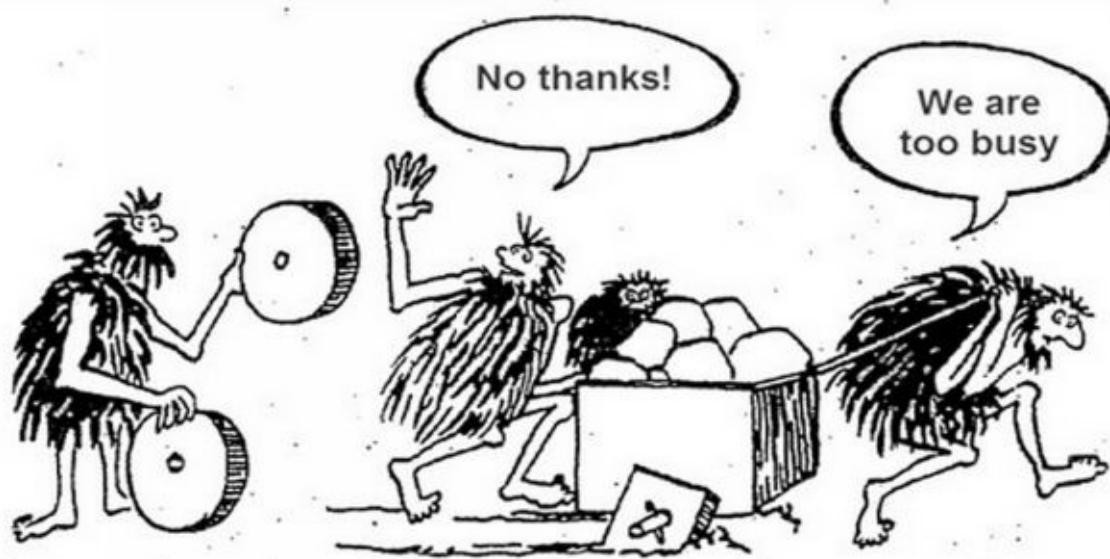
● Exercise Complete

Your app should be working at localhost:4200.

16.12 Conclusion

If you are serious about building a polished Angular app then you should use a pre-built widget library module and use it. It will be easier to write code that is more maintainable.

But you don't have to....



17 Routes and Navigation

17.1 Introduction

• Most Web Apps

In most web applications, users navigate from one page to the next as they perform application tasks. Users can navigate by:

1. Entering a url in the address bar.
2. Following links, clicking buttons etc.
3. Going backward or forward in the browser history.

• Angular Apps

In Angular applications, users can navigate in the same 3 manners above but they are navigating through components (the building-block of Angular apps).

17.2 Router

In Angular we can navigate because we have the Angular Router.

It can interpret a browser URL as an instruction to navigate to a Component and pass optional parameters (which contain information) to the Component to give it contextual information (to help it decide what specific content to present or what it needs to do).

We can bind the Router to links on a page and it will navigate to the appropriate Component when the user clicks a link.

We can navigate imperatively when the user clicks a button, selects from a drop box, or in response to some other stimulus from any source.

And the router logs activity in the browser's history journal so the back and forward buttons work as well.

17.3 Router Routes on The Client Side

● Fragments

Any URL that contains a # character is a fragment URL. The portion of the URL to the left of the # identifies a resource that can be accessed (from the server) and the portion on the right, known as the fragment identifier, specifies a location within the resource (on the client):

www.cnn.com/index.html#section2

In this case, the fragment name is ‘section2’ and it specifies a location in the document ‘index.html’.

The original purpose of a fragment (i.e. the part after the '#') was to allow the user to ‘jump’ to a link on a specified part of the current page, scrolling up or down. Now fragments are often used for client-side navigation, as by their nature they do not invoke a request to pull resources from the server.

● Fragments and Browsers

Newer HTML5 browsers can work with client-side and non-client-side routing for URLs, including those with hashes and those without.

However, some older browsers won’t work with client-side routing for URLs that are not hashed. By ‘hashed’ it means that the client-side part of the url needs to be behind the '#' sign (ie a fragment). So if you are deploying a single page application to production then you probably need to do the following:

1. Turn on ‘hash routing’ on the router. This will make your single page application more compatible with older non-html 5 browsers. When you import the router module in your module, you should do the following:

```
@NgModule({  
  imports: [  
    RouterModule.forRoot(appRoutes, {useHash: true}) ],  
  ...  
})  
export class ...
```

2. Ensure that your ‘404’ page on the server re-routes to the web page (for example ‘index.html’) that contains the single page application. This will return pages to the single page app if for some reason the browser attempts to pull server resources in error.

17.4 Router Module

Before you start using the Component Router you need to know that this module is included in the node package dependencies but is not included by default in the Angular CLI project. Routing is not included in the application module.

However, you can change this by adding the ‘—routing’ parameter to the end of the ng command.
For example: `ng new ch16-ex300 --inline-template --inline-style --routing`

17.5 Router Module – Objects Included

There may seem to be a lot of objects in the table below but they will become more understandable after some exercises.

Object	Type	Description
RouterModule	Module	A separate Angular module that provides the necessary service providers and directives for navigating through application views.
Router		Displays the application component for the active URL. Manages navigation from one component to the next.
Routes		Defines an array of Routes, each mapping a URL path to a component.
Route		Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type.
RouterOutlet	Directive	The directive (<router-outlet>) that marks where the router displays a view.
RouterLink	Directive	The directive for binding a clickable HTML element to a route. Clicking an element with a routerLink directive that is bound to a string or a link parameters array triggers a navigation.
RouterLinkActive	Directive	The directive for adding/removing classes from an HTML element when an associated routerLink contained on or inside the element becomes active/inactive.
ActivatedRoute		A service that is provided to each route component that contains route specific information such as route parameters, static data, resolve data, global query params, and the global fragment.
RouterState		The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.

17.6 Simple Routing - Example

This is a pizza selection component that uses routing to allows the user to click on links to display different types of pizzas, with each type of pizza in its own component. This example also shows the use of route parameters: you can route to the ‘everything’ pizza with a ‘size’ parameter of small or large. This will be example ‘ch17-ex100’.



- **Step 1 – Build the App using the CLI**

```
ng new ch17-ex100 --inline-template --inline-style --routing
```

- **Step 2 – Start Ng Serve**

```
cd ch17-ex100  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Edit Routing Class**

Edit ‘app-routing.module.ts’ and change it to the following:

```
import { NgModule } from '@angular/core'; import { Routes, RouterModule } from '@angular/router'; import { PepperoniComponent, EverythingComponent} from './app.component';  
const routes: Routes = [  
{ path: '',  
  redirectTo: '/pepperoni',  
  pathMatch: 'full'  
},  
{  
  path: 'pepperoni',  
  component: PepperoniComponent  
},  
{
```

```

    path: 'everything/:size',
    component: EverythingComponent
}
];
}

@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
export class AppRoutingModule { }

```

● Step 5 – Edit Components Class

Edit ‘app.component.ts’ and change it to the following:

```

import { Component } from '@angular/core'; import { Router, ActivatedRoute, ActivatedRouteSnapshot } from '@angular/router';
@Component({
selector: 'pepperoni',
template: `
<h2>Pepperoni</h2>
 `,
styles: []
})
export class PepperoniComponent {
}

@Component({
selector: 'everything',
template: `
<h2>Everything</h2>
Size:{ ${size} }
 `,
styles: []
})
export class EverythingComponent {
private size: String = "";
constructor(private route: ActivatedRoute) {
    route.params.subscribe(
      (params: Object) =>
        this.size = params['size']);
}
}

@Component({
selector: 'app-root',
template: `
<h1>
Pizzas
</h1>
<a [routerLink]=["'pepperoni'"]>Pepperoni</a> <a [routerLink]=["'everything','small'"]>Everything Small</a> <a [routerLink]=["'everything','large'"]>Everything Large</a> <router-outlet></router-outlet> `,
styles: []
})
export class AppComponent {
title = 'app';
}

```

● Step 6 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

import { AppRoutingModule } from './app-routing.module'; import { AppComponent, PepperoniComponent, EverythingComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent,
    PepperoniComponent,
    EverythingComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

The file ‘app-routing.module.ts’ was generated by the CLI. It defines a module AppRoutingModule just for the routing. This module contains a data structure that sets up urls with accompanying components. Note how the first url maps the default url to another. Note how the path for the EverythingComponent specifies a ‘size’ parameter.

```
{
  path: 'everything/:size',
  component: EverythingComponent
}
```

The file ‘app.component.ts’ contains all the components. It uses RouterLink directives to modify links to work with the Angular router. The EverythingComponent is used to display the everything pizza and it can accept a ‘size’ parameter. Note how it subscribes to the route parameters object to receive parameter updates. This is necessary in case the user switches from one size of everything to another, updating the size parameter.

```
constructor(private route: ActivatedRoute){
  route.params.subscribe(
    (params: Object) =>
    this.size = params['size']);
}
```

The file ‘app.module.ts’ declares all the components so that they can be accessed in the app module. It also imports the AppRoutingModule that we setup in ‘app.routing.module.ts’.

17.7 Nested Routing

Nested Routing means the ability to route and navigate to Sub Components that are inside other Components which that are navigated to. This is definitely possible in Angular, as you can see in the following example.

17.8 Nested RoutingUrls

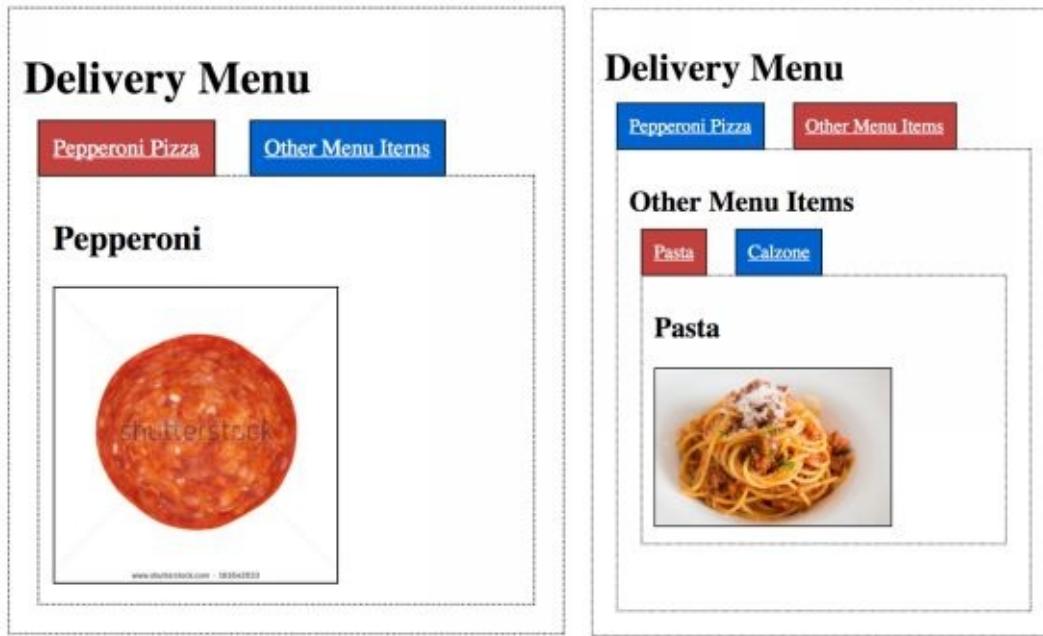
Nested routing link urls have more than one ‘level’ because now there is a hierarchy of routes and their children, rather than just routes.

- Example ‘ch17-ex100’ urls versus ‘ch17-ex200’ urls:

Ch17-ex100	Ch17-ex200
/pepperoni	/pepperoni
/everything	/other/pasta
	/other/canzone

17.9 Nested Routing – Example

This is another pizza selection component that uses routing. However, this time it uses a nested route for the ‘other’ menu and component. When you click on the ‘other’ link, you can select from a submenu of ‘pasta’ or ‘calzone’. The display of these menu items is handled by nested routing. This will be example ‘ch17-ex200’.



- Step 1 – Build the App using the CLI

```
ng new ch17-ex200 --inline-template --inline-style --routing
```

- Step 2 – Start Ng Serve

```
cd ch17-ex200  
ng serve
```

- Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- Step 4 – Edit Routing Class

Edit ‘app-routing.module.ts’ and change it to the following:

```
import { NgModule } from '@angular/core'; import { Routes, RouterModule } from '@angular/router'; import { PepperoniComponent } from './app.component';  
  
import { OtherComponent } from './app.other-component'; import { NestedPastaComponent, NestedCalzoneComponent } from './app.other-component';  
const routes: Routes = [  
{  
  path: '',  
  redirectTo: '/pepperoni',  
  pathMatch: 'full'  
},  
{
```

```

    path: 'pepperoni',
    component: PepperoniComponent
},
{
  path: 'other',
  component: OtherComponent,
  children: [
  {
    path: '',
    redirectTo: 'pasta',
    pathMatch: 'full'
  },
  {
    path: 'pasta',
    component: NestedPastaComponent
  },
  {
    path: 'calzone',
    component: NestedCalzoneComponent
  }
]
}
];
}

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

● Step 5 – Edit Components Class

Edit ‘app.component.ts’ and change it to the following:

```

import { Component } from '@angular/core'; import { Router, ActivatedRoute, ActivatedRouteSnapshot } from '@angular/router';
@Component({
  selector: 'pepperoni',
  template: `
    <div>
      <h2>Pepperoni</h2>
       </div>
  `,
  styles: []
})
export class PepperoniComponent {
```



```

@Component({
  selector: 'app-root',
  template: `
    <div>
      <h1>
        Delivery Menu
      </h1>
      <a [routerLink]=["'pepperoni'"] routerLinkActive="router-link-active">Pepperoni Pizza</a> <a [routerLink]=["'other'"]>
        routerLinkActive="router-link-active">Other Menu Items</a> <router-outlet></router-outlet> </div>
    `,
  styles: []
})
```

```
export class AppComponent {  
  title = 'app';  
}
```

● Step 7 – Edit Other Component

Edit ‘app.other-component.ts’ and change it to the following:

```
import { Component } from '@angular/core';  
@Component({  
  selector: 'pasta',  
  template: `  
    <div>  
      <h2>Pasta</h2>  
       </div>  
  `,  
  styles: []  
})  
export class NestedPastaComponent {  
}  
  
@Component({  
  selector: 'calzone',  
  template: `  
    <div>  
      <h2>Calzone</h2>  
       </div>  
  `,  
  styles: []  
})  
  
export class NestedCalzoneComponent {  
}  
  
@Component({  
  selector: 'other',  
  template: `  
    <div>  
      <h2>Other Menu Items</h2> <a [routerLink]=["'pasta'" routerLinkActive="router-link-active">Pasta</a> <a [routerLink]=["'calzone'"  
routerLinkActive="router-link-active">Calzone</a> <router-outlet></router-outlet> <br/>  
      <br/>  
    </div>  
  `,  
  styles: []  
})  
export class OtherComponent {  
}
```

● Step 8 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

```
import { BrowserModule } from '@angular/platform-browser'; import { NgModule } from '@angular/core';  
  
import { AppRoutingModule } from './app-routing.module'; import { AppComponent, PepperoniComponent } from './app.component'; import  
{ OtherComponent, NestedCalzoneComponent, NestedPastaComponent } from './app.other-component';  
@NgModule({
```

```

declarations: [
  AppComponent,
  PepperoniComponent,
  OtherComponent,
  NestedCalzoneComponent,
  NestedPastaComponent
],
imports: [
  BrowserModule,
  AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

● Step 9 – Edit Styles

Edit ‘app.module.ts’ and change it to the following:

```

/* You can add global styles to this file, and also import other style files */
img {
  width: 200px;
  border: 1px solid #000000;
}
a {
  background-color: #0066CC;
  color: #ffffff;
  border: 1px solid #000000;
  padding: 10px;
  margin: 10px;
}
.router-link-active {
  background-color: #C14242;
}
div {
  border: 1px dotted #000000;
  margin: 10px;
  padding: 10px;
}

```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

The file ‘app-routing.module.ts’ was generated by the CLI. It defines a module AppRoutingModule just for the routing. This module contains a data structure that sets up urls with accompanying components. Note that this time the data structure contains child routing using the ‘children’ property:

```
{
  path: 'other',
  component: OtherComponent,
  children: [
    {
      path: '',
      redirectTo: 'pasta',
      pathMatch: 'full'
    },
    {
      {
        path: 'calzone',
        component: NestedCalzoneComponent
      }
    }
  ]
}
```

```
path: 'pasta',
component: NestedPastaComponent
},
{
path: 'calzone',
component: NestedCalzoneComponent
}
]
```

The file ‘app.component.ts’ is used to define the app component and the ‘pepperoni’ component (not nested). It also contains the non-nested router links and the router outlet, into which the non-nested components are injected.

The file ‘app.other-component.ts’ is used to define the ‘other’ component (not nested) and the ‘pasta’ and ‘calzone’ nested components. The ‘other’ component contains the nested router links and the router outlet, into which the nested components are injected.

The file ‘app.module.ts’ declares all the components so that they can be accessed in the app module. It also imports the AppRoutingModule that we setup in ‘app.routing.module.ts’.

The file ‘styles.css’ declares some styles (badly, I admit) that are used for the links and for a tabbed effect. Notice how the ‘routerLinkActive’ style is setup on the router links to highlight the currently active links (this applies to nested and non-nested links).

Router Links:

```
<a [routerLink]=["'pepperoni'"] routerLinkActive="router-link-active">Pepperoni Pizza</a> Styling for Active  
Router Links:
```

```
.router-link-active {
  background-color: #C14242;
}
```

17.10 Route Configuration

● Introduction

Angular applications route by using a single instance of the Router service. When navigation occurs, the Router attempts to resolve a Route for the new location. To resolve routes, routes must be configured for the Router. The routes are configured as an array of route objects. Each route object needs a path (to resolve it) and (usually) a component (what will be displayed in the router outlet for the resolved route). Route objects can have more properties also, see below for more details.

● Redirection

You can configure route paths that redirect to other paths. For example, the code below redirects an empty route to the ‘pepperoni’ route. In the case of an empty URL we also need to add the pathMatch: 'full' property so Angular knows it should be matching exactly the empty string and not partially the empty string.

```
const routes: Routes = [
{
  path: '',
  redirectTo: '/pepperoni',
  pathMatch: 'full'
},
...
];
```

● ‘Catch All’ Route

We can also add a catch all route by using the path **, if the URL doesn’t match any of the other routes it will match this route.

```
const routes:Routes = [
...
{path: '**', component: CatchAllComponent}
];
```

17.11 Route Configuration – Data

When you configure your routes, you configure them using an array of route objects. Each route object can have a ‘data’ property that contains other properties which can be extracted later by the target component for that route.

17.12 Route Configuration – Data – Example

The code below sets up a route with data, including a message for the ‘not found’ path: { path: '500', component: ErrorPageComponent, data: {message: 'Unexpected Server Error'} }

The code below accesses that data so that it can be used to show a message: Either:

```
this.errorMessage = this.snapshot.data['message']; or:
```

```
this.route.data.subscribe(  
(data: Data) => { this.errorMessage = data['message']; }  
);
```

Note that this allows you to re-use the same component for different purposes with different data. For example, you could also setup a route for path ‘401’ which would re-use the ‘ErrorPageComponent’ but this time with the message ‘Unauthorized’.

17.13 Route Path Parameters

You can pass data parameters to components in the routes as part of the URL path, for example: customer/123.

When you write the code for the component that receives the parameter, you have two different implementations to choose from:

1. You can read the parameter from the route ‘snapshot’ (which is a one-off snapshot of the route). This is useful when you are routing to a child component inside a parent component once only and this parameter never changes.

```
constructor(route: ActivatedRoute) {  
  this.customerId = route.snapshot.paramMap.get('id'); }
```

2. You can read the parameter by subscribing to an observable parameter map. This is useful when you are routing to a child component inside a parent component and the child component may ‘re-route’ (being passed a new parameter) when something changes.

```
constructor(route: ActivatedRoute) {  
  route.paramMap.subscribe(  
    params => this.customerId = params.get('id') );  
}
```

17.14 Route Path Parameters – Example

This has already been done in example ‘ch17-ex100’.

17.15 Route Query Parameters

You can pass data parameters to components in the routes using query strings, for example:
customer?id=123.

This works in a similar manner to path parameters

When you write the code for the component that receives the parameter, you have two different implementations to choose from:

1. You can read the query parameter from the route ‘snapshot’ (which is a one-off snapshot of the route). This is useful when you are routing to a child component inside a parent component once only and this parameter never changes.

```
constructor(route: ActivatedRoute) {  
  this.customerId = route.snapshot.queryParams['id'];  
}
```

2. You can read the parameter by subscribing to an observable query parameter map. This is useful when you are routing to a child component inside a parent component and the child component may ‘re-route’ (being passed a new query parameter) when something changes.

```
constructor(route: ActivatedRoute) {  
  route.queryParams.subscribe(  
    params => this.customerId = params.get('id') );  
}
```

17.16 Route Query Parameters – Example

This is a component that shows a list of customers at the top, with details of the selected customer below. This will be example ‘ch17-ex300’.

The screenshot shows a user interface for managing customers. On the left, under 'Customer List', there is a list of names: Mark, Peter, Jill, and Brian. Below this, under 'Customer Detail', there is a card for the selected customer, Peter, showing his name, address (Belvue, CA), and balance (\$5000).

- **Step 1 – Build the App using the CLI**

```
ng new ch17-ex300 --inline-template --inline-style --routing
```

- **Step 2 – Start Ng Serve**

```
cd ch17-ex300  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Create Customer Class**

Create ‘customer.ts’:

```
export class Customer {  
    private _id: number;  
    private _name: string;  
    private _city: string;  
    private _state: string;  
    private _balance: number;  
  
    constructor(id: number, name: string, city: string, state: string, balance: number) {  
        this._id = id;  
        this._name = name;  
        this._city = city;  
        this._state = state;  
        this._balance = balance;  
    }  
  
    get id(): number {  
        return this._id;  
    }
```

```

}
get name(): string {
return this._name;
}
get city(): string {
return this._city;
}
get state(): string {
return this._state;
}
get balance(): number {
return this._balance;
}
}

```

● Step 5 – Create Customer Service Class

Create ‘customerService.ts’.

```

import { Injectable } from '@angular/core'; import { Customer } from './customer';

@Injectable()
export class CustomerService {
private _customers: Array<Customer> = [
    new Customer(1, 'Mark', 'Atlanta', 'GA', 12000), new Customer(2, 'Peter', 'Belvue', 'CA', 5000), new Customer(3, 'Jill', 'Colombia', 'SC', 2000), new Customer(4, 'Brian', 'Augusta', 'GA', 2000) ];

get customers() {
    return this._customers;
}

getCustomerById(id: number){
    for (let i=0, ii=this._customers.length; i<ii; i++){
        const customer = this._customers[i]; if (customer.id == id){
            return customer;
        }
    }
    return null;
}
}

```

● Step 6 – Edit App Routing Module

Edit ‘app-routing.module.ts’ and change it to the following:

```

import { NgModule } from '@angular/core'; import { Routes, RouterModule } from '@angular/router'; import { DetailComponent, PleaseSelectComponent } from './app.component';
const routes: Routes = [
{
    path: '',
    component: PleaseSelectComponent,
    children: []
},
{
    path: 'detail',
    component: DetailComponent,
    children: []
}
]

```

```
];
@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
export class AppRoutingModule { }
```

● Step 7 – Edit App Component

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { ActivatedRoute } from '@angular/router';
import { CustomerService } from './customerService';
import { Customer } from './customer';

@Component({
selector: 'pleaseSelect',
template: `
<div>
<h2>Please make a selection.</h2> </div>
`,
styles: ['div { background-color: #FFFFFF; padding: 10px; border: 1px solid #000000 }']
})
export class PleaseSelectComponent {
}

@Component({
selector: 'detail',
template: `
<div>
<h2>Customer Detail {{id}}</h2> <p>{{customer.name}}</p>
<p>{{customer.city}}, {{customer.state}}</p> <p>Balance: &#36;{{customer.balance}}</p> </div>
`,
styles: ['div { background-color: #FFE4E1 }']
})
export class DetailComponent {
customer: Customer;
constructor(
  private customerService: CustomerService, private route: ActivatedRoute) {
  route.queryParams.subscribe(
    (queryParams: Object) =>
      this.customer = customerService.getCustomerById(queryParams['id']));
}
}

@Component({
selector: 'app-root',
template: `
<div>
<h1>
Customer List
</h1>
<p *ngFor="let customer of _customerService.customers"> <a [routerLink]=["'detail'"] [queryParams]={{id: customer.id}}"
  routerLinkActive="active">{{customer.name}}</a> </p>
</div>
<router-outlet></router-outlet> `,
styles: ['div { background-color: #faebd7 }',]
})
```

```
export class AppComponent {  
constructor(private _customerService: CustomerService){  
}  
}
```

● Step 8 – Edit App Module

Edit ‘app.module.ts’ and change it to the following:

```
import { BrowserModule } from '@angular/platform-browser'; import { NgModule } from '@angular/core';  
  
import { AppRoutingModule } from './app-routing.module'; import { AppComponent, DetailComponent, PleaseSelectComponent } from  
'./app.component'; import { CustomerService } from './customerService';  
import { Customer } from './customer';  
  
@NgModule({  
declarations: [  
    AppComponent,  
    DetailComponent,  
    PleaseSelectComponent  
],  
imports: [  
    BrowserModule,  
    AppRoutingModule  
],  
providers: [CustomerService],  
bootstrap: [AppComponent]  
})  
export class AppModule { }
```

● Step 9 – Styles

Edit ‘styles.css’ and change it to the following:

```
div {  
    padding: 10px; border: 1px solid #000000; }  
h1,h2 {  
    margin: 0px;  
}  
.active {  
    font-weight: bold;  
}  
.active::before {  
    content: ">>> ";  
}  
.active::after {  
    content: "<<<";  
}
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

The file ‘customer.ts’ sets up the customer class.

The file ‘customerService.ts’ is a service injected into the app component and the ‘Detail’ component. It contains a list of customers, along with methods to access the customer data.

The file ‘app-routing.module.ts’ sets up the route for the ‘Please Select’ component and the ‘Detail’ component.

The file ‘app.component.ts’ sets up the components. Note how it uses a different syntax for specifying query parameters to a router link: `<a [routerLink]=“[‘detail’]” [queryParams]=“{id: customer.id}” routerLinkActive=“active”>{{customer.name}}` The file ‘app.module.ts’ declares the components used, imports the app routing module and sets up the CustomerService class as a provider for the customer dependency injection in the Detail and App Components.

The file ‘styles.css’ is used to setup common styles for the h1, h2 and div tags.

17.17 Router - Imperative Navigation

So far, we have written code that provides the user with the ability to click on a link to navigate.

Imperative navigation is different. This is not generating links, this is simply telling the router to go somewhere, performing navigation in your code. Navigation is an asynchronous event, it does not lock the code until completed. The imperative navigation methods below return a Promise object when completed, which is a callback for success or failure.

To use imperative navigation, you first need to inject the router into your class. Obviously to use this you should inject the router into your class using Constructor Injection.

• **Callbacks**

Both navigation methods below return a Promise, which enables the user to add two callback methods to handle the navigation result: the first one for a success handler, the second one for an error handler. There is example of this in the example below.

• **NavigationExtras**

Both navigation methods have the ability to accept an additional parameter for a NavigationExtras object. This object allows you to pass additional information to the router to further specify the desired route.

17.18 Router - Imperative Navigation Methods

● **Router.navigate**

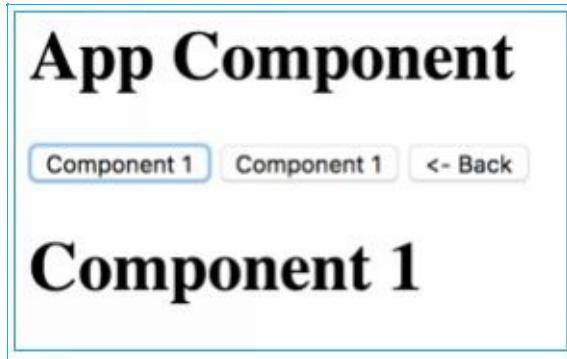
Navigate to a component relatively (to the current route) or absolutely based on an array of commands or route elements. Returns a promise that resolves when navigation is complete. This uses the Link DSL as specified above in Router Link DSL Format. It's basically the same as clicking on a Router Link.

● **Router.navigateByUrl**

Navigate to a complete absolute URL string. Returns a promise that resolves when navigation is complete. It's preferred to navigate with navigate instead of this method, since URLs are more brittle. If the given URL begins with a ‘/’, router will navigate absolutely. If the given URL does not begin with ‘/’, the router will navigate relative to this component.

17.19 Router - Imperative Navigation – Example

This component allows the user to navigate between components and also go back. It also logs when navigation is completed. This is example ‘ch17-ex400’.



- **Step 1 – Build the App using the CLI**

```
ng new ch17-ex400 --inline-template --inline-style --routing
```

- **Step 2 – Start Ng Serve**

```
cd ch17-ex400  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Edit Routing Module**

Edit ‘app-routing.module.ts’ and change it to the following:

```
import { NgModule } from '@angular/core'; import { Routes, RouterModule } from '@angular/router'; import { AppComponent, Component1, Component2 } from './app.component';  
const routes: Routes = [  
{  
    path: 'component1',  
    component: Component1  
},  
{  
    path: 'component2',  
    component: Component2  
},  
{  
    path: '**',  
    component: Component1  
},  
];  
  
@NgModule({  
imports: [RouterModule.forRoot(routes)],  
exports: [RouterModule]  
})  
export class AppRoutingModule { }
```

● Step 5 – Edit Components Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { Router } from '@angular/router';
import { Location } from '@angular/common';
@Component({
  selector: 'component1',
  template: `
    <h1>
      {{title}}
    </h1>
    <router-outlet></router-outlet> `,
  styles: []
})
export class Component1 {
  title = 'Component 1';
}

@Component({
  selector: 'component2',
  template: `
    <h1>
      {{title}}
    </h1>
    <router-outlet></router-outlet> `,
  styles: []
})
export class Component2 {
  title = 'Component 2';
}

@Component({
  selector: 'app-root',
  template: `
    <h1>
      {{title}}
    </h1>
    <button (click)="component1()">Component 1</button> <button (click)="component2()">Component 2</button> <button
      (click)="back()">- Back</button> <router-outlet></router-outlet> `,
  styles: []
})
export class AppComponent {
  title = 'App Component';
  constructor(private router: Router, private location: Location){}
  component1(){
    this.router.navigate(['component1']).then(result => { console.log("navigation result: " + result)}); }
  component2(){
    this.router.navigateByUrl("/component2"); }
  back(){
    this.location.back(); }
}
```

● Step 6 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

import { AppRoutingModule } from './app-routing.module'; import { AppComponent, Component1, Component2 } from './app.component';
@NgModule({
declarations: [
  AppComponent,
  Component1,
  Component2
],
imports: [
  BrowserModule,
  AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

• Exercise Complete

Your app should be working at localhost:4200. Note the following:

The App Component injects the router and the location.

It contains code to navigate imperatively when the user clicks on a button.

It contains a callback that is fired when navigation is completed.

It also contains code in the location for the back button.

17.20 Router – Extracting Data

You can extract the following information out of the Router object that is injected into your class.

Property	Description
errorHandler	Error handler that is invoked when a navigation errors.
navigated	Indicates if at least one navigation happened.
urlHandlingStrategy	Url handling strategy.
routeReuseStrategy	Route reuse strategy.
routerState	Current router state.
url	Current url.
events	An observable of router events. Allows you to add callbacks to router events.

17.21 Dynamically Changing Route Configuration

You normally define the router routes using a configuration object and this doesn't change. However you can reload a different configuration object into the router whenever you wish using the 'resetConfig' method. This would be very useful if you wanted to load the routes from the server or other data source.

17.22 Route Guards

Routes enable the user to navigate through the application. Sometimes the user needs to do something before being allowed access to a certain part of the application, for example log in. Route Guards can be used to control access to certain routes.

There are various types of route guards:

CanActivate.

- Can the user navigate to a route?
- In this class, you can inject the router. This is useful to navigate the user away to another resource if the user is not allowed to navigate to a route?

CanDeactivate.

- Can the user move away from a route? Useful fr prompting to save changes.

17.23 Route Guards – Example

This component will display menu links. Some of the links will only work once the user has logged in. This will be example ‘ch17-ex500’.

• Step 1 – Build the App using the CLI

```
ng new ch17-ex500 --inline-template --inline-style --routing
```

• Step 2 – Start Ng Serve

```
cd ch17-ex500  
ng serve
```

• Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

• Step 4 – Edit Activating Service

Edit ‘activate.service.ts’ and change it to the following:

```
import { Injectable } from '@angular/core'; import { UserService } from './user.service';  
import { CanActivate } from '@angular/router';  
  
@Injectable()  
export class ActivateService implements CanActivate{  
  constructor(private _userService: UserService){}  
  canActivate() {  
    return this._userService.authenticated; }  
}
```

• Step 5 – Edit Routing Module

Edit ‘app.routing.module.ts’ and change it to the following:

```
import { NgModule } from '@angular/core'; import { Routes, RouterModule } from '@angular/router'; import { AuthenticatedComponent,  
NonAuthenticatedComponent } from './app.component'; import { UserService } from './user.service';  
import { ActivateService } from './activate.service';  
  
const routes: Routes = [  
{  
  path: 'authenticated',  
  component: AuthenticatedComponent,  
  canActivate: [  
    ActivateService  
  ]  
},  
{  
  path: '**',  
  component: NonAuthenticatedComponent }  
];
```

```

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  providers: [UserService, ActivateService]
})
export class AppRoutingModule { }

```

● Step 6 – Edit Components

Edit ‘app.component.ts’ and change it to the following:

```

import { Component, ViewChild } from '@angular/core'; import { UserService } from './user.service';

@Component({
  selector: 'non-authenticated-component',
  template: `
    <div>
      <h2>Non-authenticated</h2> <p>This component can be accessed without authentication.</p> </div>
    `,
  styles: []
})
export class NonAuthenticatedComponent {
}

@Component({
  selector: 'authenticated-component',
  template: `
    <div>
      <h2>Authenticated</h2>
      <p>This component cannot be accessed without authentication.</p> </div>
    `,
  styles: []
})
export class AuthenticatedComponent {
}

@Component({
  selector: 'app-root',
  template: `
    <span *ngIf="!_userService.authenticated"> User:<input type="input" #name /> Password:<input type="input" #password /> <input
    type="button" (click)="login()" value="Login" />
    </span>
    <hr/>
    Authenticated:{{ _userService.authenticated}}
    <hr/>
    <a [routerLink]="['non-authenticated']">Non-Authenticated</a> <a [routerLink]="['authenticated']">Authenticated</a> <router-outlet>
    </router-outlet> `,
    styles: []
})
export class AppComponent {
  loggedIn: boolean = false;
  @ViewChild('name') name;
  @ViewChild('password') password;
  constructor(private _userService: UserService){}
  login(){
    this._userService.authenticate(
      this.name.nativeElement.value,
      this.password.nativeElement.value);
  }
}

```

● Step 7 – Edit App Module

Edit ‘app.module.ts’ and change it to the following:

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

import { AppRoutingModule } from './app-routing.module'; import { AppComponent, AuthenticatedComponent, NonAuthenticatedComponent } from './app.component'; import { UserService } from './user.service';

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

● Step 8 – Edit User Service

Edit ‘user.service.ts’ and change it to the following:

```
import { Injectable } from '@angular/core';
@Injectable()
export class UserService {
  private _authenticated: boolean = false; public get authenticated(): boolean{
  return this._authenticated;
}
public set authenticated(value: boolean){
this._authenticated = value;
}
public authenticate(name, password){
if ((name === 'user') && (password === 'password')){
this._authenticated = true;
}
}
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

The service ‘activate.service.ts’ is a route guard that allows or disallows a route from being activated. The ‘canActivate’ method is invoked – true allows the activation, false does not allow the activation. This service is injected into the routing module so it can be used in the route configuration.

The service ‘user.service.ts’ is a service that tracks the state of the user – if he or she is

authenticated or not. This service is injected into the service ‘activate.service.ts’ and the app component.

18 Observers, Reactive Programming & RxJS

18.1 Introduction

Angular includes RxJS.

Reactive Extensions for JavaScript (RxJS) is a reactive streams library that allows you to work with asynchronous data streams. The project is actively developed by Microsoft in collaboration with a community of open source developers.

The sole purpose of this chapter is to introduce RxJS basic concepts and cover some of the library's functionality. We will cover using RxJS and Angular together in another chapter. This chapter will be long enough without any Angular

18.2 Reactive Extensions

In computing, reactive programming is a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow. Reactive extensions code is available on almost every computing platform, not just JavaScript and its purpose is to bring the capability for reactive programming to the computing platform.

18.3 RxJS Libraries

These are a set of libraries to compose asynchronous and event-based reactive programs using observable collections in JavaScript.

18.4 What Are Asynchronous Data Streams?

Asynchronous	In JavaScript means we can call a function and register a <i>callback</i> to be notified when results are available, so we can continue with execution and avoid the Web Page from being unresponsive. This is used for ajax calls, DOM-events, Promises, WebWorkers and WebSockets.
Data	Raw information in the form of JavaScript data types as: Number, String, Objects (Arrays, Sets, Maps).
Streams	Sequences of data made available over time. As an example, opposed to Arrays you don't need all the information to be present in order to start using them.

18.5 Examples of Asynchronous Data Streams

Things that you are watching.

Stock quotes.

Tweets.

Computer events, for example mouse clicks.

Web service requests.

18.6 Observable Sequences (Observables)

In RxJS, you represent Asynchronous Data Streams using Observable Sequences or also just called Observables. So, you could watch stock quotes or mouse clicks using Observables. Observables are flexible and can be used using push or pull patterns.

● Push

When using the push pattern, we subscribe to the source stream and react to new data as soon as is made available (emitted). You can listen to a stream and react accordingly.

● Pull

When using the pull pattern, we are using the same operations but synchronously. This happens when using Arrays, Generators or Iterables.

● Operators

Because observable sequences are data streams, you can query them using Operators implemented by the Observable type.

Things you can do with Observable Operators (just a few of many):

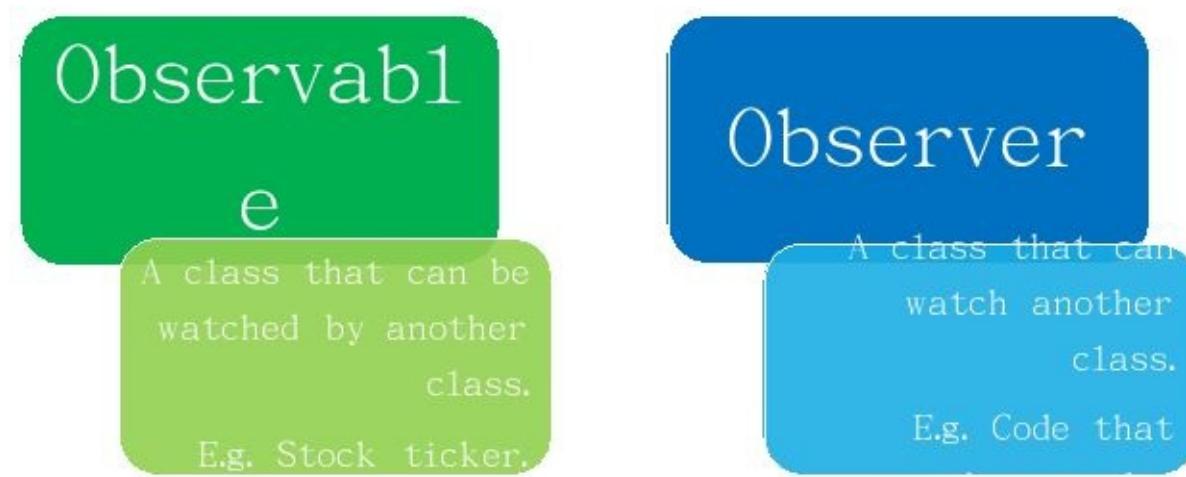
Filter – filter out stock changes for stocks you don't own.

Aggregate – get all the typing in the first 5 seconds.

Perform time-based operations on multiple events.

18.7 Observers

If Observables are things that can be watched, Observers are the things that watch them.



Observers are classes that can respond to events (things happening).

To respond they must implement the following methods:

onNext	An Observable calls this method whenever the Observable emits an item. This method takes as a parameter the item emitted by the Observable.
onError	An Observable calls this method to indicate that it has failed to generate the expected data or has encountered some other error. This stops the Observable and it will not make further calls to onNext or onCompleted. The onError method takes as its parameter an indication of what caused the error.
onCompleted	An Observable calls this method after it has called onNext for the final time, if it has not encountered any errors.

18.8 Observers - Example Code

In the code below we create an Observable with two events. We then Observe it. If you look at the

```
Next: event1,event2  
Completed
```

console you will see the following:

- **Step 1 – Build the App using the CLI**

```
ng new ch18-ex100 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch18-ex100  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Edit Class**

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import * as Rx from 'rxjs';  
@Component({  
  selector: 'app-root',  
  template: `  
      
  `,  
  styles: []  
})  
export class AppComponent {  
  constructor(){  
  
    const array: Array<string> = ['event1', 'event2']; const observable: Rx.Observable<string[]> = Rx.Observable.of(array); const  
    subscription: Rx.Subscription = observable.subscribe(  
      // Observer  
      function (x) {  
        console.log('Next: ' + x);  
      },  
      function (err) {  
        console.log('Error: ' + err);  
      },  
      function () {  
        console.log('Completed');  
      }  
    );  
  }  
}
```

- **Exercise Complete**

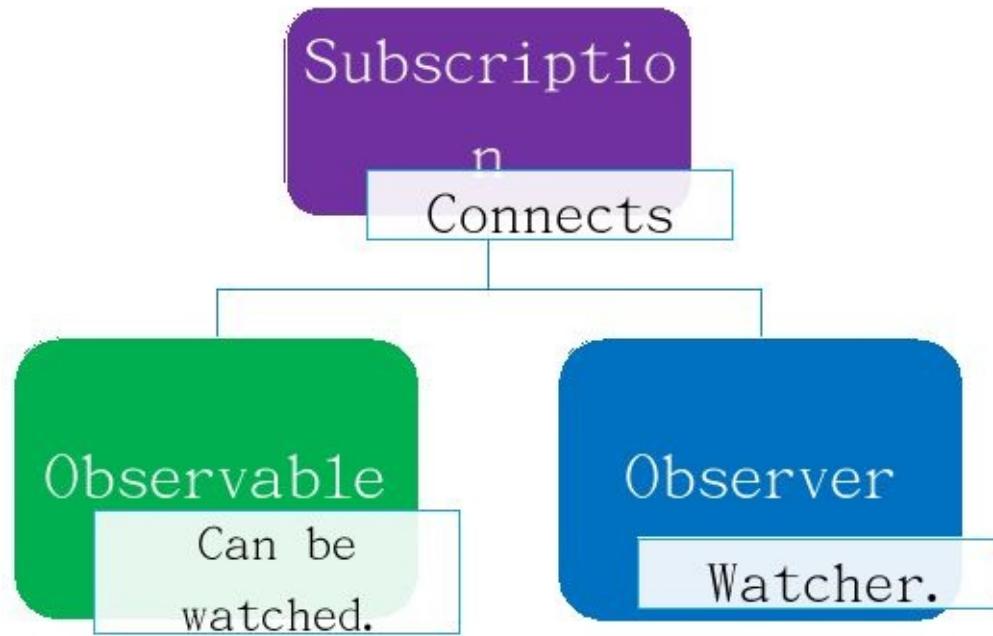
Your app should be working at localhost:4200. Notice that the app component does the following:
An array is created.

An observable is created from the array.

A subscription is created from a subscription to the observable. This subscription implements observer code to handle events.

18.9 Subscriptions

A subscription is like a connection between an Observable and an Observer.



• Linking an Observable and Observer

Use a subscription to link an Observable and an Observer together.

```
const subscription: Rx.Subscription = observable.subscribe(  
  // Observer  
  function (x) {  
    console.log('Next: ' + x);  
  },  
  function (err) {  
    console.log('Error: ' + err);  
  },  
  function () {  
    console.log('Completed');  
  }  
);
```

• Unlinking an Observable and Observer.

To unlink the Observable and Observer, call the method ‘dispose’ in the subscription:
`subscription.dispose();`

18.10 Observables, Observers and Future JavaScript ES7

ES7 is an upcoming proposed standard for JavaScript. It is going to include `Object.observe`. It allows an observer to receive a time-ordered sequence of change records which describe the set of changes which took place to a set of observed objects.

Similar to what RxJS does, only native in the browser.

It's already implemented in some browsers, for example Chrome 36.0.1985.146

18.11 Operators

Operators perform a variety of tasks. Their purpose is to make it more convenient to Observe an Observable: Create Observables.

- Filter Observables.

- Combine Observables.

- Handle Errors

- Perform Utilities

Most operators operate on an Observable and return an Observable. This allows you to apply these operators one after the other, in a chain. Each operator in the chain modifies the Observable that results from the operation of the previous operator.

18.12 Operators - Example Code

In the code below we create an Observable with two events. We then Observe it. If you look at the console you will see event logs. This will be example ‘ch18-ex200’.

- **Step 1 – Build the App using the CLI**

```
ng new ch18-ex200 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch18-ex200  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Edit Class**

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import * as Rx from 'rxjs';  
@Component({  
  selector: 'app-root',  
  template: `  
      
  `,  
  styles: []  
})  
export class AppComponent {  
  constructor(){  
    const observable: Rx.Observable<number> = Rx.Observable.range(0,100); const subscription: Rx.Subscription = observable.subscribe(  
      // Observer  
      val => { console.log(`Next: ${val}`) }, err => { console.log(`Error: ${err}`) }, () => { console.log(`Completed`) }  
    );  
  }  
}
```

- **Exercise Complete**

Your app should be working at localhost:4200. Note the following:

The ‘range’ operator creates a range of events.

The Observer uses arrow functions to handle the events.

18.13 Operators that Create Observables

There are many operators that are used just to create observables.

18.14 Operator 'From'

This operator creates an Observable from other objects that emits multiple values.

In the example below we are creating an observable from an array that emits two values.

```
const array: Array<string> = ['event1', 'event2']; const observable: Rx.Observable<string> = Rx.Observable.from(array);
```

18.15 Operator ‘Interval’

Creates an Observable that emits a value after each period e.g. 0,1,2,3,4

In the example below we are creating an observable emitting a value every ½ second.

Next: 86

Next: 87

Next: 88

Next: 89

```
const observable: Rx.Observable<number> = Rx.Observable.interval(500);
```

```
var observable:Rx.Observable = new Rx.Observable.interval(500);
```

18.16 Operator ‘Of (Was ‘Just’)

Converts an item into an Observable that emits just that item.

In the example below we are creating an observable that emits 500 only once.

```
const observable: Rx.Observable<number> = Rx.Observable.of(500);
```

18.17 Operator ‘Range’

Creates an Observable that emits a range of integers.

In the example below we are creating an observable that emits 1 to 100

```
const observable: Rx.Observable<number> = Rx.Observable.range(0,100);
```

18.18 Operator ‘Repeat’

Creates an Observable that emits the repetition of a given element a specific number of times.

In the example below we are creating an observable that emits the following: 1 2 3 1 2 3 1 2 3 1 2 3

```
const observable: Rx.Observable<number> = Rx.Observable.range(1,3).repeat(4);
```

18.19 Operator ‘Timer’

Creates an Observable that emits a value after due time has elapsed and then after each period.

```
const observable: Rx.Observable<number> = Rx.Observable.timer(2000,500);
```

18.20 Operators that Transform Items Emitted by Observables

Now we know how to create Observables that emit values. Now we need to know how to modify these values.

18.21 Operator 'Buffer'

This is an Operator to periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time.

In the example below we create an observable that emits a value every 100 milliseconds. It then bundles emissions up every 5000 milliseconds. Here are the results:

```
Next:                                         app.component.ts:18
Next:                                         app.component.ts:18
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46
,47,48,49
Next:                                         app.component.ts:18
50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71
,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,9
3,94,95,96,97,98
Next:                                         app.component.ts:18
99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,11
5,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131
,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147
```

```
const observable: Rx.Observable<any> = Rx.Observable .timer(0,100)
.buffer( Rx.Observable.timer(0, 5000) );
```

18.22 Operator ‘Map’

This is an Operator used to transform the items emitted by an Observable by applying a function to each item. This is commonly used.

In the example below we simply put a pipe around the emitted value.

Next: |0|

Next: |1|

Next: |2|

```
const observable: Rx.Observable<string> = Rx.Observable.range(0,100) .map((val) => '|' + val + '|');
```

18.23 Operator ‘Scan’

This is an Operator used to apply a function to each item emitted by an Observable, sequentially, and emit each successive value. It’s like a ‘map’ except that the result from the first function call is fed into the second and so on....

```
Next: 1  
Next: 4  
Next: 25  
Next: 676  
Next: 458329
```

```
const observable: Rx.Observable<number> = Rx.Observable.range(1,5) .scan((val) => { val++; return val * val } );
```

18.24 Operators that Filter Items Emitted By Observables

You don't always need to watch everything. Sometimes you need to only watch certain things.

18.25 Operator ‘Debounce’

This is an Operator used to ensure that an Observer only emits one item in a certain period of time. Useful in observing UI elements. For example, if you have a ‘filter’ box and you don’t want it to respond ‘too’ fast and get ahead of itself with multiple requests. This will stop the networks and computers from getting ‘overloaded’ with too many search requests.

18.26 Operator ‘Debounce’ - Example Code

```
Search: testing123abcde
Search: testing
Search: testing123
Search: testing123abc
Search: testing123abcde
```

In the code below we have a searchbox that uses the ‘debounce’ and ‘distinctUntilChanged’ methods to filter the user’s input.

- **Step 1 – Build the App using the CLI**

```
ng new ch18-ex300 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch18-ex300
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Edit Class**

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import * as Rx from 'rxjs';

@Component({
selector: 'app-root',
template: `
  Search: <input type="text" (keyup)="onChange($event.target.value)" /> <div *ngFor="let log of _logs">Search: {{log}}</div> `,
styles: []
})
export class AppComponent {
  _searchText: string;
  _searchSubject: Rx.Subject<string>;
  _logs: Array<string> = [];

  constructor() {

    // Create new Subject.
    this._searchSubject = new Rx.Subject<string>();

    // Set the Subject up to subscribe to events and filter them by // debounce events and ensure they are distinct.
    this._searchSubject
      .debounceTime(300)
      .distinctUntilChanged()
      .subscribe(
        // Handle event. Log it.
        searchText => this._logs.push(searchText));
  }
}
```

```
}
```

```
public onChange(searchText: string) {
```

```
    // Emit an event to the Subject.
```

```
    this._searchSubject.next(searchText); }
```

```
}
```

• Exercise Complete

Your app should be working at localhost:4200. Note the following:

The constructor sets up the Rxjs subject ‘_searchSubject’ to subscribe to events and filter them. After filtering, each event is added to a log, which is displayed in the component. A Subject is an object than can act both as an Observer and as an Observable. Because it is an observer, it can subscribe to one or more Observables, and because it is an Observable, it can pass through the items it observes by reemitting them, and it can also emit new items.

The ‘onChange’ method is fired when the user types a key on the search box. The code in this method events a string event to the ‘_searchSubject’ Rxjs subject.

18.27 Operator ‘Distinct’

This is an Operator used to suppress the emission of duplicate items.

In the example below we generate a new value every $\frac{1}{2}$ second. We then use map to transform it into string “unchanging value”. Then we add distinct to suppress duplicate values. Thus, we only ever get 1 value emitted.

```
const observable: Rx.Observable<string> = Rx.Observable.interval(500) .map((val) => 'unchanging value').distinct();
```

18.28 Operator 'Filter'

```
Next: 0  
Next: 7  
Next: 14
```

This is an Operator used to emit only the first item, or the first item that meets a condition, from an Observable.

In the example below we generate a new value every $\frac{1}{2}$ second. We then filter out any value not dividable by 7.

```
const observable: Rx.Observable<number> = Rx.Observable.range(0,100) .filter((val) => val % 7 === 0);
```

18.29 Operator ‘Take’

This is an Operator used to emit only the first only the first n items emitted by an Observable.

In the example below we emit new values from 0 to 100 but only take the first 3.

```
const observable: Rx.Observable<number> = Rx.Observable.range(0,100) .take(3);
```

18.30 Operators that Combine Other Observables

Operator	Description
And / Then / When	Combine sets of items emitted by two or more Observables by means of Pattern and Plan intermediaries.
CombineLatest	When an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function.
Join	Combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable.
Merge	Combine multiple Observables into one by merging their emissions.
StartWith	Emit a specified sequence of items before beginning to emit the items from the source Observable.
Switch	Convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables.
Zip	Combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function.

18.31 Share Operator

The share operator allows to share an instance of a Subscription to one or more Observers.

This Operator creates a Subscription when the number of Observers goes from zero to one, then shares that subscription with all subsequent Observers until the number of observers returns to zero, at which point the subscription is disposed.

Useful if you want to watch the same thing from multiple places.

18.32 Share Operator – Example Code

- The code below:

```
var obs = Rx.Observable.interval(500).take(5)
    .do(i => console.log("obs value " + i) )
    .share();

obs.subscribe(value => console.log("observer 1 received " + value));

obs.subscribe(value => console.log("observer 2 received " + value));
```

- Produces the following console logs:

```
obs value 0
observer 1 received 0
observer 2 received 0

obs value 1
observer 1 received 1
observer 2 received 1
```


19 RxJS with Angular

19.1 Introduction

In the last chapter, we went through the core concepts of Reactive Extensions and we learnt about Observables, Observers, Subscriptions and Operators in RxJS. Now we are going to see how we can use Reactive Extensions with Angular.

19.2 AngularJS

Reactive Extensions weren't around at the time of writing AngularJS but Promises were. AngularJS itself used a lot of Promise objects. Promise objects were used by the \$http, \$interval and \$timeout modules. Promise objects could be used to represent asynchronous results: success (return value) or failure (return error). Promise objects were used in Http communication with server and many other objects.

```
function asyncGreet(name) {
  // perform some asynchronous operation, resolve or reject the promise when appropriate.
  return $q(function(resolve, reject) {
    setTimeout(function() {
      if (okToGreet(name)) {
        resolve('Hello, ' + name + '!');
      } else {
        reject('Greeting ' + name + ' is not allowed.');
      }
    }, 1000);
  });
}

var promise = asyncGreet('Robin Hood');
promise.then(function(greeting) {
  alert('Success: ' + greeting);
}, function(reason) {
  alert('Failed: ' + reason);
});
```

19.3 Angular Uses Observables Instead of Promises

In Angular, Promises are now going away in favor of Observables. They haven't completely gone away though.

- **Observables have Several Advantages over Promises:**

Promises only emitted one value/error. Observables can emit multiple values over time. For example, with an Observable you can listen for events on a web socket for a period of time. You can only listen once for a Promise.

You can use Operators with Observers to map, filter etc.

You can cancel Observables.

19.4 Observables and Angular

Angular uses Observables for Asynchronous Data Streams in:

- **DOM Events**

Listening for DOM Events. You can observe a steady stream of data of what the user is doing in the user interface: observe keystrokes, mouse events etc.

DOM

The Document Object Model (DOM) is a cross-platform and language-independent convention for representing and interacting with objects in HTML documents (and other types also). The nodes of every document are organized in a tree structure, called the DOM tree. Objects in the DOM tree may be addressed and manipulated by using methods on the objects. So the DOM gives you a way of manipulating the currently visible HTML document.

- **HTTP Services**

Listening for Server Response(s). Having a connection open with a server and responding to incoming data.

19.5 Observables and DOM Events

Angular DOM events can be observable. In order to use DOM-events we will use the module Rx.DOM (HTML DOM bindings for RxJS) through rx.angular.

So, you can add all these operators to events:

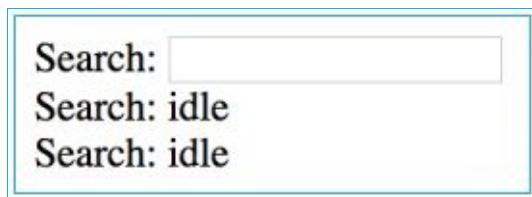
- Filter events.

- Combine watching multiple different events and observing in one place.

- Etc.

19.6 Observables and DOM Events – Example Code

In the code below, we detect when the user has not done anything for a period of 5 seconds. When the user has not done anything for 5 seconds, we add a line saying ‘idle’ to the component’s display. This will be example ‘ch19-ex100’.



Search:

Search: idle

Search: idle

● Step 1 – Build the App using the CLI

```
ng new ch19-ex100 --inline-template --inline-style
```

● Step 2 – Start Ng Serve

```
cd ch19-ex100  
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Edit Class

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import * as Rx from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
    Search:<input type="text"> <div *ngFor="let log of _logs">Search: {{log}}</div> `,
  styles: []
})
export class AppComponent {
  _logs: Array<string> = [];
  constructor() {
    const observable: Rx.Observable<any> = Rx.Observable.merge(
      Rx.Observable.fromEvent(document, 'keydown'), Rx.Observable.fromEvent(document, 'click'),
      Rx.Observable.fromEvent(document, 'mousemove'), Rx.Observable.fromEvent(document, 'scroll'),
      Rx.Observable.fromEvent(document, 'touchstart') );
    const idleEventObservable = observable.bufferTime(5000).filter(function(arr) {
      return arr.length == 0;
    })
    .subscribe(idleEvent => this._logs.push('idle'));
  }
}
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

In the constructor, we merge the emissions from the document events ‘keydown’, ‘click’, ‘mousemove’, ‘scroll’, ‘touchstart’ into one observable. We then buffer it to every five seconds and filter out occurrences when events occur within those five seconds. We then subscribe to the result and when it occurs we add an ‘idle’ log, which is displayed in the component.

19.7 Observables and Http Services

• \$http and Http Module

AngularJS had its own Http module. The \$http service was a core AngularJS service that facilitates communication with the remote HTTP servers via the browser's XMLHttpRequest object or via JSONP.

The Angular Http Module is similar to the Http module in the previous version of Angular, except that it now uses Reactive Extensions (in other words Observables). This is the main difference because Reactive Extensions bring a lot to the table, offering all these Operators mentioned in prior chapters.

We are going to cover the Angular Http module in the next chapter.

20 *Http*

20.1 **Introduction**

99% of Angular projects involve communication between a client (your browser) and some remote server. Normally this is done with Http. So, it is very important to know how this communication works and how you can write code for this communication. That is what this chapter is about.

20.2 Http & Http Methods

• **Http**

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers. HTTP works as a request-response protocol between a client and server. We will cover this in more detail in this chapter: [HTTP](#).

• **Http Methods**

Http methods have been around for a long time (way before AJAX and different types of web application). They can be used in traditional server-side web applications and in client-side AJAX web applications also.

Whenever a client talks to a web server using Http, it includes information about the request ‘method’. The ‘method’ describes what the client wants the server to do, what is the intent of the request. The most commonly used methods are ‘get’ and ‘post’. The ‘get’ method is used to request data from the server. The ‘post’ method is used to send data to the server, to save it or update it.

The most-commonly-used HTTP methods are POST, GET, PUT, PATCH, and DELETE.

20.3 Http Headers

HTTP headers allow the client and the server to pass additional information with the request or the response. A request header consists of its case-insensitive name followed by a colon ':', then by its value (without line breaks).

If you go to a website then use the hamburger menu to access the developer tools then you can see the network communications, including the http calls. If you examine the http requests then you will see the requests made to the server and those returned back from the server.

● Http Request Headers:

Name	Headers	Preview	Response	Timing
font-check-blue.77286...				
analytics.js				
fontawesome-webfont...				
OpenSans-Light-webf...				
OpenSans-Regular-we...				

▼ General

Request URL: <https://www.google-analytics.com/analytics.js>
Request Method: GET
Status Code: 200 (from disk cache)
Remote Address: [2607:f8b0:4002:807::200e]:443
Referrer Policy: no-referrer-when-downgrade

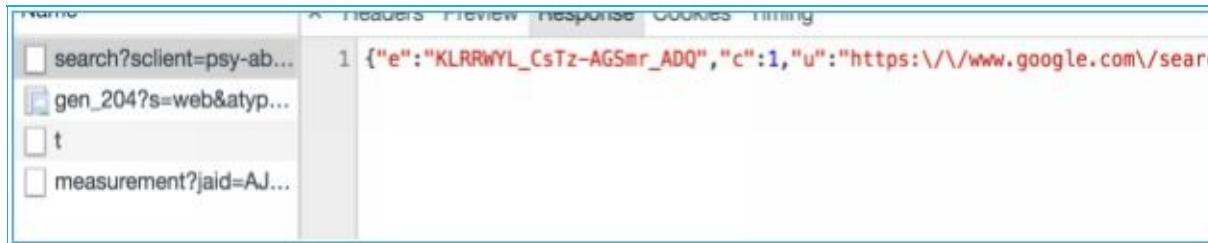
● Http Response Headers:

OpenSans-LightItalic...	▼ Response Headers
OpenSans-Semibold...	age: 1109
OpenSans-Italic-webf...	alt-svc: quic=":443"; ma=2592000; v="39,38,37,36,35"
header-background.4c...	cache-control: public, max-age=7200
data:image/png;char...	content-encoding: gzip
logo_sprite.7d36c4a14...	content-length: 12343
javascript.b28203373c...	content-type: text/javascript
23 requests 39.3KB trans...	date: Tue, 27 Jun 2017 00:44:41 GMT
	expires: Tue, 27 Jun 2017 02:44:41 GMT
	last-modified: Tue, 06 Jun 2017 00:25:39 GMT
	server: Golfe2

20.4 Http Body

The http body allows the client and the server to pass additional information with the request or the response after the header. Http bodies are not always required because a body of information is not always needed. For example, Http ‘get’ requests don’t need to include information in the body – all the information is already contained in the header.

Here is an example of the http body of a response from a server:



A screenshot of a browser's developer tools Network tab. The tab is currently selected, showing a list of network requests. One specific request is highlighted with a blue border. The details for this request are shown in the main pane. At the top of the details pane, there are tabs labeled 'Headers', 'Preview', 'Response', 'Cookies', and 'Timing'. The 'Response' tab is active, indicated by a blue background. In the 'Response' tab, the status code '1' is displayed, followed by the response body: `{"e": "KLRRwYL_CsTz-AGSmr_ADQ", "c": 1, "u": "https://www.google.com/search?hl=en&q=site%3Aexample.com+example.com"}`. The response body is displayed in a monospaced font.

20.5 Http – Passing Information

There are various ways to pass information from the browser to the server. The server normally returns the information in the body, although it can pass information by returning data in the http headers.

● Query Parameters

The angular http client allows you to pass information to the server in the URL using query parameters. For example: <http://localhost:4200/sockjs-node/info?t=1498649243238>

Some characters cannot be part of a URL (for example, a space) and some other characters have a special meaning in a URL. To get around this, the URL syntax allows for encoding on parameters to ensure a valid URL. For example, the ‘space’ character between ‘Atlantic’ and ‘City’ is encoded to ‘%20’ in query in the next example: [https://trailapi-trailapi.p.mashape.com/?q\[city_cont\]=Atlantic%20City](https://trailapi-trailapi.p.mashape.com/?q[city_cont]=Atlantic%20City)

This encoding can be performed when building the URL with string concatenation, using the JavaScript method ‘encodeURIComponent’. Or if you use an Angular object (such as URLSearchParams) to build the query parameter string, it will automatically do this for you.

When navigating in your browser, query parameters are visible to the user on the address bar. When you are performing an AJAX request using the Angular Http client, they will not be visible though!

Query parameters cannot be used to pass as much information as putting it in the request body.

● Matrix Parameters

The angular http client allows you to pass information to the server in the URL using matrix parameters. For example: <http://localhost:4200/sockjs-node/info;t=1498649243238>

Matrix parameters are similar to query strings but use a different pattern. They also act differently because (not having a ‘?’) they can be cached. Also, matrix parameters can have more than one value. You can use matrix parameters in Angular by specifying a URL that includes them. However Angular currently does not have any built-on objects to create URLs with matrix parameters.

Matrix parameters cannot be used to pass as much information as putting it in the request body.

● Path Parameters

The angular http client allows you to pass information to the server in the URL using path parameters. For example: <http://localhost:4200/api/badges/9243238>

● Passing Data in the Request Body

In the old days, HTML Forms (with form tags and input fields) were the best way to send data to the server. The user would fill in a form and hit submit and the data would be posted (using the http ‘post’ method) to the server in the request body.

Now the angular http client allows you to do the same thing programmatically - pass information to the server in the request body using the Http client’s post method.

You can pass more data in the request body than passing it in the URL using query or matrix parameters.

20.6 REST

A RESTful application is a server application that exposes its state and functionality as a set of resources that the clients (the browsers) can manipulate and conforms to a certain set of principles. Examples of resources could be a list of clients, or their orders.

All resources are uniquely addressable, usually through URIs; other addressing can also be used, though. For example, you could use ‘orders/23’ to access order number 23. Or you could use ‘orders/24’ to access order number 24.

All resources can be manipulated through a constrained set of well-known actions, usually CRUD (create, read, update, delete), represented most often through the HTTP methods (see above): POST, GET, PUT and DELETE. Sometimes just some of these HTML methods are used, not all of them. For example, you could use an Http ‘Delete’ to ‘orders/23’ to delete that order.

The data for all resources is transferred through any of a constrained number of well-known representations, usually HTML, XML or JSON. JSON is most common (see below).

20.7 JSON

JSON stands for JavaScript Object Notation. It is a data format used to pass data between the client and the server (in both directions). It is the same data format used by the JavaScript language. It uses a comma to separate items and a colon to separate the name of a property with the data for that property. It uses different types of brackets to denote objects and arrays.

• JSON For Passing an Object Containing Data.

Note how the '{' and '}' brackets are used to denote the start and end of an object.

```
{ "name":"John", "age":31, "city":"New York" }
```

• JSON For Passing an Array

Note how the '[' and ']' brackets are used to denote the start and end of an array.

```
[ "Ford", "BMW", "Fiat" ]
```

• JSON For Passing an Array of Objects

Note how the brackets are combined to create a cars object, which has two properties 'Nissan' and 'Ford'. Each property has an array of models.

```
{
  "cars": {
    "Nissan": [
      {"model":"Sentra", "doors":4}, {"model":"Maxima", "doors":4}
    ],
    "Ford": [
      {"model":"Taurus", "doors":4}, {"model":"Escort", "doors":4}
    ]
  }
}
```

20.8 Http Client

The Angular Http client is a service that you can inject into your classes to perform HTTP communication with a server. This service is available through the Angular Http Module (@angular/http) and you will need to modify your module class (the one for your project) to import this module.

```
@NgModule({  
imports: [  
...  
HttpModule,  
...  
],  
declarations: [ AppComponent ],  
bootstrap: [ AppComponent ]  
})
```

20.9 Using the Http Client

You can inject the Angular Http service directly into your components in this manner:

```
@Injectable()
class CustomerComponent {
...
constructor(private http: Http) {
...
}
```

This is fine for prototyping but not advisable for code maintainability in the long term.

It is better practice to write supporting service classes that inject the Angular Http service and use it to perform http communication with the server:

```
@Injectable()
class CustomerCommunicationService {
...
constructor(private http: Http) {
...
}
```

```
class CustomerComponent {
...
constructor(private http: CustomerCommunicationService) {
...
}
```

If you look at the official Angular documentation at angular.io, you will see the following: **This is a golden rule:** always delegate data access to a supporting service class.

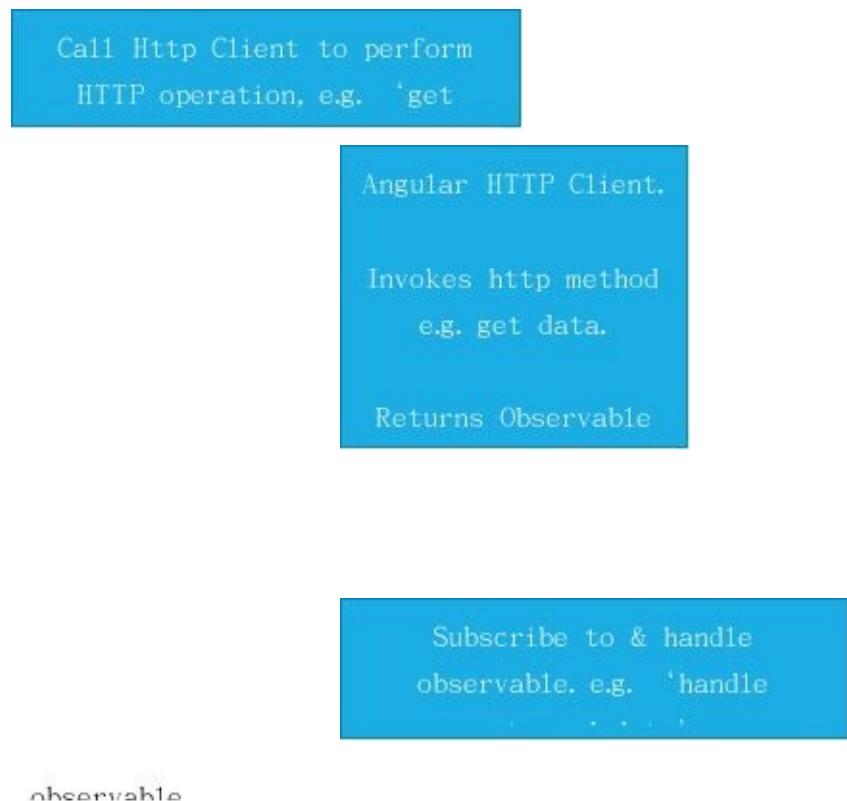


20.10 Asynchronous Operations

In JavaScript making HTTP requests is an *asynchronous* operation. It just sends the HTTP request to the API and *doesn't* wait for a response before continuing with the next line of code.

When the API responds milliseconds or seconds or minutes later then we get notified and we can start processing the response.

In Angular there are two ways of handling these asynchronous operations. We can use *Promises*, or we can use *Observables* which we covered in the Chapter ‘RxJs With Angular’.



Normally we make calls to our supporting service classes and they return the asynchronous result, which we handle in the component.

20.11 Request Options

We will soon cover each type of http call you may make, but before this we will cover the Request Options. When you invoke http communication with a server, you have many ways of configuring the communication. For example: what headers do I use, what media I should accept from the server, what credentials should I pass to the server? You set these options in an Angular object called ‘RequestOptionsArgs’ and you then pass this as an argument to the Angular Http client method call.

Here is an example of the use of the RequestOptionsArg when making a ‘get’ call. Notice how this object is used to specify the url, the http method, parameters, authentication token and body.

```
var basicOptions:RequestOptionsArgs = {  
url: 'bankInfo',  
method: RequestMethods.Get,  
params: {  
    accountNumber: accountNumber  
},  
headers: new Headers({'Authentication': authenticationStr}), body: null  
};
```

20.12 Http Method ‘Get’

This http method is very commonly used to ‘get’ data from a server. It typically does not use the request body.

● Example:

url:

```
/customers/getinfo.php?id=123
```

request body:

none

● Http ‘Gets’:

Idempotent. Calling the same put multiple times has the same effect as calling it once.

Can remain in the browser history Can be bookmarked Have length restrictions Request uses http header.

Response returned as http body.

● Idempotence

This method should be implemented in an idempotent manner on the server. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests).

20.13 Http Method ‘Get’ – Example

Countries

- Asturian (ast)
- Belarusian (be)
- Breton (br)
- Catalan (ca)
- Catalan (Valencian) (ca)
- Chinese (zh)

This is a component that gets the list of languages and language codes from Snapchat. This will be example ‘ch20-ex100’.

● Step 1 – Build the App using the CLI

```
ng new ch20-ex100 --inline-template --inline-style
```

● Step 2 – Install Http Node Module & Start Ng Serve

```
cd ch20-ex100
npm install @angular/http --save
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

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

import { HttpClientModule } from '@angular/http';

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

● Step 5 – Edit Service

Edit ‘swagger.service.ts’ and change it to the following:

```
import { Injectable } from '@angular/core'; import { Http } from '@angular/http';

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

  getLanguages() {
    return this._http.get('https://languagetool.org/api/v2/languages');
  }
}
```

● Step 6 – Edit Component

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit } from '@angular/core'; import { SwaggerService } from './swagger.service';

@Component({
  selector: 'app-root',
  template: `
    <h1>Countries</h1>
    <ul>
      <li *ngFor="let language of _languages"> {{language.name}}&nbsp;({{language.code}})
    </li>
    </ul>
  `,
  styles: []
})
export class AppComponent implements OnInit{
  _languages = new Array<any>();

  constructor(private _swaggerService: SwaggerService) {}

  ngOnInit(){
    this._swaggerService.getLanguages().subscribe(
      res => {
        this._languages = res.json();
      },
      error => { console.log('an error occurred'); }
    )
  }
}
```

● Exercise Complete

Your app should be working at localhost:4200 and you should see a list of languages. Note the following: The file ‘swagger.service.ts’ creates a service. Note that this service has the injectable annotation to enable it to be injected into the app component. This service has a constructor into which the Angular Http module is injected. This service contains a method ‘getLanguages’, which makes an http call to a server, which returns an observable.

The file ‘app.component.ts’ creates a component. Note that the swagger service is injected

into this component using the constructor. When the component initializes it calls the swagger service and subscribes to the observable result with two methods: the first one for success and the second one for failure.

The first method (the ‘success’ one) accepts the http result as a parameter and calls the ‘.json’ method to parse the response body from JSON into JavaScript objects. This method sets the instance variable ‘_languages’ to the returned array of JavaScript objects, which is then visible in the component.

20.14 Http Method ‘Get’ Using Parameters

Now we have covered a basic ‘get’, we should now cover a get ‘what’. A get that uses parameters to get a specific thing (or things). Very useful if you want to get the information for a specific customer off the server. You can do this simply modifying the URI of the ‘get’ to include uquery parameters, or you can do this using the Angular search or parameter objects embedded into the RequestOptionsArgs object.

20.15 Http Method ‘Get’ Using Parameters – Example

This is an example of performing an http ‘get’ in three different ways using query parameters, triggered by three different buttons. This will be example ‘ch20-ex200’.

{ "places": [{"city": "Atlanta", "state": "Georgia", "country": "United States", "name": "Boat Rock", "Industrial Boulevard go south for 3.8 miles, turn left onto Bakers Ferry Road SW, go 0.5 miles small 6 car parking lot. There is a small kiosk at the edge of the lot with a rough map of the area (see droptow map).

1220 [], "activities": [{"name": "Boat Rock", "unique_id": "2-1012", "place_id": "5370", "activity_type_id": {"length": "1\\n"}, "description": "For those of us who like hiking AND rock climbing! Very boulders. A great experience for families and it's fun getting to watch the expert climbers on the 15T16:12:21Z", "id": 2, "name": "hiking", "updated_at": "2012-08-15T16:12:21Z"}, "thumbnail": "(\"city\": \"Atlanta\", \"state\": \"Georgia\", \"country\": \"United States\", \"name\": \"Brookhaven Park\", \"pan": Atlanta", "lat": 33.86519, "lon": -84.33776, "description": null, "date_created": null, "children": []}, {"u": 4958, "place_id": "19072, "activity_type_id": 2, "activity_type_name": "hiking", "url": "http://www and picnic area in 9 acre DeKalb county park.", "length": 1, "activity_type": {"created_at": "2012-15T16:12:21Z"}, "thumbnail": null, "rank": null, "rating": 0}], "city": "Atlanta", "state": "Michigan", "country": "United States", "name": "Campground", "parent_id": null, "unique_id": 17708, "directions": "Atlanta Field Office
Forest campground is closed due to budget cuts. Effective May 5, 2009, until further notice. Will come, first-serve basis. No reservations. 12 sites for tent and small trailer use. Carry-in boat lau

- Step 1 – Build the App using the CLI

ng new ch20-ex200 --inline-template --inline-style

- Step 2 – Install Http Node Module & Start Ng Serve

```
| cd ch20-ex200  
| npm install @angular/http --save  
| ng serve
```

• Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

• Step 4 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

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

● Step 5 – Edit Component

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { Http, Headers, URLSearchParams, RequestOptions, RequestMethod } from '@angular/http';
@Component({
selector: 'app-root',
template: `
<input [(ngModel)]="search" placeholder="city"> <button (click)="doSearchConcatenatedUrl()">Search (Concatenated URL)</button>
<button (click)="doSearchOptionParameters()">Search (Parameters Object)</button> <button (click)="doSearchOptionSearch()">Search (Search Object)</button> <p>{ {_result} }</p>
`,
styles: []
})
export class AppComponent {
_search = 'Atlanta';
_result = "";

constructor(private _http: Http){}
}

doSearchConcatenatedUrl(){
  const mashapeKey = '0xWYjpdztcmsheZU9AWLNQcE9g9wp1qdRkFjsneaEp2Yf68nYH'; const headers: Headers = new Headers();
headers.append('Content-Type', 'application/json'); headers.append('X-Mashape-Key', mashapeKey); const options = new RequestOptions({headers: headers });
  // concatenated string
  const concatenatedUrl: string =
"https://trailapi-trailapi.p.mashape.com?q[city_cont]="+
encodeURIComponent(this._search);
  return this._http.get(concatenatedUrl, options).subscribe(
res => { this._result = JSON.stringify(res.json()); }
)
}

doSearchOptionParameters(){
  const mashapeKey = '0xWYjpdztcmsheZU9AWLNQcE9g9wp1qdRkFjsneaEp2Yf68nYH'; const headers: Headers = new Headers();
headers.append('Content-Type', 'application/json'); headers.append('X-Mashape-Key', mashapeKey); const params: URLSearchParams =
new URLSearchParams(); params.append('q[city_cont]', this._search);
  // params object
  const options = new RequestOptions({ headers: headers, params: params });
  return this._http.get("https://trailapi-trailapi.p.mashape.com?q[city_cont]="+
encodeURIComponent(this._search), options).subscribe(
res => { this._result = JSON.stringify(res.json()); }
)
}

doSearchOptionSearch(){
  const mashapeKey = '0xWYjpdztcmsheZU9AWLNQcE9g9wp1qdRkFjsneaEp2Yf68nYH'; const headers: Headers = new Headers();
headers.append('Content-Type', 'application/json'); headers.append('X-Mashape-Key', mashapeKey); const params: URLSearchParams =
new URLSearchParams(); params.append('q[city_cont]', this._search);
  // search object
  const options = new RequestOptions({ headers: headers, search: params });
  return this._http.get("https://trailapi-trailapi.p.mashape.com?q[city_cont]="+
encodeURIComponent(this._search), options).subscribe(
res => { this._result = JSON.stringify(res.json()); }
)
```

● Exercise Complete

Your app should be working at localhost:4200. Notice the following:

Method ‘doSearchConcatenatedUrl’ manually builds the url string by manually appending the url with the encoded parameter (encoded using the ‘encodeURIComponent’ method).

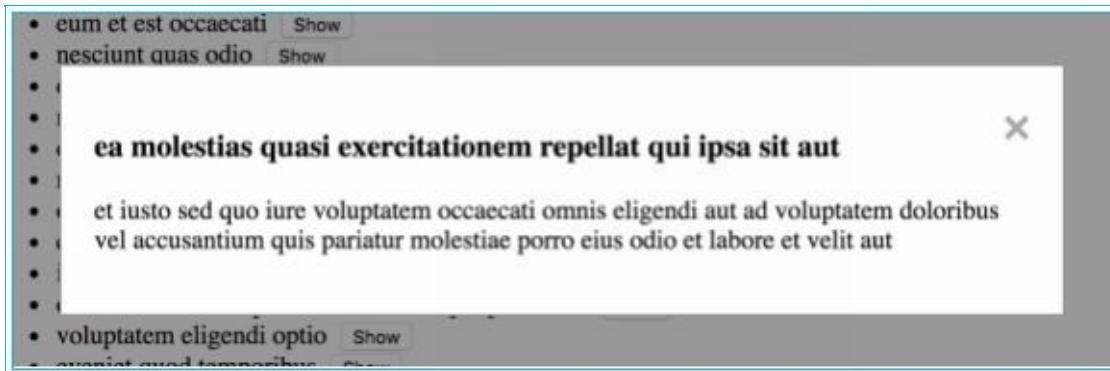
Method ‘doSearchOptionParameters’ builds a URLSearchParams object and adds parameters to that object. Then it sets the ‘params’ property of the RequestOptions object to that object. Then the ‘get’ method is called on the http client, passing the RequestOptions object as the second argument.

Method ‘doSearchOptionSearch’ builds a URLSearchParams object and adds parameters to that object. Then it sets the ‘search’ property of the RequestOptions object to that object. Then the ‘get’ method is called on the http client, passing the RequestOptions object as the second argument.

All three methods result in the same url, for example: [https://trailapi-trailapi.p.mashape.com/?q\[city_cont\]=Atlanta](https://trailapi-trailapi.p.mashape.com/?q[city_cont]=Atlanta)

20.16 Http Method ‘Get’ Using Path Parameters – Example

This is an example of performing an http ‘get’ in using path parameters. The user is displayed a list of articles and each one has a ‘show’ button. The user can click on this button and an http ‘get’ will be called (with path parameters) to get the details of the article. This article is then displayed on a popup modal. This will be example ‘ch20-ex300’.



● Step 1 – Build the App using the CLI

```
ng new ch20-ex300 --inline-template --inline-style
```

● Step 2 – Install Http Node Module & Start Ng Serve

```
cd ch20-ex300
npm install @angular/http --save
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

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

import { HttpClientModule } from '@angular/http';

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

● Step 5 – Edit Component

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit, AfterViewInit, ViewChild } from '@angular/core'; import { Http, Headers, URLSearchParams, RequestOptions, RequestMethod } from '@angular/http';
@Component({
  selector: 'app-root',
  template: `
    <ul>
      <li *ngFor="let post of _posts">
        {{post.title}}&ampnbsp&ampnbsp<button (click)="showPost(post.id)">Show</button> </li>
    </ul>
    <div #myModal id="myModal" class="modal"> <div class="modal-content"> <span class="close" (click)="closeModal()">&times;</span> <h3>{{this._post.title}}</h3> <p>{{this._post.body}}</p> </div>
  </div>
`,
  styles: []
})
export class AppComponent implements OnInit {
  _posts = [];
  _post = {};
  @ViewChild('myModal') _myModal: any;

  constructor(private _http: Http) {}

  ngOnInit() {
    return this._http.get("http://jsonplaceholder.typicode.com/posts").subscribe(
      res => {
        this._posts = res.json();
      }
    );
  }

  showPost(postId: number) {
    this._http.get(`http://jsonplaceholder.typicode.com/posts/${postId}`).subscribe(
      res => {
        this._post = res.json();
        this._myModal.nativeElement.style.display = 'block';
      }
    );
  }

  closeModal() {
    this._myModal.nativeElement.style.display = 'none';
  }
}
```

● Step 6 – Edit Styles

Edit ‘styles.css’ and change it to the following:

```
.modal {
  display: none; /* Hidden by default */
  position: fixed; /* Stay in place */
  z-index: 1; /* Sit on top */
  left: 0;
```

```
top: 0;  
width: 100%; /* Full width */  
height: 100%; /* Full height */  
overflow: auto; /* Enable scroll if needed */  
background-color: rgb(0,0,0); /* Fallback color */  
background-color: rgba(0,0,0,0.4); /* Black w/ opacity */  
}  
  
/* Modal Content/Box */  
.modal-content {  
    background-color: #fefefe;  
    margin: 15% auto; /* 15% from the top and centered */  
    padding: 20px;  
    border: 1px solid #888;  
    width: 80%; /* Could be more or less, depending on screen size */  
}  
  
/* The Close Button */  
.close {  
    color: #aaa;  
    float: right;  
    font-size: 28px;  
    font-weight: bold;  
}  
  
.close:hover,  
.close:focus {  
    color: black;  
    text-decoration: none;  
    cursor: pointer;  
}
```

● Exercise Complete

Please note the following:

1. In the app component method ‘showPost’ we use template literals to inject the post id into the url string.

20.17 Http Method ‘Post’

This http method is very commonly used to ‘get’ data from a server. It typically does not use the request body.

● Example:

url:

/customers/new	request body:
name=Mark&city=Atlanta&state=GA	

● Http ‘Posts’:

Not idempotent. Calling the same put multiple times will result in a different effect from calling it once.

Cannot be cached Cannot remain in the browser history Cannot be bookmarked Don’t have length restrictions Request uses http body.

Response returned as http body.

● Idempotence

By its very nature, http posts are not idempotent. They have side effects, for example adding a customer.

20.18 Http Method ‘Post’ – Example

This is an example of performing an http ‘post’. The user can input a title and a body and hit ‘add’ to post them to the server. The server returns information to the browser, which is added to the list of ‘added’ at the bottom. This will be example ‘ch20-ex400’.

The screenshot shows a simple Angular application interface. At the top, there's a form with two input fields: 'Title:' containing 'About Angular' and 'Body:' containing 'Angular is a User interface library.' Below the form is a blue 'Add' button. Underneath the form, the text 'You Added:' is displayed in bold, followed by the entry 'About Angular'.

- **Step 1 – Build the App using the CLI**

```
ng new ch20-ex400 --inline-template --inline-style
```

- **Step 2 – Install Http Node Module & Start Ng Serve**

```
cd ch20-ex400
npm install @angular/http --save
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Edit Module**

Edit ‘app.module.ts’ and change it to the following:

```
import { BrowserModule } from '@angular/platform-browser'; import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/http';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
@NgModule({
declarations: [
  AppComponent
],
imports: [
  BrowserModule,
  HttpClientModule,
  FormsModule
],
providers: [],
bootstrap: [AppComponent]
```

```
})  
export class AppModule { }
```

● Step 5 – Edit Component

Edit ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit, AfterViewInit, ViewChild } from '@angular/core'; import { Http, Headers, URLSearchParams, RequestOptions, RequestMethod } from '@angular/http';
@Component({
  selector: 'app-root',
  template: `

    <div>
      Title:
      <br/>
      <input type="text" [(ngModel)]=" _title" size="50" /> <br/>
      <br/>
      Body:
      <br/>
      <textarea [(ngModel)]=" _body" rows="2" cols="50"> </textarea>
      <br/>
      <button (click)="onAdd()">Add</button> </div>
      <p><b>You Added:</b></p> <p *ngIf=" _added.length == 0">None</p> <p *ngFor="let added of _added"> {{added.title}} </p>
      `;
  styles: ['div { padding: 20px; background-color: #C0C0C0 }']
})
export class AppComponent {
  _title: string;
  _body: string;
  _added: Array<any> = new Array<any>();

  constructor(private _http: Http) {}

  onAdd(){
    const requestBody = {
      title: this._title || '[Unspecified]', body: this._body || '[Unspecified]', };
    this._http.post("http://jsonplaceholder.typicode.com/posts", requestBody).subscribe(
      res => {
        this._added.push(res.json());
      }
    )
  }
}
```

● Exercise Complete

Please note the following:

1. In the app component method ‘onAdd’ we create the object that is going to be posted to the server in the request body.
2. In the app component method ‘onAdd’ we use the ‘or’ operator to ensure that we pass something valid to the server - either ‘this._title’ (if that exists), or the text ‘[Unspecified]’.

```
title: this._title || '[Unspecified]'
```

3. In the app component method ‘onAdd’ we subscribe to the http post and we use an arrow function to process the returned result. We add the returned the result to the array of ‘_added’, so it appears at the bottom.

20.19 Http Method ‘Put’ Using Path Parameters

The ‘put’ method is similar to the ‘post’ method, except that with a rest service it is typically used to update a resource rather than create it.

- **Http ‘Puts’:**

- Idempotent. Calling the same put multiple times has the same effect as calling it once.

- Don’t have length restrictions Not cacheable.

- Request uses http body.

- Response returned as http header.

20.20 Http Method ‘Patch’ Using Path Parameters

The ‘patch’ method is like the ‘put’, except that it is not idempotent.

For example, this would be useful to increase the value on a resource by a certain amount.

- **Http ‘Patches’:**

- Not Idempotent. Calling the same put multiple times has a different effect as calling it once.

- Don’t have length restrictions Not cacheable.

- Request uses http body.

- Response returned as http header.

20.21 Http Method ‘Delete’ Using Path Parameters

The ‘delete’ method is used to remove a resource from the server.

- **Http ‘Deletes’:**

- Idempotent. Calling the delete put multiple times has the same effect as calling it once.

- Not cacheable.

- Request uses http header.

- Response returned as http header.

20.22 Modifying the Server Response

Remember that http client service calls return Observable objects. That means that the server returns an asynchronous data stream to the client (browser) and that you can use the RxJS module to process that data using the operators we mention in the chapter ‘Observers, Reactive Programming & RxJS’.

This includes the ‘map’ operator that we can use to transform the data.

20.23 Modifying the Server Response - Example

This is an example of using the RxJs ‘map’ operator (with a function) to modify the response returned from the server. In this case we use it to transform data into typed data (i.e. data structured in a class). This will be example ‘ch20-ex500’.

- **sunt aut :** quia et suscipit su
- **qui est e:** est rerum tempore v
- **ea molest:** et iusto sed quo iu
- **eum et es:** ullam et saepe reic
- **nesciunt :** repudiandae veniam
- **dolorem e:** ut aspernatur corpo
- **magnam fa:** dolore placeat quib

● Step 1 – Build the App using the CLI

```
ng new ch20-ex500 --inline-template --inline-style
```

● Step 2 – Install Http Node Module & Start Ng Serve

```
cd ch20-ex500
npm install @angular/http --save
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Create Class

Create the following typescript class ‘post.ts’ in the ‘app’:

```
export class Post {
  _title: string = "";
  _body: string = "";

  constructor(title: string, body: string){
    const titleNaN = title || "";
    const bodyNaN = body || "";
    this._title = titleNaN.length > 10 ? titleNaN.substring(0,9) : titleNaN; this._body = bodyNaN.length > 20 ? bodyNaN.substring(0,19) : bodyNaN; }

  get title(): string{
    return this._title;
  }

  get body(): string{
    return this._body;
  }
}
```

● Step 5 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

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

import { HttpClientModule } from '@angular/http';

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

● Step 6 – Edit Component

Edit ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { Http } from '@angular/http';
import { Post } from './Post';
import 'rxjs/Rx';

@Component({
  selector: 'app-root',
  template: `
    <ul>
      <li *ngFor="let post of _posts"> <b>{{post.title}}</b> {{post.body}}
    </li>
    </ul>
  `,
  styles: []
})
export class AppComponent {
  _posts: Array<Post>;
  constructor(private _http: Http) {}
  ngOnInit() {
    return this._http.get("http://jsonplaceholder.typicode.com/posts") .map(
      response => response.json()
    )
    .map(
      response => {
        const postsArray: Array<Post> = new Array<Post>(); for (const responseItem of response){
          const post =
            new Post(responseItem['title'], responseItem['body']); postsArray.push(post);
        }
        return postsArray;
      }
    )
    .subscribe(
```

```
response => {
this._posts = response;
}
);
}
}
```

● Exercise Complete

Please note the following:

1. We create the typescript class ‘Post’ to store each post. Note that this class has a constructor that trims the title and body of each post. Note that we use the ‘or’ trick to convert ‘non-truthy’ values to empty string:

```
const titleNaN = title || ";
```

2. We use the code below to get data from the server. We use the first map to convert the response json into an object. We use the second map to convert the response (an array of objects) into a typed array of Post classes. We then subscribe to the result.

```
return this._http.get("http://jsonplaceholder.typicode.com/posts") .map(
response => response.json()
)
.map(
response => {
const postsArray: Array<Post> = new Array<Post>(); for (const responseItem of response){
const post =
new Post(responseItem['title'], responseItem['body']); postsArray.push(post);
}
return postsArray;
}
)
.subscribe(
response => {
this._posts = response;
}
);
```

20.24 Handling an Error Server Response

When you subscribe to an HTTP method call, you supply a ‘handler’ method that processes the results. However, you can supply other ‘handler’ method also – one to handle errors and another to handle completion.

```
.subscribe(  
    function(response) { console.log("Success " + response)}, function(error) { console.log("Error " + error)}, function() {  
    console.log("Completion")}  
);
```

20.25 Handling an Error Server Response - Example

This is an example of handling a server error and displaying an appropriate error message. This will be example ‘ch20-ex600’.



Error: 404: Not Found

- **Step 1 – Build the App using the CLI**

```
ng new ch20-ex600 --inline-template --inline-style
```

- **Step 2 – Install Http Node Module & Start Ng Serve**

```
cd ch20-ex600
npm install @angular/http --save
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Create Service Class**

Create the following typescript class ‘service.ts’ in the ‘app’:

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

@Injectable()
export class Service {

constructor(private _http: Http) {}

getPosts() : Observable<any> {
  return this._http.get("http://jsonplaceholder.typicode.com/postss") .map(
    response => response.json()
  )
};

}
```

- **Step 5 – Edit Module**

Edit ‘app.module.ts’ and change it to the following:

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

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

```

import { HttpModule } from '@angular/http';
import { Service } from './Service';

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

```

● Step 6 – Edit Component

Edit ‘app.component.ts’ and change it to the following:

```

import { Component } from '@angular/core'; import { Service } from './Service';
import 'rxjs/Rx';

@Component({
  selector: 'app-root',
  template: `
    <ul>
      <li *ngFor="let post of _posts"> <b>{{post.title}}</b> {{post.body}}
    </li>
    </ul>
    <div *ngIf="_error"> Error: {{_error.status}}: {{_error.statusText}}
    </div>
  `,
  styles: ['div {font-size:20px; padding: 5px; background-color: red;color: white}']
})
export class AppComponent {
  _posts = [];
  _error;

  constructor(private _service: Service) {}

  ngOnInit() {
    this._service.getPosts()
      .subscribe(
        response => {
          this._posts = response;
        },
        error => {
          this._error = error;
        }
      );
  }
}

```

● Exercise Complete

Please note the following:

1. The service class ‘service.ts’ uses an incorrect URL that will throw a 404 error.
2. The component ‘app.component.ts’ calls the method ‘getPosts’ and subscribes to its result, using two methods, each one implemented with an arrow function. The first processes a successful result, the second processes an error. In this example we process the error, setting the instance variable ‘_error’ to its result.
3. The ‘_error’ instance variable is referred to in the template. If set, it shows a message in red.

20.26 Asynchronous Pipes

The async pipe subscribes to an Observable or Promise and returns the latest value it has emitted. When a new value is emitted, the async pipe marks the component to be checked for changes. When the component gets destroyed, the async pipe unsubscribes automatically to avoid potential memory leaks.

20.27 Asynchronous Pipes – Example

This is an example of using a map to transform the output from a server then using a pipe to output it. This will be example ‘ch20-ex700’.

Post Title Names

```
sunt aut facere repellat provident occaecati excepturi optio reprehenderit  
qui est esseca molestias quasi exercitationem repellat qui ipsa sit auteum et est  
occaecatinescunt quas odiodolorem eum magni eos aperiam quiamagnam  
facilis autemdolorem dolore est ipsammescunt iure omnis dolorem tempora  
et accusantiumoptio molestias id quia eumet ea vero quia laudantium  
autemin quibusdam tempore odit est dolorem dolorum ut in voluptas mollitia  
et saene quo animivoluntatem eliqendi ontioeveniet mod termoribussint
```

● Step 1 – Build the App using the CLI

```
ng new ch20-ex700 --inline-template --inline-style
```

● Step 2 – Install Http Node Module & Start Ng Serve

```
cd ch20-ex700  
npm install @angular/http --save  
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Edit Module

Edit ‘app.module.ts’ and change it to the following:

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

● Step 5 – Edit Component

Edit ‘app.component.ts’ and change it to the following:

```

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Post Title Names</h1> <p>{{_result|async}}</p> `,
  styles: []
})
export class AppComponent {
  _result: any;

  constructor(private _http: Http) {}

  ngOnInit() {
    this._result =
      this._http.get("http://jsonplaceholder.typicode.com/posts") .map(
        response => response.json()
      )
      .map(
        response => {
          let titles = "";
          for (const responseItem of response){
            titles += responseItem['title'];
          }
          return titles;
        }
      );
  }
}

```

● Exercise Complete

Please note the following:

1. The component ‘app.component.ts’ calls the http ‘get’ method to return a list of posts then it uses the ‘map’ operator (and a function) to convert it into a string of all the titles of the posts. It then assigns the result to the ‘_result’ instance variable.
2. The component ‘app.component.ts’ uses a template to display the value of the ‘_result’ instance variable.

21 Forms

21.1 Introduction

You cannot enter data in an application without forms. AngularJS allowed the user to create forms quickly, using the ‘NgModel’ directive to bind the input element to the data in the \$scope. You can also do the same in Angular. However Angular 4 has a new forms module that makes it easier to: Create forms dynamically.

- Validate input with common validators (required).

- Validate input with custom validators.

- Test forms.

21.2 Two Methods of Writing Forms

You can continue writing forms in a similar way to how you used to in AngularJS but I recommend using the new Forms module as it will do more of the work for you. The Forms module offers two main ways of working with forms: template-driven forms and reactive forms. Both ways work with the same Forms module.

• **Template-Driven Forms**

This is similar to how we did things in AngularJS. We build the html template, add a few directives to specify additional information (such as validation rules) and Angular takes charge of building the model objects for us behind the scenes: the underlying forms, form groups and controls.

Advantages: simple, quick to get started, perfect for simple forms, don't need to know how form model objects work Disadvantages: html & business rules are coupled, no unit testing

• **Reactive Forms**

This is different, we build the model objects ourselves (including validation form rules) and it binds (and syncs) to the template.

Advantages: more control, perfect for more advanced forms, enable unit testing, html & business rules are de-coupled Disadvantages: need to know how form model objects work, take more time to develop

21.3 Form Module

As of the time of writing this book, the Angular CLI generates projects with the node dependency to the Forms module already setup. So, all you have to do is adjust your module to import the forms module. Here is an example of the ‘app.module.ts’ file:

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

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

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

21.4 Form Model Objects

The information below applies to both methods of writing forms: template and reactive. Both methods of writing forms use the same model objects. Let's take a quick look at them.

● NgForm

Stores state information for the form, including the following:

- Values for all the controls inside the form.

- Groups of fields in the form.

- Fields in the form.

- Validators.

● FormGroup

Stores the value and validity state of a group of FormControl instances.

- Values for all the controls inside the form group.

● FormControl

Stores the value and validity state of an individual control, for instance a listbox.

- Value.

- Validation state.

- Status e.g. disabled.

You can add a subscriber to respond to form control value changes.

```
this.form.controls['make'].valueChanges.subscribe(  
(value) => { console.log(value); }  
);
```

You can add a subscriber to respond to form control status changes.

```
this.form.controls['make'].statusChanges.subscribe(  
(value) => { console.log(value); }  
);
```

Example output:

```
INVALID  
VALID
```

● FormArray

Used to track the value and state of multiple FormControls, FormGroups or FormArrays. Useful for dealing with multiple form objects and tracking overall validity and state.

21.5 Forms and CSS

The information below applies to both methods of writing forms: template and reactive.

When you have form validation, you need to highlight invalid data when it occurs. The forms module has been designed to work with CSS to make it very easy to highlight invalid user input. The following styles are automatically added to the form elements. All you need to do is add the css code to produce the required visual effect.

ng-touched	Style applied if control has lost focus.
ng-untouched	Style applied if control has not lost focus yet.
ng-valid	Style applied if control passes validation.
ng-invalid	Style applied if control does not pass validation.
ng-dirty	Style applied if user has already interacted with the control.
ng-pristine	Style applied if user has not interacted with the control yet.

21.6 Template Forms

As mentioned earlier, template forms use directives to create the form model objects. So, you build the input form and inputs in the template, add a few directives and the form is ready and working.

Template forms are perfect for quickly building simple forms that have simple validation.

Template forms work asynchronously. So, the model objects are not available until the view has been initialized and the directives have been processed. So not all of the model objects are even available in the `AfterViewInit` lifecycle method.

21.7 Template Forms – Module Setup

To use Angular template forms, your application module needs to import the Forms Module from the ‘@angular/forms’ node module.

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

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

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

21.8 Template Forms - Example

Let's go through creating a template form and see what is needed to make it work. This will be example 'ch21-ex100'.

- **Step 1 – Build the App using the CLI**

```
ng new ch21-ex100 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch21-ex100  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- **Step 4 – Edit Module**

Edit 'app.module.ts' and change it to the following:

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

- **Step 5 – Edit Component**

Edit 'app.component.ts' and change it to the following:

```
import { Component, ViewChild } from '@angular/core'; import { NgForm, RequiredValidator } from '@angular/forms';  
@Component({  
  selector: 'app-root',  
  template: `<form #f novalidate>  
    <p>First Name <input name="fname"/></p> <p>Last Name <input name="lname"/></p> Valid: {{ f.valid }}  
    Data: {{ f.value | json }}  
  </form>  
  `,  
  styles: []
```

```
})
export class AppComponent {
@ViewChild('f') f: NgForm;
}
```

● Step 6 – View App

Notice that this component just displays the input forms. It does not display any further information.

First Name asdasd

Last Name asa

● Step 7 – Edit Component

Now we are going to add some directives to the form and input tags to get the form working as a template form. The changes are outlined in bold below.

```
import { Component, ViewChild } from '@angular/core'; import { NgForm, RequiredValidator } from '@angular/forms';
@Component({
selector: 'app-root',
template: `
<form #f="ngForm" novalidate> <p>First Name <input name="fname" ngModel required /></p> <p>Last Name <input name="lname"
ngModel required /></p> Valid: {{ f.valid }}  

    Data: {{ f.value | json }}  

</form> `,
styles: []
})
export class AppComponent {
    @ViewChild('f') f: NgForm;
}
```

● Step 8 – View App

Notice that this component displays the input forms and the state of the form below – its validity and its data.

First Name peter

Last Name smith

Valid: true Data: { "fname": "peter", "lname": "smith" }

● Exercise Complete

This shows how quickly you can use the ngForm and ngModel directives to make a template form, with a form object that holds the form state (including data). Note also how the html input fields use the ‘name’ attribute – this is picked up by the form directives and used to identify that control and its value.

21.9 Template Variables and Data Binding

• Template Variables

Sometimes you need access to each control to access its state, its value etc. You can use the syntax below to set a template variable to the ‘ngModel’ of the control (ie its FormControl object). You can also use the ViewChild to access the FormControl as a variable.

```
import { Component, ViewChild } from '@angular/core'; import { NgForm, FormControl, RequiredValidator } from '@angular/forms';
@Component({
selector: 'app-root',
template: `
<form #f="ngForm" novalidate>
  <p>First Name <input name="fname" ngModel #fname="ngModel" required /> </p>
  <h2>Form Template Variable</h2> Valid {{ fname.valid}}
  Data: {{ fname.value | json }}
  <h2>From Instance Variable</h2> Valid {{ fname2.valid}}
  Data: {{ fname2.value | json }}
</form> `,
styles: []
})
export class AppComponent {
  @ViewChild('f') f: NgForm;
  @ViewChild('fname') fname2: FormControl;
}
```

You can also use template variables to query form control states. This makes it very easy to add logic in the template to hide and show error messages.

.touched	Has the user performed any input in this field? Returns true or false.
.valid	Does the field input pass validation? Returns true or false.
.value	The current form value.
.hasError('required')	Has the specified error occurred? Returns true or false.

• Binding Form Control Value

Sometimes you need to 2-way bind each control’s value to the model so that you can get and set each control’s value as required. Useful if you want to set the form control’s value. Change the ‘ngModel’ directive to use two-way binding and link it to the instance variable, in the case below ‘_name’:

```
<input type="text" class="form-control" name="name" placeholder="Name (last, first)" [(ngModel)]="_name" required>
```

21.10 Template Forms and Data Binding - Example

Let's go through creating a template form and binding the form controls to instance variables. Also, let's build this form with bootstrap styling so it looks good. Also, the submit form has a button that enables or disables according to the user's input. This will be example 'ch21-ex200'.

The screenshot shows a simple HTML form with a light blue border. The form is titled 'Appointment'. It includes fields for 'Name' (with value 'mark'), 'Password', 'Appointment Time' (with radio buttons for 12pm, 2pm, 4pm, where 12pm is selected), and 'Ailment' (a text area). At the bottom is a 'Submit' button. Below the form, a message box displays the validation status: 'Valid: false Data: { "name": "mark", "password": "", "time": "", "ailment": "" }'.

• Step 1 – Build the App using the CLI

```
ng new ch21-ex200 --inline-template --inline-style
```

• Step 2 – Start Ng Serve

```
cd ch21-ex200  
ng serve
```

• Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see 'app works!'.

• Step 4 – Edit Web Page.

Edit the file 'index.html' and change it to the following:

```
<!doctype html>  
<html lang="en">  
<head>  
<meta charset="utf-8">  
<title>Ch21Ex200</title>  
<base href="/">  
  
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" integrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous"> <script src="https://code.jquery.com/jquery-3.1.1.slim.min.js" integrity="sha384-A7FZj7v+d/sdmMqp/nOQwliLvUsJfdHW+k9Omga/EheAdgtzNs3hpfrag6Ed950n" crossorigin="anonymous"></script> <script src="https://cdnjs.cloudflare.com/ajax/libs/tether/1.4.0/js/tether.min.js" integrity="sha384-DztdAPBWRXSA/3eYEEUWrWCy7G5KFbe8fFjk5JAIXUYHKkDx6Qin1DkWx51bBrb" crossorigin="anonymous"></script> <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/js/bootstrap.min.js" integrity="sha384-vBWWzIJ8ea9aCX4pEW3rVHjgit7zpkNpZk+02D9phzyeVkJE+jo0ieGizqPLForn" crossorigin="anonymous"></script>
```

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

● Step 5 – Edit Module

Edit file ‘app.module.ts’ and change it to the following:

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

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

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

● Step 6 – Edit Component

Edit file ‘app.component.ts’ and change it to the following:

```
import { Component, ViewChild } from '@angular/core'; import { NgForm, RequiredValidator } from '@angular/forms';
@Component({
  selector: 'app-root',
  template: `
    <form #appointmentForm="ngForm" novalidate (ngSubmit) = "onSubmitForm(appointmentForm)"> <legend>Appointment</legend> <div
      class="form-group"> <label for="name">Name</label> <input type="text" class="form-control" name="name" placeholder="Name (last,
      first)" [(ngModel)]="_name" required> </div>

    <div class="form-group"> <label for="password">Password</label> <input type="password" class="form-control" name="password"
      placeholder="Password" [(ngModel)]="_password" required> </div>

    <div class="form-group"> <div class="form-check"> <div>
      <label>Appointment Time</label> </div>
      <label class="form-check-label"> <input type="radio" class="form-check-input" name="time" value="12pm" [(ngModel)]="_time"
        required> 12pm
      </label>
    </div>
    <div class="form-check"> <label class="form-check-label"> <input type="radio" class="form-check-input" name="time" value="2pm"
      [(ngModel)]="_time" required> 2pm
    </label>
    </div>
    <div class="form-check"> <label class="form-check-label"> <input type="radio" class="form-check-input" name="time" value="4pm"
      [(ngModel)]="_time" required> 4pm
    </label>
  </div>
  </form>
`}
```

```

</div>
</div>
<div class="form-group"> <label for="exampleTextarea">Ailment</label><textarea class="form-control" name="ailment" rows="3"
[(ngModel)]="_ailment" required ></textarea> </div>
    <button type="submit" class="btn btn-primary" [disabled]="!_appointmentForm.valid">Submit</button> Valid: {{ _appointmentForm.valid
}} Data: {{ _appointmentForm.value | json }}</form>
`,
styles: ['form { padding: 20px }', '.form-group { padding-top: 20px }']
})
export class AppComponent {
@ViewChild('appointmentForm') _appointmentForm: NgForm; _name: string = 'mark';
_password: string = "";
_time: string = "";
_ailment: string = "";

onSubmitForm() {
    alert("Submitting data:" + JSON.stringify(this._appointmentForm.value)); }
}

```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The file ‘index.html’ is modified to link to the bootstrap Css and JavaScript files.
2. The file ‘app.component’ does the following:
 - Sets up a form that is a template variable ‘appointmentForm’. The form fires method ‘onSubmitForm’ when it is submitted.
 - Sets up input fields and uses 2-way binding with the ‘ngModel’ directive to link the value of each field to an instance variable.
 - Contains the following markup in the template to enable or disable the Submit button:

```
<button type="submit" class="btn btn-primary" [disabled]="!_appointmentForm.valid">Submit</button>
```

- Displays form validity and values underneath.

21.11 Template Forms and CSS – Example

Let's go through creating an input form with color coding to form validation state. Green indicates valid input, red indicates invalid input. There is also code for error messages.

The image contains two side-by-side screenshots of a web form. Both forms have three fields: 'First Name' (with value 'Tom'), 'Last Name' (with value 'Brown'), and 'Email' (with value 'mar'). A 'Submit' button is at the bottom of each form. In the left screenshot, the 'Email' field has a green border, indicating valid input. In the right screenshot, the 'Email' field has a red border, and the text 'Invalid email' is displayed in red next to it, indicating invalid input.

- Step 1 – Build the App using the CLI

```
ng new ch21-ex300 --inline-template --inline-style
```

- Step 2 – Start Ng Serve

```
cd ch21-ex300  
ng serve
```

- Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- Step 4 – Edit Styles.

Edit the file ‘styles.css’ and change it to the following:

```
input.ng-valid {  
border-left: 5px solid #42A948; /* green */  
}
```

```
input.ng-invalid {  
border-left: 5px solid #a94442; /* red */  
}
```

```
.error {  
color: #ff0000;  
}
```

```
label {  
display: inline-block;  
width: 100px;  
}
```

```
button {  
border: 1px solid black;  
margin: 20px;  
}
```

● Step 5 – Edit Module

Edit file ‘app.module.ts’ and change it to the following:

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

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

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

● Step 6 – Edit Component

Edit file ‘app.component.ts’ and change it to the following:

```
import { Component, ViewChild } from '@angular/core'; import { NgForm, FormControl, RequiredValidator } from '@angular/forms';
@Component({
  selector: 'app-root',
  template: `
<form #f="ngForm" novalidate>
  <p><label>First Name</label><input name="fname" ngModel #fname="ngModel" required /> <span class="error" *ngIf="fname.touched
  && fname.hasError('required')">Required</span> </p>
  <p><label>Last Name</label><input name="lname" ngModel #lname="ngModel" required /> <span class="error" *ngIf="lname.touched
  && lname.hasError('required')">Required</span> </p>
  <p><label>Email</label><input name="email" ngModel #email="ngModel" required email /> <span class="error" *ngIf="email.touched
  && email.hasError('required')">Required</span> <span class="error" *ngIf="email.value && email.touched &&
  email.hasError('email')">Invalid email</span> </p>
  <button (click)="onSubmit()" [disabled]="!f.valid">Submit</button> </form>`,
  styles: []
})
export class AppComponent {
  onSubmit(){
    alert('Submitted');
  }
}
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The file ‘styles.css’ applies the required styles to the appropriate states. For example, setting the ‘ng-valid’ style to show a green indicator when the form control has valid data.
2. The file ‘app.component.ts’ contains logic to display error messages based on the form control

states.

21.12 Reactive Forms

You build the model objects for the form (they are the same ones as the template forms) then you bind them to the input controls in the template. So, you are building the form controls in your class then amending your template to link to those controls.

This gives you complete control over the form, its values and its validations. You can directly manipulate the model objects (for example change values) and the binding immediately takes affect synchronously. In fact, value and validity updates are always synchronous and under your control.

21.13 Reactive Forms – Module Setup

To use Angular template forms, your application module needs to import the Reactive Forms Module from the ‘@angular/forms’ node module.

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

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

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

21.14 Reactive Forms – Bind Template to Model

You create the html form and html inputs in the component's template and you create a form model in your component's class. Now you bind the two together using the following directives:

<form [formGroup]="registerForm">	Connects the form model with the form html in the template.
<fieldset formGroupName="address">	Connects the form group with the fieldset html in the template.
<input formControlName="name">	Connects the form control in the model with the form input html in the template.



21.15 Reactive Forms - Example

Let's go through creating a reactive form and see what is needed to make it work. This will be example 'ch21-ex400'.

• Step 1 – Build the App using the CLI

```
ng new ch21-ex400 --inline-template --inline-style
```

• Step 2 – Start Ng Serve

```
cd ch21-ex400  
ng serve
```

• Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see 'app works!'.

• Step 4 – Edit Styles

Edit file 'styles.css' and change it to the following:

```
input.ng-valid {  
border-left: 5px solid #42A948; /* green */  
}  
  
input.ng-invalid {  
border-left: 5px solid #a94442; /* red */  
}  
  
.error {  
color: #ff0000;  
}  
  
label {  
display: inline-block;  
width: 100px;  
}  
  
button {  
border: 1px solid black;  
margin: 20px;  
}
```

• Step 5 - Edit Module

Edit file 'app.module.ts' and change it to the following:

```
import { BrowserModule } from '@angular/platform-browser'; import { NgModule } from '@angular/core';  
import { ReactiveFormsModule } from '@angular/forms';  
  
import { AppComponent } from './app.component';  
  
@NgModule({
```

```

declarations: [
  AppComponent
],
imports: [
  BrowserModule,
  ReactiveFormsModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

● Step 6 – Edit Component

Edit file ‘app.component.ts’ and change it to the following:

```

import { Component, OnInit } from '@angular/core'; import { FormGroup, FormControl, FormControlName, Validators } from
'@angular/forms';
@Component({
selector: 'app-root',
template: `

<form #form [formGroup]="formGroup" (ngSubmit)="onSubmit(form)" novalidate>
  <label>Name:</label>
  <input formControlName="name">
  <br/>
  <label>Location:</label>
  <input formControlName="location">
  <br/>
  <input type="submit" value="Submit" [disabled]="!formGroup.valid">
</form>
`,
styles: []
})
export class AppComponent implements OnInit{



formGroup: FormGroup;

ngOnInit(){
  this.formGroup = new FormGroup({
    name: new FormControl("", Validators.required), location: new FormControl("", Validators.required) });
}

onSubmit(form: FormGroup){
  alert('sumit');
}
}

```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The file ‘styles.css’ sets up the css styles.
2. The file ‘app.component.ts’ contains the html in the template for the form.
3. The file ‘app.component.ts’ initializes the model: a form group with form controls in the ngOnInit method, when the component initializes.
4. The file ‘app.component.ts’ links the html in the template to the model.

- It links html form to form group using the following:

```
<form #form [formGroup]="formGroup" (ngSubmit)="onSubmit(form)" novalidate>
```

- It links html input to form control using the following:

```
<input formControlName="name">
```

21.16 Reactive Forms – Form Builder

This class is designed to help you build the form model with less code. Inject the FormBuilder into your component's class and use its following methods:

Method	Purpose	Arguments	Returns
group	Create a form group.	configuration object, extra parameters (eg validators, async validators)	FormGroup
control	Create a form control.	current form state (value / disabled status), array of validators, array of async validators	FormControl
array	Create a form array.	configuration objects (array), validator, async validator	FormArray

We will start using FormBuilder in the upcoming examples.

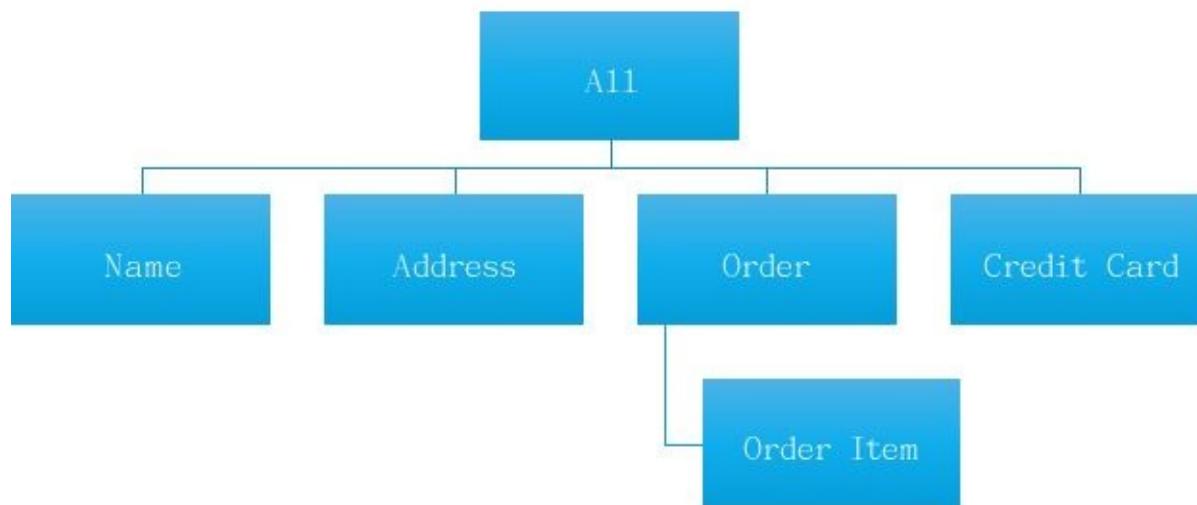
21.17 Reactive Forms - Form Group Nesting

Sometimes our forms consist of multiple different elements. For example, if you were entering a customer order, this information could be structured in the following manner: Name Address Order

- Order Items

Credit Card Info Each element of the above can contain one or more form controls so we need the ability to manage each. This is where form groups come in.

In this case you could have the following hierarchy of form groups:



21.18 Reactive Forms - Form Group Nesting – Example

This example enables the user to enter and submit an order: the customer name, customer address and a list of items. This will be example ‘ch21-ex500’.

The screenshot shows a user interface for entering an order. It consists of three main sections: 'Name', 'Address', and 'Items'.
The 'Name' section contains fields for First Name ('Mark') and Last Name ('Clow').
The 'Address' section contains fields for Address #1 ('2387 Welton Drive'), Address #2, City ('Milton'), State ('GA'), and Zip ('30342').
The 'Items' section contains a table with two rows. The first row has 'Name: Brush' and 'Qty: 1'. The second row has 'Name: Toilet Paper' and 'Qty: 1'. Both rows have a 'Price' field with a dropdown menu open, showing '1.99'.

• Step 1 – Build the App using the CLI

```
ng new ch21-ex500 --inline-template --inline-style
```

• Step 2 – Start Ng Serve

```
cd ch21-ex500  
ng serve
```

• Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

• Step 4 – Edit Module

Edit file ‘app.module.ts’ and change it to the following:

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

● Step 5 – Edit Component Class

Edit file ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit } from '@angular/core'; import { FormGroup, FormArray, FormBuilder, Validators } from '@angular/forms';
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styles: ['div { background-color: #f2f2f2; padding: 15px; margin: 5px }', 'p { margin: 0px }'
])
)

export class AppComponent implements OnInit {

  public _parentForm: FormGroup;
  public _name: FormGroup;
  public _addr: FormGroup;
  public _items: FormArray;

  constructor(private _fb: FormBuilder){}

  ngOnInit() {
    this._name = this._fb.group({
      fname: ['', [Validators.required]],
      lname: ['', [Validators.required]]
    });
    this._addr = this._fb.group({
      addr1: ['', [Validators.required]],
      addr2: [''],
      city: ['', [Validators.required]],
      state: ['', [Validators.required]],
      zip: ['', [Validators.required, Validators.minLength(5), Validators.maxLength(5)]]}, {});
    this._items = this._fb.array([
      this.createItemFormGroup()
    ]);
    this._parentForm = this._fb.group({
      name: this._name,
      addr: this._addr,
      items: this._items
    });
  }

  createItemFormGroup(){
    return this._fb.group({
      name: ['', Validators.required],
      qty: ['1', Validators.required],
      price: ['', Validators.required]
    });
  }

  addItem(){
    this._items.push(this.createItemFormGroup());
  }

  deleteItem(index){
    delete this._items[index];
  }

  onSubmit(form: FormGroup){

  }
}
```

```
    alert('Submitted');
}
}
```

● Step 6 – Edit Component Template

Edit file ‘app.component.html’ and change it to the following:

```
<form [formGroup]="_parentForm" novalidate (ngSubmit)="onSubmit(parentForm)"> <div formGroupName="name">
  <b>Name</b>
  <br/>
  <label>First Name
  <input type="text" formControlName="fname"> <small *ngIf="!_name.controls.fname.touched && !_name.controls.fname.valid">Required.</small> </label>
  <br/>
  <label>Last Name
  <input type="text" formControlName="lname"> <small *ngIf="!_name.controls.lname.touched && !_name.controls.lname.valid">Required.</small> </label>
</div>
<br/>
<div formGroupName="addr">
  <b>Address</b>
  <br/>
  <label class="left">Address #1
  <input type="text" formControlName="addr1"> <small *ngIf="!_addr.controls.addr1.touched && !_addr.controls.addr1.valid">Required.</small> </label>
  <br/>
  <label>Address #2
  <input type="text" formControlName="addr2"> </label>
  <br/>
  <label>City
  <input type="text" formControlName="city"> <small *ngIf="!_addr.controls.city.touched && !_addr.controls.city.valid">Required.</small> </label>
  <br/>
  <label>State
  <select formControlName="state"> <option>AL</option>
  <option>GA</option>
  <option>FL</option>
  </select>
  <small *ngIf="!_addr.controls.state.touched && !_addr.controls.state.valid">Required.</small> </label>
  <br/>
  <label>Zip
  <input type="number" formControlName="zip"> <small *ngIf="!_addr.controls.zip.touched && !_addr.controls.zip.valid">Required.</small> </label>
</div>
<br/>
<div formArrayName="items">
  <b>Items</b>
  <br/>
  <p [formGroupName]="i" *ngFor="let item of _items.controls; let i=index"> <label>Name:&nbsp;<input type="text" formControlName="name" size="30"> <small *ngIf="item.controls.name.touched && !item.controls.name.valid">Required.</small> </label>
  <label>Qty:&nbsp;<input type="number" formControlName="qty" min="1" max="10"> <small *ngIf="item.controls.qty.touched && !item.controls.qty.valid">Required.</small> </label>
  <label>Price:&nbsp;<input type="number" formControlName="price" min="0.01" max="1000" step=".01"> <small *ngIf="item.controls.price.touched && !item.controls.price.valid">Required.</small> </label>
```

```
</p>
</div>
<br/>
<div>
  <input type="button" value="Add Item" (click)="addItem()"/> <input type="submit" value="Submit" [disabled]="_parentForm.valid"/>
</div>
</form>
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. We have at least four fixed FormGroup objects: one for the name, one for the address, one for the first item and another for the parent form.
2. The FormArray contains one FormGroup object but it can contain other FormGroup objects if the user clicks on the ‘Add Item’ button.
3. The overall form validity still controls the enablement and disablement of the Submit button.

21.19 Validators

Angular provides some validators for our forms. You can add multiple validators to the same FormControl (an item in the FormGroup). Let's cover them.

• Required Validation

```
this.form = fb.group({  
  'name': ['', Validators.required],  
});
```

• Minimum Length Validation

```
this.form = fb.group({  
  'name': ['', Validators.required, Validators.minLength(4)]  
});
```

• Maximum Length Validation

```
this.form = fb.group({  
  'name': ['', Validators.required, Validators.maxLength(4)]  
});
```

21.20 Combining Multiple Validators

The Validators class provides the ‘compose’ method to allow the user to specify multiple validators to a control.

```
constructor(private fb: FormBuilder){  
  this.form = fb.group({  
    'name': ['', Validators.compose( [Validators.required, Validators.maxLength(6)] ) ],  
  });  
}
```

21.21 Custom Validation

The Angular Forms module allows you to create a custom class to validate your input. Note that the validation method is static and that it returns a Validation Result only when there is an error. If everything is ok, this method returns a null. This custom class can be used when specifying the field in the FormBuilder and it can also be used in the Component Template to provide a visual cue.

21.22 Custom Validation – Example Code

This component won't allow the user to enter 'Mercedes'. This will be example 'ch21-ex600'.

Make: Model:
Submit

Make: Model:
Submit

- **Step 1 – Build the App using the CLI**

```
ng new ch21-ex600 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch21-ex600  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- **Step 4 – Edit Styles**

Edit the file 'styles.css' and change it to the following:

```
input.ng-valid {  
border-left: 5px solid #42A948; /* green */  
}  
  
input.ng-invalid {  
border-left: 5px solid #a94442; /* red */  
}
```

- **Step 5 – Edit Module**

Edit the file 'app.module.ts' and change it to the following:

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

```
export class AppModule { }
```

● Step 6 – Edit Component

Edit the file ‘app.component.ts’ and change it to the following:

```
import { Component, OnInit } from '@angular/core'; import { AbstractControl, FormGroup, FormControl, FormControlName, Validators } from '@angular/forms';
export function validateNotMercedes(control: AbstractControl) {
  return (control.value.toLowerCase() != 'mercedes') ?
    null :
    { validateNotMercedes: {
      valid: false
    }
  }
}

@Component({
  selector: 'app-root',
  template: `
    <form #form [formGroup]="formGroup" (ngSubmit)="onSubmit(form)" novalidate>
      <label>Make:</label>
      <input formControlName="make">
      <br/>
      <label>Model:</label>
      <input formControlName="model">
      <br/>
      <input type="submit" value="Submit" [disabled]="${!formGroup.valid}">
    </form>
  `,
  styles: []
})
export class AppComponent implements OnInit {

  formGroup: FormGroup;

  ngOnInit(){
    this.formGroup = new FormGroup({
      make: new FormControl('', [Validators.required, validateNotMercedes]),
      model: new FormControl('', Validators.required)
    });
  }

  onSubmit(form: FormGroup){
    alert('sumit');
  }
}
```

● Exercise Complete

Your app should be working at localhost:4200. Note the following:

1. The code in file ‘app.component.ts’ exports the ‘validateNotMercedes’ function to validate the make. Note that it returns a null to indicate validity, otherwise it returns an object with the property ‘valid’ set to false.
2. The code in file ‘app.component.ts’ sets up the form group using the FormControl objects. Notice how here the ‘make’ FormControl specifies the ‘validateNotMercedes’ function as a

validator.

Pipes

22.1 Introduction

Pipes have been around since AngularJS. They are useful at transforming data, especially when the same transformation is used throughout the application.

Pipes make it easy to add these transformations into your Component Template.

22.2 Angular Pipes

Angular 4 includes several pipes to add to your template. You don't need to import them, add them as directives or anything. Just start using them, they are useful.

● Lowercase

```
Lowercase: {{ "The Quick Brown Fox Jumped Over The Lazy Dogs" | lowercase }}
```

Produces:

```
Lowercase: the quick brown fox jumped over the lazy dogs
```

● Uppercase

```
Uppercase: {{ "The Quick Brown Fox Jumped Over The Lazy Dogs" | uppercase }}
```

Produces:

```
Uppercase: THE QUICK BROWN FOX JUMPED OVER THE LAZY DOGS
```

● Currency

```
Currency: {{ 2012.55 | currency }}
```

Produces:

```
Currency: USD2,012.55
```

● UK Pound Currency

```
UK Pound Currency: {{ 2012.55 | currency: 'gbp':true }}
```

Produces:

```
UK Pound Currency: £2,012.55
```

● Percentage

```
Percentage: {{ 0.5 | percent }}
```

Produces:

```
Percentage: 50%
```

● Date

```
Date: {{ dt | date }}
```

Produces:

● Short Date

```
Short Date: {{ dt | date:shortdate }}
```

Produces:

```
Short Date: Jul 12, 2017
```

● Special Date Format

```
Special Date Format: {{ dt | date:'yMMMMEEEEd' }}
```

Produces:

```
Special Date Format: Wednesday, July 12, 2017
```

Pre-defined date formats:

Name	Format	Example (English/US)
medium	yMMMdjms	Sep 3, 2010, 12:05:08 PM
short	yMdjm	9/3/2010, 12:05 PM
fullDate	yMMMMEEEEd	Friday, September 3, 2010
longDate	yMMMMd	September 3, 2010
mediumDate	yMMMd	Sep 3, 2010
shortDate	yMd	9/3/2010
mediumTime	jms	12:05:08 PM
shortTime	jm	12:05 PM

Date format elements (can be combined):

Name	Format	Text Form Full	Text Form Short	Numeric Form	Numeric Form 2 Digit
era	G	GGGG	G		
year	y			y	yy
month	M	MMMM	MMM	M	MM
day	d			d	dd

weekday	E	EEEE	EEE		
hour	j			j	jj
12 hour	h			h	hh
24 hour	H			H	HH
minute	m			m	MM
second	s			s	ss
timezone	z / Z	z	Z		

22.3 Angular Pipes – Example

This component displays information using the variety of Angular pipes. This will be example ‘ch22-ex100’.

```
Lowercase: the quick brown fox jumped over the lazy dogs  
Uppercase: THE QUICK BROWN FOX JUMPED OVER THE LAZY DOGS  
Currency: USD2,012.55  
UK Pound Currency: £2,012.55  
Percentage: 50%  
Date: Jul 12, 2017  
Short Date: Jul 12, 2017  
Special Date Format: Wednesday, July 12, 2017
```

● Step 1 – Build the App using the CLI

```
ng new ch22-ex100 --inline-template --inline-style
```

● Step 2 – Start Ng Serve

```
cd ch22-ex100  
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Edit Component

Edit the file ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  template: `<p>  
    Lowercase: {{ "The Quick Brown Fox Jumped Over The Lazy Dogs" | lowercase }}  
  </p>  
  <p>  
    Uppercase: {{ "The Quick Brown Fox Jumped Over The Lazy Dogs" | uppercase }}  
  </p>  
  <p>  
    Currency: {{ 2012.55 | currency }}  
  </p>  
  <p>  
    UK Pound Currency: {{ 2012.55 | currency: 'gbp':true }}  
  </p>  
  <p>  
    Percentage: {{ 0.5 | percent }}  
  </p>  
  <p>  
    Date: {{ dt | date }}  
  </p>  
  <p>
```

```
Short Date: {{ dt | date:shortdate }}  
</p>  
<p>  
Special Date Format: {{ dt | date:'yMMMMEEEEd' }}  
</p>  
,  
styles: []  
})  
export class AppComponent {  
dt = new Date();  
}
```

● Exercise Complete

The app should be working and displaying the formatted data.

22.4 Custom Pipes

● Introduction

Writing custom pipes is straightforward. However, some new syntax is introduced so there are a few things to remember.

● Component that Uses Pipe

The Component that uses a Custom Pipe needs to declare the Pipe Class both as an import and specify it in the '@Component' annotation.

● Pipe Class

The Pipe Class is prefixed by a '@Pipe' annotation. It also needs to import the Pipe and PipeTransform, as well as implement the PipeTransform interface.

22.5 Generate Custom Pipe using CLI

You can get the Angular CLI command ‘ng generate pipe <pipe name>’ to generate a custom pipe in a CLI-generated project. Ignore the ‘<pipe name>.pipe.spec.ts’ file (its for testing) but edit the ‘<pipe name>.pipe.ts’ file.

```
ng generate pipe reverse
```

```
installing pipe
create src/app/reverse.pipe.spec.ts
create src/app/reverse.pipe.ts
update src/app/app.module.ts
```

Your custom pipe should be a TypeScript class that implements the PipeTransform interface:

```
interface PipeTransform {
  transform(value: any, ...args: any[]): any
}
```

22.6 Custom Pipes - Example Code

This component allows the user to reverse some text. It also has an optional argument, the number of spaces to be put between each character of the reversed text. This will be example ‘ch22-ex200’.

```
My name is eniaC leahciM
```

```
My name is e n i a C l e a h c i M
```

- **Step 1 – Build the App using the CLI**

```
ng new ch22-ex200
```

- **Step 2 – Start Ng Serve**

```
cd ch22-ex200  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

- **Step 4 – Generate the Custom Pipe using the CLI**

```
ng generate pipe reverse
```

- **Step 5 – Edit Pipe**

Edit the file ‘reverse.pipe.ts’ and change it to the following:

```
import { Pipe, PipeTransform } from '@angular/core';  
@Pipe({  
name: 'reverse'  
})  
export class ReversePipe implements PipeTransform {  
  
transform(value: any, args?: any): any {  
    let spaces = 0;  
    if (args){  
        spaces = parseInt(args);  
    }  
    let reversed = "";  
    for (let i=value.length-1;i>=0;i--){  
        reversed += value.substring(i, i+1); reversed += Array(spaces + 1).join(' '); }  
    return reversed;  
}  
}
```

- **Step 6 – Edit Component**

Edit the file ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { ReversePipe } from './reverse.pipe';
@Component({
selector: 'app-root',
template: `
<p>My name is {{name | reverse}}</p>
<p>My name is {{name | reverse:5}}</p>
`,
styles: []
})
export class AppComponent {
name: string = 'Michael Caine';
}
```

● Exercise Complete

The app should be working and displaying the formatted data. Please note the following:

1. The class ‘ReversePipe’ implements the PipeTransform interface as any pipe would.
2. The class ‘ReversePipe’ adds extra spaces by using the Array object constructor. If you supply a single value to the constructor, it sets the array length to that value. The ‘join’ method then specifies a string to separate each element of the array.

23 Zones & Change Detection

23.1 Introduction

Angular uses a JavaScript Module Called Zone.js. Its purpose is to produce an execution context that persists across asynchronous tasks. Currently the browser DOM and JavaScript have a limited number of asynchronous activities, activities such as DOM events, promises, and server calls. Zone.js can intercept these activities and to give your code the opportunity to take action before and after the asynchronous activity completes. This is useful when you need to see all the information pertinent to that task, especially when an error occurs.

Changes occur as a result of something:

A DOM event. Example: someone clicks on something.

Communication. Example: the browser gets data back from the server.

A timer event happens. Example: refresh every 10 seconds.

23.2 Change Detection

We covered Model View Controller briefly in Chapter ‘**Error! Reference source not found.**’. Remember that the Model is the data and the View displays the Data in the Model.

The purpose of Change Detection in Angular is to look for changes in the Model and to ensure that the View (i.e. the DOM) is kept up to date with it. Change Detection can get complicated as it needs to figure out when the View needs to be redrawn when code is running.

Here is an example of some code that changes the Model. An http call is made to the server and data is returned. A customer list is updated in the Model. So now this Change needs to be Detected by Angular and the UI needs to be refreshed. How does Angular know that something may have changed and it should look for Changes? Because NgZone tells it!

```
@Component()
class App implements OnInit{
customers:Customer[] = [];
constructor(private http: Http) {}
ngOnInit() {
  this.http.get('/customers')
    .map(res => res.json())
    .subscribe(customers => this.customers = customers);
}
```

23.3 NgZone is Zone.js for Angular

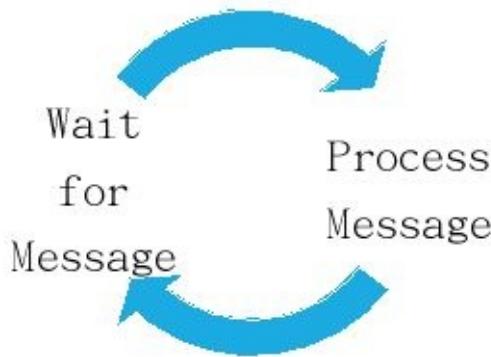
The NgZone class is a wrapper around the zone.js framework. The dependency injector can also pass in the zone through Constructor Injection.

23.4 How Does NgZone / Zone.js Notice Events?

• Event Loop and Messages

JavaScript has a concurrency model based on an "event loop". JavaScript runtime contains a message queue, which is a list of messages to be processed. Messages are taken out of the queue and processed by the browser UI thread.

So, the browser basically works in a loop, picking up and processing messages to do things.



• Browser UI Thread

The browser UI thread is a single thread that updates the user interface by running the Event Loop code, processing messages. Each message is processed completely before any other message is processed. Only one thread is used to update the user interface (the document that the user views). If the browser UI thread is overloaded the browser displays this message (or one similar) to the



• Monkey Patching

With NgZone/Zones.js, system JavaScript code is Monkey Patched (when it has to be) so it hooks into the “event loop” code to see what is happening with the messages being processed. This enables it to provide additional information about events occurring or code being called in the Zone, for example an asynchronous server call completing.

NgZone emits ‘onTurnStart’ and ‘onTurnEnd’ events to inform observers of when something is about to occur and when something has occurred.

23.5 NgZone Is Used by Angular to Notice Events

NgZone is used by Angular 4 to look for events that require Change Detection. In core Angular 4 code, Angular listens for the NgZone ‘onTurnDone’ event. When this event fires, Angular 4 performs Change Detection on the model and updates the UI.

Monkey Patches A monkey patch is a way for a program to extend or modify supporting system software locally. In terms of Angular 4 and Zone.js, Zone will monkey-patch JavaScript core code when it has to in order to provide execution information.

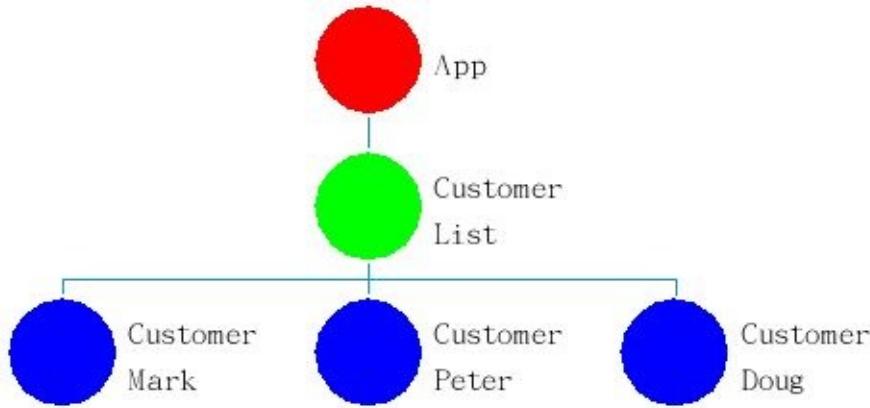
23.6 How Does Angular Perform Change Detection?

Angular 4 applications are built as multiple Lego-like Components, with a tree-like hierarchy. You have the main Application Component, then you have Sub Components and so forth.

Component UI



Component Tree



Each Angular 4 Component has its own change detector for its variables. You don't see it happen but Angular creates the Change Detector classes when it runs.

So, if you have a tree of Components, then you have a tree of Change Detectors. Core angular code scans the tree for changes (calling each change detector) from the bottom up to see what changed.

23.7 Mutable Objects vs Immutable Objects

Mutable objects can change. Immutable objects cannot. Obviously, the Change Detection is quicker when it runs against objects that don't change. If you want your Angular 4 code to run faster, start looking into using Immutable objects for things that don't change.

23.8 Using NgZone

We know that NgZone is used for Change Detection in Angular 4. NgZone is a class and it is useful to us (as well as to the system Angular code) because it allows us to run asynchronous processes inside or outside the Angular zone.

• Run Methods Inside Angular Zone

They update the Angular UI.

They run slower.

We run asynchronous processes inside the Angular zone when we need the Change Detection to occur and to have the UI constantly updated. To run asynchronous processes inside the Angular Zone this we call the ‘run’ method in the injected NGZone object, passing in the ‘process’ function.

• Run Functions Outside Angular Zone

They don’t update the Angular UI.

They run faster.

We run asynchronous processes outside the Angular zone when we don’t need the Change Detection to occur and we don’t want the UI constantly updated. This may seem unnecessary but when ultimate performance is required then this should be considered. To run asynchronous processes outside the Angular Zone this we call the ‘runOutsideAngular’ method in the injected NGZone object, passing in the ‘process’ method.

23.9 Example Code – Running Asynchronous Code within Angular Zone.

It is based off the default Angular 4 Typescript Plunker Application.

• File ‘app.ts’.

```
1 import {Component, NgZone} from 'angular2/core'

@Component({
  selector: 'my-app',
  providers: [],
  template:
    <button (click)="doCountInAngular()">Count</button>
    {{counter}}
  ,
  directives: []
})
export class App {
 2 constructor(private _ngZone: NgZone) {
    this.counter = 0;
  }

3 doCountInAngular(){
  this._ngZone.run(() => { this.initiateCount()});
}

4 initiateCount(){
  this.counter = 0;
  var intervalFn = () => { this.updateCount()};
  this.interval = setInterval(intervalFn), 500;
}

updateCount(){
  this.counter++;
  if (this.counter > 1000){
    clearInterval(this.interval);
    alert('done!!!!');
  }
  for (var i=0;i<10;i++){
    console.log(this.counter + " " + i);
  }
}
}
```

- | | |
|---|--|
| 1 | Import NgZone. |
| 2 | Use Constructor Injection to inject instance of NgZone. |
| 3 | This method is fired by ‘Count’ button. It runs the ‘initiateCount’ method using the injected NgZone. Notice it calls the method ‘run’ to run the method inside the injected Angular Zone. |
| 4 | Methods ‘initiateCount’ and ‘updateCount’ produce console logs as an asynchronous task, using the interval timer. They update the counter and finish counting when the counter is over 1000. |

When you run this app and click on ‘Count’ you see the Counter updating 1,2,3,4 all the way up to 1000 and the alert appears. The User Interface shows the count. This is because the count is being performed in a function inside the Angular Zone, with NgZone watching the events and causing Change Detection. The Change Detection detects that the count variable has changed and updates

the UI.

Count 0

23.10 Example Code – Running Asynchronous Code Outside Angular Zone.

It is based off the default Angular 4 Typescript Plunker Application.

• File ‘app.ts’

```
1 import {Component, NgZone} from 'angular2/core'  
2  
@Component({  
  selector: 'my-app',  
  providers: [],  
  template:  
    <button (click)="doCountOutsideAngular()">Count</button>  
    {{counter}}  
  ,  
  directives: []  
})  
export class App {  
  constructor(private _ngZone: NgZone) {  
    this.counter = 0;  
  }  
  
3 doCountOutsideAngular(){  
  // this._ngZone.run(() => { this.initiateCount()});  
  this._ngZone.runOutsideAngular(() => { this.initiateCount()});  
}  
  
4 initiateCount(){  
  this.counter = 0;  
  var intervalFn = () => { this.updateCount()};  
  this.interval = setInterval(intervalFn, 500);  
}  
  
updateCount(){  
  this.counter++;  
  if (this.counter > 1000){  
    clearInterval(this.interval);  
    alert('done!!!');  
  }  
  for (var i=0;i<10;i++){  
    console.log(this.counter + " " + i);  
  }  
}
```

Count 0

1	Import NgZone.
2	Use Constructor Injection to inject instance of NgZone.
3	This method is fired by ‘Count’ button. It runs the ‘initiateCount’ method using the injected NgZone. Notice it calls the method ‘runOutsideAngular’ to run the method outside the injected Angular Zone.
4	Methods ‘initiateCount’ and ‘updateCount’ produce console logs as an asynchronous task, using the interval timer. They update the counter and finish counting when the counter is over 1000.

When you run this Application and click on ‘Count’ you don’t see the Counter change. The User Interface shows the count 0 until the alert appears. This is because the count is being performed in a function outside the Angular Zone, without NgZone watching the events and causing Change Detection. Notice how it’s a bit faster?

Count 0

24 Testing

24.1 Introduction

This book is mainly about how to get started and productive with Angular. However, it would be incomplete without at least introducing ways to test the code that you write. The testing framework is quite complicated so don't expect to know everything about it after reading this chapter. So, let's introduce some of the concepts then go into the details of writing code to automate the testing of a project that was generated with the Angular CLI.

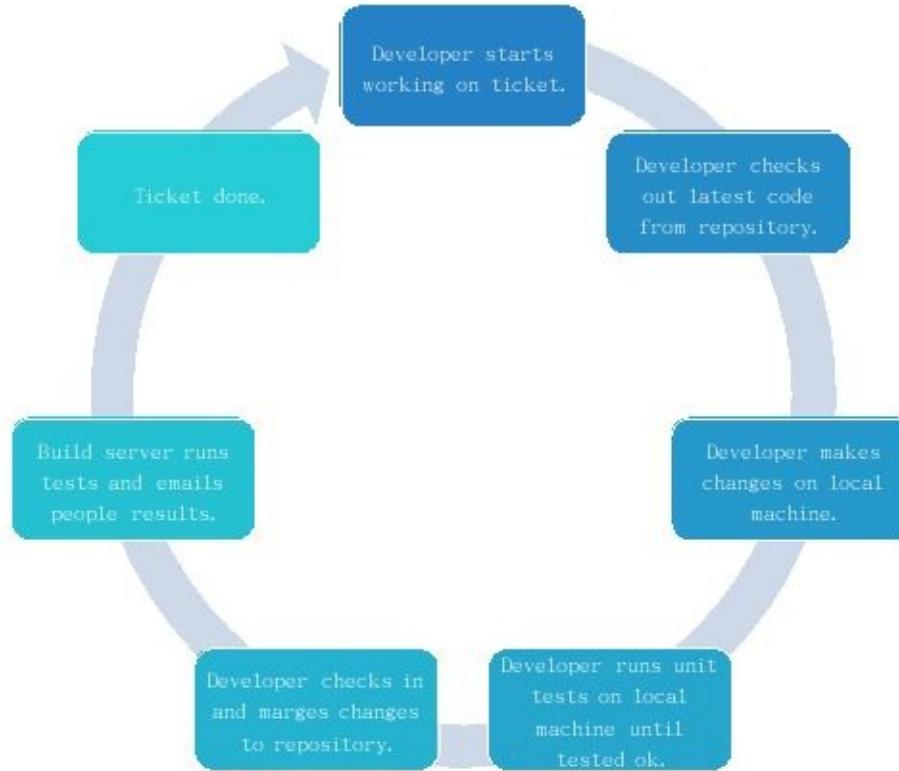
24.2 Unit Testing

Unit testing is the testing of the smallest possible units of the application, either in a manual or automated form. The point of unit testing is to ensure the following: that the code is performing as expected and that new code does not break old code. The process of test-driven development is the development of code in the following order: writing the test code (the test harness), writing application code to pass the tests, cleaning up and refactoring application code to pass coding standards and check it still passes the tests. This process should be applied to smaller units of code and this process should be repeated frequently. So, unit tests are essential in the modern process of software development.

24.3 Continual Integration

Software development uses the process of developers checking out the latest code from a central repository and working on it. After work is completed (and the code is tested), the developers check in the completed code. Continual integration is the process of integrating (or merging) all developer code into a shared codebase several times a day. Integrating code as often as possible highlights merging issues quickly and avoids larger code incompatibilities. The aim is to check out code for as short as time as possible and to check in and integrate the changes as soon as possible before someone changes things too much in the meantime!

This is a (very general) diagram of the development process working. It does not take into account code branches, merging issues etc.



24.4 Automated Unit Tests

Automating the unit tests takes some upfront work but in the long run saves people time. Automated tests can find problems very quickly and they should be used at least in the following two occasions:

1. When a user is about to checking code changes he or she should invoke the automated unit tests on his or her local machine to ensure that the code is working as expected.
2. The build server should invoke the automated unit tests whenever a developer checks in code changes. The build server should track the results of these tests and let people know if they passed or failed.

24.5 Integration Testing

Integration testing occurs after unit testing has occurred. It tests the combined code, simulating a user running the complete application. This is a higher-level of testing, testing areas of the system without knowing anything about its structure or implementation. It ensures the application works as expected as for the user, that the component parts of the application work together.

24.6 Mocks

Your Angular application is made up of components which have dependencies. You need to develop your unit tests so that they test units of code in isolation. For example, if you want to test a component that uses a service to get data from a server, you probably need to test the component and service separately. You will probably need to do the following:

1. Write code to test the component, injecting it with a mock (or dummy) version of the service that acts in a pre-determined manner. The mock service simulates an output from the service. That way we can test that the component processes the output from the service as expected.
2. Wrote code to test the service, injecting it with a mock (or dummy) version of the communication layer (back end) that talks to the server (for example http service). The mock communication layer simulates connections and these mock connections have the ability to simulate a response from the server. That way we don't need a real server and we can test that the component processes the output from the server as expected.

● Asynchronous Operations

One thing that complicates the testing is that a lot of the code we are testing is asynchronous, it does not block and wait until the code completes. The testing library (and your testing code) has code to deal with asynchronous operations and this complicates things even further. Sometimes the code has to be run in a special asynchronous zone to simulate these operations.

24.7 Karma

● Introduction

Karma is an automated test runner that was developed by Angular team during development of AngularJS. It is able to run unit tests fast and on real browsers. So, you use Karma to start a server on which a group of Jasmine tests are run.

Karma opens a web browser and automates it to perform tests. You can see it running the tests in that browser. Sometimes it even leaves the browser open after the tests.

● Karma Configuration

When you build your CLI project, it creates the file ‘karma.conf.js’ to allow you to configure karma for the project. Configuration options include the base path, which test files to include / exclude, autowatch files, which browsers to test on, colors, timeouts, testing framework (e.g. Jasmine), server hostname and port (e.g. localhost:8080), logging, plugins, preprocessors, reporters, single run etc.

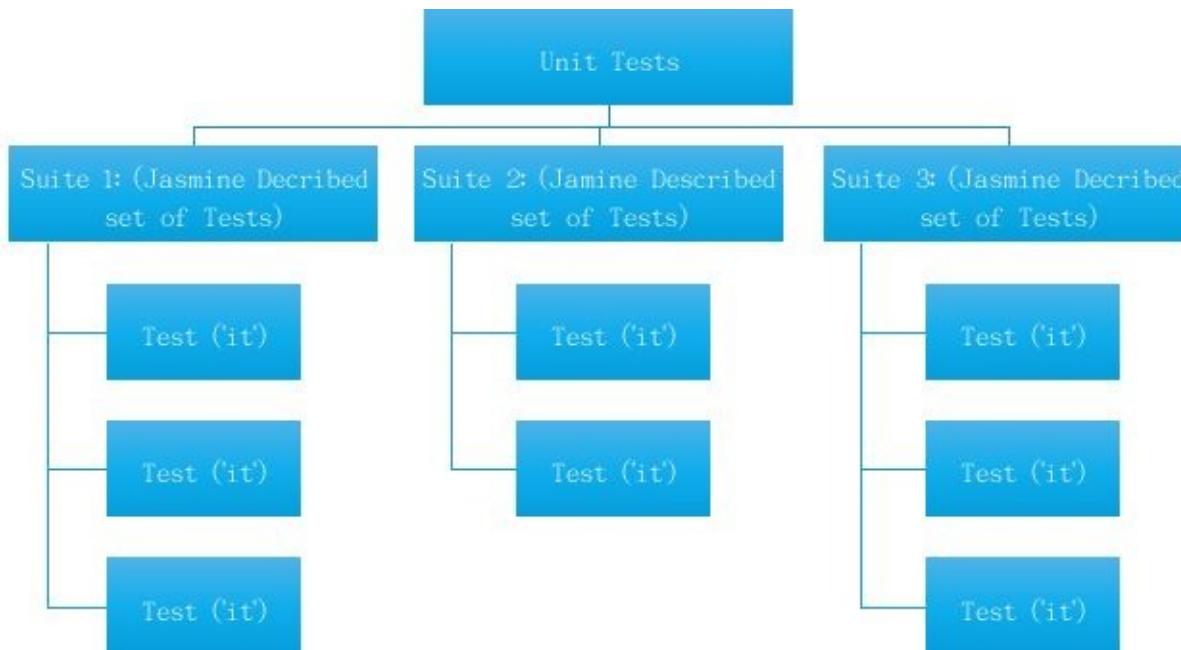
The single run configuration is useful if you want to leave your browser open after tests finish. This is sometimes useful if there is a failure and you need to see what went on by looking at the browser developer tools.

24.8 Jasmine

Jasmine is an open-source automated unit testing JavaScript framework that is very commonly-used with Angular and other JavaScript libraries.

● Jasmine - Structure

When you write Jasmine tests, you have to follow its way of doing things. You write sets of described tests in ‘.spec.ts’ files (one or more per file) and each described set of tests contains multiple tests. Each test does something with the code it tests, gets a result then checks the result for validity.



Jasmine unit tests have a two-level structure:

1. A 'described' suite of tests. Developers use the 'describe' function to setup a suite of tests executed together. For example, 'connectivity tests'. Notice that the 'describe' method is also used to provide the dependencies for the object to be tested. **Variables declared in a 'describe' are available to any 'it' block of code inside the suite.**
2. 'It' blocks of code that perform test inside the 'described' suite of tests. Developers use the 'it' function to setup a test where code is performed and a comparison occurs between the expected and actual results. Developers use the 'expect' method inside a test to set result expectations. If they are met the code passes the test, if not it fails. Jasmine uses 'matchers' to compare expected and actual results, for example: expect(a).toEqual(12);

```
describe("[The class you are about to test]", () => {  
  
beforeEachProviders(() => {  
    return [Array of dependencies];  
});  
it("test1", injectAsync([TestComponentBuilder], (tcb: TestComponentBuilder) => {  
    return tcb.createAsync([The class you are about to test]).then((fixture) => {  
  
// test code ...  
// expect a result  
});  
});
```

```

}););

it("test2", injectAsync([TestComponentBuilder], (tcb: TestComponentBuilder) => {
  return tcb.createAsync([The class you are about to test]).then((fixture) => {
    // test code ...
    // expect a result
  });
}));

});

```

● Jasmine - Concepts

These are the Jasmine concepts you need to learn and the code keyword associated with each concept. Take a look at the code for a basic Jasmine test (underneath the table) and see how it corresponds to the concepts in the table.

Name	Description	Code Keyword
Suite	Described set of tests that corresponds to an area of code that needs testing. There is usually one suite of tests per unit test file e.g. 'app.component(suite.ts'. However, you can have more than one described set of tests in a unit test file.	describe
Spec	A test. Performs code and checks the result against expectations. There can be multiple Specs in a Suite.	it
Expectations	Used within a test to check the result.	expect
Matchers	Used by an expectation to specify the expectation as a rule.	toBe, toEqual, toBeNull, toContain, toThrow, toThrowError etc

```

describe("CalcUtils", function() { // suite
  //Spec for sum operation
  it("2 plus 2 equals 4", function() { // spec
    var calc = new CalcUtils();
    expect(calc.sum(2,2)) // expect .toEqual(4); // matcher });
  });

  //Spec for sum operation with decimal
  it("2.5 plus 2 equals 4.5", function() { // spec
    var calc = new CalcUtils();
    expect(calc.sum(2.5,2)) // expect .toEqual(4.5); // matcher });
  });
});

```

24.9 Jasmine – Setup and Teardown

You have a suite of tests (described) which contains one or more tests (specs). Quite often the specs will be quite similar and will be testing the same object again and again. This can cause repetitive code because in every spec you would be instantiating the object to test, testing it and then destroying it. You can see this in the code directly above this paragraph.

Jasmine offers a solution to this, the setup and teardown methods. These functions are invoked immediately before and immediately after each test (spec) is run. This enables you to setup all your tests and clean up all of your tests with as little code as possible.

Take a look at how the setup cleans up the code directly above this paragraph.

```
describe("CalcUtils", function() { // suite var calc;  
  
  //This will be called before running each spec beforeEach(function() { // setup var calc = new CalcUtils();  
});  
  
describe("calculation tests", function(){ // suite  
  //Spec for sum operation  
  it("2 plus 2 equals 4", function() { // spec expect(calc.sum(2,2)) // expect .toEqual(4); // matcher });  
  
  //Spec for sum operation with decimal it("2 plus 2 equals 4", function() { // spec expect(calc.sum(2.5,2)) // expect .toEqual(4.5); // matcher });  
});  
});
```

24.10 CLI

When we use the Angular CLI to generate our Angular project, it automatically (by default) generates unit test code for you that works with Karma and Jasmine. For example, when you generate the Angular project it generates an application component called ‘app.component.ts’ and a unit test file ‘app.component.spec.ts’. This unit test file already has methods stubbed out to unit test your component.

24.11 CLI - Running Unit Tests

When you issue the following command, Angular performs a compile of the project (the one in the current working directory) then invokes Karma to run all the unit tests: `ng test`

This command includes a file watcher. If you change one of the project files, it will automatically rebuild the project and re-run the tests.

24.12 CLI – Unit Test Files

When you use the Angular CLI to generate an Angular project, the project generates unit test files that use Karma and Jasmine. These unit tests files: Typically end with ‘.spec.ts’.

Follow the Jasmine format, having a ‘Describe’ block that contains a block of ‘It’ tests.

Can be modified, allowing you to add more tests.

Can be written from scratch and Karma will pick up and run them for you.

Use many of the Angular Testing Objects in the Angular ‘@angular/core/testing’ module.

24.13 CLI – Dependency Injection

Each described suite of tests is kind of like a ‘mini module’ because it runs code that has dependencies – therefore it needs to set them up like a module does (an Angular @NgModule).

24.14 Angular Testing Objects

Angular provides a module '@angular/core/testing' that contains helper objects to make it easier to write unit tests.

```
import { TestBed, async } from '@angular/core/testing';
```

There are the objects that you are most likely to use in the testing module.

Name	Type	Description
TestBed	Class	<p>Enables the developer to create an enclosure in which the code to be tested can run. Provides the following:</p> <ul style="list-style-type: none">Instantiation of component within enclosure.Methods to control dependency injection for component.Methods to query the component's DOM elements.Methods to invoke Angular change detection.Method to compile the components being tested.
async	Function	<p>It takes a parameter-less function and <i>returns a function</i> which becomes the true argument to the 'beforeEach'.</p> <p>It lets you perform the initialization code in the 'beforeEach' (the spec 'setup') asynchronously.</p>

24.15 Component Fixture

The TestBed method ‘createComponent’ enables you to create the component inside a testing enclosure and returns you an instance of a ComponentFixture object. One of the reasons that the component fixture is very useful is that it provides access to the component being debugged.

The ‘debugElement’ property of the ComponentFixture represents the Angular component and its corresponding DOM elements. It contains the following properties:

Property	Description
componentInstance	A reference to your component class. Useful if you want to access instance variables, methods within your component.
nativeElement	A reference to your component class’s corresponding html element in the DOM. Useful if you want to access the DOM to see how your component is being rendered by the template.

ComponentInstance

Within the ‘debugElement’ property of the fixture, the user can access the Angular Component via the ‘componentInstance’ property. Once you access the ‘debugElement’ you can call your methods in your component to test it.

• NativeElement

Within the ‘debugElement’ property of the fixture, the user can access the DOM element via the ‘nativeElement’ property. The ‘nativeElement’ gives us the root element of the html generated by the Angular Component. This root element is represented by a HTMLElement object, which is a fully-fledged object with many properties and methods.

• Native Element – HTMLElement Object

The HTMLElement object is not angular-specific, it is a very commonly-used object in web development. Please refer to <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement> for more information.

When you get the native element for the debugElement, this returns the html element object for your component. Not the entire DOM!

Sometimes developers make the mistake of expecting this element to contain html elements that are outside the scope of your Component. They will not be available!

Some of the more useful methods and properties of the HTMLElement are:

Name	Description
innerText	Useful for returning the text inside the element.

(Property)	Remember that this element may include unexpected whitespace.
innerHTML (Property)	Returns the HTML syntax of the markup inside the element belonging to the component.
outerHTML (Property)	Returns the HTML syntax of the markup inside the element, including its descendants.
querySelector (Method)	Returns the first element that is a descendant of the element on which it is invoked that matches the specified group of selectors. Useful for finding an element inside the element belonging to the component. For example the code below expects that the button with css class 'button-primary' button is defined: <code>expect(element.querySelector("button.button-primary")).toBeDefined();</code>
querySelectorAll (Method)	Returns a NodeList of all elements descended from the element on which it is invoked that match the specified group of CSS selectors. Useful for finding sub elements inside the element belonging to the component. For example, the code gets a list of text-area elements in the element: <code>let textAreas = element.querySelectorAll("text-area");</code>
getAttribute ([name]) (Method)	Returns an attribute (identified by name) for the element, for example disabled. For example, the code below expects that the approve button is disabled: <code>expect(approveButton.getAttribute("disabled")).toBeDefined();</code>

24.16 CLI Unit Test – Example #1

This example is not going to be exciting but it will show the generation of an example CLI project and examine the generated test code. This will be example ‘ch24-ex100’.

- **Step 1 – Build the App using the CLI**

```
ng new ch24-ex100 --inline-template --inline-style
```

- **Step 2 – Navigate to Folder**

```
cd ch24-ex100
```

- **Step 3 – Open File ‘app.component.spec.ts’**

Note the following:

The ‘beforeEach’ method is invoked before each spec. This method configures the testing module to test the AppComponent component.

There are three specs (tests). Each one is invoked asynchronously using the ‘async’ method in the testing module.

The first spec (test) creates a fixture then gets the component instance from the debug element. It checks that the component is truthy (i.e. has an assigned value).

The second spec (test) creates a fixture then gets the component instance from the debug element. It checks that the component’s ‘title’ instance variable has the value ‘app works!’.

The third spec (test) creates a fixture then gets the component’s element from the debug element. It checks that this element has an ‘h1’ element that contains the value ‘app works!’.

- **Step 4 – Run Tests**

```
ng test
```

24.17 CLI Unit Test – Example #2

Let's create a simple component that allows you to increment a counter. Then we will write a unit test for it. This will be example 'ch24-ex200'.



- **Step 1 – Build the App using the CLI**

```
ng new ch24-ex200 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch24-ex200  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- **Step 4 – Edit Class**

Edit the file 'app.component.ts' and change it to the following:

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `
<h1>
{{counter}}
</h1>
<button (click)="incrementCounter()">Increment Counter</button> `,
styles: []
})
export class AppComponent {
counter = 0;

incrementCounter(){
  this.counter++;
}
```

● Step 5 – Edit Unit Test

Edit the file ‘app.component.spec.ts’ and change it to the following:

```
import { TestBed, async } from '@angular/core/testing'; import { AppComponent } from './app.component';

describe('AppComponent', () => {
beforeEach(async() => {
  TestBed.configureTestingModule({
    declarations: [
      AppComponent
    ],
    }).compileComponents();
}));

it('should create the app', async(() => {
  const fixture = TestBed.createComponent(AppComponent); const app = fixture.debugElement.componentInstance;
expect(app).toBeTruthy();
}));

it(`should have as title '0'`, async(() => {
  const fixture = TestBed.createComponent(AppComponent); const app = fixture.debugElement.componentInstance;
expect(app.counter).toEqual(0);
}));

it(`should render '0' in a h1 tag`, async(() => {
  const fixture = TestBed.createComponent(AppComponent); fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement; expect(compiled.querySelector('h1').textContent).toContain('0'); });

it('should increment counter ten times', async(() => {
  const fixture = TestBed.createComponent(AppComponent); fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement; for (let i=0;i<10;i++){
    compiled.querySelector('button').click(); fixture.detectChanges();
    const nbrStr = (i + 1) + "";
    expect(compiled.querySelector('h1').textContent).toContain(nbrStr); }
  }));

});
```

● Step 6 – Run Tests

```
ng test
```

● Exercise Complete

Note that there was an additional test added at the end that clicks the ‘increment’ button ten times. Note that the additional test does not work until the ‘fixture.detectChanges’ method is called to perform change detection once the button is clicked.

24.18 CLI Unit Test – Example #3

Let's create a simple component that allows you to search for trails. Then we will write a unit test for it. This will be example 'ch24-ex300'.

Trail Finder

Atlanta

Name: Boat Rock
State: Georgia

Directions: From the intersection of Interstate 20 and Fulton Industrial Boulevard go south for 3.8 miles, turn left onto Bakers Ferry Road SW, go 0.5 miles, turn left on Boat Rock Road SW, go 0.4 miles, look for small gravel driveway on the right, pull into small 6 car parking lot. There is a small kiosk at the edge of the lot with a rough map of the area and a trail leading up to the boulders. The lake area is located a few hundred yards to the southeast (see drtopo map).

1220 Boat Rock Road Mapquest Link

Activities:

- hiking For those of us who like hiking AND rock climbing! Very cool place just inside of Atlanta. We took our children here and they could climb some of the boulders. A great experience for families and it's fun getting to watch the expert climbers on the rocks!

Trail Finder

Atlanta

We could not find a trail here. :(

Karma v1.7.0 - connected

Chrome 59.0.3071 (Mac OS X 10.12.5) is idle

Jasmine 2.6.4

• • •

4 specs, 0 failures

```
AppComponent (data found)
  should create the app
  should render the first trail found

AppComponent (no data found)
  should create the app
  should render an error message
```

- **Step 1 – Build the App using the CLI**

```
ng new ch24-ex300 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch24-ex300
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see ‘app works!’.

● Step 4 – Edit Class

Edit the file ‘app.component.ts’ and change it to the following:

```
import { Component } from '@angular/core'; import { Service } from './service';
import { FormsModule } from '@angular/forms';
import 'rxjs/Rx';

@Component({
  selector: 'app-root',
  template: `
    <h2>Trail Finder</h2>
    <input [(ngModel)]="search" placeholder="city"> <button (click)="doSearch()">Find Me a Trail</button> <div id="notFound"
      class="notFound" *ngIf="_searched && !_result"> We could not find a trail here. :(

    </div>
    <div class="found" *ngIf="_searched && _result"> <p id="name">Name: {{_result?.name}}</p> <p id="state">State:
      {{_result?.state}}</p> <p id="directions">Directions: {{_result?.directions}}</p> <p>Activities:</p>
      <ul id="activities" *ngIf="_result?.activities"> <li *ngFor="let activity of _result.activities"> {{activity.activity_type_name}}</li>
      {{activity.description}}
    </ul>
  `,
  styles: [`.found {
    border: 1px solid black;
    background-color: #8be591;
    color: black;
    margin: 10px;
    padding: 10px;
  },
  .notFound {
    border: 1px solid black;
    background-color: #d13449;
    color: white;
    margin: 10px;
    padding: 10px;
  }]`]
})
export class AppComponent {
  search = 'Atlanta';
  _searched = false;
  _result = "";

  constructor(private _service: Service) {}

  doSearch() {
    this._service.search(this._search).subscribe(
      res => {
        const obj = res.json();
        if ((obj) && (obj.places) && (obj.places.length) && (obj.places.length > 0)){
          this._result = obj.places[0];
        }
      }
    );
  }
}
```

```

} else{
this._result = undefined;
}
},
err => {
console.log(err);
},
() => {
this._searched = true;
}
);
}
}

```

● Step 5 – Edit Unit Test

Edit the file ‘app.component.spec.ts’ and change it to the following:

```

import { TestBed, async } from '@angular/core/testing';
import { HttpModule } from '@angular/http';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { Service } from './service';
import { Observable } from 'rxjs/Rx';

describe('AppComponent (data found)', () => {

beforeEach(async(() => {

const MOCKDATA =
{
"places": [
{
"city":"Atlanta", "state":"Georgia", "country":"United States", "name":"Boat Rock", "parent_id":null,
"unique_id":5370,
"directions":"From the intersection of Interstate 20 and Fulton Industrial Boulevard go south for 3.8 miles, turn left onto Bakers Ferry Road SW, go 0.5 miles, turn left on Boat Rock Road SW, go 0.4 miles, look for small gravel driveway on the right, pull into small 6 car parking lot. There is a small kiosk at the edge of the lot with a rough map of the area and a trail leading up to the boulders. The lake area is located a few hundred yards to the southeast (see drtopo map).&lt;br /&gt;&lt;br /&gt;&lt;br /&gt;&lt;br /&gt;&lt;br /&gt;&lt;br /&gt;1220
Boat Rock Road Mapquest Link ", "lat":0.0,
"lon":0.0,
"description":null,
"date_created":null,
"children":[
],
"activities":[
{
"name":"Boat Rock", "unique_id":"2-1012", "place_id":5370,
"activity_type_id":2,
"activity_type_name":"hiking", "url":"http://www.tripleblaze.com/trail.php?c=3&i=1012", "attribs":{
"\\"length\\":\"1\""
},
"description":"For those of us who like hiking AND rock climbing! Very cool place just inside of Atlanta. We took our children here and they could climb some of the boulders. A great experience for families and it's fun getting to watch the expert climbers on the rocks!",
"length":1.0,
"activity_type":{
"created_at":"2012-08-15T16:12:21Z", "id":2,
"name":"hiking", "updated_at":"2012-08-15T16:12:21Z"
}
}
]
})
});
```

```

},
"thumbnail":"http://images.tripleblaze.com/2009/07/Myspace-Pictures-130-0.jpg", "rank":null,
"rating":0.0
}
]
}
]
};

const mockResponse = {
json() {
return MOCKDATA;
}
}

const mockService = {
search(search) {
return Observable.of(mockResponse);
}
};

TestBed.configureTestingModule({
declarations: [
AppComponent
],
imports: [
HttpModule,
FormsModule
],
providers: [{provide: Service, useValue: mockService } ]
}).compileComponents();
}));

it('should create the app', async(() => {
  const fixture = TestBed.createComponent(AppComponent); const app = fixture.debugElement.componentInstance;
expect(app).toBeTruthy();
}));

it('should render the first trail found', async(() => {
  const fixture = TestBed.createComponent(AppComponent); fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement; compiled.querySelector('button').click(); fixture.detectChanges();
  expect(compiled.querySelector('#name').textContent).toContain('Boat Rock');
expect(compiled.querySelector('#state').textContent).toContain('Georgia');
expect(compiled.querySelector('#directions').textContent).toContain('Interstate 20 and Fulton Industrial Boulevard');
expect(compiled.querySelector('#activities').children.length.toBe(1); }));
});

describe('AppComponent (no data found)', () => {

beforeEach(async(() => {

  const mockNoResponse = {
json() {
return undefined;
}
}

```

```

const mockService = {
  search(search) {
    return Observable.of(mockNoResponse);
  }
};

 TestBed.configureTestingModule({
 declarations: [
  AppComponent
 ],
 imports: [
  HttpClientModule,
  FormsModule
 ],
 providers: [{provide: Service, useValue: mockService }]
 }).compileComponents();
});

it('should create the app', async(() => {
  const fixture = TestBed.createComponent(AppComponent); const app = fixture.debugElement.componentInstance;
  expect(app).toBeTruthy();
}));

it('should render an error message', async(() => {
  const fixture = TestBed.createComponent(AppComponent); fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement; compiled.querySelector('button').click(); fixture.detectChanges();
  expect(compiled.querySelector('#notFound').textContent).toContain('We could not find a trail here'); }));
});

```

● Step 6 – Run Tests

ng test

● Exercise Complete

Note the following:

1. The test code sets up a dummy response object that is an object that implements the ‘json’ method.
2. The test code provides a mock version of the Service class that returns an Observable of the dummy response object above.
3. This enables the mock version of the Service class to produce exactly the same as the real Service class.
4. The file ‘app.component.spec.ts’ contains code for two suites for two scenarios:
 - There is data returned for the search.
 - There is no data returned for the search.

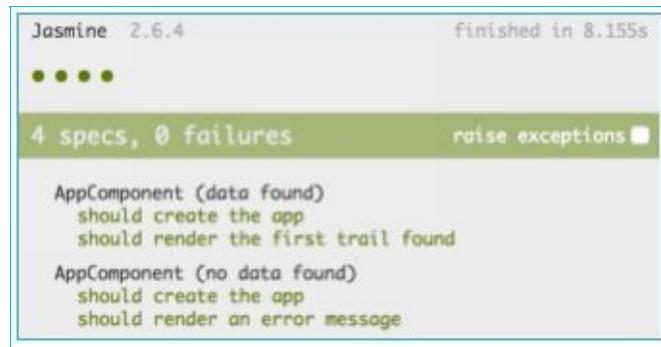
24.19 Angular Http Testing Objects

Note that Angular provides the class 'MockBackend' to be used as a mock backend to the Http service. It resides in the 'Angular 4/http/testing' namespace.

MockBackEnd	The MockBackEnd contains one or more MockConnections. It is used to fake communication with a server and allows developers to fake server responses with MockConnections. Developers can inject the MockBackEnd into communication service dependencies (such as Http) to make them work with a mock server (with mock responses) rather than a real one.
MockConnection	MockConnections allow developers to produce fake server responses (mock) to unit test code.

24.20 Angular Http Testing Objects - Example

Let's create a simple component that allows you to search for trails (the same as before). Then we will write a unit test for it using a mock backend and a mock connection. This will be example 'ch24-ex400'.



Jasmine 2.6.4 finished in 8.155s
• • •
4 specs, 0 failures raise exceptions
AppComponent (data found)
should create the app
should render the first trail found
AppComponent (no data found)
should create the app
should render an error message

● Step 1 – Build the App using the CLI

```
ng new ch24-ex400 --inline-template --inline-style
```

● Step 2 – Start Ng Serve

```
cd ch24-ex400  
ng serve
```

● Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see 'app works!'.

● Step 4 – Edit Class

Edit the file 'app.component.ts' and change it to the following:

```
import { Component } from '@angular/core'; import { FormsModule } from '@angular/forms';  
import { Http, Headers, URLSearchParams, RequestOptions, RequestMethod } from '@angular/http'; import 'rxjs/Rx';  
  
@Component({  
  selector: 'app-root',  
  template: `  
    <h2>Trail Finder</h2>  
    <input [(ngModel)]="search" placeholder="city"> <button (click)="doSearch()">Find Me a Trail</button> <div id="notFound"  
    class="notFound" *ngIf="!_searched && !_result"> We could not find a trail here. :(  
    </div>  
    <div class="found" *ngIf="!_searched && _result"> <p id="name">Name: {{_result?.name}}</p> <p id="state">State:  
    {{_result?.state}}</p> <p id="directions">Directions: {{_result?.directions}}</p> <p>Activities:</p>  
    <ul id="activities" *ngIf="!_result?.activities"> <li *ngFor="let activity of _result.activities"> {{activity.activity_type_name}}  
    {{activity.description}}</li>  
    </ul>  
  `,  
  styles: ['.found {  
    border: 1px solid black;  
    background-color: #8be591;  
    color: black;
```

```

margin: 10px;
padding: 10px;
},
.notFound {
  border: 1px solid black;
  background-color: #d13449;
  color: white;
  margin: 10px;
  padding: 10px;
}]
})
export class AppComponent {
  _search = 'Atlanta';
  _searched = false;
  _result = "";

  constructor(private _http: Http) {
  }

  doSearch() {
    this.search(this._search).subscribe(
      res => {
        const obj = res.json();
        if ((obj) && (obj.places) && (obj.places.length) && (obj.places.length > 0)){
          this._result = obj.places[0];
        }else{
          this._result = undefined;
        }
      },
      err => {
        console.log(err);
      },
      () => {
        this._searched = true;
      }
    );
  }

  search(search) {
    const mashapeKey = '0xWYjpdztcmsheZU9AWLNQcE9g9wp1qdRkFjsneaEp2Yf68nYH'; const headers: Headers = new Headers();
    headers.append('Content-Type', 'application/json'); headers.append('X-Mashape-Key', mashapeKey); const options = new RequestOptions({
      headers: headers });
    // concatenated string
    const concatenatedUrl: string =
      "https://trailapi-trailapi.p.mashape.com?q[city_cont]=" +
      encodeURIComponent(search);
    return this._http.get(concatenatedUrl, options);
  }
}

```

● Step 5 – Edit Function to Create Fake Backend with Data

Edit the file ‘fakeBackendWithData.ts’ and change it to the following:

```

import { Http, BaseRequestOptions, Response, ResponseOptions, XHRBackend, RequestOptions } from '@angular/http'; import {
  MockBackend, MockConnection } from '@angular/http/testing';
  export function fakeBackendWithDataFactory(backend: MockBackend, options: BaseRequestOptions, realBackend: XHRBackend) {

    backend.connections.subscribe((connection: MockConnection) => {

```

```

setTimeout(() => {
const MOCKDATA =
{
"places": [
{
"city": "Atlanta", "state": "Georgia", "country": "United States", "name": "Boat Rock", "parent_id": null,
"unique_id": 5370,
"directions": "From the intersection of Interstate 20 and Fulton Industrial Boulevard go south for 3.8 miles, turn left onto Bakers Ferry Road SW, go 0.5 miles, turn left on Boat Rock Road SW, go 0.4 miles, look for small gravel driveway on the right, pull into small 6 car parking lot. There is a small kiosk at the edge of the lot with a rough map of the area and a trail leading up to the boulders. The lake area is located a few hundred yards to the southeast (see drtopo map).&lt;br /&gt;&lt;br /&gt;&lt;br /&gt;&lt;br /&gt;&lt;br /&gt;&lt;br /&gt;1220 Boat Rock Road Mapquest Link ", "lat": 0.0,
"lon": 0.0,
"description": null,
"date_created": null,
"children": [
],
"activities": [
{
"name": "Boat Rock", "unique_id": "2-1012", "place_id": 5370,
"activity_type_id": 2,
"activity_type_name": "hiking", "url": "http://www.tripleblaze.com/trail.php?c=3&i=1012", "attribs": {
"\\"length\\": "\\'1\\'"
},
"description": "For those of us who like hiking AND rock climbing! Very cool place just inside of Atlanta. We took our children here and they could climb some of the boulders. A great experience for families and it's fun getting to watch the expert climbers on the rocks!", "length": 1.0,
"activity_type": {
"created_at": "2012-08-15T16:12:21Z", "id": 2,
"name": "hiking", "updated_at": "2012-08-15T16:12:21Z"
},
"thumbnail": "http://images.tripleblaze.com/2009/07/Myspace-Pictures-130-0.jpg", "rank": null,
"rating": 0.0
}
]
}
];
};

connection.mockRespond(new Response(new ResponseOptions({ status: 200, body: MOCKDATA })));
}, 500);

});

return new Http(backend, options);
};

export let fakeBackendWithDataProvider = {
// use fake backend in place of Http service for backend-less development provide: Http,
useFactory: fakeBackendWithDataFactory,
deps: [MockBackend, BaseRequestOptions, XHRBackend]
};

```

● Step 6 – Edit Function to Create Fake Backend without Data

Edit the file ‘fakeBackendWithNoData.ts’ and change it to the following:

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

```

MockBackend, MockConnection } from '@angular/http/testing';
export function fakeBackendWithNoDataFactory(backend: MockBackend, options: BaseRequestOptions, realBackend: XHRBackend) {

backend.connections.subscribe((connection: MockConnection) => {
  setTimeout(() => {
    const MOCKDATA = { "places": []};
    connection.mockRespond(new Response(new ResponseOptions({ status: 200, body: MOCKDATA })));
  }, 500);
});

return new Http(backend, options);
};

export let fakeBackendWithNoDataProvider = {
  // use fake backend in place of Http service for backend-less development provide: Http,
  useFactory: fakeBackendWithNoDataFactory,
  deps: [MockBackend, BaseRequestOptions, XHRBackend]
};

```

● Step 7 – Edit Unit Test

Edit the file ‘app.component.spec.ts’ and change it to the following:

```

import { ReflectiveInjector } from '@angular/core'; import { TestBed, tick, async } from '@angular/core/testing'; import { HttpModule,
Response, RequestOptions, ConnectionBackend, BaseRequestOptions, RequestOptions, Http } from '@angular/http'; import {
MockBackend, MockConnection } from '@angular/http/testing'; import { fakeBackendWithDataProvider } from './fakeBackendWithData';
import { fakeBackendWithNoDataProvider } from './fakeBackendWithNoData'; import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { Observable } from 'rxjs/Rx';

describe('AppComponent (data found)', () => {

beforeEach(async() => {

  TestBed.configureTestingModule({
  declarations: [
  AppComponent
  ],
  imports: [
  HttpModule,
  FormsModule
  ],
  providers: [
  fakeBackendWithDataProvider,
  MockBackend,
  BaseRequestOptions
  ]
  }).compileComponents();

});

it('should create the app', async() => {
  const fixture = TestBed.createComponent(AppComponent); const app = fixture.debugElement.componentInstance;
  expect(app).toBeTruthy();
});
});
```

```

it('should render the first trail found', async(() => {

  const fixture = TestBed.createComponent(AppComponent); fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement; compiled.querySelector('button').click(); setTimeout(() => {
    expect(compiled.querySelector('#name').textContent).toContain('Boat Rock');
    expect(compiled.querySelector('#state').textContent).toContain('Georgia');
    expect(compiled.querySelector('#directions').textContent).toContain('Interstate 20 and Fulton Industrial Boulevard');
    expect(compiled.querySelector('#activities').children.length).toBe(1); }, 2000);
  }));

});

describe('AppComponent (no data found)', () => {

beforeEach(async(() => {

  TestBed.configureTestingModule({
  declarations: [
  AppComponent
  ],
  imports: [
  HttpClientModule,
  FormsModule
  ],
  providers: [
  fakeBackendWithNoDataProvider,
  MockBackend,
  BaseRequestOptions
  ]
}).compileComponents();

}));

it('should create the app', async(() => {
  const fixture = TestBed.createComponent(AppComponent); const app = fixture.debugElement.componentInstance;
  expect(app).toBeTruthy();
}));

it('should render an error message', async(() => {
  const fixture = TestBed.createComponent(AppComponent); fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement; compiled.querySelector('button').click(); fixture.detectChanges();
  setTimeout(() => {
    expect(compiled.querySelector('#notFound').textContent).toContain('We could not find a trail here'); }, 2000);
  }));

});

```

● Step 8 – Run Tests

ng test

● Exercise Complete

Note the following:

1. The file ‘fakeBackendWithData’ sets up a factory method to return a mock backend that responds to a request with some test data. It then sets up a provider that uses the fake backend.
2. The file ‘fakeBackendWithNoData’ sets up a factory method to return a mock backend that responds to a request with empty data. It then sets up a provider that uses the fake backend.
3. The file ‘app.component.spec.ts’ contains code for two suites for two scenarios:
 - There is data returned for the search.
 - There is no data returned for the search.

25 More Advanced Topics

25.1 View Encapsulation

Remember how you can apply styles to a component using the ‘styles’ or ‘styleUrls’ properties of the @Component annotation?

The meaning of the word ‘encapsulation’ is ‘the action of enclosing something in or as if in a capsule’.

Angular view encapsulation is to do with which method angular uses to enclose these styles (the ones you applied the ‘styles’ or ‘styleUrls’ properties) with the component.

25.2 Why is View Encapsulation Required?

When you use the ‘styles’ or ‘styleUrls’ properties to style a component, Angular adds styling code into a ‘style’ tag in the ‘head’ part of the html document.

This is fine but you need to watch out for some things. What happens if you have conflicting css style rules in different Components? What if (for example) you have ‘.h2 {color:red}’ in one Component and ‘.h2 {color:green}’ in another Component?

If your Components are using a Shadow Dom (or Emulated Shadow Dom) you don’t need to worry about these conflicting styles. You probably are using a Shadow Dom (or at least an Emulated Shadow Dom) because this is what Angular 4 gives you by default.

However, you need to know about Shadow Doms because if your Components are not using a Shadow Dom (or Emulated Shadow Dom) then these conflicting styles could cause you headaches.

25.3 Shadow DOMs

Scope has been a problem for some time on the browser. Developers have been able to make sweeping global changes to the html document easily with little work. They can add a few lines of CSS and impact many DOM elements immediately. This is powerful but can leave your component's style easy to override or break accidentally.

Shadow DOM is a new emerging standard on the web. Shadow DOMs work on most browsers except for Internet Explorer. The idea behind Shadow DOM is to give developers the option of creating Components with their own separate DOM trees, encapsulated away from the other Components, contained within Host Elements. This gives the developers of having styles 'scoped' to just that single Component that cannot affect the rest of the document.

When you write a component, you don't have to use a Shadow DOM, but it is an option available to use, which you can control by using the 'encapsulation' option of the '@Component' annotation.

25.4 Component Encapsulation

The ‘encapsulation’ option of the ‘@Component’ annotation gives the developer control over the level of View Encapsulation, in other words to implement or not implement a Shadow DOM.

Option	Description
ViewEncapsulation. Emulated	Emulated Shadow DOM. This is the default mode for Angular
ViewEncapsulation. Native	Native Shadow DOM.
ViewEncapsulation. None	No Shadow DOM at all.

25.5 ViewEncapsulation.Emulated – Example

Let's create an example component with a style and specify the View Encapsulation as emulated. This is the default mode for Angular 4. This will be example 'ch25-ex100'.

- **Step 1 – Build the App using the CLI**

```
ng new ch25-ex100 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch25-ex100  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- **Step 4 – Edit Component**

Edit the file 'app.component.ts' and change it to the following:

```
import { Component, ViewEncapsulation } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  template: `<h1>  
    {{title}}  
  </h1>  
`,  
  styles: ['h1 { color: red }'],  
  encapsulation: ViewEncapsulation.Emulated  
)  
export class AppComponent {  
  title = 'app';  
}
```

- **Exercise Complete**

The app should be working and displaying the word 'app' in red. If you look at the document you will see the following:

```

<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Ch25Ex100</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <style type="text/css">
      /* You can add global styles to this file, and also import other style files */
    </style>
    <style>h1[_ngcontent-c0] { color: red }</style>
  </head>
  <body>
    <app-root _ngcontent-c0 ng-version="4.3.1">
      <h1 _ngcontent-c0>
        app
      </h1>
    </app-root>
    <script type="text/javascript" src="inline.bundle.js"></script>
    <script type="text/javascript" src="polyfills.bundle.js"></script>
    <script type="text/javascript" src="styles.bundle.js"></script>
    <script type="text/javascript" src="vendor.bundle.js"></script>
    <script type="text/javascript" src="main.bundle.js"></script>
  </body>
</html>

```

As you can see, the style is written to the Head of the document. Also Angular rewrote our style for the component, adding an identifier to both the Style and the Component to link just the two together and avoid conflicts with other components with other identifiers. In the case below the identifier is '_ngcontent-c0'.

25.6 ViewEncapsulation.Native – Example

Let's create an example component with a style and specify the View Encapsulation as native. This will be example 'ch25-ex200'

- Step 1 – Build the App using the CLI

```
ng new ch25-ex200 --inline-template --inline-style
```

- Step 2 – Start Ng Serve

```
cd ch25-ex200  
ng serve
```

- Step 3 – Open App

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- Step 4 – Edit Component

Edit the file 'app.component.ts' and change it to the following:

```
import { Component, ViewEncapsulation } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  template: `<h1>  
    {{title}}  
  </h1>  
`,  
  styles: ['h1 { color: red }'],  
  encapsulation: ViewEncapsulation.Native  
})  
export class AppComponent {  
  title = 'app';  
}
```

- Exercise Complete

The app should be working and displaying the word 'app' in red. If you look at the document you will see the following:

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Ch25Ex200</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
  </head>
  <body>
    <app-root ng-version="4.3.1">
      <#shadow-root open>
        <style>h1 { color: red }</style>
        <h1>
          app
        </h1>
      </#shadow-root>
    </app-root>
    <script type="text/javascript" src="inline.bundle.js"></script>
    <script type="text/javascript" src="polyfills.bundle.js"></script>
    <script type="text/javascript" src="styles.bundle.js"></script>
    <script type="text/javascript" src="vendor.bundle.js"></script>
    <script type="text/javascript" src="main.bundle.js"></script>
  </body>
</html>
```

The style is no longer written to the Head of the document, it is instead written inside the component's Shadow DOM. To see this output, you must turn on 'Display Shadow DOM' in your browser. Now it is easy to see how your styles are only applied to the Component, which resides in the Host Element 'app-root'.

25.7 ViewEncapsulation.None – Example

Let's create an example component with a style and specify the View Encapsulation as none. This will be example 'ch25-ex300'

- **Step 1 – Build the App using the CLI**

```
ng new ch25-ex300 --inline-template --inline-style
```

- **Step 2 – Start Ng Serve**

```
cd ch25-ex300  
ng serve
```

- **Step 3 – Open App**

Open web browser and navigate to localhost:4200. You should see 'app works!'.

- **Step 4 – Edit Component**

Edit the file 'app.component.ts' and change it to the following:

```
import { Component, ViewEncapsulation } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  template: `<h1>  
    {{title}}  
  </h1>  
`,  
  styles: ['h1 { color: red }'],  
  encapsulation: ViewEncapsulation.None  
)  
export class AppComponent {  
  title = 'app';  
}
```

- **Exercise Complete**

The app should be working and displaying the word 'app' in red. If you look at the document you will see the following:

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Ch25Ex300</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <style type="text/css">
      /* You can add global styles to this file, and also import other style files */
    </style>
    <style>h1 { color: red }</style>
  </head>
  <body>
    <app-root ng-version="4.3.1">
      <h1>
        app
      </h1>
    </app-root>
    <script type="text/javascript" src="inline.bundle.js"></script>
    <script type="text/javascript" src="polyfills.bundle.js"></script>
    <script type="text/javascript" src="styles.bundle.js"></script>
    <script type="text/javascript" src="vendor.bundle.js"></script>
    <script type="text/javascript" src="main.bundle.js"></script>
  </body>
</html>
```

The style is written to the Head of the document and this style applies to entire document, possibly conflicting with other styles from other components. Be careful with this mode.

25.8 View Encapsulation - Conclusion

Angular offers you the best of both worlds: view encapsulation plus the ability to share styles.

It offers Emulated View Encapsulation as default: your component-specific styles are protected for you even if you don't add the 'encapsulation' specification to the '@Component' annotation.

If you need to share styles in your Components, you can use the 'styleUrls' specification in your '@Component' annotation to specify shared common style files.

25.9 Styling Content Children

Remember how you can apply styles to a component using the ‘styles’ or ‘styleUrls’ properties of the `@Component` annotation?

These styles only apply to the HTML in the component's own template.

What happens if you go get HTML content from the server and you inject this content into your component dynamically? How do you style that?

The answer is that you use special style tags to apply styles to your component and its sub-elements (like the HTML content from the server for example).

For example, the style rule below styles all the h3 elements in your component and its sub-elements.

```
:host /deep/ h3 { font-style: italic; }
```


26 Resources

26.1 Introduction

I hope this book has turned out to be useful to you. I did not develop this book in a vacuum, I relied on many sources of information when writing it. In this Chapter, we will get through some of the resources available to you to help you in your Angular development.



26.2

Fast

Mobile

Flexible

‘Official Website’

<https://angular.io/> is the official Angular website. It has a ton of great information and it is well laid out. This should be your starting point for any Angular 4 research.

I found the API Preview page <https://angular.io/docs/ts/latest/api/> especially useful (see below). Type in what you are looking for and it displays the search results below. The search results include the objects that match the search, grouped by their packages. This package information is useful for writing the ‘imports’ at the top of the classes. When you click on an object in the search results it shows you detailed information about its API.

26.3 GitHub

GitHub is a web-based [Git](#) repository hosting service and people use it to publish their code and to manage it using Git (see below). Git offers paid and free accounts. The paid accounts offer the advantage of private repositories. The free accounts are popular and are frequently used when people are writing open-source software projects. GitHub reports having over 12 million users and over 31 million repositories, making it the largest host of source code in the world.

Check out <https://github.com/mark-clow> for code samples and the example project from this book.

● Git

Git is a widely used source code management system for software development. Unlike older and more conventional source code management systems, Git allows developers to work in a distributed manner, managing their own local repositories on their computers, with or without a network. There is no ‘central’ repository, there are only ‘peer’ distributed repositories. Once a developer has completed code changes, he or she can merge their changes into shared repositories.

26.4 Angular 4-Related Blogs

http://blog.thoughtram.io/	Advanced Angular articles.
https://toddmotto.com/	Advanced Angular articles.
http://www.tryAngular 4.com/	A website to which people post questions or their Angular articles.
http://www.syntaxsuccess.com/angular-2-articles	Angular articles.
http://victorsavkin.com/	Angular articles.
http://blog.jhades.org/	Lots of JavaScript and Angular articles.
http://johnpapa.net/	Lots of articles, including those for Angular.

26.5 Angular Air

This is a superb video podcast about Angular:

<https://www.youtube.com/channel/UCdCOpvRk1lsBk26ePGDPLpQ>

27 Conclusion

I hope you enjoyed the book and I hope the code samples are working out for you. If you find one that does not work, send me an email at markclow@hotmail.com and I will fix it.

If you feel I am leaving something valuable out of this book please feel free to email me.

I intend to keep updating this book in my spare time as I want it to find it useful for myself as well as for yourselves. If you want the latest version of the book pdf, shoot me an email at with some kind of proof of purchase. I will be happy to send you an updated version.

Anyway, I wish you the best of luck in your endeavours.