

WA3142 Comprehensive Angular 12 Programming



Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-877-812-8887
Web: <http://www.webagesolutions.com>

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

IBM, WebSphere, DB2 and Tivoli are trademarks of the International Business Machines Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com

Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Copyright © 2021 Web Age Solutions Inc.

This publication is protected by the copyright laws of Canada, United States and any other country where this book is sold. Unauthorized use of this material, including but not limited to, reproduction of the whole or part of the content, re-sale or transmission through fax, photocopy or e-mail is prohibited. To obtain authorization for any such activities, please write to:

Web Age Solutions Inc.
821A Bloor Street West
Toronto
Ontario, M6G 1M1

Table of Contents

Chapter 1 - Introducing Angular.....	11
1.1 What is Angular?.....	11
1.2 Central Features of the Angular Framework.....	11
1.3 Appropriate Use Cases.....	12
1.4 Building Blocks of an Angular Application.....	12
1.5 Basic Architecture of an Angular Application.....	14
1.6 Installing and Using Angular.....	14
1.7 Anatomy of an Angular Application.....	15
1.8 Running the Application.....	15
1.9 Building and Deploying the Application.....	16
1.10 Angular for Native Mobile Apps.....	17
1.11 Summary.....	17
Chapter 2 - Introduction to TypeScript.....	19
2.1 Programming Languages for Use with Angular.....	19
2.2 TypeScript Syntax.....	20
2.3 Programming Editors.....	20
2.4 The Type System – Defining Variables.....	21
2.5 The Type System – Defining Arrays.....	22
2.6 Basic Primitive Types.....	22
2.7 Type in Functions.....	23
2.8 Type Inference.....	23
2.9 Defining Classes.....	24
2.10 Class Methods.....	24
2.11 Visibility Control.....	25
2.12 Class Constructors.....	25
2.13 Class Constructors – Alternate Form.....	26
2.14 Uninitialized Fields.....	26
2.15 Interfaces.....	27
2.16 Working with ES6 Modules.....	28
2.17 var vs let.....	28
2.18 Arrow Functions.....	29
2.19 Arrow Function Compact Syntax.....	29
2.20 Template Strings.....	30
2.21 Generics in Class.....	30
2.22 Generics in Function.....	31
2.23 Summary.....	31
Chapter 3 - Components.....	33
3.1 What is a Component?.....	33
3.2 An Example Component.....	34
3.3 Creating a Component Using Angular CLI.....	34
3.4 The Component Class.....	34
3.5 The @Component Decorator.....	35
3.6 Registering a Component to Its Module.....	35
3.7 Component Template.....	36

3.8 Example: HelloComponent Template.....	36
3.9 Example: The HelloComponent Class.....	36
3.10 Using a Component.....	37
3.11 Run the Application.....	37
3.12 Component Hierarchy.....	37
3.13 The Application Root Component.....	38
3.14 The Bootstrap File.....	38
3.15 Component Lifecycle Hooks.....	39
3.16 Example Lifecycle Hooks.....	39
3.17 CSS Styles.....	40
3.18 Summary.....	40
Chapter 4 - Component Templates.....	43
4.1 Templates.....	43
4.2 Template Location.....	44
4.3 The Mustache {{ }} Syntax.....	44
4.4 Setting DOM Element Properties.....	45
4.5 Setting Element Body Text.....	45
4.6 Event Binding.....	46
4.7 Expression Event Handler.....	46
4.8 Prevent Default Handling.....	47
4.9 Attribute Directives.....	47
4.10 Apply Styles by Changing CSS Classes.....	48
4.11 Example: ngClass.....	48
4.12 Applying Styles Directly.....	49
4.13 Structural Directives.....	49
4.14 Conditionally Execute Template.....	50
4.15 Example: ngIf.....	50
4.16 Looping Using ngFor.....	51
4.17 ngFor Local Variables.....	52
4.18 Manipulating the Collection.....	52
4.19 Example - Deleting an Item.....	53
4.20 Item Tracking with ngFor.....	53
4.21 Swapping Elements with ngSwitch.....	54
4.22 Grouping Elements.....	54
4.23 Template Reference Variable.....	55
4.24 Summary.....	55
Chapter 5 - Inter Component Communication.....	57
5.1 Communication Basics.....	57
5.2 The Data Flow Architecture.....	57
5.3 Preparing the Child to Receive Data.....	57
5.4 Send Data from Parent.....	58
5.5 More About Setting Properties.....	58
5.6 Firing Event from a Component.....	59
5.7 @Output() Example - Child Component.....	59
5.8 @Output() Example - Parent Component.....	60
5.9 Full Two Way Binding.....	60

5.10 Setting up Two Way Data Binding in Parent.....	61
5.11 Summary.....	61
Chapter 6 - Template Driven Forms.....	63
6.1 Template Driven Forms.....	63
6.2 Importing Forms Module.....	64
6.3 Basic Approach.....	65
6.4 Setting Up a Form.....	65
6.5 Getting User Input.....	65
6.6 Omitting ngForm Attribute.....	66
6.7 Initialize the Form.....	66
6.8 Two Way Data Binding.....	67
6.9 Form Validation.....	67
6.10 Angular Validators.....	68
6.11 Displaying Validation State Using Classes.....	68
6.12 Additional Input Types.....	69
6.13 Checkboxes.....	70
6.14 Select (Drop Down) Fields.....	71
6.15 Rendering Options for Select (Drop Down).....	72
6.16 Date fields.....	73
6.17 Radio Buttons.....	74
6.18 Summary.....	75
Chapter 7 - Reactive Forms.....	77
7.1 Reactive Forms Overview.....	77
7.2 The Building Blocks.....	77
7.3 Import ReactiveFormsModule.....	78
7.4 Construct a Form.....	78
7.5 Design the Template.....	79
7.6 Getting Input Values.....	79
7.7 Initializing the Input Fields.....	80
7.8 Setting Form Values.....	80
7.9 Subscribing to Input Changes.....	81
7.10 Validation.....	82
7.11 Built-In Validators.....	82
7.12 Showing Validation Error.....	82
7.13 Custom Validator.....	83
7.14 Using a Custom Validator.....	84
7.15 Supplying Configuration to Custom Validator.....	84
7.16 FormArray - Dynamically Add Inputs.....	85
7.17 FormArray - The Component Class.....	85
7.18 FormArray - The Template.....	86
7.19 FormArray - Values.....	86
7.20 Sub FormGroups - Component Class.....	87
7.21 Sub FormGroups - HTML Template.....	87
7.22 Why Use Sub FormGroups.....	88
7.23 Summary.....	88
Chapter 8 - Services and Dependency Injection.....	91

8.1 What is a Service?	91
8.2 Creating a Basic Service	91
8.3 The Service Class	92
8.4 What is Dependency Injection?	93
8.5 Injecting a Service Instance	93
8.6 Injectors	94
8.7 Injector Hierarchy	95
8.8 Registering a Service with the Root Injector	95
8.9 Registering a Service with a Component's Injector	96
8.10 Register a Service with a Feature Module Injector	96
8.11 Where to Register a Service?	97
8.12 Dependency Injection in Other Artifacts	97
8.13 Providing an Alternate Implementation	98
8.14 Dependency Injection and @Host	99
8.15 Dependency Injection and @Optional	100
8.16 Summary	101
Chapter 9 - HTTP Client	103
9.1 The Angular HTTP Client	103
9.2 Using The HTTP Client - Overview	103
9.3 Importing HttpClientModule	104
9.4 Service Using HttpClient	105
9.5 Making a GET Request	105
9.6 What does an Observable Object do?	106
9.7 Using the Service in a Component	107
9.8 The PeopleService Client Component	107
9.9 Error Handling	108
9.10 Customizing the Error Object	108
9.11 Making a POST Request	109
9.12 Making a PUT Request	109
9.13 Making a DELETE Request	110
9.14 Summary	110
Chapter 10 - Pipes and Data Formatting	113
10.1 What are Pipes?	113
10.2 Built-In Pipes	113
10.3 Using Pipes in HTML Template	114
10.4 Chaining Pipes	114
10.5 Internationalized Pipes (i18n)	115
10.6 Loading Locale Data	115
10.7 The date Pipe	116
10.8 The number Pipe	116
10.9 Currency Pipe	118
10.10 Create a Custom Pipe	119
10.11 Custom Pipe Example	120
10.12 Using Custom Pipes	120
10.13 Using a Pipe with ngFor	121
10.14 A Filter Pipe	121

10.15 Pipe Category: Pure and Impure.....	122
10.16 Pure Pipe Example.....	123
10.17 Impure Pipe Example.....	125
10.18 Summary.....	125
Chapter 11 - Introduction to Single Page Applications.....	127
11.1 What is a Single Page Application (SPA).....	127
11.2 Traditional Web Application.....	128
11.3 SPA Workflow.....	128
11.4 Single Page Application Advantages.....	129
11.5 HTML5 History API.....	130
11.6 SPA Challenges.....	130
11.7 Implementing SPA's Using Angular.....	131
11.8 Summary.....	132
Chapter 12 - The Angular Component Router.....	133
12.1 The Component Router.....	133
12.2 View Navigation.....	134
12.3 The Angular Router API.....	134
12.4 Creating a Router Enabled Application.....	135
12.5 Hosting the Routed Components.....	136
12.6 Navigation Using Links and Buttons.....	136
12.7 Programmatic Navigation.....	137
12.8 Passing Route Parameters.....	137
12.9 Navigating with Route Parameters.....	137
12.10 Obtaining the Route Parameter Values.....	138
12.11 Retrieving the Route Parameter Synchronously.....	138
12.12 Retrieving a Route Parameter Asynchronously.....	139
12.13 Query Parameters.....	139
12.14 Supplying Query Parameters.....	140
12.15 Retrieving Query Parameters Asynchronously.....	140
12.16 Problems with Manual URL entry and Bookmarking.....	141
12.17 Summary.....	141
Chapter 13 - Advanced HTTP Client.....	143
13.1 Request Options.....	143
13.2 Returning an HttpResponse Object.....	143
13.3 Setting Request Headers.....	144
13.4 Creating New Observables.....	144
13.5 Creating a Simple Observable.....	145
13.6 The Observable Constructor Method.....	145
13.7 Observable Operators.....	146
13.8 The map and filter Operators.....	146
13.9 The flatMap() Operator.....	147
13.10 The tap() Operator.....	147
13.11 The zip() Combinator.....	148
13.12 Caching HTTP Response.....	148
13.13 Making Sequential HTTP Calls.....	149
13.14 Making Parallel Calls.....	149

13.15 Customizing Error Object with <code>catchError()</code>	150
13.16 Error in Pipeline.....	151
13.17 Error Recovery.....	151
13.18 Summary.....	152
Chapter 14 - Angular Modules.....	153
14.1 Why Angular Modules?.....	153
14.2 Anatomy of a Module Class.....	153
14.3 <code>@NgModule</code> Properties.....	154
14.4 Feature Modules.....	155
14.5 Example Module Structure.....	155
14.6 Create a Domain Module.....	156
14.7 Create a Routed/Routing Module Pair.....	156
14.8 Create a Service Module.....	157
14.9 Creating Common Modules.....	158
14.10 Using One Module From Another.....	158
14.11 Summary.....	159
Chapter 15 - Advanced Routing.....	161
15.1 Routing Enabled Feature Module.....	161
15.2 Using the Feature Module.....	162
15.3 Lazy Loading the Feature Module.....	162
15.4 Creating Links for the Feature Module Components.....	163
15.5 More About Lazy Loading.....	163
15.6 Preloading Modules.....	164
15.7 <code>routerLinkActive</code> binding.....	164
15.8 Default Route.....	165
15.9 Wildcard Route Path.....	165
15.10 <code>redirectTo</code>	165
15.11 Child Routes.....	166
15.12 Defining Child Routes.....	167
15.13 <code><router-outlet></code> for Child Routes.....	167
15.14 Links for Child Routes.....	168
15.15 Navigation Guards.....	168
15.16 Creating Guard Implementations.....	169
15.17 Using Guards in a Route.....	169
15.18 Summary.....	170
Chapter 16 - Unit Testing Angular Applications.....	171
16.1 Unit Testing Angular Artifacts.....	171
16.2 Testing Tools.....	171
16.3 Typical Testing Steps.....	172
16.4 Test Results.....	173
16.5 Jasmine Test Suites.....	173
16.6 Jasmine Specs (Unit Tests).....	174
16.7 Expectations (Assertions).....	174
16.8 Matchers.....	175
16.9 Examples of Using Matchers.....	176
16.10 Using the <code>not</code> Property.....	176

16.11	Setup and Teardown in Unit Test Suites.....	177
16.12	Example of beforeEach and afterEach Functions.....	177
16.13	Angular Test Module.....	178
16.14	Example Angular Test Module.....	178
16.15	Testing a Service.....	178
16.16	Injecting a Service Instance.....	179
16.17	Test a Synchronous Method.....	179
16.18	Test an Asynchronous Method.....	180
16.19	Using Mock HTTP Client.....	180
16.20	Supplying Canned Response.....	181
16.21	Testing a Component.....	182
16.22	Component Test Module.....	182
16.23	Creating a Component Instance.....	183
16.24	The ComponentFixture Class.....	183
16.25	Basic Component Tests.....	184
16.26	The DebugElement Class.....	184
16.27	Simulating User Interaction.....	185
16.28	Summary.....	185
Chapter 17 -	Debugging.....	187
17.1	Overview of Angular Debugging.....	187
17.2	Viewing TypeScript Code in Debugger.....	187
17.3	Using the debugger Keyword.....	188
17.4	Debug Logging.....	189
17.5	What is Angular DevTools?.....	189
17.6	Using Angular DevTools.....	190
17.7	Angular DevTools - Component Structure.....	190
17.8	Angular DevTools - Change Detection Execution.....	190
17.9	Catching Syntax Errors.....	191
17.10	Summary.....	192

Chapter 1 - Introducing Angular

Objectives

Key objectives of this chapter

- Introduce the Angular Framework
- Learn how to install Angular
- Review a Basic Angular application

1.1 What is Angular?

Angular is:

- Angular is a frontend web development framework.
- Application code is written using TypeScript, HTML, CSS and SASS.
- Maintained by Google and community
- Open Source (MIT License)

Angular on the Web:

- Web Site: <http://angular.io>
- GitHub: <https://github.com/angular/angular>

Notes

Angular is an open source client side web development framework. Your code using higher level language and technologies like TypeScript and SASS. But they get compiled into JavaScript and CSS prior to execution in the browser.

Angular is the next iteration of the AngularJS framework. It has been re-engineered to improve load times and add features such as view animation and enhanced navigational routing. The changes are significant enough an entirely new project was created with a slightly different name.

1.2 Central Features of the Angular Framework

- Specifically geared for modern single page applications (SPA).
- Application code is developed using Component, Service, Directive etc. Each type of artifact has clearly defined roles and responsibilities.

- Provides Angular CLI tooling to quickly create an application and code artifacts.
- Encourages unit testing using Jasmine and Karma frameworks.
- Supports accessibility and localization.

Notes

Angular almost mandates a native app like application behavior known as Single Page Application or SPA. As the user interacts with the application, Angular transitions pages locally without reloading the whole DOM from the server. This is a major departure from server side scripting where individual views are created on the server that must be retrieved and loaded into the browser separately for each view transition. Loading the pages in this way causes noticeable delays when users navigate between views.

The framework supports modular application architectures such as Model-View-Controller and Model-View-View-Model.

1.3 Appropriate Use Cases

- Use Angular when:
 - ◇ It is a single page application (SPA).
 - ◇ The frontend GUI is complex.
 - ◇ Team size is large. Different members can contribute to different artifacts like Component and Service.
- Do not use Angular when
 - ◇ Pages are traditionally rendered on the server using technologies like ASP, JSP or PHP. Frameworks like Vue.js or jQuery will be better suited for this.
 - ◇ Pages need to be SEO optimized. SPA apps are by nature not SEO optimized. They will need a lot of extra work for Google to index the pages.

1.4 Building Blocks of an Angular Application

- Angular employs different types of code artifacts for different responsibilities.

- **Components** display data on a page and accept user input. Each component has a **template** that outputs the HTML used to render the display.
- **Directives** manipulate DOM elements for stylistic reasons. They are used in templates.
- **Pipes** format data. For example a pipe can display a number in currency format. They are also used in templates.
- **Services** contain pure business logic without any concern for the front end display. They often interact with the backend web services to fetch or update data.
- **Modules** group a related set of components, directives, services etc. There are many benefits to breaking up a large application in multiple modules as we will discuss later.

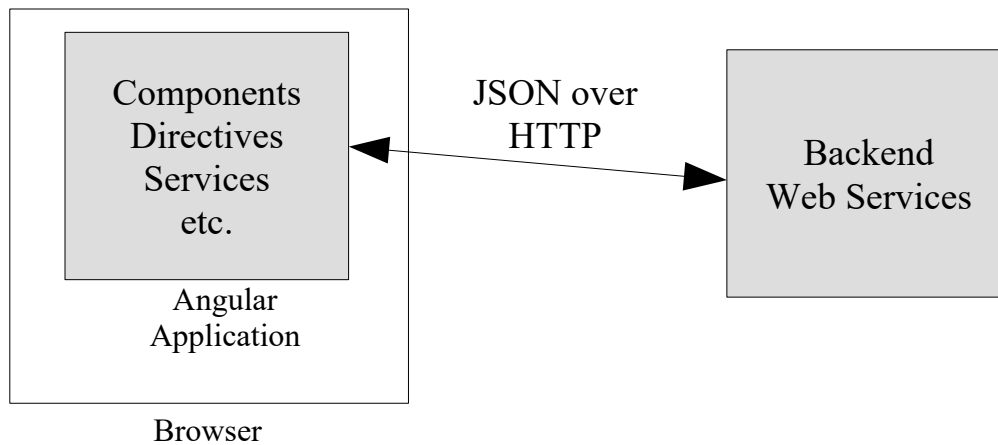
Notes

The building blocks of Angular include code that we create like Modules, Components and Services as well as techniques such as data binding and dependency injection.

While a simple application may only use a few of these building blocks any real-world apps will take advantage of most if not all of them.

As we make our way through the course each building block will be introduced in turn.

1.5 Basic Architecture of an Angular Application



Notes

Angular is a framework to build the front end GUI of a web application. It is developed and maintained independently of the backend very similar to the way one would develop an Android or iOS application.

Once fully compiled an Angular application becomes a collection of static files like JavaScript and CSS. It can then be deployed in a plain web server like Apache or Ngnix.

An Angular application would interact with the back end web services by making HTTP requests and exchanging JSON data. (Other data formats like XML or protobuf can also be used).

The only contract that exists between the front end and the back end is the specification for the web services. This specification includes the protocol (http, https), verb (GET, PUT, DELETE etc.), resource path and the JSON data structure.

Usually, the front and backend services will be developed independently and kept in separate source code control repositories.

1.6 Installing and Using Angular

- Angular development and build tools are written using Node.js. First install Node.js.
 - ◇ Note: Node.js is only needed during development. You don't need to install it in production machine.

- The Angular Command Line interface (CLI) tool makes it easy to create a project and install all the NPM packages. Install it.

```
npm install -g @angular/cli
```

- Then create a new Angular application project.

```
ng new register-app --defaults
```

1.7 Anatomy of an Angular Application

- **package.json** - contains all the frameworks and libraries the application depends on:
 - ◇ dependencies – These packages are used at runtime. Such as @angular/core and @angular/forms.
 - ◇ devDependencies – These packages are used during development only. Such as Typescript compiler.
- **angular.json** – The project descriptor file.
- **src/main.ts** – The main entry point for the application.
- **src/styles.css** – Global stylesheet file. Applies to the whole application.
- **src/index.html** – The main HTML file. The browser requests this file to load the whole application.
- **src/app/app.module.ts** – The application module file.
- **src/app/app.component.ts** – The root component.

1.8 Running the Application

- From the project's root folder run:

```
ng serve
```

- That will build the application and launch the development web server.
- From a browser enter <http://localhost:4200/>. Alternatively have the browser

opened for you:

```
ng serve --open
```

- When you change any code file, the server will automatically do a rebuild and refresh the browser.
- If you view the source of the page, you will see that the build system has added various JavaScript files at the end of index.html.

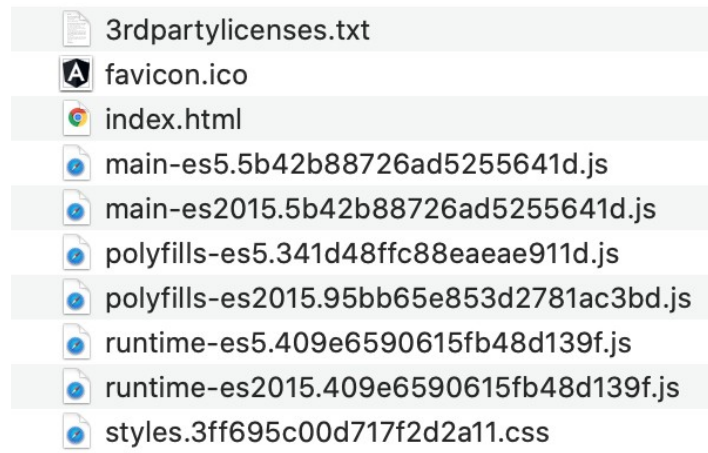
```
<body>
  <app-root></app-root>
<script type="text/javascript" src="inline.bundle.js">
</script>
<script type="text/javascript" src="polyfills.bundle.js">
</script>
...
</body>
```

1.9 Building and Deploying the Application

- When you are ready to deploy your application to QA or production, build the application by running this command from the project's root folder.

```
ng build
```

- This will output a modified index.html and a small number of highly efficient and compact JavaScript files in **dist/<app-name>** folder.



- These static files can now be simply copied to the document root of a web server to make your application available to the users.

1.10 Angular for Native Mobile Apps

- You can reuse most of an Angular app's code to create fully native mobile apps for iOS and Android. This is done using the Nativescript project.
 - ◇ <https://nativescript.org/>
- The same code base can build apps for the web, iOS and Android.
 - ◇ Only the template files need to be different.
 - ◇ Component class code, services and most other code will be reused across all platforms.

1.11 Summary

- In this chapter we:
 - ◇ Looked at a high level overview of Angular, what it is and why its used.
 - ◇ Saw how to install the Angular and its related JavaScript libraries.
 - ◇ Reviewed the building blocks of a basic Angular Application.

Chapter 2 - Introduction to TypeScript

Objectives

Key objectives of this chapter

- ◇ An Overview of TypeScript
- ◇ Defining Variables, Arrays and Objects
- ◇ Using Interfaces,
- ◇ Working with Modules
- ◇ Converting TypeScript to JavaScript
- ◇ Cover Arrow Functions & Template Strings
- ◇ Generics

2.1 Programming Languages for Use with Angular

- TypeScript is the primary programming language for Angular.
 - ◇ Dart support is now moved to the AngularDart project.
- TypeScript is a super-set of standard JavaScript.
- JavaScript - ES5
 - Widely supported by browsers
 - No type support
- JavaScript - ES6
 - Adds full object orientation
 - Still no type support
 - Not yet supported by all browsers
- TypeScript
 - ◇ Is "transpiled" into JavaScript (ES6 as of Angular 10) for use in browsers
 - ◇ Includes full type support
 - ◇ Includes full support for Object Orientation
 - ◇ Angular itself is written in TypeScript

Notes

Although JavaScript has been evolving even the latest version, ES6, does not include some features

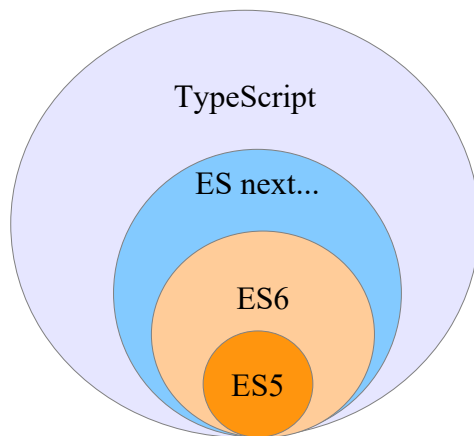
you might expect such as strongly typed variables and full object orientation.

Typescript is a strongly typed language that, due to its inclusion of types, catches many coding errors at compile time thus reducing defects. TypeScript code is compiled or converted into JavaScript code in a process called 'transpilation'.

TypeScript was used to write Angular itself.

This course will walk you through using TypeScript to code Angular apps.

2.2 TypeScript Syntax



TypeScript:

- Extends existing JavaScript
- Adds types and advanced object support
- Is easy to learn and use
- Can co-exist with legacy JS code
- Can be used on non-Angular projects as well

Notes

TypeScript is a superset of JavaScript that adds features which make it easier to write modular object oriented code. Time spent learning TypeScript is offset by an increase in productivity and decrease in defects. Developers do not need to learn all of TypeScript to start seeing these benefits. TypeScript plays well with existing JavaScript code and can be used on non-Angular projects as well.

2.3 Programming Editors

- While any text editor can be used to create TypeScript code the following editors include features that can make working with TypeScript easier:

WebStorm	\$	Web development IDE from JetBrains. Well regarded. Includes TS support.
Visual Studio Code	Free	Lightweight cross-platform editor from Microsoft that includes TS support. Also install the Angular Language Service plugin.

Sublime Text	\$	Programming editor. Supports TS via plugin.
Atom	Free	Open source text editor. TS support via plugin package.
Brackets	Free	Open source code editor. TS support via plugin extension.

2.4 The Type System – Defining Variables

- Standard JavaScript variables are untyped:

```
let x = 'stuff';      // string data
let y = true          // boolean data
var z = 33            // numeric data

x = y // OK. Type switching allowed.
```

- TypeScript variables are strongly typed:

```
let x: string = 'stuff';    // string data
var y: boolean = true      // boolean data
let z: number = 33         // numeric data
let a: any = 'hello' //any type

x = y // Compile error
let s: string = 25 //Compile error

a = y //OK
```

Notes

Variables in JavaScript can hold any type of data. This allows you to get going quickly but becomes a problem when you assign values of the wrong type to a variable, when you pass a variable of the wrong type to a function that requires a specific type or when you try to process an array with elements of various types.

TypeScript requires you to specify the variable type in the variable declaration:

```
var x: string = 'stuff';    //a string type
```

This line, which is fine in JavaScript, will produce an error at compile time in TypeScript:

```
var x: string = 25;        // assigns a number to a string var
```

If you do need to create a variable that holds various types of data you can do that using the 'any' type:

```
var a: any = 'stuff';      // any data
a = 35;                    // this works
```

2.5 The Type System – Defining Arrays

■ Arrays in standard JavaScript

```
let colors = ['red', 'white', 'blue'];
let values = [10, 20, 30];
let people = [{name:'John'}, {name:'Lisa'}];
```

■ Arrays in TypeScript

```
let colors: string[] = ['red', 'white', 'blue'];
let values: number[] = [10, 20, 30];
let names: Object[] = [{name:'John'}, {name:'Lisa'}];
```

Notes:

Arrays can have types too. Setting the type of an array restricts the kind of data that can be added to it. For example the following array can only hold numbers:

```
var my_values: number[] = [10,20,30];
```

This array can hold various types of objects:

```
var names: any[];
```

This array can only hold Person type objects:

```
var people: Person[];
```

We will take a closer look at Classes and Objects shortly.

2.6 Basic Primitive Types

■ Basic primitive types:

- ◇ boolean

- ◇ number
- ◇ string
- ◇ void - Declares absence of any variable. Usually only used as a return type for a function that returns nothing.
- ◇ null and undefined - Declares that variable can have null or undefined values.

2.7 Type in Functions

- Parameter and return types need to be specified.

```
function sayHello(name: string): string {  
    return `Hello ${name}!`  
}
```

```
var h1 = sayHello("Daffy Duck") //OK  
var h2 = sayHello(10) //Compile error  
var h3 = sayHello("Daffy Duck") //Compile error
```

2.8 Type Inference

- The compiler can infer types of variables from initial value assignment.

```
var x = "hello" // x is a string
```

```
x = 10 // Compile error.
```

```
var a // Uninitialized variable inferred as any type
```

```
a = "hello" // OK  
a = 10      // Also OK
```

- Function return type can be inferred.

```
function sayHello(name: string) {  
    return `Hello ${name}!`  
}
```

```
var h3: number = sayHello("Daffy Duck") //Compile error
```

- Explicitly specify types for better readability of your code.

2.9 Defining Classes

- Classes define a custom datatype:

```
class Cat {
    name: string = ""
    breed: string = ""
}

function meow(cat: Cat) {
    console.log(`${cat.name} says meow!`)
}

let c = new Cat

c.name = "Fluffy"
c.breed = "Persian"

meow(c)
```

2.10 Class Methods

- Does not use the **function** keyword.
- Must access properties using the **this** keyword.

```
class Cat {
    name: string = ""
    breed: string = ""

    meow() {
        console.log(`${this.name} says meow!`)
    }
}

let c = new Cat
```



```
c.name = "Fluffy"  
c.breed = "Persian"
```

```
c.meow()
```

2.11 Visibility Control

- Class methods and variables can be marked as **public**, **private** and **protected**.

```
class Employee {  
    name: string = "" //Public by default  
    private salary: number = 0.0  
    private giveRaise() {this.salary += 100.00}  
}
```

```
let e = new Employee  
e.giveRaise() //Compile error!
```

- **public** - By default items are public. They are accessible from outside the class.
- **private** - Items are accessible only from within the class.
- **protected** - Items are accessible from within the class and any other class that extends from it.

2.12 Class Constructors

- TypeScript allows only one constructor implementation.

```
class Building{  
    address: string;  
    units: number;  
  
    constructor(address: string, units: number){  
        this.address = address;  
        this.units = units;  
    }  
}
```

```
}

var bld1 = new Building("1 main street", 4);
var bld2 = new Building("13 park ave.", 5);

var properties: Building[] = [bld1, bld2];
```

2.13 Class Constructors – Alternate Form

- This Class works just like the one on the previous slide:

```
class Building{
    constructor(public address: string,
               public units: number){}
}
var bld1 = new Building(`1 main street`, 4);
console.log(`${bld1.address}, ${bld1.units}`);
```

- Note that it does not explicitly define properties.
- Adding visibility qualifiers to the inputs of the constructor also defines them as class properties.

```
public address: string,
public units: number
```

Notes

The syntax shown here is commonly used when defining classes.

2.14 Uninitialized Fields

- Member variables of a class must be initialized from the constructor or from the variable declaration.
- Uninitialized fields must be made either optional or be asserted that the field will be somehow initialized prior to use.

```
class Cat {
    name = "" //Initialized here
    id: number //Initialized in the constructor
    breed!: string //Assertion about initialization
```

```
price?: number //Optional. May be undefined.

constructor(id: number) {
    this.id = id
}
}
```

2.15 Interfaces

- Interfaces are used to define classes and types:

```
interface ITrip{
    destination: string;
    days: number;
    display();
}

class BizTrip implements ITrip {
    constructor(public destination: string,
                public days: number){}
    display(){
        console.log(`${this.destination}, ${this.days}`);
    }
}

var trip = new BizTrip("New York", 3);
trip.display();
```

Notes

The interface ITrip defines a template for creating classes and can be used as a type to define variables. Once we've created the interface we can create any number of classes from it and use those classes to construct objects that satisfy the interface. Using our ITrip interface we could create classes such as, Vacation, CampingTrip, FestivalTrip etc. Objects made from these classes can be assigned to variables with the type equal to the interface type. In addition any of these various objects can be passed into a function that takes the ITrip interface type:

```
var vacation = new Vacation("Hawaii", 5);

function showTrip(trip: ITrip){
    trip.display();
}
```

```
showTrip(vacation);
```

2.16 Working with ES6 Modules

- A module is basically a file containing classes, functions and variables. You need to export items from the file to be used elsewhere in the app.

```
// person.ts
export class Person {
  constructor(public fname, public lname){}
  display(){console.log(`${fname}, ${lname}`);}
}
export var settings = {...}
```

- You need to import items from another module to use them.

```
// app.ts
import {Person} from './person';

var p1 = new Person("Joe", "Smith");
p1.display();
```

Notes

Notice how the term "export" is added before the class definition.

"export" is used to make a class, variable or function visible outside the file where it is defined.

"import" is used to access the exported item from inside a separate file.

Notice how we reference the name of the class we are importing. Also note that the extension is left off of the name of the module file when defining the 'from' clause.

```
'./person'
```

2.17 var vs let

- **let** defines a variable within a scope. **var** defines it in the scope of the function. Use let wherever possible.

```
function test() {
  let i = 20

  if (i > 0) {
```

```
    let i = 10 //A new i
    let j = 5
    var k = 15
  }

  console.log(i) //Prints 20 not 10
  console.log(j) //Error: j not available here.
  console.log(k) //Prints 15. k is available.
}
```

2.18 Arrow Functions

- Arrow function is a way to define higher order functions that can be passed to other functions as argument and invoked at a later time:

```
let list = [1, 12, 5, 7, 20]
let evenNumbers = list.filter((n: number) => {
  if (n % 2 == 0) {
    return true
  } else {
    return false
  }
})

console.log(evenNumbers) // Prints [ 12, 20 ]
```

Notes

In the example above we are passing an arrow function to the filter() method. It takes as input a number and returns a boolean value. The filter() method includes the number if the arrow function returns true.

2.19 Arrow Function Compact Syntax

- Typescript can infer the parameter and return types of arrow functions. This results in less typing.

```
let evenNumbers = list.filter(n => {
  if (n % 2 == 0) {
    return true
  } else {
```

```
    return false
  }
})
```

- If the arrow function has only one statement then the "return" keyword and `{ }` can be omitted.

```
let evenNumbers = list.filter((n) => n % 2 == 0)
```

- If the arrow function takes only one parameter then parenthesis can be omitted.

```
let evenNumbers = list.filter(n => n % 2 == 0)
```

2.20 Template Strings

- Defined using a pair of backticks. Allows expressions to be embedded inside such a string. Example:

```
var name = "Harold" //Regular string
var x = `Hello There ${name}` //Template string
console.log(x) //Prints: Hello There Harold
```

- Template strings can be multi-line:

```
var html = `
April is the cruellest month, breeding
Lilacs out of the dead land, mixing
Memory and desire, stirring
`;
```

2.21 Generics in Class

- Generics allow type specification to be deferred from when a class is created to when the code using the class is written.

```
class Stack<T> {
  list: T[] = []
```

```
push(item: T) {
    this.list.push(item)
}
pop(): T|undefined {
    return this.list.pop()
}
}
```

```
let s = new Stack<number>() // T is now a number
s.push(10)
s.push("Hello") // Error!
```

2.22 Generics in Function

- You can also write functions that defer type specification. The function below returns an array filled with two items:

```
function arrayFilled<T>(item1: T, item2: T): T[] {
    let list = new Array<T>()

    list.push(item1)
    list.push(item2)

    return list
}

let numList = arrayFilled<number>(1, 2)
let strList = arrayFilled<string>("One", "Two")
let list = arrayFilled<boolean>(true, 100) //Compile error!
```

2.23 Summary

In this chapter we covered:

- ◇ An Overview of TypeScript
- ◇ Defining Variables, Arrays and Objects
- ◇ Using Interfaces,

- ◇ Working with Modules
- ◇ Arrow Functions & Template Strings
- ◇ Generics

Chapter 3 - Components

Objectives

Key objectives of this chapter

- What is a Component?
- A Basic Component
- HTML Templates
- CSS Styling
- Hooking up Inputs
- Hooking up Buttons
- Component Decorator Properties,
- Lifecycle Hooks

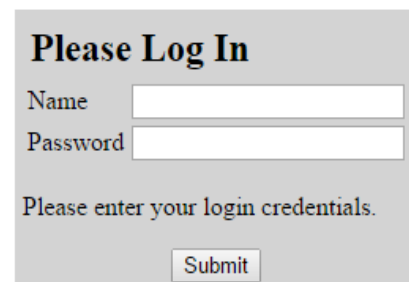
3.1 What is a Component?

Components:

- Are TypeScript classes used to write the GUI display logic.
- They display data using:
 - ◇ HTML Templates
 - ◇ CSS Styles
- They accept user input and validate them.
- They handle events occurring within the component's DOM.

Components Can:

- Use Other Components,
- Use Directives,
- Use Services



Please Log In

Name

Password

Please enter your login credentials.

A Login Component

3.2 An Example Component

The Login Component Includes:

- Two Input Fields
- An Output Message that changes when the user submits
- Submit Button



Please Log In

Name

Password

Jackie is now logged in!

Login Component

3.3 Creating a Component Using Angular CLI

- Basic command.

```
ng generate component hello
```

- This will create these files:
 - ◇ **src/app/hello/hello.component.ts** – The component class file.
 - ◇ **src/app/hello/hello.component.html** – HTML template for the component.
 - ◇ **src/app/hello/hello.component.css** – Component's own stylesheet.
 - ◇ **src/app/hello/hello.component.spect.ts** – Unit test script.
- This will also add the component to the application module's declarations list.

3.4 The Component Class

- A component is a regular TypeScript class decorated with the **@Component** decorator.

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

```
@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {
  constructor() { }
  ngOnInit(): void {}
}
```

3.5 The @Component Decorator

- Designates a class as a component.
- Has these key properties:
 - ◇ **selector** – The CSS selector that will be used to add the component to a template. This is usually a tag name like "app-hello".
 - ◇ **templateUrl** – Location of the component's template.
 - ◇ **styleUrls** – A list of CSS files that contain styles private to the component.

3.6 Registering a Component to Its Module

- Every component class must be added to the **declarations** list of its module. CLI does this for us.

```
import { HelloComponent } from
  './hello/hello.component';

@NgModule({
  declarations: [
    HelloComponent, ...
  ],
  ...
})
export class AppModule { }
```

3.7 Component Template

- A component uses its template to render display.
- For web applications, the template will output HTML DOM elements. For native mobile applications the template will create native iOS or Android widgets.
- Templates look like plain HTML with some embedded code inside. This makes it easy to convert HTML delivered by a designer into templates.
- Templates get compiled into JavaScript code by the Ahead of Time (AOT) compiler. As a result most coding errors in a template should be detected at build time.

3.8 Example: HelloComponent Template

- `src/app/hello/hello.component.html`:

```
<div>
  <p>{{greeting}}</p>
  <button (click)="showGreeting()">Click Me!</button>
</div>
```

- We see some dynamic code in the HTML.
 - ◇ **{{greeting}}** – Shows the member variable `greeting` of the component class.
 - ◇ **(click)="showGreeting()"** - Sets up a click event handler for the button.

3.9 Example: The HelloComponent Class

```
export class HelloComponent implements OnInit {
  greeting = "Please click the button below"

  showGreeting() {
    this.greeting = "Hello there!"
  }
}
```

```
constructor() { }  
ngOnInit(): void {}  
}
```

3.10 Using a Component

- To use a component add its selector tag to the parent component's template.
- For example we can add HelloComponent to the root (AppComponent) component's template src/app/app.component.html:

```
<app-hello></app-hello>
```

3.11 Run the Application

- Run **ng serve** and access <http://localhost:4200/>.

Please click the button below

Click Me!

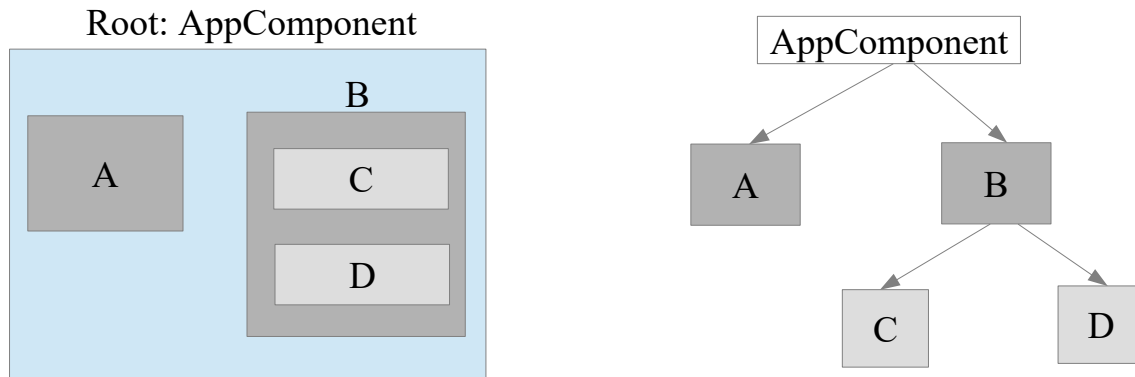
- After you click the button the display will show:

Hello there!

Click Me!

3.12 Component Hierarchy

- Every application has a root component created as **AppComponent** class in **app.component.ts**.
- A component can use other components from its template. This builds a tree like hierarchy.



3.13 The Application Root Component

- The root component is added to the **index.html** file.

```
<body>
  <app-root></app-root>
</body>
```

- The root component is marked as a bootstrap (entry point) component for the application module.

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

- When the module is bootstrapped into **index.html** the root component gets inserted to the page where the corresponding tag appears.

3.14 The Bootstrap File

- **main.ts** attaches (bootstraps) the application module into **index.html**.

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

Notes

This file is generated by Angular CLI. Usually you never manually modify this file.

When you are just getting started this business with bootstrapping may seem like over engineering. But there is a reason behind this. Angular was designed to eventually create native mobile apps. Here we are attaching the AppModule to the web browser. In the case of a native application, we will bootstrap it slightly differently.

3.15 Component Lifecycle Hooks

- Angular actively manages components; It creates them, renders views, monitors changes to properties, and destroys them.
- Specific "lifecycle hook methods" are called at defined times :
 - ◇ `ngOnInit()`: After the component instance is created and all dependency injections have been satisfied. Use this callback to do one time initialization of the component instead of the constructor.
 - ◇ `ngOnDestroy()`: Just before Angular destroys the component
 - ◇ `ngOnChanges()`: After Angular updates any data-bound member variable
 - ◇ `ngAfterViewInit()`: After all the children components have been initialized
- Developers can take advantage of these to run code at specific times.

Notes

This is just a short list. For a full list of available lifecycle methods see the angular documentation:

<https://angular.io/guide/lifecycle-hooks>

3.16 Example Lifecycle Hooks

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

@Component({...})
export class HelloComponent implements OnInit, OnDestroy {
```

```
constructor() { }  
ngOnDestroy(): void {  
}  
  
ngOnInit(): void {  
}  
}
```

3.17 CSS Styles

- Styles that are private to a component should go inside the component's own CSS file. For example **src/app/hello/hello.component.css**.

```
div {  
  border: solid thin black;  
  max-width: 300px;  
  padding: 5px;  
}
```

- Here even though we are setting a style for the `<div>` tag it will be applied only within the `HelloComponent`. Other components on the page will not be affected, not even the children components.
- Styles that are common across all components should be put in **src/styles.css**.

3.18 Summary

Topics covered in this chapter included:

- What is a component?
- A basic component
- HTML templates
- CSS styling
- Basic event handling
- `@Component` decorator properties,

- Lifecycle hooks

Chapter 4 - Component Templates

Objectives

Key objectives of this chapter

- Learn details about component template design.
- Learn the mustache {{ }} syntax.
- Learn how to set DOM element attributes.
- Learn how to handle DOM events.
- Manipulate DOM elements using attribute directives like **ngClass**.
- Control template execution using structural directives like **ngFor** and **ngIf**.

4.1 Templates

- Every component has a template that it uses to render display on screen.
- Templates are compiled into JavaScript code during build time.
- Execution of a template produces HTML elements that are inserted into the DOM of the page.
- A template:
 - ◇ Contains HTML code.
 - ◇ Displays data using the mustache {{ }} syntax.
 - ◇ Sets DOM element attributes using the [] syntax.
 - ◇ Sets up DOM event handling using the () syntax.
 - ◇ Uses attribute directives that operate on a specific DOM element. For example, **ngClass** and **ngStyle** set CSS styles to an element.
 - ◇ Uses structural directives that can conditionally execute a chunk of template. For example **ngFor** can repeatedly execute some template to render a list. And **ngIf** and **ngSwitch** can conditionally exclude portions of a template from execution.

4.2 Template Location

- A template is usually defined outside of the component class and referred to using the **templateUrl** property.

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

- Alternatively, you can define the template inline.

```
@Component({
  selector: 'app-hello',
  template: `
    I say {{greeting}}
  `,
  styleUrls: ['./hello.component.css']
})
```

4.3 The Mustache {{ }} Syntax

- Renders member variable, method output and expression results.

```
<p>First: {{firstName}}</p>
<p>Last: {{lastName}}</p>
<p>Full: {{fullName()}}</p>
<p>Slogan: {{lastName == 'Duck' ? 'Ducks Rule' : ''}}</p>
```

```
export class HelloComponent {
  firstName = "Daffy"
  lastName = "Duck"

  fullName() : string {
    return this.firstName + " " + this.lastName
  }
}
```

```
}
```

4.4 Setting DOM Element Properties

- The DOM API defines various properties for an element like, src, href, disabled, innerHTML and className. Angular lets you set these properties using the [] syntax.

```
<img [src]="imageUrl"/>
<p><a [href]="website">View Website</a></p>
<button [disabled]="!policyAccepted">Submit</button>

export class HelloComponent {
  imageUrl = "https://www.gstatic.com/webp/gallery/5.jpg"
  website = "https://www.jaipurmaharajabrassband.com/"
  policyAccepted = false
}
```

- Properties with simple string values can also be set using {{ }}.

```

```

4.5 Setting Element Body Text

- You can set element body text using either [innerHTML] or using {{ }}.

```
description = 'Hello <b>World</b> <script>alert('Hacked')</script>'
```

```
<div [innerHTML]="description"></div>
<div>{{description}}</div>
```

- innerHTML will preserve the HTML formatting in the string but remove harmful elements like <script> tags.

Hello World

- {{ }} escapes HTML tags and will literally display the string as is.

```
Hello <b>World</b> <script>alert('Hacked')
</script>
```

4.6 Event Binding

- Use the () syntax to bind a component method to any DOM event emitted by view elements. Such as click, keydown, mouseover, etc.
- You can access the DOM event object using the special variable **\$event**.

```
<div (mousemove)="showCoordinates($event)">
X: {{xCoord}} Y: {{yCoord}}
</div>
```

```
export class HelloComponent {
  xCoord = 0
  yCoord = 0

  showCoordinates(e:MouseEvent) {
    this.xCoord = e.clientX
    this.yCoord = e.clientY
  }
}
```

4.7 Expression Event Handler

- You can bind an expression as the event handler. Avoid writing complicated logic this way. Move them to a method instead.

☒ Accept policy ☐ Accept policy

```
<input type="checkbox"
[checked]="policyAccepted"
(change)="policyAccepted = $event.target.checked"/>
Accept policy
<button [disabled]="!policyAccepted">Register</button>
```

```
export class HelloComponent {  
  policyAccepted = true  
}
```

4.8 Prevent Default Handling

- You may need to prevent default event handling in some cases:
 - ◇ A click on a hyperlink will do navigation.
 - ◇ A click on a button inside a form will submit the form.

```
<a href (click)="showPolicy($event)">Show Policy</a>
```

```
showPolicy(e:Event) {  
  //Do something...  
  //...  
  
  //Prevent navigation  
  e.preventDefault()  
}
```

4.9 Attribute Directives

- Attribute Directives are TypeScript classes that are used to manipulate DOM elements. Unlike components, they do not output any DOM structure of their own but work on existing elements.
- They are applied to elements from a component's template. Example:

```
template: `  
  <p importantText>Register Here Awesome Deal!</p>  
`,
```

- Angular comes with a few useful attribute directives: **NgStyle**, **NgClass**.

Notes:

Attribute directives are used to capture DOM manipulation logic that appears in multiple places in your application. For example, if you need to style important text in a certain way you can create a directive for it and apply it to different elements. Of course, that can also be done by CSS styles. But attribute directives can do more. They can also handle events emitted by the elements where they are applied.

4.10 Apply Styles by Changing CSS Classes

- The **ngClass** directive can be used to dynamically apply CSS classes to an element.

```
<div [ngClass]="expression" > ... </div>
```

- The expression can be:

Type	Example	Comments
string	"active bordered"	A space separate CSS class names
Array	["active", "bordered"]	An array of class names
Object	{ active: isActive, bordered: hasBorder }	Property name is a CSS class. Value is a boolean condition. If condition is true the class is applied.

4.11 Example: ngClass

Show Policy

Lorem, ipsum...

- Component CSS:

```
.policy-hidden {  
  visibility: hidden;  
}
```

- Template:


```
<button (click)="policyShown = true">Show Policy</button>
<p [ngClass]="{'policy-hidden': !policyShown}">
  Lorem, ipsum...
</p>
```

- **Code:**

```
export class HelloComponent {
  policyShown = false
}
```

4.12 Applying Styles Directly

- The **ngStyle** directive can be used to directly apply styles to HTML elements. Syntax:
 - ```
<div [ngStyle]="styleObject"></div>
```
- Property names of the style object are CSS style names. Values are style values. This example changes background color dynamically:

```
<div [ngStyle]="{'background-color': color}">
 I can be styled
</div>
<select (change)="color=$event.target.value">
 <option>Gold</option>
 <option>Violet</option>
 <option>Ivory</option>
</select>
```

```
export class HelloComponent {
 color = 'Gold'
}
```

## 4.13 Structural Directives

- A structural directive wraps around a portion of a component's template and do things like:
  - ◇ Conditionally run the template or exclude it from execution. **ngIf** and

**ngSwitch** do this.

- ◊ Repeatedly execute that portion of the template to generate multiple instances of DOM elements for that portion. **ngFor** does this.

## 4.14 Conditionally Execute Template

- The **ngIf** directive can be used to remove and restore DOM elements.
- It is often associated with a boolean variable from the component class. When the condition is false the template associated with **ngIf** is excluded from execution.

```
<div *ngIf="showMessage">Message Text</div>
```

- The asterisk (\*) prefix is used with all structural directives. It tells us that we are using a shorthand version. The full syntax shows how an **ng-template** is created for the **ngIf** directive:

```
<ng-template [ngIf]="showMessage">
 <div>Message Text</div>
</ng-template>
```

- Use **ngIf** to show/hide elements instead of CSS classes. Because **ngIf** suppresses portion of the template from execution templates execute faster and produce smaller DOM.

## 4.15 Example: **ngIf**

☐ Accept policy

Please accept policy to register with us.

Register

- Template:

```
<input type="checkbox"
 [checked]="acceptPolicy"
 (change)="acceptPolicy=$event.target.checked"/>
Accept policy
```

```
<p *ngIf="!acceptPolicy">Please accept policy to register
with us.</p>
```

```
<button [disabled]="!acceptPolicy">Register</button>
```

- **Code:**

```
export class HelloComponent {
 acceptPolicy = false
}
```

## 4.16 Looping Using ngFor

- **ngFor** loops over a JavaScript array and executes a section of template repeatedly.

Fiffy (Cat)

Fido (Dog)

Mimi (Turtle)

```
<p *ngFor="let p of pets">{{p.name}} ({{p.type}})</p>
```

```
export class AppComponent {
 pets = [
 {id: 100, name: "Fiffy", type: "Cat", age: 2},
 {id: 123, name: "Fido", type: "Dog", age: 1},
 {id: 561, name: "Mimi", type: "Turtle", age: 2}]
}
```

- "let p" creates a local variable for each item in the pets collection.

## 4.17 ngFor Local Variables

- ngFor provides several loop-related local variables for use in the repeated template:

index, first, last, even, odd,

- Usage:

```
<li *ngFor="let pet of pets;
 let i = index; let e = even;"
 [ngClass]="{evenrow:e}" >
 {{i}}-{{pet.type}}

```

### Pets - local vars

0-Cat  
1-Dog  
2-Hamster  
3-Rabbit  
4-Fish  
5-Bird  
6-Turtle

- In the above code index and even are assigned to local variables and then used in the template

## Notes

**Index** is a number type while **First**, **Last**, **Even** and **Odd** are boolean types.

**Index** contains the array index used to retrieve the current item from the backing array.

**First** is true when the current item is the first in the backing array.

**Last** is true when the current item is the last in the array.

**Even** and **Odd** are true based on the current items position in the array.

## 4.18 Manipulating the Collection

- ngFor constantly watches the collection it iterates over. It behaves as follows:
  - ◇ If a new object is added ngFor runs the template for it and adds the resulting elements to the DOM.
  - ◇ If an object is removed ngFor removes the elements that were generated for it from DOM.
  - ◇ If the objects are re-ordered in the collection ngFor simply reorders their elements in the DOM.

- This kind of surgical change to the DOM creates a very efficient and smooth display.

## 4.19 Example - Deleting an Item

- Component template:

```
<p *ngFor="let p of pets">
 {{p.name}} ({{p.type}})
 <button (click)="deletePet(p)">Delete</button>
</p>
```

- Component code:

```
deletePet(petToDelete) {
 this.pets = this.pets.filter(
 p => p.id !== petToDelete.id)
}
```

## 4.20 Item Tracking with ngFor

- When you add or remove items to a list ngFor attempts to reuse the DOM elements for the existing items. For this to work ngFor needs to uniquely identify each item. By default it uses JavaScript object identity. But if you create a new array (by refreshing the list from the backend service for example) ngFor needs to recreate all the DOM. To solve this problem use **trackBy** to specify a unique identifier for the items.

```
<p *ngFor="let p of pets; trackBy: trackByPetId">
 {{p.name}} ({{p.type}})
</p>
```

```
export class HelloComponent implements OnInit {
 pets = [...]
 trackByPetId(index:number, pet:any) : number {
 return pet.id
 }
}
```

```
}
```

## 4.21 Swapping Elements with ngSwitch

- ngSwitch works along with ngSwitchCase and ngSwitchDefault to execute one of many possible templates depending on the condition.
- Example template:

```
<div [ngSwitch]="selectedPage">
 <p *ngSwitchCase="1">Page-1</p>
 <p *ngSwitchCase="2">Page-2</p>
 <p *ngSwitchCase="3">Page-3</p>
 <p *ngSwitchDefault>Default</p>
</div>
<button (click)="next()">Next</button>
```

- Code:

```
export class MyComponent {
 selectedPage = 1
 next() {this.selectedPage += 1}
}
```

## 4.22 Grouping Elements

- You may need to control a group of elements using ngIf or ngFor. You could wrap them in a <div> tag.

```
<div *ngFor="...">
 <h1>Title</h1>
 <div>Description</div>
</div>
```

- But this may have styling consequences such as extra spacing and margin. A better way is to wrap them in **ng-container**.

```
<ng-container *ngFor="...">
 <h1>Title</h1>
 <div>Description</div>
</ng-container>
```

## 4.23 Template Reference Variable

- You can get a reference to a DOM element, component and directive instance using the **#variableName="selector"** syntax.
- Getting reference of the DOM element:

```
<input type="text" #userId/>

<button (click)="handleClick(userId)">Submit</button>
```

```
handleClick(userId:HTMLInputElement) {
 console.log("You entered:", userId.value)
}
```

- A reference to a component or directive gives you direct access to their properties and methods. Below we use the `ngModel` directive which we will learn more about in a future chapter.

```
<input type="text" #userId="ngModel" ngModel
 required minlength="3"/>

<p *ngIf="userId.invalid">Please enter a valid user ID.</p>
```

## 4.24 Summary

Topics covered in this chapter included:

- Mustache `{{ }}` syntax displays class variables and expression values.
- DOM element attributes can be set using `[ ]`.
- DOM events are handled using `( )` syntax.
- Manipulate DOM elements using attribute directives like `ngClass`.
- Control template execution using structural directives like `ngFor` and `ngIf`.





## Chapter 5 - Inter Component Communication

---

### *Objectives*

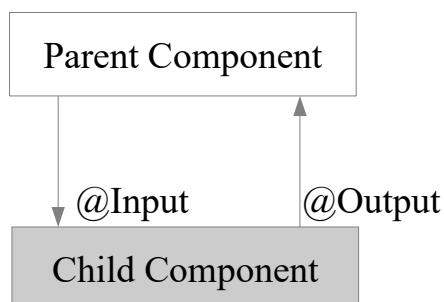
Key objectives of this chapter

- Learn how parent and child components can communicate.
- How a parent can send data to a child using @Input
- How a child can notify events to parent using @Output
- How to setup two way data binding between parent and child

### 5.1 Communication Basics

- Angular allows immediate parent and child components to communicate with each other through data and event binding.
- A parent can set properties of a child using data binding (@Input decorator).
- A child can supply data and events to its parent through event binding (@Output decorator).
- The same mechanism applies between a parent component and any directives used in its template. This is outside the scope of this chapter.

### 5.2 The Data Flow Architecture



### 5.3 Preparing the Child to Receive Data

- The child component needs to decorate its member variables with @Input

decorator:

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

export class HelloComponent {
 @Input()
 greeting = ""
 @Input()
 user = {name: "", email: ""}

 isValid = false //Internal variable not decorated
}
```

- Use the variables in the template as usual:

```
<p>{{greeting}}</p>
<p>Name: {{user.name}} Email: {{user.email}}</p>
```

## 5.4 Send Data from Parent

- Parent uses the data binding [ ] syntax to supply values.

```
export class AppComponent {
 message = "Hello earthling"
 human = {name: "Count Dooku", email: "cd@example.com"}
}
```

```
<app-hello [greeting]="message" [user]="human"></app-hello>
```

- This creates a live data binding. Meaning, if the values of "message" or "human" variables change the updated values will be again sent to the child.

## 5.5 More About Setting Properties

- If the property type is string and you wish to hard code the value in template you can use a simpler syntax (drop the [ ]).

```
<app-hello greeting="Hello earthling" [user]="human"></app-hello>
```

- You can expose the property using a different name than the class variable.

```
export class HelloComponent {
 @Input("greeting-text")
 greeting = ""
}
```

```
<app-hello [greeting-text]="message" [user]="human">
</app-hello>
```

## 5.6 Firing Event from a Component

- A component can fire custom events just like a button fires the **click** event.
- A parent component can attach an event listener using the syntax (event\_name)="handler()".
- The event can carry a payload object. This payload is used to output data from the child component to the parent.
- A component fires an event using an **EventEmitter** object that is decorated with **@Output**.

```
@Output() myEvent = new EventEmitter();
...
this.myEvent.emit(payload_object)
```

## 5.7 @Output() Example - Child Component

- The child component triggers the "on-registration" event every time the button is clicked. It does this by calling emit() on the EventEmitter.

```
<input type="text" #email/>
<button (click)="registerUser(email.value)">Subscribe
</button>
```

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

export class HelloComponent {
 @Output("on-registration")
 emitter = new EventEmitter()

 registerUser(email:string) {
 this.emitter.emit(email)
 }
}
```

## 5.8 @Output() Example - Parent Component

- The parent component listens for the "on-registration" event. It receives the payload using the **\$event** variable.

```
<app-hello (on-registration)="register($event)">
</app-hello>
```

```
export class AppComponent {
 register(email) {
 console.log("Registering user:", email)
 }
}
```

## 5.9 Full Two Way Binding

- In two way data binding the parent sets the initial value of a child's property. Then it receives any changes made to that property.
- To do this call the EventEmitter variable name @Input variable name + "Change".
- Below is the child component code.

```
<input type="text" [value]="age" #ageInput/>
<button (click)="enterAge(ageInput.value)">Go</button>
```

```
export class HelloComponent implements OnInit {
 @Input() age = 0
 @Output() ageChange = new EventEmitter()

 enterAge(age:string) {
 this.ageChange.emit(parseInt(age))
 }
}
```

## 5.10 Setting up Two Way Data Binding in Parent

- Parent uses the [( )] syntax to both send value to child and receive updates.

```
<app-hello [(age)]="myAge"></app-hello>
<p>You have entered: {{myAge}}</p>
```

```
export class AppComponent {
 myAge = 25
}
```

- The text box will initially show 25. If user changes the age and clicks on Go button the parent will get the updated value.

25	Go	45	Go
----	----	----	----

You have entered: 25

You have entered: 45

## 5.11 Summary

- Parent sends data to a child using the @Input mechanism. This creates a live data binding.
- Child notifies parents of any events and supply payload using @Output.
- Parent can set the initial value of a child's property and then receive any updates made to that property using two way data binding.



## Chapter 6 - Template Driven Forms

---

### *Objectives*

Key objectives of this chapter

- What is a Template Driven Form
- Binding input fields to component properties
- Binding input fields to form.value
- Bindings for common input fields
- Basic Validation
- Displaying and using validation state
- Accessing data via the form object
- Submitting forms

### 6.1 Template Driven Forms

- Angular offers two approaches to creating forms:
  - ◇ Template Driven Forms
  - ◇ Reactive Forms
- They provide different ways to achieve the same basic functionalities:
  - ◇ Initialize the form
  - ◇ Accept user input
  - ◇ Perform validation
- Template Driven Forms can be created entirely in the template and requires little or no coding. But they are harder to unit test (require the use of end-to-end testing methods).

#### Notes:

In this chapter, we take a look at Template Driven Forms plus some functionality that works in both types of forms.

The first thing we will look at is the [(ngModel)] bindings which not only give us access to field values but also to validation states.

In a Template Driven Form all validation requirements are defined using HTML attributes like; required, minlength, maxlength, etc.

For applications that require custom validation rules Reactive Forms will be easier. We cover Reactive Forms in a later chapter.

End-to-end testing methods refer to those which require access to the DOM and manipulation of the user interface which are difficult to automate. This is opposed to something like unit testing in which your tests are executed at the class level, do not require DOM access, and are easy to automate.

## 6.2 Importing Forms Module

- To use template driven forms you need to import the **FormsModule** from the app.module.ts file:
  - ◇ Import the module name:

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

- ◇ Update the @NgModule imports array:

```
@NgModule({
 imports: [BrowserModule, FormsModule],
```

### Notes:

Full contents of app.module.ts file to enable template driven forms.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
 imports: [BrowserModule, FormsModule],
 declarations: [AppComponent],
```



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

## 6.3 Basic Approach

- Use the **ngForm** directive with the `<form>` tag. Using this we can get the:
  - ◇ Overall validation status of the form.
  - ◇ Collection of all the user inputs in the form in a single object.
- Use the **ngModel** directive with every input element. Using this we can get the:
  - ◇ Validation status of the field.
  - ◇ Value of the field.
- Every input element must have the "name" attribute.

## 6.4 Setting Up a Form

```
<form ngForm #theForm="ngForm" (submit)="doLogin(theForm)">
 <input type="text" name="userId" ngModel/>
 <input type="password" name="password" ngModel/>

 <button>Login</button>
</form>
```

- Note:
  - ◇ We supply a reference to the `ngForm` directive to the submit handler. We need that to get user input.
  - ◇ Every input must have a name.

## 6.5 Getting User Input

- The **NgForm.value** object gives you all user input in a single object. The property names of the object are same as the "name" attribute used for an

input.

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

export class LoginComponent {
 doLogin(form:NgForm) {
 //Access user's input
 let userId = form.value.userId
 let pass = form.value.password
 }
}
```

## 6.6 Omitting ngForm Attribute

- ngForm directive declares "form" as the selector. Hence every <form> tag automatically gets an instance of ngForm. You don't have to explicitly use the ngForm directive attribute. Example:

```
<form #theForm="ngForm" (submit)="doLogin(theForm)">
 <input type="text" name="userId" ngModel/>
 <input type="password" name="password" ngModel/>

 <button>Login</button>
</form>
```

## 6.7 Initialize the Form

- You can pre-populate an input by binding the value of ngModel directive.

```
<form #theForm="ngForm" (submit)="doLogin(theForm)">
 <input type="text" name="userId" [ngModel]="uId"/>
 <input type="password" name="password" [ngModel]="pass"/>

 <button>Login</button>
</form>
```

```
export class LoginComponent {
```

```
uId = "Daffy" //Initial values
pass = "pa$$word"
}
```

## 6.8 Two Way Data Binding

- Two way data binding provides an alternate way to get user input.

```
<form (submit)="doLogin()">
<input type="text" name="userId" [(ngModel)]="uId"/>
<input type="password" name="password" [(ngModel)]="pass"/>

 <button>Login</button>
</form>

export class LoginComponent {
 uId = "Daffy" //Initial values
 pass = "pa$$word"

 doLogin() {
 //Get user's input
 console.log("User:", this.uId, "Password:", this.pass)
 }
}
```

## 6.9 Form Validation

- Both `ngForm` and `ngModel` have properties like **valid** and **invalid** that will tell you about their validation status.

```
<form #theForm="ngForm" (submit)="doLogin(theForm)">
 <input type="text" name="userId"
 #uId="ngModel" ngModel required/>
 <input type="password" name="password"
 #pass="ngModel" ngModel required/>

 <p *ngIf="uId.invalid">Invalid user ID.</p>
 <p *ngIf="pass.invalid">Invalid password.</p>
```

```
<button [disabled]="theForm.invalid">Login</button>
</form>
```

## 6.10 Angular Validators

- Angular has several built-in validators for use in template driven forms:

- ◇ required
- ◇ minLength
- ◇ maxLength
- ◇ pattern

- Example Usage:

```
<form>
 <input type="text" required name="name"
 [(ngModel)] =name >
 <input type="text" minlength="3" name="street"
 [(ngModel)] =street >
 <input type="text" maxlength="10" name="city"
 [(ngModel)] =city >
 <input type="text" pattern="[0-9]{5}"
 name="zip" [(ngModel)] =zip>
</form>
```

### Notes:

It is also possible to combine multiple validators in the same input element:

```
<input type="text" required minlength="3" maxlength="10" name="name" [(ngModel)] =name >
```

## 6.11 Displaying Validation State Using Classes

- These classes are added to input fields by Angular based on the current state of the field:

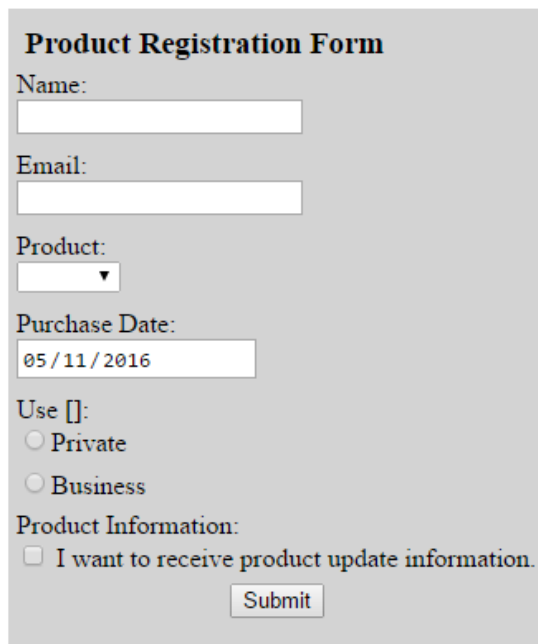
Control's State	Class if true	Class if false
Input field has been visited (has had focus)	ng-touched	ng-untouched

Value has been changed	ng-dirty	ng-pristine
Value is valid	ng-valid	ng-invalid

- CSS Styles can be created to take advantage of these classes

```
input.ng-invalid{ border: 1px solid red;}
input.ng-valid{ border: 1px solid blue; }
input.ng-pristine{ background-color: lightgrey;}
```

## 6.12 Additional Input Types



The screenshot shows a 'Product Registration Form' with the following fields and controls:

- Name:** A text input field.
- Email:** A text input field.
- Product:** A dropdown menu.
- Purchase Date:** A date input field showing '05/11/2016'.
- Use []:** Two radio buttons labeled 'Private' and 'Business'.
- Product Information:** A checkbox labeled 'I want to receive product update information.'
- Submit:** A button at the bottom right.

- Our basic form included two text type input fields.
- Other input types exist including:
  - ◇ Checkboxes
  - ◇ Select (dropdown)
  - ◇ Dates
  - ◇ Radio Buttons

### Notes

Forms often use various types of input fields other than the basic text field. In the next part of this chapter, we will look at how to integrate these types of input fields into an Angular application.

## 6.13 Checkboxes

- Checkboxes use a simple ngControl binding

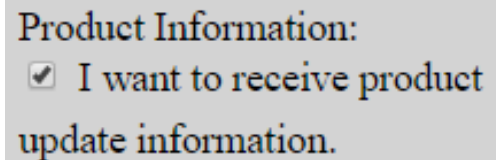
```
<p>Product Information:</p>
<input type=checkbox name='updateInfo'
 [(ngModel)]="updateInfo">
```

- The value is bound to a boolean field in the component class:

```
updateInfo: boolean;
```

- The value is also bound to the form.value:

```
{ "updateInfo": false }
```



Product Information:  
☒ I want to receive product  
update information.

### Notes:

The binding to the component property updateInfo is created with:

```
[(ngModel)]="updateInfo"
```

The binding to the form is created when you set the name property of the element:

```
name="updateInfo"
```

Complete checkbox example component:

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

@Component({
 selector: 'checkboxform',
 template: `<div class=block ><h3>Checkbox Form</h3>
<form #form=ngForm (ngSubmit)="onSubmit(form);">
<p>Product Information:</p>
<input type=checkbox name='updateInfo' [(ngModel)]="updateInfo">
Send product update information.

<input id=submit type=submit value="Submit">

</form></div>`,
})

export class CheckboxComponent {

 updateInfo: boolean = true;
```

```
onSubmit = function(form){
 var data = JSON.stringify(form.value, null, 2);
 alert(data);
 console.log(data);
}
}
```

### 6.14 Select (Drop Down) Fields

- A select field can be bound just like an input field using [(ngModel)]
- The value of the field will be added to the form.value object.
- Here is an example with the options hardcoded:

```
Product:

<select name=product [(ngModel)]="product">
 <option value="Laptop">Laptop</option>
 <option value="Tablet">Tablet</option>
 <option value="Phone">Phone</option>
 <option value="Watch">Watch</option>
 <option value="Camera">Camera</option>
</select>
```

## 6.15 Rendering Options for Select (Drop Down)

- In Angular we can render the options values using `NgFor`
- Use this code in the form's component template

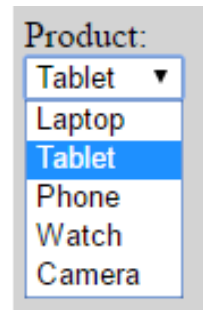
```
Product:

<select name=product
 [(ngModel)]="product" >
 <option *ngFor="let item of products"
 [ngValue]="item">{{item}}</option>
</select>
```

- ◇ Use `[value]` for the value if it is a simple type, `[ngValue]` if the value when the option is selected is an object

- Use this code in the form's component class:

```
export class FormComponent {
 products:string[] = ["Laptop", "Tablet",
 "Phone", "Watch", "Camera"];
 ...
}
```



### Notes:

`ngFor` is one of Angular's built-in directives. We use it here to render the options values for the select.

The Asterisk "\*" is used with built-in directives that use the host element as a template. In this case the `<option>` element is the host. The `ngFor` will create multiple copies of `<option>` using the current one as a template. The asterisk is also used with the `ngIf` and `ngSwitch` directives.

The keyword "let" used here is like "var" except that it restricts the scope of defined variable to the local scope whereas "var" will create variables in the global scope.

Use the `[ngValue]` directive if the value when that option is selected is an object and not just a simple type (String, number, etc).

```
<select [(ngModel)]="selectedEdition">
 <option *ngFor="let e of editions"
 [ngValue]="e">{{e.editionName}}</option>
</select>
```



## 6.16 Date fields

- Date input fields work with dates as **strings**.
- The input element includes a 2-way binding:

```
<input type=date
 [(ngModel)] = "dateStr"
 name="dateStr">
```
- Set the initial date value like this:

```
dateStr:string = new
Date().toISOString().split("T")[0];
```
- Convert the output string back to a date:

```
new Date(this.dateStr + "00:00.00");
```

**Date Form.**

  
  
Date as String: 2016-07-01  
Date as Date: Fri Jul 01  
2016 00:00:00 GMT-0400  
(Eastern Daylight Time)

## 6.17 Radio Buttons

- Input fields with type="radio" work together to update a single value.
- They are related to each other via the name attribute.
- Each radio button input field is paired with a label like this:

```
<input id=one type="radio"
value="one" name="choice"
[(ngModel)]="choice" >
<label for=one >One</label>
```

- Like other input fields radio buttons are bound to the form and to a component property using [(ngModel)] and the name attribute.
- Radio buttons must be enclosed in a <form> tag for two-way binding to work

Radio Button Form

**Choice:**

☒ One  
☐ Two  
☐ Three

**Value:** one

### Notes:

Full code for the radio button example component (radio.button.form.ts):

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

@Component({
 selector: 'radiobuttonform',
 template: `
<div class=block><h3>Radio Button Form</h3>
<form #f="ngForm">
 Choice:

 <input id=one type="radio" name="choice" [(ngModel)]="choice" value="one">
 <label for=one >One</label>

 <input id=two type="radio" name="choice" [(ngModel)]="choice" value="two">
 <label for=two >Two</label>

 <input id=three type="radio" name="choice" [(ngModel)]="choice" value="three">
 <label for=three >Three</label>

 <input type=button value=submit (click)=onSubmit(f) >

</form>Value: {{f.value.choice}}</div>`
})
```

```
export class RadioButtonFormComponent {
 choice: string = 'two';
 onSubmit(form) { alert(JSON.stringify(form.value, null, 2)); }
}
```

### 6.18 Summary

#### In this chapter we covered:

- What is a Template Driven Form
- Binding input fields to component properties
- Binding input fields to form.value
- Bindings for common input fields
- Basic Validation
- Displaying and using validation state
- Accessing data via the form object
- Submitting forms



## Chapter 7 - Reactive Forms

---

### *Objectives*

This chapter includes the following reactive forms topics:

- How do reactive forms differ from template driven form?
- FormGroup
- FormControl
- Validation
- SubForms

### 7.1 Reactive Forms Overview

- Reactive forms is an alternative API to template driven forms.
- A reactive form is setup using code. Very little work is needed in the template. This has a few advantages:
  - ◇ It is easier to unit test than template driven form.
  - ◇ It is easier to write custom validation logic.

### 7.2 The Building Blocks

- The **FormControl** class represents individual input elements in a form. Useful properties:
  - ◇ **value** - Data entered by the user in the input field.
  - ◇ **valid, invalid, touched, dirty** - If user input is valid, invalid and so on.
- The **FormGroup** class represents the entire form and contains a number of FormControl objects. Useful properties:
  - ◇ **value** - An object containing data entered by the user in the form.
  - ◇ **valid, invalid, touched, dirty** - If all the inputs in the form are valid, invalid and so on.
  - ◇ **controls** - Contains all the FormControl objects in the form
- These classes are available from the **ReactiveFormsModule** module.

## 7.3 Import ReactiveFormsModule

- To use the full features of reactive forms, use the 'ReactiveFormsModule' instead of the regular 'FormsModule'. You can use both approaches in your application depending on the situation. In which case you need to import both modules.
- Add the following in `app.module.ts` boot file:

```
import { ReactiveFormsModule } from
 '@angular/forms';
...
@NgModule({
 imports: [BrowserModule, ReactiveFormsModule],
```

## 7.4 Construct a Form

- Design the form in the component class.

```
import { FormGroup, FormControl } from '@angular/forms';
import { OnInit } from '@angular/core';
```

```
@Component({...})
export class MyComponent implements OnInit {
 myForm!: FormGroup

 ngOnInit() {
 this.myForm = new FormGroup({
 first: new FormControl,
 last: new FormControl
 })
 }
}
```

## 7.5 Design the Template

First:

Last :

- In HTML Template
  - ◇ The `<form>` element references the FormGroup with the **formGroup** directive
  - ◇ The individual input elements reference the FormControl objects with the **formControlName** directive

```
<form [formGroup]="myForm">
 First: <input formControlName="first" >

 Last : <input formControlName="last" >

 <button (click)="submitForm()">Submit</button>
</form>
```

## 7.6 Getting Input Values

- Obtain the data entered by the user using the **FormGroup.value** property. This will return an object with properties having the same names as the FormControl objects.

```
submitForm() {
 let data = this.myForm.value

 /*
 data will be like this:
 { first: "Bob", last: "Builder" }
 */
}
```

- You can also get the data entered in a specific input field.

```
let firstName = this.myForm.controls.first.value
```

```
let lastName = this.myForm.controls.last.value
```

## 7.7 Initializing the Input Fields

- To set an initial value for an input field use the first parameter of the FormControl class constructor.

```
ngOnInit() {
 this.myForm = new FormGroup({
 first: new FormControl("Bugs"),
 last: new FormControl("Bunny")
 })
}
```

First:

Last :

## 7.8 Setting Form Values

- You can set the initial value of an input using the first argument of the FormControl constructor.
- In some situations you may have to asynchronously fetch data and then set the initial value of the input fields. To do this use one of these methods:
  - ◇ **FormGroup.setValue** - Sets values of all input fields in the form in one shot.
  - ◇ **FormGroup.patchValue** - Updates the values of only some of the input fields.
  - ◇ **FormControl.setValue** - Set the value of a specific input field.

```
ngOnInit() {
 //Fetch data from server
 this.authService.getUser().subscribe((u:User) => {
 this.myForm.setValue({
```



```
 first: u.firstName,
 last: u.lastName
 })
 })
}
```

## 7.9 Subscribing to Input Changes

- This lets you do live changes to the page (like auto complete) as user enters input.

```
export class MyComponent implements OnInit, OnDestroy {
 myForm:FormGroup
 valueSubscription:Subscription

 ngOnInit() {
 this.myForm = new FormGroup({...})
 this.valueSubscription = this.myForm.valueChanges
 .subscribe(v => console.log(v))
 }

 ngOnDestroy() {
 this.valueSubscription.unsubscribe()
 }
}
```

### Notes

The valueChanges property of FormGroup is an Observable and we can subscribe to it. But this observable never completes since the user can keep entering data indefinitely. As a result we must unsubscribe to the subscription from ngOnDestroy. Otherwise we will have a memory leak.

You can also subscribe to changes to a specific FormControl.

```
this.valueSubscription = this.myForm.controls.first.valueChanges.subscribe(v => {
 console.log(v)
})
```

## 7.10 Validation

- One of the biggest differences between template driven forms and reactive forms is where validation is declared
  - ◇ Template driven forms have validation properties defined in the HTML template
  - ◇ Reactive forms define validation by customizing the construction of FormControl objects in the component
- When FormControl objects are created, there is an optional, second parameter to the constructor
  - ◇ We can pass a single validator or an array of validators as the second parameter

```
fieldname: new FormControl (default_value,
 validator | validator[])
```

- Example validators array:

```
[Validators.minLength(5), Validators.required]
```

## 7.11 Built-In Validators

- Angular has a few built-in validators:

```
Validators.required
Validators.email
Validators.minLength(minLen: number)
Validators.maxLength(maxLen: number)
Validators.pattern(regex-pattern: string)
```

- Usage examples:

```
name: new FormControl('', Validators.required),
street: new FormControl('', Validators.minLength(3)),
city: new FormControl('', Validators.maxLength(10)),
zip: new FormControl('', Validators.pattern('[A-Za-z]{5}'))
```

## 7.12 Showing Validation Error

```
<form [formGroup]="myForm">
```

```
<input formControlName="first"/>
Please enter
a valid first name

<input formControlName="last"/>
Please enter a
valid last name

<button [disabled]="myForm.invalid"
(click)="submitForm()">Submit</button>
</form>
```

- You can also use the `ng-invalid`, `ng-dirty` etc classes to visually indicate errors.

## 7.13 Custom Validator

- Custom validators are functions that:
  - ◇ Take an `AbstractControl` as input
  - ◇ Return null when the control is valid
  - ◇ Return an object with validation info when control is invalid
- In the following example we write a custom validator that makes sure that an entered email address ends with a valid domain name. File: `email.validator.ts`:

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

export function validateEmailDomain(control:
AbstractControl) {
 if (typeof (control.value) === 'string') {
 if (control.value.endsWith("@example.com")) {
 //Valid
 return null;
 }
 }
 //Invalid. Return any object.
 return { validateEmail: { valid: false } }
```

```
}
```

## 7.14 Using a Custom Validator

- Add the validator function to the array of validators for a FormControl.

```
import { validateEmailDomain } from './email.validator';

export class MyComponent {
 myForm: FormGroup;

 ngOnInit() {
 this.myForm = new FormGroup({
 fullName: new FormControl,
 emailAddress: new FormControl("", [
 Validators.required,
 Validators.email,
 validateEmailDomain])
 })
 }
}

<form [formGroup]="myForm">
 Full name: <input formControlName="fullName">

 Email: <input formControlName="emailAddress">

 <button [disabled]="myForm.invalid">Submit</button>
</form>
```

## 7.15 Supplying Configuration to Custom Validator

- If a validator needs configuration then create a factory function that returns the custom validator and captures the configuration in its closure.
- In the example below we supply a list of valid domain names.

```
export function validateEmailDomain(domains: string[]) {
 return (control: AbstractControl) => {
 if (typeof (control.value) === 'string') {
```

```
 if (domains.find(
 d => control.value.endsWith(`@${d}`))) {
 return null; //Valid
 }
 }
 //Invalid
 return { validateEmail: { valid: false } }
}

//Use the validator
this.myForm = new FormGroup({...
 emailAddress: new FormControl("", [
 Validators.required, Validators.email,
 validateEmailDomain(["example.com", "xyz.com"])]
 })
})
```

## 7.16 FormArray - Dynamically Add Inputs

- In some cases you need to dynamically add or remove inputs to a form. For example, in the shopping cart page you need a quantity text box for each item in the cart. This can be achieved using a FormArray.
- As an example we will build this form where user can enter multiple hobbies.

Your name:  
  
Your hobbies.

## 7.17 FormArray - The Component Class

```
export class AppComponent implements OnInit {
 myForm!: FormGroup
 hobbyInputs!: FormArray

 ngOnInit(): void {
 this.hobbyInputs = new FormArray(
```

```
 [], [Validators.required])

 this.myForm = new FormGroup({
 fullName: new FormControl,
 hobbies: this.hobbyInputs
 })
 }

 addHobbyInput() {
 this.hobbyInputs.push(new FormControl)
 }
}
```

## 7.18 FormArray - The Template

```
<form [formGroup]="myForm" (ngSubmit)="submitForm()">
 Your name:

 <input formControlName="fullName">

 Your hobbies. <button type="button"
(click)="addHobbyInput()">Add</button>
 <div formArrayName="hobbies">
 <input type="text"
 *ngFor="let h of hobbyInputs.controls; let i =
index;"
 [formControlName]="i"/>
 </div>

 <button type="submit">Submit</button>
</form>
```

## 7.19 FormArray - Values

- All the input entered in a FormArray is returned as an array. In our example the FormGroup.value property will be like this.

```
{
 fullName: "Bugs Bunny",
```

```
hobbies: [
 "Golf", "Soccer"
],
}
```

## 7.20 Sub FormGroups - Component Class

- Component Class Code:

```
export class FormComponent implements OnInit {

 myForm: FormGroup;
 ngOnInit() {
 this.myForm = new FormGroup({
 first: new FormControl('Jim', []),
 last: new FormControl('Doe', []),
 address: new FormGroup({
 address: new FormControl('Main Street', []),
 city: new FormControl('Newark', []),
 state: new FormControl('NJ', []),
 zip: new FormControl('07102', []),
 })
 });
 }
 onSubmit() {
 console.log(JSON.stringify(
 this.myForm.value, null, 2));
 }
}
```

## 7.21 Sub FormGroups - HTML Template

- In HTML Template

```
<form [formGroup]="myForm">
 First: <input formControlName="first" >

 Last : <input formControlName="last" >

 <div formGroupName="address">
 Address: <input formControlName="address" >

 City: <input formControlName="city" size=12 >
```

```
State: <input formControlName="state" size=2 >
Zip: <input formControlName="zip" size=5 >
</div>
<input type=button value=submit (click)=onSubmit()>
</form>
```

- In the above HTML template we have two fields (first, last) at the root level of the main form group "myForm"
- Then we have four fields (address, city, state, zip) in the sub Form Group Address.
  - ◇ Note the sub FormGroup is referenced by the 'formGroupName' property as 'formGroup' can only reference something declared as a component property and not the nested FormGroup 'address'

## 7.22 Why Use Sub FormGroups

- It formats the form data for us:

```
/* this.myForm.value */
{
 "first": "Jim",
 "last": "Doe",
 "address": {
 "address": "Main Street",
 "city": "Newark",
 "state": "NJ",
 "zip": "07102"
 }
}
```

- Validation can be applied separately to different sub groups.

## 7.23 Summary

In this chapter we covered the following Reactive Forms topics:

- Setup
- FormGroup initialization
- FormControl object



- Validation
- SubForms



## Chapter 8 - Services and Dependency Injection

---

### *Objectives*

Key objectives of this chapter

- What is a Service?
- Creating Services
- Dependency Injection
- Using Services in Components
- Using Shared Service Instances
- The @Optional DI decorator
- The @Host DI decorator

### 8.1 What is a Service?

#### Services in Angular:

- Are used to perform pure business logic without any concern for display rendering:
  - ◇ Data retrieval from back end web services
  - ◇ Data validation
  - ◇ Logging
  - ◇ etc.
- Are implemented using simple classes decorated with **@Injectable**.
- Are used by Angular components.
- Services are easier to unit test and reusable in more situations than components. Always attempt to move pure business logic into services and away from components.

### 8.2 Creating a Basic Service

- To create a service class called PetService run this CLI command in project root.

ng generate service pet

- This will create the following files:
  - ◇ src/app/pet.service.ts – Contains the PetService class.
  - ◇ src/app/pet.service.spec.ts – Contains boilerplate code for unit testing the service.

## 8.3 The Service Class

### To create a basic service:

- Create a class and export it
- Import Injectable
- Add @Injectable annotation

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

@Injectable({
 providedIn: 'root'
})
export class PetService{
 pets: string[] = ["Cat", "Dog",
 "Hamster", "Rabbit", "Fish",
 "Bird", "Turtle"];
 getPets(){ return this.pets;}
}
```

### Notes

The code above is saved in a file named "pet.service.ts"

In order to use the @Injectable annotation we first need to import Injectable from Angular.

Note: @Injectable is sometimes referred to as a "Decorator". You can refer to what we are doing here as either "annotating" the class or "decorating" the class.

This class exposes a getPets() method that returns an array of pets.

Defining the getPets() method in a service allows it to be reused in multiple Angular components.

While the getPets() method here implements synchronous behavior many real world services will need to function asynchronously. Asynchronous behavior can be implemented using "Promise" objects or publish-subscribe methods. One example of using a service asynchronously appears in the chapter on the HTTP service.

## 8.4 What is Dependency Injection?

- Services in Angular are supplied to components through something called "dependency injection"
- Using dependency injection allows developers to:
  - ◇ Use classes without hard-coding constructor calls.
  - ◇ Avoid tight coupling of classes. For example an alternate implementation of a service can be plugged in without modifying the application code.
  - ◇ Effortlessly implement the singleton pattern.
  - ◇ Easily substitute mocked services for testing

### Notes:

Dependency injection (DI) is used in Angular to supply service instances to components.

Before learning how to use services we need to take a detour and talk about dependency injection.

The alternative to using DI is to have components hard-code calls to constructors which can cause problems. It tightly couples the component to the classes it uses and requires component to know how to create instances. With DI creating instances is done by an Injector. The Injector knows how to create instances and keeps track of the instances it creates so that it can reuse instances if necessary instead of creating new ones.

When components and services are connected through DI it's easy to replace services with new implementations without requiring any modification of the component. It is also easy to supply alternate implementations of a service for use in testing.

## 8.5 Injecting a Service Instance

- Service class instances are injected through constructor parameters
- You will use code like this:

```
export class MyComponent{
 //Injection happens here
 constructor(private petSvc: PetService) {}
}
```

- Do not write code like this:

```
export class MyComponent{
```

```
 petSvc: PetService = new PetService();
 }
```

- A service, pipe and directive can also inject other services the same way.

### Notes:

The constructor parameter above tells Angular to:

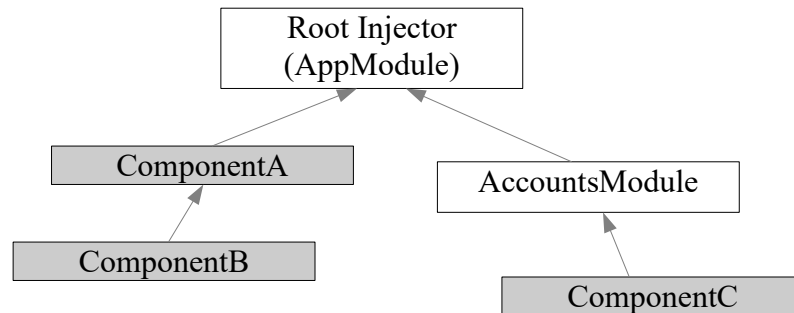
- Create an instance of PetService
- Pass the instance into the constructor with the name "petSvc"

The "private" keyword is a shortcut that tells angular to add "petSvc" as a property of MyComponent. Alternatively you could use "public" also.

## 8.6 Injectors

- Injectors are responsible for creating instances of a service and injecting them in constructors.
- There are many injectors available in an application. They are arranged in a tree like structure.
  - ◇ There is a root injector that is at the very top of the hierarchy.
  - ◇ Every feature module has its own injector.
  - ◇ Every component has its own injector.
- A service class needs to be registered with an injector for it to be able to create an instance.

## 8.7 Injector Hierarchy



- At an injection point the nearest injector is asked to supply the service instance.
- If that injector does not know about the service it will ask its parent injector and so on. If no injector up the hierarchy knows about the class then an exception is thrown.

### Notes

In the diagram above we see a hierarchy of injectors. It very closely follows the hierarchy of modules and components within them. Every component has its own injector. So does every module.

As an example, if ComponentB tries to inject a service its own injector will be asked to do so first. If that injector doesn't know how to create an instance of the service it will ask ComponentA's injector. ComponentA's injector may then ask the root injector.

## 8.8 Registering a Service with the Root Injector

- By default a service is registered with the root injector.

```
@Injectable({
 providedIn: 'root'
})
export class PetService {
 constructor() { }
}
```

- This makes the service available throughout the application from any component, directive and service in any module.
- This setup is good enough in most cases. Although, you can also register a service:
  - ◇ At a component level.
  - ◇ At a feature module level. This is discussed in a separate chapter.

## 8.9 Registering a Service with a Component's Injector

- Add the service to the providers list of `@Component`.

```
@Component ({
 ...
 providers: [BillingService],
})
export class ComponentA {
 constructor(private billSvc: BillingService) {}
}
```

- This approach is needed if you wish the component to have its own private instance of a service.
- All children component will share this instance (unless you register the service with them as well).

## 8.10 Register a Service with a Feature Module Injector

- Use the `providedIn` property of the `@Injectable` decorator.

```
@Injectable({
 providedIn: AccountsModule,
})
export class BillingService {...}
```

- Before you can inject such a service from another module you must import the provider feature module.



## 8.11 Where to Register a Service?

- In most cases it is sufficient to register a service at the root injector. It has a few basic advantages:
  - ◇ The service becomes usable throughout the application.
  - ◇ A single instance of the service is injected throughout by the root injector thus implementing the singleton pattern.
- If your application defines feature sub-modules and you wish to make a service part of such a feature module then register the service at the module level. If this module is imported by an application then effectively the service will get registered at the root level otherwise the service code will be excluded from the build output.
- If a component does not wish to use a global singleton instance of a service and wishes to have its own private copy then register the service at a component level. For example, if a service caches user input data then a component probably wants to have a private cache and this approach will then becomes useful.

## 8.12 Dependency Injection in Other Artifacts

- You can also inject services into directives, pipes and other services. Injection is done in the constructor exactly the same way as for components.

```
@Injectable({providedIn: 'root'})
export class UserService {
 constructor(private svc:AccountService) {}
}
```

- These artifacts are eventually brought into a running application by a component. For example, component A uses component B which injects service S1 which injects service S2.
  - ◇ When these artifacts inject a service the injector of the nearest component is used. In the example above when service S1 injects service S2 the injector of component B will be used.

## 8.13 Providing an Alternate Implementation

- When registering a service with an injector you can supply an alternate implementation class for the service.

<pre>@Injectable({   providedIn: 'root' }) export class HelloService {   greet() {     return "Hello"   } }</pre>	<pre>@Injectable({   providedIn: 'root' }) export class GreetService {   greet() {     return "Greeting"   } }</pre>
-------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------

```
@Component({...
 providers: [
{provide: HelloService, useClass: GreetService}
])
export class MyComponent {...}
```

- Make sure that the alternate implementation class has the same method signature. One way to ensure that is to extend both classes from the same abstract class (DI does not work with interfaces). See full example in the notes.

### Notes

To make sure that all implementations of a service conform to the same method signatures we need to write an abstract class and extend the implementations from that class.

```
export abstract class AbstractHello {
 abstract greet() : string
}

@Injectable({
 providedIn: 'root'
})
export class HelloService extends AbstractHello {

 constructor() { super() }

 greet() {
 return "Hello"
 }
}
```

```
@Injectable({
 providedIn: 'root'
})
export class GreetService extends AbstractHello {

 constructor() { super() }

 greet() {
 return "Greeting"
 }
}
```

When registering the service use the abstract class.

```
@Component({
 ...
 providers: [
 {provide: AbstractHello, useClass: GreetService}
]
})
```

When injecting the service also use the abstract class.

```
@Directive({...})
export class MyDirective {
 constructor(private svc:AbstractHello) {...}
}
```

### 8.14 Dependency Injection and @Host

You may want to limit how far up the Injector hierarchy Angular searches for a dependency.

```
constructor(private petSrv: PetService){{}
```

- The above constructor requests an instance of the `PetService`:
- Taken as written any Injector in the hierarchy can fulfill the request.
- When using "`@Host()`" the search for an Injector that knows about the service stops with the host's(parent's) injector.

```
constructor(@Host() private petSrv: PetService){{}
```

- As written above if neither the current component nor its host (parent) has access to the service then an error is thrown.
- This safeguards the component from using an instance of the service that was intended for a different component higher up in the hierarchy.

### Notes

Note: "Host" needs to be imported for `@Host()` to work:

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

If an error is returned when using `@Host()` the proper response is to make sure the service is available to the host (parent) by adding the service's name to the host's providers array.

```
providers:[PetService],
```

### 8.15 Dependency Injection and `@Optional`

You may want to avoid the exception thrown by Angular when a requested dependency is not found.

```
constructor(private logger: LogService){}
```

- The above constructor requests an instance of the `LogService`:
  - ◇ If one of the Injectors in the hierarchy finds `LogService` then an instance is returned
  - ◇ If it can't be found an error is thrown.

- If needed you can mark dependencies with "`@Optional()`":

```
constructor(@Optional() private logger: LogService){}
```

- When `@Optional` dependencies are not found:
  - ◇ No error is thrown
  - ◇ The Injector returns a null value
  - ◇ The developer can detect the null value and branch to alternate code.

### Notes

Note: "Optional" needs to be imported for `@Optional()` to work:

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

When using `@Optional()` you can branch to alternate code if the returned instance is null:

```
export class MyComponent implements OnInit {
 constructor(@Optional() private logger: LogService){}
 ngOnInit() {
 if(this.logger){
 this.logger.log("In MyComponent.init() - using logger service.");
 }else{
 console.log("in MyComponent.init() - using console.log.");
 }
 }
}
```

```
 }
 }
```

`@Host()` and `@Optional()` can also be used together. In this case, a null instance is returned (instead of an error) if the service is not found by the host's Injector. When doing this make sure to import both "Host" and "Optional".

```
import { Component, OnInit, Host, Optional } from '@angular/core';
```

### 8.16 Summary

In this chapter we covered:

- What a Service is,
- Creation of Services,
- Dependency Injection,
- Using Services in Components,
- Using Shared Service Instances,
- The `@Optional` DI decorator,
- The `@Host` DI decorator.



## Chapter 9 - HTTP Client

---

### *Objectives*

Key objectives of this chapter

- What is the Angular HTTP Client
- Importing HttpClientModule
- Making Get/ Post Calls
- Working with Observables

### 9.1 The Angular HTTP Client

The Angular HTTP Client:

- Provides a simplified API for network communication. It is a wrapper over the JavaScript XMLHttpRequest API.
  - ◇ The API is asynchronous. JavaScript is single threaded. Doing a blocking synchronous HTTP call will otherwise freeze the UI.
- Supports:
  - ◇ Making HTTP requests (GET, POST, etc.)
  - ◇ Working with request and response headers
  - ◇ Asynchronous programming
- Makes use of the rxjs async library Observable object

### 9.2 Using The HTTP Client - Overview

- The core client API is available from the **HttpClient** Angular service class. This is available from the **HttpClientModule** Angular module.
  - ◇ Import HttpClientModule from your application module.
- A Data Service is created that makes network requests:
  - ◇ Inject the HttpClient service instance.
  - ◇ Use various methods of HttpClient to make network calls. They return an Observable object. Usually, this Observable is returned from the

service method.

- An Angular Component is used to display the response data:
  - ◇ The data service is injected into the constructor
  - ◇ The component calls a method of the data service and obtains an Observable object.
  - ◇ The component then "subscribes" to the Observable to receive the response from the HTTP call.
  - ◇ The component's template displays the data

### Notes:

The initial use-case for network requests involves the retrieval of data from a server and the display of that data in a view. In this case a GET request is used.

## 9.3 Importing HttpClientModule

- In order to use the Http client throughout the application, we need to import `HttpClientModule` in the application module
  - ◇ `HttpClientModule` is a collection of service providers from the Angular HTTP library

```
import {HttpClientModule} from '@angular/common/http';

@NgModule({
 imports: [BrowserModule, HttpClientModule],
 ...})
```

- Now the **HttpClient** service is injectable throughout your application

### Importing HttpClientModule

Note that `HttpClientModule` configures `HttpClient` as a provider. You don't have to do this again from your application module. As a result, `HttpClient` is now injectable anywhere in your application.



## 9.4 Service Using HttpClient

- Sample Service Code ("people.service.ts"):

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

export class Person {id:number, name:string ...}

@Injectable(...)
export class PeopleService{
 constructor(private http: HttpClient) {}

 getPerson(id:number) : Observable<Person> {
 return this.http.get<Person>(`/app/person/${id}`)
 }
 getAllPersons() : Observable<Person[]> {
 return http.get<Person[]>(`/app/person`)
 }
}
```

- This code is reviewed in the next few slides

### Notes:

We review various lines of the code above in the next few slides including:

- Import statements
- Injection of the HttpClient object
- Making a GET call and returning a response.

## 9.5 Making a GET Request

- Call the get() method of HttpClient service.
  - ◇ All network call methods like get(), post(), put() return an Observable.
- There are many overloaded versions of the get() call. Some are:

```
let obs:Observable<Object> = http.get(url)
//Strong typing
let obs:Observable<Person> = http.get<Person>(url)
```

- For the `get()` calls it is recommended you use the strongly typed version (second one from above).
- The `get()` method returns an `Observable` immediately. One needs to subscribe to the `Observable` to obtain the response data asynchronously.

### Notes:

As they are used with the Angular `HttpClient` object `Observable` objects can be thought of as the *glue* that connects network requests with data consumers. There are several aspects to this relationship that our discussion will cover one at a time. From the current slide, we see that `http` requests return `Observable` objects that we can assign to a variable for later use. In the next slide, we will take a look at how `Observable` objects are used.

## 9.6 What does an Observable Object do?

- Angular uses the Reactive Extensions for JavaScript (RxJS) implementation of the `Observable` object.
- `Observable` objects provide a convenient way for applications to consume asynchronous data/event streams.
- `Observable` objects are exposed to events which they then make available to consumers.
- Applications consume events by subscribing to the `Observable` object.
- The `Observable` object can be used to transform data before returning it to a consumer if needed.

### Notes:

Reactive programming and `Observable` objects can be used anywhere an asynchronous programming model is required. For more information on these topics see:

<http://reactivex.io/intro.html>

## 9.7 Using the Service in a Component

- Once the service has been created we can create an Angular component that uses it to retrieve and display data.
- Our component will have to import the service.
- In order to retrieve data, the code in our component will have to work with the Observable object that was created in the service.
- The screen shot at right shows how the data will look when displayed.

### PeopleList

Raymond,Ward,rward0@oracle.com  
Daniel,Chavez,dchavez1@dedecms.com  
Sharon,Dunn,sdunn2@so-net.ne.jp  
Jonathan,Kennedy,jkennedy3@google.es  
Joe,Harrison,jharrison4@wiley.com

## 9.8 The PeopleService Client Component

```
import { Component, OnInit } from '@angular/core';
import { PeopleService, Person } from '../people.service';

@Component({
 selector: 'app-people',
 template: `
 <p *ngFor='let person of list' >{{person.name}}</p>`
})
export class PeopleComponent implements OnInit {
 list: Person[]
 constructor(private peopleService: PeopleService){}
 ngOnInit() {
 this.peopleService.getAllPersons().subscribe(
 (data: Person[]) => this.list = data
)
 }
}
```

## 9.9 Error Handling

- Two types of errors can happen during a network call:
  - ◇ No network connection can be made to the server.
  - ◇ The server returns an invalid response code in 4XX and 5XX range.
- A subscriber can handle these errors by supplying an error handler function to `subscribe()`.

```
ngOnInit() {
 this.peopleService.getAllPersons().subscribe(
 (data: Person[]) => this.list = data,
 (error: HttpResponse) => console.log(error)
)
}
```

## 9.10 Customizing the Error Object

- A service may decide to offer more meaningful errors in a custom error object instead of the default `HttpResponse`.
- You can intercept an error with the **`catchError()`** operator and supply a custom error object using **`throwError()`**.
- The following transforms the error object from `HttpResponse` to a string.

```
import {Observable, throwError} from 'rxjs'
import {catchError} from 'rxjs/operators'

export class PeopleService {
 constructor(httpClient: HttpClient){}

 getAllPersons() : Observable<Person[]> {
 return httpClient
 .get<Person[]>(`/app/person`)
 .pipe(
 catchError(error =>
```

```
 throwError("There was a problem with the network"))
)
}

//The component
ngOnInit() {
 this.peopleService.getAllPersons().subscribe(
 (data: Person[]) => this.list = data,
 (errMsg:string) => alert(errMsg))
}
```

## 9.11 Making a POST Request

- Supply the request body as the second argument to the post() method.

```
createPerson(person:Person) : Observable {
 return this.http.post("/app/person", person)
}
```

- If a POST request returns data in the response body, you can make a more type safe call.

```
createPerson(person:Person) : Observable<AddPersonResult> {
 return this.http.post<AddPersonResult>("/app/person",
 person)
}

//The component
this.peopleService.createPerson(...)
 .subscribe((result: AddPersonResult) => {...})
```

## 9.12 Making a PUT Request

- Supply the request body as the second argument to the put() method.

```
updatePerson(person:Person) : Observable {
 return this.http.put("/app/person", person)
```

```
}
```

- If a PUT request returns data in the response body, you can make a more type safe call.

```
updatePerson(person:Person) : Observable<UpdateResult> {
 return this.http.put<UpdateResult>("/app/person",
 person)
}
//The component
this.peopleService.updatePerson(...)
 .subscribe((result: UpdateResult) => {...})
```

### 9.13 Making a DELETE Request

- A DELETE request does not take any body

```
deletePerson(id:number) : Observable {
 return this.http.delete(`/app/person/${id}`)
}
```

- If a DELETE request returns data in the response body, you can make a more type safe call.

```
deletePerson(id:number) : Observable<DeleteStatus> {
 return this.http.delete<DeleteStatus>(`/app/person/${id}`)
}
//The component
this.peopleService.deletePerson(12)
 .subscribe((result: DeleteStatus) => {...})
```

### 9.14 Summary

In this chapter we covered:

- What is the Angular HTTP Client

- Importing HttpClientModule
- Making Get/ Post Calls
- Working with Observables





## Chapter 10 - Pipes and Data Formatting

---

### *Objectives*

Key objectives of this chapter

- What are Pipes?
- Angular Built-in Pipes
- Using pipes in HTML.
- Internationalized Pipes
- Using pipes in TypeScript.
- Creating Custom Pipes
- Pure and Impure Pipes

### 10.1 What are Pipes?

- Pipes are used by components to format data in templates. For example, format number, currency and date.

Salary: \$1,023.10

```
<p>Salary: {{salary | currency}}</p>
```

```
export class RegisterComponent {
 salary = 1023.10293
}
```

### 10.2 Built-In Pipes

- Various Pipes are Built in to Angular:

Class	Name
UpperCasePipe	uppercase
LowerCasePipe	lowercase

DecimalPipe	number
DatePipe	date
CurrencyPipe	currency

- Built-in pipes are imported by default.

**Notes:**

A full list of built-in pipes is available here:

<https://angular.io/docs/ts/latest/api/#!?apiFilter=pipe>

Built-in pipes are imported by default.

Custom pipes need to be imported before they can be used.

### 10.3 Using Pipes in HTML Template

- The syntax for using a pipe to output data in HTML:

```
{{ data-expression | pipe-name:param1:param2] }}
```

- Example:

```
birthday: Date = new Date(1993, 9, 31);
```

```
{{birthday|date:"M-dd-yyyy"}}
```

```
outputs: "10-31-1993"
```

- ◇ Here the pipe-name is: "date"
- ◇ The pipe-parameter is: "M-dd-yyyy"

### 10.4 Chaining Pipes

- Chained pipes are executed from left to right:

```
{{ birthday | date | uppercase }}
```

- The effect of the chained pipes is:

- ◇ The `birthday` property is first converted to a date string
- ◇ The date string is then converted to uppercase.

- **Example-Data:**

```
birthday: Date = new Date(1993, 9, 31);
```

- **Example-Pipes:**

HTML	Output
{{birthday date:'MMMM dd,yyyy'}}	October 31, 1993
{{birthday date:'MMMM dd,yyyy' uppercase}}	OCTOBER 31, 1993

## 10.5 Internationalized Pipes (i18n)

- Pipes like date, currency and number are sensitive to the locale. You can set the locale in one of two ways.
- You can set the locale for the whole application. Below we set the locale to `fr-FR`:

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

@NgModule({
 providers: [{provide: LOCALE_ID, useValue: 'fr-FR'}],
 ...
})
export class AppModule { }
```

- Alternatively, you can supply the locale in the template

```
{{ dateValue | date:'full':'':'fr-CA' }}
```

## 10.6 Loading Locale Data

- By default Angular is only able to work with the 'en-US' locale.
- To add support for additional locales you will need to load their data.

- To load locale data, add this to **app.module.ts**.

```
import '@angular/common/locales/global/fr';
import '@angular/common/locales/global/fr-CA';
import '@angular/common/locales/global/es';
```

- Register additional locales as follows:

```
import localeFrCa from '@angular/common/locales/fr-CA'
// ...
registerLocaleData(localeFrCa)
```

## 10.7 The date Pipe

- Formats a JavaScript Date object. Syntax:

```
{{dateObj | date:format:timezone:locale }}
```

- Where:
  - ◇ format – The date format string, such as "M/d/yy" or "shortDate", "fullDate" etc. Please see documentation for full details.
  - ◇ timezone – The date and time will be converted to this timezone. Example, "GMT", "EDT" etc.
  - ◇ locale – The date will be formatted for this locale. Such as "fr-FR".
- Example below shows date in Chicago timezone.

```
<p>{{today | date:"shortDate":"CDT"}}</p>
```

## 10.8 The number Pipe

- Takes a number as input
- Optionally supply these parameters:
  - ◇ A string parameter defining the number format:  
"{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}"
  - ◇ The locale

**■ Example:**

- ◇ **Usage:** `{{ 65.243 | number: '3.2-4' }}`
- ◇ **Output:** `065.243`
- ◇ **Usage:** `{{ 123.456 | number: '': 'fr' }}`
- ◇ **Outputs:** `123,456`

**■ Note:** numbers are rounded not truncated**Notes:**

Additional Examples:

Binding with Pipe	Displayed Value
<code>{{ 25   number: "1.0-2" }}</code>	25
<code>{{ 25   number: "1.2-4" }}</code>	25.00
<code>{{ 25   number: "2.2-4" }}</code>	25.00
<code>{{ 3.14   number: "1.2-4" }}</code>	3.14
<code>{{ 3.14   number: "1.4-4" }}</code>	3.1400
<code>{{ 3.141592   number: "1.2-4" }}</code>	3.1416

Full Example code (decimal.pipe.component.ts):

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

@Component({
 selector: 'decpipe',
 template: `
 <h3>DecimalPipe</h3>
 {{ 25 | number: "1.0-2" }}

 {{ 25 | number: "1.2-4" }}

 {{ 25 | number: "2.2-4" }}

 {{ 3.14 | number: "1.2-4" }}

 {{ 3.14 | number: "1.4-4" }}

 {{ 3.141592 | number: "1.2-4" }}

 {{ 65.243 | number: "3.2-4" }}

 {{ piFmt }}

 `,
 providers: [DecimalPipe],
})
```

```
})
export class DecimalPipeComponent {
 constructor(private decimalPipe: DecimalPipe){}
 bignum: number = 82364.23947;
 pi: number = 3.141592;
 piFmt = this.decimalPipe.transform(this.pi, "2.2-2");
}
```

## 10.9 Currency Pipe

- Takes a number as input
- Uses these parameters:
  - ◇ The ISO 4217 currencyCode as string: e.g. 'USD', 'CAD', 'EUR',...
  - ◇ How the currency should be shown
    - [ 'code' | 'symbol' | 'symbol-narrow' ]
  - ◇ And a string parameter defining the number format:  
"{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}"
  - ◇ Locale
- Example:
  - ◇ Usage: {{65.243 | currency: 'USD': 'symbol': "2.2-2"}}
  - ◇ Output: \$65.24
  - ◇ Usage: {{1023.10293 | currency:'EUR':":":'fr'}}
  - ◇ Output: 1 023,10 €

### Notes:

Additional Examples:

Binding with Pipe	Displayed Value
{{ 25   currency: "USD": 'code': "1.0-2" }}	USD25
{{ 25   currency: "CAD": 'symbol': "1.2-2" }}	CA\$25.00
{{ 25   currency: "USD": 'symbol': "2.2-2" }}	\$25.00
{{ salary   currency: "CAD" }}	CAD65,000.00

{{ price   currency: "EUR": 'symbol': "1.2-2"}}	€9.95
{{ price   currency: "USD": 'symbol': "1.2-2"}}	\$9.95
{{65.243  currency: "USD": 'symbol': "3.2-2"}}	\$065.24

Full example (currency.pipe.component.ts):

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

@Component({
 selector: 'currpipe',
 template: `
 <h3>CurrencyPipe</h3>
 {{ 25 | currency: "USD": 'code': "1.0-2" }}

 {{ 25 | currency: "CAD": 'symbol': "1.2-2" }}

 {{ 25 | currency: "USD": 'symbol': "2.2-2" }}

 {{ salary | currency: "CAD" }}

 {{ price | currency: "EUR": 'symbol': "1.2-2" }}

 {{ price | currency: "USD": 'symbol': "1.2-2" }}

 {{65.243| currency: "USD": 'symbol': "3.2-2" }}

 {{ salaryFmt }}

 `,
 providers: [CurrencyPipe],
})

export class CurrencyPipeComponent {
 constructor(private currPipe: CurrencyPipe){}
 price: number = 9.95;
 salary: number = 65000;
 salaryFmt: string = this.currPipe.transform(this.salary, "USD", 'code', "2.2-2");
}
```

## 10.10 Create a Custom Pipe

- Run this CLI command to create a pipe called "truncate":

```
ng generate pipe truncate
```

- This will:
  - ◇ Create **src/app/truncate.pipe.ts**. It has the TruncatePipe class.
  - ◇ Create **src/app/truncate.pipe.spec.ts**. It has unit test code.
  - ◇ Add the pipe class to the declarations list of the AppModule.

## 10.11 Custom Pipe Example

- The code below implements a custom Pipe that truncates string data based on a length parameter:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'truncate' })
export class TruncatePipe implements PipeTransform {
 transform(value: string, length: number) {
 return value.substring(0, length);
 }
}
```

- The Pipe's class name is `TruncatePipe`
- The name used to invoke the pipe is `"truncate"`
- The `transform` method uses the JavaScript `substring` method to limit the length of the original data.

### Notes:

The code above is the complete code for the custom pipe with the filename `"character.length.pipe.ts"`

## 10.12 Using Custom Pipes

- Import the Pipe class into the application module and add to the module declarations (in `app.module.ts`)

```
import { TruncatePipe } from './truncate.pipe';
```

```
@NgModule({
 imports: [...],
 declarations: [..., TruncatePipe],
 ■ Use the Pipe name in a template
```

```
{{ "Hello World" | truncate:5 }}
```

```
//Outputs: Hello
```



## 10.13 Using a Pipe with ngFor

- You can sort or filter items in a collection before it is used by ngFor.
- Example:

```
<li *ngFor="let p of products |
 textsearch:searchTerm | sortproduct:'price'">

```

## 10.14 A Filter Pipe

- Example of a Pipe that **filters** out array items that don't contain a given search term:

```
import { Pipe, PipeTransform } from "@angular/core";

@Pipe({ name: "arrayfilter" })
export class ArrayFilterPipe implements
PipeTransform{
 transform(array_instance, filter_term){
 return array_instance.filter(function(item) {
 return item.includes(filter_term);
 });
 }
}
```

- The Pipe:
  - ◇ Includes a single `transform()` method
  - ◇ Uses the JavaScript array `filter()` method
  - ◇ Uses the JavaScript string `includes()` method

## Notes

Full example of a component using the custom **filter** pipe (`array.filter.component.ts`):

```
import { Component } from '@angular/core';
import { ArrayFilterPipe } from '../array.filter.pipe';

@Component({
 selector: 'arrayfilter',
```

```
providers: [ArrayFilterPipe],
template: `


```

Note the following:

The filter is imported:

```
import { ArrayFilterPipe } from './array.filter.pipe';
```

The filter is added to the providers array so it can be injected into the constructor:

```
providers: [ArrayFilterPipe],
```

Here is where the filter is invoked in HTML:

```
<li *ngFor="let x of cars|arrayfilter:searchTerm">
```

Here is where the filter is invoked in TypeScript:

```
petsFiltered: string[] = filterPipe.transform(this.pets, this.searchTerm);
}
```

### 10.15 Pipe Category: Pure and Impure

- There are two pipe categories: pure and impure.
- The pipe category affects how deeply into an array or object Angular

watches for changes to a pipe's input value:

- ◇ **pure:** Angular watches for changes to the array reference variable only. This means that changes to the data held by the array will not cause the pipe to be reapplied.
- ◇ **impure:** Angular reapplies the pipe if any data within the array changes. This requires more effort on Angular's part and can degrade performance.
- Pipes are pure by default.
- To create an impure custom pipe add the following in @Pipe:

```
pure: false
```

## 10.16 Pure Pipe Example

- Table on the left is unfiltered
- Table on right is filtered via a custom pipe: FavoriteFilterPipe
- FavoriteFilterPipe has the "pure" property set to true.
- Turtle shows in the table at the right because its 'favorite' property is true when the filter is first applied.

Pets Unfiltered		Favorite Pets	
Pet	Favorite	Pet	Favorite
Turtle	<input checked="" type="checkbox"/>	Turtle	<input checked="" type="checkbox"/>
Cat	<input type="checkbox"/>		
Rabbit	<input checked="" type="checkbox"/>		
Dog	<input type="checkbox"/>		
Bird	<input checked="" type="checkbox"/>		

- The filter pipe on the table at the right is 'pure' so it is not reapplied when the 'favorite' properties of Rabbit and Bird are changed.

## Notes

Full code for the custom filter pipe (favorite.filter.pipe.ts):

```
import { Pipe, PipeTransform } from "@angular/core";

@Pipe({
 name: "favorite",
 pure: true
})

export class FavoriteFilterPipe implements PipeTransform{
 transform(array_instance) {
```

```
 return array_instance.filter(function(item) {
 if(typeof item.favorite !== 'undefined'){
 return item.favorite;
 }else{
 // ignore filter if favorite not a property
 return true;
 }
 });
 }
}
```

Full code for the component that uses the filter ( pure.impure.component.ts ):

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

@Component({
 selector: 'pureimpure',
 template: `<div class=left>
<h3>Pure & Impure Pipes</h3>
<div class=left>
<h4>Pets Unfiltered</h4>
<table>
 <tr><th>Pet</th><th>Favorite</th></tr>
 <tr *ngFor="let x of pets">
 <td>{{x.type}}</td>
 <td><input type=checkbox [(ngModel)]="x.favorite" ></td>
 </tr>
</table>
</div>
<div class=left>
<h4>Favorite Pets</h4>
<table>
 <tr><th>Pet</th><th>Favorite</th></tr>
 <tr *ngFor="let x of pets|favorite">
 <td>{{x.type}}</td>
 <td><input type=checkbox disabled [(ngModel)]="x.favorite" ></td>
 </tr>
</table>
</div>
</div>
`,
})

export class PureImpureComponent {
 pets: Object[] = [
 {type:"Turtle", favorite: true },
 {type:"Cat", favorite: false },
 {type:"Rabbit", favorite: false },
 {type:"Dog", favorite: false },
 {type:"Bird", favorite: false },
];
}
```

## 10.17 Impure Pipe Example

- The same example as on the previous page but now the "pure" property of FavoriteFilterPipe is set to "false"
- As before Turtle shows in the table at the right because its 'favorite' property is true when the filter is first applied.
- This time though when the 'favorite' properties of Rabbit and Bird are changed the filter pipe on the table to the right is reapplied and includes the changed rows.

Pets Unfiltered		Favorite Pets	
Pet	Favorite	Pet	Favorite
Turtle	<input checked="" type="checkbox"/>	Turtle	<input checked="" type="checkbox"/>
Cat	<input type="checkbox"/>	Rabbit	<input checked="" type="checkbox"/>
Rabbit	<input checked="" type="checkbox"/>	Bird	<input checked="" type="checkbox"/>
Dog	<input type="checkbox"/>		
Bird	<input checked="" type="checkbox"/>		

### Notes:

The example on this page is exactly the same as on the previous page except for the value of the "pure" property in the @Pipe section of the custom pipe, which is now false:

```
@Pipe({
 name: "favorite",
 pure: false
})
```

## 10.18 Summary

In this chapter we covered:

- What are Pipes?
- Angular Built-in Pipes
- Using pipes in HTML.
- Internationalized Pipes
- Using pipes in TypeScript.
- Creating Custom Pipes
- Pure and Impure Pipes



## Chapter 11 - Introduction to Single Page Applications

---

### *Objectives*

Key objectives of this chapter

- What is a Single Page Application (SPA)?
- Advantages of SPA
- Downsides of SPA
- How to build SPA's using Angular

### 11.1 What is a Single Page Application (SPA)

Single Page Applications:

- Make a distinction between "pages" and "views".
- Makes the initial HTTP request, downloads an HTML document and builds the DOM. We will call it the **page**.
- As the user interacts with the application, it transitions a portion of the page to a new **view** by displaying and hiding HTML elements.
- Views use JavaScript to upload/retrieve data to/from the server.
- Use HTML5 features:
  - ◇ History interface for navigation
  - ◇ Local storage to save and reuse data.
- Are easier to implement using Angular.

### Notes

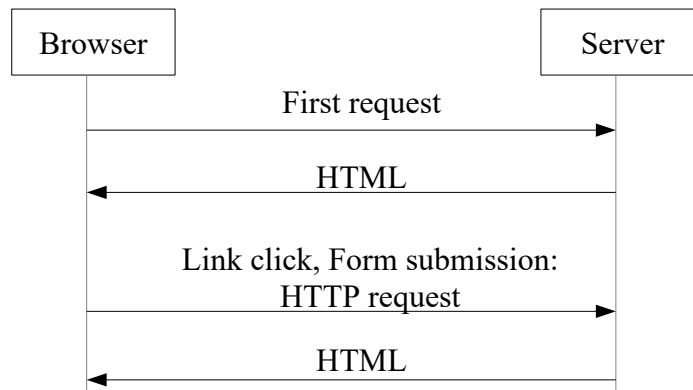
An SPA loads a single HTML document from the server and then selectively shows different DOM elements to create the illusion of multiple page navigation.

A page is a DOM document that is loaded only once. We can show multiple views using the same page.

HTML5 provides the necessary APIs, such as history and local storage, to make SPA possible.

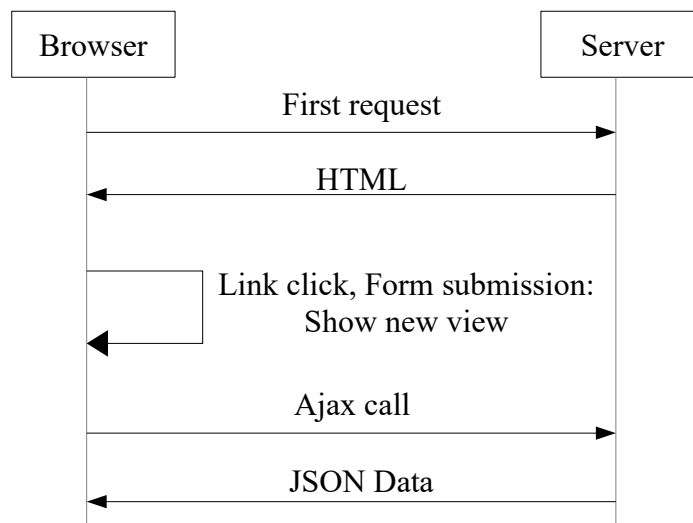
Frameworks like Angular leverage HTML5 and simplify the task of developing complex SPA.

## 11.2 Traditional Web Application



- Every link click and form submission results in a new HTML document fetched from the server and the page DOM rebuilt.
- Browser address bar reflects the current page. It can be book marked and shared.

## 11.3 SPA Workflow





## Notes

The first request is fulfilled the same way as a traditional application. The HTML document is retrieved and a DOM is built in the browser. When the user clicks on a link or submits a form something different happens. A portion of the page switches to a new view. The view may then decide to make Ajax calls and render the response data. The transition of views appear to be identical to any traditional application. Except, all of this is achieved using a single DOM document.

A small to medium size SPA will load all the necessary HTML and JavaScript one time. It will interact with the server by issuing Ajax calls.

A more advanced application can speed up initial load time by only loading the JavaScript and HTML necessary at that time. Subsequently, as new views need to be loaded, it can download additional HTML and scripts using Ajax.

## 11.4 Single Page Application Advantages

- Navigation between views in SPA's:
  - ◇ Does not require the browser to reload the HTML page. Views transition much more quickly than traditional page transitions
  - ◇ Is more fluid (screen does not flash)
- View data can be:
  - ◇ Downloaded once
  - ◇ Reused in multiple views

## Notes

View transition and overall performance are enhanced in SPA's. The biggest reason behind SPA is rapid page transition. SPA lets the user navigate through the application views more rapidly than possible if the application was reloading every page from the server. The transitions also appear visually more fluid. Overall, this creates higher user satisfaction level.

The need for SPA becomes even more urgent when the pages need to load large amounts of data and the backend is slow. For example, a retail web site that lets user filter products based on price, color, gender etc. needs to download the entire catalog. This can be slow, especially when the backend is

legacy and slow. SPA lets you download this data once and then share it between different views.

## 11.5 HTML5 History API

- The history API lets an application manipulate the browser's history stack. You can:
  - ◇ Push new URL to the stack
  - ◇ Pop the stack
  - ◇ Listen for any changes to the history stack
- SPAs use this API to keep the application state consistent with the browser's history. For example:
  - ◇ When a new view is transitioned into, a new URL is added to the history stack. The address bar shows this new URL.
  - ◇ When the user clicks the back button, the browser pops the stack. The application listens for this event and reacts to it by transitioning to the previous view. A similar thing happens when the user clicks the forward button.
- A good SPA framework will also allow users to bookmark and share the "artificial" URLs for the views. We will call them **deep links**.

## 11.6 SPA Challenges

Challenges that emerge when using SPA's include:

- Search engine indexing (SEO)
- Web Analytics
- View state preservation during forward and back navigation
- Deep linking: Book marking and sharing of links

### Notes:

The last two items above are addressed by the Angular Component router.

SEO issues can be addressed by Angular Universal which is part of Angular or through the use of prerender.io.

Web Analytics can be accomplished through custom HTTP calls to an analytics server.

Search engines are designed to index pages on a web server. Since SPA apps don't produce pages on a server there is nothing to index. This is addressed by Angular Universal which renders Angular application screens as pages on a server. See: <https://angular.io/guide/universal>. Prerender.io can be used for the same purpose though it is not specific to Angular projects.

Many pages use analytics API to keep track of user activities. Most of these APIs do the tracking at page load time. In an SPA the page is loaded only once. As the user navigates between the views, the analytics tool may not be able to capture the transitions.

As the user moves forward from one view to the next, we need to preserve the state of the previous views so that we can restore them when the user starts to navigate back. For example, if user searches for product and then clicks on a product from the search result. This shows the product details page. When the user clicks the back button of the browser we should show the search result. This can be done by either saving the search result or saving the search term and doing a fresh query.

Deep link allows a user to bookmark and share a link while the user has navigated deep inside the application. For example, the user may search for a product and click on a product details link. Then click the reviews link. She should be able to share the reviews link with others. This can be challenging in an SPA.

### 11.7 Implementing SPA's Using Angular

- Very simple SPAs can be quickly built using structural directives like `ngIf` and `ngSwitch`. Use them to show and hide components in a page.
- For any meaningful SPA use the router API.
- The Component Router enables developers to implement:
  - ◇ Forward navigation via links and buttons.
  - ◇ Back navigation via the browser's back button.
  - ◇ Navigation via bookmarks.
  - ◇ Manual navigation by typing into the address bar.
- These features are covered in detail in the Angular Component Router chapter.

## **11.8 Summary**

- In this chapter we covered:
- The Concept of Single Page Applications (SPA).
- SPA Advantages.
- SPA Challenges.
- Implementation of SPA's using Angular

## Chapter 12 - The Angular Component Router

---

### *Objectives*

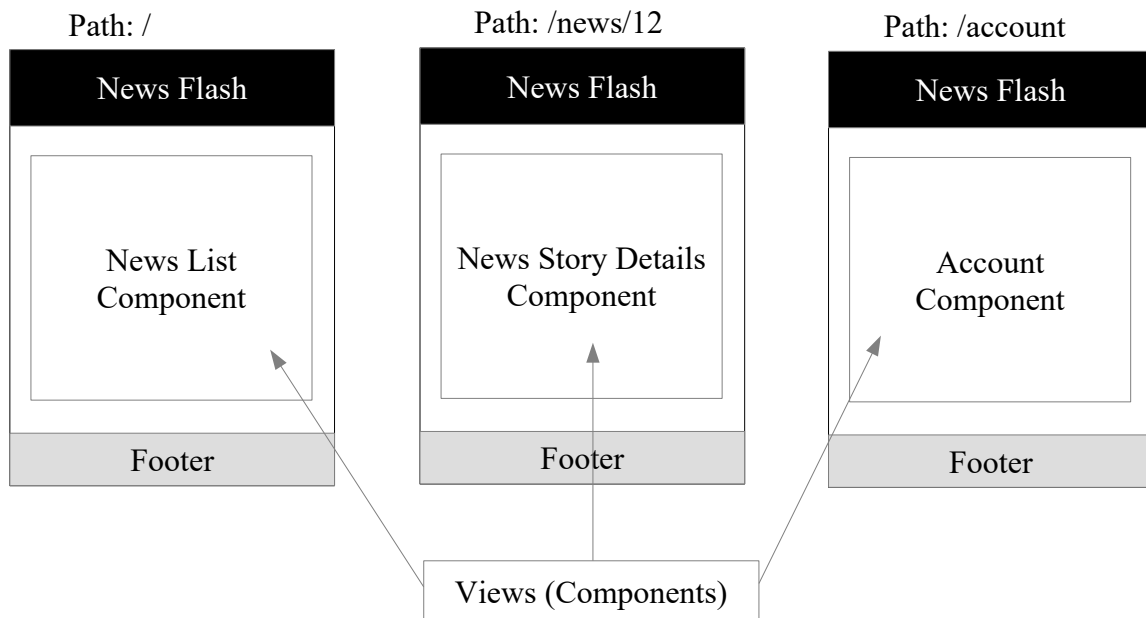
Key objectives of this chapter

- What is web app navigation?
- The Angular Component Router
- Defining route tables
- Navigation using anchor tags
- Navigating programmatically
- Route parameters
- Query parameters
- HTML5 Mode
- Enabling Bookmarks

### 12.1 The Component Router

- The Angular Component Router can be used to navigate between application views. Each view is implemented by a component.
- Navigation is triggered in various ways:
  - ◇ By a user clicking a link
  - ◇ Programmatically in code
- The Router allows navigation in Angular apps to behave the same way as in traditional browser based apps. For example:
  - ◇ The user can use the back and forward button to navigate through history.
  - ◇ A view can be book marked.
- Not all applications require routing capabilities so:
  - ◇ The Component Router is not included by default
  - ◇ When used it must be imported from "`@angular/router`" package

## 12.2 View Navigation



- Only a portion of the page changes as the user navigates through the application. A separate component is used to render that portion.
- Angular lets us associate a URL path with each view component. As the user navigates between the views the corresponding path is displayed in the address bar.

## 12.3 The Angular Router API

- The API is available from **RouterModule**. This needs to be imported into the AppModule.
- A few commonly used classes and services available from the module are:
  - ◇ **Routes** - Used to setup the route table (association between a path and a component).
  - ◇ **RouterLink** - A directive used to add a link to a routed component in a template.
  - ◇ **Router** - A service used for programmatic navigation.

## Notes

Basic definitions of the terms above are listed here for reference:

Router	When the browser's URL changes the Router looks up and displays the appropriate ComponentView.
Routes	The Routes array contains URL to component mappings used to navigate the app.
Route	Individual URL to component mappings are called Routes.
RouterOutlet	The RouterOutlet directive (<router-outlet>) appears in the HTML template. It tells Angular where to insert a selected ComponentView.
RouterLink	RouterLink is an Angular directive inserted in an HTML element to map that element to a Route so that clicking on the element triggers navigation.
Link Parameters Array	An array that the router translates into a routing instruction. It is used when navigating via anchor links or when navigating programmatically.
Routing Component	The main or host component that includes the RouterOutlet directive.
Component View	The view defined by a component. It is inserted at the RouterOutlet location and made visible when a Route referencing it is selected.

## 12.4 Creating a Router Enabled Application

- Create a project using the **--routing** option.

```
ng new router-test --defaults --routing
```

- This creates a separate feature module in **app-routing.module.ts**. You need to define the route table there.

```
const routes: Routes = [
 { path: '', component: NewsListComponent },
 { path: 'news/:id', component: StoryDetailComponent },
 { path: 'account', component: AccountComponent },
 { path: '**', component: InvalidPageComponent }
]
```

```
];
@NgModule({
 imports: [RouterModule.forRoot(routes)],
 exports: [RouterModule]
})
export class AppRoutingModule { }
```

## Notes

The generated AppModule is setup to import AppRoutingModule.

As you can see above AppRoutingModule imports RouterModule and re-exports it. As a result we don't have to directly import RouterModule from AppModule.

## 12.5 Hosting the Routed Components

- The <router-outlet> tag is used to show the currently routed component. This is usually done from the template of the root component src/app/app.component.html.

```
<!-- Elements common to all views -->
<h1>Welcome to Our Site</h1>

<!-- Host routed component-->
<router-outlet></router-outlet>
```

## 12.6 Navigation Using Links and Buttons

- Use the routerLink directive to supply the path to navigate to:

```
<a href [routerLink]="['/account']">Account
<button [routerLink]="['/']">Home</button>
```

- Note: The link's path must start with a "/" even though in the route table we do not define the path using a leading "/".



## 12.7 Programmatic Navigation

- Sometimes a component needs to programmatically perform navigation.
- The template:

```
<button (click)="goToAccount()">Account</button>
```

- In the component inject Router service and call the navigate() method:

```
import { Router } from '@angular/router'
export class MyComponent {
 constructor(private router: Router) {}

 goToAccount() {
 this.router.navigate(['/account']);
 }
}
```

## 12.8 Passing Route Parameters

- Route parameters let us pass basic information during navigation. Such as pass the news story ID to the "/news" path.
- A route like this has one or more parameter placeholders:

```
{ path: 'news/:id', component: StoryDetailComponent }
```

- This route translates to a URL like this: <http://localhost/news/3001>
- Parameters are mandatory. This means that if the parameter is missing during navigation the indicated Route will not be selected.

## 12.9 Navigating with Route Parameters

- Route parameters are defined for navigation by adding a value to the link parameters array.
- By setting an anchor tag's routerLink:

```
<div *ngFor="let news of newsList">
 <a href
 [routerLink]="['/news', news.idx]">{{news.title}}
</div>
```

- Or by navigating programmatically:

```
this.router.navigate(['/news', idx]);
```

## 12.10 Obtaining the Route Parameter Values

- After the Router takes care of displaying the detail view the detail view needs to:
  - ◇ Retrieve the router parameter value.
  - ◇ Use it to obtain a record to display
- The parameter value can be accessed using an `ActivatedRoute` object. This is injected in the component's constructor.

```
import { ActivatedRoute } from '@angular/router';

export class StoryDetailComponent implements OnInit {
 constructor(private route: ActivatedRoute) {}
 ...
}
```

- Retrieving the route parameter from `ActivatedRoute` can be done either synchronously or asynchronously

## 12.11 Retrieving the Route Parameter Synchronously

- To retrieve the route parameter synchronously add this code to `ngOnInit()`:

```
ngOnInit() {
 let index = this.route.snapshot.paramMap.get('id');
 //Download data about this item
}
```
- This line in bold in the above code retrieves the parameter. Note, use the

same name for the parameter as used in the route table.

## 12.12 Retrieving a Route Parameter Asynchronously

- If the route parameter changes but the path remains the same Angular does not recreate the component instance. Which means `ngOnInit()` is not called again. We can not get the new route parameter if we get it synchronously.
- The solution is to subscribe to the **paramMap** Observable property of the activated route.
- Add the following code in `ngOnInit()`:

```
this.sub = this.route.paramMap.subscribe(
 params => {
 let index = params.get('id');
 this.news =
 this.newsService.getNews(index);
 });
```

## 12.13 Query Parameters

- Query Parameters are key value pairs of data added to the URL that:
  - ◇ Are passed during navigation along with the link array,
  - ◇ Are added to the end of the URL in the address bar,
  - ◇ Can supply multiple pieces of data to the View Component,
  - ◇ Can be used to pass optional data,
  - ◇ Do not require a dedicated entry in the Routes array.
- An example URL including 'id' and 'name' query parameters

`http://servername/appname?id=23&name=steve`

### Notes

Query parameters are passed along with but not inside the link array.

They show up at the end of the URL like this:

`http://server/app?name=value`

Multiple name/value pairs can be passed using query parameters. In contrast, only a single value can be passed when using Route parameters:

```
http://server/app?name=value&email=email_value&phone=555-1212
```

Query parameters are not part of the route so supplying them during navigation is optional. Default values can be set in the component class if desired.

The same Route entry in the Routes array no matter what query parameters are used. For examples these URLs all use the same Route entry:

```
http://server/app
http://server/app?name=value
http://server/app?name=value&phone=5551212
http://server/app?email=joe@xyz.com
```

### 12.14 Supplying Query Parameters

- Navigating via Anchor link:

```
<a href [routerLink]="['/people']"
 [queryParams]="{id:23, name:'steve'}" >Account
```

- Navigating programmatically:

```
this.route.navigate(['people'],
 { queryParams: {id:23, name:'steve'}});
```

- In both cases, the queryParams are set independently of (outside) the link array.

### 12.15 Retrieving Query Parameters Asynchronously

- Subscribe to the **queryParams** property of the ActivatedRoute object.

```
this.route.queryParams.subscribe(params => {
 let id:number = +params.get('id') //Convert to number
 let name:string = params.get('name')
});
```

## Notes:

Just like path parameters if the query parameter changes but the path remains the same Angular does not recreate the component. In that case, a component must always monitor changes to the query parameters. This is done by subscribing to the `routerState` observable.

## 12.16 Problems with Manual URL entry and Bookmarking

- As the user navigates through the application Angular keeps changing the URL in the browser's address bar. Example:

```
http://localhost/news/2001
```

```
http://localhost/account
```

- But what happens when we share this URL with others or bookmark it? Entering the same URL directly in the browser's address bar will result in a 404 "not found" error from the server. That is because the web server does not know anything about paths like `/account`.
- To solve this problem you need to configure the web server to always return **index.html** if the requested path is not found. For details see: <https://angular.io/guide/deployment>.

## 12.17 Summary

In this chapter we covered:

- The Angular Component Router
- Defining route tables
- Navigation using anchor tags
- Navigating programmatically
- Route parameters
- Query parameters
- Enabling Bookmarks



## Chapter 13 - Advanced HTTP Client

---

### *Objectives*

Key objectives of this chapter are to learn

- How to customize the HTTP calls using options
- More details about Observables
- How to make sequential and parallel HTTP calls

### 13.1 Request Options

- All the HttpClient request methods like get() and post() take an optional object as the last argument. You can use this option to change the behavior of the calls. Example:

```
this.http.get(`/app/person`, {'observe': 'response'})
```

- The options object has the following type. All properties are optional.

```
options: {
 headers: HttpHeaders;
 observe: HttpObserve;
 params: HttpParams;
 reportProgress: boolean;
 responseType: 'arraybuffer' | 'blob' | 'json' | 'text';
 withCredentials: boolean
}
```

### 13.2 Returning an HttpResponse Object

- By default the Observable returned by HttpClient gives us the response body. Advanced applications may need to access to the full response including the status code and headers.
- By setting the "observe" option to "response", we can get an Observable that gives us an **HttpResponse** object.

```
import { HttpResponse } from '@angular/common/http';

getAllPersons() : Observable<HttpResponse<Person[]>> {
 return this.http.get<Person[]>(`/app/person`,
 {'observe': 'response'})
}
```

- A subscriber can get the response details:

```
this.peopleService.getAllPersons().subscribe(
 (response: HttpResponse<Person[]>) => {
 this.list = response.body; //body is a Person[]
 //Get extra info about the response
 let headers:HttpHeaders = response.headers
 console.log(`Content: ${headers.get('content-type')}`);
 console.log(`Status: ${response.status}`);
 console.log(`Url: ${response.url}`);
 }
);
```

### 13.3 Setting Request Headers

- You may need to set custom header information for a request, such as cookies and authentication token. This can be done by setting the **headers** option property to an **HttpHeaders** object.

```
import { HttpHeaders } from '@angular/common/http';

let myHeaders = new HttpHeaders({
 'X-custom-header': 'My stuff',
 'Authorization': 'my-auth-token'
})

//Make a call
this.http.get(`/app/person`, {'headers': myHeaders})
```

### 13.4 Creating New Observables

- So far we have had get(), post() etc. methods create Observable for us. More advanced applications will need to create custom observables.



- RxJS provides us with various functions to create new Observable
  - ◇ of(), from(), new Observable() and more.
  - ◇ Import them from the 'rxjs' module.

## 13.5 Creating a Simple Observable

- The from() and of() functions take a list of payloads that will be emitted.

```
import { Observable, of, from } from 'rxjs';
```

```
let o:Observable<number> = of(1, 2, 3)
let o:Observable<number> = from([1, 2, 3])
```

```
o.subscribe((num) => {
 console.log(num) //Prints 1, 2 and 3
})
```

- Use these functions when you already know what data should be delivered by the Observable at the time of its creation.

## 13.6 The Observable Constructor Method

- Creates an Observable that can asynchronously emit events.

```
let o:Observable<number> = new Observable(observer => {
 let payload = 0
 let timerId = setInterval(()=>{
 observer.next(payload++) //Emits an event
 }, 1000)

 //Return an unsubscribe handler
 return () => {clearInterval(timerId)}
})
```

- The subscriber.

```
let subscriber = o.subscribe((num) => {
 console.log(num)
```

```
 if (num == 10) subscriber.unsubscribe()
 })
```

## Notes

The `Observer()` constructor method takes an arrow function that is called after the observable is created. The arrow function takes as argument the observer. We can call the `next()` method of the observer any time to emit an event.

Optionally, the arrow function can return a cleanup function that is called when the subscriber unsubscribes.

## 13.7 Observable Operators

- Operators are functions that are used to manipulate the data emitted by Observables. For example:
  - ◇ **map** – Converts the data from one type to another.
  - ◇ **filter** – Filters out some of the data from getting delivered to the subscriber.
- Import these functions from the **'rxjs/operators'** module.
- Operators are chained using the **Observable.pipe()** method.

```
import { Observable, of } from 'rxjs'
import { map, filter } from 'rxjs/operators'

let o1: Observable<number> = of(1, 2, 3, 4)
let o2 : Observable<string> = o1.pipe(
 filter(n => n % 2 == 0),
 map(n => `We got ${n}`))

o2.subscribe((msg:string) => console.log(msg))
//Prints: "We got 2", "We got 4"
```

## 13.8 The map and filter Operators

- `filter()` takes a predicate function that receives the emitted data and returns

a boolean value. If the returned value is false then the data is not delivered downstream.

- `map()` takes a function that receives the emitted data, converts it into another data structure and returns it.

## 13.9 The `flatMap()` Operator

- Same as `map()` except the arrow function returns an Observable that emits the transformed value. This is useful when the transformation requires an asynchronous process like an HTTP call.

```
let o1 : Observable<number> = of(1, 2, 3)
let o2 : Observable<string> = o1.pipe(
 flatMap(val => of(`We got ${val}`)))
o2.subscribe(val => console.log(val))
```

```
//Prints
We got 1
We got 2
We got 3
```

## 13.10 The `tap()` Operator

- Intercepts every event emitted by the source Observable but does not modify the source in any way. It is useful for performing some kind of side effect like logging and caching.

```
let o1:Observable<number> = of(1, 2)
let o2:Observable<number> = o1.pipe(
 tap(val => console.log(`Source emitted ${val}`)))
o2.subscribe(val => console.log(`Subscriber got ${val}`))
//Prints
Source emitted 1
Subscriber got 1
Source emitted 2
Subscriber got 2
```

## 13.11 The zip() Combinator

- Combines multiple observables into one. This lets a subscriber wait for events emitted from multiple Observables.
- Import it from 'rxjs/index'.

```
import { zip } from 'rxjs/index'

let o1:Observable<number> = of(1, 2, 3)
let o2:Observable<string> = of("One", "Two")
let o3: Observable<[number, string]> = zip(o1, o2)

o3.subscribe((results:[number, string]) => {
 console.log("Received %d and %s", results[0], results[1])
})

//Prints. The third number is not printed
Received 1 and One
Received 2 and Two
```

### Notes

The Observable o1 fires three events. But o2 fires only two events. The zip() operator correlates events from all sources. As a result, the subscriber gets only the first two events.

## 13.12 Caching HTTP Response

- A service can cache responses in memory or local storage.

```
import { of, Observable } from 'rxjs'
import { tap } from 'rxjs/operators'

export class PeopleService{
 private cachedList:Person[] //The cache

 constructor(http: HttpClient){}

 getAllPersons() : Observable<Person[]> {
 if (this.cachedList !== undefined) {
```

```
 return of(this.cachedList) //Cache hit!
 }
 return this.http.get<Person[]>("/app/person").pipe(
 tap(list => this.cachedList = list))
 }
}
```

## Notes

In the example above, the service caches the array of all People objects. If a cache is found it constructs an Observable using the of() operator and returns it. If a cache does not exist it makes a call but intercepts it using the tap() operator. From the tap() operator it caches the response.

### 13.13 Making Sequential HTTP Calls

- If a web service call depends on another call we need to make the calls sequentially.
- The following makes two HTTP calls sequentially. The final observable emits the combined responses from the two calls.

```
let o:Observable<Object[]> =
this.http.get("https://localhost/news/1")
 .pipe(flatMap(news =>
 this.http.get("https://localhost/news/1/comments")
 .pipe(map(comment => [news, comment]))))

o.subscribe(result => {
 console.log(result[0]) //News
 console.log(result[1]) //Comments
})
```

### 13.14 Making Parallel Calls

- Calls that do not depend on each other can be made in parallel.

```
import { zip } from 'rxjs/index';

let o1 = this.http.get("https://localhost/news/1")
let o2 = this.http.get("https://localhost/news/1/comments")
```

```
zip(o1, o2)
 .subscribe(([news, comments]) => {
 console.log(news)
 console.log(comments)
 })
```

### 13.15 Customizing Error Object with `catchError()`

- A service may decide to offer more meaningful errors in a custom error object instead of the default `HttpErrorResponse`.
- You can intercept an error with the **`catchError()`** operator and create a new Observable with a custom error object using **`throwError()`**.
- The following transforms the error object from `HttpErrorResponse` to a string.

```
//The service
import { Observable, throwError } from 'rxjs'
import { catchError } from 'rxjs/operators'

return this.http.get("https://localhost/posts/1").pipe(
 catchError(err => throwError("Sorry there was a
 problem")))

//The component
let o:Observable = ...

o.subscribe(
 (result) => console.log(result),
 (error:string) => console.log(error)
)
```

#### Notes

The projection function for `catchError` needs to return a new Observable. Here we create a failed Observable using `throwError` with a custom error descriptor (a string in this case).

## 13.16 Error in Pipeline

- If an operator returns a failed observable then the subsequent operators are skipped and the error is delivered to the subscriber.

```
let o = of(1, 2, 3, 4)

o.pipe(
 tap(n => console.log("First tap:", n)),
 flatMap(n => n == 2 ? throwError("Two is bad") : of(n)),
 tap(n => console.log("Last tap:", n)))
.subscribe(n => console.log("Subscriber got:", n),
 err => console.log("Error:", err))

//Prints
First tap: 1
Last tap: 1
Subscriber got: 1
First tap: 2
Error: Two is bad
```

## 13.17 Error Recovery

- Mobile applications need to handle poor connectivity. Two common strategies:
  - ◇ Retry failed HTTP calls. Use **retry** or **retryWhen** operator for this.
  - ◇ In case of failure return a previously cached result if available. Use the **catchError** operator to convert a failure into a successful Observable that delivers the cached data.

```
let cachedData = {...} //Previously cached data
let o = this.http.get("https://localhost/news/1")
 .pipe(
 retry(3), //Try 3 more times in case of an error
 catchError(err => cachedData ? of(cachedData) :
 throwError(err)))
```

```
o.subscribe(result => console.log(result))
```

### Notes

In the example above we pipe the observable returned by the `get()` call through the `retry` and `catchError` operators. If the source observable fails the `retry` operator will resubscribe and try up to 3 times. If all of these attempts fail then an error will be raised by the `retry` operator. In that case the `catchError` operator get involved. If data is available in the cache then `catchError` will return a successful observable that emits that cached data. Else it will return a failed observable by calling `throwError`.

Both `retry` and `catchError` operators have no impact if the source observable was successful. They simply emit the data emitted by the source.

Generally speaking limit retries to the GET calls only. GET calls are supposed to have no permanent side effects on the state of the business data in the backend. This makes them safe to retry.

Making the POST, PUT etc. calls retryable will require extra care. You need to make sure that they are idempotent. Meaning if the server has already executed the call and the client mistakenly attempts to try it again the server will handle the situation gracefully and essentially do nothing.

## 13.18 Summary

In this chapter we covered:

- We can use options with the HTTP calls to:
  - ◇ Obtain the full response container header and status information.
  - ◇ Supply custom headers with a request
- Operator methods manipulate an Observable and return a new one. We can use them to achieve all kinds of common use cases like:
  - ◇ Caching
  - ◇ Making sequential and parallel calls



## Chapter 14 - Angular Modules

---

### *Objectives*

Key objectives of this chapter

- Why Angular modules?
- Types of feature modules
- Creating modules
- @NgModule decorator
- Importing modules

### 14.1 Why Angular Modules?

- An Angular module groups components, services, directives etc. that can then be imported when needed.
- This has many benefits:
  - ◇ Modules help you organize your application in logical groupings.
  - ◇ Modules can hide complexity by only exposing components, services etc. that are of use by the rest of the application.
  - ◇ If you decide not to use a module in future then it's entire code base including all components and services will be excluded from the production build by the tree shaker.
  - ◇ A module can be "lazy loaded" only when required to improve application initialization.
  - ◇ Knowing how to create modules will help you create Angular libraries. Libraries contain reusable code that are of use across multiple applications.
- An Angular application always has at least one module, the "root" AppModule. You can create additional ones if needed.

### 14.2 Anatomy of a Module Class

- A TypeScript class decorated with the **@NgModule** decorator.

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { CreditCard } from '...';
import { MakePayment } from '...';

@NgModule({
 //Other modules used
 imports: [FormsModule],
 //Components, directives, pipes defined in this module
 declarations: [CreditCard],
 //Components, directives, pipes exported
 exports: [CreditCard],
 providers: [MakePayment] //Services provided
})
export class PaymentModule { }
```

### 14.3 @NgModule Properties

- Some of the most important properties of @NgModule are:
  - ◇ **declarations** - list of components/directives/pipes that belong to this module
  - ◇ **imports** - list of modules whose exported components/directives/pipes should be available to templates in this module
  - ◇ **exports** - list of components/directives/pipes/modules that are to be made available for use outside of this module.
  - ◇ **providers** - list of services available inside this module and also exported by this module. Deprecated and should not use. This prevents proper tree shaking from happening.

#### @NgModule Properties

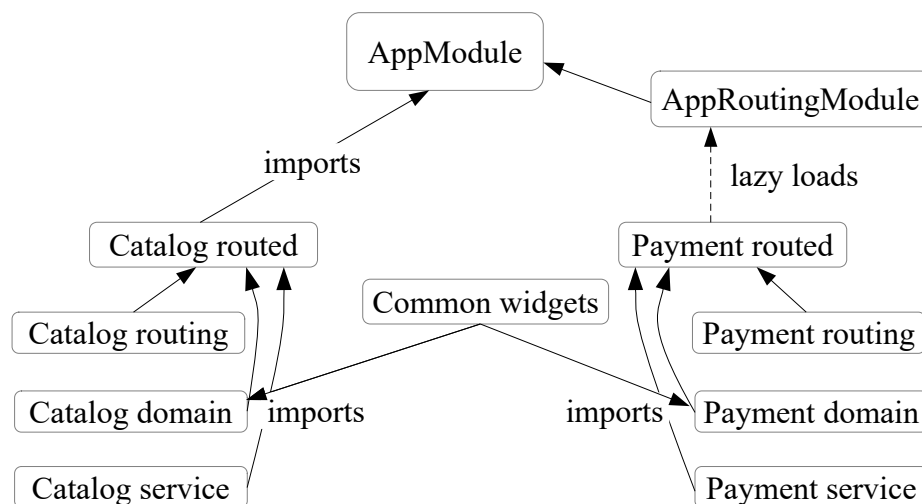
Every component, directive, and pipe defined inside the module must be listed in the **declarations** array. But not all of them need to be exposed to the outside world. If a module needs to expose some of these artifacts for use by other modules they need to be also listed in the **exports** list.

Services in the module that are also available to other modules can be listed in the **providers** list.

## 14.4 Feature Modules

- A "Feature Module" is simply an Angular module that is not the root module of an Angular application.
- You can create new feature modules based on these guidelines:
  - ◇ **Domain module** – Contains components related to specific business functions. Such as: product catalog display, payment processing and account management.
  - ◇ **Routed module** – Contains top level view components that are added to the route table.
  - ◇ **Routing module** – Contains the route table, route guards etc. for a routed module.
  - ◇ **Service module** – Contains only services.
  - ◇ **Widget module** – Contains reusable components, directives and pipes that are used throughout the application.

## 14.5 Example Module Structure



### Notes

The diagram above shows a common pattern. The application has two main feature areas – Catalog and

Payment. Each of these areas have:

- A **routed** and a **routing** module. All top level view components are declared in the routed module. But there is no need to export them. These components are added to the route table in the routing module.
- A **domain** module where lower level components are declared. They need to be also exported so that the top level view components can use them.
- A service module where services are provided.

The payment routed module is imported by AppModule. This will eagerly load that module.

The catalog routed module is lazily loaded by the AppRoutingModuleModule. We will learn to do that in another chapter.

Common UI components are declared in a widgets module and also exported. This module is imported by AppModule so that these components can be used anywhere in the application.

## 14.6 Create a Domain Module

- Use this command from the root folder.

```
ng generate module payment-domain
```

- Create components inside it as follows.

```
ng g c payment-domain/credit-card-form
```

- You need to manually export the component from the domain module.

```
@NgModule({
 exports: [CreditCardForm, ...]
})
export class PaymentDomainModule { }
```

## 14.7 Create a Routed/Routing Module Pair

- To create a routed module along with an associated routing module:

```
ng generate module payment-views --routing
```

- Create a top level view component:

```
ng g c payment-views/billing-view
```

- Add the component to the route table of the routing module. There's no need to export these components.

```
const routes: Routes = [
 {path: "billing", component: BillingViewComponent},
 ...
];
```

```
@NgModule({...})
export class PaymentViewsRoutingModule { }
```

## 14.8 Create a Service Module

- Create the module.

```
ng generate module payment-services
```

- Create a service inside it.

```
ng g s payment-services/refund
```

- Configure the service to be provided by the service module. This is going to ensure that the service can be injected only if the service module is imported.

```
@Injectable({
 providedIn: PaymentServicesModule
})
export class RefundService {
 constructor() { }
}
```

## 14.9 Creating Common Modules

- Create a common widget or service module.

```
ng g m common-widgets
ng g m common-services
```

- Add artifacts there.

```
ng g c common-widgets/modal-popup
ng g s common-services/logging
```

- Export the components in the widget module.
- Configure the services to be provided the by the service module.
- These common modules are imported by the AppModule there by making the artifacts available throughout the application.

## 14.10 Using One Module From Another

- Although modules define what is in a module, the contents are not automatically available in other modules. One module must be imported by another module in order to be used.
  - ◇ All components, directives, pipes exported by the imported module will become available for use.
  - ◇ All services provided by the imported module will become available for injection.
- In this example the payment services module is being imported by the payment routed module.

```
@NgModule({
 imports: [PaymentServicesModule],
 ...
})
public class PaymentViewsModule{}
```

## 14.11 Summary

In this chapter we covered:

- Modules break up a large application in logical groups of components, services etc.
- Every application has at least the root AppModule. You can create additional feature modules as needed.
- When creating a new feature module follow the guidelines given here.
- A feature module can serve as a service injector. In that case the module must be imported by another module for the services to be available there.





## Chapter 15 - Advanced Routing

---

### *Objectives*

Key objectives of this chapter

- ◇ External route configuration file
- ◇ Dedicated routing module
- ◇ routerLink Prefixes
- ◇ routerLinkActive binding
- ◇ Wildcard route
- ◇ redirectTo
- ◇ Default Route
- ◇ Child Routes
- ◇ Lazy Loading Modules via Child Routes
- ◇ Navigation Guards

### 15.1 Routing Enabled Feature Module

- You can define top level view components in a feature module. These components are only used from a route table and never used in a template. As a result, you don't need to add them to the exports list of the feature module.
- Use the `--routing` option with Angular CLI to create the feature module.

```
ng g module checkout --routing
```

- This creates the "routed" feature module `CheckoutModule` in **`checkout.module.ts`** and a separate routing module in **`checkout-routing.module.ts`**. Add all your routes there. Example:

```
const routes: Routes = [
 {path: "shipping", component: ShippingComponent},
 {path: "billing", component: BillingComponent}
];
```

```
@NgModule({
 imports: [RouterModule.forChild(routes)],
 exports: [RouterModule]
})
export class CheckoutRoutingModule { }
```

## Notes

Note for feature modules the route table is installed using `RouterModule.forChild(routes)`. This causes the feature module's routes to be merged into the root route table.

## 15.2 Using the Feature Module

- Simply import the feature module from the app module as usual. This will automatically merge its routes to the application's route table.

```
import {CheckoutModule} from './checkout/checkout.module'

@NgModule({
 imports: [CheckoutModule, ...]
 ...
})
```

- In the templates add links to the view components in the feature module.

```
<a href [routerLink]="['/shipping']">Shipping
<a href [routerLink]="['/billing']">Billing
```

- Note, when a feature module is imported and used this way it is loaded eagerly. Meaning the feature module's code gets downloaded when the application is first accessed in the browser.

## 15.3 Lazy Loading the Feature Module

- Lazy loading a feature module will speed up the initial loading of the application in a browser.
- Do not import the feature module from the application module as we have shown prior. Instead in the route table of the main application module add

a route for the feature module. In **app-routing.module.ts**:

```
const routes: Routes = [
 ...
 {path: "checkout",
 loadChildren: ()=>import('./checkout/checkout.module')
 .then(m => m.CheckoutModule)
 }
];
```

- ◇ loadChildren points to a function that calls a standard module import()
- ◇ Note that we are importing module, not a component
- View components of the feature module are relative to the path of the feature module. For example, the fully qualified path of ShippingComponent will be "/checkout/shipping".

### Notes:

Notice the lack of import statement for the lazy-loaded module. If we were to import it here then Angular would load it up front as usual.

Similarly, the feature module can not be listed in the **imports** list of the main application module. A side effect of lazy loading is that if the feature module defines any services, components, and directives, they can't be used by the rest of the application.

## 15.4 Creating Links for the Feature Module Components

- Supply fully qualified path for the components. Prefix with the path for the feature module.
- Example:

```
<a href [routerLink]="['/checkout/shipping']">Shipping
<a href [routerLink]="['/checkout/billing']">Billing
```

## 15.5 More About Lazy Loading

- Angular CLI build system will output a separate JS file for a lazily loaded feature module.
- The JS code for the lazy loaded module will be downloaded only after the

user visits a view component from that module. Once loaded the feature module will stay loaded in the browser.

- Because of lazy loading any component, directive, services etc. defined in the feature module can not be used by the rest of the application.

## 15.6 Preloading Modules

- You can proactively load lazy loaded modules in background after the application loads. This improves UI performance. When the user navigates to a lazy loaded module it will likely be already loaded.
- Enable preloading of all lazy loaded modules when installing the route table. In **src/app/app-routing.module.ts** do:

```
import { PreloadAllModules } from '@angular/router';

@NgModule({
 imports: [RouterModule.forRoot(routes,
 {preloadingStrategy: PreloadAllModules})],
 exports: [RouterModule]
})
export class AppRoutingModule { }
```

## 15.7 routerLinkActive binding

- Multiple navigation links are often shown in the same view.

Views: [Home](#) [Products](#) [About](#)

- When a user chooses a specific link you may wish to highlight the link element in some way to indicate the current/active view

Views: [Home](#) [Products](#) **[About](#)**

- The routerLinkActive directive lets you do this by specifying a CSS class (or classes) to be added to the element when the view it points to is active:

```
<a [routerLink]="['/about']"
 routerLinkActive="active">About
```

- The CSS class 'active' is added to the anchor tag when the '/about' path in the routerLink is active (when the *about* view is displayed):

```
.active { font-weight: bold; }
```

## 15.8 Default Route

- A default route can be applied when the application's base URL is used

```
http://servername/appname
```

- The default route is specified in the route definition as a blank path:

```
{path: '', component: HomeComponent},
{path: 'home', component: HomeComponent},
```

- Both "/home" and default path will load HomeComponent.

## 15.9 Wildcard Route Path

- The Wildcard path:
  - ◇ Is used in route definitions
  - ◇ Is indicated by two asterisks: '\*\*'
  - ◇ Example: { path: '\*\*', component: ErrorComponent }
  - ◇ Matches any route
  - ◇ Is typically used to redirect to an error view.
  - ◇ When used appears in the last route
- Routes are evaluated in the order they appear in the Routes array so the last route is only applied if none of the other routes match

## 15.10 redirectTo


- **redirectTo** allows redirection of a given path to another existing route.  
Example:

```
{path: 'home', component: HomeComponent},
{path: '', redirectTo: '/home', pathMatch: 'full' }
{path: 'overview', redirectTo: '/home', pathMatch: 'full' }
{path: 'error', component: ErrorComponent},
{path: '**', redirectTo: '/error', pathMatch: 'full' }
```

- Note, the path for `redirectTo` starts with a `"/`. Without the `"/` a relative redirection will take place.
- The **`pathMatch`** property can be:
  - ◇ 'full' - Path in URL must match exactly for a redirection to take place. Example: `"/overview/page"` will not redirect to `"home"`.
  - ◇ 'prefix' - Path in URL needs to start with the path in the route for a redirection to take place. `"/overview/page"` will redirect to `"home"`.

## 15.11 Child Routes

- Some views can be navigated to only within another view. Example: a product details view may have tabs for Overview, Specification and Reviews. When the user clicks on a tab, the corresponding subview is displayed and the URL address changes accordingly. This can be implemented in Angular using Child Routes



**LG OLED TV<sub>4K</sub>**

**PRODUCT HIGHLIGHTS**

- UHD 3840 x 2160 OLED Panel
- HDR10, Dolby Vision, HLG HDR Cc
- Screen Mirroring Technology
- Built-In Wi-Fi & Ethernet Connecti

[Show more](#)

★★★★★ Reviews 17 | [Q&A](#)

Screen Size: 65"

55" 65"

Configuration: TV Only

TV Only with Wall Mount

OVERVIEW SPECS QUICK COMPARE REVIEWS 17 Q&A 12 AI

## 15.12 Defining Child Routes

- Child routes are added to an existing route via its ***children*** property:

```
export const routes: Routes = [
 { path: 'product-list', component: ProductList },
 { path: 'product/:id', component: ProductDetails,
 children: [
 { path: '', redirectTo: 'overview', pathMatch: 'full' },
 { path: 'overview', component: Overview },
 { path: 'reviews', component: Reviews },
 { path: 'specs', component: Specifications }
]
 }
];
```

- Example URLs for the child views:

```
http://servername/product/11/overview
http://servername/product/12/reviews
```

- The plain product path is set up so that it defaults (redirects) to the overview tab.

## 15.13 <router-outlet> for Child Routes

- Components with child routes must include a <router-outlet> element in their HTML template to hold the child route's component.

```
@Component({
 selector: 'product-details',
 template: `
 <p>Product Details: {{id}}</p>
 <p>
 <a [routerLink]="['overview']">Overview
 <a [routerLink]="['reviews']">Reviews
 <a [routerLink]="['specs']">Technical Specs
 </p>
 <router-outlet></router-outlet>`
})
```

```
export default class ProductDetails {}
```

- This is in addition to the `<router-outlet>` in the app component:

```
<!-- app.component.ts -->
<h1>{{title}}</h1>
<a [routerLink]="['/home']" >Home
<a [routerLink]="['/about']" >About
<a [routerLink]="['/products']" >Products
<router-outlet></router-outlet>
```

## 15.14 Links for Child Routes

- A component with child routes should link to them using relative paths (that is, do not prefix with `"/`).

```
<a [routerLink]="['overview']">Overview
<a [routerLink]="['reviews']">Reviews
```

- A child view component can have links to the parent using the `"../"` prefix.

```
<a [routerLink]="['../home']">Home
```

## 15.15 Navigation Guards

*navigation guards* are used to restrict access to views

- Guards are useful when:
  - ◇ A view is restricted to authorized ( logged in ) users
  - ◇ Data needs to be fetched before a view is shown
  - ◇ Data needs to be saved before leaving a view
- A guard is written as a service that implements one of these interfaces:
  - ◇ **CanActivate** - guards navigation to routes
  - ◇ **CanActivateChild** - guards navigation specifically to child routes
  - ◇ **CanDeactivate** - guards navigation away from current view



- ◇ **Resolve** - retrieves data before route is activated
- ◇ **CanLoad** - guards access to asynchronously loaded modules

## 15.16 Creating Guard Implementations

- Create the service.

ng g s auth

- Guards are created as services that implement one of the guard interfaces:

```
import { CanActivate } from '@angular/router';

@Injectable({
 providedIn: 'root'
})
export class AuthService implements CanActivate {
 isLoggedIn: boolean = false;
 login(){ this.isLoggedIn = true; }
 logout(){ this.isLoggedIn = false; }
 canActivate() { return this.isLoggedIn; }
}
```

- The `canActivate()` method of *CanActivate* guard interface needs to return true or false indicating if a new route can be transitioned into.
- To set login state the service would be injected into a component where its `login()/logout()` methods could be called by the component's code.

## 15.17 Using Guards in a Route

- Install one or more route guards in the route table:

```
{
 path: 'admin',
 component: AdminComponent,
 canActivate: [AuthService]
}
```

}

- The route above can only be activated when the AuthService's *isLoggedIn* property is true.
- If needed multiple guard interface implementations can be assigned in the *canActivate* array. Each guard will be checked in turn before activating the route.

## 15.18 Summary

In this chapter we covered:

- Creating routing enabled project and feature modules.
- Lazy loading feature modules.
- routerLink Prefixes
- routerLinkActive binding
- Wildcard route
- redirectTo
- Default Route
- Child Routes
- Navigation Guards

## Chapter 16 - Unit Testing Angular Applications

---

### *Objectives*

Key objectives of this chapter

- Angular testing technologies and setup
- Jasmine unit test basics
- Angular TestBed and test configuration
- Testing a service
- Testing a component

### 16.1 Unit Testing Angular Artifacts

- Just like any other development project, it is important to test Angular applications. This includes:
  - ◇ Manual unit testing during development
  - ◇ Automated unit testing. This will be the focus of this chapter.
  - ◇ Manual quality assurance testing by professional testers
- Angular applications are modular so with the right tools and techniques it is possible to perform robust unit/integration testing
- You are encouraged to write unit test scripts and run them after major changes to the codebase and before doing every release build.

### 16.2 Testing Tools

- **Jasmine** - The Jasmine test framework is for testing JavaScript code
  - ◇ Jasmine is probably the most important tool used, besides Angular itself, as the unit tests themselves are written according to Jasmine
- **Karma** - Helps simplify running Jasmine tests although it can also work with other frameworks
  - ◇ Karma is a Node.js tool and requires Node.js and npm installed
  - ◇ You can run your Jasmine test scripts in multiple browsers like Firefox

and Chrome (default)

- ◊ Karma will automatically watch all JavaScript files and re-run the tests if any one of them is modified
- **Protractor** - Simulate user activities on a browser for "end to end" testing
  - ◊ This chapter will not focus on Protractor
- **Angular testing utilities** - Angular provides API and tools to write unit tests. They can be called from your Jasmine test cases.

## Testing Tools

With Jasmine tests, you can generate an HTML "test runner" to run in a browser without Karma. Karma can automatically generate this for running tests and run the tests in a browser automatically, so Karma simplifies the testing process.

The main focus of this chapter is the Angular testing utilities and integration with Jasmine tests. Protractor testing is less Angular-specific and does not require the Angular testing utilities.

Jasmine – <https://jasmine.github.io/>

Karma – <https://github.com/karma-runner/karma>

Protractor – <http://www.protractortest.org>

## 16.3 Typical Testing Steps

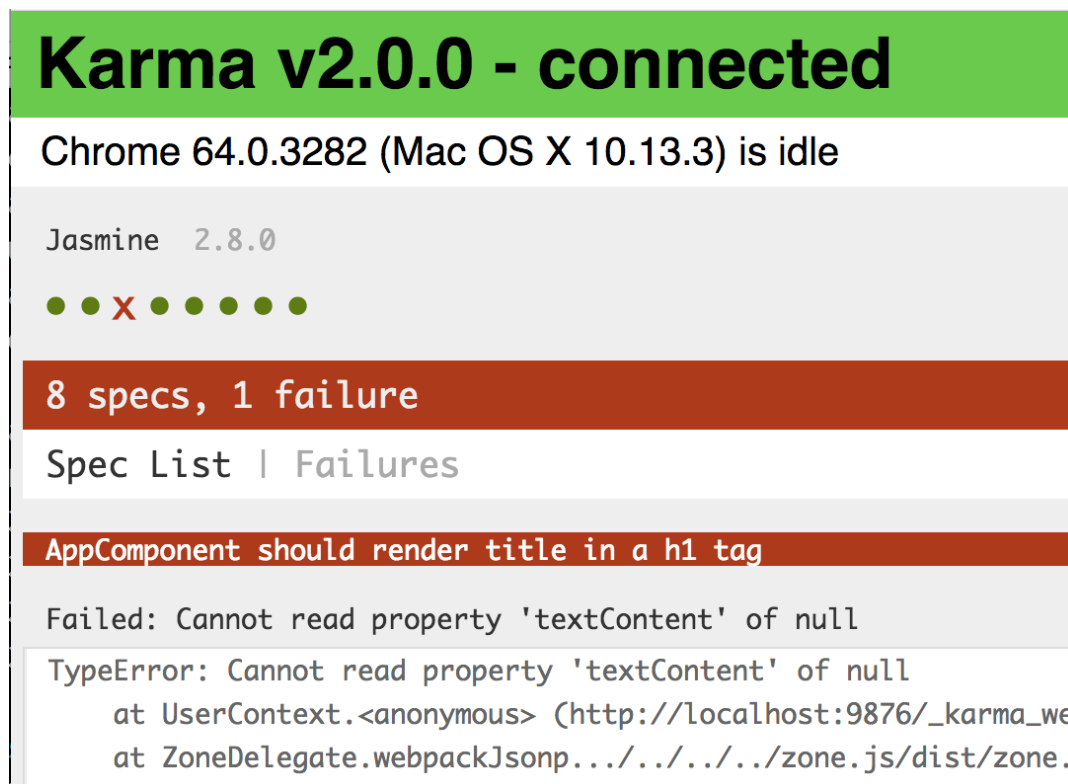
- Define test scripts. Angular CLI creates a file ending in **'spec.ts'** for a generated component, service etc. Example:

`banner.component.ts` is tested by `banner.component.spec.ts`

- You can create additional test scripts. Just give them a **'spec.ts'** extension.
- Run the **'npm test'** or **'ng test'** command to run the tests. This will:
  - ◊ Do a build.
  - ◊ Launch a separate browser window (Chrome by default) and run all test scripts inside it.
  - ◊ Show the test outcomes in the browser window.

- Examine the test output for any test failures
- If you change your code the tests will be run again and the browser will be updated with the new test results
- End testing by hitting Control+C. This will also close the test browser.

## 16.4 Test Results



- Shows a green dot for each successful test and a red X for failed test.
- For each failed test the name of the test and a stack trace is shown

## 16.5 Jasmine Test Suites

- A test suite in Jasmine is defined with the **describe** function, which takes two parameters:
  - ◇ A string, which acts as a title (it is displayed on the spec runner page).
  - ◇ A function that implements the test suite:

```
describe("Test Suite #1", function() {
 // . . . unit tests (specs) or other describe functions
});
```

- A test suite acts as a container for:
  - ◇ Actual unit tests
  - ◇ Other test suites
    - So, you may have a hierarchy of nested test suites.
- Usually you write one test suite in each spec.ts file.

## 16.6 Jasmine Specs (Unit Tests)

- In Jasmine, a unit test is referred to as a **spec**.
- Specs are defined by the **it** function which takes two parameters:
  - ◇ A string, which acts as a title (it is displayed on the spec runner page facilitating the behavior-driven development).
  - ◇ A function that implements the actual unit test.
- Variables declared in the parent *describe* function (the test suite container) are visible in the unit tests (*it* function(s))
- A spec is a container for one or more expectations (assertions) about the code outcome (pass/fail).

```
describe('Simple test suite', () => {
 it("Test addition", () => {
 expect(1+2).toBe(3)
 })
})
```

## 16.7 Expectations (Assertions)

- Expectations are defined by the **expect** function that performs an assertion on the expected value of a variable or a function passed to the *expect* function as a parameter.
  - ◇ The *expect* function's parameter is called the **actual**.
- An expectation is evaluated to either *true* or *false*.

- ◊ An expectation evaluated to *true* is treated as test success (test passed).
- ◊ A *false* result is treated as a failed unit test.
- The *expect* function is chained with a *matcher* function, which takes the *expected* value.
- Example:

```
expect (<actual>) . toBe (<expected value>);
```

where *toBe* is a *matcher* function chained to the *expect* function via the dot-notation.

- **Note:** You can also chain other matchers.

## 16.8 Matchers

- A **matcher** is a function that performs a boolean comparison between the actual value (passed to the chained *expect* function) and the expected value (passed as a parameter to the *matcher*).
- Jasmine comes with a rich catalog of built-in matchers:

<code>toBe</code>	<code>toBeUndefined</code>
<code>toBeCloseTo</code>	<code>toContain</code>
<code>toBeDefined</code>	<code>toEqual</code>
<code>toBeFalsy</code>	<code>toHaveBeenCalled</code>
<code>toBeGreaterThan</code>	<code>toHaveBeenCalledWith</code>
<code>toBeLessThan</code>	<code>toMatch</code>
<code>toBeNaN</code>	<code>toThrow</code>
<code>toBeNull</code>	<code>toThrowError</code>
<code>toBeTruthy</code>	

- Developers can also create their own (custom) matchers.

### Matchers

**toBe** compares using the triple equal sign: `===`

**toEqual** works for simple literals and variables

**toMatch** is used for regular expressions

**toBeDefined** and **toBeUndefined** compare against *undefined*

**toBeNull** compares against null

**toBeTruthy** checks for true

**toBeFalsy** checks for false

**toContain** is used for finding an element in an Array

**toBeLessThan** is used for mathematical '<'

**toBeGreaterThan** is used for mathematical '>'

**toBeCloseTo** is used for precision math comparison

**toThrow** is used for testing if a function throws an exception

## 16.9 Examples of Using Matchers

- Here is a full example of a spec used to test a simple function

```
it ("Test #234", function() {

 var testFunction = function () {
 return 1000 + 1;
 };

 expect(testFunction()) .toEqual(1001);
});
```

- ◇ **Note:** The *testFunction* function would normally be placed in a separate JavaScript file of functions to be tested.

- Similarly, you can test a variable.
- **Note:** The *expect* function performs an expression evaluation, so you can assert math expressions as well, e.g.

```
expect(1000 + 1).toEqual(1001);
```

## 16.10 Using the not Property

- To reverse the expected value (e.g. from *false* to *true*), matchers are chained to the *expect* function via the **not** property.
- For example, the following assertions are functionally equivalent:

```
expect(<variable or a function>).toBe(true);
```



```
expect(<variable or a function>).not.toBe(false);
```

## 16.11 Setup and Teardown in Unit Test Suites

- Jasmine lets you write functions that are called before and after each spec (test). They are registered using *beforeEach()* and *afterEach()*.
- A *beforeEach* function runs the common initialization code (if needed).
- An *afterEach* function contains the clean up code (if needed).
- You can register multiple *beforeEach()* and *afterEach()* functions.
- You can register setup and cleanup functions for the entire test suite using *beforeAll()* and *afterAll()*.

## 16.12 Example of beforeEach and afterEach Functions

```
describe("A Test suite with setup and teardown", function(){
 var obj = {};

 beforeEach(function() {
 obj.prop1 = true;
 // obj.fooBar - undefined !
 });

 afterEach(function() {
 obj = {};
 });

 it("has one property, for sure", function() {
 expect(obj.prop1).toBeTruthy(); // is true, indeed
 });

 it("has undefined property", function() {
 expect(obj.fooBar).toBeUndefined();
 });
});
```

### Example of beforeEach and afterEach Functions

None of the code above is specific to Angular. This just shows the Jasmine-related code and the structure of a Jasmine test.

## 16.13 Angular Test Module

- Every test suite needs to define an Angular module.
- Tests are run using this module and not the real application module. This is necessary so that you can use mock services during testing.
- The item under test and all of its dependencies must be added as providers, declarations, and imports of that module.
- A test module is created using the **configureTestingModule()** method of the **TestBed** class.

## 16.14 Example Angular Test Module

```
import { TestBed, inject } from '@angular/core/testing';
import { HttpClientModule } from '@angular/common/http';
import { BlogService } from '../blog.service';

describe('BlogService', () => {
 beforeEach(() => {
 TestBed.configureTestingModule({
 imports: [HttpClientModule]
 });
 });
 //...
})
```

- Here we are testing BlogService which depends on HttpClient. Our test module had to be set up for that to work.

## 16.15 Testing a Service

- Services are easier to test than components. This should encourage you to write more test scripts for services.
- We will test this service.

```
export class BlogService {
 constructor(private http:HttpClient) { }
```

```
sayHello(name:string) : string {
 return `Hello ${name}`
}

getBlogPost(postId:number) : Observable<Object> {
 return this.http.get<Object>(`/posts/${postId}`)
}
}
```

## 16.16 Injecting a Service Instance

- Before we can call service methods we need to obtain an instance through injection.
  - ◇ We can't simply create an instance using `new`. Doing so will not perform injection within the service if it depends on other services such as `HttpClient`.
- Among the many ways to inject a service the easiest is to call **`TestBed.inject()`**.

```
describe('BlogService', () => {
 let service:BlogService

 beforeEach(() => {
 TestBed.configureTestingModule({...});

 service = TestBed.inject(BlogService)
 })

 it("Should use injection", () => {
 expect(service).toBeTruthy()
 })
})
```

## 16.17 Test a Synchronous Method

- Nothing special about testing a synchronous method. Just call the method and verify its behavior.

```
it("Should call sayHello", () => {
 let name = "Bob"
 let greeting = service.sayHello(name)

 expect(greeting).toBe(`Hello ${name}`)
})
```

## 16.18 Test an Asynchronous Method

- There are several ways to test an asynchronous method in Angular. The easiest is to use the `done()` callback. It is a callback that is passed to an `it()` arrow function. We need to call the `done` function to indicate when the asynchronous test has completed.

```
it("Should get blog post", (done) => {
 service.getBlogPost(1).subscribe((blogPost) => {
 expect(blogPost["id"]).toBe(1)
 expect(blogPost["title"]).toBe("My cool blog post")

 done()
 })
})
```

## 16.19 Using Mock HTTP Client

- To ensure a consistent response from a web service during testing you may need to supply canned responses right from the test script.
- Angular gives us the **HttpClientTestingModule** that provides an alternate implementation of the `HttpClient` service. This mock implementation lets us supply static local data as responses to HTTP calls.
- From the test module import `HttpClientTestingModule` instead of `HttpClientModule`.

```
import {
 HttpClientTestingModule,
 HttpTestingController
} from '@angular/common/http/testing'

beforeEach(() => {
```

```
TestBed.configureTestingModule({
 imports: [HttpClientTestingModule],
 providers: [BlogService]
});
});
```

## 16.20 Supplying Canned Response

```
it("Should get blog post", (done) => {
 service.getBlogPost(1).subscribe((blogPost) => {
 expect(blogPost["title"]).toBe("Mock title")
 expect(blogPost["body"]).toBe("Mock body")

 done()
 })

 let controller: HttpTestingController =
 TestBed.inject(HttpTestingController)

 //Get a mock request for the URL
 let mockRequest = controller.expectOne("/posts/1")

 //Supply mock data
 mockRequest.flush({
 "id": 1,
 "title": "Mock title",
 "body": "Mock body"
 })
})
```

### Notes

The `expectOne()` method returns a `TestRequest` object for a given URL. This mock request can be used to supply canned response data.

Finally, the `TestRequest.flush()` method is used to supply mock response data.

Note: With the alternate implementation of the `HttpClient` service you must provide local data as response. If you don't the subscriber will never receive any response.

## 16.21 Testing a Component

- Here we will test a very simple component.

```
@Component({
 selector: 'app-greet',
 templateUrl: './greet.component.html'
})
export class GreetComponent {
 customerName = "Daffy Duck"

 welcomeBugsBunny() {
 this.customerName = "Bugs Bunny"
 }
}

//Template
<p>Welcome {{customerName}}</p>
<button (click)="welcomeBugsBunny()">Bugs is Here!</button>
```

## 16.22 Component Test Module

- Set up a test module and add all services and modules the component depends on.
- Compile the template of the component. External templates are compiled asynchronously. So we need to wrap the whole module definition in an **async()** call.

```
beforeEach(async () => {
 await TestBed.configureTestingModule({
 imports: [FormsModule, ...],
 declarations: [GreetComponent]
 })
 .compileComponents();
});
```

## 16.23 Creating a Component Instance

- A component is tested in isolation and not as a part of its parent component's template. We need to create an instance of the component using **TestBed.createComponent()**. This gives us a **ComponentFixture** object which gives us access to the actual component instance and much more.
- Angular CLI generates the necessary code. Here it is for review.

```
describe('GreetComponent', () => {
 let component: GreetComponent;
 let fixture: ComponentFixture<GreetComponent>;
 //...
 beforeEach(() => {
 fixture = TestBed.createComponent(GreetComponent);
 component = fixture.componentInstance;
 fixture.detectChanges();
 });
}
```

## 16.24 The ComponentFixture Class

- Commonly used properties:
  - ◇ **componentInstance** - The component instance.
  - ◇ **nativeElement** - The DOM Element object that is at the root of the DOM generated by the component. We can use the DOM API to verify its correctness.
  - ◇ **debugElement: DebugElement** - Provides utility methods useful for testing.
- Useful methods:
  - ◇ **detectChanges()** - By default Angular does not do component state change detection during testing. We need to call this to initiate change detection.
  - ◇ **autoDetectChanges(true)** - Enable automatic change detection.

## 16.25 Basic Component Tests

- Verify initial state.

```
it('Initial state', () => {
 expect(component.customerName).toBe("Daffy Duck");
});
```

- Verify component state change.

```
it('State change', () => {
 component.customerName = "Bob" //Change state

 fixture.detectChanges(); //Trigger state change handling

 let paraText =
 fixture.nativeElement.querySelector('p').textContent
 expect(paraText).toBe('Welcome Bob');
});
```

### Notes

Here **querySelector('p').textContent** is a standard DOM API used to get the contents of the <p> tag.

## 16.26 The DebugElement Class

- This class provides utilities to help write test scripts.
- Useful properties:
  - ◇ **componentInstance** - Same as `ComponentFixture.componentInstance`
  - ◇ **nativeElement** - Same as `ComponentFixture.nativeElement`
  - ◇ **parent : DebugElement** - The parent element.
- Useful methods:
  - ◇ **query()/queryAll()** - Obtain child `DebugElement` by CSS selector `query`.
  - ◇ **triggerEventHandler()** - Used to simulate user interactions like button



clicks.

## 16.27 Simulating User Interaction

- Here we trigger a button click event and verify the change in the component's state.

```
import { By } from '@angular/platform-browser';

it('Should handle click', () => {
 let button : DebugElement =
 fixture.debugElement.query(By.css("button"))

 button.triggerEventHandler("click", null)

 fixture.detectChanges();

 expect(component.customerName).toBe("Bugs Bunny")

 let paraText =
 fixture.nativeElement.querySelector('p').textContent
 expect(paraText).toBe('Welcome Bugs Bunny');
});
```

## 16.28 Summary

- Jasmine and Karma are the two main technologies used in Angular component unit testing
  - ◇ Test code is written using Jasmine
  - ◇ Karma provides automation for configuring and running the tests
- Jasmine tests are defined within 'describe' functions that have 'beforeEach' functions for test setup and 'it' functions for the actual tests
- The Angular TestBed class is the most important class for test configuration and execution
- A separate Angular module is created for each test suite. Tests are run using that module and not using the application module.

- Angular provides us ways to inject service instances and test synchronous and asynchronous methods.
- Components are tested in isolation and not as a part of a larger application. We can use standard DOM API to test the validity of the DOM generated by a component.

## Chapter 17 - Debugging

---

### *Objectives*

Key objectives of this chapter

- Basic Debugging Practices
- Inspecting Components with `ng.probe()`
- Typescript Breakpoints
- Angular DevTools
- Common Exceptions

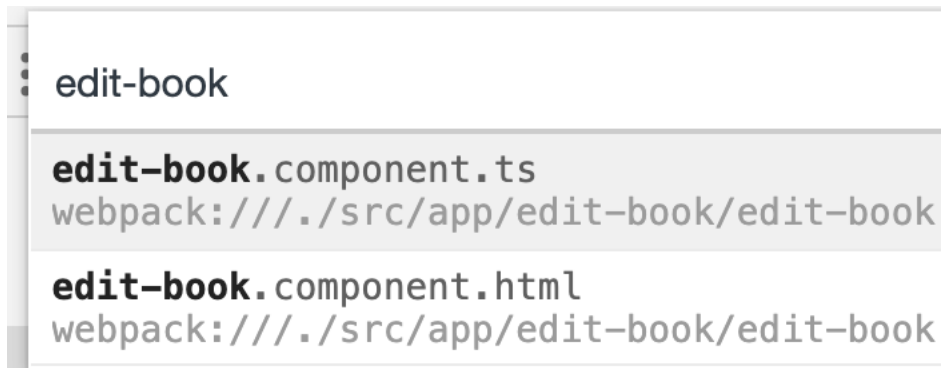
### 17.1 Overview of Angular Debugging

- We employ a mix of tools to aid diagnosis of defects
  - ◇ Conventional breakpoints and debugging techniques available from the browsers (Chrome is recommended)
  - ◇ Logging
  - ◇ Inspecting the component instances
  - ◇ Angular DevTools - a Chrome plugin purpose built for debugging Angular apps
- During development Angular automatically configures itself to run in debug mode. This has these benefits:
  - ◇ The build system generates the map files for the JS files. The debugger shows your Typescript code exactly the way you typed. This preserves the line numbers which aids in putting breakpoints and reading the stack trace.
  - ◇ Angular generates an extra error log

### 17.2 Viewing TypeScript Code in Debugger

- In debug mode, the builder generates map files. This maps the transpiled JS code back to the original TypeScript code.
- Open the **Sources** tab in Chrome and hit Control+P or Command+P.

- Start typing the TS file name and pick it from the list.



- You can now put a breakpoint in the TS code and debug it in a normal fashion

### 17.3 Using the debugger Keyword

- Another way to put a break point is to add the **debugger** keyword in your TS code. (Make sure you remove it once debugging is done!)

- Example:

```
ngOnInit() {
 let i = 10
```

**debugger**

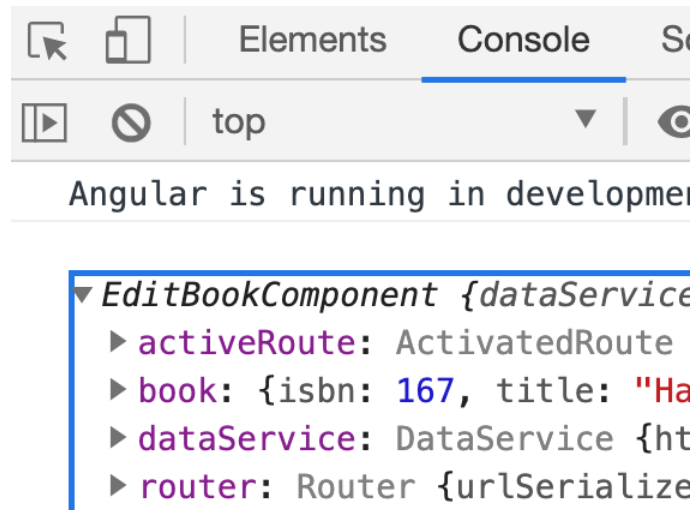
```
}
```

- Use the app and navigate to that area of the code. The debugger will halt at that line.

```
8 export class NewsListComponent implements OnInit {
9
10 constructor() { }
11
12 ngOnInit() {
13 let i = 10; i = 10
14 debugger
15 }
16
17 }
```

## 17.4 Debug Logging

- Use `console.log()` or `console.error()` to print data to help with debugging.
  - ◇ You can log a whole component instance using **`console.log(this)`** and then inspect the object in console.



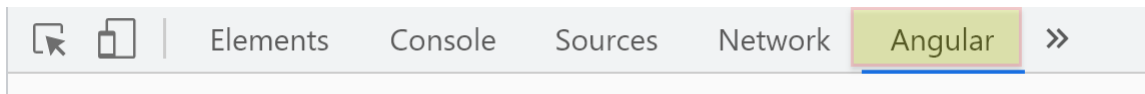
- Use a third party logger like **ngx-logger** for more features. For example, it can POST log messages to a server.

## 17.5 What is Angular DevTools?

- Angular DevTools extends the Developer Tools in the browsers to allow debugging of Angular applications.
- Batarang is a similar extension that is used to debug AngularJS (as opposed to Angular) applications
- Angular DevTools supports applications built with Angular v9.
- Google developed Angular DevTools in collaboration with Rangle.io, the team that built the very first debugging tool for Angular -- Augury.
- Angular DevTools appears as a separate tab in Chrome's developer tools.
- The current release focuses on:
  - ◇ Component structure visualization
  - ◇ Understanding the change detection execution

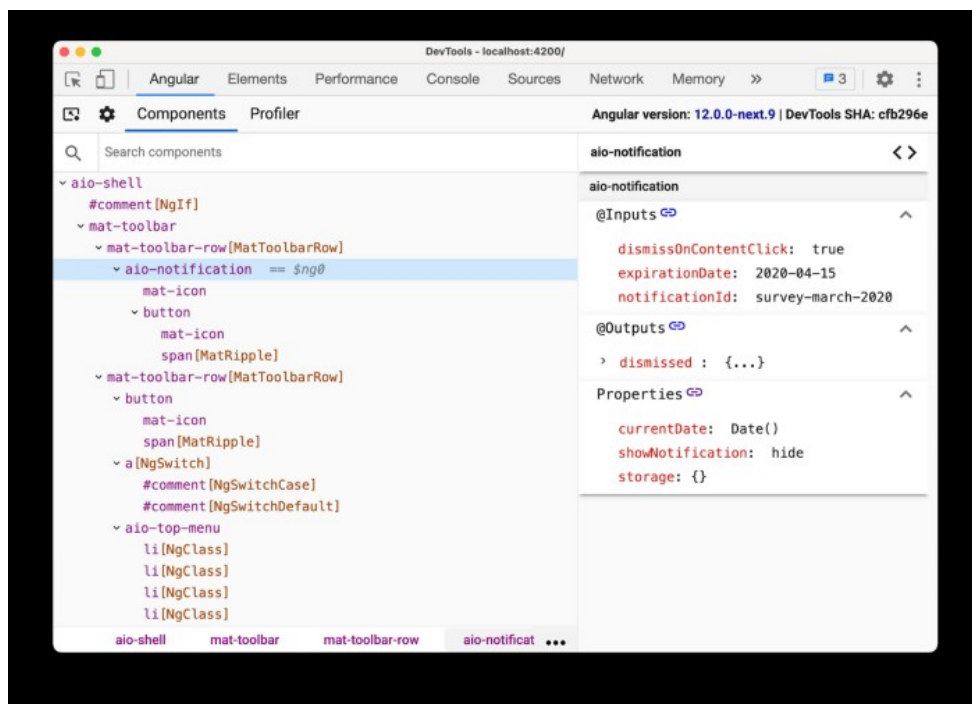
## 17.6 Using Angular DevTools

- Install Angular DevTools extension from Chrome web store (<https://chrome.google.com/webstore/>).
- Press **F12** or **CTRL+SHIFT+I** to open developer tool and use the **Angular** tab to open the tool.



## 17.7 Angular DevTools - Component Structure

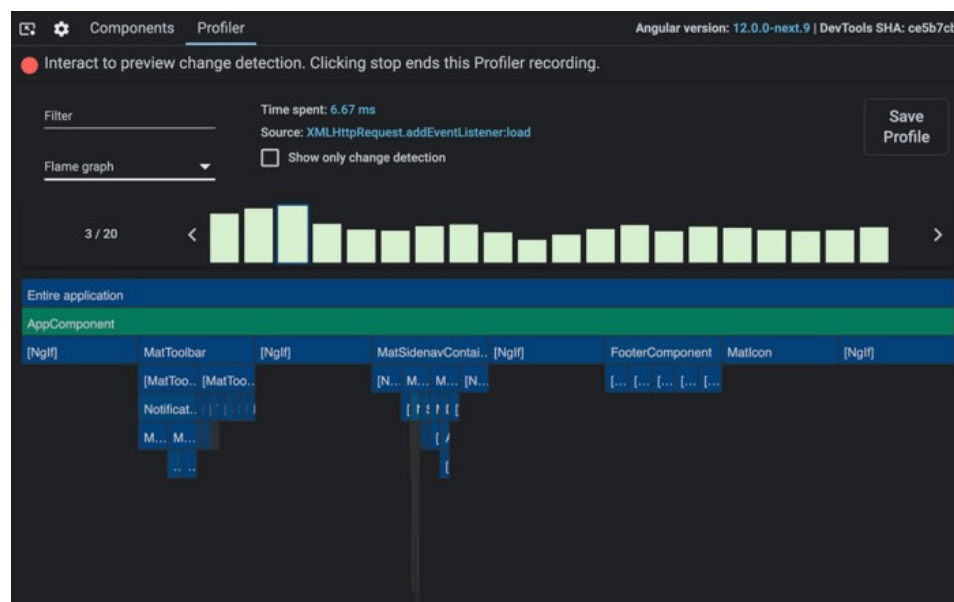
- The **Components** tab displays components' metadata properties, inputs, and outputs.
- You can also search for components in the component tree.



## 17.8 Angular DevTools - Change Detection Execution

- The **Profiler** tab lets developers preview change detection cycles as they occur in real time.

- The tab facilitates developers to identify possible performance bottlenecks.
- By selecting a bar, developers can see what triggered change detection and how much time Angular spends in the change detection phase.
- The profiler also includes a flame graph and treemap visualizations to understand the execution of change detection cycles.



## 17.9 Catching Syntax Errors

- Typescript errors are reported by the dev server.
- Template syntax errors are reported in diverse ways:
  - ◇ Some are reported by the dev server.
  - ◇ Some are reported in browser console.
  - ◇ Some are not reported, such as errors inside `{{ }}` and event names in `( )`.

## 17.10 Summary

In this chapter we covered:

- Basic Debugging Practices
- Development (Debug) Mode
- Typescript Breakpoints
- Angular DevTools