

WA2890 Comprehensive Angular 8 Programming



Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-866-206-4644
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: 1-866-206-4644 toll free, email: getinfo@webagesolutions.com

Copyright © 2019 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.
439 University Ave
Suite 820
Toronto
Ontario, M5G 1Y8

Table of Contents

Chapter 1 - Introducing Angular.....	13
1.1 What is Angular?.....	13
1.2 Central Features of the Angular Framework.....	13
1.3 Why Angular?.....	14
1.4 Building Blocks of an Angular Application.....	15
1.5 Basic Architecture of an Angular Application.....	16
1.6 Installing and Using Angular.....	16
1.7 A Basic Angular Application.....	17
1.8 Anatomy of a Basic Application.....	18
1.9 The Main Component File.....	18
1.10 The Application Module File.....	19
1.11 The index.html File.....	20
1.12 The Bootstrap File.....	21
1.13 Running the Application.....	21
1.14 Building the Application.....	22
1.15 Summary.....	23
Chapter 2 - Development Setup of Angular.....	25
2.1 Angular Development Lifecycle.....	25
2.2 Angular is Modular.....	25
2.3 What is Node.js?.....	26
2.4 Installing Node.js and NPM.....	26
2.5 Node Package Manager (NPM).....	27
2.6 Package Descriptor File - package.json.....	27
2.7 Semantic Version Numbering.....	28
2.8 Package Version Numbering Syntax.....	29
2.9 Updating Packages.....	29
2.10 Uninstalling Packages.....	30
2.11 Installing Angular Packages.....	30
2.12 Angular CLI.....	31
2.13 Creating a New Project.....	31
2.14 Angular Development Dependencies.....	32
2.15 TypeScript Typings.....	32
2.16 Testing Tools.....	33
2.17 Development Web Server.....	33
2.18 Configuring the Web Server.....	34
2.19 The Build System.....	34
2.20 Configuring the Build.....	35
2.21 Summary.....	36
Chapter 3 - Introduction to TypeScript and ES6.....	37
3.1 Programming Languages for Use with Angular.....	37
3.2 TypeScript Syntax.....	38
3.3 Programming Editors.....	38
3.4 The Type System – Defining Variables.....	39
3.5 The Type System – Defining Arrays.....	40

3.6 Type in Functions.....	40
3.7 Type Inference.....	41
3.8 The Type System – Classes & Objects.....	41
3.9 Class Constructors.....	42
3.10 Class Constructors – Alternate Form.....	43
3.11 Interfaces.....	43
3.12 Working with ES6 Modules.....	44
3.13 Visibility Control.....	45
3.14 var, let and const - defined.....	45
3.15 var, let and const - usage.....	46
3.16 Arrow Functions.....	47
3.17 Arrow Function Compact Syntax.....	47
3.18 Arrow Function and Caller Context.....	48
3.19 Template Strings.....	49
3.20 Template Strings – Variables and Expressions.....	49
3.21 Template Strings – Multiline.....	50
3.22 Generics - Class.....	50
3.23 Generics - Methods.....	51
3.24 Generics - Restricting Types.....	52
3.25 Generics - Restricting Types: Example.....	52
3.26 TypeScript Transpilation.....	53
3.27 Summary.....	54
Chapter 4 - Components in Angular	57
4.1 What is a Component?.....	57
4.2 An Example Component.....	58
4.3 The Component Class.....	58
4.4 Adding a Component to Its Module.....	58
4.5 Creating a Component Using Angular CLI.....	59
4.6 Developing a Simple Login Component.....	59
4.7 Component Template.....	60
4.8 Login Component: Add HTML.....	60
4.9 The HTML Component Template.....	61
4.10 The templateUrl property.....	62
4.11 Login Component: Add CSS Styling.....	62
4.12 Login Component: Hook Up Input Fields and Button.....	63
4.13 Login Component: Fields & Button in the Component Class.....	63
4.14 Component Decorator Properties.....	64
4.15 Component Hierarchy.....	65
4.16 The Application Root Component.....	65
4.17 Using a Regular Component.....	66
4.18 The Build System.....	66
4.19 Component Lifecycle Hooks.....	66
4.20 Using a Lifecycle Hook: OnInit.....	67
4.21 Summary.....	68
Chapter 5 - Data and Event Binding.....	69
5.1 Binding Syntax.....	69

5.2 One-Way Output Binding.....	69
5.3 Binding Displayed Output Values.....	70
String Value.....	70
Date Value.....	70
Paragraph Text.....	70
5.4 Setting Component Properties.....	71
5.5 More About Setting Properties.....	72
5.6 Setting DOM Element Properties.....	72
5.7 Event Binding.....	73
5.8 Binding Events Examples.....	74
5.9 Firing Event from a Component.....	75
5.10 @Output() Example - Child Component.....	75
5.11 @Output() Example - Parent Component.....	76
5.12 Two-Way Binding of Input Fields.....	77
5.13 Input Binding Examples.....	77
Text Input.....	77
Date Value.....	77
5.14 Two Way Binding in a Component.....	78
5.15 Use Two Way Binding.....	79
5.16 Breaking Down ngModel.....	79
5.17 Summary.....	80
Chapter 6 - Attribute Directives.....	81
6.1 What are Attribute Directives.....	81
6.2 Apply Styles by Changing CSS Classes.....	81
6.3 Changing Classes – Example.....	82
6.4 Applying Styles Directly.....	83
6.5 Applying Styles Directly - Example.....	84
6.6 Controlling Element Visibility.....	85
6.7 Setting Image Source Dynamically.....	86
6.8 Setting Hyperlink Source Dynamically.....	87
6.9 Writing a Custom Attribute Directive.....	88
6.10 Using a Custom Attribute Directive.....	88
6.11 Supplying Input to a Directive.....	89
6.12 Handling Event from a Custom Directive.....	89
6.13 Summary.....	89
Chapter 7 - Structural Directives.....	91
7.1 Structural Directives.....	91
7.2 Adding and Removing Elements Dynamically.....	91
7.3 If-Else Syntax of ngIf.....	92
7.4 Looping Using ngFor.....	93
7.5 ngFor - Basic Example.....	93
7.6 Creating Tables with ngFor.....	94
7.7 ngFor Local Variables.....	94
7.8 Manipulating the Collection.....	95
7.9 Example - Deleting an Item.....	95
7.10 Swapping Elements with ngSwitch.....	96

7.11 ngSwitch - Basic Syntax.....	97
7.12 Summary.....	97
Chapter 8 - Template Driven Forms.....	99
8.1 Template Driven Forms.....	99
8.2 Importing Forms Module.....	100
8.3 A Basic Angular Form.....	101
8.4 Binding Input Fields.....	101
8.5 Accessing the NgForm Directive.....	102
8.6 Binding the Form Submit Event.....	102
8.7 The Submit Function.....	103
8.8 Basic HTML5 Validation - "required" Attribute.....	104
8.9 HTML5 vs. Angular Validation.....	104
8.10 Angular Validators.....	105
8.11 The NgModel Directive.....	106
8.12 Controlling when validation is applied.....	106
8.13 Displaying Form Validation State.....	107
8.14 Displaying Field Validation State.....	108
8.15 Displaying Validation State Using Classes.....	108
8.16 Disabling Submit when Form is Invalid.....	109
8.17 Submitting the Form.....	110
8.18 Binding to Object Variables.....	111
8.19 Binding to Object Variables - Code	112
8.20 Additional Input Types.....	113
8.21 Checkboxes.....	114
8.22 Select (Drop Down) Fields.....	115
8.23 Rendering Options for Select (Drop Down).....	116
8.24 Date fields.....	117
8.25 Radio Buttons.....	118
8.26 Summary.....	119
Chapter 9 - Reactive Forms.....	121
9.1 Reactive Forms Overview.....	121
9.2 The Building Blocks.....	121
9.3 Import ReactiveFormsModule.....	122
9.4 Construct a Form.....	122
9.5 Design the Template.....	123
9.6 FormControl Constructor.....	123
9.7 Getting Form Values.....	123
9.8 Setting Form Values.....	124
9.9 The Synchronous Nature.....	124
9.10 Subscribing to Input Changes.....	125
9.11 Validation.....	126
9.12 Built-In Validators.....	126
9.13 Showing Validation Error.....	126
9.14 Custom Validator.....	127
9.15 Using a Custom Validator.....	128
9.16 Sub FormGroups - Component Class.....	129

9.17 Sub FormGroups - HTML Template.....	130
9.18 Why Use Sub FormGroups.....	130
9.19 Summary.....	131
Chapter 10 - Angular Modules.....	133
10.1 Why Angular Modules?.....	133
10.2 But, We Already Had ES6 Module.....	134
10.3 Angular Built-in Modules.....	134
10.4 The Root Module.....	134
10.5 Feature Modules.....	135
10.6 Create Feature Module Using CLI.....	135
10.7 The Module Class.....	136
10.8 @NgModule Properties.....	136
10.9 Using One Module From Another.....	137
10.10 Importing BrowserModule or CommonModule.....	138
10.11 Lazy-Loaded Modules.....	138
10.12 How to Organize Modules?.....	139
10.13 Third Party Modules.....	140
10.14 Summary.....	140
Chapter 11 - Services and Dependency Injection.....	141
11.1 What is a Service?.....	141
11.2 Creating a Basic Service.....	141
11.3 What is Dependency Injection?.....	142
11.4 What Dependency Injection Looks Like.....	143
11.5 Injectors.....	144
11.6 Injector Hierarchy.....	144
11.7 Register a Service with a Module Injector.....	145
11.8 Registering a Service with the Root Injector.....	146
11.9 Registering a Service with a Component's Injector.....	146
11.10 Where to Register a Service?.....	147
11.11 Dependency Injection in Other Artifacts.....	147
11.12 Providing an Alternate Implementation.....	148
11.13 Dependency Injection and @Host.....	149
11.14 Dependency Injection and @Optional.....	150
11.15 Summary.....	151
Chapter 12 - HTTP Client.....	153
12.1 The Angular HTTP Client.....	153
12.2 Using The HTTP Client - Overview.....	153
12.3 Importing HttpClientModule.....	154
12.4 Simple Example.....	155
12.5 Service Using HttpClient.....	155
12.6 ES6 Import Statements.....	156
12.7 Making a GET Request.....	156
12.8 What does an Observable Object do?.....	157
12.9 Using the Service in a Component.....	157
12.10 The PeopleService Client Component	158
12.11 Error Handling.....	158

12.12 Making a POST Request.....	159
12.13 Making a PUT Request.....	159
12.14 Making a DELETE Request.....	160
12.15 Summary.....	160
Chapter 13 - Pipes and Data Formatting.....	161
13.1 What are Pipes?.....	161
13.2 Built-In Pipes.....	161
13.3 Using Pipes in HTML Template.....	162
13.4 Chaining Pipes.....	163
13.5 Using Pipes in Code.....	163
13.6 Internationalized Pipes (i18n).....	164
13.7 Loading Locale Data.....	166
13.8 Decimal Pipe.....	166
13.9 Currency Pipe.....	168
13.10 Custom Pipes.....	169
13.11 Custom Pipe Example.....	170
13.12 Using Custom Pipes.....	170
13.13 Using a Pipe with ngFor.....	171
13.14 A Filter Pipe.....	171
13.15 A Sort Pipe.....	173
13.16 Pipe Category: Pure and Impure.....	174
13.17 Pure Pipe Example.....	175
13.18 Impure Pipe Example.....	177
13.19 Summary.....	177
Chapter 14 - Introduction to Single Page Applications.....	179
14.1 What is a Single Page Application (SPA).....	179
14.2 Traditional Web Application.....	180
14.3 SPA Workflow.....	180
14.4 Single Page Application Advantages.....	181
14.5 HTML5 History API.....	182
14.6 SPA Challenges.....	182
14.7 Implementing SPA's Using Angular.....	183
14.8 Summary.....	184
Chapter 15 - The Angular Component Router.....	185
15.1 The Component Router.....	185
15.2 View Navigation.....	186
15.3 The Angular Router API.....	186
15.4 Creating a Router Enabled Application.....	187
15.5 Hosting the Routed Component.....	188
15.6 Navigation Using Links and Buttons.....	188
15.7 Programmatic Navigation.....	189
15.8 Passing Route Parameters.....	189
15.9 Navigating with Route Parameters.....	189
15.10 Obtaining the Route Parameter Values.....	190
15.11 Retrieving the Route Parameter Synchronously.....	190
15.12 Retrieving a Route Parameter Asynchronously.....	191

15.13 Query Parameters.....	191
15.14 Supplying Query Parameters.....	192
15.15 Retrieving Query Parameters Asynchronously.....	192
15.16 Problems with Manual URL entry and Bookmarking.....	193
15.17 Summary.....	193
Chapter 16 - Advanced HTTP Client.....	195
16.1 Request Options.....	195
16.2 Returning an HttpResponse Object.....	195
16.3 Setting Request Headers.....	196
16.4 Creating New Observables.....	196
16.5 Creating a Simple Observable.....	197
16.6 The Observable.create() Method.....	197
16.7 Observable Operators.....	198
16.8 More About map.....	199
16.9 Piping Operators.....	199
16.10 The flatMap() Operator.....	200
16.11 The tap() Operator.....	200
16.12 The zip() Operator.....	200
16.13 Caching HTTP Response.....	201
16.14 Making Sequential HTTP Calls.....	202
16.15 Making Parallel Calls.....	202
16.16 Customizing Error Object with catchError().....	203
16.17 Error in Pipeline.....	203
16.18 Error Recovery.....	204
16.19 Summary.....	205
Chapter 17 - Consuming WebSockets Data in Angular.....	207
17.1 Web Sockets Overview.....	207
17.2 Web Sockets Use Cases.....	207
17.3 Web Socket URLs.....	208
17.4 Web Sockets Servers.....	208
17.5 Web Socket Client.....	209
17.6 The socket.io-client library	209
17.7 Using socket.io-client in JavaScript.....	210
17.8 Setting up socket.io-client in Angular Projects.....	210
17.9 Using socket.io-client in an Angular service.....	211
17.10 Angular websocket.service Notes:.....	212
17.11 The Angular Web Socket Client Sample App.....	212
17.12 Angular websocket.component.ts.....	212
17.13 The Full websocket.component.ts code.....	213
17.14 The Full websocket.component.ts code.....	214
17.15 The Full websocket.component.ts code.....	214
17.16 Implementation Modifications.....	215
17.17 Summary.....	215
Chapter 18 - Advanced Routing.....	217
18.1 Routing Overview.....	217
18.2 Routing Enabled Project.....	218

18.3 Routing Enabled Feature Module.....	218
18.4 Using the Feature Module.....	219
18.5 Lazy Loading the Feature Module.....	220
18.6 Creating Links for the Feature Module Components.....	220
18.7 More About Lazy Loading.....	221
18.8 routerLinkActive binding.....	221
18.9 Default Route.....	221
18.10 Wildcard Route Path.....	222
18.11 redirectTo.....	222
18.12 Child Routes.....	223
18.13 Defining Child Routes.....	223
18.14 <router-outlet> for Child Routes.....	224
18.15 Links for Child Routes.....	224
18.16 Navigation Guards.....	225
18.17 Creating Guard Implementations	225
18.18 Using Guards in a Route.....	226
18.19 Route Animations.....	226
18.20 Summary.....	227
Chapter 19 - Introduction to Testing Angular Applications.....	229
19.1 Unit Testing Angular Artifacts.....	229
19.2 Testing Tools.....	229
19.3 Testing Setup.....	230
19.4 Typical Testing Steps.....	231
19.5 Test Results.....	232
19.6 Jasmine Test Suites.....	232
19.7 Jasmine Specs (Unit Tests).....	233
19.8 Expectations (Assertions).....	233
19.9 Matchers.....	234
19.10 Examples of Using Matchers.....	235
19.11 Using the not Property.....	235
19.12 Setup and Teardown in Unit Test Suites	235
19.13 Example of beforeEach and afterEach Functions.....	236
19.14 Angular Test Module.....	236
19.15 Example Angular Test Module.....	237
19.16 Testing a Service.....	237
19.17 Injecting a Service Instance.....	238
19.18 Test a Synchronous Method.....	238
19.19 Test an Asynchronous Method.....	239
19.20 Using Mock HTTP Client.....	239
19.21 Supplying Canned Response.....	240
19.22 Testing a Component.....	240
19.23 Component Test Module.....	241
19.24 Creating a Component Instance.....	241
19.25 The ComponentFixture Class.....	242
19.26 Basic Component Tests.....	242
19.27 The DebugElement Class.....	243

19.28 Simulating User Interaction.....	243
19.29 Summary.....	244
Chapter 20 - Debugging.....	245
20.1 Overview of Angular Debugging.....	245
20.2 Viewing TypeScript Code in Debugger.....	245
20.3 Using the debugger Keyword.....	246
20.4 Inspecting Components.....	247
20.5 Saving ng.probe Component References	247
20.6 Modifying Values using Component References.....	247
20.7 Debug Logging.....	248
20.8 What is Augury?.....	248
20.9 Installing Augury.....	249
20.10 Opening Augury.....	250
20.11 Augury - Component Tree.....	250
20.12 Augury - Router Tree.....	251
20.13 Augury - NgModules Tab.....	251
20.14 Common Exceptions.....	252
20.15 Common Exceptions: 'No such file: package.json'.....	252
20.16 Common Exceptions: 'Cant bind to ngModel'.....	252
20.17 Common Exceptions: 'router-outlet not a known element'.....	253
20.18 Common Exceptions: 'No provider for Router!'.....	253
20.19 Summary.....	254

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 client side web development framework. It evolved from the AngularJS 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).
- Code artifacts have clear separation of concerns like Model, View and

Controller.

- Supports rendering the initial view in the server side for better SEO and faster load time.
- Allows you to load the entire application code at start up time or lazily load modules as the user navigates through various features of the application.
- Supports unit testing through Karma.
- Provides Angular CLI tooling to quickly create code artifacts.
- 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 Why Angular?

- Web apps today have complex requirements
 - ◇ Retrieve data from the backend server and display it to the users
 - ◇ Collect data from users, validate the input and then submit it to the backend server.
 - ◇ Navigate between views without reloading the entire DOM.
- A lot of JavaScript code is needed to support these requirements. This is where Angular comes in. It lets developers rapidly build modern web applications in a highly maintainable modular fashion.

Notes

Angular was developed to solve some basic problems in existing web development. The HTML markup language was originally designed to display static web pages and to move from one page to

another using simple links. As individual pages grew larger and more complex they took longer to load into the browser. Simple web sites evolved into web applications displaying richly designed pages with dynamic content. Forms enabled data capture but often involved multi-page work flows and complex validation code. Although various JavaScript libraries emerged to assist web developers the amount of code and complexity of applications continued to increase.

Angular was created to simplify and reduce the coding required to support data management, forms validation and handling, network requests, navigation and other web development tasks. In addition, the way it's structured encourages modularity, code reuse, and testing.

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.

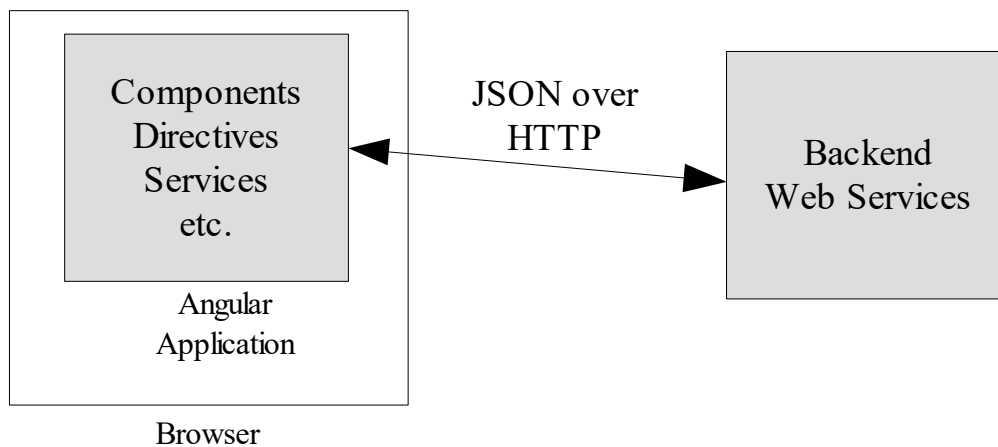
Notes

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 Nginx.

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 is made available as a collection of Nodejs NPM packages.
- The Angular CLI tool makes it easy to create a project and install all the NPM packages.
- First, install Angular Command Line interface (CLI).


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

- Then create a new Angular application project.

```
ng new register-app
```

- Dependencies in `package.json` file:

```
"dependencies": {  
  "@angular/common": "...",  
  "@angular/compiler": "...",  
  "@angular/core": "...",  
  "@angular/forms": "...",  
  ...  
}
```

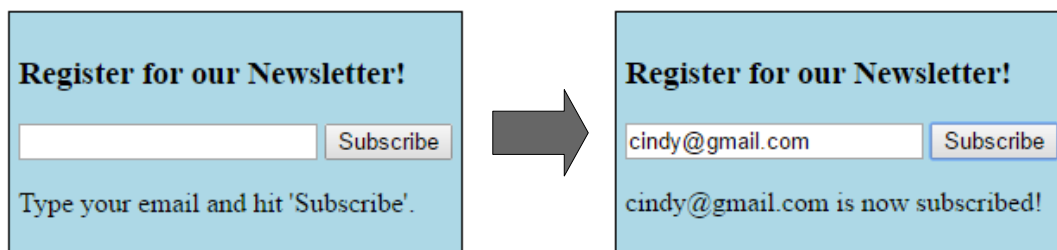
Notes

Node Package Manager (NPM) is used to install Angular and other required libraries in a development machine. The process is made easier by the use of the Angular CLI. When you create a new project using Angular CLI it will generate the `package.json` file for you and install the necessary packages using NPM.

Important: NodeJS and NPM are used in development and build machines only. You do not have to install NodeJS in QA or production where an Angular application is deployed.

1.7 A Basic Angular Application

- A basic Angular Application:



- The Angular application is made up of the following files:

<code>app.component.ts</code>	The main component for the application
-------------------------------	----------------------------------------

<code>index.html</code>	The HTML file that displays the main component
<code>app.module.ts</code>	Angular module for the application
<code>main.ts</code>	Bootstraps or attaches the application module into <code>index.html</code>

Notes

The application shown above consists of an input field, button and output message field. The user enters their email address in the input box and clicks the "subscribe" button. A message is created and displayed in the space below. We'll take a look at the individual files that make up the app over the next few pages.

1.8 Anatomy of a Basic Application

- All the artifacts of an application (components, services etc.) must be packaged inside a single application module.
 - ◇ For more complex applications this application module may depend on other sub-modules. But there is always one main application module.
 - ◇ Angular CLI generates the file **app/app.module.ts**.
- Each application has one main component. This renders the initial view of the application. Angular CLI creates it in **app/app.component.ts** file.
- The application component is rendered inside the DOM created by **index.html**. This static HTML file usually contains the common page elements like header, footer, and sidebar. As the user navigates through the app, Angular keeps switching the component that is displayed on the screen. But the common elements of the page remain the same.
- The act of loading the application module and displaying its main component inside of `index.html` is called bootstrapping. This is done by the **main.ts** file.

1.9 The Main Component File

app.component.ts

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

```
@Component({
  selector: 'app-root',
  template: `<h3>Register for our Newsletter!</h3>
<input type="text" [(ngModel)]="email"/>
<button (click)="registerEmail()">Subscribe</button>
<p>{{message}}</p>`
})
export class AppComponent {
  email: string
  message = "Enter email and hit 'Subscribe'.";

  registerEmail() {
    this.message = this.email + " is now subscribed!";
  }
}
```

Notes

The code above shows the application's main component class AppComponent. The @Component decorator is used for the class to declare various metadata.

1. The selector is "app-root". Which means this component will be rendered where <app-root> element is encountered in DOM.
2. The template is the HTML for the component. Here we are binding the text box with the email field of the class. The click event of the button is bound to the registerEmail() method of the class.

1.10 The Application Module File

app.module.ts

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

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

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

Notes

Every application has one main module and may have a few sub-modules. Angular CLI generates the module class and calls it `AppModule` by default. Angular modules are very helpful as we will learn later. For now just inspect a few basic features.

A component is made a member of a module by adding it to the "declared" list. We have added `AppComponent` there. The `AppComponent` is also special in the sense it is the main component and renders the initial view of the application. As a result we added it to the "bootstrap" list.

Our module has imported a few other Angular modules. Artifacts defined in an imported module become available for use throughout the module that does the import. For example, the `FormsModule` defines the `ngModel` directive. We can now use that directive in the template of `AppComponent`.

1.11 The index.html File

- Defines the DOM within which components are displayed. Specifically it needs to contain the element for the application's main component.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>RegisterApp</title>
  <base href="/">
  <meta name="viewport" content="width=device-width,
initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Notes

Note that we do not explicitly import the JavaScript for our application code or for Angular. That is done at build time by Angular CLI.

1.12 The Bootstrap File

main.ts

This file attaches (bootstraps) the application module into index.html.

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

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

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

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.

1.13 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 the body.

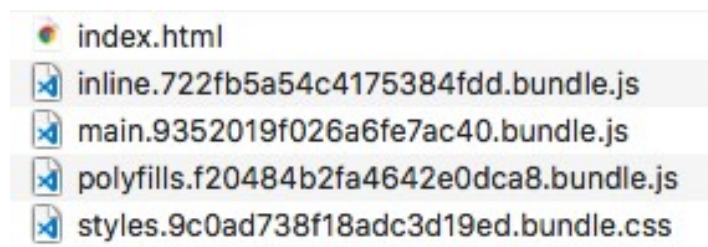
```
<body>
  <app-root></app-root>
  <script type="text/javascript" src="inline.bundle.js">
</script>
  <script type="text/javascript" src="polyfills.bundle.js">
</script>
  ...
</body>
```

1.14 Building 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 --prod
```

- This will output a modified `index.html` and a small number of highly efficient and compact JavaScript files.



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

1.15 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 - Development Setup of Angular

Objectives

Key objectives of this chapter

- Understand how an Angular application project is created
- Learn how to install the packages your application depends on
- Gain expertise in Node Package Manager
- Learn how to use the development web server
- Learn how to perform a build for deployment to QA or production

2.1 Angular Development Lifecycle

- Development with Angular involves these steps:
 - ◇ Install Angular and dependent packages
 - ◇ Create and edit Angular code like components and services
 - ◇ Run unit tests
 - ◇ Build and serve compiled application files from a development web server
 - ◇ Test and debug the app in a browser
- Moving a developed app to production typically involves:
 - ◇ Build your code in production mode. This produces highly efficient and minified files
 - ◇ Move files to a production web server

2.2 Angular is Modular

- Angular code is distributed as a collection of Node.js packages.

```
@angular/common  
@angular/core  
@angular/forms  
@angular/http
```

```
@angular\platform-browser  
@angular\router  
etc.
```

- For Angular development, these modules should be installed locally in the development machine. **Important:** Node.js or these packages do not have to be installed in a QA or production server machine.
- Node.js has an excellent package manager called NPM which makes installing, uninstalling and upgrading the Angular packages easy.
- This modularity lets us pick and choose only the packages we need. The build tooling provided by Angular CLI performs static analysis of your application code to only include the packages that are actually in use. This leads to fast application startup.

2.3 What is Node.js?

- Node.js is an application development platform
- Node applications:
 - ◇ Are written in JavaScript or a higher level language like TypeScript
 - ◇ Are run from a command prompt and not in a browser
- The Node environment:
 - ◇ Is event driven
 - ◇ Is single threaded
 - ◇ Is non-blocking
 - ◇ Follows an asynchronous programming paradigm
- Many code libraries (packages) are available for Node development
- Node Package Manager (NPM) is used to install packages and manage dependencies for Node based applications
- More information is available at: <https://nodejs.org>

2.4 Installing Node.js and NPM

- Node and NPM need to be installed in development and build machines

- Windows and Mac installer packages can be downloaded from nodejs.org.
- NPM is installed along with the Node.js installation
- After installation check that node and npm are working:
 - ◇ Open a command prompt to any directory.
 - ◇ Check Node:

```
node --version
```
 - ◇ Check NPM:

```
npm --version
```

2.5 Node Package Manager (NPM)

- Code libraries, called packages, are installed with the npm package mgr.
- NPM uses simple commands like the following to install packages from a central repository on the web maintained by node.org:

```
npm install jquery  
npm install -g gulp
```
- The **-g** parameter installs the specified package in a central location on the development machine. It is typically used to install large shared code libraries or node applications that include command line interfaces.
- When the -g parameter is not used packages are installed in a local sub-directory named **node_modules**
- When npm install is run without a package name it looks for a file named **package.json** file in the local directory that includes the required information.
- Using the package.json file multiple libraries can be installed at once

2.6 Package Descriptor File - package.json

- The package.json file describes a Node.js package including the names and versions of packages it depends on.
- It can be generated using:

`npm init`

- Add external packages you depend on in the ***dependencies*** section:

```
"dependencies": {  
  "colors": "1.1.2",  
  "lodash": "4.17.3"  
},
```
- The package.json containing the above dependencies section is used to install two packages at once, the ***colors*** package and the ***lodash*** package.
- Notice how the required version number is supplied for each package.

2.7 Semantic Version Numbering

- Node Package Manager makes use of semantic version numbering.
- Semantic version numbers let you specify the exact major, minor and patch releases for a package
- Take for example the following package dependency:

```
"lodash": "4.17.3"
```
- Here the major release number is 4, the minor release is 17 and the patch release number is 3.
- Release numbers are changed for specific reasons:
 - ◇ **Major** release number are changed when a release includes "breaking" changes.
 - ◇ **Minor** release numbers are changed when new features are added while backward compatibility with earlier versions is maintained
 - ◇ **Patch** release numbers are changed when a new version includes mostly bug fixes while maintaining backward compatibility with earlier versions
- More information about semantic version numbering is available at: semver.org

2.8 Package Version Numbering Syntax

- When entering a package version number in the package.json file you can request a specific version or allow NPM to return the latest major, minor or patch release:

What you need	How to specify (example)
Exact version	2.1.5
Latest patch release	2.1 2.1.x ~2.1.0
Latest minor release	2 2.x ^2.0.0
Latest major release	*

2.9 Updating Packages

- As newer package versions are released previously downloaded versions can become obsolete.
- Use the **outdated** command to check if any packages have been updated since they were installed:

```
npm outdated
```

- Packages defined with an exact version number in package.json are not included in this check.
- Running the following command will bring all packages up to the latest desired version as specified in package.json:

```
npm update
```

- Updating to the latest version of a package can in some cases break your application. For this reason the update command should be used with caution.

2.10 Uninstalling Packages

- Packages no longer being used can be uninstalled using the following commands. Note though that this does not update package.json:

```
npm uninstall package_name
```
- If you are using a package.json file and wish to uninstall a package you should:
 - ◇ Edit the package.json and remove the entry for the unused package.
 - ◇ Then running **npm prune** will remove the package from the node_modules directory
- Alternately you can uninstall a package and update the package.json at the same time using this command:

```
npm uninstall package_name --save
```

- Globally installed packages can be removed using this command:

```
npm uninstall package_name -g
```

2.11 Installing Angular Packages

- Use Angular CLI to create a new project. It will:
 - Create the project directory
 - Add basic project files
 - Create a "package.json" file and add common Angular, Karma, Jasmine and other dependencies to it.
 - Run npm install to download the packages into the node_modules directory
- As your project matures you may need additional packages. Simply add them to your package.json and install them using NPM.
- The Angular Getting Started page has more details on using Angular CLI to create a new project:

```
https://angular.io/guide/quickstart
```

2.12 Angular CLI

- Angular CLI:
 - ◇ Is a Node.js command line utility for creating new Angular projects
 - ◇ Can be installed using Node Package Manager (npm)
 - ◇ Can generate template code for: Components, Modules, Services, etc.
 - ◇ Includes utilities for compiling and packaging Angular applications. Internally it uses Webpack
 - ◇ Includes a development web server
 - ◇ Integrates unit tests and end-to-end testing out of the box
- For more information see:
 - <https://cli.angular.io/>
 - <https://github.com/angular/angular-cli/wiki>

2.13 Creating a New Project

- Basic command to create a new project using Angular CLI.

```
ng new your-app-name
```

- You can supply these optional flags to customize the new project
 - ◇ `--skip-install` - Do not run npm install. Just create package.json. You must install the packages later before doing any kind of development.
 - ◇ `--routing true` - Generates a routing module. This saves a bit of work if you know you are building a single page app and will need routing.
 - ◇ `--minimal` - Creates a simpler project that lacks a few features like unit testing. You can use it for proof of concept projects.
 - ◇ `--skip-git false` - Initializes git for the project and also creates a very useful .gitignore file.
- Example:

```
ng new my-app --routing true --skip-git false
```

2.14 Angular Development Dependencies

- A few additional Node.js packages are needed that provide various development tooling. Example:
 - ◇ Angular CLI packages
 - ◇ Unit testing packages like Karma and Jasmine
 - ◇ Typescript compiler
- They are added as **devDependencies** to package.json. Example:

```
"devDependencies": {  
  "@angular/cli": "...",  
  "@angular/compiler-cli": "...",  
  "@angular/language-service": "...",  
  "typescript": "...",  
  ...  
}
```

- Angular CLI automatically adds the most common development dependencies at project creation time.

2.15 TypeScript Typings

- TypeScript is a typed language but many packages are written in JavaScript that has no type information. For example, the Jasmine and Karma unit test frameworks are written in JavaScript.
- TypeScript provides a way to supply type information for these libraries through **Typings** files.
- Typings provide extra information not included in standard JS libraries like:
 - ◇ interface and class definitions
 - ◇ function parameter and return types
- Typings are used to:
 - ◇ Provide code completion and documentation in programming editors
 - ◇ Verify correct usage of functions during TypeScript compilation
- Typings are added to devDependencies in package.json. Angular CLI

adds some of them for you:

```
"@types/jasmine": "~2.5.53",  
"@types/jasminewd2": "~2.0.2",  
"@types/node": "~6.0.60",
```

2.16 Testing Tools

- Various testing frameworks/tools designed for use with JavaScript web applications can also be used when developing Angular Applications
- The following testing tools are all Node.js based applications:
 - ◇ Jasmine: A JavaScript unit testing framework for writing tests.
 - ◇ Karma: A test runner for unit testing.
 - ◇ Protractor: An end-to-end testing framework that lets you run UI based tests in various browsers
- These tools are added to devDependencies of package.json. Angular CLI already adds pretty much what you need.

2.17 Development Web Server

- Angular CLI provides a web server that should be used during development. It performs two main tasks:
 - ◇ Constantly watches your codebase and performs a re-build when you modify a file.
 - ◇ Automatically refreshes the browser once the build finishes.
- Start the server by running this from the project's root folder:

`ng serve`

- Access the application using **http://localhost:4200/**
- After you make any changes to a file watch for build errors in these places:
 - ◇ The server command line console
 - ◇ The web browser's development console. Always have it open during

development

- ◇ The web page itself sometimes shows errors

Development Web Server

The server often refreshes the web page even when the build has errors. There may not be any obvious signs on the page that something has gone wrong. Always watch the server's command line output as well as the browser's development console.

2.18 Configuring the Web Server

- To enable SSL run this command and use the URL `https://localhost:4200/`.

```
ng serve --ssl
```

- Live reload uses web sockets. If this becomes a problem for some reason disable it:

```
ng serve --live-reload false
```

- Most real life applications will make web service calls. These web services can run on other servers such as IIS or Tomcat. You just need to setup a reverse proxy in the Angular server to satisfy the same origin policy.

2.19 The Build System

- Before your application can be deployed to QA or production it must be built using Angular CLI. The basic command is:

```
ng build --prod
```

- It does a few things:
 - ◇ Compiles the TypeScript code into JavaScript
 - ◇ Processes the decorators like `@Component` and converts them to JavaScript code

- ◇ Compiles any external templates into JavaScript
- ◇ Compiles all stylesheets used by the components into JavaScript
- ◇ Performs static analysis of your application code and determines the Node.js packages it actually uses. Bundles all of those packages in a single JavaScript file.
- It outputs a small number of JavaScript files and a modified index.html (that includes the JS files using `<script>` tag) and saves them in the **dist** folder. To deploy the app simply copy these files to the document root of a web server.
- Most of the heavy lifting during build is done by a software called Webpack (<https://webpack.js.org>).

The Build System

Angular CLI very nicely isolates us from the complexities of configuring Webpack and you should never have to directly work with Webpack.

Even though a build can be done from a development machine most companies will dedicate a separate build machine. Node.js, Angular CLI and all the packages that your project depends on needs to be installed in that machine. So the basic steps will be as follows:

1. First checkout a branch from git
2. Run `npm install` to install dependencies
3. And then run `ng build`.

2.20 Configuring the Build

- Always use the `--prod` option to get the most optimized code.
- Use a custom output folder (default is `dist`):

```
--output-path=my-output
```

- If you will copy the output files to a sub folder of the document root then you need to supply the folder name as the base host name. For example, if you will copy the files in a folder called `customer-survey` under the document root:

```
--base-href=/customer-survey/
```

- Normally the generated JavaScript files are copied to the same folder as index.html. If you must copy the JS files in another place like a CDN then supply the URL prefix:

```
--deploy-url=http://cdn.example.com/my-app/script
```

2.21 Summary

In this chapter we covered:

- Angular code is installed as a collection of Node.js packages
- Node.js packages are installed using Node Package Manager (NPM).
- You can define all packages your application depends on in package.json.
- Angular CLI can create a new Angular project for you with a package.json that has almost everything you need to get started.
- Angular CLI supplies us with a development time web server which can rapidly build your code after every change and refresh the browser.
- Your application code has to be built using ng build before it can be deployed to QA or production.

Chapter 3 - Introduction to TypeScript and ES6

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

3.1 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 superset 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 (ES5) for use in browsers
 - ◇ Includes full type support
 - ◇ Includes full support for Object Orientation
 - ◇ Angular itself is written in TypeScript

Notes

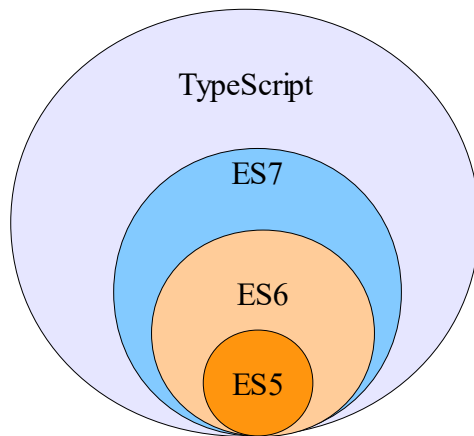
Although JavaScript has been evolving even the latest version, ES6, does not include some features you might expect such as strongly typed variables and full object orientation.

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.

3.2 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.

3.3 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.
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.

3.4 The Type System – Defining Variables

- Standard JavaScript variables are untyped:

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

x = y // OK. Allowed.
```

- TypeScript variables are strongly typed:

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

x = y // Compile error
var 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
```

3.5 The Type System – Defining Arrays

- Arrays in standard JavaScript

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

- Arrays in TypeScript

```
var colors: string[] = ['red', 'white', 'blue'];
var my_nums: number[] = {10, 20, 30};
var 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_nums: number[] = {10,20,30};
```

This array can hold various types of objects:

```
var names: Object[];
```

This array can only hold Person type objects:

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

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

3.6 Type in Functions

- Parameter and return types need to be specified.

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



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

3.7 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.

3.8 The Type System – Classes & Objects

- Classes define a type that can be used with variables and arrays

```
class Cat{
    name: string;
    type: string;
}

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

```
var cat1: Cat = {name:"fluffy", type:"tabby"};
var cat2: Cat = {name:"ginger", type:"siamese"};
var cat3: Cat = new Cat

var cats: Cat[] = [cat1, cat2, cat3]

cats.forEach(cat => meow(cat))
```

Notes:

The TypeScript compiler will throw errors if the object literals do not match the class definition for cat. To compile properly they must have both properties and no other properties and the type of the properties in the object literal must be string. For example errors would be generated for the following lines of code:

```
var cat1:Cat = {name:"fluffy", breed:"tabby"}; // no "type" prop
var cat2:Cat = {name:"ginger", weight:12};      // no "type" prop
```

Note that only a Cat type object can be added to the cats array and only object variables with the correct properties can be of the Cat type.

3.9 Class Constructors

- Classes can have constructors:

```
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];
```

3.10 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 these to the constructor also defines them as class properties.

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

- The keyword "public" is used to define properties that are accessible from outside the class.
- The keyword "private" is also valid. Properties defined with "private" can only be accessed inside the class's methods.

Notes

The syntax shown here is commonly used when defining classes.

3.11 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: ITrip = 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: ITrip = new Vacation("Hawaii", 5);

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

showTrip(vacation);
```

3.12 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'
```

3.13 Visibility Control

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

```
class Employee {  
    name:string //Public by default  
    private salary:number  
    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.

3.14 var, let and const - defined

- **var**, **let** and **const** are used to define variables
 - ◇ **var** has always been part of JavaScript
 - ◇ **let** and **const** were introduced in ES6 and are supported in TypeScript
- Variables declared with **var**:

- ◇ Are global when defined outside a function
- ◇ Are local to a function when defined inside the function
- ◇ Are local to a function when defined in a code block inside the function
- ◇ Can be accessed before they are declared (due to hoisting)
- **let** is used to declare variables that are limited in scope
 - ◇ When used inside a function **let** works the same as **var**
 - ◇ When used inside a code block **let** defines a variable that is local to that block. (i.e. condition, loop and sub-function code blocks)
- **const** is used to define variables whose values are only assigned once.
 - ◇ Be careful. If your **const** points to an object the object's properties **can** be assigned more than once!

Notes:

Both var and let variables are hoisted to the top of their scope before execution but vars are also initialized to undefined whereas lets are not which can cause reference errors.

3.15 var, let and const - usage

- *Use let for loop variables, conditional blocks and anonymous functions:*

```
for(let i =0; i < 5;i++){  
    console.log(i);  
}  
  
if(a === b){  
    let x = 5;  
    ...  
}  
  
element.onclick = function(){  
    let someVal = "some text";  
    ...  
}
```

- *Use const for primitive data types (string, number, boolean)*

```
const PI: number = 3.14159;  
const URL: string = "http://company.com/products";
```

Note: Names for constant references should be all upper case.

3.16 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:

```
function testSquare(square:(n: number) => number) {  
    let result = square(10) //Invoke arrow function  
  
    console.log(result) //Prints 100  
}
```

```
testSquare((n: number) : number => {  
    return n * n  
})
```

```
testSquare((n: number) : number => {  
    return Math.pow(n, 2)  
})
```

Notes

In the example above the testSquare() function takes as argument an arrow function. The arrow function should compute the square of a number. We call the testSquare() function with two different implementations of squaring functions. These implementations are arrow functions shown in bold face.

3.17 Arrow Function Compact Syntax

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

```
testSquare((n) => {  
    return Math.pow(n, 2)  
})
```

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

```
testSquare((n) => n * n)
```

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

```
testSquare(n => n * n)
```

Notes

Since Typescript knows what type of arrow function `testSquare()` takes as input you don't have to explicitly state them when writing the arrow function.

3.18 Arrow Function and Caller Context

- The **this** keyword within an arrow function refers to the caller of the function.

```
class StorageCabinet {
  myStuff = ["PS3", "Laptop", "TV", "Cooler"]
  sortAscending:boolean = true

  organize() {
    this.myStuff = this.myStuff.sort((item1, item2) => {
      if (item1 === item2) return 0
      if (item1 > item2) return this.sortAscending ? 1 : -1
      if (item1 < item2) return this.sortAscending ? -1 : 1
    })
  }
}

let cab = new StorageCabinet
cab.organize()
```


Notes

A key difference between ES6 arrow function and a regular JavaScript function is that an arrow function preserves the context of the caller. In the example above the `organize()` method is called in the context of an instance of the `StorageCabinete` class. The `organize()` method then calls the `Array.sort()` function. We supply an arrow function to the `Array.sort()` function. The **this** keyword within the arrow function will point to the original caller context which is the instance of the `StorageCabinete` class.

A regular JavaScript function on the other hand define **this** as the function object itself. This has lead to a lot of confusion and error when defining higher order functions in JavaScript. Fortunately, arrow functions solve that problem.

3.19 Template Strings

- Template Strings is a new ES6 syntax also available in TypeScript
- Uses the back-tick character [`] rather than single or double quotes to define string literals.
- Back-tick is the key to the left of number 1 on keyboard
- Added Features:
 - ◇ Include string substitution and expressions with `${}`
 - ◇ Create multiline strings

Notes

Template strings simplify the creation of string literals. They allow expressions and variables to be substituted inline. They also make Angular HTML templates and CSS style properties which tend to require multiple lines easier to work with.

3.20 Template Strings – Variables and Expressions

- Standard JavaScript String

```
var name = 'Harold';  
var x = 'Hello There ' + name;
```

- Template String Equivalent

```
var name = `Harold`;  
var x = `Hello There ${name}`;
```

- Embedding an expression

```
var x = `Cindy got ${2 * 3} calls today!`;
```

- Embedding a function

```
function getName(){return "Harold";}
var x = `Hello There ${getName()}`;
```

3.21 Template Strings – Multiline

- HTML Snippet as JavaScript String

```
var html = 'Name:'
+ '<input type="text" [(ngModel)]="name"/><br>'
+ 'Password:'
+ '<input type="password" '
+ '[(ngModel)]="password"/><br>'
+ '<button (click)="submit()">Submit</button>';
```

- Template String Equivalent

```
var html = `Name:
<input type="text" [(ngModel)]="name"/><br>
Password:
<input type="password"
[(ngModel)]="password"/><br>
<button (click)="submit()">Submit</button>`;
```

3.22 Generics - Class

TypeScript includes several features related to something called 'Generics':

- Generics allow type specification to be deferred from when a class is created to when the code using the class is written.
- A letter inside of angle brackets indicates a generic type:

```
ArrayMaker<T>
```

- An actual type needs to be supplied when the class is used:

```
var maker: ArrayMaker<Cat> = new ArrayMaker<Cat>();
```

- Specifying the type determines how the class behaves.
- The type associated with `ArrayMaker` determines the type of array emitted by its `createArray` method:

```
var cats: Cat[] = maker.makeArray();
```

- To create an array of `Dogs` we simply change the type:

```
var maker: ArrayMaker<Dog> = new ArrayMaker<Dog>();  
var dogs: Dog[] = maker.makeArray();
```

Notes:

The term 'generic' means "relating to a class or group of things - not specific"

`ArrayMaker` is a factory class that creates test data of various types.

Although 'T' is used in many cases it is not required. The letter in angle brackets can be anything (e.g. `ArrayMaker<A>`).

If needed multiple generic types can be specified:

```
var af: ArrayFactory<A,B> = new ArrayFactory<A,B>();
```

The examples above assume that we have already defined `Cat` and `Dog` classes.

3.23 Generics - Methods

- Generic types can also be used with methods:

```
function getData<T>():T{  
    var data: T;  
    // assign value to data  
    return data;  
}
```

- The generic type in the function above determines its return type:

```
var cat: Cat = getData<Cat>();
```

- In this example two generic types are used:

```
function getData<I,O>(param: I):O{  
    var data: O;  
    // use param to lookup data  
    return data;  
}
```

- ```
var order:Order = getData<string, Order>("1025");
```
- The first type determines the function's input parameter the second determines its return type.

### Notes:

Generics allow us to create functions and methods that work with multiple data types while maintaining the ability to type check at compile time.

## 3.24 Generics - Restricting Types

- Generics let you plug in any type into classes and functions.
- This may not always be what you need.
- Consider this method. It is meant to calculate an employee's weekly pay based on their salary:

```
function pay<T>(payee: T):number{

 return payee.salary/52;
};

var amount: number = pay<string>("joe");
```

- The syntax of the `pay()` function allows you to pass it any type including a string type.
- But `pay()` needs to return a number and the string type does not contain enough information to provide an amount.
- We need to restrict the input parameter to types that can provide a salary.

## 3.25 Generics - Restricting Types: Example

- We can fix the `pay()` function by only allowing types that include the information we need:

```
function pay<T extends IPayable>(payee: T) {
 return payee.salary/52;
}
```

```
};
```

- This restricts the function to accepting classes that implement the IPayable interface:

```
interface IPayable{ salary: number; };
```

- The Employee class implements IPayable:

```
class Employee implements IPayable{
 name:string;
 salary: number
};
```

- Passing an instance of Employee gives us the information we need to calculate the pay amount:

```
var emp: Employee = new Employee("joe",75000);
var amount: number = pay<Employee>(emp);
```

## Notes:

Full example code for restricting generic types:

```
interface IPayable{ salary: number; };

function pay<T extends IPayable>(payee: T){
 return payee.salary/52;
};

class Employee implements IPayable{
 constructor(public name:string, public salary: number){}
};

var emp: Employee = new Employee("joe",75000);

var amount: number = pay<Employee>(emp);

console.log("pay " + emp.name + " amount " + amount);
```

## 3.26 TypeScript Transpilation

- Typescript code must be transpiled to JavaScript
- TranScript compiler 'tsc' is installed via the **typescript** NPM package
- Is run from the command line as. Can be setup to watch code changes and run automatically.

- Will use a `tsconfig.json` project file if available
- Editor TypeScript plugins can report compile errors as you type and compile files when you save a file.

## Notes

Browsers understand JavaScript not Typescript. TypeScript needs to be converted into JavaScript before a browser can use it. This conversion step is sometimes referred to as 'transpilation'.

The NodeJS based command line tool that performs transpilation is called 'tsc' and can be installed via the node package manager.

```
npm install -g typescript
```

Individual files can be processed using command line arguments:

```
tsc app.ts
```

produces an 'app.js' file

The root directory of projects that include typescript should include a `tsconfig.json` configuration file. The config file includes information about which files to convert and which ones to leave alone. 'tsc' will use the config file when its run by itself with no command line arguments:

```
tsc
```

Uses `tsconfig.json`

When 'tsc' is started in 'watch' mode it will monitor TypeScript files and perform a conversion whenever a file is saved. 'tsc' will watch until you enter Ctrl-C or close the terminal window.

```
tsc app.ts -w
```

- for individual files

```
tsc -w
```

- when `tsconfig.json` exists

Plugins exist for some programming text editors that let them call the TypeScript transpiler whenever a file is saved. Some examples are: MS Visual Studio, MS Visual Studio Code, Atom, Brackets. In some cases support is built in. In others you will need to install a corresponding TypeScript plugin.

## 3.27 Summary

### In this chapter we covered:

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

- ◇ Working with Modules
- ◇ Transpilation of TypeScript
- ◇ Arrow Functions & Template Strings
- ◇ Generics





## Chapter 4 - Components in Angular

---

### *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

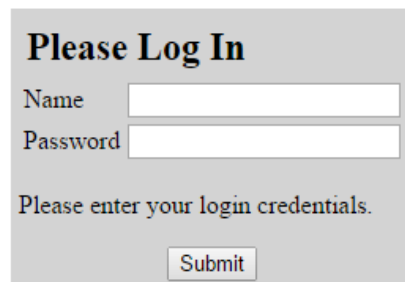
### 4.1 What is a Component?

Components:

- Are TypeScript classes used to write the GUI display logic.
- They render display using:
  - ◇ HTML Templates
  - ◇ CSS Styles
- They accept user input using data binding.

Components Can:

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



**Please Log In**

Name

Password

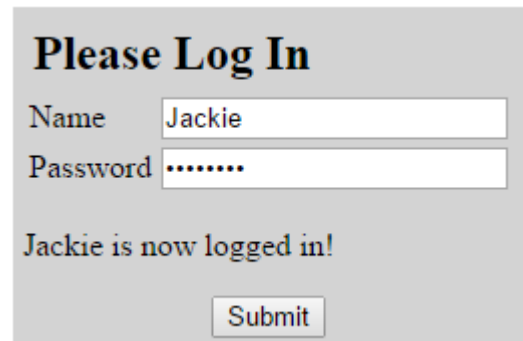
Please enter your login credentials.

A Login Component

## 4.2 An Example Component

The Login Component Includes:

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



Login Component

### Notes

To use the component, we add the following to our HTML file:

```
<body>
 <login>Loading...</login>
</body>
```

## 4.3 The Component Class

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

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

@Component({
 selector: 'app-simple',
 template: `<p>Hello World!</p>`
})
export class SimpleComponent {

}
```

## 4.4 Adding a Component to Its Module

- Every component class must be added to the **declarations** list of its module.

```
import { SimpleComponent } from
 './simple/simple.component';

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

## 4.5 Creating a Component Using Angular CLI

- Basic command.

```
ng generate component simple
```

- This will create the component in src/app/simple/simple.component.ts.
- This will also add the component to the module's declarations list.
- If testing is enabled for the project a unit test script will also be created.
- Optional arguments:
  - ◇ --inline-template false - Create a separate HTML template file.
  - ◇ --inline-style false - Create a separate CSS file for styles.

## 4.6 Developing a Simple Login Component

- Generate it:

```
ng generate component login
```

- It will look like this:

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

@Component({
```

```
 selector: 'app-login',
 template: `<p>login works!</p>`,
 styles: []
 })
export class LoginComponent implements OnInit {
 constructor() { }
 ngOnInit() {}
}
```

## Notes

The component will be created in `src/app/login/login.component.ts` file.

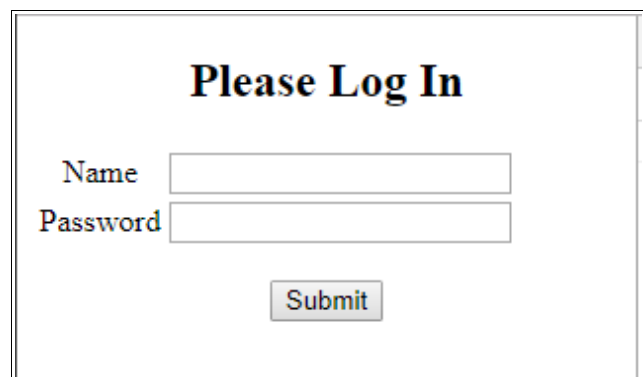
## 4.7 Component Template

- A component uses its template to render display.
- For web applications, the template will output HTML DOM elements. For native mobile applications, the template can be something else like React Native.
- Templates look like plain HTML. This makes it easy to convert HTML delivered by a designer into component templates. During build, a template gets compiled into JavaScript code.

## 4.8 Login Component: Add HTML

- HTML code for the form and input elements replaces the contents of the `@Component` `template` property.

```
 template: `<h2>Please
 Log In</h2> ...`,
```
- The component still lacks styling and we have not yet hooked up the fields and buttons.



The image shows a simple login form. At the top, it says "Please Log In" in a bold, black font. Below this, there are two input fields. The first is labeled "Name" and the second is labeled "Password". Both labels are in a blue font. Below the input fields is a button labeled "Submit" in a black font. The entire form is enclosed in a thin black border.

## Notes

The component section now looks like this.

```
@Component({
 selector: 'app-login',
 template: `
 <h2>Please Log In</h2>
 <p>Name:

 <input type="text" [(ngModel)]="name"/></p>
 <p>Password:

 <input type="password" [(ngModel)]="password"/></p>
 <p>{{message}}</p>
 <button (click)="submit()">Submit</button>
 `,
 styles: [],
})
```

## 4.9 The HTML Component Template

- Lets look at the HTML template in more detail. The code has been rearranged here to make it easier to read.

```
template: `
 <h2>Please Log In</h2>
 <p>Name:

 <input type="text" [(ngModel)]="name"/></p>
 <p>Password:

 <input type="password" [(ngModel)]="password"/></p>
 <p>{{message}}</p>
 <button (click)="submit()">Submit</button>
`,
```

## Notes

Some of what we see here is typical HTML code and some is not.

The input fields look normal except for a few attributes like this:

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

ngModel is an attribute specific to Angular. The way it is written here tells Angular to connect the input and output value of the input element to a variable called 'name' in the corresponding LoginComponent class. A similar attribute statement in the other input field links that field to a variable called 'password' in the LoginComponent class.

{{message}} outputs the value of the 'message' variable from the LoginComponent to the view:

```
{{message}}
```

And finally the following attribute in the button element links the element's click event to the "submit" function in the LoginComponent class:

```
(click)="submit() "
```

## 4.10 The templateUrl property

- Placing HTML in the `template` property is OK for smaller components.
- For more complex components HTML code can be moved to a separate file and referenced with the `templateUrl` property instead:

```
templateUrl: "../login.component.html",
```

- The HTML code can then be moved to an external file:

```
<!-- login.component.html -->
<h2>Please Log In</h2>
<p>Name:

<input type="text" [(ngModel)]="name"/></p>
<p>Password:

<input type="password" [(ngModel)]="password"/></p>
<p>{{message}}</p>
<button (click)="submit()">Submit</button>
```

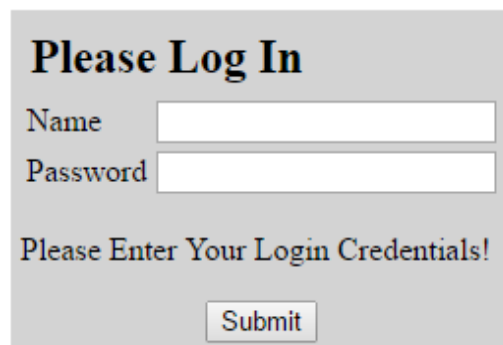
## 4.11 Login Component: Add CSS Styling

- CSS styles can be added into the `@Component styles` array.

```
styles:
 [`h1{color:red}`],
```

- Styles can be placed in external files by using the `styleUrls` instead:

```
styleUrls:
 ["../login.css"],
```



The image shows a web form with a light gray background. At the top, it says "Please Log In" in a bold, dark blue font. Below this, there are two input fields. The first is labeled "Name" and the second is labeled "Password". Both labels are in a dark blue font. Below the input fields, there is a line of text that says "Please Enter Your Login Credentials!". At the bottom of the form, there is a button labeled "Submit".

## Notes:

The component section now looks like this.

```
@Component({
 selector: 'login',
 template: `...`,
 styles: [
 div, h2 {margin: 0px; padding: 5px;}
 div {background-color: lightgrey; display: inline-block;}
 button {margin: 0 auto; display: block;}
],
})
```

Message text can be set by adding a 'message' property to the Component class.

The Component class and updated message property:

```
export class LoginComponent {
 message: string = "Please Enter Your Login Credentials!";
}
```

## 4.12 Login Component: Hook Up Input Fields and Button

Bindings in the HTML template refer to variables in the LoginComponent class.

- This input field binds to the 'name' variable:

```
<input type="text" [(ngModel)]="name"/>
```

- This input field binds to the 'password' variable:

```
<input type="password" [(ngModel)]="password"/>
```

- The click event of this button is mapped to the submit function:

```
<button (click)="submit()">Submit</button>
```

## 4.13 Login Component: Fields & Button in the Component Class

- The component class side of the bindings look like this:

```
export class LoginComponent {
 name: string;
 password: string;
 message: string = "Please enter your login details.";
 submit() {
```

```
 this.message = this.name + " is now logged in!"
 };
}
```

### Notes

The first few lines define the name, password and message class variables. Notice how we have specified the type of 'string' for each one.

After that comes the submit() method which is bound to a button element in the view.

The completed login component looks like this:

```
// login.component.ts
import { Component } from '@angular/core';

@Component({
 selector: 'app-login',
 template:
 `<h2>Please Log In</h2>
 <p>Name:

 <input type="text" [(ngModel)]="name"/></p>
 <p>Password:

 <input type="password" [(ngModel)]="password"/></p>
 <p>{{message}}</p>
 <button (click)="submit()">Submit</button>`,
 styles:[`
 div,h2{margin:0px;padding:5px;}
 div{background-color:lightgrey;display:inline-block;}
 button{margin: 0 auto;display: block;}
 `],
})
export class LoginComponent {
 name:string;
 password:string;
 message:string = "Please enter your login credentials.";
 submit(){
 this.message = this.name + " is now logged in!"
 };
}
```

## 4.14 Component Decorator Properties

- The component decorator `@Component` allows the developer to attach metadata to the component class. In this chapter we've used only a few of the available properties.
  - ◇ `selector`: the name to use when including the component in HTML
  - ◇ `template`: HTML template as a string



- ◇ templateUrl: a url pointing to a file containing HTML template code
- ◇ styles: an array of css style definitions
- ◇ styleUrls: an array of file names pointing to css style files
- ◇ providers: defines objects for injection into the component

### Notes:

We have used some of these in the current chapter. The 'providers' property will be introduced in an upcoming chapter.

## 4.15 Component Hierarchy

- Every application has a root component that is added to the **index.html**.
- A component can use other components from its template.
- In a single page application, each page is built using a component which may use other components to render the page.
- Deciding on how much functionality to include in a component and how much to extract out in a child component can be difficult. Here are a few guidelines:
  - ◇ If you are building a single page application, each page needs to be a separate component.
  - ◇ If a portion of the page appears in multiple pages, then create a reusable child component for that area.

## 4.16 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 bootstrappable 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.

## 4.17 Using a Regular Component

- All other components of the application are used in template HTML. Simply use the selector element for the component in a template.
- For example, we can use our LoginComponent from the template of AppComponent.

```
@Component({
 selector: 'app-root',
 template: `
 <app-login></app-login>
 `,
 styles: []
})
export class AppComponent {}
```

## 4.18 The Build System

- Angular CLI build system tries its best to output the smallest amount of code possible. To do so it only includes code for components that are actually used by the application.
- The builder starts with the bootstrap component of the application module. It then inspects the template of that component to determine what other components being used. It then walks the tree of these components and inspects their template for other components.

## 4.19 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 constructor is executed
  - ◇ `ngOnDestroy()`: Just before Angular destroys the component
  - ◇ `ngOnChanges()`: After Angular updates any data-bound member variable
  - ◇ `ngAfterViewInit()`: After the DOM elements for the component first rendered on screen
- 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/docs/ts/latest/guide/lifecycle-hooks.html>

## 4.20 Using a Lifecycle Hook: `OnInit`

- Code that needs to run every time a component is created can be placed in an `ngOnInit()` method in the component class.
- To run code in the `ngOnInit()` method you need to:
  - ◇ Import `OnInit` at the top of the component class file:

```
import { Component, OnInit } from '@angular/core';
```
  - ◇ Add "`implements OnInit`" to the signature of the component class:

```
export class MyComponent implements OnInit { ... }
```
  - ◇ Add a method with the exact name: `ngOnInit()` to the component class.

```
ngOnInit() { ... }
```
- `OnInit` is often used to call code that retrieves data required by the component.
- Examples `ngOnInit()` being used in this way appear through the course.

## 4.21 Summary

Topics covered in this chapter included:

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

## Chapter 5 - Data and Event Binding

---

### *Objectives*

Key objectives of this chapter:

- Learn how to bind values to view components,
- Bind view events to component functions,
- Bind input values to component properties,
- Use property bindings to manipulate view elements.

### 5.1 Binding Syntax

- Bindings appear in a component's template. They significantly simplify rendering data on a page and receiving user input.
- Input fields, view outputs, and view events can be connected to variables and code in the component class using the following syntax:

Output or display a variable in the view	<code>{{variable_name}}</code>
Bind input field values to a variable in the component class	<code>[(ngModel)] = "variable_name"</code>
Bind view events to a function on the component class	<code>(click) = 'submit()'</code>
Bind DOM element properties to component properties or expressions	<code>[disabled] = "isDisabled"</code>

- We will look at each of the above binding types in this chapter.

### 5.2 One-Way Output Binding

- One-way binding outputs data to the view
- Double Curly Bracket Syntax - `{{ expression }}`
- Is also referred to as "interpolation"
- Interpolation means, "adding or inserting something into something else"
- Can include expressions - `{{ a + b }}`

- Can show the return from a function - `{{ getMessage() }}`
- Properties & methods referenced in the output binding generally appear in the component class but can also refer to template local variables.

## Notes

Output binding takes the value of the given variable and inserts it into the HTML template. This is a one-way binding. The value of the variable is only displayed and cannot be changed by the user. In addition to outputting individual variables, the double brackets can hold expressions or function calls. In these cases, the expression or function is evaluated first, and its result value is displayed.

Template variables are used with `*ngFor`. Here "item" is a variable local to the template:

```
<p *ngFor="let item of items">{{item}}</p>
```

Template variables are also used in Angular forms to refer to forms and input fields. "f" and "first" in the code below are template variables.

```
<form #f="ngForm">
 <input name="first" ngModel required #first="ngModel">
</form>
<p>First name value: {{ first.value }}</p>
<p>Form valid: {{ f.valid }}</p>
```

## 5.3 Binding Displayed Output Values

```
...
template: `<h4>String Value</h4>
 <p>There are {{days}} {{unit}}
 in a year!</p>
 <h4>Date Value</h4>
 <p>{{today|date:'MM/dd/yyyy'}}</p>
 <h4>Paragraph Text</h4>
 <p>{{text}}</p>
`))
export class OutputComponent {
 text:string = `Lorem ipsum...`;
 days: number = 365;
 unit: string = "days";
 today: Date = new Date();
}
```

### Output Examples

#### String Value

There are 365 days in a year!

#### Date Value

05/09/2016

#### Paragraph Text

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

### Notes

Binding output values uses the double bracket syntax. In the first example two variables from the component class are output in the context of a sentence:

```
<p>There are {{days}} {{unit}} in a year!</p>
```

In the second example, a data variable is output. In this case, the output brackets also enclose some extra text besides the variable name. This is referred to in Angular as a 'pipe' and its purpose is to format the variable. Here we use it to output the date with a short date format. We will look at pipes in more detail in a later chapter.

```
<p>{{today|date:'MM/dd/yyyy'}}</p>
```

The last example displays an entire paragraph of text:

```
<p>{{text}}</p>
```

All the variables whose data was output appear in the OutputComponent class.

The full code of an example output component:

```
// output.component.ts
import { Component } from '@angular/core';

@Component({
 selector: 'output',
 template: `
 <h4>String Value</h4>
 <p>There are {{days}} {{unit}} in a year!</p>

 <h4>Date Value</h4>
 <p>{{today | date:'MM/dd/yyyy'}}</p>

 <h4>Paragraph Text</h4>
 <p>{{text}}</p>
 `
})

export class OutputComponent {
 text:string = `Lorem ipsum dolor
 sit amet, consectetur adipiscing elit.`;
 days: number = 365;
 unit: string = "days";
 today: Date = new Date();
}
```

## 5.4 Setting Component Properties

- Use the **@Input()** decorator for a field to allow it to be set by a parent

component.

```
import {Component, Input} from '@angular/core';
@Component({
 selector: 'alert',
 template: `<p>We have a message: {{message}}</p>`
})
export class AlertMessageComponent {
 @Input() message : string
}
```

- A parent component can supply data like this:

```
@Component({
 selector: 'parent',
 template: `<alert [message]="statusText"></alert>`
})
export class ParentComponent{
 statusText="This is a status message...";
}
```

## 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. The following are the same:

```
<alert message="Hello There"></alert>
<alert [message]=" 'Hello There' "></alert>
```

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

```
export class AlertMessageComponent {
 @Input("alertMessage") message : string
}
```

```
<alert [alertMessage]="statusText"></alert>
```

## 5.6 Setting DOM Element Properties

- The DOM API defines various properties for an element, like innerHTML



and `className`. Angular lets you set these properties.

- **Syntax:**

```
[property_name] = "expression"
```

- **Example:**

```
template:`<p [style.color] = "isRed ? 'red':'blue'">...</p>
<button [disabled] = "isDisabled()">...</button>
<div [style.width] = "'50px'">...</div>`
```

```
export class SimpleComponent {
 isRed : boolean = false
 isDisabled() : boolean {...}
}
```

## 5.7 Event Binding

- Used to bind component functions to any DOM event emitted by view elements: click, keydown, mouseover, etc.

- **Syntax:**

```
(event_name)="myEventHandler(arg1, arg2) "
```

- Binding is placed inside the element whose event is being bound.
- Example binding to a button's 'click' event:

```
<button (click)="saveData()">Save</button>
```

### Notes

Events are triggered for various reasons including when users interact with a view element. Code can be bound to any DOM event in Angular by placing the event name in parenthesis and assigning it a function or expression. The binding expression is placed inside the tag of the element where the event is expected occur.

In addition to triggering functions events can be used to execute code expressions directly. In this example the click event toggles a boolean 'flag' field in the component class:

```
<button (click)="flag=!flag">Button</button>
```

## 5.8 Binding Events Examples

```

/* Keydown Event */
<input id=txt1 type=text
 (keydown) = "toggle('txt1')"
 [(ngModel)]="first_name" >

/* Click Event */
<button id=btn1
 (click) = "toggle('btn1')">
 Button</button>

/* Mouse Events */
<div id=div1
 (mouseover)="toggle('div1')"
 (mouseleave)="toggle('div1')">
 Mouse over me!</div>

```

### Event Examples

#### Change Event

Enter Text:

#### Click Event

Click Me:

#### Mouse Over Event

Mouse over me!

### Notes

In the example we have three elements with bound events. In each case the events are bound to a simple function that toggles their background color.

The input field is set up so that every time the user enters a character the 'keydown' event is triggered and the field's background color is toggled.

The event being bound on the button element is the 'click' event. Each click on the button toggles the button's background.

Both the 'mouseover' and 'mouseleave' events are bound on the div element. Mousing over the div changes the color and moving the mouse out of the div changes it back.

The complete code for the example event component:

```

// event.component.ts
import { Component } from '@angular/core';

@Component({
 selector: 'eventcmp',
 template: `
 <h4>Change Event</h4>
 Enter Text:
 <input id=txt1 type=text (keydown) = "toggle('txt1')"

```

```
[ngModel]="first_name" >
<h4>Click Event</h4>
Click Me:
<button id=btn1 (click) = "toggle('btn1')">Button</button>
<h4>Mouse Over Event</h4>
<div id=div1 (mouseover)="toggle('div1')"
(mouseleave)="toggle('div1')" >Mouse over me!</div>`,
styles: [`#div1{width:200px;
height:50px; border:1px black solid;}`]
})

export class EventComponent {
 toggle(id){
 let e = document.getElementById(id);
 let bc = e.style.backgroundColor;
 bc = (bc != "lightblue")? "lightblue":"yellow";
 e.style.backgroundColor = bc;
 };
 first_name:string;
}
```

### 5.9 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.
- An event is fired using an EventEmitter field that is decorated with @Output.

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

### 5.10 @Output() Example - Child Component

- The child component triggers the "nameChange" event every time the input field changes. It does this by calling emit() on the valueChange EventEmitter.

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

```
@Component({
 selector: 'name-input',
 template: `
 Please enter your full name:

 <input type="text" (keyup)="onChange()"
 [(ngModel)]="fullName"/>
 <button>OK</button>`,
})
export class NameInputComponent {
 @Input() fullName: string;
 @Output() nameChange = new EventEmitter();

 onChange() { this.nameChange.emit(this.fullName) }
}
```

## 5.11 @Output() Example - Parent Component

- The parent component listens for the event and updates its own text property:

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

@Component({
 selector: 'my-app',
 template: `<h3>Input Output Example</h3>
 <name-input [fullName]="userName"
 (nameChange)="onValueChange($event)" >
 </name-input>`,
})
export class AppComponent {
 userName: string = '';
 onValueChange(event: string) {
 this.userName = event;
 }
}
```

### Notes

The payload of the event is saved in the **\$event** local variable by Angular. We can use that variable to supply the payload to the event handler method.

## 5.12 Two-Way Binding of Input Fields

- For use with input fields. The two ways are:
  - ◇ Pre-populate an input field with a component variable.
  - ◇ Update the component variable as the user enters data.
- Composite Bracket Syntax - `[(ngModel)] = "variable_name"`
- `ngModel` is a directive available from **FormsModule**. It must be imported in the application module
- The bound variable must come from the component class
- Example

```
<input type="text" [(ngModel)]="first_name">
```

### Notes

Two-way binding is typically used with input fields. Data is retrieved from the given variable in the component class and used to set the initial value of the input field. This part is similar to the one-way binding we saw earlier. Next, when the user enters the input field and modifies its value the new value is used to set the value of the given variable in the component class. This is the second half of the two way binding.

The syntax used associates the built in directive `ngModel` with a component class variable.

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

## 5.13 Input Binding Examples

```
import { Component } from
 '@angular/core';
@Component({
 selector: 'inputcmp',
 template: `<h4>Text Input</h4>
 First Name:<input type=text
 [(ngModel)]="first_name">

 Your first name is {{first_name}}
 <h4>Date Value</h4>
 Choose Vacation Start Date:
 <input type=date
 [(ngModel)]="vacation" >

```

### Input Examples

#### Text Input

First Name: Jason  
Your first name is Jason

#### Date Value

Choose Vacation Start

```

 Your Vacation starts: {{vacation}}
 `))
export class InputComponent {
 first_name:string;
 vacation: Date;
}

```

```

Date: 06/10/2016
Your Vacation starts: 2016-
06-10

```

## Notes

Full code of the input binding example component:

```

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

@Component({
 selector: 'inputcmp',
 template: `
 <h4>Text Input</h4>
 First Name:
 <input type=text [(ngModel)]="first_name">

 Your first name is {{first_name}}
 <h4>Date Value</h4>
 Choose Vacation Start Date:
 <input type=date [(ngModel)]="vacation" >

 Your Vacation starts: {{vacation}}
 `})

export class InputComponent {
 first_name:string;
 vacation: Date;
}

```

## 5.14 Two Way Binding in a Component

- Call the EventEmitter variable name @Input variable name + "Change".

```

@Component({
 selector: 'name-input',
 template: `
 Please enter a customer name:

 <input type="text" [(ngModel)]="fullName"/>
 <button (click)="onChange()">OK</button>`
})
export class NameInputComponent {
 @Input() fullName: string;
 @Output() fullNameChange = new EventEmitter();
}

```

```
onChange() { this.fullNameChange.emit(this.fullName) }
}
```

## Implement Two Way Binding

Two-way binding becomes necessary when a child component needs to report back any changes to an input data. This way a parent can supply an initial value of the input and constantly be kept up to date when that data changes.

### 5.15 Use Two Way Binding

- Use the composite bracket syntax from the parent.

```
@Component({
 selector: 'app-root',
 template: `
<h3>Input Output Example</h3>
<name-input [(fullName)]="userName"></name-input>
 {{userName}}
 `,
})
export class AppComponent {
 userName: string = 'Daffy Duck';
}
```

### 5.16 Breaking Down ngModel

- The two way binding syntax [(ngModel)]="propertyName" is actually a short cut for two separate bindings:

```
<input type="text" [ngModel]="age"
 (ngModelChange)="onAgeChanged($event)"/>
```

```
export class SimpleComponent {
 age:string

 onAgeChanged(newAge:string) {
 this.age = newAge
 }
}
```

}

## 5.17 Summary

In this chapter we:

- Learned how to bind values to view components,
- Bound view events are to component functions,
- Bound input values to component properties.
- Used property bindings to manipulate view elements.
- Output data from a child component to a parent component.



## Chapter 6 - Attribute Directives

---

### Objectives

Key objectives of this chapter

- What are Directives?
- Applying Styles by Changing Classes,
- Applying Styles Directly,
- Property Binding,
- Controlling Element Visibility,
- Setting Image and Hyperlink Source Dynamically.

### 6.1 What are 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.

### 6.2 Apply Styles by Changing CSS Classes

- The `ngClass` directive can be used to dynamically add or remove an element's CSS classes.

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

- Condition can be a literal value, expression or function call.

- The result of the directive depends on the type of value assigned to it:

string	"active bordered"	Adds all the classes in the string
Array	["active", "bordered"]	Adds all the classes from the array
Object	{active: isActive, bordered: hasBorder}	Adds any class whose corresponding variable is true

- When classes are added to an element, the browser looks up the corresponding styles and applies them to the element.
- When classes are removed from the element, the browser removes the corresponding styles.

### Notes:

In the object in the table above "active" and "bordered" are class names with corresponding style definitions:

```
.active{
 color: red;
}
.bordered{
 border:solid 1px black;
}
```

"isActive" and "hasBorder" are boolean variables from the component class:

```
export class MyView {
 hasBorder: boolean = false;
 isActive: boolean = true;
}
```

## 6.3 Changing Classes – Example

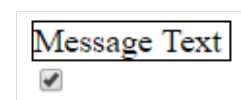
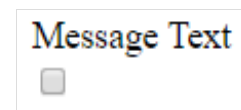
- Add the following element to a component template:

```
<span [ngClass]="{bordered:
 isBordered}">
 Message Text

```

- Add isBordered as a class variable:

```
export class MyView {
 isBordered: boolean = false;
```



```
}
```

- Add some class based styles also:

```
.bordered{ border:solid 1px black}
```

- Clicking the checkbox toggles the border

```
<input type=checkbox
[(ngModel)]="isBordered">
```

## Notes:

Full ngClass example:

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

@Component({
 selector: 'myview',
 template:
 `

Message Text

<input type=checkbox [(ngModel)]="isBordered">`,
 styles: ['.bordered{ border:solid 1px black}'],
})
export class MyView {
 isBordered: boolean = false;
}
```

## 6.4 Applying Styles Directly

- The **ngStyle** directive can be used to directly apply styles to HTML elements.

```
<div [ngStyle]="styleExp"></div>
```

- styleExp can be an Object literal or an expression that evaluates to an object:

Object Literal	[ngStyle]="{ color: 'blue' }"
Component Var	[ngStyle]="styles" // in html template styles:Object = { color: 'blue' } // in component class
Function	[ngStyle]="getStyles()" // in html template getStyles(){ return { color: 'blue' }; } // in component class

**Notes:**

The term "expression" can refer to any of the following:

- A variable: e.g. `myVar`
- A code expression: e.g. `myVar + myVar2`
- A function call: e.g. `myFunction()`

## 6.5 Applying Styles Directly - Example

- Styles for `Message_1` are set directly:  
`[ngStyle]="{fontStyle:'italic', fontFamily:'sans-serif' }"`
- Styles for `Message_2` are set using a variable:

```
[ngStyle]="styles"
```

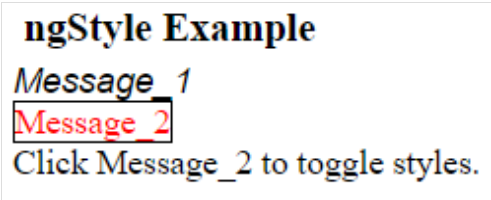
- The "styles" variable is set by clicking on `Message_2`  
`(click)="toggleStyle() "`

- `toggleStyle()` is a function in the component class:

```
toggleStyle(){
 if(this.styles === this.styles1){
 this.styles = this.styles2;
 } else { this.styles = this.styles1;}
}
```

- The styles are also in the component class:

```
styles: Object=this.styles1;
styles1: Object={border:'solid 1px black', color:'red'};
styles2: Object={border: 'none', color: 'black' };
```

**Notes:**

The full `ngStyles` example:

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

```
import {NgStyle} from '@angular/common';

@Component({
 selector: 'ngstyleview',
 template:
 `<h3>ngStyle Example</h3>

 Message_1

 Message_2

 Click Message_2 to toggle styles.` ,
 styles: ['h3{margin:5px;}']
})
export class NgStyleView {
 styles: Object = this.styles1;
 styles1: Object = {border: 'solid 1px black', color: 'red' };
 styles2: Object = {border: 'none', color: 'black' };
 toggleStyle(){
 if(this.styles === this.styles1){
 this.styles = this.styles2;
 }else{
 this.styles = this.styles1;
 }
 }
}
```

## 6.6 Controlling Element Visibility

- An element's "hidden" property can be used to control element visibility.

Value	Behavior
hidden is true	The element is hidden but remains in the DOM
hidden is false	The element is displayed

- The hidden property can be set like this when using Angular:  
**[hidden]="condition"**
- Condition is any variable or expression that resolves to a boolean value.
- An example setting the hidden property:  
`<div [hidden]="!showDiv" >some content...</div>`
- Hiding an element in this way does not remove it from the DOM.

### Notes:

"showDiv" in the code above refers to a variable in the component class:

```
showDiv: boolean = true;
```

If this variable is modified at runtime (e.g. by a click event) Angular sees the change and applies it to the element property so that the element is hidden or shown.

### 6.7 Setting Image Source Dynamically

- The **"src"** property of an image element can be set in Angular via a property binding.  

```

```
- The value that the "src" property can be set to a variable or function whose output changes over time.
- For example, changing the "imageUrl" variable will automatically change the image.
- This is different than setting the "src" attribute which only provides an initial value for the image source.



#### Notes:

Full image src example:

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

@Component({
 selector: 'srcview',
 template:
 `<h3>[src] Example</h3>

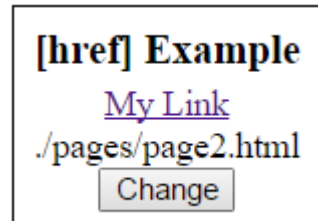
 <input type="button" value="Change" (click)="changeImage()">
 `,
 styles: [`host:{ border:solid 1px black;}
 img{height:100px; width:100px}`]
})
export class SrcView {
 imageUrl:string = "../image/badge0.png";
 badges:string[] = ["../image/badge0.png","../image/badge1.png","../image/badge2.png"];
 index:number = 0;
 changeImage() {
 this.index = this.index + 1;
 this.index = (this.index < 3)?this.index:this.index = 0;
 this.imageUrl = this.badges[this.index];
 }
}
```

## 6.8 Setting Hyperlink Source Dynamically

- The **"href"** property of an anchor element can be set in Angular via a property binding.

```
<a [href]="hrefUrl">My Link
```

- The value that the **"href"** property can be set to a variable or function whose output changes over time.



- For example, changing the **"hrefUrl"** variable will automatically change the location to be opened when someone clicks on the link.
- Hovering over the link in most browsers will display the href property value in the browsers status bar.
- This is different than setting the **"href"** attribute which only provides an initial value for the anchor link.

### Notes:

Full example using [href] to change the link location dynamically:

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

@Component({
 selector: 'hrefview',
 template:
 `<div><h3>[href] Example</h3>
 <a [href]="hrefUrl">My Link

 {{hrefUrl}}

 <input type="button" value="Change"
 (click)="changePage()"></div>
 `,
 styles: [
 `div{ border:solid 1px black; display:inline-block;
 text-align:center; padding:5px;}
 h3{margin:5px;}img{height:100px; width:100px}`]
})
export class HrefView {
 hrefUrl:string = "../pages/page0.html";
 pages:string[] =
 ["../pages/page0.html","../pages/page1.html","../pages/page2.html"];
 index:number = 0;
 changePage() {
 this.index = this.index + 1;
 this.index = (this.index < 3)?this.index:this.index = 0;
 this.hrefUrl = this.pages[this.index];
 }
}
```

```
}
```

## 6.9 Writing a Custom Attribute Directive

- Create a class with `@Directive` decorator. Inject `ElementRef` in constructor.

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

@Directive({
 selector: '[appImportant]'
})
export class ImportantTextDirective implements OnInit {
 constructor(private el: ElementRef) {}

 ngOnInit() {
 this.el.nativeElement.style.fontWeight = "bold"
 this.el.nativeElement.style.color = "red"
 }
}
```

### Notes

The `ElementRef` injected here refers to the DOM element where the attribute directive is applied. The `nativeElement` property of `ElementRef` is the actual DOM Element object. With that you have access to the full DOM API to manipulate the element.

## 6.10 Using a Custom Attribute Directive

- Register the directive class in the app module.

```
import { ImportantTextDirective } from './important-
text.directive';
@NgModule({
 declarations: [..., ImportantTextDirective]
 ...
})
```

- Use the directive from a component's template.

```
template: `
```



```
<p appImportant>Awesome Deal!</p>
```

## 6.11 Supplying Input to a Directive

- An attribute directive can take a single input. We need to define an input variable with the same name as the directive's selector attribute.

```
export class ImportantTextDirective implements OnInit {
 @Input('appImportant') fontSize:string = "24px"
 constructor(private el: ElementRef) {}

 ngOnInit() {
 ...
 this.el.nativeElement.style.fontSize = this.fontSize
 }
}
```

- Supply input from the template.

```
<p appImportant="34px">Awesome Deal!</p>
```

## 6.12 Handling Event from a Custom Directive

- Use the `@HostListener` decorator with a listener method. The `@HostListener` decorator lets you subscribe to events of the DOM element that hosts your custom attribute directive.

```
@HostListener("click")
itemClicked() {
 this.el.nativeElement.style.borderColor = "green"
 this.el.nativeElement.style.borderWidth = '2px'
 this.el.nativeElement.style.borderStyle = "solid"
}
```

- The following import statement is required at the top of the file:

```
import { Directive, ElementRef, HostListener } from
'@angular/core';
```

## 6.13 Summary

**In this chapter we covered:**

- Attribute Directives,
- Applying Styles by Changing Classes,
- Applying Styles Directly,
- Property Binding,
- Controlling Element Visibility,
- Setting Image and Hyperlink Source Dynamically.
- Develop Custom Attribute Directives

## Chapter 7 - Structural Directives

---

### *Objectives*

Key objectives of this chapter

- Adding and Removing an Element Dynamically with **ngIf**
- Looping using **ngFor**
- Conditional rendering using **ngSwitch** and **ngSwitchWhen**

### 7.1 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.

### 7.2 Adding and Removing Elements Dynamically

- The **ngIf** directive can be used to remove and restore DOM elements.
- It is often associated with a boolean variable from the component class:

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

- The asterisk (\*) prefix 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>
```

- If the condition is false then **ngIf** completely excludes its **ng-template** from execution. That means no DOM elements are created for it and more importantly if the **ng-template** contains any child components they are not created.

- ◇ In contrast, using the [hidden] attribute directive to hide elements do not prevent ng-template from execution.

## Notes:

When the element is removed it no longer exists in the DOM and since it is no longer in the DOM it is no longer processed along with other DOM elements. This is an important distinction. It can result in improved performance when showing and hiding elements that contain many subelements like divs representing an entire page.

The right side of the directive statements can be a condition or function call:

```
*ngIf="myVar === 'showit'"
*ngIf="showIt()"
```

In the above code myVar and showIt() are part of the corresponding component class:

```
export class MyView{
 myVar: string = "showit";
 showDiv = false;
 showIt(){
 return this.showDiv;
 }
}
```

## 7.3 If-Else Syntax of ngIf

- Angular 4 added an additional syntax for the ngIf directive
- You can use an 'else' to indicate the name of an <ng-template> element that would be shown if the condition is false

```
<div *ngIf="isLoggedIn; else login">
 The user is logged in.
</div>
<ng-template #login>Please login to continue.</ng-template>
```

- You can also use 'if then, else' syntax to create two <ng-template> elements that will be chosen between

```
<div *ngIf="isLoggedIn; then logout else login"></div>

<ng-template #logout>Hi Gary, <button>Logout
now</button>.</ng-template>
```

```
<ng-template #login>Please login to continue.</ng-template>
```

## If-Else Syntax of ngIf

In the 'then .. else' example above, one or the other of the `<ng-template>` elements will be substituted depending on if the variable 'isLoggedIn' is true or false.

## 7.4 Looping Using ngFor

- **ngFor** is a *repeater* directive
- Loops over a JavaScript array and executes a section of template repeatedly
- Often used to construct lists or tables

## 7.5 ngFor - Basic Example

```
@Component({...
 template: `
 <p *ngFor="let p of pets">{{p.name}} ({{p.type}})</p>`
})
export class AppComponent {
 pets = [
 {name: "Fiffy", type: "Cat", age: 2},
 {name: "Fido", type: "Dog", age: 1},
 {name: "Mimi", type: "Turtle", age: 2}]
}
```

Fiffy (Cat)

Fido (Dog)

Mimi (Turtle)

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

## 7.6 Creating Tables with ngFor

```
<table border="1">
 <tr>
 <td>Name</td><td>Type</td><td>Age</td>
 </tr>
 <tr *ngFor="let p of pets">
 <td>{{p.name}}</td>
 <td>{{p.type}}</td>
 <td>{{p.age}}</td>
 </tr>
</table>
```

Name	Type	Age
Fiffy	Cat	2
Fido	Dog	1
Mimi	Turtle	2

## 7.7 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.name}}

```

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

### Pets - local vars

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

## 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.

## 7.8 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.

## 7.9 Example - Deleting an Item

- Component template:

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

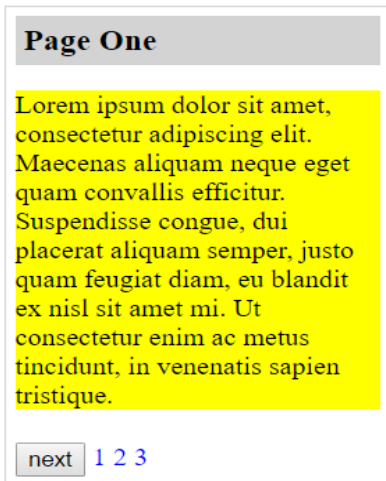
- Component code:

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

## 7.10 Swapping Elements with ngSwitch

### ngSwitch

- Is a *structural* directive
- Is used along with ngSwitchCase and ngSwitchDefault
- Shows one element while hiding others
- Can be used to implement tabs or basic single page apps
- Has basic and template syntaxes



### Notes

The screen shot shows a component that uses ngSwitch to switch between component directives to implement page-like navigation.



## 7.11 ngSwitch - Basic Syntax

### ■ ngSwitch Basic Syntax

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

- When the type in the array matches the ngSwitchCase the element is shown and all others are hidden.

### Notes

The example shown here switches between simple <p> tags. In a real application what get switched is up to you and can include a few simple elements or whole pages of content.

## 7.12 Summary

### In this chapter we covered:

- Adding and Removing an Element Dynamically with ngIf
- Looping using ngFor
- Conditional rendering using ngSwitch

Page-1

Next



## Chapter 8 - 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
- The updateOn property
- Displaying and using validation state
- Accessing data via the form object
- Submitting forms

### 8.1 Template Driven Forms

- Angular Defines two approaches to creating forms:
  - ◇ Template Driven Forms
  - ◇ Model Driven Forms
- This chapter focuses on Template Driven Forms
- Template Driven Forms:
  - ◇ Can use ngModel bindings
  - ◇ Support validation rules in the HTML form template
  - ◇ Require the use of end-to-end testing methods
  - ◇ Work well for small to medium sized forms with limited user interaction

### 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 Model Driven Forms must be used. We cover Model Driven 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.

## 8.2 Importing Forms Module

- To import the FormsModule make the following changes to the app.module.ts file:
  - ◇ Import the following:

```
import { FormsModule } from '@angular/forms';
```
  - ◇ Update the @NgModule imports array:

```
@NgModule({
 imports: [BrowserModule, FormsModule],
```
- All code listed from here on in this chapter assumes that these changes have been applied.

### 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 { }
```

### 8.3 A Basic Angular Form

Description:

- Uses the form object
- Input fields are bound to the form
- Includes a Submit button
- A function is bound to the submit operation
- Data from the form's fields is passed to the submit function



**Form1: Registration**

Name:

Email:

### 8.4 Binding Input Fields

- Fields are bound to form controls using ngModel:

```
[(ngModel)]="name"
[(ngModel)]="email"
```

- Form HTML Template:

```
<input type=text [(ngModel)]="name" required />
<input type=email [(ngModel)]="email" required />
```

- Using [(ngModel)] as shown above implements two way binding between the input field and a variable of the given name in the component class.
- Using ngModel to bind input fields contained within <form></form> tags also associates those fields with the form object. Later we'll see how this allows us to access the form's "valid" and "value" properties.

## Notes:

Two way binding with input fields means that:

- The value of the component class variable will be used to set the input field's initial value.
- And that changes the user makes to the input field will be applied to the component class variable as well

## 8.5 Accessing the NgForm Directive

The NgForm directive represents a <form> element and has various useful properties.

- You can get a reference to the NgForm object like this:  

```
<form #theForm="ngForm" >
```
- The above code creates a variable named "theForm" that points to an instance of the NgForm directive.
- Some properties of NgForm include:
  - ◇ value - An object holding the value of the forms input fields
  - ◇ valid, invalid - A boolean indicating if the form is fully valid or has invalid input.

## Notes

#theForm is a template variable referring to the form element. You could use any name here in fact #f is often used instead of #theForm)

Technically the #theForm variable is of the Angular type NgForm. See here for more details:  
<https://angular.io/api/forms/NgForm>

## 8.6 Binding the Form Submit Event

- This part binds a function to the form's submit event:  

```
<form #theForm="ngForm"
 (ngSubmit)="onSubmit(theForm)">
```
- The "theForm" passed to onSubmit() has a "value" property pointing to

a JSON object.

- The `theForm.form.value` property includes values from all the form's fields
- A submit button added to the form will invoke the "onSubmit() :  
`<input type=submit value=Submit >`
- Note that a form `action` attribute is not required.

### Notes:

(ngSubmit) refers to the submit event on the enhanced form object.

We bind the form's submit event and pass the form template variable to the onSubmit() function:

```
(ngSubmit)="onSubmit(theForm);"
```

We use a standard input element of type "submit" to trigger the form's submit event:

```
<input type=submit value=Submit>
```

Angular does not use the form "action" attribute. The 'action' attribute is only required when submitting plain non-angular forms.

## 8.7 The Submit Function

- An example of a function bound to Submit:

```
onSubmit(form) {
 console.log("Submitted:" +
 JSON.stringify(form.form.value, null, 2));
};
```

- The Angular form object is being passed in:

```
onSubmit(form)
```

- Which allows us to access the form object's "value" property:

```
form.value (e.g. {"password":"mypass"})
```

- The names of `form.value`'s properties come from the name attributes of our input elements:

```
<input name=password [(ngModel)]="person.pass" >
```

### Notes:

Note that in the last line above the value of the input field is bound to two separate objects:

- person.pass
- form.value.password

Person is an object in the component class.

Value is an object on the form object.

## 8.8 Basic HTML5 Validation - "required" Attribute

- The "required" attribute in the input element causes the field's value to be checked before allowing the form to be submitted:

```
<input type=text required />
```

- A message is displayed after clicking on submit if the required field is empty:

"Please fill out this field"

### Notes

You might be familiar with validation messages like the one you see here. This behavior is based on the HTML5 standard and is implemented by the browser. When the user clicks on submit the browser checks the form's input fields to see if they are valid based on the input type, the required attribute, and other validation attributes. When a field is not valid, the browser displays the message you see here. When all fields are valid, no messages are displayed, and the submit is executed.

## 8.9 HTML5 vs. Angular Validation

While HTML5 validation may be good for some situations it lacks some features that are needed for more serious applications.

- For example your application may require that you:



- ◇ Have error messages appear as soon as you exit an invalid field.
- ◇ Customize error message text.
- ◇ Programmatically test form validation
- ◇ Create and apply your own validation rules
- Angular turns off HTML5 validation automatically by adding the `html novalidate` attribute to form tags. (you can see this by inspecting the form html in the browser at runtime)
- If you want to have both Angular and HTML5 validation be active then you need to add the angular `ngNativeValidate` directive like this:

```
<form #f="ngForm" ngNativeValidate
 (ngSubmit)="onSubmit(f)" >...</form>
```

### Notes:

The form and field validation states generated by Angular do not take into account input field type validation which is imposed by HTML. (e.g. checking for @ symbol when input type=email.)

Angular's "pattern" validator can be used to validate various types of data.

For example the following pattern implements a rough email format:

```
<input type=text name=email [(ngModel)]="person.email"
pattern="[a-zA-Z]+@[a-zA-Z]+[\.]+[a-zA-Z]{3}"
required >
```

For more complex cases you may need to look into writing a custom validator.

## 8.10 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 >
```

## 8.11 The NgModel Directive

- Besides being used for two way data binding, NgModel also keeps track of the validity status of the input control
- It has these properties: valid, invalid, touched, untouched, dirty, pristine

```
<p *ngIf="emailTxt.invalid">Please enter a valid email</p>
<input name="userId" type="email" email required
 #emailTxt="ngModel" ngModel/>
```

- Validation state Information is also available via pre-defined classes that are automatically applied to input elements:

```
ng-valid, ng-pristine, ng-touched
ng-invalid, ng-dirty, ng-untouched
```

## 8.12 Controlling when validation is applied

- By default, validation state is calculated after any input field change which depending on the field could mean after every keystroke.

- It is also possible to apply validation less frequently by setting the **updateOn** property to one of the following values:
  - ◇ blur - validation is done when field loses focus
  - ◇ submit - validation is done when form is submitted
  - ◇ change(default) - validation is done after every keystroke
- The following code sets updateOn to blur for an individual input field:

```
<input [ngModelOptions]="{ updateOn: 'blur' }" ... >
```
- The following code sets updateOn to submit for the entire form:

```
<form [ngFormOptions]="{ updateOn: 'submit' }" ... >
```

### 8.13 Displaying *Form* Validation State

- The "form" template variable contains a property for the overall validation state of the form.

```
theForm.form.valid
```
- Using our "theForm" template variable we can easily display the form's state:

```
<form #theForm=ngForm
 (ngSubmit)="onSubmit(theForm);">
 Name:

 ...
 <p>Form Valid: {{theForm.form.valid}}</p>
</form>
```
- The paragraph <p> section above displays one of the following based on the form's validation state:

```
Form Valid: true
Form Valid: false
```

## 8.14 Displaying *Field* Validation State

- Controls for input fields have their own validation states.
- To display field state we first setup a template variable for the field:

```
<input [(ngModel)]="fname" type="text"
 name="first_name" #first_name="ngModel"
 required >
```

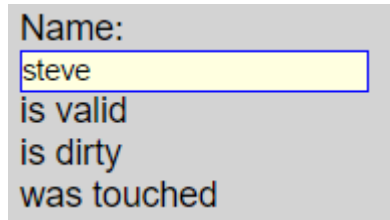
- With this template variable we can display information about the field's state:

```
is
valid

is
dirty

was
touched

```



### Notes

"Touched" refers to whether or not the user has tabbed through the field. Using this information you could exclude untouched fields from any updates.

"Dirty" refers to the state where the user has not only tabbed to the field but they made changes to the field when focused on it.

## 8.15 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;}
```

## 8.16 Disabling Submit when Form is Invalid

- You may wish to disable the submit button when the form is not valid to avoid users submitting invalid data.
- To do this we use the form template variable:

```
<form #f=ngForm (ngSubmit)="onSubmit(f);">
 ...
 <input id=submit type=submit value="Submit"
 [disabled]="!f.form.valid">
</form>
```

- This sets the submit input element's disabled property to true when the form is invalid:

```
[disabled]="!f.valid"
```

- The following will also disable the button once the form has been submitted:

```
[disabled]="!f.form.valid || f.form.submitted"
```

### Notes:

Full code of a form component that has its submit button conditionally disabled:

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

@Component({
 selector: 'validatingform',
 template: `<h3>Validating Form</h3>
<form #f="ngForm" novalidate (ngSubmit)="onSubmit(f)" >
<input name=xname [(ngModel)]="person.name" required >

<input name=xpass [(ngModel)]="person.pass" required >

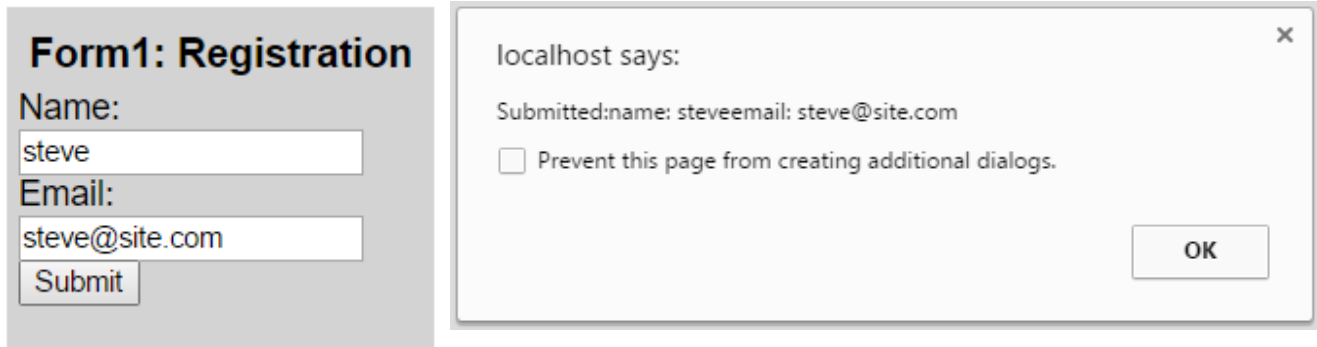
<input type=submit value=submit [disabled]="!f.form.valid || f.form.submitted">

</form>`
})
export class ValidatingFormComponent {
 person: Object = {name:"cindy", pass:"sass"};
 onSubmit(form){
 console.log("person: " + JSON.stringify(this.person, null, 2));
 console.log("form.value: " + JSON.stringify(form.value, null, 2));
 }
}
```

}

## 8.17 Submitting the Form

- Once valid data has been entered the form can be successfully submitted:



- The submit function in our simple form posts an alert message. A real life form might send the data to a server to be stored or to initiate some other operation.

### Notes:

The complete code for our basic form is in three files:

- basic.formcomponent.ts
- basic.form.component.html
- basic.form.component.css

FORM Component (basic.form.component.ts):

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

@Component({
 selector: 'basicform',
 templateUrl: './app/basic.form.component.html',
 styleUrls: ['./app/basic.form.component.css'],
})
export class BasicFormComponent {
 person: Object = {name:"cindy", email:"cindy@gmail.com"};
 onSubmit(form) {
 alert("Submitted: " + JSON.stringify(this.person));
 console.log("person: " + JSON.stringify(this.person, null, 2));
 console.log("form.value: " + JSON.stringify(form.value, null, 2));
 }
}
```

```
}
```

```
HTML Template(basic.form.component.html):
<h3>Basic Form</h3>
<form #f="ngForm" (ngSubmit)="onSubmit(f);">
Name:

<input type="text" [(ngModel)]=person.name name=xname required />
valid

Email:

<input type="email" [(ngModel)]=person.email name=xemail required />
valid

<input id=submit type=submit value=Submit
 [disabled]="!f.form.valid || f.form.submitted" >
<p *ngIf="f.form.valid && !f.form.submitted">Form is valid, please submit!</p>
<p *ngIf="f.form.submitted">Registration has been submitted!</p>
</form>
```

CSS styles (basic.form.component.css):

```
:host{
 background-color:lightgrey;
 margin:5px;
 display:inline-block;
 padding:5px;
 font-family:sans-serif;
}
h3{ margin:0px; padding:5px;}
#submit{
 margin: 0 auto; margin-top: 5px;
 display: block;
}
input.ng-invalid{ border: 1px solid red;}
input.ng-valid{ border: 1px solid blue; }
input.ng-pristine{ background-color: lightgrey;}
input.ng-touched{ background-color: lightyellow;}
```

### 8.18 Binding to Object Variables

- When forms include a large number of fields it is often useful to group them all into a class and then bind fields on the form to the properties on a single instance of the class.
- In prior examples we used an object defined in the component class to hold input field values:

```
export class FormComponent {
 person: Object = { name:"cindy",
```

```
 email:"sass@gmail.com"};
 onSubmit(form) {...}
}
```

- If the same object is used in several places in your program you may want to define it in a separate class that can be reused.
- Steps:
  - ◇ Create and export a class with properties for each field on the form
  - ◇ Import the class into the form component
  - ◇ Inject an object/instance of the class into the form component class
  - ◇ Bind form variables to this object/instance

## 8.19 Binding to Object Variables - Code

- Here is an example class:

```
/* file: registration.class.ts */
export class Registration {
 name:string;
 email:string;
}
```

- Here is the component that uses the class:

```
import { Component } from '@angular/core';
import { Registration } from './registration.class';

@Component({
 selector: 'form2',
 templateUrl: './app/form2.component.html',
 styleUrls:['./app/form2.component.css'],
})
export class Form2Component {
 constructor(private registration: Registration){}
 onSubmit(data){
 let msg = "Submitted:" +
 JSON.stringify(registration);
```

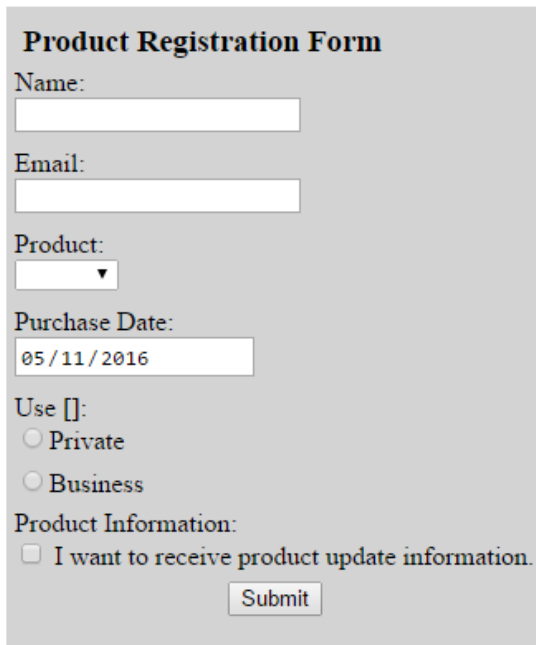


```
 console.log(msg) ;
 } ;
}
```

- Here is an example of the input fields in the HTML template that bind to the object instance:

```
<input type=text name=name
[(ngModel)]="registration.name" />
<input type=email name=email
[(ngModel)]="registration.email"/>
```

## 8.20 Additional Input Types



The screenshot shows a web form titled "Product Registration Form". It contains several input fields: a text field for "Name:", a text field for "Email:", a dropdown menu for "Product:", and a date field for "Purchase Date:" showing "05/11/2016". Below these are two radio buttons labeled "Private" and "Business" under the heading "Use []:". At the bottom, there is a checkbox labeled "I want to receive product update information." and a "Submit" button.

- 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.

## 8.21 Checkboxes

- Checkboxes use a simple ngControl binding

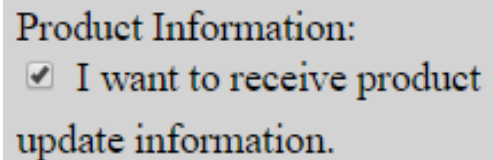
```
<p>Product Information:</p>
<input type=checkbox name='updateInfo'
 [(ngModel)]="updateInfo">
```

- 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);
}
}
```

### 8.22 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>
```

## 8.23 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

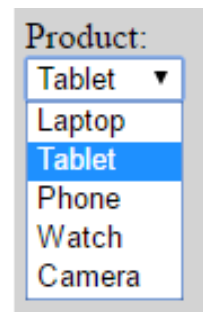
```
Product:

<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>
```

## 8.24 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)

## 8.25 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)); }
}
```

### 8.26 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
- The updateOn property
- Displaying and using validation state
- Accessing data via the form object
- Submitting forms





## Chapter 9 - Reactive Forms

---

### *Objectives*

This chapter includes the following model driven forms topics:

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

### 9.1 Reactive Forms Overview

- A **template driven form** uses `[(ngModel)]` directive to directly modify component state variables. This data binding works asynchronously. For example, updates to the component's state is reflected in the input controls in the next clock tick.
- A **reactive form**, in contrast, maintains its own state separate from the component's state. As the user enters data the form's state is modified synchronously. The component's state remains unchanged.
- A reactive form requires a little bit of extra coding but the synchronous nature of it's data binding has a few advantages:
  - ◇ Easier to unit test compared to template driven form.
  - ◇ In some complex scenarios the asynchronous nature of template driven form can present problems.

### 9.2 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.

### 9.3 Import ReactiveFormsModule

- To use the full features of model driven 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],
```

### 9.4 Construct a Form

- Design the form in the component class.

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

@Component({...})
export class MyComponent {
 myForm: FormGroup

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

```
}
```

## 9.5 Design the Template

- 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>
```

## 9.6 FormControl Constructor

- The `FormControl` class constructor takes these optional arguments:
  - ◇ The initial value that will be used to populate the input field.
  - ◇ An array of validators.
  - ◇ An array of asynchronous validators. This is outside the scope of this chapter.

## 9.7 Getting Form 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
```

## 9.8 Setting Form Values

- You can set the initial value of an input using the first argument of the FormControl constructor.
- In some situations like in an edit form 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
 })
 })
}
```

## 9.9 The Synchronous Nature

- When you get or set values of a reactive form (using the approaches discussed above) you are pulling and pushing data synchronously.

- ◇ In contrast, [(ngModel)] based data binding works asynchronously. For example, after user enters some data it takes a few cycles for Angular to update the bound component variable.
- This synchronous nature avoids complications during unit testing and in certain advanced situations.

## 9.10 Subscribing to Input Changes

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

```
export class MyComponent {
 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)
})
```

## 9.11 Validation

- One of the biggest differences between template driven forms and model driven forms is where validation is declared
  - ◇ Template driven forms have validation properties defined in the HTML template
  - ◇ Model driven 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]
```

## 9.12 Built-In Validators

- Angular has a few built-in validators:

```
Validators.required
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}'))
```

## 9.13 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.

## 9.14 Custom Validator

- Custom Validators:
  - ◇ Take a Control as input
  - ◇ Return null when the control is valid
  - ◇ Return an object with validation info when control is invalid
- Example ( `email.validator.ts` ):

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

export function validateEmail(control: FormControl)
{
 if(typeof(control.value) === 'string'){
 if(control.value.includes("@")){
 return null;
 }else{
 return {validateEmail: {valid: false}}
 }
 }
}
```

### Notes

To create a custom validator:

- create a separate file for the validator

- import FormControl so we can reference it
- create and exporting the validation function
- have the validation function accept a FormControl parameter
- have the validation function return null or object
- 

The example validator above simply checks for the presence of the '@' symbol in the string.

### 9.15 Using a Custom Validator

#### ■ Steps:

##### ◇ Import the validator function

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

##### ◇ Reference the function in a FormControl

```
this.myForm = fb.group({
 email: ["james@abc.com", [validateEmail]]
});
```

#### Notes:

Full code of validator function ( email.validator.ts ):

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

export function validateEmail(control: FormControl) {
 if(typeof(control.value) === 'string'){
 if(control.value.includes("@")){
 return null;
 }else{
 return {validateEmail: {valid: false}}
 }
 }
}
```

Full code of component that uses the custom validator ( custom.validator.component.ts ):

```
import { Component, OnInit } from '@angular/core';
import { validateEmail } from './email.validator';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
 selector: 'customvalform',
 template: `
<h3>Custom Validator Form</h3>
<form [formGroup]="myForm">
 Email:
 <input formControlName="email" >

 </form>
`
})
export class CustomValidatorComponent implements OnInit {
 myForm: FormGroup;

 ngOnInit() {
 this.myForm = new FormGroup({
 email: new FormControl('', [validateEmail])
 });
 }
}
```



```
<input type=button value=submit (click)=onSubmit() >
</form>
Form Valid: {{myForm.valid}}

Form Value: {{myForm.value|json}}
`,
})
export class CustomValidatorComponent implements OnInit {

 myForm: FormGroup;

 ngOnInit() {
 this.myForm = new FormGroup({
 email: new FormControl('Jim@abc.com', [validateEmail]),
 });
 }

 onSubmit() {
 console.log("Valid: " + this.myForm.valid);
 console.log("Value: " + JSON.stringify(this.myForm.value, null, 2));
 }
}
```

### 9.16 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));
 }
}
```

```
}
```

## 9.17 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'

## 9.18 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.

## 9.19 Summary

In this chapter we covered the following Reactive Forms topics:

- Setup
- FormGroup initialization
- FormControl object
- Validation
- SubForms



## Chapter 10 - Angular Modules

---

### *Objectives*

Key objectives of this chapter

- Why Angular modules?
- The Root Module
- Defining modules
- @NgModule decorator
- Importing modules
- Organizing modules

### 10.1 Why Angular Modules?

- Angular modules help break up an application into blocks of functionality
- Modules allow the definition of important parts of an application or library that can then be imported when needed. This has two benefits:
  - ◇ We can use hundreds of components, services, and directives available from a module by simply importing that module. There is no need to import each artifact separately.
  - ◇ The build system can walk the module dependency tree and only output code that is being used by the application.
- Definitions within modules can be "lazy loaded" only when required to improve application initialization
- For a very large application, you can create modules that hide functionality and only expose components, services etc. that are of use by the rest of the application.
- The module structure allow 3<sup>rd</sup> party extensions to Angular to be provided in a similar way to core Angular features
- An Angular application always has at least one module, the "root" module

## 10.2 But, We Already Had ES6 Module

- JavaScript ES6 has introduced the idea of modules.
- The problem with ES6 modules was that all the classes in the module had to be defined in a single file. Also, there was no way to work with HTML template and CSS files. This is why the designers felt the need to introduce the notion of a module in Angular.

## 10.3 Angular Built-in Modules

- Many important Angular libraries are modules
  - ◇ **BrowserModule** – provides services that are essential to launch and run a browser app
  - ◇ **CommonModule** – Contains core Angular definitions needed in nearly all Angular components
  - ◇ **FormsModule** (and **ReactiveFormsModule**) – components and directives for working with Angular forms
  - ◇ **HttpModule** – Service for sending and receiving HTTP requests and responses
  - ◇ **RouterModule** – Controlling navigation within an Angular application

## 10.4 The Root Module

- Every Angular application has at least one module, the "root" module
- Although the root module does differ somewhat between applications, there are usually some common aspects
  - ◇ Usually defined in the 'app.module.ts' file
  - ◇ Imports standard Angular modules like 'BrowserModule' and 'FormsModule' or 'ReactiveFormsModule'
  - ◇ Declares at least the main component of the application, usually called 'AppComponent'
  - ◇ Bootstraps the main application component so it can be used on the first application HTML page

## 10.5 Feature Modules

- A "Feature Module" is simply an Angular module that is not the root module of an Angular application
  - ◇ A feature module is meant to extend the application with additional components and functionality
  - ◇ Feature modules help partition the app into areas of specific interest and purpose
- Feature modules have the same structure and are defined the same way as the root module
  - ◇ They are distinguished primarily by the intent of the module
- A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares
- The root module and feature modules share the same execution context
  - ◇ They share the same dependency injector, which means the services in one module are available to all
- Feature modules can not only provide better reusability of components but also a better division of development responsibilities

### Feature Modules

Only the root module will launch the application.

Generally, you could split components and other definitions into a separate module with only a few changes:

- Move all files related to the desired module contents into a subdirectory for separation
- Create a class with the `@NgModule` decoration and the various declarations
- Modify the `@NgModule` declaration of other module to import the new module and remove the various declarations that were moved to the new feature module

## 10.6 Create Feature Module Using CLI

- Run this command:

```
ng generate module payment --routing
```

- This will create a module class called `PaymentModule` in `payment/payment.module.ts`.
- To create component, services etc. in a feature module do:

```
ng generate component payment/CreditCard
```

```
ng generate service payment/MakePayment
```

- Components, directives, and pipes created this way will be added to the declarations list of the feature module.

## 10.7 The Module Class

- A TypeScript class decorated with the `@NgModule` decorator.
- Components, services, directives etc. are made a part of the module using the `@NgModule` decorator.

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { CreditCard } from '...';
import { MakePayment } from '...';
```

```
@NgModule({
 imports: [FormsModule],
 declarations: [CreditCard],
 exports: [CreditCard],
 providers: [MakePayment]
})
export class PaymentModule { }
```

## 10.8 @NgModule Properties

- The `@NgModule` decorator on the module class has several different properties which are arrays of references
  - ◇ These properties are the "metadata" about the module



- Some of the most important properties
  - ◇ **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

```
@NgModule ({
 imports: [CommonModule, FormsModule],
 declarations: [ContactComponent, AwesomePipe],
 exports: [ContactComponent],
 providers: [ContactService]
})
export class ContactModule { }
```

## @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.

## 10.9 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
- In this example:
  - ◇ MyModule is importing PaymentModule
  - ◇ All directives, components, and pipes exported by the PaymentModule will become available inside of MyModule.
  - ◇ All services listed in the providers property of PaymentModule will

become available inside MyModule.

```
import { PaymentModule } from '../payment/payment.module';

@NgModule({
 imports: [PaymentModule],
 ...
})
public class MyModule{}
```

## 10.10 Importing BrowserModule or CommonModule

- The root application module of almost every browser application should import **BrowserModule**
  - ◇ BrowserModule provides services that are essential to launch and run a browser app
- BrowserModule also re-exports CommonModule which contains many of the common Angular directives (NgIf, NgFor, etc)
- BrowserModule should NOT be imported by any feature module
  - ◇ Feature modules should explicitly load CommonModule if required instead of trying to get it with a BrowserModule import

### Notes:

BrowserModule will also throw an error if you try to lazy-load a module that imports it.

## 10.11 Lazy-Loaded Modules

- Any modules that are imported, either directly or indirectly, from the root module are "eagerly" loaded
  - ◇ This would be done as part of and at the time of application initialization
- Any modules used by the entry point (app-root) of the application must be loaded eagerly.
- Modules can be "lazy-loaded" upon request by the Angular Router:
  - ◇ This can improve application performance

- ◊ These modules would not be referenced, directly or indirectly, from the root module but instead only referenced and loaded as part of the Router configuration
- The `@NgModule` definition of a lazy-loaded module is no different from one that is not lazy-loaded.

## 10.12 How to Organize Modules?

- Separate "view" feature modules could be used for different definitions that are only required for one particular view in an application
  - ◊ Most of these might be lazily loaded by the Router only if that view is shown to the user
- You could define a "core" application module to contain
  - ◊ Definitions only used in the root module so they are not cluttering up the root module
  - ◊ Shared service providers that are used in several places in the application but should only ever have a single instance
- The "core" module could then be only imported by the root module
  - ◊ Since service providers are accessed using Angular dependency injection, you don't need to import the entire module everywhere they are used
- A "shared" module that has components reused by more than one view of the application but not common enough to justify inclusion in the "core" module
- "View", "core" and "shared" modules are all just feature modules in terms of definition and only differ in what they contain, scope of reuse, and where they are imported

## How to Organize Modules?

There is a situation where including a service provider in a module that is imported several times can cause a problem. Let's say we want a particular service to behave as an application-wide singleton, meaning that everywhere it is used, the same instance of the service is used. If you include the service definition in a "shared" module and then import that module into a lazy-loaded module, the lazy-loaded module will make it's own copy of the service. This would then not follow the "singleton" pattern.

Instead, include the service provider definition in the "core" module that is imported into the root module and nowhere else so only a single instance of the service exists.

### 10.13 Third Party Modules

- You may need to use modules written by other authors to add certain features to your application. Examples:
  - ◇ HTML editor
  - ◇ Image slide show
  - ◇ Video player, etc.
- To use a third party module:
  - ◇ First, add the NPM package to package.json and install it using NPM.
  - ◇ Import the module from the root module

### 10.14 Summary

In this chapter we covered:

- Why Angular modules?
- The Root Module
- Defining modules
- @NgModule decorator
- Importing modules
- Organizing modules

## Chapter 11 - 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

### 11.1 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
  - ◇ 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.

### 11.2 Creating a Basic Service

**To create a basic service:**

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

- Create a class and

```
@Injectable({
```

export it	providedIn: 'root'
■ Import Injectable	}))
■ Add @Injectable annotation	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.

## 11.3 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.

## 11.4 What Dependency Injection Looks Like

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

```
export class MyComponent{
 constructor(private myclass: MyClass) {}
 // use myclass properties and methods
}
```

- Instead of code like this:

```
export class MyComponent{
 myclass: MyClass = new MyClass();
 // use myclass properties and methods
}
```

- 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 MyClass
- Pass the instance into the constructor with the name "myclass"

The "private" keyword is a shortcut that tells angular to add "myclass" as a property of MyComponent.

Without the "private" keyword the code would look something like this:

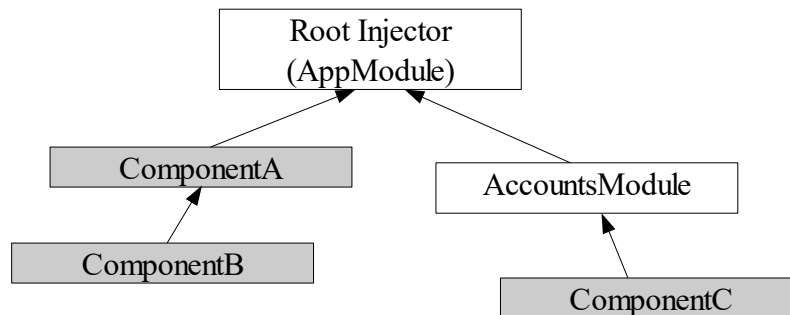
```
export class MyComponent{
```

```
myclass: MyClass;
constructor(_myclass: MyClass){
 this.myclass = _myclass;
}
// use myclass properties and methods
}
```

### 11.5 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.

### 11.6 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.

## 11.7 Register a Service with a Module Injector

- There are two ways you can register a service with a module's injector.
- Simplest and recommended way as of Angular 6 is to use the **providedIn** property of **@Injectable**.

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

- Alternatively, you can add the service class to the providers array of the **@NgModule**.

```
@NgModule({
 providers: [BillingService],
 ...
})
export class AccountsModule { }
```

- **A special rule:** If a module imports another module then all services registered with the child module also become registered with the parent module. So, for example, a component that belongs to the parent module can now inject a service from the child module. *Since all eagerly loaded modules are directly or indirectly imported by the AppModule, all services registered with a module injector become globally available anywhere in the application.*

## Note

The first approach (using `providedIn`) is recommended. This helps the tree shaker exclude the service class from the build output if it detects that the module is not in use by the application (has not be directly or indirectly imported by the `AppModule`).

## 11.8 Registering a Service with the Root Injector

- For small to medium sized applications that are not using any feature sub-modules you can directly register a service to the root injector. There are two ways to do that.
- As of Angular 6 you can set the **`providedIn`** property of `@Injectable` to "root". This is the default of a service generated by the CLI and the recommended approach.

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

- Alternatively, you can add the service class to the providers array of `AppModule`. This is the legacy approach.

```
@NgModule({
 providers: [BillingService],
 ...
})
export class AppModule { }
```

## 11.9 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).

### 11.10 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.

### 11.11 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.

## 11.12 Providing an Alternate Implementation

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

```
@Injectable({
 providedIn: 'root'
})
export class HelloService {
 greet() {
 return "Hello"
 }
}
```

```
@Injectable({
 providedIn: 'root'
})
export class GreetService {
 greet() {
 return "Greeting"
 }
}
```

```
@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) {...}
}
```

### 11.13 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],
```

## 11.14 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';
```

### 11.15 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 12 - HTTP Client

---

### *Objectives*

Key objectives of this chapter

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

### 12.1 The Angular HTTP Client

The Angular HTTP Client:

- Provides a simplified API for network communication
- Supports:
  - ◇ Making HTTP requests (GET, POST, etc.)
  - ◇ Working with request and response headers
  - ◇ Asynchronous programming
- Makes use of the rxjs async library Observable object

### Notes

The Angular Http client offers a simplified API compared to the JavaScript XMLHttpRequest object.

### 12.2 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.

## 12.3 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.

## 12.4 Simple Example

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

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

 test() {
 let o: Observable<Object> = this.http.get("/posts/1")

 o.subscribe(result => console.log(result))
 }
}
```

## 12.5 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(httpClient: HttpClient){}

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

## 12.6 ES6 Import Statements

- There are three imports in total that we must add to the data service:

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

## 12.7 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.

## 12.8 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>

## 12.9 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

### PeopleList

Raymond, Ward, rward0@oracle.com  
Daniel, Chavez, dchavez1@dedecms.com  
Sharon, Dunn, sdunn2@so-net.ne.jp  
Jonathan, Kennedy, jkenney3@google.es  
Joe, Harrison, jharrison4@wiley.com

the Observable object that was created in the service.

- The screen shot at right shows how the data will look when displayed.

## 12.10 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
)
 }
}
```

## 12.11 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().

```
let o = this.http.get("https://localhost/posts/1")

o.subscribe(
```

```
(result) => console.log(result),
(error:HttpErrorResponse) => console.log(error)
)
```

## 12.12 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) => {...})
```

## 12.13 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) => {...})
```

## 12.14 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) => {...})
```

## 12.15 Summary

In this chapter we covered:

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



## Chapter 13 - Pipes and Data Formatting

---

### *Objectives*

Key objectives of this chapter

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

### 13.1 What are Pipes?

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

```
@Component({
 selector: 'app-register',
 template: `Salary: {{salary | currency}}`
})
export class RegisterComponent {
 salary = 1023.10293
}
```

### 13.2 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.

## 13.3 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, 10, 31);
{{birthday|date:"M-dd-yyyy"}}
outputs: "10-31-1993"
```

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

### Notes:

Full information on the DatePipe class is available here:

<https://angular.io/api/common/DatePipe>

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

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

@Component({
 selector: 'pipeinhtml',
 template: `

Pipe In HTML

<p>Data:
{{birthday}}</p>
<p>Formatted:
{{birthday|date:"M-dd-yyyy"}}</p>`,
})
```

```
export class PipeInHTMLComponent {
 birthday: Date = new Date(1993, 10, 31);
}
```

## 13.4 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(93,10,31);`
- Example-Pipes:

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

## 13.5 Using Pipes in Code

- The syntax for using a pipe to output data in HTML:  
`new PipeClass().transform(value, parm1, parm2,...);`
- Example code in component:

```
constructor(private datePipe: DatePipe){}
birthday: Date = new Date(1993, 10, 31);
birthdayFmt: string = this.datePipe
 .transform(this.birthday, "M-dd-yyyy");
```

  - ◇ Here the PipeClass is: "DatePipe"
  - ◇ Parm1 is: "M-dd-yyyy"
- Import of DatePipe is required:

```
import { DatePipe } from '@angular/common';
providers: [DatePipe],
```

## Notes

Full code example (pipe.injs.component.ts):

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

@Component({
 selector: 'pipeinjs',
 providers: [DatePipe],
 template: `<h3>Pipe In TypeScript</h3>
<p>Data:
{{birthday}}</p>
<p>Formatted:
{{birthdayFmt}}</p>`,
})

export class PipeInJSComponent {
 constructor(private datePipe: DatePipe){}
 birthday: Date = new Date(93, 10, 31);
 birthdayFmt: string = this.datePipe.transform(this.birthday, "M-dd-yyyy");
}
```

View Output:

### Pipe In TypeScript

Data: Sun Oct 31 1993 00:00:00 GMT-0400 (Eastern Daylight Time)

Formatted: 10-31-1993

Note the following:

DatePipe is imported, added to the providers array and injected into the constructor

The first parameter of transform() is the data being formatted

The second parameter is a date format string

Further note that while DatePipe is fully available in HTML templates without any importing it will not work in TypeScript without the above import statement.

## 13.6 Internationalized Pipes (i18n)

- Pipes like date, currency and number are sensitive to the locale.
- You can set the locale for the whole application:

```
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' }}
```

### Notes:

The code here shows how to set a new default LOCALE\_ID for the application:

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

// setup for internationalized pipes
import { registerLocaleData } from '@angular/common';
import localeFrCa from '@angular/common/locales/fr-CA';

// register french Canadian locale
registerLocaleData(localeFrCa, 'fr-CA');

@NgModule({
 declarations: [AppComponent],
 imports: [BrowserModule],
 providers: [
 HttpClient,
 { provide: LOCALE_ID, useValue: 'fr-CA' }
],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Code here shows how to import and register various locales so that they can be used to override the LOCALE\_ID as the last parameter of an i18n pipe.

```
// setup for internationalized pipes
import { Component, OnInit } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localeFrCa from '@angular/common/locales/fr-CA';
import localeDe from '@angular/common/locales/de';

@Component({
 selector: 'app-pipes',
 template: `
 <h4>Internationalized Date Pipes</h4>
```

```
 today: {{ today }}

 today en-US: {{ today | date: 'full':'+0600':'en-US' }}

 today fr-CA: {{ today | date: 'full':'+0600':'fr-CA' }}

 today de: {{ today | date: 'full':'':'de' }}

`,
 templateUrl: './pipes.component.html'
 })
}
export class PipesComponent implements OnInit {
 today: Date = new Date();
 ngOnInit(): void {
 registerLocaleData(localeFrCa);
 registerLocaleData(localeDe);
 }
}
```

### 13.7 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 localeFr from '@angular/common/locales/fr';
import localeEs from '@angular/common/locales/es';

registerLocaleData(localeFr);
registerLocaleData(localeEs);
```

### 13.8 Decimal 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");
}
```

```
}
```

## 13.9 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
<code>{{ 25   currency: "USD": 'code': "1.0-2"}}</code>	USD25
<code>{{ 25   currency: "CAD": 'symbol': "1.2-2"}}</code>	CA\$25.00
<code>{{ 25   currency: "USD": 'symbol': "2.2-2"}}</code>	\$25.00
<code>{{ salary   currency: "CAD"}}</code>	CAD65,000.00
<code>{{ price   currency: "EUR": 'symbol': "1.2-2"}}</code>	€9.95
<code>{{ price   currency: "USD": 'symbol': "1.2-2"}}</code>	\$9.95
<code>{{65.243   currency: "USD": 'symbol': "3.2-2"}}</code>	\$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");
}
```

### 13.10 Custom Pipes

- To create a custom Pipe start with an empty file and:

- ◇ Import the Angular Pipe class:

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

- ◇ Export the class

```
export class MyPipe implements PipeTransform{}
```

- ◇ Add the @Pipe annotation and name your pipe

```
@Pipe({ name: "mypipe" })
```

- ◇ Implement a method named "transform":

```
transform(data){
 // modify the data
 return modified_data;
}
```

```
}
```

- To use custom pipes in a component, you need to import the pipe and add it to the 'declarations' array in `@NgModule` for the application module

### 13.11 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"`

### 13.12 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
```

### 13.13 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'">

```

### 13.14 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: `<h3>Array Filter Pipe: Example</h3>
 <input type="text" [(ngModel)]="searchTerm">
 <h4>Cars</h4>
 <li *ngFor="let x of cars|arrayfilter:searchTerm">
 {{x}}

 <h4>Pets</h4>
 <li *ngFor="let x of petsFiltered">
 {{x}}`
})

export class ArrayFilterComponent {

 constructor(private filterPipe: ArrayFilterPipe){}

 searchTerm: string = "";

 cars: string[] = ["BMW", "Chevy",
 "Honda", "Subaru", "Ford",
 "Mazda", "Hyundai"];

 pets: string[] = ["Cat", "Dog",
 "Hamster", "Rabbit", "Fish",
 "Bird", "Turtle"];

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

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);
```

## 13.15 A Sort Pipe

- Example Pipe that **sorts** an array in ascending order:

```
import { Pipe, PipeTransform } from '@angular/core';
import { PipesComponent } from '../pipes.component';

@Pipe({ name: 'arraysort' })
export class ArraySortPipe implements PipeTransform{
 transform(array_instance) {
 return array_instance.sort();
 }
}
```

- This Pipe:

- ◇ Includes a single `transform()` method
- ◇ Uses the JavaScript array `sort()` method
- ◇ Is invoked using the name "arraysort"

```
<li *ngFor="let x of array_data | arraysort">
```

### Notes

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

```
import { Component } from '@angular/core';
import { ArraySortPipe } from '../array.sort.pipe';

@Component({
 selector: 'arraysort',
 providers: [ArraySortPipe],
 template: `<h3>Array Sort Pipe: Example</h3>
 <h4>Pets Unsorted</h4>
 <li *ngFor="let x of pets">
 {{x}}

 <h4>Pets Sorted in Template</h4>
 <li *ngFor="let x of pets2|arraysort">
 {{x}}
 <h4>Pets Sorted in JavaScript</h4>
 <li *ngFor="let x of petsSorted()">
 {{x}}
 `
})

export class ArraySortComponent {
 constructor(private sortPipe: ArraySortPipe){}
}
```

```
 pets: string[] = ["Turtle", "Cat", "Rabbit",
 "Hamster", "Dog", "Fish", "Bird",];

 pets2: string[] = this.pets.slice(0);
 pets3: string[] = this.pets.slice(0);

 petsSorted(){ return this.sortPipe.transform(this.pets3);} } }
```

### 13.16 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
```

## 13.17 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.
- 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.

Pets Unfiltered

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

Favorite Pets

Pet	Favorite
Turtle	<input checked="" type="checkbox"/>

### 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 },
];
}
```



## 13.18 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**

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

**Favorite Pets**

Pet	Favorite
Turtle	<input checked="" type="checkbox"/>
Rabbit	<input checked="" 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
})
```

## 13.19 Summary

In this chapter we covered:

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



## Chapter 14 - 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

### 14.1 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

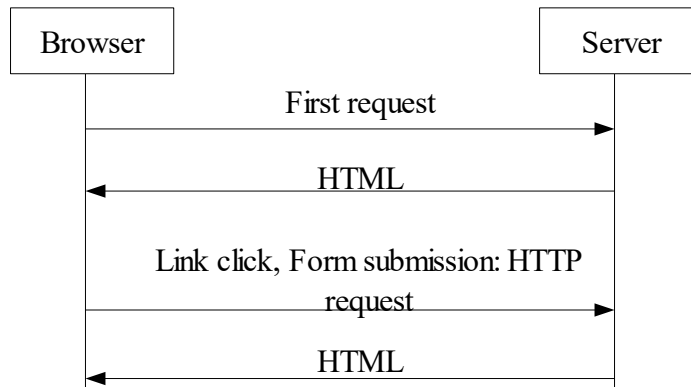
An SPA loads a single HTML document from the server and then selectively shows different DOM elements to create the illusion of multiple page navigation.

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.

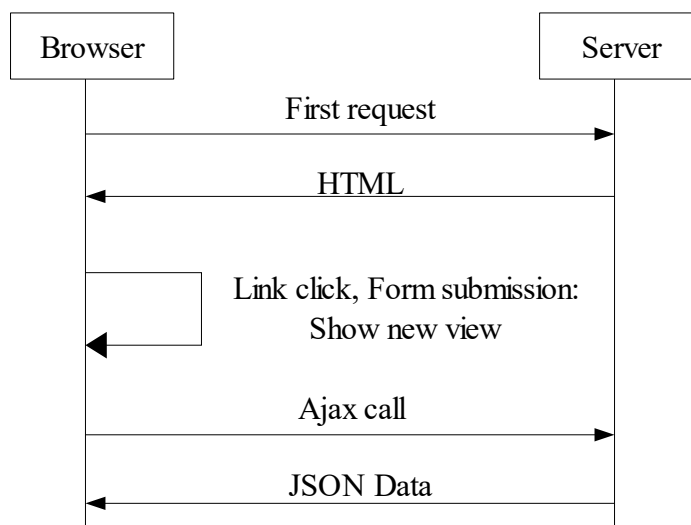
Frameworks like Angular leverage HTML5 and simplify the task of developing complex SPA.

## 14.2 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.

## 14.3 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.

## 14.4 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.

## 14.5 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**.

## 14.6 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.

### 14.7 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.

## **14.8 Summary**

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



## Chapter 15 - The Angular Component Router

---

### ***Objectives***

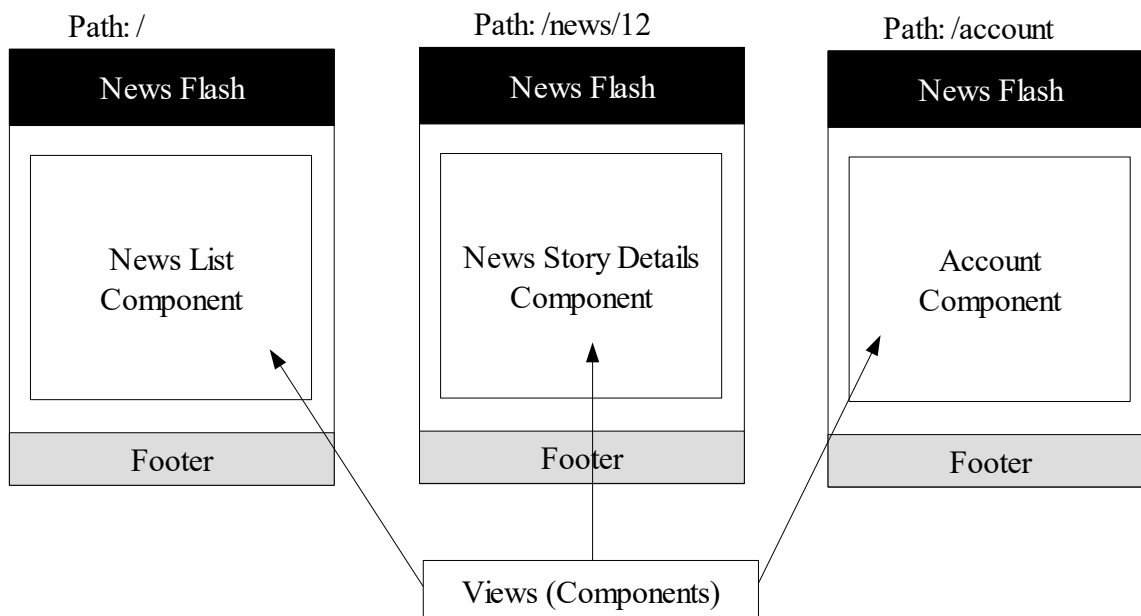
Key objectives of this chapter

- 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

### **15.1 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

## 15.2 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.

## 15.3 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.

## 15.4 Creating a Router Enabled Application

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

```
ng new router-test --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.

## 15.5 Hosting the Routed Component

- The <router-outlet> tag is used to show the currently routed component. This is usually done from the application root component.

```
@Component({
 selector: 'app',
 template: `<h3>News Flash</h3><nav>
<a href [routerLink]="['/']" >Home
<a href [routerLink]="['/account']" >Account

<router-outlet></router-outlet>`
})
export class AppComponent {}
```

## 15.6 Navigation Using Links and Buttons

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

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

## 15.7 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']);
 }
}
```

## 15.8 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.

## 15.9 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]);
```

## 15.10 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 PersonDetailComponent implements OnInit {
 constructor(private route: ActivatedRoute) {}
 ...
}
```

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

## 15.11 Retrieving the Route Parameter Synchronously

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

```
ngOnInit() {
 let index = this.route.snapshot.params['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.

## 15.12 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 any change to the route parameters.
- Add the following code in `ngOnInit()`:

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

- Here `route.params` is an `Observable`. To retrieve data a callback method needs to be passed to its `subscribe()` method.

## 15.13 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
```

### 15.14 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.

### 15.15 Retrieving Query Parameters Asynchronously

- The view component being navigated to needs to retrieve the query parameters before it can use them.
- Query parameters are retrieved from the **routerState** property of the **ActivatedRoute** object.

```
this.route.routerState.queryParams
```



```
.subscribe(params => {
 let id:number = +params['id']; //Convert to number
 let name:string = params['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.

## 15.16 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 book mark it? Entering the same URL directly in the browser's address bar will results in a 404 "file not found" error from the server. That is because the web server does not know anything about the 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>.

## 15.17 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 16 - 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

### 16.1 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: {
 body: any;
 headers: HttpHeaders;
 observe: HttpObserve;
 params: HttpParams;
 reportProgress: boolean;
 responseType: 'arraybuffer' | 'blob' | 'json' | 'text';
 withCredentials: boolean
}
```

### 16.2 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}`);
 }
);
```

## 16.3 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 headers = new HttpHeaders({
 'X-custom-header': 'My stuff',
 'Authorization': 'my-auth-token'
})

//Make a call
this.http.get(`/app/person`, { 'headers': headers })
```

## 16.4 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(), create() and more.
  - ◇ Import them from the 'rxjs' module.

## 16.5 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.

## 16.6 The Observable.create() Method

- Creates an Observable that can asynchronously emit events.

```
import { Observable } from 'rxjs'
```

```
let o:Observable<number> = Observable.create((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.create()` 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.

## 16.7 Observable Operators

- Operators are functions that are used to manipulate the data emitted by Observables. For example, you can transform or filter the emitted data.
- Import them from the **'rxjs/operators'** module.
- An operator is just a regular function that returns an operator creator function. The operator creator function takes as input a source Observable and returns a new Observable.
- As an example see how the **map** operator is used to transform data.

```
import { Observable, of } from 'rxjs'
import { map } from 'rxjs/operators'

let o1 : Observable<number> = of(1, 2, 3)
let projectionFunction = function(val : number) : number {
 return val * val
}
let opCreatorFunction = map(projectionFunction)
let o2 : Observable<number> = opCreatorFunction(o1)

o2.subscribe(val => console.log(val)) //Prints: 1 4 9
```

## 16.8 More About map

- `map()` takes as input a function that does the data conversion. It's called a **projection function**. Most operators take a projection function that contains the business logic appropriate for the operator.
- The new Observable created by `map()` emit exactly the same number of events as the source. But the data type can be different.

```
let o1 : Observable<number> = of(1, 2, 3)
let opFunc = map((val:number) => `We got ${val}`)
opFunc(o1).subscribe(val => console.log(val))

//Prints strings
We got 1
We got 2
We got 3
```

## 16.9 Piping Operators

- Multiple operators can be chained together using the **Observable.pipe** method.
- The output observable from one operator is fed as input to the next operator and so on.
- `pipe()` returns the Observable created by the final operator.

```
let o1 : Observable<number> = of(1, 2, 3)
let o2 : Observable<string> = o1.pipe(
 map(val => val * val), //Square
 map(val => 2 * val), //Double
 map(val => `We got ${val}`) //To string
)
o2.subscribe(val => console.log(val))
//Prints
We got 2
We got 8
We got 18
```

## 16.10 The flatMap() Operator

- Same as map() except the projection 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
```

## 16.11 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 = 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
```

## 16.12 The zip() Operator

- 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.

## 16.13 Caching HTTP Response

- A service can cache response.

```
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.

### 16.14 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
})
```

### 16.15 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)
 })
```

## 16.16 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).

## 16.17 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
```

## 16.18 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.

## 16.19 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 17 - Consuming WebSockets Data in Angular

---

### *Objectives*

Key objectives of this chapter

- Web Sockets Overview
- Web Sockets Use Cases
- Web Sockets Servers
- The socket.io.client library
- Using socket.io.client with Angular
- Consuming Web Sockets data in Angular

### **17.1 Web Sockets Overview**

- Web Sockets provide an alternative to HTTP for communicating with back-end servers
- With HTTP communication is
  - ◇ 'connection-less'
  - ◇ Based on a request/response protocol
  - ◇ Always initiated by the client
- Web Sockets create and use client/server connections:
  - ◇ First, a connection is created between the client and server
  - ◇ Once the connection is created, either side can initiate a communication
  - ◇ Any number of communications can be sent in either direction at any time as long as the connection is maintained

### **17.2 Web Sockets Use Cases**

- The two-way (full-duplex) nature of Web Sockets allows server applications to 'push' data to web apps enabling applications such as:
  - ◇ Real-time dashboards

- ◇ Collaborative Editors
- ◇ Financial tickers
- ◇ Chat applications
- ◇ Sports/News updates
- ◇ Multiplayer games

### 17.3 Web Socket URLs

- Web Socket connections are initiated using a URL similar to those used for HTTP but with a schema of "**ws**" or "**wss**"

- ◇ Example:

```
ws://server.com:4000/wsserver
```

- ◇ Where:

- protocol is: "ws"
  - server is: "server.com"
  - port is: 4000
  - server app is: wsserver
- The following URL connects to a live sample Web Socket server implementation:

```
ws://echo.websocket.org/
```

### 17.4 Web Sockets Servers

- Web Socket Servers can be implemented using a variety of technologies including:
  - ◇ Java
  - ◇ PHP
  - ◇ Node.js (JavaScript)
  - ◇ C++
- Web Sockets servers are standards-based



- JavaScript clients can connect to any Web Socket server no matter what language the server is implemented in.
- Additional information about Web Sockets can be found at:

<http://websockets.org>

## 17.5 Web Socket Client

- Current browsers support the creation of Web Socket clients using the `WebSocket` object:

```
var socket = new WebSocket('ws://localhost:3000');
socket.onopen = function(event) {
 // Send an initial message
 socket.send('start sending...');
 // Listen for messages
 socket.onmessage = function(event) {
 // do something with
 // event data
 };
 // respond to disconnect
 socket.onclose = function(event) {
 console.log('ws disconnected');
 };
};
```

- The above JavaScript code implements a Web Socket client that sends an initial message and is then setup to receive multiple ongoing events from the Web Socket server.

## 17.6 The `socket.io-client` library

- The `Socket.IO` libraries provide a robust Web Socket API as an alternative to the standard JavaScript `WebSocket` object. It features:
  - ◇ Enhanced reliability
  - ◇ Auto-reconnection support
  - ◇ Disconnection detection
  - ◇ Support for binary messages

- Socket.IO comes in two parts
  - ◇ **socket.io** library for creating Web Socket server applications
  - ◇ **socket.io-client** for implementing Web Socket client applications
- Information and code examples for socket.io can be found at:
  - `http://socket.io/`
  - `https://github.com/socketio/socket.io`
  - `https://github.com/socketio/socket.io-client`
- Alternatives to socket.io available at [www.npmjs.com](http://www.npmjs.com) include:
  - ◇ `websocket.io`
  - ◇ `websocket`

## 17.7 Using socket.io-client in JavaScript

- A JavaScript Web Socket client using socket.io-client:

```
var socket = io('http://localhost:3000');
socket.emit('start sending...');
socket.on('connect', function(){
 console.log('ws connected');
});
socket.on('event', function(data){
 // do something with
 // data
});
socket.on('disconnect', function(){
 console.log('ws disconnected');
});
```
- On the following pages we will see how to use socket.io-client in Angular

## 17.8 Setting up socket.io-client in Angular Projects

- To use socket.io-client in angular we need to:
  - ◇ Install socket.io-client and add it to package.json:

```
npm install socket.io-client --save
```
  - ◇ Add typings so we can use it in typescript:

- ```
npm install @types/socket.io-client --save
```
- ◊ **Modify `systemjs.config.js` so the module loader can find it (*)**:
 - add to map: section

```
'socket.io-client': 'npm:socket.io-client',
```
 - add to packages: section:

```
'socket.io-client': {
  main: './dist/socket.io.js',
  defaultExtension: 'js'
}
```
 - ◊ **Import it at the top of our Angular service**:

```
import * as io from 'socket.io-client';
```

(*) is only required when using the `system.js` module loader

17.9 Using `socket.io-client` in an Angular service

- `websocket.service.ts`:

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { Subscriber } from 'rxjs/Subscriber';
import * as io from 'socket.io-client';
@Injectable()
export class WebsocketService {
  socket: SocketIOClient.Socket;
  url: string = 'ws://localhost:4000';
  getMessages() {
    let observable = new Observable(
      (observer: Subscriber<string>) => {
        this.socket = io(this.url);
        this.socket.emit('getquotes', 'start');
        this.socket.on('newquote', (data: string) => {
          observer.next(data);
        });
      });
    return observable;
  }
  stopMessages() { this.socket.disconnect(); }
```

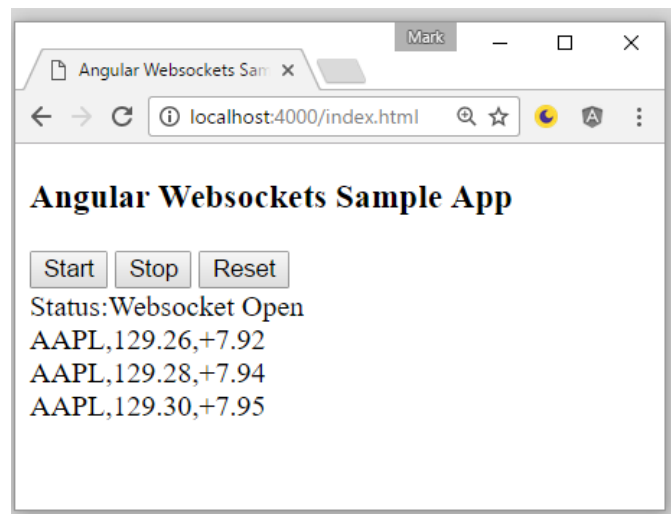
}

17.10 Angular websocket.service Notes:

- The code on the previous page:
 - ◇ Imports the rxjs Observable and Subscriber objects
 - ◇ Wraps the Web Socket client in an rxjs Observable object.
 - ◇ Any client calling the service's `getMessages()` method receives the Observable type object which it can use to retrieve data sent by the Web Socket server

17.11 The Angular Web Socket Client Sample App

- A sample app using the service just described is shown on the right.
- Data displayed on the screen is obtained from messages sent by the Web Sockets server
- The next few pages go over the component implementation.



17.12 Angular websocket.component.ts

- Data from the Angular service is consumed by an Angular Component.
- Method from `websocket.component.ts`:

```
startWebsocketCommunications() {  
    let obs: Observable<{}> =  
        this.websocketService.getMessages();  
}
```

```
        this.status = 'Websocket Open';
        obs.subscribe( (msg: string) => {
            console.log('msg: ' + msg);
            this.quotes.push(JSON.parse(msg));
        }
    );
}
```

- The code adds incoming data to the quotes array continually as long as the Web Socket is open.
- For this code to work we need to:
 - ◇ import and inject the websocket.service
 - ◇ import the Observable object

17.13 The Full websocket.component.ts code

```
import { Component } from '@angular/core';
import { WebSocketService } from
    './websocket/websocket.service';
import { Observable } from 'rxjs/Observable';

@Component({
    selector: 'my-app',
    template: `<h3>Angular Websockets Sample App</h3>
    <button (click)='startWebSocketCommunications()'>
        Start</button>
    <button (click)='stopWebSocketCommunications()'>
        Stop</button>
    <button (click)='resetMessages()'>Reset</button>
    <div>Status:{{status}}</div>
    <div *ngFor="let quote of quotes" >
        {{quote.ticker}},{{quote.price}},{{quote.change}}
    </div>
    `,
    providers: [ WebSocketService ]
})
```

17.14 The Full websocket.component.ts code

```
export class AppComponent {
  quotes: Array<{}> = [];
  status: string = 'Websocket Closed';
  name: string = 'Angular';
  url: string = 'ws://localhost:4000';

  constructor(private websocketService: WebsocketService )
  {}

  startWebsocketCommunications() {
    let obs: Observable<{}> =
      this.websocketService.getMessage();
    this.status = 'Websocket Open';
    obs.subscribe( (msg: string) => {
      console.log('msg: ' + msg);
      this.quotes.push(JSON.parse(msg));
    }
    );
  }
}
```

17.15 The Full websocket.component.ts code

```
stopWebsocketCommunications() {
  this.websocketService.stopMessages();
  this.status = 'Websocket Closed';
}

resetMessages() {
  if (this.status === 'Websocket Open') {
    this.stopWebsocketCommunications();
    this.quotes = [];
  } else {
    this.quotes = [];
  }
}
}
```

17.16 Implementation Modifications

- The sample application shown in this chapter is typical of a dashboard application that:
 - ◇ Makes an initial request
 - ◇ Receives continuous updates from the server
- The code used to setup the communication with the Web Sockets server can easily be modified to:
 - ◇ Post continuous data to a server
 - ◇ Receive and update the screen in various ways
- With changes such as these, the app can implement each of the various use cases mentioned previously.
- Web Socket client apps work closely with Web Socket server apps so changes on one side (server or client) may require changes to the other.

17.17 Summary

In this chapter we covered:

- WebSockets Overview
- WebSockets Use Cases
- WebSockets Servers
- The socket.io.client library
- Using socket.io.client with Angular
- Consuming Websockets data in Angular

Chapter 18 - 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

18.1 Routing Overview

- Angular applications use the Angular Router to implement navigation
- The Angular Router synchronizes view changes with the browser's address bar.
- Routes are created that link paths and components
- Navigating to a specific route causes the component specified in the route to be loaded into the page.
- The location to insert the component is dictated by the <router-outlet></router-outlet> element.
- Data can be passed to the component during navigation using:
 - ◇ Router Parameters
 - ◇ Query Parameters

18.2 Routing Enabled Project

- You can use Angular CLI to create a new project with routing enabled.

```
ng new my-app --routing
```

- This creates an additional module solely to define your routes. It is called **AppRoutingModule** and created in **app-routing.module.ts**. Example:

```
const routes: Routes = [
  {path: "", component: HomeComponent},
  {path: "catalog", component: CatalogComponent}
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

- The main application module imports this routing module.

Notes

Even though you can manually define the routes directly in the application module file it is recommended that you use the `--routing` option with Angular CLI to create the project. This way things are done in a consistent manner from project to project. Also as we will see next this is also the way a feature module can add to the global route table.

18.3 Routing Enabled Feature Module

- A feature module can also have view components. In that case it needs to be able to add these components to the application's route table.
- Use the `--routing` option with Angular CLI to create the feature module.

```
ng g module checkout --routing
```

- This creates the 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.

18.4 Using the Feature Module

- Simply import the feature module from the app module as usual. This will automatically install 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>
<a href [routerLink]="['/billing']">Billing</a>
```

- 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.

18.5 Lazy Loading the Feature Module

- 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:  
    "./checkout/checkout.module#CheckoutModule"}  
];
```

- ◇ Note the use of *loadChildren*. It points to the location of the feature module file relative to `app-routing.module.ts`.
- ◇ Also, note that we are adding a route for a module, not a component
- Now the view components of the feature module will be 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.

18.6 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>  
<a href [routerLink]="['/checkout/billing']">Billing</a>
```

18.7 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.

18.8 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</a>
```

- 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; }
```

18.9 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`.

18.10 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

18.11 redirectTo

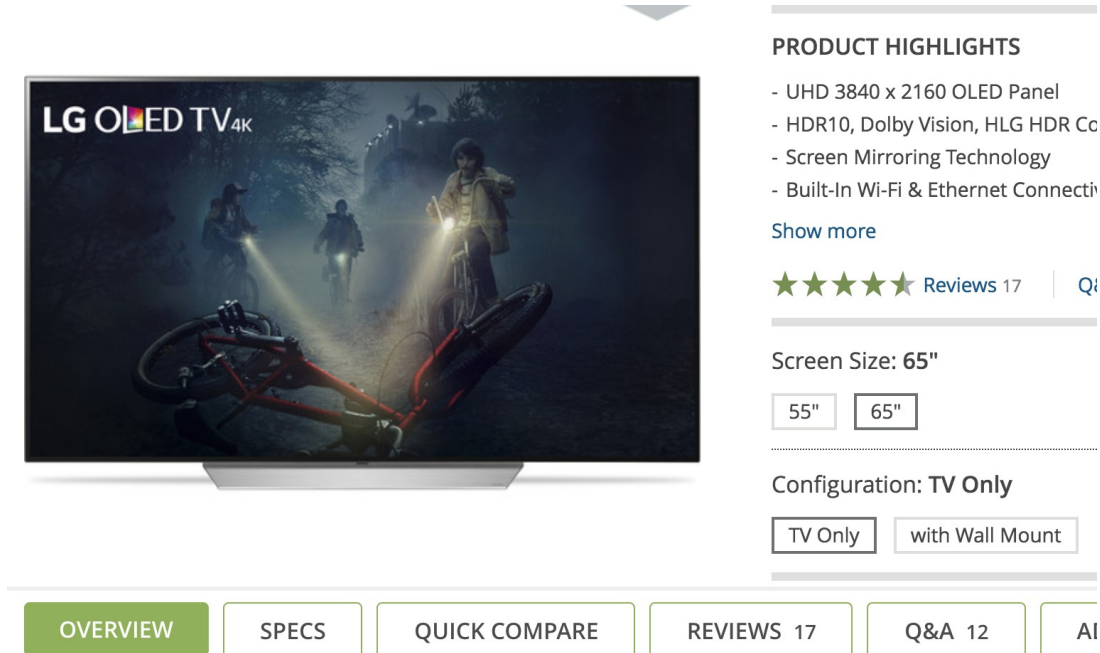
- **redirectTo** allows redirection of a given path to another existing route.
Example:

```
{path: 'home', component: HomeComponent},  
{path: '', redirectTo: '/home', pathMatch: 'full' }  
{path: 'overview', redirectTo: '/home', pathMatch: 'full' }  
{path: 'error', component: ErrorComponent},  
{path: '**', redirectTo: '/error', pathMatch: 'full' }
```

- The `pathMatch` property can be:
 - ◇ `'full'` - Path in URL must match fully for a redirection to take place. `"/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"`.

18.12 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



18.13 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.

18.14 <router-outlet> for Child Routes

- Components with child routes must include a <router-output> 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>
      <a [routerLink]="['reviews']">Reviews</a>
      <a [routerLink]="['specs']">Technical Specs</a>
    </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>
<a [routerLink]="['/about']" >About</a>
<a [routerLink]="['/products']" >Products</a>
<router-outlet></router-outlet>
```

18.15 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>
<a [routerLink]="['reviews']">Reviews</a>
```

- A child view component can have links to the parent using the "../" prefix.

```
<a [routerLink]="['../home']">Home</a>
```

18.16 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
- Guard interfaces available for use with the Angular Router include:
 - ◇ 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

18.17 Creating Guard Implementations

- Guards are created as services that implement one of the guard interfaces:

```
// auth.service.ts
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';

@Injectable()
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.

18.18 Using Guards in a Route

- The service implementing the Guard interface is imported and added to the providers array in the *app-routing.module.ts* file:

```
import { AuthService } from './auth.service';  
providers: [ AuthService ]
```

- Now it can be used in route definitions:

```
{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.

18.19 Route Animations

- When a route is activated the component currently tied to `<router-outlet>` is removed and replaced by the component specified in the new route.
- The new component's appearance and the old one's disappearance happen simultaneously.
- The change-over in components can be enhanced with Angular animations.

- Possible Animations include:
 - ◇ Sliding one component in and the other out,
 - ◇ Fading one component out and the other one in
 - ◇ etc.
- For more information about route animations see:
<https://angular.io/guide/route-animations>

18.20 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 19 - Introduction to 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

19.1 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.
- Unit tests should also be run after a build to verify the build.

19.2 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 specifications (test scripts) in multiple browsers like Firefox and Chrome
- ◊ 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 as there is not as much low-level integration required as with Jasmine
- **Angular testing utilities** - Angular provides API and tools to write unit tests.

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>

19.3 Testing Setup

- Create a regular project to set it up for testing.

```
ng new simple-app
```

- That does these things:
 - ◊ It adds Karma and Jasmine dependencies in package.json.
 - ◊ Generates the Karma config file **karma.conf.js**.
 - ◊ It configures **.angular-cli.json** file so that a test script file is created by Angular CLI with every new component, service etc.
- A minimal project created using --minimal flag does not have support for

testing. Do not use it for real projects.

- By default, Karma is configured to use the Chrome browser which must be installed.

Testing Setup

This chapter will focus on using the Angular project setup (or "quickstart") to setup for testing although Angular CLI setup should be similar.

The details of some of the configuration files above are not provided but can be found in the documentation of the relevant framework. This slide is mainly to be familiar with some of the files important in setting up testing.

There is also a 'protractor.config.js' file for Protractor configuration although that is not covered here.

Karma can integrate with other frameworks besides Jasmine but the Angular testing configuration uses this combination.

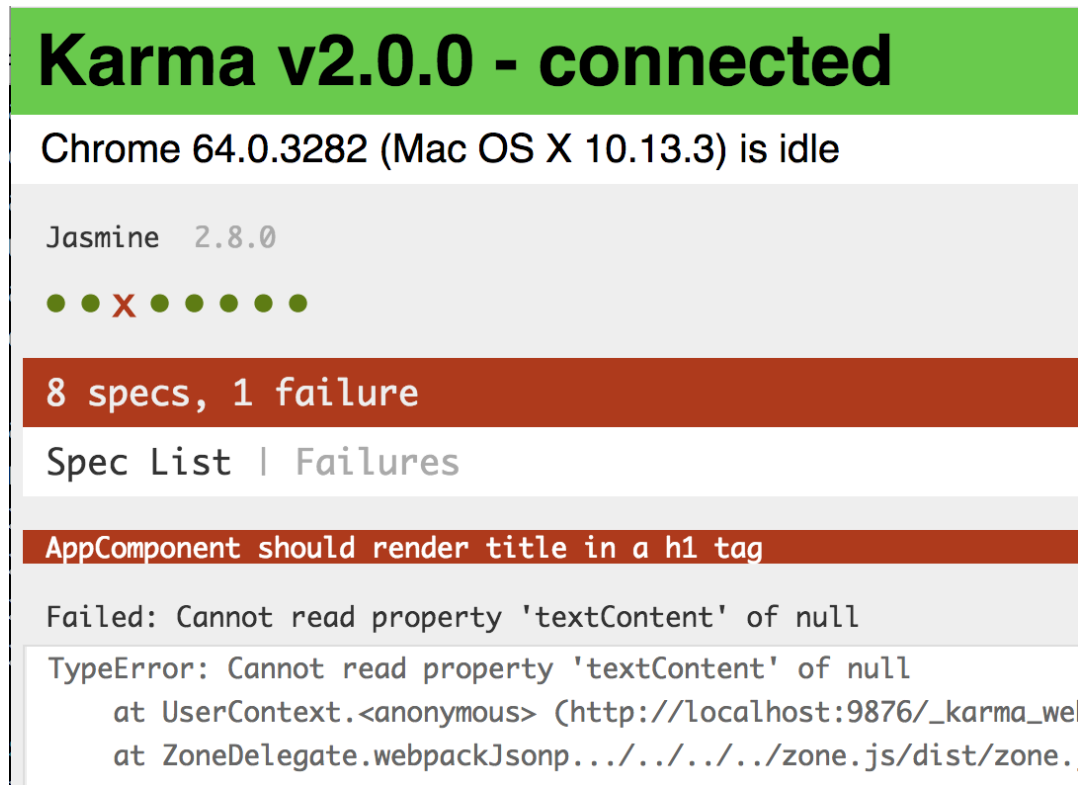
19.4 Typical Testing Steps

- 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.
 - ◇ Run all test scripts in the project.
 - ◇ Show the test result along with the application in a browser.
- 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.

19.5 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

19.6 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.

19.7 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)  
  })  
})
```

19.8 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.

19.9 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

19.10 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);
```

19.11 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);
```

19.12 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()*.

- The *beforeEach* function runs the common initialization code (if needed).
- The *afterEach* function contains the clean up code (if needed).
- Both functions take a function as a parameter.
- You can have multiple *beforeEach()* and *afterEach()* functions.

19.13 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.

19.14 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.

19.15 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],
      providers: [BlogService]
    });
  });
  //...
})
```

- Here we are testing BlogService which depends on HttpClient. Our test module had to be set up for that to work.

19.16 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}`)
```

```
}  
}
```

19.17 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.get()`**.

```
describe('BlogService', () => {  
  let service: BlogService  
  
  beforeEach(() => {  
    TestBed.configureTestingModule({...});  
  
    service = TestBed.get(BlogService)  
  })  
  
  it("Should use injection", () => {  
    expect(service).toBeTruthy()  
  })  
})
```

19.18 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}`)  
})
```

19.19 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()
  })
})
```

19.20 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]
  });
});
```

19.21 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.get(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.

19.22 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>
```

19.23 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(() => {
  TestBed.configureTestingModule({
    imports: [FormsModule, ...],
    providers: [...],
    declarations: [ GreetComponent ]
  })
  .compileComponents();
}));
```

19.24 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();
  });
})
```

19.25 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.

19.26 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.

19.27 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.

19.28 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');
});
```

19.29 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 20 - Debugging

Objectives

Key objectives of this chapter

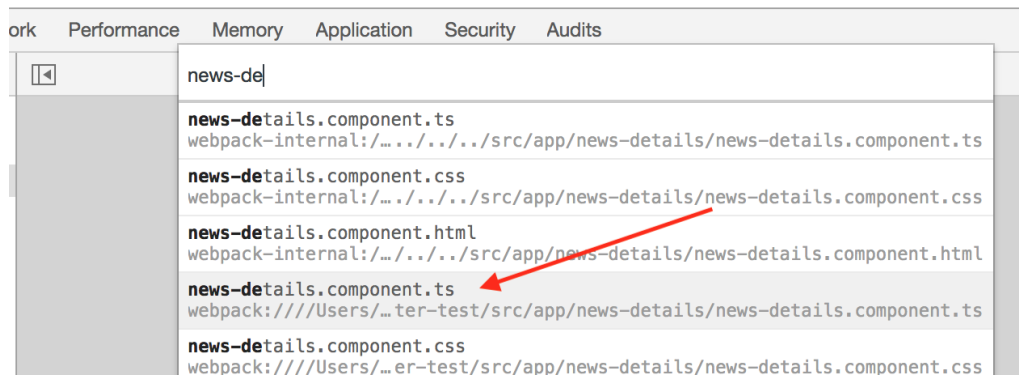
- Basic Debugging Practices
- Inspecting Components with `ng.probe()`
- Typescript Breakpoints
- The Augury Debugging Tool
- Common Exceptions

20.1 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
 - ◇ Augury - 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 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

20.2 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. Choose the **webpack:///** version and not the **webpack-internal:/** version.



- You can now put a breakpoint in the TS code and debug it in a normal fashion

20.3 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 }
```

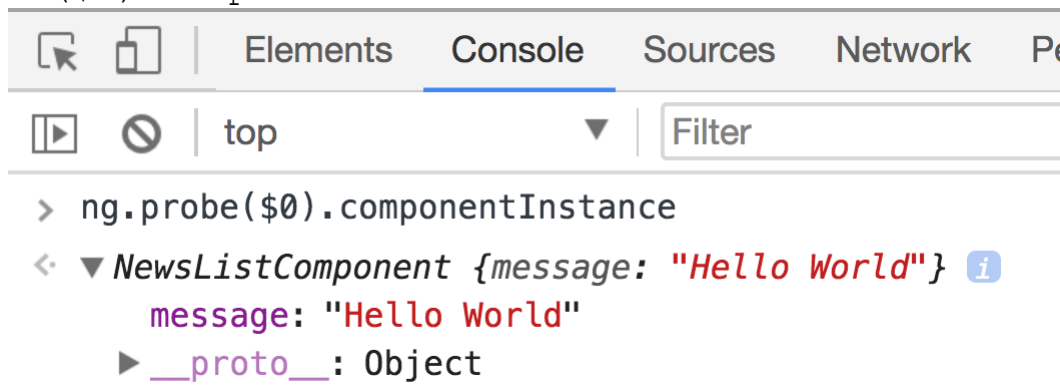
20.4 Inspecting Components

- In Chrome developer tool's "Elements" tab clicking on the  button and then click on an element on the browser screen.
- In the JavaScript console enter this. This will dump the DebugElement for the selected DOM element:

```
ng.probe($0)
```

- This will dump the component instance. You can then view the member variables of the instance.

```
ng.probe($0).componentInstance
```



20.5 Saving ng.probe Component References

- Component references accessed via ng.probe can be saved in variables:

```
var myComp = ng.probe($0);
```



```
myComp.message
```

 - returns the value of message

```
myComp.doSomething();
```

 - executes doSomething()- The variable retains the Component reference even after another component or element is selected.
- This makes the component reference easier to work with

20.6 Modifying Values using Component References

- Assuming that you've saved a component reference to a variable in the JavaScript console you might be tempted to try and update properties

using it like this:

```
myComp.message = "new message";
```

- You might expect this to change the text on the screen that was output using `{{message}}` but it doesn't. At least not right away.
- That's because Angular change detection needs to run in order for the text on the screen to be updated.
- Change detection can be triggered manually by clicking in an input field on the screen and then tabbing out of it.

Modifying Values using Component References

In previous versions of Angular, change detection could be called directly from the JavaScript console with the following command. This seems to have been removed in Angular 4.


```
myComp._debugInfo._view.changeDetectorRef.detectChanges();
```

20.7 Debug Logging

- Use `console.log()` or `console.error()` to print data to help with debugging.
- Use a third party logger like **ngx-logger** for more features. For example, it can POST log messages to a server.

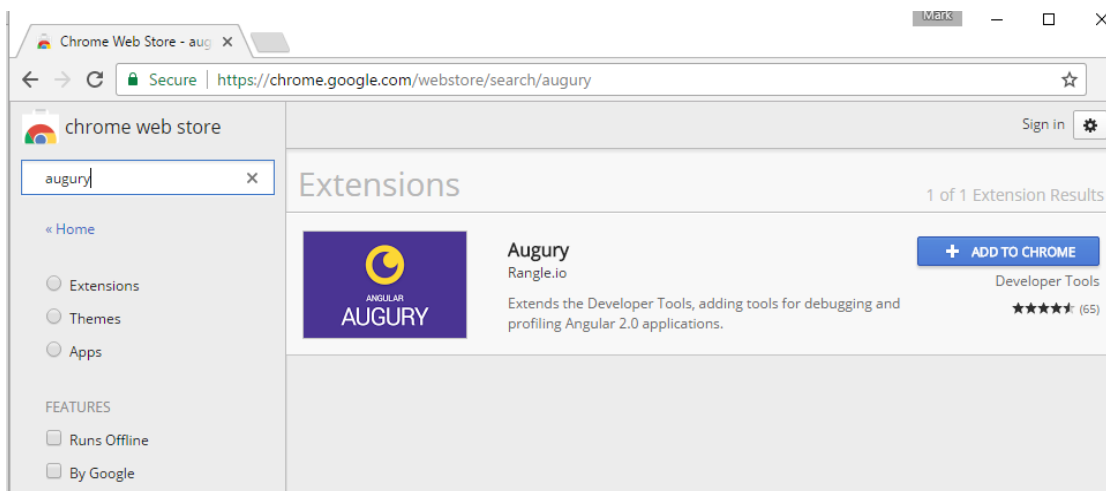
20.8 What is Augury?

- Augury extends the Developer Tools in the Chrome browser to allow debugging of Angular applications.
- Batarang is a similar extension that is used to debug AngularJS (as opposed to Angular) applications
- Augury appears as a separate tab in Chrome's developer tools.
- It includes views showing the details of the following for the currently loaded Angular application:
 - ◇ Components
 - ◇ Router
 - ◇ Modules

- When the Augury extension is loaded the following icon appears next to the browser's address box 

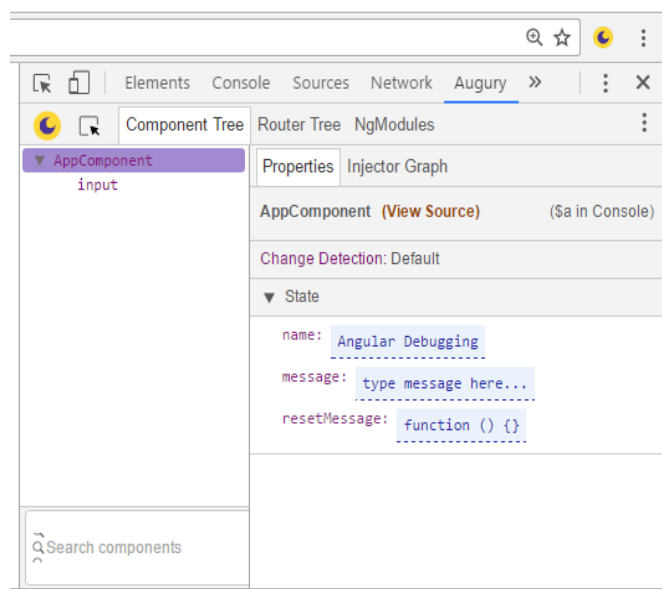
20.9 Installing Augury

- Augury is installed via the Chrome WebStore. Use the following link:
`https://chrome.google.com/webstore`
- Type "Augury" into the search box and hit enter to show the extension
- Click on the "add to chrome" button



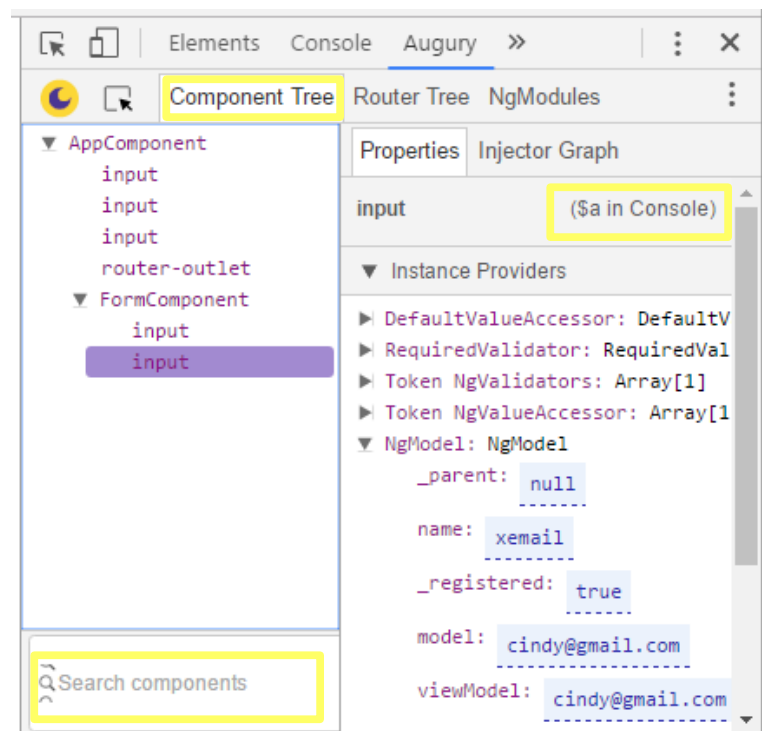
20.10 Opening Augury

- To open Augury first open the Chrome Developer Tools (F12)
- Click on the tab titled "Augury". Depending on the width of the screen it may be hidden. If so you will need to click on ">>" first.



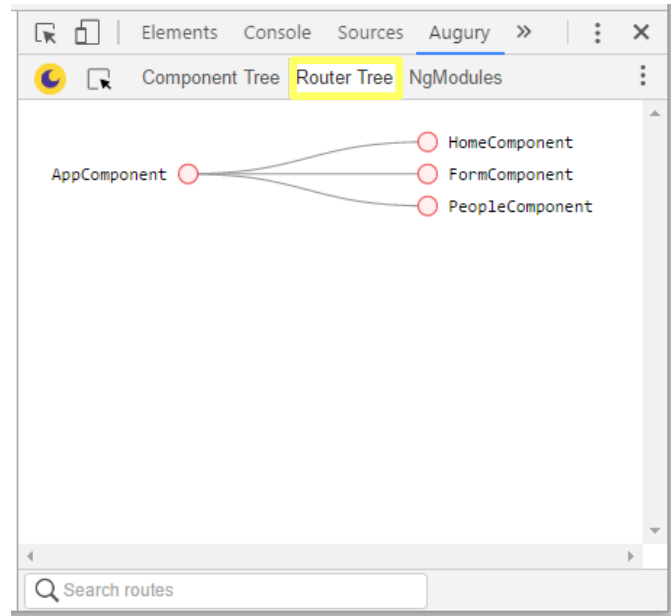
20.11 Augury - Component Tree

- The **component tree** allows you to browse through a view's Angular components and view Angular related properties
- For views with many components you can search to find a specific component
- Once a component has been selected you can also access it in the console using the variable **\$a**
- Information about injectors in-use for each component is also available



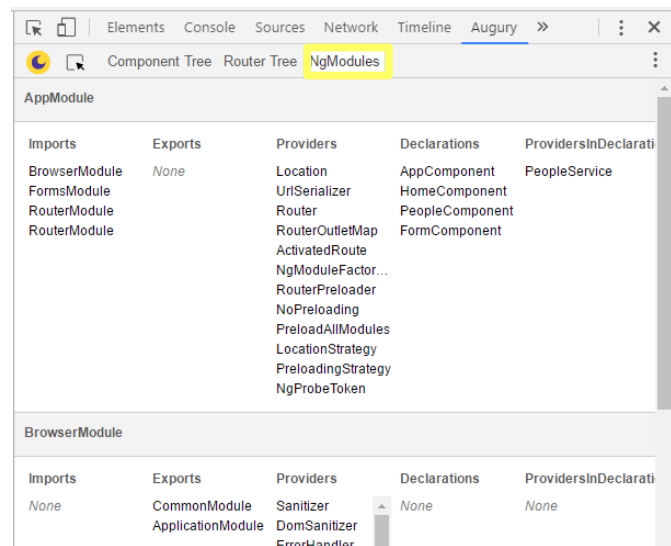
20.12 Augury - Router Tree.

- The **Router Tree** allows you to inspect routes that have been setup for the application and see
- Hovering over a route reveals more information:
 - ◇ path: /people
 - ◇ handler: PeopleComponent
 - ◇ data:
- For applications with a large number of routes search is available



20.13 Augury - NgModules Tab

- The **NgModules** tab displays the contents of each module's various arrays:
 - ◇ Imports
 - ◇ Exports
 - ◇ Providers
 - ◇ etc.
- The NgModules tab makes it easy to check the availability of services and components required by one of the modules components



20.14 Common Exceptions

- On the following pages, we review a few common exceptions.
- While the actual error messages in these cases may seem cryptic the solutions we will see are fairly easy:
 - ◇ 'Cant' bind to ngModel'
 - ◇ 'router-outlet not a known element'
 - ◇ 'No provider for Router!'
 - ◇ 'routerLink isn't a known property of input'

20.15 Common Exceptions: 'No such file: package.json'

- Exception reported at the command prompt:

```
ENOENT: no such file or directory, open
'C:\LabWork\package.json'
This is most likely not a problem with npm itself
and is related to npm not being able to find a file.
```
- This can happen when you run the npm command from the wrong directory.
- npm command like 'npm start' should be run from the root of your project where the project's package.json is located.
- To fix this navigate into the proper directory and retry the command.

20.16 Common Exceptions: 'Cant bind to ngModel'

- Exception:

```
Unhandled Promise rejection: Template parse errors:
Can't bind to 'ngModel' since it isn't a known
property of 'input'. ("<h1>Hello {{name}}</h1>
  <input type=text [ERROR ->][(ngModel)]= "message" />
```
- This means Angular cannot recognize the ngModel directive
- To fix this add the FormsModule to app.module.ts:

- ◇ import the FormsModule

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

- ◇ Add FormsModule to the imports array

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

20.17 Common Exceptions: 'router-outlet not a known element'

- Exception:

```
"'router-outlet' is not a known element:"
```

```
AND
```

```
"'routerLink' isn't a known property of 'input'"
```

- In these cases the router directives(<router-outlet> or routerLink) were added to the template without first adding the RouterModule.
- To fix add the following to app.module.ts:

- ◇ Import RouterModule

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

- ◇ Add the RouterModule to imports array:

```
imports:[ BrowserModule, RouterModule, ... ],
```

20.18 Common Exceptions: 'No provider for Router!'

- Exception:

```
"No provider for Router!"
```

- In this case, the routes table was not set up or set up incorrectly.
 - To fix:
 - ◇ Create an app.routes.ts file that exports an array of routes as ROUTES
 - ◇ Import ROUTES to app.module.ts
- ```
import {ROUTES} from './app.routes';
```
- ◇ Add the following to the imports array in app.module.ts
- ```
imports: [ ... , RouterModule.forRoot(ROUTES) ]
```

20.19 Summary

In this chapter we covered:

- Basic Debugging Practices
- Development (Debug) Mode
- Inspecting Components with `ng.probe()`
- Typescript Breakpoints
- The Augury Debugging Tool
- Common Exceptions