



Angular 7.0

For Beginners



Author

Debasis Saha



C#Corner

Table of Contents

01. Basics of Angular.....	5
Introduction	5
History of Angular	5
Why Angular Is Called a Framework	6
Advantages of Angular	6
Introduction to Angular 7	7
What's New in Angular 7	7
Summary	8
02. Environment Setup for Angular	9
Perquisites for Angular 7	9
Install Node.js	9
Install Angular CLI	10
Install TypeScript	11
Install Code Editor	11
Summary	12
03. Create First Angular Application	13
Some Essential Angular CLI Commands	13
Angular Solution Structure	14
tsconfig.json	15
package.json	15
main.ts	17
app.module.ts (Angular Module)	17
app.component.ts (Angular Component)	18
Index.html	18
Run Application	19
Summary	19
04. Components	20
What Is a Component?	20
Advantages of Component-Based Architecture	20
@Component Metadata	22
Component Life Cycle Events	23

Nested Component	24
Summary	26
05. Data Binding	27
What Is Data Binding?	27
Concept of Data Binding	27
Types of Data Binding	28
Interpolation	28
Property Binding	29
Two-Way Binding	30
Event Binding	31
@Input() Decorator	32
@Output() Decorator	33
Summary	35
06. Directives	36
What Is a Directive?	36
Comparison between Component & Directives	36
Attributes of Directives	37
ngClass	38
ngStyle	38
Structural Directives	39
ngIf	39
ngFor	40
ngSwitch	41
Summary	43
07. Pipes	44
What Is a Pipe?	44
Why Use Pipes?	44
Uses of Pipes	45
Types of Pipes	45
Filters vs Pipes	45
Built-In Pipes	45
How to Create Custom Pipes	47
Summary	48

08. Service	49
What Is a Service?	49
Benefits of Using Angular Services	49
How to Create a Service	50
@Injectable	50
Dependency Injection in Angular	51
What Is a Provider?	52
Summary	53
09. Ajax Request Handling	54
What's New in HTTP Service	54
Observables vs Promises	54
How to Define Observables	55
Specifications of Observables	55
Observables Array Operators	56
HTTP Post	57
What's New in HTTP Service	57
Summary	60
10. Routing	61
Why Routing?	61
Route Definition Objects	61
Router Module	62
Redirect Route to Another Route	62
Router Link	62
Adding Routing Component Dynamically	63
Nested Child Routes	63
Summary	64

01. Basics of Angular

Introduction

Angular (previously known as Angular JS) is an open source, JavaScript-based web application development framework. Currently, Angular is one of the best frameworks for developing SPA (Single Page Applications). The definition of Angular according to its official documentation is as follows:

“Angular is a structural framework for dynamic web applications. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application components clearly and succinctly. Its data binding and dependency injection eliminate much of the code you currently have to write. And it all happens within the browser, making it an ideal partner with any server technology.” – (Angular Conference, 2014)

So, the most common question before starting Angular is “What is Angular?” In very common words, the answer is:

- An MVC-based structured framework
- A SPA (Single Page Application)-based framework
- A client-side template
- A scripting language where code can be easily tested by unit testing

History of Angular

AngularJS was first created by two developers, Miško Hevery and Adam Abrons, as a side project in 2009. They created a project called **GetAngular**, which is an end-to-end tool that allows web designers or developers to interact with both the front end and back end. Afterwards, Google took this project, renamed it **AngularJS**, and started to build the entire framework. Then Google released the first API version 1.0 of AngularJS in May 2011. But from the 2.0 version, Google renamed the project from **AngularJS** to **Angular**. The table below demonstrates the release history of Angular.

Angular Version	Release Date
May 2011	Angular JS 1.0
October 2014	Angular 2.0
December 2016	Angular 4.0
November 2017	Angular 5.0
May 2018	Angular 6.0
October 2018	Angular 7.0

Why Angular Is Called a Framework

Before exploring Angular in depth, let us consider exactly what Angular is. What do we mean by a “framework,” and why would we want to use one? The dictionary definition tells us that a framework is “an essential supporting structure.” That sums up Angular very nicely, although Angular is much more than that. Angular is a large and helpful community—an ecosystem in which you can find new tools and utilities, an ingenious way of solving common problems, and, for many, a new and refreshing way of thinking about application structure and design. We could, if we wanted to, make life harder for ourselves by writing our own framework.

Realistically, for most of us, this just isn’t possible. It almost goes without saying that you need the support of some kind of framework, and that this framework almost certainly should be something other than your own undocumented (or less than well understood) ideas and thoughts about how things should be done. A good framework, such as Angular, is already well tested and well understood by others. Keep in mind that others may inherit your code, be on your team, or otherwise need to benefit from the structure and support a framework provides.

Advantages of Angular

Angular is an open-source web application framework maintained by Google and a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. The library works by first reading the HTML page, which has additional custom tag attributes embedded into it. Those attributes are interpreted as directives that tell Angular to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code or retrieved from static or dynamic JSON resources.

The main advantages of Angular are:

- There is no need to use observable functions. Angular analyzes the page DOM and builds the bindings based on the Angular-specific element attributes. That requires less writing, meaning the code is cleaner, easier to understand, and has fewer errors.
- Angular modifies the page DOM directly instead of adding inner HTML code. That is faster.
- Data binding does not occur on each control or value change (no change listeners) but at particular points of the JavaScript code execution. That dramatically improves performance, as a single bulk Model/View update replaces hundreds of cascading data change events.
- There are quite a number of different ways to do the same things, thus accommodating to particular development styles and tasks.
- Extended features such as dependency injection, routing, animations, view orchestration, and more are available.
- It is supported by IntelliJ IDEA and Visual Studio .NET IDEs.
- It is supported by Google and a great development community.
- AngularJS is more intuitive, as it makes use of HTML as a declarative language. Moreover, it is less brittle for reorganizing.

- AngularJS is a comprehensive solution for rapid front-end development. It does not need any other plugins or frameworks. Moreover, there are a range of other features that include restful actions, data building, dependency injection, enterprise-level testing, etc.
- AngularJS is unit testing ready, and that is one of its most compelling advantages.

Introduction to Angular 7

So, after several releases, finally in October 2018 Google released the new version of Angular i.e. Angular 7. Angular 7 is basically an upgraded version over previous Angular version i.e. Angular 6 or Angular 5. Although, Angular 7 comes with many new features, significant changes, and updates on the existing features. The main improvement in this version is to Angular Material SDK and Angular CLI. Also, Google introduced some new features like CLI prompts, Scrolling, and Virtual Drag and Drop within the tool chain.

What's New in Angular 7

Angular 7 or the Angular (i.e. version 2 onward) is a TypeScript-based open-source front-end web application platform led by the Angular Team at Google. Angular is a complete rewrite from the same team that built AngularJS. It is necessary to be clear that Angular is completely different from AngularJS. Let us understand the differences between Angular and AngularJS:

- The architectural framework of an Angular application is different from AngularJS. The main building blocks for Angular are modules, components, templates, metadata, data binding, directives, services, and dependency injection.
- Angular was a complete rewrite of AngularJS.
- Angular does not have a concept of “scope” or controllers. Instead, it uses a hierarchy of components as its main architectural concept.
- Angular has a simpler expression syntax, focusing on “[]” for property binding, and “()” for event binding.
- **Mobile development** – Desktop development is much easier when mobile performance issues are handled first. Thus, Angular first handles mobile development. The new Angular version will be focused on the development of mobile apps.
- **Modularity** – Angular follows modularity. Similar functionalities are kept together in the same modules. This gives Angular a lighter and faster core. Various modules from previous versions of AngularJS have been removed from Angular’s Core for better performance.
- Angular 7 will target ES6.0 and almost all modern browsers. Building for those browsers means that various hacks and workarounds that make angular harder to develop can be eliminated, allowing developers to focus on code related to their business domain.

Now, let us talk about Angular 7. The Angular community has introduced some significant changes and new features to Angular 7. The following features were introduced in the Angular 7:

1. **CLI Prompts** – In Angular 7, when we try to create new projects using Angular CLI, it will prompt the user to select or provide options if they want to add features like routing, format of the style sheet, etc. for the applications.
2. **Performance Increase** – Now in Angular 7, we can fix the application bundle size. This feature will warn developers when the application bundle size exceeds during deployment or build process. The default value of application bundle size is 2 MB and for error logs, the default size is 5 MB. But it is configurable, and we can change the value of it by using angular.json file. This feature undoubtedly increases the performance of the applications.
3. **Angular Material & SDK** – In Angular 7, there are major changes and upgrades in the Angular Material design. Since Angular Material SDK provides many reusable components for any angular applications which provides an automatic browser compatibility, maintain all web design principles. It also provides a full responsive web design for the browser or any type of devices.
4. **Virtual Scrolling** – In Angular 7, we can take advantage of virtual scrolling with the help of a newly added CDK called ScrollingModule. If we define the virtual scrolling, then scrolling module can load or unload the DOM elements based on the noticeable aspects of the data list.
5. **Drag & Drop** – Another new feature of Angular 7 is the Drag & Drop CDK. With the help of this CDK, we can implement features like drag and drop automatically. And it behaves like automatic rendering of items as the user moves items. It can also transfer any item from one list to another list by using the helper methods like *moveItemInArray* or *transferArrayItem*.
6. **Angular Elements** – Angular Elements will help us to project the content in the web application by using the web standard for the custom elements.
7. **Dependency Updates** – With the release of Angular 7, Angular-related dependencies and documentations also have been upgraded. Angular 7 now supports TypeScript ver 3.1, RxJS 6.3, and Node JS ver 10.0.
8. **Ivy Rendered** – Angular compiles our templates into equivalent TypeScript Code. The code is then compiled along with the TypeScript to Javascript code, and the result is shipped to users. Ivy renderer is the new rendering engine, which is designed to support backward compatibility with existing rendering and also focused on improving the speed of rendering and optimizing the size of the final package. In Angular, it will not be the default renderer, but we can manually enable it using compiler options. This important feature is not completely released in Angular 7 since it is in experimental mode. The complete version will come in the next release.

Summary

In this chapter we will learn about basic concepts of Angular along with its features. Now, we are ready to explore more about angular in the next chapter onwards.

02. Environment Setup for Angular

In this chapter, you will learn how to create the environment for the Angular 7 development work. You also learn how to create projects for Angular 7 using Angular CLI tools.

Perquisites for Angular 7

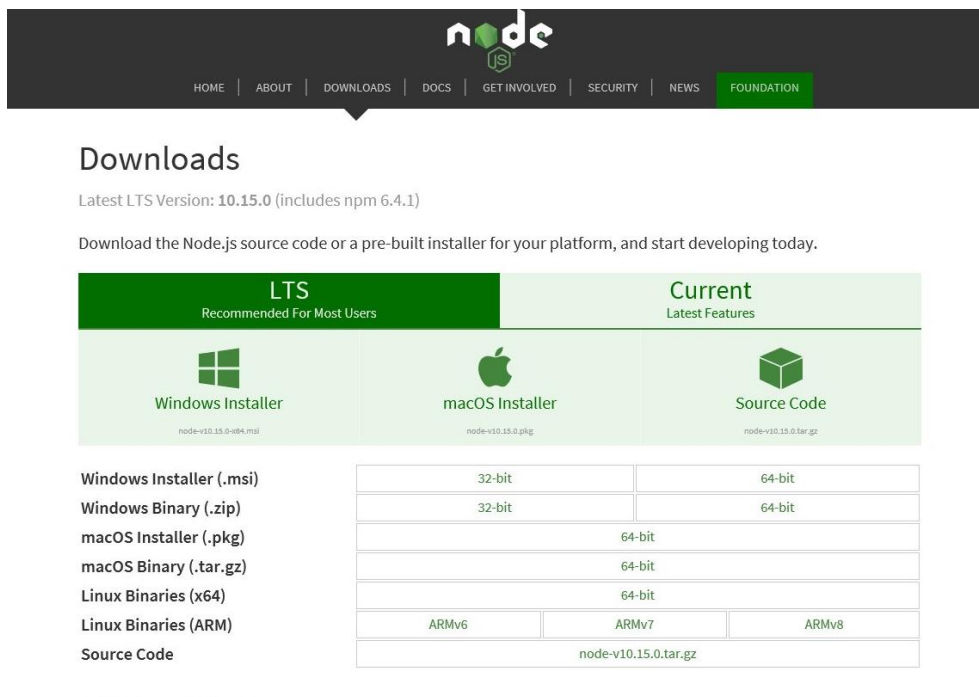
To start development in Angular 7, we need to install some perquisites that are required to run the Angular 7 application smoothly. Theseperquisites are required:

1. Node JS 10.0
2. Angular CLI 7.0
3. TypeScripts 3.1
4. Visual Studio Code or Microsoft Visual Studio (for code editor)

Install Node.js

The latest version of NodeJS can be downloaded from the URL below:

<https://nodejs.org/en/>

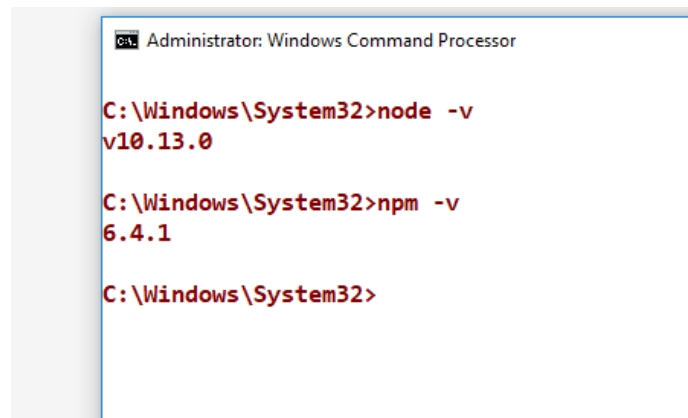


The screenshot shows the Node.js website's 'Downloads' section. It features a dark navigation bar with the Node.js logo and links for HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, NEWS, and FOUNDATION. The main content area is titled 'Downloads' and states 'Latest LTS Version: 10.15.0 (includes npm 6.4.1)'. It instructs users to download the Node.js source code or a pre-built installer. Below this, there are two main columns: 'LTS Recommended For Most Users' and 'Current Latest Features'. The 'LTS' column includes links for Windows Installer, macOS Installer, and Source Code. The 'Current' column includes links for Windows Installer, macOS Installer, and Source Code. A table below these links lists various system architectures and their corresponding download links.

Architecture	Download Link
32-bit	node-v10.15.0-x86.msi
64-bit	node-v10.15.0-x86.msi
32-bit	node-v10.15.0.pkg
64-bit	node-v10.15.0.pkg
64-bit	node-v10.15.0.tar.gz
64-bit	node-v10.15.0.tar.gz
ARMv6	node-v10.15.0.tar.gz
ARMv7	node-v10.15.0.tar.gz
ARMv8	node-v10.15.0.tar.gz

Installing NodeJS also installs the npm on the computer. So, after installing the NodeJS, open command prompt and run the following commands to check the version of the NodeJS and npm:

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



```
Administrator: Windows Command Processor

C:\Windows\System32>node -v
v10.13.0

C:\Windows\System32>npm -v
6.4.1

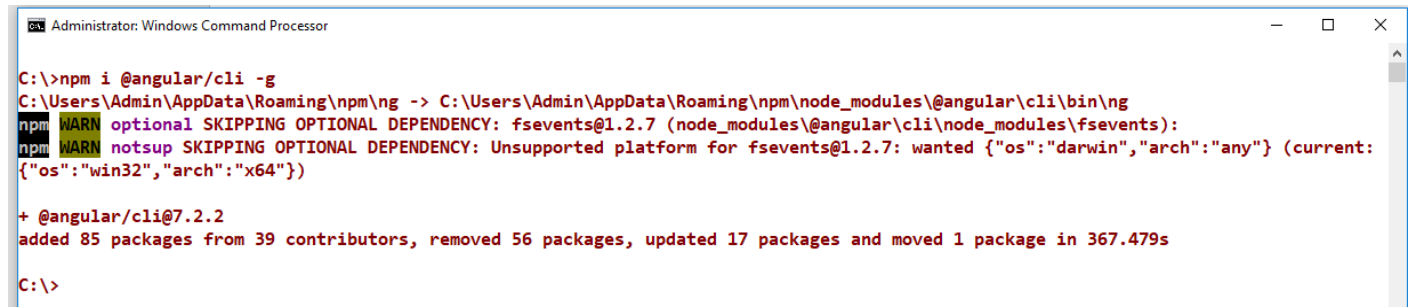
C:\Windows\System32>
```

Install Angular CLI

Angular CLI is the Command Line Interface for Angular. This tool will help us to initialize, develop, and maintain the angular application easily and efficiently. To install the Angular CLI on the computer, run the below commands in the command prompts:

- `npm i @angular/cli -g`

This command installs the Angular CLI tool globally on your computer. Please refer the image below for reference:



```
Administrator: Windows Command Processor

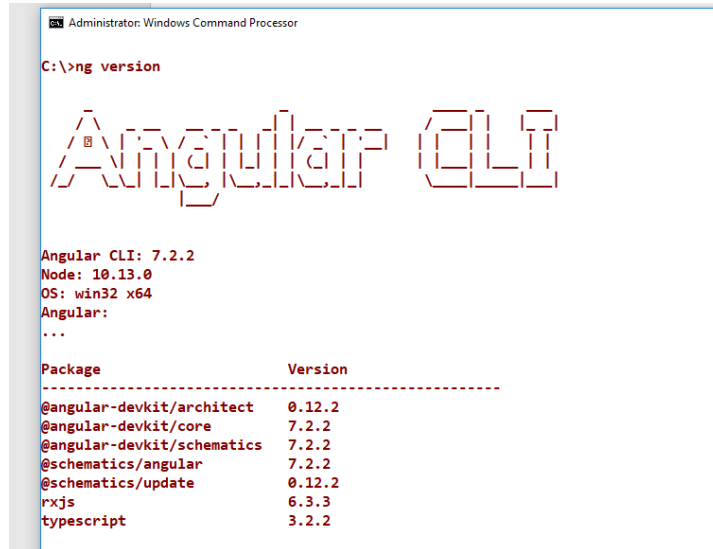
C:\>npm i @angular/cli -g
C:\Users\Admin\AppData\Roaming\npm\ng -> C:\Users\Admin\AppData\Roaming\npm\node_modules\@angular\cli\bin\ng
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.7 (node_modules\@angular\cli\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.7: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ @angular/cli@7.2.2
added 85 packages from 39 contributors, removed 56 packages, updated 17 packages and moved 1 package in 367.479s

C:\>
```

After installation, run the command below to check the version installed on your computer. Refer to the image below as a reference.

- `ng version`



```
Administrator: Windows Command Processor

C:\>ng version

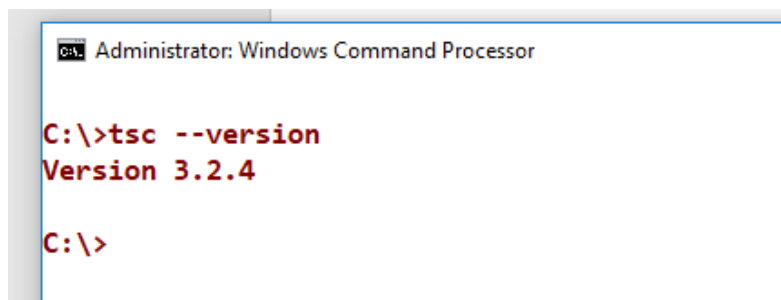
Angular CLI
Angular CLI: 7.2.2
Node: 10.13.0
OS: win32 x64
Angular:
...
Package      Version
-----
@angular-devkit/architect 0.12.2
@angular-devkit/core      7.2.2
@angular-devkit/schematics 7.2.2
@schematics/angular       7.2.2
@schematics/update        0.12.2
rxjs                  6.3.3
typescript              3.2.2
```

Install TypeScript

Angular 7 is based on the TypeScript language. So next, we need to install TypeScript. We need to run the below command from the command prompt to install the latest version of TypeScript on the local computer:

- `npm install -g typescript`

This command installs TypeScript globally on your computer. To check the version of TypeScript, run the command shown in the image below:



```
Administrator: Windows Command Processor

C:\>tsc --version
Version 3.2.4

C:\>
```

Install Code Editor

Now, we need a code editor to write the Angular 7 application. We can use any code editor where we can write code for html or JavaScript files. The most popular code editor for this purpose are:

1. Visual Studio Code (An open source tool that is available with a free license)
2. Microsoft Visual Studio
3. Web Storm

Summary

In this chapter, we learned how to install the related prerequisites of Angular 7 on the local computer so that, at the time of development, the application runs smoothly. In the next chapter, we will learn how to develop an Angular 7 application using these tools and the environment.

03. Create First Angular Application

In this chapter, we will discuss how to create an Angular application project using Angular CLI. We will also discuss the project structure along with some configuration files.

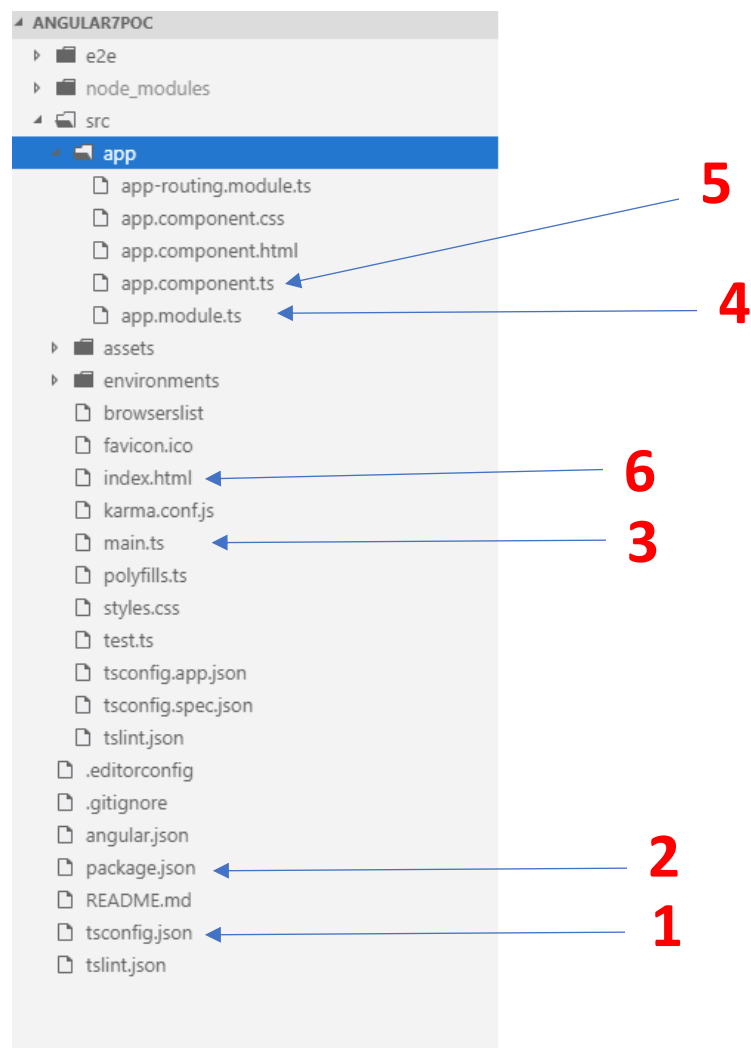
Some Essential Angular CLI Commands

Before we start learning Angular, we will first discuss some important commands of Angular CLI that are required to run an Angular application. The tables below list most of the important commands related to the Angular CLI:

Commands	Descriptions
ng new <projectname>	<p>This command creates a new project in the Angular application. It will create a folder named ProjectName and create an entire solution structure within that folder. This command does the following:</p> <ol style="list-style-type: none"> 1. Creates the directory name (Same as project name) 2. Generates solution directory structure and related source files 3. Installs any related dependency 4. Configures TypeScript 5. Configures Karma test runner 6. Creates Environment files
ng init	<p>This command will do the same work as the ng new command. The only difference is that the ng init command does not create a new directory, and it will create the related solution structure within the current directory.</p>
ng serve	<p>This command is used to run the Angular application. After running this command from command prompt, it will open the browser and run the application in the browser. At the same type, this command continues running to detect the changes, recompile the project, and then refresh in the browser.</p>
ng generate ng g	<p>This command is used to create different Angular elements as per our requirement. This command can be used to create any one of the following elements:</p> <ol style="list-style-type: none"> 1. Class 2. Interface 3. Component 4. Service 5. Module 6. Enumeration <p>Here are some examples of the syntax:</p> <pre>ng generate class <classname> ng generate component <componentname> ng generate service <servicename> ng generate module <modulename></pre>

Angular Solution Structure

Now it's time to create our first Angular 7 solution. For this purpose, we just need to open the command prompt and then go to the location or folder where we want to create the solution. Now, run the command `ng new project_name`, and it will create a folder with the same name as the `project_name`. Within that folder, it will create all the related files for running any Angular applications. The below image shows the folder structure of the Angular application:



In the above picture, there are several files that are key files. So, we need to understand the uses and importance of those files. These files are:

1. tsconfig.json
2. package.json
3. main.ts
4. app.module.ts
5. app.component.ts
6. index.html

Also, in the above structure, some folders are also created that contain some specific types of files:

1. **node_modules** – node_modules folders contain all the installed, related dependency files and library.
2. **src** – src folders contain the source code i.e. Angular-related files, including html files.
3. **assets** – This folder contains static files like third-party js files, custom css, or stylesheet files.
4. **environments** – This folder contains environment-related definition files. These environment files will be required when we finally try to compile and prepare the solution for the final deployments.

tsconfig.json

tsconfig.json is known as the TypeScript configuration files. They contain some configuration values as a key-value pair combination. The existence of a tsconfig.json file in a project directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project. The sample code of the tsconfig.json file is shown below:

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "module": "es2015",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "importHelpers": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2018",
      "dom"
    ]
  }
}
```

package.json

package.json files contain all the signature information related to the dependent packages and library. These packages are downloaded within the node_modules folder. It must be actual JSON, not just a JavaScript object literal. A lot of the behavior described in this document is affected by the config settings. This file contains some important property values like:

- **name** – name is one of the most important fields in the package.json file. This field name is required, and your package won't install without this value. The name must be less than or equal to 214 characters. The name can't start with a dot or an underscore. New packages must not have uppercase letters in the name.

- **version** – version is another important field that is mandatory in the package.json files. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come with changes to the version.
- **descriptions** – Put a description in it. It's a string. This helps people discover your package, as it's listed in npm search.
- **homepage** – The URL to the project homepage.
- **license** – We should specify a license for your package so that people know how they are permitted to use it. Include any restrictions you're placing on it.

```
{
  "name": "angular7-poc",
  "version": "1.0.1",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "~7.2.0",
    "@angular/common": "~7.2.0",
    "@angular/compiler": "~7.2.0",
    "@angular/core": "~7.2.0",
    "@angular/forms": "~7.2.0",
    "@angular/platform-browser": "~7.2.0",
    "@angular/platform-browser-dynamic": "~7.2.0",
    "@angular/router": "~7.2.0",
    "core-js": "^2.5.4",
    "rxjs": "~6.3.3",
    "tslib": "^1.9.0",
    "zone.js": "~0.8.26"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~0.12.0",
    "@angular/cli": "~7.2.2",
    "@angular/compiler-cli": "~7.2.0",
    "@angular/language-service": "~7.2.0",
    "@types/node": "~8.9.4",
    "@types/jasmine": "~2.8.8",
    "@types/jasminewd2": "~2.0.3",
    "codacy": "~4.5.0",
    "jasmine-core": "~2.99.1",
    "jasmine-spec-reporter": "~4.2.1",
    "karma": "~3.1.1",
    "karma-chrome-launcher": "~2.2.0",
    "karma-coverage-istanbul-reporter": "~2.0.1",
    "karma-jasmine": "~1.1.2",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.",
    "ts-node": "~7.0.0",
    "tslint": "~5.11.0",
    "typescript": "~3.2.2"
  }
}
```


main.ts

The main.ts file acts as a main entry point of our Angular application. This file is responsible for the bootstrapper operation of our Angular modules. It contains some important statements related to the modules and some initial setup configurations like:

- **enableProdMode** – This option is used to disable Angular's development mode and enable Productions mode. Disabling Development mode turns off assertions and other model-related checks within the framework.
- **platformBrowserDynamic** – This option is required to bootstrap the Angular app in the browser.
- **AppModule** – This option indicates which module acts as a root module in the applications.
- **environment** – This option stores the values of the different environment constants.

```
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.error(err));
```

app.module.ts (Angular Module)

In every Angular application, at least one angular module file is required. An Angular application may contain more than one Angular module. Angular modules is a process or system to assemble multiple angular elements, like components, directives, pipes, service, etc., so that these Angular elements can be combined in such a way that all elements can be related with each other and ultimately create an application.

In Angular, @NgModule decorator is used to define the Angular module class. Sometimes, this class is called an NgModule class. @NgModule always takes a metadata object, which tells Angular how to compile and launch the application in the browser. So, to define the Angular module, we need to define some steps as follows:

1. First, we need to import Angular BrowserModule into the Angular module file at the beginning. This BrowserModule class is responsible for running the application in the browser.
2. In the next step, we need to declare the Angular elements like component within the Angular module so that those components or elements can be associated with the Angular module.
3. In the last step, we need to mention one Angular component as a root component for the Angular module. This component is always known as a bootstrap component. So, one Angular module can contain hundreds of components. But out of those components, one component needs to be a root or bootstrap component that will be executed first when the Angular module will be bootstrapped in the browser.

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

app.component.ts (Angular Component)

app.component.ts files are Angular component files. Components are the main building blocks of the Angular framework. We will discuss in details about components and their features in the next chapter.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular 7 Samples';
}
```

Index.html

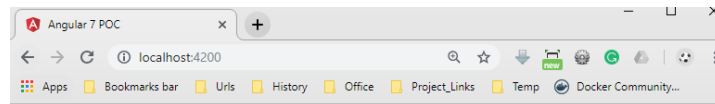
Index.html are the start-up files in the Angular application. So, when the application starts in the browser, this file opens those files. Normally, index files contain the tag select of the bootstrap components.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Angular 7 POC</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>
```

Run Application

Now, it's time to run the application. So, open the console in the root folder position and run the command `ng serve` in the console. It will compile the entire Angular application and then open the application in the browser.



Welcome to Angular 7 Samples!



Summary

In this chapter, we discuss how to create an Angular application using Angular CLI. Also, we discussed the different folders and files that are created automatically by the Angular CLI. Now, in the next chapter, we will discuss the main building block of Angular framework i.e. components.

04. Components

In this chapter, we will discuss components. Angular 7 is a component-based MVVM framework. In Angular 7, components are known as the main building blocks of the Angular framework. Before discussing components, first we need to understand what a component is and why the component-based framework was introduced by Google in Angular.

What Is a Component?

A component is a class that is defined as any visible element on the screen. The components have some properties, and by using them, we can manipulate how the element should look and behave on the screen. We can create, destroy, and update our own components as per our requirement. But when we write code using TypeScript, a component is basically a TypeScript class decorated with an `@Component()` decorator.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular 7 Samples';
}
```

Decorators are JavaScript functions that amend the decorated class in some way. A component is an independent, complete block of code that has the required logic, view, and data as one unit. Logically every component in Angular 7 acts as an MVC concept itself.

Since each component class is an independent unit, it is highly reusable and can be maintained without messing with other components. Scaling is as easy as adding more components to your existing application.

Advantages of Component-Based Architecture

As per the current leading web developers and technical experts, component-based architecture will be act as a most usable architecture in the future web development. Because, with the help of this technique, we can reduce both development time and cost in a large volume in any large-scale web development projects. That's why technical experts currently recommend to implement this architecture in the web-based application development. So, before going to discuss about components in depth, lets discuss some key benefits and advantages related to the Component-Based Architecture.

- **Reusability** – As per the component-based frameworks, components are the most granular units in the development process, and development with components allows gives us a provision of reusability in future development cycles. Since today, technology is changing rapidly. So, if we develop an application in a component-based format, then we are able to swap the best components in and out. A component-

based UI approach allows your application architecture to stay up-to-date over time instead of rebuilding it from scratch.

- **Increase Development Speed** – Component-based development always supports agile development. Components can be stored in a library that the team can access, integrate, and modify throughout the development process. In a broad sense, every developer has specialized skills. As an example, someone can be an expert in JavaScript, another in CSS, etc. With this framework, every specialized developer can contribute their to developing a proper component.
- **Consistent User Experience** – Component-based framework always provides us a consistent user experience for any web development application. In this framework, we can prepare component library so that this component library can be act as a single point of resource for the designers, developers and quality assurance teams. In this way, the Quality Assurance (QA) team always ask the developers to maintained the already approved UI design from the component library. So that, the behaviour of the user interface throughout the entire application can be consistent. Also, with the help of this component library, Quality Assurance team can be develop their test case related to their test on the basis of the available component library design.
- **Easy Integration** – So, as discussed in the previous point, in component-based framework we can develop a library repository related to the component. This component repository can be used a centralized code repository for the current development as well as the future new development also. As the other centralized code repository, we can maintain this library in any source control. In this way, developer can access those repositories and can be updated with new features or functionality as per the new requirement and submit for the approval through their own process.
- **Optimize Requirement and Design** – We can use component-library as a base UI component reference source and so using this source analysis team members like product managers, business analysist or technical lead need to spend less time for finalise the UI design for their new requirements. Because, they already have a bunch of fully tested component with full functionality. Just they need to decide the process about the enhancement points including new business logic only. In this way, this component based framework provide faster speed for the development process.
- **Lower Maintenance Costs** – Since the component-based framework supports reusability, this framework reduces the requirement of the total number of developers when we want to create a new application. Logic-based components are normally context-free, and UI-based components always come with great UX and UI. So, the developer can now focus on integrating those components in the application and how to establish connections between these types of components. Also, system attributes such as security, reliability, performance, maintainability, scalability, and usability (also known as non-functional requirements or NFRs) can also be guaranteed and tested.

@Component Metadata

@Component decorator basically classifies a TypeScript class as a component object. In fact, @Component is a function that takes different types of parameters. In the @Component decorator, we can set the values of different properties to finalize the behaviour of the components. The most commonly used properties of the @Component decorator are as follows:

1. **selector** – A component can be used by the selector expression. Many people treat components as a custom html tag because ultimately, when we want to use the component in the HTML file, we need to provide the selector just like an HTML tag.
2. **template** – The template is the part of the component which is rendered in the browser. We can directly define the html tags or code within the template properties. Sometimes, we call this the inline template. To write multiple lines of html code, all code needs to be covered within tilt (`) symbol.
3. **templateUrl** – This is another way of rendering html tags in the browser. Here, we need to provide the html file name with its related file path. Sometimes it is known as the external template. It is a better approach if the html part of the component is complex.
4. **moduleId** – This is used to resolve the related path of template URL or style URL for the component objects.
5. **styles / styleUrls** – Components can be used in their own style by providing custom css, or they can refer to external style sheet files, which can be used by multiple components at a time. To provide inline style, we need to use styles, and to provide an external file path or url, we need to use styleUrls.
6. **providers** – In the real-life application, we need to use or inject different types of custom services within the component to implement the business logic for the component. To use any custom service within the component, we need to provide the service instance within the provider. Basically, the provider is an array-type property where multiple service instance names can be provided by comma separation.

In the below example, we demonstrate how to define a component using some of the above properties like selector and template:

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

@Component({
  selector: 'first-prog',
  template: '<h1>First Programe in Angular 7.0.</h1>'
})

export class FirstProgComponent {
  constructor() {
  }
}
```

Now, in the below example, we will demonstrate how to use other @Component decorator properties like templateUrl or styles etc.:

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

@Component({
  moduleId: module.id,
  selector: 'template-style',
  templateUrl: 'app.component.template.html',
```

```
styles: ['h1{color:red;font-weight:bold}','h2{color:blue}']
})

export class TemplateUrlStyleComponent {
  constructor() {

  }
}
```

So, in the above example we will separate the html file for storing the HTML part related to the components. As per the above example, we need to place both the TypeScript file and html file in the same location. If we want to place the html in a separate folder, then we can use that file in the component decorator using a relative file path. Below is the sample code written in the app.component.template.html file:

```
<div>
  <h1>Angular 7 Component example with HTML Template</h1>
  <br />
  <h2>C# Corner</h2>
</div>
```

Component Life Cycle Events

Just like other frameworks, Angular components have their own life cycle events that are mainly maintained by Angular itself. Below is the list of life cycle events of Angular 7 components. In Angular, every component has a life-cycle, a number of different stages it goes through from initialization to destruction. There are eight different stages in the component lifecycle. Every stage is called a life cycle hook event. So, we can use these hook events in different phases of our application to obtain fine controls on the components.

- **constructor** – This method is executed before execution of any one life cycle method. It is used for dependency injection.
- **ngOnChanges** – This event executes every time a value of an input control within the component has been changed. This event activates first when a value of a bound property has been changed.
- **ngOnInit** – This event initializes after Angular first displays the data-bound properties or when the component has been initialized. This event is called only once, just after the ngOnChanges() events. This event is mainly used to initialize data in a component.
- **ngDoCheck** – This event is triggered every time the input properties of a component are checked. We can use this hook method to implement the check with our own logic check.
- **ngAfterContentInit** – This lifecycle method is executed when Angular performs any content projection within the component views. This method executes only once, when all the bindings of the component need to be checked for the first time. This event executes just after the ngDoCheck() method.
- **ngAfterContentChecked** – This life cycle hook method executes every time the content of the component has been checked by the change detection mechanism of Angular. This method is called after the ngAfterContentInit() method. This method is also called on every subsequent execution of ngDoCheck().
- **ngAfterViewInit** – This life cycle hook method executes when the component's view has been fully initialized. This method is initialized after Angular initializes the component's view and child views. It is called only once, after ngAfterContentChecked(). This lifecycle hook method only applies to components.

- **ngAfterViewChecked** – This method is called after the `ngAfterViewInit()` method. It is executed every time the view of the given component has been checked by the change detection algorithm of Angular. This method executes every subsequent execution of the `ngAfterContentChecked()`.
- **ngOnDestroy** – This method will be executed just before Angular destroys the components. This method is very useful for unsubscribing the observables and detaching the event handlers to avoid memory leaks. It is called just before the instance of the component being destroyed. This method is called only once, just before the component is removed from the DOM.

The code block below demonstrates how to use these life cycle events within the component class:

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

@Component({
  moduleId: module.id,
  selector: 'demo-event',
  templateUrl: 'app.component.lifecycle.html'
})

export class AppLifeCycleComponent {
  data:number=100;
  constructor() {
    console.log(`new - data is ${this.data}`);
  }

  ngOnChanges() {
    console.log(`ngOnChanges - data is ${this.data}`);
  }

  ngOnInit() {
    console.log(`ngOnInit - data is ${this.data}`);
  }
}
```

HTML part related to the above component as below:

```
<span class="setup">Given Number</span>
<h1 class="punchline">{{ data }}</h1>
```

Nested Component

In the section above, we discussed the different aspects of components, like the definition, metadata, and life-cycle events. So, when we develop an application, it is very often that there are some requirements or scenarios where we need to implement a nested component. A nested component is one component inside another component, or we can say it is a parent-child component. The first question that might be raised in our mind: “Does Angular framework support these types of components?” The answer is yes. We can put any number of components within another component. Also, Angular supports nth level of nesting in general.

So, in this section, we will build two different components separately first and display them in the browser. Then we will place one component within another.

First Component (app.component.sample1.ts)

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

@Component({
  selector: 'app-sample1',
  templateUrl: './sample1.component.html'
})
export class Sample1Component implements OnInit {

  constructor() {}

  ngOnInit() {
  }
}
```

Code of the HTML file (sample1.component.html)

<h2>A component is basically a class that is defined to visible any element on the screen</h2>

Second Component (app.component.sample2.ts)

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

@Component({
  selector: 'app-sample2',
  templateUrl: './sample2.component.html'
})
export class Sample2Component implements OnInit {

  constructor() {}

  ngOnInit() {
  }
}
```

Code of the HTML file (sample2.component.html)

<h3>Life Cycle Events of Components</h3>

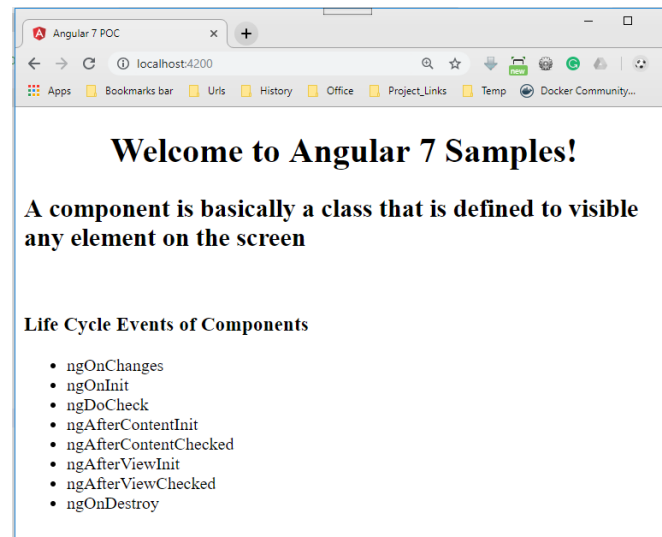
- ngOnChanges
- ngOnInit
- ngDoCheck
- ngAfterContentInit
- ngAfterContentChecked
- ngAfterViewInit
- ngAfterViewChecked
- ngOnDestroy

These two components are developed as an independent component. Now, we want to include Sample2Component as a nested component within the Sample1Component. To do this, we just need to place the selector of Samle2Component within the HTML file code of Sample1Component as shown below.

<h2>A component is basically a class that is defined to visible any element on the screen</h2>

<app-sample2></app-sample2>

The output of the above example is below:



Summary

In this chapter, we discussed different aspects of the component. Also, we discussed the different metadata properties of the components and benefits of the component structure, including the life cycle events of a component. In the next chapter, we will discuss another important features of Angular: data binding.

05. Data Binding

In this chapter, we will discuss one of the most important and fantastic features of Angular: data binding. This is because, in any web application, pushing and pulling of data is always a key aspect of development. This can be accomplished using data binding in Angular.

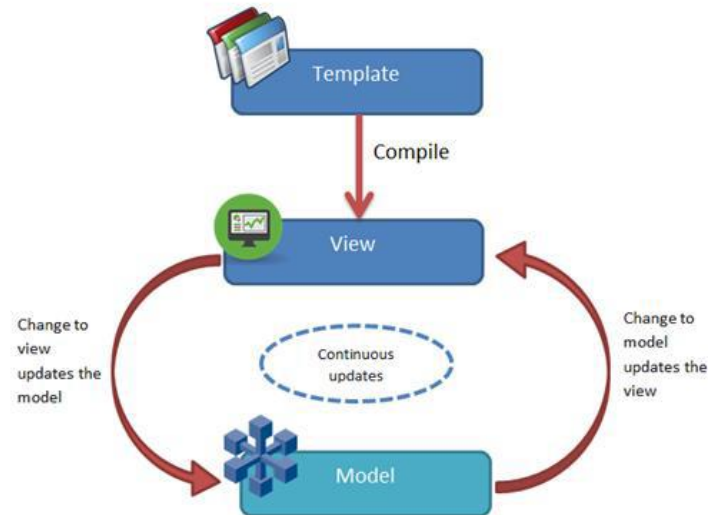
What Is Data Binding?

Data binding is one of the main features of the Angular framework. Data binding in Angular apps is the automatic synchronization of data between the model and view components. The way that Angular implements data binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. Data binding is the process that establishes a connection between the application UI and business logic. If the binding has the correct settings, and the data provides the proper notifications, then, when the data changes its value, the elements that are bound to the data reflect changes automatically.

Concept of Data Binding

In traditional web development, we need to develop a bridge between the front end, where the user performs their application data manipulation, and the back end, where the data is stored. Now, this process is driven by consecutive networking calls, communicating changes between the server (i.e. back end) and the client (i.e. front end).

Most web framework platforms focus on one-way data binding. Basically, this involves reading the input from DOM, serializing the data, sending it to the back end or server, and waiting for the process to finish. After that, the process modifies the DOM to indicate if any errors occurred or reloads the DOM element if the call is successful. While this process provides a traditional web application all the time it needs to perform data processing, this benefit is only really applicable to web apps with highly complex data structures. If your application has a simpler data structure format, with relatively flat models, then the extra work can needlessly complicate the process and decrease the performance of your application.



Angular framework addresses this with the data binding concept. With data binding, the user interface changes are immediately reflected in the underlying data model and vice-versa. This allows the data model to serve as an atomic unit that the view of the application can always depend upon to be accurate. Many web frameworks implement this type of data binding with a complex series of event listeners and event handlers – an approach that can quickly become fragile. Angular, on the other hand, makes this approach to data a primary part of its architecture. Instead of creating a series of call-backs to handle the changing data, Angular does this automatically without any needed intervention by the programmer. Basically, this feature is a great relief for the programmer.

So, the primary benefit of data binding is that updates to (and retrievals from) the underlying data store happen more or less automatically. When the data store updates, the UI updates as well. In essence, it allows for true data encapsulation on the front end, reducing the need to do complex and destructive manipulation of the DOM.

Types of Data Binding

In Angular there are four different types of Data binding processes available. They are:

1. Interpolation
2. Property Binding
3. Two-Way Binding
4. Event Binding

Interpolation

Interpolation data binding is the most popular and easiest way of data binding in angular 7. This mechanism is also available in earlier versions of the Angular framework. Actually, context between the braces is the template expression that Angular first evaluates and then convert into strings. Interpolation use the braces expression i.e. `{{ }}` to render the bound value to the component template. It can be a static string, numeric value, or an object of your data model. In Angular we use it like this: `{{firstName}}`

The below example shows how we can use the interpolation in the component to display data in the front end.

Code of interpolation.component.ts

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

@Component({
  selector: 'app-interpolation',
  templateUrl: './interpolation.component.html',
  styles: []
})
export class InterpolationComponent implements OnInit {

  value1: number = 10;
  array1: Array<number> = [10, 22, 14];
  dt1: Date = new Date();
  status: boolean = true;

  constructor() {}

  ngOnInit() {
  }

}
```

Code of interpolation.component.html

```
<div>
  <span>Current Number is {{value1}}</span>
  <br><br>
  <span>Current Number is {{value1 | currency}}</span>
  <br /><br />
  <span>Current Number is {{dt1}}</span>
  <br /><br />
  <span>Current Number is {{dt1 | date}}</span>
  <br /><br />
  <span>Status is {{status}}</span>
  <br /><br />
</div>
```

Property Binding

In AngularJS 7.0, another binding mechanism exists, which is called property binding. In nature it is just same as interpolation. Some people also called this process as one-way binding like the previous AngularJS concept. Property binding used [] to send the data from the component to the HTML template. The most common way to use property binding is to assign any property of the html element tag into the [] with the component property value, i.e:

```
<input type="text" [value]="data.name"/>
```

To implement the property binding, we will just make the below changes in the previous HTML file from the interpolation sample i.e. interpolation.component.html

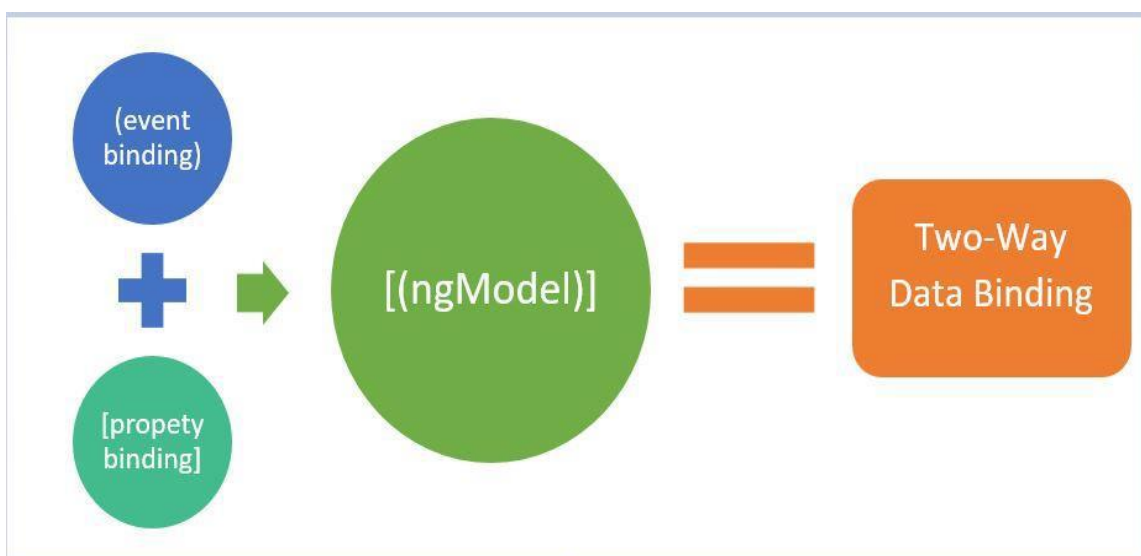
```
<div>
  <span>Current Number is {{value1}}</span>
  <br><br>
  <span>Current Number is {{value1 | currency}}</span>
  <br /><br />
  <span>Current Number is {{dt1}}</span>
  <br /><br />
  <span>Current Number is {{dt1 | date}}</span>
  <br /><br />
  <span>Status is {{status}}</span>
  <br /><br />
  <input [value]="value1" />
  <br /><br />
</div>
```

Two-Way Binding

The most popular and widely used data binding mechanism in the Angular framework is two-way binding. Two-way binding is mainly used in the input type field or any form element where the user types or provides any value or changes any control value in the one side. On the other side, the same is automatically updated into the controller variables and vice versa. Similarly, in Angular 7 we have a directive called ngModel, and it needs to be used as below:

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

We use [] since it is actually a property binding, and parentheses is used for the event binding concept.



ngModel performs both property binding and event binding. Actually, property binding of the ngModel (i.e. [ngModel]) performs the activity to update the input element with a value. Whereas (ngModel) ((ngModelChange) event) instructs the outside world when any change occurred in the DOM Element.

The below example demonstrates the implementation of two-way binding. In this example, we define a string variable called strName and assign that variable with a Textbox control. So, whenever we change any content in the textbox, the value of the variable will be changed automatically.

Code of twowaybinding.component.ts

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

@Component({
  selector: 'app-twowaybinding',
  templateUrl: './twowaybinding.component.html',
  styles: []
})
export class TwowaybindingComponent implements OnInit {
  strName:string = "";
  constructor() { }

  ngOnInit() {
  }
}
```

Code of twowaybinding.component.html

```
<h3>Demonstration of Two Way Binding</h3>
<div>
  <input [(ngModel)]="strName" type="text"/>
  <br>
  <span>{{strName}}</span>
  <br />
  <input [value]="strName" (keyup)="num = $event.target.value" />
  {{num}}
</div>
```

Event Binding

Event binding is another of the data binding techniques available in Angular. This data binding technique does not work with the value of the UI elements—it works with the event action of the UI elements like click-event, blur-event, etc. In the previous version of AngularJS, we always used different types of directives like ng-click, ng-blur to bind any particular event action of an html control. But in the current Angular version, we need to use the same property of the HTML element (like click, change, etc.) and use it within parentheses. In Angular 7, for properties, we use square brackets, and in events, we use parentheses.

```
<button (click)="showAlert();">Click</button>
```

Code of eventbinding.component.ts

```
import { Component, OnInit } from '@angular/core';
import { stringify } from '@angular/core/src/render3/util';

@Component({
  selector: 'app-eventbinding',
  templateUrl: './eventbinding.component.html',
  styles: []
})
export class EventbindingComponent implements OnInit {
  strName:string="";

  constructor() { }

  ngOnInit() {
  }

  fnSubmit():void{
    alert("Your Name is " + this.strName);
  }
}
```

Code of eventbinding.component.html

```
<h3>Demonstration of Event Binding</h3>
<div>
  Enter Your Name : <input type="text" [(ngModel)]=strName"/>
  <br>
  <input type="submit" value="Submit" (click)="fnSubmit()"/>
</div>
```

@Input() Decorator

Every component in the Angular framework always used either as a stateless component or a stateful component. Normally, the components are used as a stateless way. But sometime we need to use some stateful components. Reason behind the using a stateful component in a web application is just the data passing or receiving from the current component to either parent component or a child component. So, in this way, we need to intimate Angular that what type of data or which data may be come to our current component. To implement this concept, we need to use the @Input() decorator against any variable. The key features of @Input() decorator are as follows:

- @Input is a decorator to mark an input property. It is used to define an input property to achieve component property binding.
- @Input decorator binds a property within one component (child component) to receive a value from another component (parent component). This is one-way communication from parent to child.
- The component property should be annotated with the @Input decorator to act as an input property. A component can receive a value from another component using component property binding.

It can be annotated as any type of property, such as number, string, array, or user-defined class. To use an alias for the binding property name, we need to assign an alias name as @Input(alias). Find the use of @Input with the string data type.


```
@Input() caption : string;
```

```
@Input('testValue') arrObjects : Array<string>
```

In the above example, testValue is the alias name. The alias name is needed when we want to pass the value to the input properties. If alias is not defined, then we need to use the input property name to pass the value.

Code of input.component.ts

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

@Component({
  selector: 'app-input',
  templateUrl: './input.component.html',
  styles: []
})
export class InputComponent {
  @Input('name') strName: string = "";
  @Input() phoneNo: number = 0;

  constructor() {

  }
}
```

Code of input.component.html

```
<div>
  <span>Your Name is : {{strName}}</span>
  <br>
  <span>You Phone No is : {{phoneNo}}</span>
</div>
```

Now, when we use this component, we need to pass the input values as below:

```
<app-input [name]="Debasis Saha" [phoneNo]="9830098300"></app-input>
```

@Output() Decorator

In Angular 7, @Output is a decorator to mark an output property. @Output is used to define output properties to achieve custom event binding. @Output will be used with the instance of Event Emitter. The key features of the @Output() decorator are as follows:

- The @Output decorator binds a property of a component to send data from one component (child component) to a calling component (parent component).
- This is one-way communication from a child to parent component.
- @Output binds a property of the EventEmitter class. This property name becomes a custom event name for the calling component.
- The @Output decorator can also alias the property name as @Output(alias), and now this alias name will be used in the custom event binding in the calling component.

It can be entered in any type of property such as number, string, array, or user-defined class. To use alias for the binding property name, we need to assign an alias name as @Output(alias). Find the use of @Output with the string data type.

To demonstrate this concept, we will develop two components. The first component will accept name and email id from the user and then emit that data to the 2nd component, which will receive that value and display in the UI.

Code of output.component.ts

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

@Component({
  selector: 'app-output',
  templateUrl: './output.component.html',
  styles: []
})
export class OutputComponent implements OnInit {

  model:any={};

  @Output('onSubmit') submitEvent = new EventEmitter<any>();

  constructor() {}

  ngOnInit() {
  }

  public fnSubmit():void {
    if (this.model.name==undefined || this.model.email==undefined){
      alert("Please input proper value !!");
    }
    else{
      this.submitEvent.emit(this.model);
    }
  }
}
```

Code of output.component.html

```
<p>
Enter Your Name : <input type="text" [(ngModel)]="model.name"/>
<br>
Enter Email Id : <input type="email" [(ngModel)]="model.email"/>
<br>
<input type="button" value="Submit" (click)="fnSubmit()"/>
</p>
```

Code of outputparent.component.ts

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

@Component({
  selector: 'app-outputparent',
  templateUrl: './outputparent.component.html',
  styles: []
})
export class OutputparentComponent implements OnInit {

  email:string="";
  name:string="";
  constructor() {}

  ngOnInit() {
  }
}
```

```
receiveData(data:any):void{  
    this.email = data.email;  
    this.name=data.name;  
}  
}
```

Code of outputparent.component.html

```
<p>  
<app-output (onSubmit)="receiveData($event)"></app-output>  
<br>  
  
Your Name : {{name}}  
<br>  
Your Email Id : {{email}}  
<br>  
</p>
```

Summary

In this chapter, we discussed different techniques of data binding in Angular. Also, we discussed input and output properties of the components. In the next chapter, we will discuss directives.

06. Directives

In this chapter, we will discuss the concept of Directives in Angular 7. Directives are one of the main building blocks in the Angular framework. In this chapter, we will discuss the different aspects of the directives in the Angular framework.

What Is a Directive?

A directive modifies the DOM by changing the appearance, behavior, or layout of DOM elements. Directives are one of the core building blocks the Angular framework uses to build applications. In fact, Angular 7 components are, in large part, directives with templates. From an Angular 1x perspective, Angular 7 components have assumed a lot of the roles directives used to. The majority of issues that involve templates and dependency injection rules will be solved with components, and issues that involve modifying generic behavior is done through directives. There are three main types of directives in Angular 7:

1. **Component** – Directives with templates.
2. **Attribute Directives** – Directives that changes the behavior of a component or element but don't affect the template.
3. **Structural Directives** – Directives that changes the behavior of a component or element by affecting the template or the DOM decoration of the UI.



Comparison between Component & Directives

Component	Directives
A component is registered with the @Component decorator	A Directive is registered with the @Directives decorator
A component is a directive that uses a shadow DOM to create encapsulated visual behavior called a	A directive is used to add behavior to an existing DOM element.

component. Components are typically used to create UI widgets.	
A component is used to break up the application into smaller components.	A directive is used to design reusable components.
Only one component can be present per DOM element.	Many directives can be used per DOM element.
@View decorator or templateUrl template are mandatory in the component.	Directives don't use View.

Attributes of Directives

Attribute directives are mainly used for changing the appearance or behavior of a component or a native DOM element. Attribute directives actually modify the appearance or behavior of an element. These directives actually act like a simple HTML attribute for any HTML tag. There are some inbuilt attribute directives available in the framework like `ngModel`. But, we can also create any type of custom attribute-based directive as per our requirement. In that case, use the attribute directive selector name as an attribute within the HTML tag in the HTML code section. To create attribute directives, we always need to use or inject the below objects in our custom attribute directive component class. To create an attribute directive, we need to remember the below topics:

1. Import required modules like `directives`, `ElementRef`, and `renderer` from the Angular core library.
2. Create a TypeScript class.
3. Use the `@Directive` decorator in the class.
4. Set the value of the selector property in the `@directive` decorator function. The directive will be used, using the selector value on the elements.
5. In the constructor of the class, inject `ElementRef` and the `renderer` object.
6. You need to inject `ElementRef` in the directive's constructor to access the DOM element.
7. You also need to inject the `renderer` in the directive's constructor to work with the DOM's element style.
8. You need to call the `renderer's setElementStyle` function. In the function, we pass the current DOM element by using the object of `ElementRef` and setting the behavior or property of the current element.

ELEMENTREF - While creating a custom attribute directive, we inject `ElementRef` in the constructor to access the DOM element. `ElementRef` provides access to the underlying native element. `ElementRef` is a service that grants us direct access to the DOM element through its `nativeElement` property. That's all we need to set the element's color using the browser DOM API.

RENDERER - While creating a custom attribute directive, we inject `Renderer` in the constructor to access the DOM element's style. Actually, we call the `renderer's setElementStyle` function. In this function, we pass the current DOM element with the help of `ElementRef` object and set the required attribute of the current element.

HOSTLISTENER - Sometimes we may need to access the input property within the attribute directive so that, as per given attribute directive, we can apply a related attribute within the DOM Element. To trap user actions, we

can call different methods to handle the user actions. To access the method to operate user actions we need to decorate the methods within the @HostListener method.

ngClass

In Angular 7, there are some built-in attribute directives available. One of them is ngClass. The ngClass directive changes the class attribute that is bound to the component or element it's attached to.

ngStyle

Another built-in attribute directive is ngStyle. ngStyle is mainly used to modify or change the element's style attribute. This attribute directive is quite similar to using style metadata in the component class.

The below example demonstrates the uses of both ngClass and ngStyle.

Code of attributedirectives.component.ts

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

@Component({
  moduleId: module.id,
  selector: 'attribute-directive',
  templateUrl: 'app.component.attrdirective.html',
  styles: ['.red {color:red;}', '.blue {color:blue}', '.cyan {color : cyan}']
})

export class AttrDirectiveComponent implements OnInit {
  showColor: boolean = false;

  constructor() { }

  ngOnInit() { }

  changeColor(): void {
    this.showColor = !this.showColor;
  }
}
```

Code of attributedirectives.component.html

```
<div>
  <h3>This is a Attribute Directives</h3>
  <span [class.red]="true">Attribute Change</span><br />
  <span [ngClass]="{'blue':true}">Attribute Change by Using NgClass</span><br />
  <span [ngStyle]="{'font-size':'14px','color':'green'}">Attribute Change by Using NgStyle</span>
  <br /><br />
  <span [class.cyan]="showColor">Attribute Change</span><br />
  <span [ngClass]="{'cyan':showColor}">Attribute Change by Using NgClass</span><br />
  <input type="button" value="Change Color" (click)="changeColor()" />
</div>
```

```
<br /><br />
<span [class.cyan]="showColor">Attribute Change</span><br />
<span [ngClass]="{'cyan':showColor, 'red' : !showColor}">Attribute Change by Using NgClass</span><br />
<br />
</div>
```

Structural Directives

Another types of directives in Angular framework are the Structural Directive. Structural directives are mainly to used to change the design pattern of the UI DOM elements. In HTML, these directives can be used as a template tag. Using this type of directives, we can change the structure of any DOM elements and can redesigned or redecorate that DOM elements. In Angular Framework, there are some system generate structural directives is available like ngIf, ngFor and ngSwitch. We can also create any custom structural directive. The most common example of the any custom structural directives is components. Since, we can consider every component as a structural directive if that component makes some change in the UI DOM elements style or design. All the system generated structural directives have a template name along with some property value which need to provide when we define that directives in the HTML code.

Angular 7 provides built-in directives, ngIf, ngFor, and ngSwitch , to modify a component or element's style attribute.

ngIf

In Angular 1.x, there were the ng-show and ng-hide directives that would show or hide the DOM elements on what the given expression evaluates by setting the display CSS property. In Angular 2.0 onwards, these two directives were removed from the framework, and a new directive was introduced named ngIf. The main difference in ngIf directive compared to ng-show or ng-hide is that it actually removes the element or components entirely from the DOM. With ng-show or ng-hide, Angular kept the DOM elements/components in the page as a display/hide condition, so any component behaviors may keep running even though the component is not visible in the page. In Angular 7.0, ng-show and ng-hide directive are not available, but we can obtain the same functionality by using the [style.display] property of any element.

Code of ngIf.component.ts

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

@Component({
  moduleId: module.id,
  selector: 'toggle-text',
  templateUrl: 'app.component.ngIf.html'
})

export class NgIfComponent implements OnInit {
  showInfo: boolean = false;
```

```
caption: string = 'Show Text';

constructor() { }
ngOnInit() { }

changeData(): void {
    this.showInfo = !this.showInfo;
    if (this.showInfo) {
        this.caption = 'Hide Text';
    }
    else {
        this.caption = 'Show Text';
    }
}
}
```

Code of ngIf.component.html

```
<div>
    <input type="button" value="{{caption}}" (click)="changeData()"/>
    <br />
    <h2 *ngIf="showInfo"><span>Demonstrate of Structural Directives - *ngIf</span></h2>
    <h2 *ngIf="!showInfo"><span>Demonstrate of Structural Directives - Without *ngIf</span></h2>

    <h2 [hidden]><span>Demonstrate of Structural Directives - Without *ngIf</span></h2>
</div>
```

ngFor

The ngFor directive instantiates a template once per item from an iterable. The context of each instantiated template inherits from the outer context with the given loop variable. ngFor provides several exported values that can be used with local variables:

- index – It will be set to the current loop iteration for each template context.
- first - It will be set to a Boolean value, indicating whether the item is the first one in the iteration.
- last - It will be set to a Boolean value, indicating whether the item is the last one in the iteration.
- even - It will be set to a Boolean value, indicating whether this item has an even index.
- odd – It will be set to a Boolean value, indicating whether this item has an odd index.

In the application, if we made some addition or deletion of elements in the array objects, then that changes automatically effected in the DOM element in the browser. Angular track this change with the help of object identity mechanism. This feature is very much important for any implementation when we need to display the list of data and that data can be updated by any input type controls. It is also important for implement any type of animation effects in angular.

Code of ngfor.component.ts

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

@Component({
```



```
moduleId: module.id,
selector: 'product-list',
templateUrl: 'app.component.ngFor.html'
})

export class NgForComponent implements OnInit {
  productList: Array<string> = ['IPhone','Galaxy 7.0','Blackberry 10Z'];

  constructor() { }
  ngOnInit() { }
}
```

Code of ngfor.component.html

```
<div>
  <h2>Demonstrate ngFor</h2>
  <ul>
    <li *ngFor="let item of productList">
      {{item}}
    </li>
  </ul>
</div>
```

ngSwitch

The ngSwitch directive is actually compromise of two directives, an attribute directive and a structural directive. It is similar to a switch statement in JavaScript or other languages. ngSwitch stamps our nested views when their match expression value matches the value of the switch expression. The expression bound to the directives defines what will be compared against in the switch structural directives. If an expression bound to ngSwitchCase matches the one given to ngSwitch, those components are created and the others destroyed. If none of the cases match, then components that have ngSwitchDefault bound to them will be created and the others destroyed. Note that multiple components can be matched using ngSwitchCase, and in those cases, all matching components will be created. Since components are created or destroyed, be aware of the costs in doing so.

Code of ngswitch.component.ts

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

@Component({
  moduleId: module.id,
  selector: 'student-list',
  templateUrl: 'app.component.ngSwitch.html'
})

export class NgSwitchComponent implements OnInit {
  studentList: Array<any> = new Array<any>();

  constructor() { }
  ngOnInit() {
    this.studentList = [
      { SrlNo: 1, Name: 'Candidate', Course: 'Bsc(Hons)', Grade: 'A' },
      { SrlNo: 2, Name: 'Candidate1', Course: 'BA', Grade: 'B' },
      { SrlNo: 3, Name: 'Candidate2', Course: 'BCom', Grade: 'A' },
    ]
  }
}
```

```
{ SrlNo: 4, Name: 'Candidate3', Course: 'Bsc-Hons', Grade: 'C' },
{ SrlNo: 5, Name: 'Candidate4', Course: 'MBA', Grade: 'B' },
{ SrlNo: 6, Name: 'Candidate5', Course: 'MSc', Grade: 'B' },
{ SrlNo: 7, Name: 'Candidate6', Course: 'MBA', Grade: 'A' },
{ SrlNo: 8, Name: 'Candidate7', Course: 'MSc.', Grade: 'C' },
{ SrlNo: 9, Name: 'Candidate8', Course: 'MA', Grade: 'D' },
{ SrlNo: 10, Name: 'Candidate9', Course: 'B.Tech', Grade: 'A' }
];
}
```

Code of ngswitch.component.html

```
<div>
  <h2>Demonstrate ngSwitch</h2>
  <table style="width:100%;border:solid;border-color:blue;border-width:thin;">
    <thead>
      <tr>
        <td>Srl No</td>
        <td>Student Name</td>
        <td>Course</td>
        <td>Grade</td>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let student of studentList;" [ngSwitch]="student.Grade">
        <td>
          <span *ngSwitchCase="A" [ngStyle]="{'font-size':'18px','color':'red'}">{{student.SrlNo}}</span>
          <span *ngSwitchCase="B" [ngStyle]="{'font-size':'16px','color':'blue'}">{{student.SrlNo}}</span>
          <span *ngSwitchCase="C" [ngStyle]="{'font-size':'14px','color':'green'}">{{student.SrlNo}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-size':'12px','color':'black'}">{{student.SrlNo}}</span>
        </td>
        <td>
          <span *ngSwitchCase="A" [ngStyle]="{'font-size':'18px','color':'red'}">{{student.Name}}</span>
          <span *ngSwitchCase="B" [ngStyle]="{'font-size':'16px','color':'blue'}">{{student.Name}}</span>
          <span *ngSwitchCase="C" [ngStyle]="{'font-size':'14px','color':'green'}">{{student.Name}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-size':'12px','color':'black'}">{{student.Name}}</span>
        </td>
        <td>
          <span *ngSwitchCase="A" [ngStyle]="{'font-size':'18px','color':'red'}">{{student.Course}}</span>
          <span *ngSwitchCase="B" [ngStyle]="{'font-size':'16px','color':'blue'}">{{student.Course}}</span>
          <span *ngSwitchCase="C" [ngStyle]="{'font-size':'14px','color':'green'}">{{student.Course}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-size':'12px','color':'black'}">{{student.Course}}</span>
        </td>
        <td>
          <span *ngSwitchCase="A" [ngStyle]="{'font-size':'18px','color':'red'}">{{student.Grade}}</span>
          <span *ngSwitchCase="B" [ngStyle]="{'font-size':'16px','color':'blue'}">{{student.Grade}}</span>
          <span *ngSwitchCase="C" [ngStyle]="{'font-size':'14px','color':'green'}">{{student.Grade}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-size':'12px','color':'black'}">{{student.Grade}}</span>
        </td>
      </tr>
    </tbody>
  </table>
</div>
```

Summary

In this chapter, we discussed different categories of built-in directives available in Angular. Also, we discussed attributes and structural directives and also demonstrated how to use these directives in the application. In the next chapter, we will discuss the concept of Pipes.

07. Pipes

In this chapter, we will discuss the concept of pipes in Angular 7. Pipes are one of the main building blocks in the Angular framework. In this chapter, we will discuss the different aspects of pipes in Angular.

What Is a Pipe?

When we want to develop any application, we always start the application with a simple task: retrieve data, transform data, and then display the data in front of the user through the user interface. Retrieval of data from any type of data source totally depends on data service providers like web services, web API, etc. So, once data arrives, we can push those raw data values directly to our user interface for viewing by the user. But sometimes, this is not exactly what happens. For example, in most use cases, users prefer to see a date in a simple format like 15/02/2017 rather than the raw string format Wed Feb 15 2017 00:00:00 GMT-0700 (Pacific Daylight Time). So, it is clear from the above example that some values require editing before being viewed in the user interface. Also, that same type of transformation might be required by us in many different user interfaces. So, in this scenario we think about some style type properties that we can create centrally and apply whenever we require it. So, for this purpose, Angular framework introduced Angular pipes, a definite way to write display – value transformations that we can declare in our HTML.

In Angular 7.0, pipes are classes that implement a single function interface, accept an input value with an optional parameter array, and return a transformed value.

Why Use Pipes?

Basically, pipes provide a sophisticated and handsome way to perform the tasks within the templates. Pipes make our code clean and structured. In Angular, Pipes accept values from the DOM elements and then return a value according to the business logic implemented within the pipes. So, pipes are one the great features through which can transform our data in the UI and display. But we need to understood one thing very clearly, that pipes do not automatically update our model value. It basically performs the transformation of data and return to the component. If we need to update the model data after transformation through pipes, then we need to update our model data manually. Expect these, we can use pipes for any of the below reason –

- If we want to retrieve the position of the elements
- If we want to track the user inputs in input type elements
- If we want to restrict user to input some value in the input controls

Uses of Pipes

We can use pipes for the below cases:

- We can display only some filtered elements from an array.
- We can modify or format the value.
- We can use them as a function.
- We can do all of the above combined.

Types of Pipes

In Angular 7.0, we can categorize the pipes in two types i.e. Pure Pipes and Impure Pipes.

Pure Pipes :- Pure pipes in angular are those pipes which always accept some arguments as input value and return some value as output according to the input values. Some examples of the pure pipes are – decimal pipes, date pipes etc. When we use these types of pipes in Angular, we provide input value with related configuration value to the pipes and pipes return us the formatted value as an output.

Impure Pipes :- Impure pipes in angular are those pipes which also accept the input values, but return the different types of value set according to the state of the input value. The example of the impure pipes is async pipes. This pipe always stores the internal state and returns different types of value as output according to the internal state and logic.

Filters vs Pipes

In Angular 1.x, filters acted as helpers, similarly to functions, where we can pass the input and other parameters, and it returns a formatted value. But in Angular 7.0, pipes work as an operator. We have an input, and we can modify the input, applying more than one pipe in it. This not only simplifies the nested pipe logic, but also gives us a beautiful and clean syntax for our templates. Secondly, in case of async operations (Newly introduced in Angular 7.0. We will discuss later in this chapter.), we need to set things manually in case of Angular 1.x filters. But pipes are smart enough to handle async operations.

Built-In Pipes

Most of the pipes provided by Angular 7.0 will be familiar with us if we already worked in Angular 1.x. Actually, pipes do not provide any new features in Angular 7.0. In Angular 7.0, we can use logics in the template. Also, we can execute or fire any function within the template to obtain its return value. The pipe syntax starts with the actual input value followed by the pipe (|) symbol and then the pipe name. The parameters of that pipe can be

sent separately by a colon (:). The order of execution of a pipe is right to left. Normally, pipes work within our template and not in JavaScript code. The most commonly used built-in pipes are:

- Currency
- Date
- Uppercase
- Lowercase
- Json
- Decimal
- Percent

Syntax of Pipes

```
myValue | myPipe:param1:param2 | mySecondPipe:param1
```

Code of inbuildpipes.component.ts

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

@Component({
  moduleId: module.id,
  selector: 'inbuild-pipe',
  templateUrl: 'app.component.inbuildpipe.html'
})

export class InBuildPipeComponent implements OnInit {
  private todayDate: Date;
  private amount: number;
  private message: string="Angular 7 is a Component Based Framework";

  constructor() { }

  ngOnInit(): void {
    this.todayDate = new Date();
    this.amount = 1000000;
    this.message = "Angular 7 is a Component Based Framework";
  }
}
```

Code of inbuildpipes.component.html

```
<div>
  <h1>Demonstrate of Pipe in Angular 7.0</h1>

  <h2>Date Format</h2>
  Full Date : {{todayDate}}<br />
  Short Date : {{todayDate | date:'shortDate'}}<br />
  Medium Date : {{todayDate | date:'mediumDate'}}<br />
  Full Date : {{todayDate | date:'fullDate'}}<br />
  Time : {{todayDate | date:'HH:MM'}}<br />
  Time : {{todayDate | date:'hh:mm:ss a'}}<br />
</div>
```

```
Time : {{todayDate | date:'hh:mm:ss p'}}<br />

<h2>Number Format</h2>
No Formatting : {{amount}}<br />
2 Decimal Place : {{amount | number:'2.2-2'}}

<h2>Currency Format</h2>
No Formatting : {{amount}}<br />
USD Doller($) : {{amount | currency:'USD':true}}<br />
USD Doller : {{amount | currency:'USD':false}}<br />
INR() : {{amount | currency:'INR':true}}<br />
INR : {{amount | currency:'INR':false}}<br />

<h2>String Message</h2>
Actual Message : {{message}}<br />
Lower Case : {{message | lowercase}}<br />
Upper Case : {{message | uppercase}}<br />

<h2>Percentage Pipes</h2>
2 Place Formatting : {{amount | percent:'.2'}}<br /><br />
</div>
```

How to Create Custom Pipes

Now we can define custom pipes in Angular 7.0. To configure custom pipes, we need to use a pipes object. For this, we need to define a custom pipe with the @Pipe decorator and use it by adding a pipes property to the @View decorator with the pipe class name. We use the transform method to do any logic necessary to convert the value that is being passed in as input value. We can get a hold of the arguments array as the second parameter and pass in as many as we like from the template.

Code of propercase.pipes.ts

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

@Pipe({
  name: 'propercase'
})

export class ProperCasePipe implements PipeTransform {
  transform(value: string, reverse: boolean): string {
    if (typeof (value) == 'string') {
      let intermediate = reverse == false ? value.toUpperCase() : value.toLowerCase();
      return (reverse == false ? intermediate[0].toLowerCase() :
        intermediate[0].toUpperCase()) + intermediate.substr(1);
    }
    else {
      return value;
    }
  }
}
```

Code of custompipes.component.ts

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

@Component({
  moduleId: module.id,
  selector: 'custom-pipe',
  templateUrl: 'app.component.custompipe.html'
})

export class CustomPipeComponent implements OnInit {
  private message: string;

  constructor() { }

  ngOnInit(): void {
    this.message = "This is a Custom Pipe";
  }
}
```

Code of custompipes.component.html

```
<div>
  <div class="form-horizontal">
    <h2 class="aligncenter">Custom Pipes - Proper Case</h2><br />
    <div class="row">
      <div class="col-xs-12 col-sm-2 col-md-2">
        <span>Enter Text</span>
      </div>
      <div class="col-xs-12 col-sm-4 col-md-4">
        <input type="text" id="txtFName" placeholder="Enter Text" [(ngModel)]="message" />
      </div>
    </div>
    <div class="row">
      <div class="col-xs-12 col-sm-2 col-md-2">
        <span>Result in Proper Case</span>
      </div>
      <div class="col-xs-12 col-sm-4 col-md-4">
        <span>{{message | propercase}}</span>
      </div>
    </div>
  </div>
</div>
```

Summary

In this chapter, we discussed the concept of pipes in Angular. Also, we discussed the different types of built-in pipes available in Angular and how to use them. Also, we discussed how to create and use custom pipes as per our requirements. Now in the next chapter, we will discuss the Angular service.

08. Service

In this chapter, we will discuss another important section of the Angular Framework: services. Angular services play a major role in fetching data from the back end and displaying it in the front end or UI, so it is very important to understand why it is required and how we can use it.

What Is a Service?

An Angular service is simply a JavaScript function, including its related properties and methods, which can perform a particular task or a group of tasks. It is a mechanism to share responsibilities within one or multiple components. As we already know, we can create components in Angular and nest multiple components together within a component using selector. Once our components are nested, we need to manipulate some data within the different components. In this case, a service is the best way to handle this. A service is the best place where we can take data from other sources or write down some calculations. Similarly, services can be shared between multiple components as needed.

Angular has greatly simplified the concept of services since Angular 1.x. In Angular 1, there were services, factories, providers, delegates, values, etc., and it was not always clear when to use which one. So, for that reason, Angular 4 simply changed the concept of Angular. There are simply two steps for creating services in Angular:

1. Create a class with @Injectable decorator.
2. Register the class with the provider or inject the class by using dependency injection.

In Angular, a service is used when a common functionality or business logic needs to be provided, written, or needs to be shared in a different name. Actually, a service is a totally reusable object. Assuming that our Angular application contains some of the components performing the logging for error tracking purpose, you will end up with an error log method in each of these components. This obviously is a bad design approach, as the error log method is duplicated across components. If you want to change the semantics of error logging, then you will need to change the code in all these components, which will impact the whole application. So, a good design approach will be to have a common service component performing the logging feature. The log method in each of the components will be removed and placed in the specific logging service class. The components can use the logging feature by injecting the logging service. Service injection is one form of dependency injection provided by Angular.

Benefits of Using Angular Services

Angular services are single objects that normally get instantiated only once during the lifetime of the Angular application. This Angular service maintains data throughout the life of an application. It means data does not get replaced or refreshed and is available all the time. The main objective of the Angular service is to use shared business logic, models, or data and functions with multiple different components of an Angular application.

The main objective of using an Angular service is the Separation of Concern. An Angular service is basically a stateless object, and we can define some useful functions within an Angular service. These functions can be invoked from any component of the application. This will help us to divide the entire application into multiple small, different, logical units so that those units can be reusable.

How to Create a Service

We can create a custom service as per our requirement. To create a service, we need to follow the below steps:

1. First, we need to create a TypeScript file with proper naming.
2. Next, create a TypeScript class with the proper name that will represent the service after a while.
3. Use the `@Injectable` decorator at the beginning of the class name that was imported from the `@angular/core` packages. Basically, the purpose of the `@Injectable` is that custom service and any of its dependents can be automatically injected by the other components.
4. Although, for design readability, Angular recommends that you always define the `@Injectable` decorator whenever you create any service.
5. Now, use the `Export` keyword against the class objects so that this service can be injectable or reused on any other components.
6. NOTE: When we create our custom service available to the whole application through the use of a provider's meta data, this provider's meta data must be defined in the `app.module.ts` (main application module file) file. If you provide the service in a main module file, then it is visible to the whole application. If you provide it in any component, then only that component can use the service. By providing the service at the module level, Angular creates only one instance of the `CustomService` class, which can be used by all the components in an application.

```
import { Injectable } from '@angular/core';
@Injectable()
export class AlertService
{
    alert(message: string)
    {
        alert(message);
    }

    constructor()
    {}
}
```

@Injectable

`@Injectable` is actually a decorator. Decorators are a proposed extension in JavaScript. In short, a decorator provides the ability to programmers to modify or use methods, classes, properties, and parameters. In angular, every Injectable class actually behave just like a normal class. That's why Injectable class does not have any special lifecycle in Angular framework. So, when we create an object of an Injectable class, the constructor of that class simply executed just like `ngOnInit()` of the component class. But, in case of Injectable class, there is no chance to

define destructor because in JavaScript, there is no concept of destructor. So, in simple work, Injectable service class can't be destroyed. If we want to remove the instance of the service class, then we need to remove the reference point of dependency injection related to that class.

@Injectable() lets Angular know that a class can be used with the dependency injector. @Injectable() is not strictly required if the class has other Angular decorators on it or does not have any dependencies.

What is important is that any class that is going to be injected with Angular is decorated. However, best practice is to decorate injectables with @Injectable(), as it makes more sense to the reader.

```
@Injectable()
export class SampleService
{
    constructor()
    {
        console.log('Sample service is created');
    }
}
```

Similar to the .NET MVC Framework, Angular 7 also provides support for the Dependency Injection or DI. We can use component constructor to inject the instances of the service class. Angular 7 provides the provider metadata in both Module level and Component level which can be used to perform Automatic Dependency Injection of any Injectable Service at the runtime.

Dependency Injection in Angular

Dependency Injection always is one of the main benefits of the Angular Framework. Due to this benefit, the Angular Framework is receiving much more appreciation and acceptance among developers which cannot be achieved by other related client-side frameworks. With the help of this feature, we can inject any types of dependency like service, external utility module in our application module. For doing this, we do not even want to know how those dependency modules or services have been developed.

So, the Angular Framework has its own mechanism for the Dependency Injection system. In this system, every Angular module has its related own injector metadata values. According to that, the injector of each module is responsible for creating the dependent object reference point and then it will inject in the module we required. Actually, every dependency behaves like a key-pair value where token acts as a key and the instance of the object which needs to be injected acts as a value. But in spite of this cool mechanism, there are some problems in the existing Dependency mechanism as below -

- **Internal cache** – In Angular, every dependency object is created as a singleton object. So, when we inject any service class as a dependent object, then the instance of that service class is created once for the entire application lifecycle.
- **Namespace collision** – In Angular, we can't inject two different service classes from two different modules with the same name. As an example, suppose we create a service called user service for our own module and we inject that service. Now, suppose we import an EmployeeModule which contains a same

name service called UserService and we also want to inject that. Then that can't be done since same name service already injected in the application.

- **Built into the framework** – In Angular Framework, Dependency Injection is totally tightly couples with Angular Modules. We can't decouple the Dependency Injection from Angular module as a standalone system

In Angular Framework, Dependency Injection contains three sections like Injector, Provider & Dependency.

1. **Injector** – The main purpose of the using Injector section is the expose an objects or APIs which basically helps us to create instances of the dependent class or services.
2. **Provider** – A Provider is basically act as an Instructor or Commander. It basically provides instruction to the injector about the process of creating instances of the dependent objects. Provider always taken token value as input and then map that token value with the newly created instances of the class objects.
3. **Dependency** – Dependency is the process or type which basically identify the nature of the created objects.

So as per the above discussion, we can perform the below tasks using the Dependency Injection in Angular Framework –

- We can create instances of the service classes in the constructor level using the provider metadata.
- We can use providers to resolve the dependencies between module level provider and component level providers.
- Dependency injection process create the instances of the dependent class and provide the reference of that objects when we required in the code.
- All the instances of the dependency injected objects are created as a Singleton objects.

What Is a Provider?

So now, one question arises after the above discussion: what are these providers that injectors register in each level? A provider is a resource or JavaScript thing or class that Angular uses to provide something we want to use:

- A class provider generates or provides an instance of a class
- A factory provider generates or provides whatever returns when we run a specified function
- A value provider does not need to take up action to provide the result, it just returns a value

Sample of a Class

```
export class testClass {  
  public message:string = "Hello from Service Class";  
  public count:number;  
  constructor() {  
    this.count=1;  
  }  
}
```

Okay, that's the class. Now let's instruct Angular to use it to register a class provider so we can ask the dependency injection system to give us an instance to use in our code. We'll create a component that will serve as the root component for our application. Adding the testClass provider to this component is straightforward:

- Import testClass
- Add it to the @Component providers property
- Add an argument of type "testClass" to the constructor

```
import { Component } from "@angular/core";  
import { testClass } from "../service/testClass";  
  
@Component({  
  module : module.id,  
  selector : 'test-prog',  
  template : '<h1>Hello {{_message}}</h1>',  
  providers : [testClass]  
})  
  
export class TestComponent  
{  
  private _message:string="";  
  constructor(private _testClass : testClass)  
  {  
    this._message = this._testClass.message;  
  }  
}
```

Under the covers, when Angular instantiates the component, the DI system creates an injector for the component that registers the testClass provider. Angular then sees the testClass type specified in the constructor's argument list, looks up the newly registered testClass provider, and uses it to generate an instance that it assigns to "_testClass". The process of looking up the testClass provider and generating an instance to assign to "_testClass" is all Angular. It takes advantage of the TypeScript syntax to know what type to search for, but Angular's injector does the work of looking up and returning the testClass instance.

Summary

In this chapter, we discussed the concept of Angular services. Also, we discussed how to develop an Angular service for any application. In the next chapter, we will discuss how to handle the Ajax request call using Angular services.

09. Ajax Request Handling

Angular 7 introduces many innovative concepts like performance improvements, component routing, sharpened Dependency Injection (DI), lazy loading, async templating, and mobile development with Native Script; all linked with a solid tooling and excellent testing support. Making HTTP requests in Angular 7 apps look somewhat different than what we're used to from Angular 1.x, a key difference being that Angular 4's HTTP returns observables.

It is very clear to us that Angular 7 always looks and feels different than Angular 1.x. In the case of Http API calling, the same scenario occurred. The `$http` service that Angular 1.x provides us works very nicely in most of the cases. Angular7 Http requires that we learn some new concepts, including how to work with observables.

Reactive Extensions for JavaScript (RxJS) is a reactive streams library that allows you to work with observables. RxJS combines observables, operators, and schedulers so we can subscribe to streams and react to changes using composable operations.

What's New in HTTP Service

Angular 7's HTTP API calling again provides a fairly straightforward way of handling requests. For starters, HTTP calls in Angular 7, by default, return observables through RxJS, whereas `$http` in Angular 1.x returns promises. Using observable streams gives us the benefit of greater flexibility when it comes to handling the responses coming from HTTP requests. For example, we have the potential of tapping into useful RxJS operators like `retry` so that a failed HTTP request is automatically re-sent, which is useful for cases where users have poor or intermittent network communication.

In Angular 7, HTTP is accessed as an injectable class from `angular7/http`, and, just like other classes, we import it when we want to use it in our components. Angular 7 also comes with a set of injectable providers for HTTP, which are imported via `HTTP_PROVIDERS`. With these, we get providers such as `RequestOptions` and `ResponseOptions`, which allow us to modify requests and responses by extending the base class for each. In Angular 1.x, we would do this by providing a `transformRequest` or `transformResponse` function to our `$httpOptions`.

Observables vs Promises

When used with HTTP, both implementations provide an easy API for handling requests, but there are some key differences that make observables a superior alternative:

- Promises only accept one value unless we compose multiple promises (Eg: `$q.all`).
- Promises can't be cancelled.

As we already discussed, in Angular 7, there are many new features introduced. An exciting new feature used with Angular is the observable. This isn't an Angular-specific feature, but rather a proposed standard for managing async data that will be included in the release of ES7. Observables open up a continuous channel of communication in which multiple values of data can be emitted over time. From this, we get a pattern of dealing

with data by using array-like operations to parse, modify, and maintain data. Angular uses observables extensively—you'll see them in the HTTP service and the event system.

In version 7, Angular mainly introduces reactive programming concepts based on the observables for dealing the asynchronous processing of data. In Angular 1.x, we basically used to promise to handle the asynchronous processing. But still, in Angular 7, we can still use the promises for the same purpose. The main concept of reactive programming is the observable element that relates to the entity that can be observed. Promises and observables seem to be very similar to each other. Both of them allow us to execute asynchronous processing, register call-backs for both successful and error responses, and notify or inform us when the result is there.

How to Define Observables

Before detailing an example of observables, first we need to understand how to define an observable objects. To create an observable, we can use the create method of the observable object. A function must be provided as parameter with the code to initialize the observable processing. A function can be returned by this function to cancel the observable.

```
var observable = Observable.create((observer) => {  
  setTimeout(() => {  
    observer.next('some event');  
  }, 500);  
});
```

Similar to promises, observables can produce several notifications using the different methods from the observer:

- **next** – Emit an event. This can be called several times.
- **error** – Throw an error. This can be called once and will break the stream. This means that the error callback will be immediately called and no more events or completion can be received.
- **complete** – Mark the observable as completed. After this, no more events or errors will be handled and provided to corresponding call-backs.

Observables allow us to register call-backs for previously described notifications. The subscribe method tackles this issue. It accepts three call-backs as parameters:

- The onNext call-back that will be called when an event is triggered.
- The onError call-back that will be called when an error is thrown.
- The onCompleted call-back that will be called when the observable completes.

Specifications of Observables

Observables and promises have many similarities. In spite of that, observables provide us some new specifications like below–

- **Lazy** - An observable is only enabled when a first observer subscribes. This is a significant difference compared to promises. Due to this, processing provided to initialize a promise is always executed even if

no listener is registered. This means that promises don't wait for subscribers to be ready to receive and handle the response. When creating the promise, the initialization process is always immediately called. Observables are lazy, so we have to subscribe a call-back to let them execute their initialization call-back.

- **Execute Several Times** - Another particularity of observables is that they can trigger several times, unlike promises, which can't be used after they were resolved or rejected.
- **Cancelling Observables** - Another characteristic of observables is that they can be cancelled. For this, we can simply return a function within the initialization call of the Observable.create function. We can refactor our initial code to make it possible to cancel the timeout function.
- **Error Handling** - If something unexpected arises, we can raise an error on the observable stream and use the function reserved for handling errors in the subscribe routine to see what happened.

Observables Array Operators

Using RxJS API, we can use different operations like filter or map and much more operation against any asynchronous collections. In this way, a connectivity established between the observable objects and its iterable pattern objects. In this concept, we can use two different types of array-based object operation – map and filter. But question is, what we can do with the help of these two methods?

1. **map** – map method is basically working with the observable's operations. Using this method, we can create a new array against the received response or result and perform some iteration process to update or manipulate every element of the response and then store that value into the new array. As example, we can use map() to create a new array resultset against the received employee list and add 'Mr' abbreviation against the every employee name using the iteration.
2. **filter** – filter method is also used with the observable's operations. Using this method, we can also create a new array for the result against the received response at a time. It means, we can't not perform any iteration against this method. As example, we can receive the list of employees as a response and then filtered that data against some criteria like salary is above 25000. Now, when subscribe method invokes, it will return the list of employees as a JSON objects according to the given criteria.

The Angular 7 http module @angular/http exposes an HTTP service that our application can use to access web services over HTTP. We'll use this utility in our PeopleService service. We start by importing it together with all types involved in doing an HTTP request:

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

These are all types and methods required to make and handle an HTTP request to a web service:

- **Http**:- The Angular 7 HTTP service that provides the API to make HTTP requests with methods corresponding to HTTP verbs like get, post, put, etc.

- **Response** - Represents a response from an HTTP service and follows the fetch API specification

An Observable is the async pattern used in Angular 7. The concept of an observable comes from the observer design pattern as an object that notifies an interested party of observers when something interesting happens. In case of RxJs, observables are basically used to implement the sequential calling of data or events. So that, every events or data calling can receive their proper response in the applications.

Angular comes with its own HTTP library, which we can use to call out to external APIs.

When we make calls to an external server, we want our user to continue to be able to interact with the page. That is, we don't want our page to freeze until the HTTP request returns from the external server. To achieve this effect, our HTTP requests are asynchronous.

Dealing with asynchronous code is, historically, trickier than dealing with synchronous code. In JavaScript, there are generally three approaches to dealing with async code:

1. Callbacks
2. Promises
3. Observables

HTTP Post

Now, suppose we need to Login in the Application form. For that, we have a Login form which accepts user name and password. So, what we need to do? We need to receive the user name and password when the form is submitted by the user and then we need to authenticate the user so that user can logged in into the application. So, after authentication, we can store the authentication token in the application which can be used for future transaction. So, when we need to call some APIs against form submit, we need to use Http Post Method as below

```
http.post(url: string, body: string, options?: RequestOptionsArgs) : Observable<Result>
```

http.post() signature always accepts some parameters as a input and return an observable result. Some of the input parameters are mandatory parameters.

1. url is the first parameter which takes the url of the API endpoint. It is the mandatory parameter
2. body is the second parameter which accepts the values from the form submit events and pass those values to the API method as arguments. It is a mandatory parameter which can be an object or an array.
3. options are the third parameter which is basically optional. We can pass some configuration key-pair values through this parameter if we required that to communicate with the API end points.

What's New in HTTP Service

Angular 7 HTTP by default returns an Observable opposed to a Promise (\$q module) in \$http. This allows us to use more flexible and powerful RxJS operators like switchMap (flatMapLatest in version 7), retry, buffer, debounce, merge, or zip. By using observables, we improve readability and maintenance of our application, as they can

respond gracefully to more complex scenarios involving multiple emitted values opposed to only a one-off single value.

Code of employeeList.component.ts

```
import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';
import { Http, Response, Headers } from '@angular/http';
import 'rxjs/Rx';

@Component({
  moduleId: module.id,
  selector: 'employee-list',
  templateUrl: 'app.component.employeeList.html'
})

export class EmployeeListComponent implements OnInit {

  private data: Array<any> = [];
  private showDetails: boolean = true;
  private showEmployee: boolean = false;
  private editEmployee: boolean = false;
  private _selectedData: any;
  private _deletedData: any;

  constructor(private http: Http) {
  }

  ngOnInit(): void {
  }

  ngAfterViewInit(): void {
    this.loadData();
  }

  private loadData(): void {
    let self = this;
    this.http.request('http://localhost:81/SampleAPI/employee/getemployee')
      .subscribe((res: Response) => {
        self.data = res.json();
      });
  }

  private addEmployee(): void {
    this.showDetails = false;
    this.showEmployee = true;
  }

  private onHide(args: boolean): void {
    this.showDetails = !args;
    this.showEmployee = args;
    this.editEmployee = args;
    this.loadData();
  }

  private onUpdateData(item: any): void {
    this._selectedData = item;
    this._selectedData.DOB = new Date(this._selectedData.DOB);
  }
}
```

```

this._selectedData.DOJ = new Date(this._selectedData.DOJ);
this.showDetails = false;
this.editEmployee = true;
}

private onDeleteData(item:any):void{
    this._deletedData = item;
    if(confirm("Do you want to Delete Record Permanently?")){
        let self = this;
        let headers = new Headers();
        headers.append('Content-Type', 'application/json; charset=utf-8');
        this.http.post("http://localhost:81/SampleAPI/employee/DeleteEmployee", this._deletedData, { headers: headers })
            .subscribe((res: Response) => {
                self.loadData();
            });
    }
}
}
}

```

Code of employeelist.component.html

```

<div class="container">
    <h3>HTTP Module Sample - Add and Fetch Data</h3>
    <div class="panel panel-default" *ngIf="showDetails">
        <div class="panel-body">
            <table class="table table-striped table-bordered">
                <thead>
                    <tr>
                        <th>Srl No</th>
                        <th>Alias</th>
                        <th>Employee Name</th>
                        <th>Date of Birth</th>
                        <th>Join Date</th>
                        <th>Department</th>
                        <th>Designation</th>
                        <th>Salary</th>
                        <th></th>
                    </tr>
                </thead>
                <tbody>
                    <tr *ngFor="let item of data">
                        <td>{{item.Id}}</td>
                        <td>{{item.Code}}</td>
                        <td>{{item.Name}}</td>
                        <td>{{item.DOB | date : 'shortDate'}}</td>
                        <td>{{item.DOJ | date : 'mediumDate'}}</td>
                        <td>{{item.Department}}</td>
                        <td>{{item.Designation}}</td>
                        <td>{{item.Salary | currency: 'INR': true}}</td>
                        <td>
                            <a (click)="onUpdateData(item);">
                                <i class="fa fa-edit" style="font-size:18px"></i>
                            </a>
                            <a (click)="onDeleteData(item);">
                                <i class="fa fa-remove" style="font-size:18px"></i>
                            </a>
                        </td>
                    </tr>
                </tbody>
            </table>
        </div>
    </div>
</div>

```

```

        </tbody>
    </table>
    <p>
        <button class="btn btn-primary" (click)="addEmployee()">
            Add Employee
        </button>
    </p>
</div>
</div>
<div class="panel panel-default" *ngIf="showEmployee">
    <employee-add (onHide)="onHide($event);"></employee-add>
</div>
<div class="panel panel-default" *ngIf="editEmployee">
    <employee-update [source]="_selectedData" (onHide)="onHide($event);"></employee-update>
</div>
</div>

```

Output

HTTP Module Sample - Add and Fetch Data

Srl No	Alias	Employee Name	Date of Birth	Join Date	Department	Designation	Salary
1	A001	RABIN	6/10/1980	Sep 1, 2006	ACCOUNTS	CLERK	₹15,000.00
2	A002	SUJIT	12/22/1986	Oct 4, 2010	SALES	MANAGER	₹35,000.00
3	A003	KAMALESH	6/3/1982	May 7, 2006	ACCOUNTS	CLERK	₹16,000.00

Add Employee

Summary

In this chapter, we discussed what an http service is and how to operate on ajax request handling. Now in the next chapter, we will discuss routing.

10. Routing

Angular 7 brings many improved modules to the Angular framework, including a new router called the component router. The component router is a totally configurable and features packed router. Features included are standard view routing, nested child routes, named routes, and route parameters.

Why Routing?

Routing allows us to specify some aspects of the application's state in the URL. Unlike with server-side, front-end solutions, this is optional. We can build the full application without ever changing the URL. Adding routing, however, allows the user to go straight into certain aspects of the application. This is very convenient, as it can keep your application linkable and bookmarkable and allow users to share links with others.

Routing allows you to:

- Maintain the state of the application
- Implement modular applications
- Implement the application based on the roles (certain roles have access to certain URLs)

Route Definition Objects

The Routes type is an array of routes that defines the routing for the application. This is where we can set up the expected paths, the components we want to use, and what we want our application to understand them as.

Each route can have different attributes; some of the common attributes are:

- **path** - URL to be shown in the browser when application is on the specific route
- **component** - component to be rendered when the application is on the specific route
- **redirectTo** - redirect route if needed; each route can have either component or redirect attribute defined in the route (covered later in this chapter)
- **pathMatch** - optional property that defaults to 'prefix'; determines whether to match full URLs or just the beginning. When defining a route with empty path string set pathMatch to 'full', otherwise it will match all paths.
- **children** - array of route definitions objects representing the child routes of this route (covered later in this chapter)

To use routes, create an array of route configurations:

```
const routes: Routes = [  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];
```

Router Module

`RouterModule.forRoot` takes the routes array as an argument and returns a configured router module. The following sample shows how we import this module in an `app.routes.ts` file.

```
import { RouterModule, Routes } from '@angular/router';
const routes: Routes = [
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];

export const routing = RouterModule.forRoot(routes);
```

We then import our routing configuration in the root of our application.

```
import { routing } from './app.routes';

@NgModule({
  imports: [ BrowserModule, routing ],
  declarations: [ AppComponent, ComponentOne, ComponentTwo ],
  bootstrap: [ AppComponent ]
})

export class AppModule { }
```

Redirect Route to Another Route

When your application starts, it navigates to the empty route by default. We can configure the router to redirect to a named route by default:

```
export const routes: Routes = [
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];
```

The `pathMatch` property, which is required for redirects, tells the router how it should match the URL provided in order to redirect to the specified route. Since `pathMatch: full` is provided, the router will redirect to `component-one` if the entire URL matches the empty path (`''`).

When starting the application, it will now automatically navigate to the route for `component-one`.

Router Link

Add links to routes using the `RouterLink` directive. For example, the following code defines a link to the route at path `component-one`.

```
<a routerLink="/component-one">Component One</a>
```

Alternatively, you can navigate to a route by calling the `navigate` function on the router:

```
this.router.navigate(['/component-one']);
```

Adding Routing Component Dynamically

Rather than define each route's component separately, use RouterOutlet, which serves as a component placeholder. Angular dynamically adds the component for the route being activated into the `<router-outlet>` element.

```
<router-outlet></router-outlet>
```

In the above example, the component corresponding to the route specified will be placed after the `<router-outlet>` element when the link is clicked.

Nested Child Routes

Child/Nested routing is a powerful new feature in the new Angular router. We can think of our application as a tree structure; components nested in more components. We can think the same way with our routes and URLs.

So, we have the following routes, `/` and `/about`. Maybe our about page is extensive and there are a couple of different views we would like to display as well. The URLs would look something like `/about` and `/about/item`. The first route would be the default about page, but the more route would offer another view with more details.

DEFINING CHILD ROUTES

When some routes may only be accessible and viewed within other routes, it may be appropriate to create them as child routes.

For example: The product details page may have a tabbed navigation section that shows the product overview by default. When the user clicks the "Technical Specs" tab, the section shows the specs instead.

If the user clicks on the product with ID 3, we want to show the product details page with the overview:

```
localhost:3000/product-details/3/overview
```

When the user clicks "Technical Specs":

```
localhost:3000/product-details/3/specs
```

overview and specs are child routes of `product-details/:id`. They are only reachable within product details.

PASSING OPTIONAL PARAMETERS

Query parameters allow you to pass optional parameters to a route such as pagination information.

For example, on a route with a paginated list, the URL might look like the following to indicate that we've loaded the second page:

```
localhost:3000/product-list?page=2
```

The key difference between query parameters and route parameters is that route parameters are essential to determining route, whereas query parameters are optional.

PASSING QUERY PARAMETERS

Use the `[queryParams]` directive along with `[routerLink]` to pass query parameters. For example:

```
<a [routerLink]="['product-list']" [queryParams]="{ page: 99 }">Go to Page 99</a>
```

Alternatively, we can navigate programmatically using the Router service:

```
goToPage(pageNum) {  
    this.router.navigate(['product-list'], { queryParams: { page: pageNum } });  
}
```

READING QUERY PARAMETERS

Similar to reading route parameters, the Router service returns an Observable we can subscribe to to read the query parameters:

```
import { Component } from '@angular/core';  
import { ActivatedRoute, Router } from '@angular/router';  
  
@Component({  
    selector: 'product-list',  
    template: `<!-- Show product list -->`  
})  
  
export default class ProductList  
{  
    constructor( private route: ActivatedRoute, private router: Router)  
    {}  
  
    ngOnInit()  
    {  
        this.sub = this.route  
            .queryParams  
            .subscribe(params => {  
                // Defaults to 0 if no query param provided.  
                this.page = +params['page'] || 0;  
            });  
    }  
  
    ngOnDestroy()  
    {  
        this.sub.unsubscribe();  
    }  
  
    nextPage()  
    {  
        this.router.navigate(['product-list'], { queryParams: { page: this.page + 1 } });  
    }  
}
```

Summary

In this chapter, we discussed how to implement routing in the Angular Framework. Also, we discussed how to move from one route to another router using either router link or child routes.

Download more eBooks

www.c-sharpcorner.com/ebooks

