Angular is one of the most powerful and performance-efficient JavaScript frameworks to build single-page applications for both web and mobile. The powerful features of Angular allow us to create complex, customizable, modern, responsive, and user-friendly web applications.

Angular follows a component-oriented application design pattern to develop completely reusable and modularized web applications. Popular web platforms like Google Adwords, Google Fiber, Adsense have built their user interfaces using Angular.

In this course, you will learn about components, modules, directives, data binding, pipes, HttpClient, Routing, and much more. You will be using Angular CLI to speed up the development process of Angular applications. Angular CLI is a command-line interface tool to scaffold and build Angular applications. Angular CLI offers all best practices right from development till the deployment stage.

Target Audience: Developers

Why angular?

Angular 1 is a JavaScript framework from Google which was used for the development of web applications.

Following are the reasons to migrate from Angular 1 to the latest version of Angular:

Cross-Browser Compliant

Internet has evolved significantly from the time Angular 1.x was designed. Creating a web application that is cross-browser compliant was difficult with Angular 1.x framework. Developers had to come up with various workarounds to overcome the issues. Angular helps to create cross-browser compliant applications easily.

Typescript Support

Angular is written in Typescript and allows the user to build applications using Typescript. Typescript is a superset of JavaScript and more powerful language. The use of Typescript in application development improves productivity significantly.

Web Components Support

Component-based development is pretty much the future of web development. Angular is focused on component-based development. The use of components helps in creating loosely coupled units of application that can be developed, maintained, and tested easily.

Better support for Mobile App Development

Desktop and mobile applications have separate concerns and addressing these concerns using a single framework becomes a challenge. Angular 1 had to address the concerns of a mobile application using additional plugins. However, the Angular framework, addresses the concerns of both mobile as well as desktop applications.

Better performance

The Angular framework is better in its performance in terms of browser rendering, animation, and accessibility across all the components. This is due to the modern approach of handling issues compared to earlier Angular version 1.x.

A Single Page Application (SPA) is a web application that interacts with the user by dynamically redrawing any part of the UI without requesting an entire new page from the server.

For example, have a look at the Amazon web application. When you click on the various links present in the navbar present in any of the web pages of this application, the whole page gets refreshed. This is because visibly, a new request is sent for the new page for almost each user click. You may hence observe that it is not a SPA.

But, if you look at the Gmail web application, you will observe that all user interactions are being handled without completely refreshing the page.

Modern web applications are generally SPAs. SPAs provide a good user experience by communicating asynchronously (a preferable way of communication) with a remote web server (generally using HTTP protocol) to dynamically check the user inputs or interactions and give constant feedback to the user in case of any errors, or wrongful/invalid user interaction. They are built block-by-block making all the functionalities independent of each other. All desktop apps are SPAs in the sense that only the required area gets changed based on user requests.

Angular helps to create SPAs that will dynamically load contents in a single HTML file, giving the user an illusion that the application is just a single page.

Evolution of Angular Framework:

Let us now understand what is Angular and what kind of applications can be built using Angular:

- Angular is an open-source **JavaScript** framework for building both mobile and desktop web applications.
- Angular is exclusively used to build **Single Page Applications (SPA).**
- Angular is completely rewritten and is not an upgrade to Angular 1.
- Developers prefer TypeScript to write Angular code. But other than TypeScript, you can also write code using JavaScript (ES5 or ECMAScript 5).

Why most developers prefer TypeScript for Angular?

- TypeScript is Microsoft's extension for JavaScript which supports object-oriented features and has a strong typing system that enhances productivity.
- TypeScript supports many features like annotations, decorators, generics, etc. A very good number of IDE's like Sublime Text, Visual Studio Code, Nodeclipse, etc., are available with TypeScript support.
- TypeScript code is compiled to JavaScript code using build tools like npm, bower, gulp, webpack, etc., to make the browser understand the code.

Let us look at the features of Angular:

- **Easier to learn**: Angular is more modern and easier for developers to learn. It is a more streamlined framework where developers will be focusing on writing JavaScript classes.
- **Good IDE support**: Angular is written in TypeScript which is a superset of JavaScript and supports all ECMAScript 6 features. Many IDEs like Eclipse, Microsoft Visual Studio, Sublime Text, etc., have good support for TypeScript.
- **Familiar**: Angular has retained many of its core concepts from the earlier version (Angular 1), though it is a complete re-write. This means developers who are already proficient in Angular 1 will find it easy to migrate to Angular.

- **Cross-Platform**: Angular is a single platform that can be used to develop applications for multiple devices.
- **Performance:** Angular performance has been improved a lot in the latest version. This has been done by automatically adding or removing reflect metadata from the polyfills.ts file which makes the application smaller in production.
- **Lean and Fast**: Angular application's production bundle size is reduced by 100s of kilobytes due to which it loads faster during execution.
- **Bundle Budgets:** Angular will take advantage of the bundle budgets feature in CLI which will warn if the application size exceeds 2MB and will give errors if it exceeds 5MB. Developers can change this in angular.json.
- **Simplicity**: Angular 1 had 70+ directives like ng-if, ng-model, etc., whereas Angular has a very less number of directives as you use [ ] and ( ) for bindings in HTML elements.
- **Component-based:**
    - Angular follows component-based programming which is the future of web development. Each component created is isolated from every other part of our application. This kind of programming allows us to use components written using other frameworks.
    - Inside a component, you can write both business logic and view.
    - Every Angular application must have one top-level component referred to as 'Root Component' and several sub-components or child components.

- Open Visual Studio Code IDE. Go to the File menu and select the "Open Folder" option. Select the MyApp folder you have created earlier.
- Observe for our AppComponent you have below files
    - app.component.ts
    - app.component.html
    - app.component.css
- Let us explore each one of them
- Go to src folder-> app -> open **app.component.ts** file
- Observe the following code

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-root',
5.   templateUrl: './app.component.html',
6.   styleUrls: ['./app.component.css']
7. })
8. export class AppComponent {
9.   title = 'AngDemo';
10. }

11.
```

Line 3: Adds component decorator to the class which makes the class a component

Line 4: Specifies the tag name to be used in the HTML page to load the component

Line 5: Specifies the template or HTML file to be rendered when the component is loaded in the HTML page. The template represents the view to be displayed

Line 6: Specifies the stylesheet file which contains CSS styles to be applied to the template.

Line 8: Every component is a class (AppComponent, here) and export is used to make it accessible in other components

Line 9: Creates a property with the name title and initializes it to value 'AngDemo'

- Open **app.component.html** from the app folder and observe the following code snippet in that file

```
1. <span>{{ title }} app is running!</span>
2.
```

Line 3: Accessing the class property by placing property called title inside {{ }}. This is called interpolation which is one of the data binding mechanisms to access class properties inside the template.

Best Practices - Coding Style Rules

☑ Always write one component per file. This will make it easier to read, maintain, and avoid hidden bugs. It makes code reusable and less mistake-prone.

☑ Always define small functions which makes it easier to read and maintain

☑ The recommended pattern for file naming convention is feature.type.ts. For example, to create a component for Login, the recommended filename is login.component.ts. Use the upper camel case for class names. For example, LoginComponent.

Use a dashed case for the selectors of a component to keep the names consistent for all custom elements. Ex: app-root

- Modules in Angular are used to **organize the application**. It sets the execution context of an Angular application.
- A module in Angular is a class with the **@NgModule** decorator added to it. @NgModule metadata will contain the declarations of components, pipes, directives, services that are to be used across the application.
- Every Angular application should have one root module which is loaded first to launch the application.
- Submodules should be configured in the root module.

In the **app.module.ts** file placed under the app folder, you have the following code:

```
1. import { BrowserModule } from '@angular/platform-browser';
2. import { NgModule } from '@angular/core';
3.
4. import { AppRoutingModule } from './app-routing.module';
5. import { AppComponent } from './app.component';
6.
7. @NgModule({
8.   declarations: [
9.     AppComponent
10.    ],
11.    imports: [
12.      BrowserModule,
13.      AppRoutingModule
14.    ],
15.    providers: [],
16.    bootstrap: [AppComponent]
17. })
18. export class AppModule { }
19.
20.
```

Line 1: imports BrowserModule class which is needed to run the application inside the browser

Line 2: imports NgModule class to define metadata of the module

Line 5: imports AppComponent class from app.component.ts file. No need to mention the .ts extension as Angular by default considers the file as a .ts file

Line 8: declarations property should contain all user-defined components, directives, pipes classes to be used across the application. We have added our AppComponent class here

Line 11: imports property should contain all module classes to be used across the application

Line 15: providers' property should contain all service classes. You will learn about the services later in this course

Line 16: bootstrap declaration should contain the root component to load. In this example, AppComponent is the root component that will be loaded in the HTML page

In the **main.ts** file placed under the src folder, observe the following code:

```
1. import { enableProdMode } from '@angular/core';
2. import { platformBrowserDynamic } from '@angular/platform-
   browser-dynamic';
3.
4. import { AppModule } from './app/app.module';
5. import { environment } from './environments/environment';
6.
7. if (environment.production) {
8.   enableProdMode();
9. }
10.
11. platformBrowserDynamic().bootstrapModule(AppModule)
12.   .catch(err => console.error(err));
13.
14.

15.
```

Line 1: imports enableProdMode from the core module

Line 2: import platformBrowserDynamic class which is used to compile the application based on the browser platform

Line 4: import AppModule which is the root module to bootstrap

Line 5: imports environment which is used to check whether the type of environment is production or development

Line 7: checks if you are working in a production environment or not

Line 8: enableProdMode() will enable production mode which will run the application faster

Line 11: bootstrapModule() method accepts root module name as a parameter which will load the given module i.e., AppModule after compilation

Open **index.html** under the src folder.

```
1. <!doctype html>
2. <html lang="en">
3. <head>
4.   <meta charset="utf-8">
5.   <title>MyApp</title>
6.   <base href="/">
7.   <meta name="viewport" content="width=device-width,
   initial-scale=1">
8.   <link rel="icon" type="image/x-icon" href="favicon.ico">
9. </head>
10. <body>
11.   <app-root></app-root>
12. </body>
13. </html>
14.

15.
```

Line 11: loads the root component in the HTML page. app-root is the selector name given to the component. This will execute the component and renders the template inside the browser.

Best Practices - Coding Style Rules

☑ Always add a Module keyword to the end of the module class name which provides a consistent way to quickly identify the modules. Ex: AppModule.

☑ Filename convention for modules should end with module.ts

☑ Always name the module with the feature name and the folder it resides in which provides a way to easily identify it.

Execute the application and check the output.

- Open terminal in Visual Studio Code IDE by selecting View Menu -> Integrated Terminal.
- Type the following command to run the application

```
1. D:\MyApp>ng serve --open
```

- **ng serve** will build and run the application
- **--open** option will show the output by opening a browser automatically with the default port.

Note: If you get an error in the terminal like 'ng is not recognized', use the Node.js command prompt to run this command.

- Use the following command to change the port number if another application is running on the default port(4200)

```
1. D:\MyApp>ng serve --open --port 3000
```

- Following is the output of the MyApp Application

Demo

Highlights:

- Creating a component using Angular CLI
- Exploring the files created

Demosteps:

Problem Statement: Creating a new component called hello and rendering Hello Angular on the page as shown below

1. In the same **MyApp** application created earlier, create a new component called hello using the following CLI command

```
D:\MyApp> ng generate component hello
```

2. This command will create a new folder with the name hello with the following files placed inside it

3. Open **hello.component.ts** file and create a property called courseName of type string and initialize it to "Angular" as shown below in Line number 9

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

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {
  courseName: string = "Angular";
```

```
10.
11.    constructor() { }
12.
13.    ngOnInit() {
14.    }
15.

16. }
```

4. Open **hello.component.html** and display the courseName as shown below in Line 2

```
1. <p>
2.   Hello {{ courseName }}

3. </p>
```

5. Open **hello.component.css** and add the following styles for the paragraph element

```
1. p {
2.     color:blue;
3.     font-size:20px;

4. }
```

6. Open **app.module.ts** file and add HelloComponent to bootstrap property as shown below in Line 11 to load it for execution

```
1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3.
4. import { AppRoutingModule } from './app-routing.module';
5. import { AppComponent } from './app.component';
6. import { HelloComponent } from './hello/hello.component';
7.
```

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

export class AppModule { }
```

7. Open **index.html** and load the hello component by using its selector name i.e., app-hello as shown below in Line 11

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>MyApp</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-hello></app-hello>
 </body>

 </html>
```

8. Now run the application by giving the following command

```
D:\MyApp>ng serve --open
```

Component Styling

CSS styles can be added to a component by adding styles property to the component metadata. Styles is an array property where multiple CSS classes for a component can be defined.

Example:

app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-root',
5.   templateUrl: './app.component.html',
6.   styles: [`
7.   .highlight {
8.       border: 2px solid red;
9.       background-color:AliceBlue;
10.        text-align: center;
11.        margin-bottom: 20px;
12.    }
13. `],
14. })
15. export class AppComponent {
16.
17. }

18.
```

Line 6-13: Add CSS styles specific to this component in the styles property.

app.component.html

```
1. <div class="highlight">
2.   Container Component
3. </div>
4. <app-child></app-child>

5.
```

Line 1: CSS class i.e., the highlight is applied to the div tag

child.component.ts

```
1. import { Component, OnInit } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-child',
5.   templateUrl: './child.component.html',
6.   styles: [
7.     `
8.       .highlight {
9.         border: 2px solid yellow;
10.          background-color: cornsilk;
11.          text-align: center;
12.          margin-bottom: 20px;
13.        }
14.      `,
15.   ],
16. })
17. export class ChildComponent {
18.   constructor() { }
19. }
20.

21.
```

Line 6-13: Add CSS styles specific to this component in the styles property.

child.component.html

```
1. <div class="highlight">
2.   Child Component

3. </div>
```

Output:

Browser Console(Press F12 inside the browser):

Another option to add CSS styles to the component is by using inline-style. The styles can be directly embedded in the HTML template using <style> tag.

Example:

Remove the highlight CSS class from the styles property in the app.component.ts file and add it to the **app.component.html** file as shown below:

```
1. <style>
2.   .highlight {
3.     border: 2px solid green;
4.     background-color: aliceblue;
5.     text-align: center;
6.     margin-bottom: 20px;
7.   }
8. </style>
9. <div class="highlight">Container Component</div>

10. <app-child></app-child>
```

Line 1-8: Add CSS styles inside the template using the style tag. These styles will be specific to this component

Similarly, remove CSS highlight class from styles property of child.component.ts file and add it to **child.component.html** file as shown below

```
1. <style>
2.   .highlight {
3.     border: 2px solid blue;
4.     background-color: cornsilk;
5.     text-align: center;
6.     margin-bottom: 20px;
7.   }
8. </style>
9. <div class="highlight">Child Component</div>

10.
```

Line 1-8: Add CSS styles inside the template using a style tag. These styles will be specific to this component

Output:

Browser Console(Press F12 inside the browser):

Highlights:

- Adding CSS styles to components
- Understanding the usage of styleUrls in applying a style

Demosteps:

Problem Statement: Applying CSS styles to components using styleUrls property. The output is as shown below:

1. Write the below-given code in **app.component.ts**

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

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

 }

```

2. Write the below-given code in **app.component.css**

```
.highlight {
  border: 2px solid coral;
  background-color: aliceblue;
  text-align: center;
  margin-bottom: 20px;
}

```

3. Write the below-given code in **app.component.html**

```
<div class="highlight">
```

```
  Container Component
</div>

<app-child></app-child>
```

4. Create another component called **ChildComponent** using the following CLI command

```
D:\MyApp>ng generate component Child
```

5. Write the below-given code in **child.component.ts**

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css'],
})
export class ChildComponent {

 }

```

6. Write the below-given code in **child.component.css**

```
.highlight {
  border: 2px solid violet;
  background-color: cornsilk;
  text-align: center;
  margin-bottom: 20px;
}
```

```
7.
```

7. Write the below-given code in **child.component.html**

```
1. <div class="highlight">
2.   Child Component
3. </div>

4.
```

8. Save the files and check the output in the browser

9. Open developer tools in your chrome browser and go to the **Elements** tab

Shadow DOM is a web components standard by W3C. It **enables encapsulation for DOM tree and styles**. Shadow DOM hides DOM logic behind other elements and confines styles only for that component.

For example, in an Angular application, n number of components will be created and each component will have its own set of data and CSS styles. When these are integrated, there is a chance that the data and styles may be applied to the entire application. Shadow DOM encapsulates data and styles for each component to not flow through the entire application.

In the below example shown, each component is having its own styles defined and they are confined to themselves:

Angular has built-in **view encapsulation** which enables you to use Shadow DOM.

View Encapsulation defines how to encapsulate CSS styles into a component without flowing them to the rest of the page. The following three modes of encapsulation provided by Angular helps in controlling how the encapsulation has to be applied:

- ViewEncapsulation.Emulated (default)
- ViewEncapsulation.ShadowDOM
- ViewEncapsulation.None

ViewEncapsulation.Emulated

- This is the default encapsulation type in Angular. It emulates style encapsulation even if shadow DOM is not available in the browsers.
- The usage of third-party components in the application might affect the application's look and feel as the third-party components usually comes with their own style. If such third-party component need to be safely added to the application, Emulated mode of ViewEncapsulation can be used.
- When this encapsulation type is used, it re-writes the styles to the document head with some attributes.

- Since there is no shadow DOM, Angular has to write the styles to the document head. In order to enable scoped styles, it extends the CSS selectors so that they don’t collide with other selectors defined in other components. That' why _ngcontent-* attributes are added.

ViewEncapsulation.ShadowDOM enables Angular to use the browser's native Shadow DOM implementation. This mode is introduced in version 7.

Example:

Create a component called **First** using the following CLI command

```
D:\MyApp>ng generate component first
```

first.component.css

```
.cmp {
    padding: 6px;
    margin: 6px;
    border: blue 2px solid;
  }

```

first.component.html

```
<div class="cmp">First Component</div>

```

Line 1: CSS class called cmp is applied to the div tag

Create a component called **Second** using the following CLI command

```
D:\MyApp>ng generate component second
```

second.component.css

```
.cmp {
  border: green 2px solid;
  padding: 6px;
```

```
4.   margin: 6px;
5. }
6.
```

second.component.html

```
1. <div class="cmp">Second Component</div>
```

Line 1: CSS class called cmp is applied to the div tag

app.component.css

```
1. .cmp {
2.  padding: 8px;
3.  margin: 6px;
4.  border: 2px solid red;
5.   }
6.
```

app.component.html

```
1. <h3>CSS Encapsulation with Angular</h3>
2. <div class="cmp">
3.     App Component
4.     <app-first></app-first>
5.     <app-second></app-second>
6. </div>
7.
8.
```

Line 2: Apply CSS class called cmp

Line 4: Load first component

Line 5: load second component

app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-root',
5.   templateUrl: './app.component.html',
6.   styleUrls: ['./app.component.css']
7. })
8. export class AppComponent {
9.
10. }

11.
```

first.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-first',
5.   templateUrl: './first.component.html',
6.   styleUrls: ['./first.component.css']
7. })
8. export class FirstComponent {
9.
10. }

11.
```

second.component.ts

```
1. import { Component, ViewEncapsulation } from
   '@angular/core';
2. @Component({
3.   selector: 'app-second',
4.   templateUrl: './second.component.html',
5.   styleUrls: ['./second.component.css'],
6.   encapsulation: ViewEncapsulation.ShadowDom
7. })
8. export class SecondComponent {
9. }
10.

11.
```

Line 7: encapsulation property sets the encapsulation mode of SecondComponent to ShadowDom

Output:

Browser Console:

Angular doesn't use shadow DOM in **ViewEncapsulation.None**.

All the styles defined in a component is applied to the entire document. i.e., a component can overwrite another component's styles. This is an unscoped strategy.

Example:

Set ViewEncapsulation to none mode in **app.component.ts** file

```
1. import { Component, ViewEncapsulation } from
   '@angular/core';
2.
3. @Component({
4.   selector: 'app-root',
5.   styleUrls: ['./app.component.css'],
6.   templateUrl: './app.component.html',
7.   encapsulation: ViewEncapsulation.None
8. })
9. export class AppComponent {
10. }

11.
```

Set ViewEncapsulation to none mode in **second.component.ts** file

```
1. import { Component, ViewEncapsulation } from
   '@angular/core';
2.
3. @Component({
4.   selector: 'app-second',
5.   templateUrl: './second.component.html',
6.   styleUrls: ['./second.component.css'],
7.   encapsulation: ViewEncapsulation.None
8. })
9. export class SecondComponent {
10.
11. }

12.
```

Output:

Browser Console:

Highlights:

- Shadow DOM using view encapsulation
- Use of ViewEncapsulation.ShadowDOM and ViewEncapsulation.None

Demosteps:

Problem Statement: Applying ShadowDOM and None encapsulation modes to components. Below is the outputs for both the modes:

ViewEncapsulation.ShadowDOM

1. Create a component called **First** using the following CLI command

```
1. D:\MyApp>ng generate component first
```

2. Write the below-given code in **first.component.css**

```
.cmp {
  padding: 6px;
  margin: 6px;
  border: blue 2px solid;
}

```

3. Write the below-given code in **first.component.html**

```
<div class="cmp">First Component</div>

```

4. Create a component called **Second** using the following CLI command

```

D:\MyApp>ng generate component second
```

5. Write the below-given code in **second.component.css**

```
.cmp {
    border: green 2px solid;
    padding: 6px;
    margin: 6px;
}

```

6. Write the below-given code in **second.component.html**

```
<div class="cmp">Second Component</div>
```

7. Write the below-given code in **second.component.ts**

```
import { Component, ViewEncapsulation } from
'@angular/core';
@Component({
  selector: 'app-second',
  templateUrl: './second.component.html',
  styleUrls: ['./second.component.css'],
  encapsulation: ViewEncapsulation.ShadowDom
})
export class SecondComponent {
}

```

8. Write the below-given code in **app.component.css**

```
.cmp {
  padding: 8px;
  margin: 6px;
  border: 2px solid red;
}

```

9. Write the below-given code in **app.component.html**

```
<h3>CSS Encapsulation with Angular</h3>
<div class="cmp">
    App Component
```

```
    <app-first></app-first>
    <app-second></app-second>
</div>


```

9. Save the files and check the output in the browser

ViewEncapsulation.None

1. Set ViewEncapsulation to none mode in **app.component.ts** file

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

@Component({
  selector: 'app-root',
  styleUrls: ['./app.component.css'],
  templateUrl: './app.component.html',
  encapsulation: ViewEncapsulation.None
})
export class AppComponent {
 }


```

2. Set ViewEncapsulation to none mode in **second.component.ts** file

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

@Component({
  selector: 'app-second',
  templateUrl: './second.component.html',
  styleUrls: ['./second.component.css'],
  encapsulation: ViewEncapsulation.None
})
export class SecondComponent {

 }


```

3. Save the files and check the output in the browser

A component has a life cycle that is managed by Angular. It includes creating a component, rendering it, creating and rendering its child components, checks when its data-bound properties change, and destroy it before removing it from the DOM.

Angular has some methods/hooks which provide visibility into these key life moments of a component and the ability to act when they occur.

Following are the lifecycle hooks of a component. The methods are invoked in the same order as mentioned in the table below:

Interface	Hook	Support
OnChanges	ngOnChanges	Directive, Component
OnInit	ngOnInit	Directive, Component
DoCheck	ngDoCheck	Directive, Component
AfterContentInit	ngAfterContentInit	Component
AfterContentChecked	ngAfterContentChecked	Component
AfterViewInit	ngAfterViewInit	Component
AfterViewChecked	ngAfterViewChecked	Component
OnDestroy	ngOnDestroy	Directive, Component

Syntax:

```
1. import { Component, OnInit,  DoCheck, AfterContentInit,
   AfterContentChecked,
2.     AfterViewInit, AfterViewChecked, OnDestroy } from
   '@angular/core';
3.
4. ...
5. export class AppComponent implements OnInit,  DoCheck,
   AfterContentInit, AfterContentChecked,
6.     AfterViewInit, AfterViewChecked,  OnDestroy {
7.
8.     ngOnInit() {  }
9.
10.     ngDoCheck() {  }
11.
12.     ngAfterContentInit() { }
13.
14.     ngAfterContentChecked() { }
15.
16.     ngAfterViewInit() {  }
17.
18.     ngAfterViewChecked() {   }
19.
20.      ngOnDestroy() {  }
21. }

22.
```

Line 1: Import the interfaces of lifecycle hooks

Line 5-6: Inherit interfaces that have life cycle methods to override

Line 8-20: Override all lifecycle hooks

Lifecycle Hooks

- ngOnChanges – It gets invoked when Angular sets data-bound input property i.e., the property attached with @Input(). This will be invoked whenever input property changes its value
- ngOnInit – It gets invoked when Angular initializes the directive or component
- ngDoCheck - It will be invoked for every change detection in the application
- ngAfterContentInit – It gets invoked after Angular projects content into its view
- ngAfterContentChecked – It gets invoked after Angular checks the bindings of the content it projected into its view
- ngAfterViewInit – It gets invoked after Angular creates component's views
- ngAfterViewChecked – It gets invoked after Angular checks the bindings of the component's views
- ngOnDestroy – It gets invoked before Angular destroys directive or component

Best Practice: Always implement the life cycle hook interfaces to flag syntax and spelling mistakes.

Example:

app.component.ts

```
1. import {
2.     Component, OnInit, DoCheck, AfterContentInit,
   AfterContentChecked,
3.     AfterViewInit, AfterViewChecked,
4.     OnDestroy
5. } from '@angular/core';
6. @Component({
7.     selector: 'app-root',
8.     styleUrls: ['./app.component.css'],
9.     templateUrl: './app.component.html'
10. })
11. export class AppComponent implements OnInit, DoCheck,
12.     AfterContentInit, AfterContentChecked,
13.     AfterViewInit, AfterViewChecked,
14.     OnDestroy {
15.     data = 'Angular';
16.     ngOnInit() {
17.         console.log('Init');
18.     }
19.     ngDoCheck(): void {
```

```
20.         console.log('Change detected');
21.     }
22.     ngAfterContentInit(): void {
23.         console.log('After content init');
24.     }
25.     ngAfterContentChecked(): void {
26.         console.log('After content checked');
27.     }
28.     ngAfterViewInit(): void {
29.         console.log('After view init');
30.     }
31.     ngAfterViewChecked(): void {
32.         console.log('After view checked');
33.     }
34.     ngOnDestroy(): void {
35.         console.log('Destroy');
36.     }
37. }

38.
```

Line 11-14: Inherit all lifecycle interfaces

Line 16-36: Override all the lifecycle methods and logging a message

Note: ngOnInit() is the first method to be invoked for AppComponent. Whenever data property value changes it invokes the ngDoCheck() method. All Init methods gets invoked only once at the beginning and from later whenever a change happens Angular invokes ngDoCheck, ngAfterContentChecked and ngAfterViewChecked methods

app.component.html

```
1. <div>
2.   <h1>I'm a container component</h1>
3.   <input type="text" [(ngModel)]='data'>
4.   <app-child [title]='data'></app-child>
5. </div>

6.
```

Line 3: textbox is bound with the data property

Line 4: Loads child component and data property is bound with the title property of the child component

child.component.ts

```
1. ...
2. export class ChildComponent implements OnChanges {
3.   @Input() title: string = 'I\'m a nested component';
4.   ngOnChanges(changes: any): void {
5.     console.log('changes in child:' +
   JSON.stringify(changes));
6.   }

7. }
```

Line 4: title is an input property that receives value from App component

Line 6: Override ngOnChanges method which gets invoked whenever input property changes its value.

Note: Whenever the input property called title changes its value, Angular invokes ngOnChanges() method which takes the changes as a JSON object. The 'changes' parameter will have the previous value and the current value of the input property

child.component.html

```
1. <h2>Child Component</h2>

2. <h2>{{title}}</h2>
```

Output:

Browser Console:

Highlights:

- Component lifecycle
- Various lifecycle hooks

Demosteps:

Problem Statement: Overriding component life-cycle hooks and logging the corresponding messages to understand the flow. The output is as shown below

1. Write the below-given code in **app.component.ts**

```
import {
    Component, OnInit, DoCheck, AfterContentInit,
AfterContentChecked,
    AfterViewInit, AfterViewChecked,
    OnDestroy
```

```
} from '@angular/core';
@Component({
    selector: 'app-root',
    styleUrls: ['./app.component.css'],
    templateUrl: './app.component.html'
})
export class AppComponent implements OnInit, DoCheck,
    AfterContentInit, AfterContentChecked,
    AfterViewInit, AfterViewChecked,
    OnDestroy {
    data = 'Angular';
    ngOnInit() {
        console.log('Init');
    }
    ngDoCheck(): void {
        console.log('Change detected');
    }
    ngAfterContentInit(): void {
        console.log('After content init');
    }
    ngAfterContentChecked(): void {
        console.log('After content checked');
    }
    ngAfterViewInit(): void {
        console.log('After view init');
    }
    ngAfterViewChecked(): void {
        console.log('After view checked');
    }
    ngOnDestroy(): void {
        console.log('Destroy');
    }
}

```

2. Write the below-given code in **app.component.html**

```
<div>
  <h1>I'm a container component</h1>
  <input type="text" [(ngModel)]="data" />
  <app-child [title]="data"></app-child>
```

```
</div>

```

3. Write the below-given code in **child.component.ts**

```
import { Component, OnChanges, Input } from
'@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent implements OnChanges {
  @Input() title!: string;
  ngOnChanges(changes: any): void {
      console.log('changes in child:' +
JSON.stringify(changes));
  }

}
```

4. Write the below-given code in **child.component.html**

```
<h2>Child Component</h2>

<h2>{{title}}</h2>
```

5. Ensure FormsModule is present in the imports section of the AppModule.

6. Save the files and check the output in the browser

@ViewChild Decorator

- ViewChild decorator helps in accessing properties/methods of a child component, directive, or DOM element. ViewChild decorator returns the first element or directive matching the selector from the DOM.

- ViewChild decorator creates an instance of a component/directive class in the parent component to access the properties or methods of that component /directive.

Accessing a child component using @ViewChild

- @ViewChild allows the parent component to access the properties and methods of the child component.
- @ViewChild decorator creates an instance of a child component in the parent component and the selector of the child component should be used in the parent component's template.

Add the below code in **app.module.ts** for loading child and container components.

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

```

Add the following code to **timer.component.ts**

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-timer',
  templateUrl: './timer.component.html',
```

```
  styleUrls: ['./timer.component.css'],
})
export class TimerComponent {
  constructor() { }
  flag = false;
   count = 1;
   begin() {
     this.flag = true;
     const start = setInterval(() => {
       if (this.flag === false) {
         clearInterval(start);
       }
       this.count += 1;
     }, 1000);
   }
   end() {
     this.flag = false;
   }

}
```

Now instantiate **timer.component.ts** in the parent component using ViewChild decorator.

```
import { Component, ViewChild } from '@angular/core';
import { TimerComponent } from './timer/timer.component';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild(TimerComponent) timerComponent!:
TimerComponent;
   startTimer() {
     this.timerComponent.begin();
   }
   stopTimer() {
     this.timerComponent.end();
   }
```

```
16. }
```

Line 12: @ViewChild decorator creates an instance of TimerComponent in AppComponent.

Line 14-19: AppComponent can now access the properties/methods of TimerComponent after loading the selector of TimerComponent in the template of AppComponent.

Add the following code in **timer.component.html**

```
1. <p>{{ count }}</p>
2.
```

Add the following code in **app.component.html**

```
1. <h3>Accessing component using @ViewChild</h3>
2. <br />
3. <br />
4. Timer Example :
5. <button type="button" (click)="startTimer()">Begin</button>
6. <button type="button" (click)="stopTimer()">End</button>
7. <br />
8. <app-timer></app-timer>
9.
```

Accessing a directive using @ViewChild

- @ViewChild creates an instance of a directive within a component and in this way the component can access the methods of the directive class.

Add the following code in the **app.module.ts** file to load the root component and a custom directive.

```
...

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

```

Add the below code in the **color.directive.ts** file.

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

@Directive({
  selector: '[appColor]'
})

export class ColorDirective implements AfterViewInit {
  constructor(private elementRef: ElementRef) { }
  ngAfterViewInit() {
     this.elementRef.nativeElement.style.color = 'green';
   }
   modify(color: string) {
     this.elementRef.nativeElement.style.color = color;
   }

}
```

Line 10: AfterViewInit hook is used to execute statements after a component view is fully initialized.

Now add the following code in the **app.component.ts** file and access the directive methods.

```
1. import { Component, ViewChild } from '@angular/core';
2. import { ColorDirective } from './color.directive';
3.
4. @Component({
5.   selector: 'app-root',
6.   templateUrl: './app.component.html',
7.   styleUrls: ['./app.component.css']
8. })
9.
10. export class AppComponent {
11.   @ViewChild(ColorDirective) colorDirective!:
   ColorDirective;
12.   modifyColor(color: string) {
13.     this.colorDirective.modify(color);
14.   }
15. }

16.
```

Line 12:@ViewChild decorator creates an instance of a color directive in AppComponent.

Line 14-15: The methods of a color directive class can be now accessed from AppComponent.

Accessing a native element using @ViewChild

A template reference variable can be accessed only in that template. If it's required to access it inside a component class, the @ViewChild decorator can be used.

@ViewChild requires the template variable name to be passed as its argument and allows the component to change the appearance or behavior of a given template element.

Observe the following code in **app.component.html**

```
1. <h3>Accessing Template variable using @ViewChild</h3>
2.
3. <div>
4.     Employee Name :
5.     <input type='text' #empname>
6.     <br/> Employee Number :
7.     <input type='number' #empnumber>

8. </div>
```

Line 5-7: There are two input boxes in the above template with 'empname' and 'empnumber' as their respective template reference variables.

Add the below code to **app.component.ts** as shown below to access the native elements.

```
1. import { Component, ViewChild, AfterViewInit, ElementRef }
   from '@angular/core';
2. ...
3. export class AppComponent implements AfterViewInit {
4.
5.   @ViewChild('empname') empName: ElementRef;
6.   @ViewChild('empnumber') empNumber: ElementRef;
7.
8.   ngAfterViewInit() {
9.     this.empName.nativeElement.style.color = 'blue';
10.      this.empNumber.nativeElement.style.color = 'red';
11.   }
12. }

13.
```

Line 5-6: The corresponding ElementRef needs to be instantiated using @ViewChild as shown above in the component to access the native element.

Line 8: AfterViewInit hook is used to execute statements after a component view is fully initialized.

Highlights:

Using @ViewChild to access properties and methods of a child component.

Demosteps:

Problem Statement: Accessing methods of child component using @ViewChild. The output is as shown below:

1. Write the below code in **app.module.ts**.

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

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

```

2. Write the below code in **timer.component.ts.**

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-timer',
  templateUrl: './timer.component.html',
  styleUrls: ['./timer.component.css'],
})
export class TimerComponent {
  constructor() { }
  flag = false;
   count = 1;
   begin() {
     this.flag = true;
     const start = setInterval(() => {
       if (this.flag === false) {
         clearInterval(start);
       }
       this.count += 1;
     }, 1000);
```

```
    }
    end() {
      this.flag = false;
    }

}
```

3. Write the below code in **app.component.ts.**

```
import { Component, ViewChild } from '@angular/core';
import { TimerComponent } from './timer/timer.component';

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

export class AppComponent {
  @ViewChild(TimerComponent) timerComponent!:
TimerComponent;
  startTimer() {
    this.timerComponent.begin();
  }
  stopTimer() {
    this.timerComponent.end();
  }
}

```

4. Write the following code in **app.component.html.**

```
<h3>Accessing component using @ViewChild</h3>
<br />
<br />
Timer Example :
<button type="button" (click)="startTimer()">Begin</button>
<button type="button" (click)="stopTimer()">End</button>
<br />
```

```
<app-timer></app-timer>

```

5. Add the following code in **timer.component.html**

```
<p> {{ count }} </p>
```

6. Save the files and check the output in the browser.

Highlights:

Using @ViewChild to access methods of a directive class in a component.

Demosteps:

Problem Statement: Accessing methods of a directive using @ViewChild. The output is as shown below:

1. Write the below code in **app.module.ts**

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

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

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

```

2. Add the following code in **color.directive.ts**

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

@Directive({
  selector: '[appColor]'
})

export class ColorDirective implements AfterViewInit {
  constructor(private elementRef: ElementRef) { }
  ngAfterViewInit() {
    this.elementRef.nativeElement.style.color = 'green';
  }
  modify(color: string) {
    this.elementRef.nativeElement.style.color = color;
  }
}

```

3. Add the following code in **app.component.ts**

```
import { Component, ViewChild } from '@angular/core';
import { ColorDirective } from './color.directive';

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

export class AppComponent {
```

```
  @ViewChild(ColorDirective) colorDirective!:
ColorDirective;
  modifyColor(color: string) {
    this.colorDirective.modify(color);
  }
}

```

4. Add the below code in **app.component.html**

```
<h3>Accessing Directive using @ViewChild</h3>
<br />
<br />
@ViewChild Example :
<div appColor>Modify Color of this Content</div>
<br />
<br />
<div>
  Modify Color :
   <input type="radio" name="color"
(click)="modifyColor('blue')" />Blue
   <input type="radio" name="color"
(click)="modifyColor('yellow')" />Yellow
   <input type="radio" name="color"
(click)="modifyColor('cyan')" />Cyan

</div>
```

5. Save the files and check the output in the browser

Highlights:

Using @ViewChild to access native elements using the ElementRef property.

Demosteps:

Problem Statement: Accessing Native Element in a component class using @ViewChild and modifying its properties. The output is as shown below:

1. Write the following code in **app.module.ts**

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
     BrowserModule
   ],
   providers: [],
   bootstrap: [AppComponent]
})

export class AppModule { }
```

2. Write the following code in **app.component.html**

```
<h3>Accessing Template variable using @ViewChild</h3>
<div>
  <table>
    <tr>
      <td>Employee Name :</td>
      <td>
        <input type="text" #empname />
      </td>
    </tr>
     <br />
     <tr>
       <td>Employee Number :</td>
       <td>
         <input type="number" #empnumber />
       </td>
     </tr>
  </table>

</div>
```

3. Write the following code in **app.component.ts**

```
import { Component, ViewChild, AfterViewInit, ElementRef } from '@angular/core';

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

export class AppComponent implements AfterViewInit {
   @ViewChild('empname') empName!: ElementRef;
   @ViewChild('empnumber') empNumber!: ElementRef;
   ngAfterViewInit() {
     this.empName.nativeElement.style.color = 'blue';
     this.empNumber.nativeElement.style.color = 'red';
   }
 }

```

4. Save the files and check the output in the browser

Highlights:

- Nested components in mCart application
- ProductListComponent and RatingComponent

Demosteps:

To explore the nested components feature used in **ProductListComponent**

- Below is the output of **ProductListComponent**. Observe the rating displayed for each product rendered by loading another component called **RatingComponent**

- Code for **product-list.component.html** is given below. This file is under the product-list folder.

```
<nav class='navbar navbar-default navbar-fixed-top navbarpos'>
  <div class='container-fluid'>
        <a class='navbar-brand txtcolor'>{{pageTitle}} <span
                    class="glyphicon glyphicon-shopping-cart txtcolor"></span></a>
        <div class="input-group pull-right col-md-3 searchboxpos">
              <input type="text" class="form-control" placeholder="Search" name="q" [(ngModel)]="listFilter"
                    (change)="searchtext()">
              <div class="input-group-btn">
                    <button class="btn btn-default">
                                <em class="glyphicon glyphicon-search"></em>
                          </button>
                    </div>
              </div>

              <!-- Displays the selected Items in the cart along with price -->
              <div class="pull-right txtcolor cartpos">
                    <span class="glyphicon glyphicon-shopping-cart"></span> <a [routerLink]="['cart']"
    class="txtcolor">{{selectedItems}} items</a>
                    <span>, {{total | currency:'USD':'symbol':'1.2-2'}} </span>
              </div>
        </div>
</nav>
<br />
<br />
<!-- Displays a carousel with the given images -->
<div class="container" class="carouselpos">
        <div id="carousel-example-generic" class="carousel slide carouselheight" data-ride="carousel" data-interval="3000">
              <ol class="carousel-indicators">
                    <li data-target="#carousel-example-generic" data-slide-to="0" class="active"></li>
                    <li data-target="#carousel-example-generic" data-slide-to="1"></li>
```

```
31.                     <li data-target="#carousel-example-
   generic" data-slide-to="2"></li>
32.               </ol>
33.               <div class="carousel-inner">
34.                     <div class="item active">
35.                           <img
   src="assets/imgs/carousel_smart_phone.jpg" alt="First
   slide" style="min-width:100%;height:350px;">
36.
37.                     </div>
38.                     <div class="item carouselimgpos">
39.                           <img
   src="assets/imgs/carousel1.jpg" alt="Second slide"
   style="min-width:100%;height:350px;">
40.
41.                     </div>
42.                     <div class="item">
43.                           <img
   src="assets/imgs/tablet_blue_stylus.jpg" alt="Third slide"
   style="min-width:100%;height:350px;">
44.                     </div>
45.               </div>
46.               <a class="left carousel-control"
   href="#carousel-example-generic" role="button" data-
   slide="prev">
47.                     <span class="glyphicon glyphicon-
   chevron-left"></span>
48.               </a>
49.               <a class="right carousel-control"
   href="#carousel-example-generic" role="button" data-
   slide="next">
50.                     <span class="glyphicon glyphicon-
   chevron-right"></span>
51.               </a>
52.         </div>
53.
54.         <!-- Displays two tabs called Tablets and Mobiles --
   >
55.         <div class='panel with-nav-tabs panel-primary
   noborder'>
56.               <div class='panel-heading noborder bgcolor'>
57.                     <ul class="nav nav-tabs noborder">
58.                           <li class="active tabpos nav-
   item">
59.                                 <a class="nav-link active"
   (click)="tabselect('tablet')" data-toggle="tab">
```

```
60.                                         <i class="fa fa-
  tablet fa-3x" aria-hidden="true"></i>
61.                                         <div>Tablets</div>
62.                                   </a>
63.                             </li>
64.                             <li class="tabpos nav-item">
65.                                   <a class="nav-link active"
  (click)="tabselect('mobile')" data-toggle="tab"><i
66.                                                class="fa fa-
  mobile fa-3x" aria-hidden="true"></i>
67.                                         <div>Mobiles</div>
68.                                   </a>
69.                             </li>
70.                       </ul>
71.                 </div>
72.                 <div class='panel-body'>
73.                       <div class="tab-content">
74.                             <!-- Filtering dropdown -->
75.                             <div class="tab-pane fade in
  active" id="tabprimary">
76.                                   <div class="btn-group">
77.                                         <button
  type="button" class="btn btn-default">Filter</button>
78.                                         <button
  type="button" class="btn btn-default dropdown-toggle" data-
  toggle="dropdown">
79.                                                <span
  class="caret"></span> <span class="sr-only">Toggle
  Dropdown</span>
80.                                         </button>
81.                                         <ul class="dropdown-
  menu multi-column columns-3 noclose">
82.                                                <div
  class="row vdivide">
83.                                                      <div
  class="col-md-4">
84.
    <ul class="multi-column-dropdown noclose">
85.
    <h4>Manufacturer</h4>
86.
    <li *ngFor="let manufac of manufacturers">
87.
          <input type="checkbox" [ngModel]="manufac.checked"
88.
                (change)="filter(manufac)">
```

```
89.
          <label>{{manufac.id}} </label>
90.
    </li>
91.
    </ul>
92.                                                                  </div>
93.                                                                  <div
   class="col-md-4">
94.
    <ul class="multi-column-dropdown noclose">
95.
    <h4>OS</h4>
96.
    <li *ngFor="let ostypes of os">
97.
          <input type="checkbox" [ngModel]="ostypes.checked"
98.
                (change)="filter(ostypes)">
99.
          <label> {{ostypes.id}}</label>
100.
          </li>
101.
    </ul>
102.
    </div>
103.
    <div class="col-md-4">
104.
    <ul class="multi-column-dropdown noclose">
105.
          <h4>Price Range</h4>
106.
          <li *ngFor="let price of price_range">
107.
                <input type="checkbox"
   [ngModel]="price.checked" (change)=filter(price)>
108.
                <label>{{ price.id}} </label>
109.
          </li>
110.
    </ul>
111.
    </div>
```

```
                                                    </div>
                                                </ul>
                                            </div>
                                            <span
  *ngIf="chkmanosprice.length> 0"> {{products.length}}
  results</span>

                                            <!-- sort dropdown -
  ->
                                            <div class="pull-
  right">
                                                <span>Sort By
  </span>
                                                <select
  [ngModel]="sortoption" #sortBy
  (change)="onChange(sortBy.value)">
                                                    <option
  value="popularity">Popularity</option>
                                                    <option
  value="pricelh">Price - Low to High</option>
                                                    <option
  value="pricehl">Price - High to Low</option>
                                                </select>
                                            </div>

                                            <!-- Displays the
  products data -->
                                            <div *ngIf='products
  && products.length'>
                                                <div
  class="row" *ngFor='let product of products |
  orderBy:sortoption ; let i = index'

    [hidden]="(i%4)>0">
                                                    <div
  class="col-xs-3">

    <span class="thumbnail text-center">

    <div>

          <img [src]='product.imageUrl'
  [title]='product.productName'

                [style.width.px]='imageWidth'
  [style.height.px]='imageHeight'
```

```
136.
                [style.margin.px]='imageMargin' alt='Product
  Image'>
137.
    </div>
138.
    <div class="caption">
139.
          <div>
140.
                <a [routerLink]="[product.productId]">
141.
                      {{product.productName}} </a>
142.
          </div>
143.
          <div>{{ product.price |
  currency:'USD':'symbol':'1.2-2'}}</div>
144.
          <div></div>
145.
          <app-rating class="ratingcolor"
  [rate]='product.rating'></app-rating>
146.
          <div>
147.
                <button (click)="addCart(product.productId)"
  class="btn btn-primary">Add to
148.
                      Cart</button>
149.
          </div>
150.
    </div>
151.
    </span>
152.                                                    </div>
153.                                                    <div
  class="col-xs-3">
154.
    <div *ngIf="products[i+1]" class="thumbnail text-center">
155.
    <div>
156.
          <img [src]='products[i+1].imageUrl'
  [title]='products[i+1].productName'
```

```
157.
                [style.width.px]='imageWidth'
  [style.height.px]='imageHeight'
158.
                [style.margin.px]='imageMargin'>
159.
    </div>
160.
    <div class="caption">
161.
          <div>
162.
                <a [routerLink]="[products[i+1].productId]">
163.
                      {{products[i+1].productName}} </a>
164.
          </div>
165.
          <div>{{ products[i+1].price |
  currency:'USD':'symbol':'1.2-2'}}
166.
          </div>
167.
          <div></div>
168.
          <app-rating class="ratingcolor"
  [rate]='products[i+1].rating'></app-rating>
169.
          <div></div>
170.
          <div>
171.
                <button
  (click)="addCart(products[i+1].productId)"
172.
                      class="btn btn-primary">Add to
  Cart</button>
173.
          </div>
174.
    </div>
175.
    </div>
176.                                                    </div>
177.                                                    <div
  class="col-xs-3">
```

```
178.
    <div *ngIf="products[i+2]" class="thumbnail text-center">
179.
    <div>
180.
          <img [src]='products[i+2].imageUrl'
   [title]='products[i+2].productName'
181.
                [style.width.px]='imageWidth'
   [style.height.px]='imageHeight'
182.
                [style.margin.px]='imageMargin'
   alt='Product'>
183.
    </div>
184.
    <div class="caption">
185.
          <div>
186.
                <a [routerLink]="[products[i+2].productId]">
187.
                      {{products[i+2].productName}} </a>
188.
          </div>
189.
          <div>{{ products[i+2].price |
   currency:'USD':'symbol':'1.2-2'}}
190.
          </div>
191.
          <div></div>
192.
          <app-rating class="ratingcolor"
   [rate]='products[i+2].rating'></app-rating>
193.
          <div></div>
194.
          <div>
195.
                <button
   (click)="addCart(products[i+2].productId)"
196.
                      class="btn btn-primary">Add to
   Cart</button>
```

```
197.
          </div>
198.
    </div>
199.
    </div>
200.                                                          </div>
201.                                                          <div
   class="col-xs-3">
202.
    <div *ngIf="products[i+3]" class="thumbnail text-center">
203.
    <div>
204.
          <img [src]='products[i+3].imageUrl'
   [title]='products[i+3].productName'
205.
                [style.width.px]='imageWidth'
   [style.height.px]='imageHeight'
206.
                [style.margin.px]='imageMargin'>
207.
    </div>
208.
    <div class="caption">
209.
          <div>
210.
                <a [routerLink]="[products[i+3].productId]">
211.
                      {{products[i+3].productName}} </a>
212.
          </div>
213.
214.
          <div>{{ products[i+3].price |
   currency:'USD':'symbol':'1.2-2'}}
215.
          </div>
216.
          <div></div>
217.
          <app-rating class="ratingcolor"
   [rate]='products[i+3].rating'></app-rating>
218.
          <div></div>
```

```
219.
          <div>
220.
                  <button
   (click)="addCart(products[i+3].productId)"
221.
                        class="btn btn-primary">Add to
   Cart</button>
222.
          </div>
223.
    </div>
224.
    </div>
225.                                                    </div>
226.                                                </div>
227.                                            </div>
228.                                            <div
   *ngIf='!products || products.length==0'>
229.                                                <div
   class="row">
230.                                                    <div
   class="alert alert-warning">
231.
    There are no products available for the selected criteria!
232.                                                    </div>
233.                                                </div>
234.                                            </div>
235.                                    </div>
236.                                    <br /><br />
237.                                </div>
238.                        </div>
239.                </div>

240.      </div>
```

Line 145: rating is another component that is loaded here in ProductList Component to display the rating of a product. The rating of each product is passed to rate property of Ratingcomponent.

- Below is the code for the **rating.component.ts** file, under the products folder.

```
1. // Rating Component which displays the number of stars
   based on the input given
2.
3. import { Component, Input, Output, EventEmitter } from
   '@angular/core';
4.
5. @Component({
6.   selector: 'app-rating',
7.   template: `
8.       <span *ngFor="let r of range; let i = index">
9.         <i class="glyphicon" [ngClass]="i < rate ?
   'glyphicon-star' : 'glyphicon-star-empty'"></i>
10.       </span>
11.   `
12. })
13. export class RatingComponent {
14.   range: Array<number> = [1, 2, 3, 4, 5];
15.   @Input() rate: number=0;
16. }

17.
```

Line 14: range property is an array that holds 1 to 5 values which represents the rating

Line 15: rate is an input property that receives value from its parent component. It receives the rating of each product

Line 8: ngFor loop will iterate over a range of values given

Line 9: It displays a filled star symbol or empty star symbol based on the rating. Bbootstrap CSS classes are used to display star symbols.

Highlights:

- Understanding usage of @ViewChild in mCart application

Demosteps:

To explore the usage of @ViewChild in mCart application

Open app.component.html and observe below code:

```
1. <div>
2.  <!-- Header with two social icons on the left and a login
   button on the right -->
3.  <nav class='navbar navbar-default navbar-fixed-top'>
4.        <div class='container-fluid'>
5.              <ul class="social-network social-circle
   iconpos">
6.                    <li><a href="#" class="icoFacebook"
   title="Facebook"><i
7.                                    class="fa fa-facebook fa-
   1x"></i></a></li>
8.                    <li class="divider-vertical"></li>
9.                    <li><a href="#" class="icoTwitter"
   title="Twitter"><i
10.                                       class="fa fa-twitter
   fa-1x"></i></a></li>
11.                   </ul>
12.                   <ul class="nav navbar-nav navbar-
   right">
13.                         <li>
14.                               <div class="panel-heading
   text-right"
   style="display:none;padding:0px;position:relative;top:15px;
   " #welcomeEl></div></li>
15.                         <li class="divider-
   vertical"></li>
16.                         <li>
17.                               <div class="btn-nav"
   style="padding-right:15px">
18.                                        <a class="btn
   btn-primary btn-small navbar-btn" (click)="login()"
   #loginEl>{{loginTitle}}</a>
19.                               </div>
20.                         </li>
21.                   </ul>
22.             </div>
23.       </nav>
24.
25.       <!-- Loads the component based on the route path -->
26.       <router-outlet></router-outlet>
27.

28. </div>
```

Line 14: Notice that the div is marked with template reference variable 'welcomeEl'

Line 18: Notice that the anchor tag <a> is marked with template reference variable 'loginEl'

Now open app.component.ts and observe below code:

```
1. import { AfterViewInit, Component, ElementRef, Renderer2,
   ViewChild } from '@angular/core';
2. import { Router } from '@angular/router';
3. import { LoginService } from './login/login.service';
4.
5. @Component({
6.     selector: 'app-root',
7.     templateUrl: 'app.component.html',
8.     styleUrls: ['app.component.css']
9. })
10. export class AppComponent implements AfterViewInit {
11.
12.     loginTitle = 'Login';
13.     userName = '';
14.     welcomeMessage = '';
15.     @ViewChild('loginEl')
16.     loginVal!: ElementRef;
17.     @ViewChild('welcomeEl')
18.     welcomeVal!: ElementRef;
19.
20.     constructor(private loginService: LoginService,
   private router: Router, private renderer: Renderer2) {
21.
22.     }
23. ngOnInit(){}
24.     ngAfterViewInit() {
25.         this.userName = this.loginService.username;
26.         this.loginService.loginElement = this.loginVal;
27.         this.loginService.welcomeElement =
   this.welcomeVal;
28.     }
29.
30.     // Invoked when user clicks on login button
31.     // Navigates to login page
32.     login() {
33.         const value =
   this.loginVal.nativeElement.innerText;
34.         this.loginTitle='';
35.         if (value === 'Login') {
```

```
36.             this.router.navigate(['/login']);
37.         } else if (value === 'Logout') {
38.             sessionStorage.clear();
39.             this.loginTitle = 'Login';
40.
   this.renderer.setProperty(this.loginVal.nativeElement,
   'innerText', 'Login');
41.
   this.renderer.setStyle(this.welcomeVal.nativeElement,
   'display', 'none');
42.             this.router.navigate(['/welcome']);
43.         }
44.     }
45. }

46.
```

Line 15: Notice the use of @ViewChild for connecting loginEl template reference variable to loginVal in the ts file

Line 17: Notice the use of @ViewChild for connecting welcome template reference variable to welcomeVal in the ts file

Line 33: Notice the use of loginVal for capturing the current value of div having template reference variable loginEl

Line 40: Using loginVal, the value of loginEl div is updated

Line 41: Using welcomeVal, display of welcome element is toggled

- Use Services to share the data and functionality only as they are ideal for sharing them across the app.

- Always create a service with single responsibility as if it has multiple responsibilities, it will become difficult to test.

- Always use @Injectable() decorator, as Angular injector is hierarchical and when it is provided to a root injector, it will be shared among all the classes that need a service.

☑ If @Injectable() decorator is used, optimization tools used by Angular CLI production builds will be able to perform tree shaking and remove the unused services from the app.

☑ When two different components need different instances of a service, provide the service at the component level.

RxJS

Reactive Extensions for JavaScript (RxJS) is a third-party library used by the Angular team.

RxJS is a reactive streams library used to work with asynchronous streams of data.

Observables, in RxJS, are used to represent asynchronous streams of data. Observables are a more advanced version of Promises in JavaScript

Why RxJS Observables?

Angular team has recommended Observables for asynchronous calls because of the following reasons:

1. Promises emit a single value whereas observables (streams) emit many values
2. Observables can be cancellable where Promises are not cancellable. If an HTTP response is not required, observables allow us to cancel the subscription whereas promises execute either success or failure callback even if the results are not required.
3. Observables support functional operators such as map, filter, reduce, etc.,

Create and use an observable in Angular

Example:

app.component.ts

```
1. import { Component } from '@angular/core';
2. import { Observable } from 'rxjs';
3.
4. @Component({
5.   selector: 'app-root',
6.   styleUrls: ['./app.component.css'],
7.   templateUrl: './app.component.html'
```

```
8. })
9. export class AppComponent {
10.
11.   data!: Observable<number>;
12.   myArray: number[] = [];
13.   errors!: boolean;
14.   finished!: boolean;
15.
16.   fetchData(): void {
17.     this.data = new Observable(observer => {
18.       setTimeout(() => { observer.next(11); }, 1000),
19.       setTimeout(() => { observer.next(22); }, 2000),
20.       setTimeout(() => { observer.complete(); }, 3000);
21.     });
22.     this.data.subscribe((value) =>
  this.myArray.push(value),
23.       error => this.errors = true,
24.       () => this.finished = true);
25.   }
26. }

27.
```

Line 2: imports Observable class from rxjs module

Line 11: data is of type Observable which holds numeric values

Line 16: fetchData() is invoked on click of a button

Line 17: A new Observable is created and stored in the variable data

Line 18-20: next() method of Observable sends the given data through the stream. With a delay of 1,2 and 3 seconds, a stream of numeric values will be sent. Complete() method completes the Observable stream i.e., closes the stream.

Line 22: Observable has another method called subscribe which listens to the data coming through the stream. Subscribe() method has three parameters. The first parameter is a success callback which will be invoked upon receiving successful data from the stream. The second parameter is an error callback which will be invoked when Observable returns an error and the third parameter is a complete callback which will be invoked upon successful streaming of values from Observable i.e., once complete() is invoked. After which the successful response, the data is pushed to the local array called myArray, if any error occurs, a Boolean value called true is stored in the errors variable and upon complete() will assign a Boolean value true in a finished variable.

app.component.html

```
1. <b> Using Observables!</b>
2.
3. <h6 style="margin-bottom: 0">VALUES:</h6>
4. <div *ngFor="let value of myArray">{{ value }}</div>
5.
6. <div style="margin-bottom: 0">ERRORS: {{ errors }}</div>
7.
8. <div style="margin-bottom: 0">FINISHED: {{ finished
   }}</div>
9.
10. <button style="margin-top: 2rem"
   (click)="fetchData()">Fetch Data</button>

11.
```

Line 4: ngFor loop is iterated on myArray which will display the values on the page

Line 6: {{ errors }} will render the value of errors property if any

Line 8: Displays finished property value when complete() method of Observable is executed

Line 10: Button click event is bound with fetchData() method which is invoked and creates an observable with a stream of numeric values

Output:

- Most front-end applications communicate with backend services using HTTP Protocol
- While making calls to an external server, the users must continue to be able to interact with the page. That is, the page should not freeze until the HTTP request returns from the external server. So, all HTTP requests are asynchronous.
- **HttpClient** from @angular/common/http to communicate must be used with backend services.
- Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable APIs, and streamlined error handling.
- **HttpClientModule** must be imported from @angular/common/http in the module class to make HTTP service available to the entire module. Import HttpClient service class into a component's constructor. HTTP methods like get, post, put, and delete are made used off.
- JSON is the default response type for HttpClient

Making a GET request:

The following statement is used to fetch data from a server

```
this.http.get(url)

```

http.get by default returns an observable

Using Server communication in the example used for custom services:

Add HttpClientModule to the **app.module.ts** to make use of HttpClient class

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

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

```
10.
```

Line 2: imports HttpClientModule from @angular/common/http module

Line 6: Includes HttpClientModule class to make use of HTTP calls

Add getBooks() method to BookService class in **book.service.ts** file as shown below

```
1. import { Injectable } from '@angular/core';
2. import { HttpClient, HttpErrorResponse, HttpHeaders } from
   '@angular/common/http';
3. import { Observable, throwError } from 'rxjs';
4. import { catchError, tap } from 'rxjs/operators';
5.
6. import { Book } from './book';
7.
8. @Injectable({
9.     providedIn:'root'
10. })
11. export class BookService {
12.
13.   constructor(private http: HttpClient) { }
14.
15.   getBooks(): Observable<Book[]> {
16.     return
   this.http.get<Book[]>('http://localhost:3020/bookList').pip
   e(
17.       tap((data: any) => console.log('Data Fetched:' +
   JSON.stringify(data))),
18.       catchError(this.handleError));
19.   }
20. ...

21. }
```

Line 2: Imports HttpClient class from @angular/common/http module.

Line 3: Imports Observable class from rxjs module

Line 4: Imports rxjs operators

Line 13: Injects HttpClient class into a service class

Line 16-18: Makes an asynchronous call (ajax call) by using the get() method of HttpClient class. This method makes an asynchronous call to the server URL and fetches the data. HttpClient receives the JSON response as of type object. To know the actual structure of the response, an interface must be created and specified that interface name as a type parameter i.e., get<Book[]>. The **pipe** function defines a comma-separated sequence of operators. Here a sequence of observables is defined by listing operators as arguments to pipe function instead of dot operator chaining. The **tap** operator is to execute some statements once a response is ready which is mostly used for debugging purposes and **catchError** operator is used to handling the errors.

Line 18: handleError is an error-handling method that throws the error message back to the component

Error handling

- What happens if the request fails on the server, or if a poor network connection prevents it from even reaching the server?
- There are two types of errors that can occur. The server might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error responses.
- Or something could go wrong on the client-side such as a network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors produce JavaScript ErrorEvent objects.
- HttpClient captures both kinds of errors in its HttpErrorResponse and it can be inspected for the response to find out what really happened.
- There must be error inspection, interpretation, and resolution in service not in the component.

Add the following error handling code in the **book.service.ts** file

```
1. import { Injectable } from '@angular/core';
2. import { HttpClient, HttpErrorResponse, HttpHeaders } from
   '@angular/common/http';
```

```
3. import { catchError, tap } from 'rxjs/operators';
4. import { Observable, throwError } from 'rxjs';
5. import { HttpErrorResponse } from '@angular/common/http';
6.
7. import { Book } from './book';
8.
9. @Injectable({
10.     providedIn: 'root'
11. })
12. export class BookService {
13.
14.   ...
15.
16.   private handleError(err: HttpErrorResponse):
  Observable<any> {
17.     let errMsg = '';
18.     if (err.error instanceof Error) {
19.       // A client-side or network error occurred. Handle
  it accordingly.
20.       console.log('An error occurred:',
  err.error.message);
21.       errMsg = err.error.message;
22.     } else {
23.       // The backend returned an unsuccessful response
  code.
24.       // The response body may contain clues as to what
  went wrong,
25.       console.log(`Backend returned code ${err.status}`);
26.       errMsg = err.error.status;
27.     }
28.     return throwError(()=>errMsg);
29.   }
30. }

31.
```

Line 5: HttpErrorResponse module class should be imported to understand the nature of the error thrown

Line 18-21: An instance of Error object will be thrown if any network or client-side error is thrown

Line 22-26: Handling of errors due to unsuccessful response codes from the backend

Modify the code in the **book.component.ts** file as shown below

```
1. ...
2. export class BookComponent implements OnInit {
3.
4.   books!: Book[];
5.   errorMessage!: string;
6.
7.   constructor(private bookService: BookService) { }
8.
9.   getBooks() {
10.    this.bookService.getBooks().subscribe({
11.       next:  books => this.books = books,
12.       error:error => this.errorMessage = <any>error
13.     })
14.   }
15.   ngOnInit() {
16.     this.getBooks();
17.   }
18. }

19.
```

Line 7: Inject the BookService class into the component class through the constructor

Line 9-14: Invokes the service class method getBooks() which makes an HTTP call to the books.json file. The getBooks() of the service class returns an Observable.

An observable in Angular begins to publish values only when someone has subscribed to it. To retrieve the value contained in the Observable returned by getBooks() of service, subscribe to the observable by calling the `subscribe()` method and pass an observer object which can listen to the three types of notifications that an observable can send: next, error and complete.

Notification Type	**Details**
next	Required. Handler for each delivered value. Gets called zero or more times once execution starts.
error	Optional. Handler for handling an error notification. If an error occurs, it stops the execution of the observable instance.

complete	Optional. Handler for handling the execution-completion notification. If any values have been delayed, those can be still delivered to the next handler even after execution is complete.

book.component.html

```
1. ...
2. <ul class="books">
3.   <li *ngFor="let book of books">
4.     <span class="badge">{{book.id}}</span> {{book.name}}
5.   </li>
6. </ul>
7.
8. <div class="error"

   *ngIf="errorMessage">{{errorMessage}}</div>
```

Line 2-6: Displays books details

Line 8: Displays error message when HTTP get operation fails

Output:

Making a POST request:

HttpClient.post() method posts data to the server. It takes two parameters.

- Data - data to be sent to the server
- HttpOptions - to specify the required headers to be sent along with the request.

Add addBook() method to BookService class in **book.service.ts** file as shown below

```
1. import { Injectable } from '@angular/core';
2. import { HttpClient } from '@angular/common/http';
3. import { Observable, throwError } from 'rxjs';
4. import { catchError, tap } from 'rxjs/operators';
5.
6. import { Book } from './book';
7.
8. @Injectable({
9.     providedIn: 'root'
10. })
11. export class BookService {
12.
13.   constructor(private http: HttpClient) { }
14.   ...
15.
16.   addBook(book: Book): Observable<any> {
17.     const options = new HttpHeaders({ 'Content-Type':
  'application/json' });
18.     return this.http.post('http://localhost:3020/addBook',
  book, { headers: options }).pipe(
19.       catchError(this.handleError));
20.   }
21.
22. ...

23. }
```

Line 16-20: Makes an asynchronous call (ajax call) by using the post() method of HttpClient class. This method makes an asynchronous call to the server URL and sends the data along with the headers. HttpClient receives the JSON response as of type object. The Pipe function lets you define a comma-separated sequence of operators. Here, a sequence of observables is defined by listing operators as arguments to pipe function instead of dot operator chaining. catchError operator is used to handle the errors.

Line 19: handleError is an error-handling method that throws the error message back to the component

Modify the code in the **book.component.ts** file as shown below

```
1. ...
2. export class BookComponent implements OnInit {
3.
```

```
4.   books!: Book[];
5.   errorMessage!: string;
6.
7.   constructor(private bookService: BookService) { }
8.
9.   getBooks() {
10.     this.bookService.getBooks().subscribe({
11.       next:  books => this.books = books,
12.       error:error => this.errorMessage = <any>error
13.     })
14.   }
15.   addBook(bookId: string, name: string): void {
16.     let id=parseInt(bookId)
17.     this.bookService.addBook({id, name })
18.       .subscribe({next:(book: any) =>
  this.books.push(book)});
19.   }
20.   ngOnInit(): void {
21.     this.getBooks();
22.   }

23. }
```

Line 7: Inject the BookService class into the component class through the constructor

Line 15-19: Invokes the service class method addBook() which makes an HTTP call to server URL and the Observable containing the response is returned. You can subscribe to the observable and use the 'next' callback to handle the successfully returned value, as needed.

Making a PUT request

HttpClient.put() method completely replaces the resource with the updated data. It is like POST requests except for updating an existing resource.

Add updateBook() method to BookService class in **book.service.ts** file as shown below:

```
1. import { Injectable } from '@angular/core';
2. import { HttpClient } from '@angular/common/http';
3. import { Observable, throwError } from 'rxjs';
```

```
4. import { catchError, tap } from 'rxjs/operators';
5.
6. import { Book } from './book';
7.
8. @Injectable({
9.    providedIn: 'root'
10. })
11. export class BookService {
12.
13.   constructor(private http: HttpClient) { }
14.   ...
15.
16.   updateBook(book: Book): Observable<any> {
17.     const options = new HttpHeaders({ 'Content-Type':
    'application/json' });
18.     return
    this.http.put<any>('http://localhost:3020/update', book, {
    headers: options }).pipe(
19.       tap((_: any) => console.log(`updated hero
    id=${book.id}`)),
20.       catchError(this.handleError)
21.     );
22.   }
23.
24. ...

25. }
```

Line 16-20: Makes an asynchronous call (ajax call) by using the put() method of HttpClient class. This method makes an asynchronous call to the server URL and sends the data along with the headers as that of POST requests. HttpClient receives the JSON response as of type object. Pipe function defines a comma-separated sequence of operators. Here a sequence of observables is defined by listing operators as arguments to pipe function instead of dot operator chaining. **tap** operator is to execute some statements once a response is ready which is mostly used for debugging purposes and **catchError** operator is used to handle the errors.

Line 20: handleError is an error-handling method that throws the error message back to the component.

Modify the code in the **book.component.ts** file as shown below

```
1. ...
2. export class BookComponent implements OnInit {
3.
4.   books!: Book[];
5.   errorMessage!: string;
6.
7.   constructor(private bookService: BookService) { }
8.
9.   getBooks() {
10.      this.bookService.getBooks().subscribe({
11.        next:  books => this.books = books,
12.        error:error => this.errorMessage = <any>error
13.      })
14.    }
15.    addBook(bookId: string, name: string): void {
16.      let id=parseInt(bookId)
17.      this.bookService.addBook({id, name })
18.        .subscribe({next:(book: any) =>
   this.books.push(book)});
19.    }
20.    updateBook(bookId: string, name: string): void {
21.      let id=parseInt(bookId)
22.      this.bookService.updateBook({ id, name })
23.        .subscribe({next:(book: any) => this.books = book});
24.    }
25.    ngOnInit(): void {
26.      this.getBooks();
27.    }
28. }

29.
```

Line 7: Inject the BookService class into the component class through the constructor.

Line 20-24: Invokes the service class method updateBook() which makes an HTTP call to server URL and the Observable containing the response is returned. You can subscribe to the observable and use the 'next' callback to handle the successfully returned value, as needed.

Making a DELETE request :

HttpClient.delete() method deletes the resource by passing the bookId parameter in the request URL.

Add deleteBook() method to BookService class in b**ook.service.ts** file as shown below

```
1. import { Injectable } from '@angular/core';
2. import { HttpClient } from '@angular/common/http';
3. import { Observable, throwError } from 'rxjs';
4. import { catchError, tap } from 'rxjs/operators';
5.
6. import { Book } from './book';
7.
8. @Injectable({
9.      providedIn: 'root'
10. })
11. export class BookService {
12.
13.   constructor(private http: HttpClient) { }
14.
15.   booksUrl = 'http://localhost:3020/bookList';
16.   ...
17.
18.   deleteBook(bookId: number): Observable<any> {
19.     const url = `${this.booksUrl}/${bookId}`;
20.     return this.http.delete(url).pipe(
21.       catchError(this.handleError));
22.   }
23.
24. ...

25. }
```

Line 18-21: Makes an asynchronous call (ajax call) by using delete() method of HttpClient class. HttpClient receives the JSON response as of type object. Pipe function defines a comma-separated sequence of operators. catchError operator is used to handle the errors.

Line 21: handleError is an error-handling method that throws the error message back to the component.

Modify the code in the **book.component.ts** file as shown below

```
1. ...
2. export class BookComponent implements OnInit {
3.
```

```
4.   books!: Book[];
5.   errorMessage!: string;
6.
7.   constructor(private bookService: BookService) { }
8.
9.   getBooks() {
10.     this.bookService.getBooks().subscribe({
11.       next:  books => this.books = books,
12.       error:error => this.errorMessage = <any>error
13.     })
14.   }
15.   addBook(bookId: string, name: string): void {
16.     let id=parseInt(bookId)
17.     this.bookService.addBook({id, name })
18.       .subscribe({next:(book: any) =>
  this.books.push(book)});
19.   }
20.   updateBook(bookId: string, name: string): void {
21.     let id=parseInt(bookId)
22.     this.bookService.updateBook({ id, name })
23.       .subscribe({next:(book: any) => this.books = book});
24.   }
25.   deleteBook(bookId: string): void {
26.     let id=parseInt(bookId)
27.     this.bookService.deleteBook(id)
28.       .subscribe({next:(book: any) => this.books = book});
29.   }
30.   ngOnInit(): void {
31.     this.getBooks();
32.   }
33. }

34.
```

Line 7: Inject the BookService class into the component class through the constructor

Line 25-29: Invokes the service class method deleteBook() which makes an HTTP call to server URL and the Observable containing the response is returned. You can subscribe to the observable and use the 'next' callback to handle the successfully returned value, as needed.

Some of the best practices of server communication are:

☑ Always use services to talk to the server as components' responsibility is only to present the data.

☑ Data services should be responsible for asynchronous calls, local storage, or any other data operations as it will become easier to test the data calls.

☑ The information like headers, HTTP methods, caching, error handling, etc., are irrelevant to components and they need to be in a service class. A service encapsulates these details and it will be easier to test the components with mock service implementations.

Nested routes

Nested Routes

In Angular, you can also create sub-routes or child routes for your components which means in an application there will be one root route just like a root component/root module and other routes will be configured for their respective components. Configuring routes module-wise is the best practice to make modular Angular applications.

Steps to create child routes using Angular:

1. Add below code to routing module **book-routing.module.ts** to implement child routing in the book module.

```
1. import { NgModule } from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3. import { BookComponent } from './book.component';
4. import { LoginGuardService } from '../login/login-
   guard.service';
5. import { DashboardComponent } from
   '../dashboard/dashboard.component';
6. import { BookDetailComponent } from '../book-detail/book-
   detail.component';
7. const bookRoutes: Routes = [
8.     {
9.       path: '',
10.        component: BookComponent,
11.        children: [
```

```
12.         { path: 'dashboard', component: DashboardComponent
  },
13.         { path: 'detail/:id', component:
  BookDetailComponent }
14.       ],
15.       canActivate: [LoginGuardService]
16.     }];
17. @NgModule({
18.     imports: [RouterModule.forChild(bookRoutes)],
19.     exports: [RouterModule]
20. })
21. export class BookRoutingModule { }

22.
```

Line 7-16: Child routes can be defined using children property of a route along with path & component properties. DashboardComponent, BookDetailComponent can be accessed using books/dashboard and books/detail/: id paths respectively.

Line 18: imports array contains the imported modules to use in the book module. forChild() method adds routing configurations to the book submodule instead of the root module.

Line 19: exports array contains classes that are exported from the current module.

2. Import BookRoutingModule in the submodule **book.module.ts** as shown below :

```
1. import { NgModule } from '@angular/core';
2. import { BookComponent } from './book.component';
3. import { BookRoutingModule } from './book-routing.module';
4. import { FormsModule } from '@angular/forms';
5. import { BookDetailComponent } from '../book-detail/book-
  detail.component';
6. import { DashboardComponent } from
  '../dashboard/dashboard.component';
7. import { CommonModule } from '@angular/common';
8. @NgModule({
9.   imports: [ CommonModule, BookRoutingModule, FormsModule],
10.    declarations: [BookComponent, BookDetailComponent,
  DashboardComponent]
11. })
12. export class BookModule { }
```

```
13.
```

3. Open **book.component.html** and add nested router outlet as shown below.

```
1. <br/>
2.  <h2>MyBooks</h2>
3.  <ul class="books">
4.    <li *ngFor="let book of books "
   (click)="gotoDetail(book)">
5.        <span class="badge">{{book.id}}</span> {{book.name}}
6.    </li>
7.  </ul>
8.  <div>
9.    <router-outlet></router-outlet>
10.       </div>
11.       <div class="error"
   *ngIf="errorMessage">{{errorMessage}}</div>

12.
```

Line 10: The nested router-outlet is used to render components of this submodule. DashboardComponent and BookDetailComponent can now be rendered inside Book.component.html. If this nested router-outlet is not added, child routes will be added to the parent router outlet of the application.

4. Add the below code in **app.component.html** to add a link for accessing books.

```
1. <h1>{{title}}</h1>
2. <nav>
3.     <a [routerLink]='["/books"]'
   routerLinkActive="active">Books</a>
4.     <a [routerLink]='["/books/dashboard"]'
   routerLinkActive="active">Dashboard</a>
5. </nav>
6. <router-outlet></router-outlet>

7.
```

5. Update **app-routing.module.ts** with below code:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
const appRoutes: Routes = [
    { path: '', redirectTo: '/login', pathMatch: 'full' },
    { path: 'login', component: LoginComponent },
    { path: 'books', loadChildren: () => import('./book/book.module').then(m => m.BookModule) },
    { path: '**', component: PageNotFoundComponent }
];
@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes)
    ],
    exports: [
        RouterModule
    ]
})
export class AppRoutingModule { }

```

6. Update app.module.ts as below:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';
import { LoginComponent } from './login/login.component';
@NgModule({
  imports: [BrowserModule, HttpClientModule, ReactiveFormsModule, AppRoutingModule],
  declarations: [AppComponent, LoginComponent],
  providers: [],
```

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

```

7.Add gotoDetail() method in **book.component.ts** as below:

```
...
 gotoDetail(book: Book): void {
    this.router.navigate(['/books/detail', book.id]);
  }

...
```

8. Update dashboard.component.html

```
<h3>Top Books</h3>
<div class="grid grid-pad">
  <div *ngFor="let book of books"
(click)="gotoDetail(book)" class="col-1-4">
    <div class="module book">
      <h4>{{ book.name }}</h4>
    </div>
  </div>
</div>

```

9. Update dashboard.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { Book } from '../book/book';
import { BookService } from '../book/book.service';
@Component({
  selector: 'app-dashboard',
```

```
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.css']
})
export class DashboardComponent implements OnInit {
  books: Book[] = [];
  constructor(
    private router: Router,
    private bookService: BookService) { }
  ngOnInit(): void {
    this.bookService.getBooks()
      .subscribe(books => this.books = books.slice(1, 5));
  }
  gotoDetail(book: Book): void {
    this.router.navigate(['/books/detail', book.id]);
  }
}

```