

Unit-5

Angular JS

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

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.

What is Angular?

Single Page Application (SPA)

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.

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).

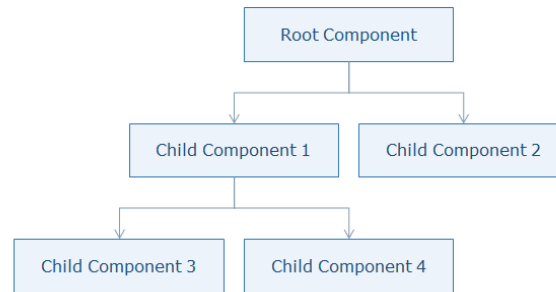
Why most developers prefer TypeScript for Angular?

- 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.

Features of Angular:

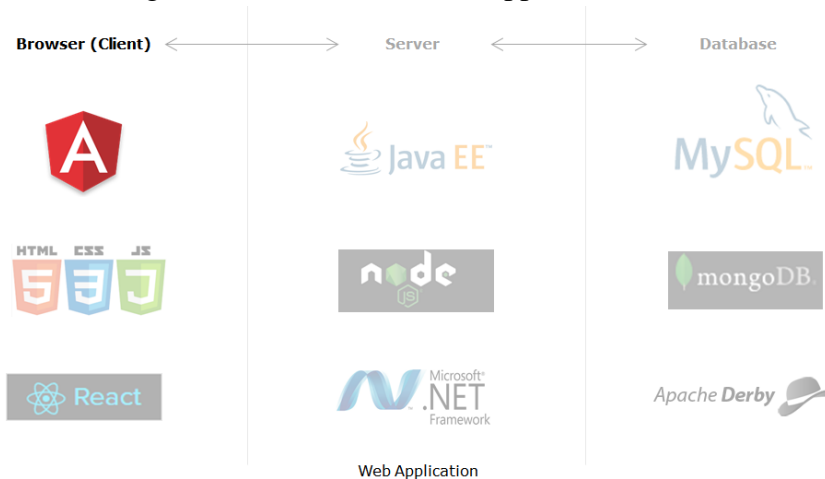
- **Easier to learn:** Angular is more modern and easier for developers to learn. It is a more streamlined framework where developers will be focusing on writing JavaScript classes.
- **Good IDE support:** Angular is written in TypeScript which is a superset of JavaScript and supports all ECMAScript 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.



Angular in Web Application Stack

Let us see where Angular fits in the entire web application stack.



Angular places itself on the client-side in the complete application stack and provides a completely client-side solution for quick application development. Angular has absolutely no dependencies and also jells perfectly with any possible server-side technology like Java, NodeJS, PHP, etc., and any database like MongoDB, MySql.

Angular Application Setup:

You will learn the Angular course by building the mCart application. Below is the roadmap to achieve it.

Angular Development Environment Setup

To develop an application using Angular on a local system, you need to set up a development environment that includes the installation of:

- Node.js (^12.20.2 || ^14.15.5 || ^16.10.0) and npm (min version required 6.13.4)

- Angular CLI
- Visual Studio Code

1. Steps to install Node.js

Install Node.js (^12.20.2 || ^14.15.5 || ^16.10.0).

To check whether the Node is installed or not in your machine, go to the Node command prompt and check the Node version by typing the following command.

```
node -v
```

It will display the version of the node installed.

```
C:\Users\username>node -v  
v16.13.0
```

2. Steps to install Angular CLI

Angular CLI can be installed using node package manager as shown below:

```
D:\> npm install -g @angular/cli
```

Test successful installation of Angular CLI using the following command

Note: Sometimes additional dependencies might throw an error during CLI installation but still check whether CLI is installed or not using the following command. If the version gets displayed, you can ignore the errors.

```
D:\> ng v
```

```
D:\>ng v  
  
Angular CLI  
  
Angular CLI: 13.1.2  
Node: 16.13.0  
Package Manager: npm 8.1.0  
OS: win32 x64  
  
Angular:  
...  
  
Package                                Version  
-----  
@angular-devkit/architect              0.1301.2 (cli-only)  
@angular-devkit/core                   13.1.2 (cli-only)  
@angular-devkit/schematics             13.1.2 (cli-only)  
@schematics/angular                    13.1.2 (cli-only)
```

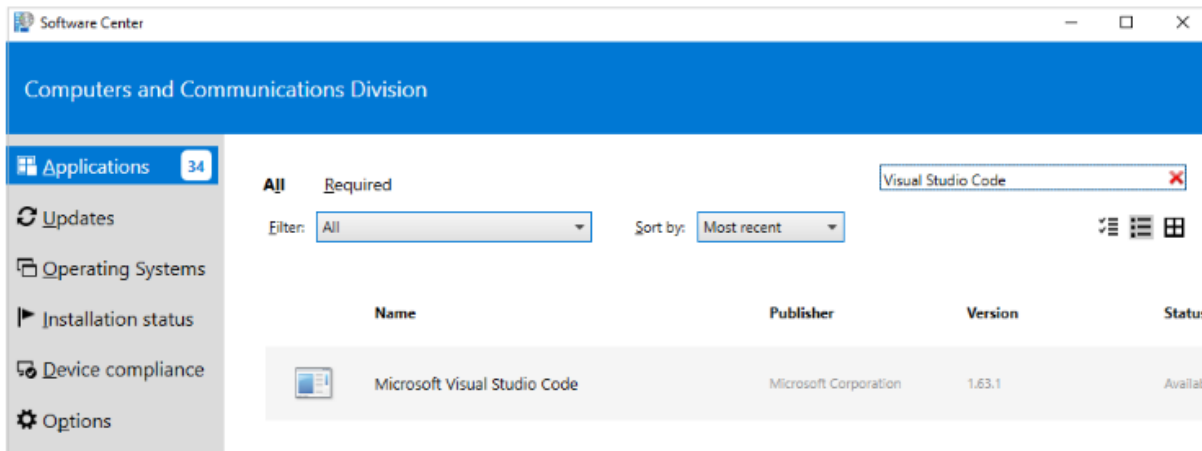
Angular CLI is a command-line interface tool to build Angular applications. It makes application development faster and easier to maintain.

Using CLI, you can create projects, add files to them, and perform development tasks such as testing, bundling, and deployment of applications.

Command	Purpose
npm install -g @angular/cli	Installs Angular CLI globally
ng new <project name>	Creates a new Angular application
ng serve --open	Builds and runs the application on lite-server and launches a browser
ng generate <name>	Creates a class, component, directive, interface, module, pipe, and service
ng build	Builds the application
ng update @angular/cli @angular/core	Updates Angular to latest version

3. Steps to install Visual Studio Code

Install Visual Studio code from software house/software center as shown below or take help from CCD to get it installed:



You will learn the Angular course by building the mCart application. Below is the roadmap to achieve it.

Angular CLI is a command-line interface tool to build Angular applications. It makes application development faster and easier to maintain.

Using CLI, you can create projects, add files to them, and perform development tasks such as testing, bundling, and deployment of applications.

Command	Purpose
npm install -g @angular/cli	Installs Angular CLI globally
ng new <project name>	Creates a new Angular application
ng serve --open	Builds and runs the application on lite-server and launches a browser
ng generate <name>	Creates class, component, directive, interface, module, pipe, and service
ng build	Builds the application

ng update @angular/cli @angular/core	Updates Angular to a newer version
--------------------------------------	------------------------------------

Creating an Angular Application

Highlights:

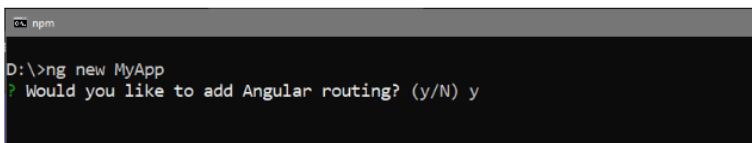
- Creating an Angular application using Angular CLI
- Exploring the Angular folder structure

Demosteps:

1. Create an application with the name 'MyApp' using the following CLI command

```
D:\>ng new MyApp
```

2. The above command will display two questions. The first question is as shown below. Typing 'y' will create a routing module file (app-routing.module.ts).



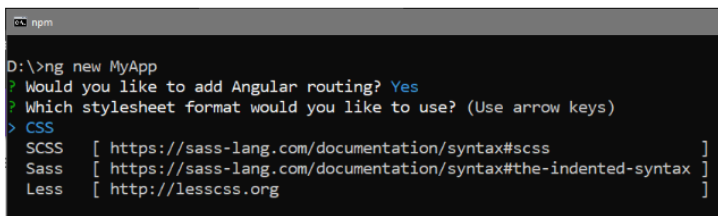
```

D:\>ng new MyApp
? Would you like to add Angular routing? (y/N) y

```

Next, you will learn about the app-routing.module.ts file.

3. The next question is to select the stylesheet to use in the application. Select CSS and press Enter as shown below:

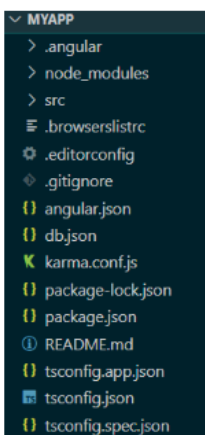


```

D:\>ng new MyApp
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
  SCSS [ https://sass-lang.com/documentation/syntax#scss ]
  Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less [ http://lesscss.org ]

```

This will create the following folder structure with the dependencies installed inside the node_modules folder.



Note: If the above command gives errors while installing dependencies, navigate to the project folder in the Node command prompt and run "npm install" to install the dependencies manually.

File / Folder	Purpose
node_modules/	Node.js creates this folder and puts all npm modules installed as listed in package.json

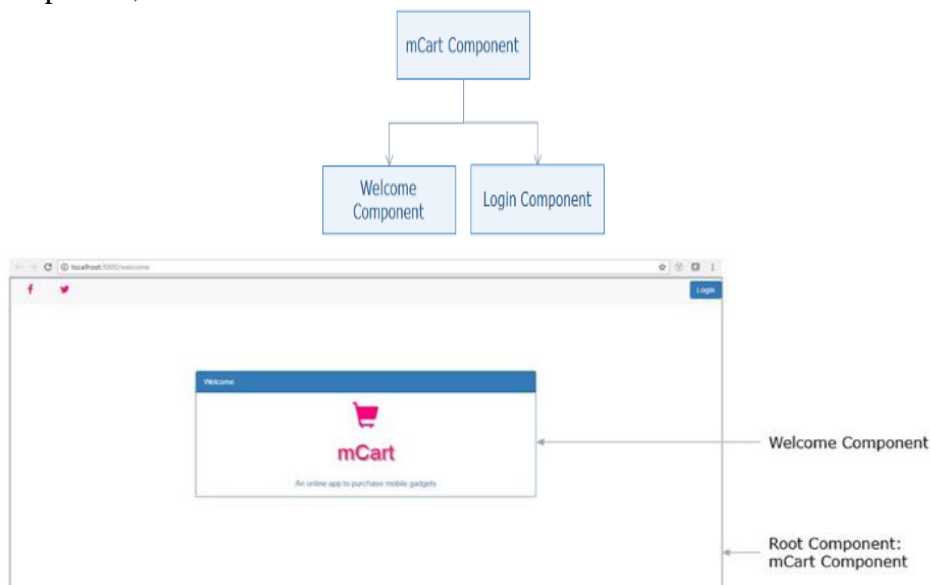
File / Folder	Purpose
src/	All application-related files will be stored inside it
angular.json	Configuration file for Angular CLI where we set several defaults and also configure what files to be included during project build
package.json	This is a node configuration file that contains all dependencies required for Angular
tsconfig.json	This is the Typescript configuration file where we can configure compiler options
.angular	From Angular version 13.0.0, .angular folder is generated in the root. This folder caches the builds and is ignored by git.

Creating Components and Modules

Why Components in Angular?

- A component is the basic building block of an Angular application
- It emphasize the separation of concerns and each part of the Angular application can be written independently of one another
- It is reusable

For example, observe in the **mCart** application the topmost component is the mCart component(AppComponent) which consists of child components called Welcome component, Login component, etc.



Creating a Component

- 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. }
```

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>
```

Line 1: 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

- 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.

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 { }
```

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

Bootstrapping Root Module

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));
```

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

Loading root component in HTML Page

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>
```

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.

Executing Angular Application

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
```

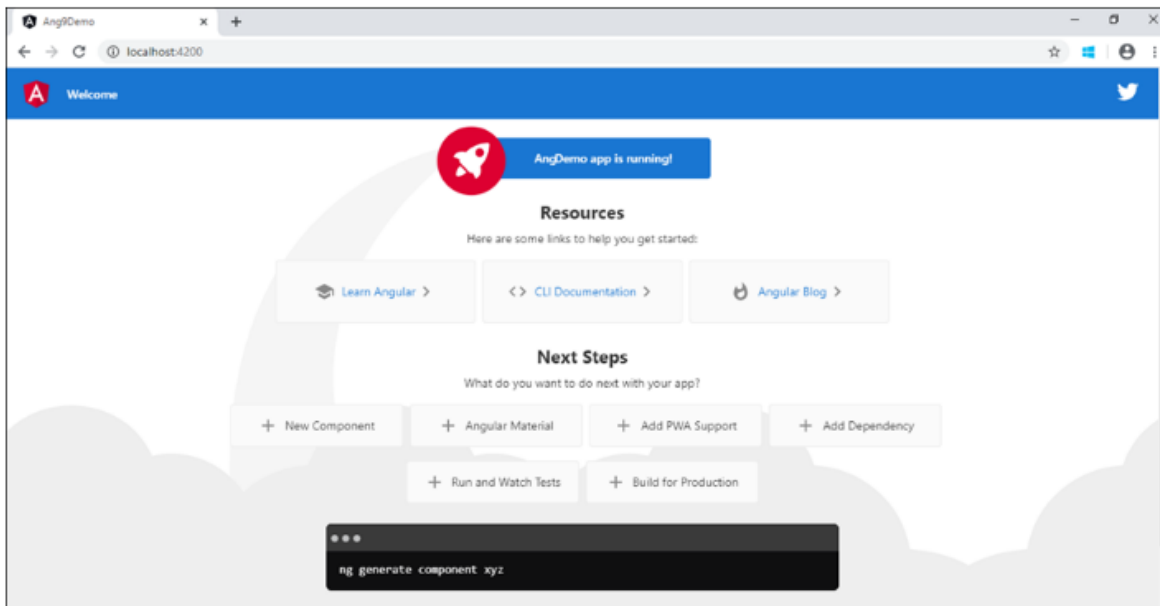
- **ng serve** will build and run the application
- **--open** option will show the output by opening a browser automatically with the default port.

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



Introduction to Templates

- Templates separate the view layer from the rest of the framework.
- You can change the view layer without breaking the application.
- Templates in Angular represents a view and its role is to display data and change the data whenever an event occurs
- The default language for templates is HTML

Creating a template

Template can be defined in two ways:

- Inline Template
- External Template

Inline Template

You can create an inline template in a component class itself using the template property of the @Component decorator.

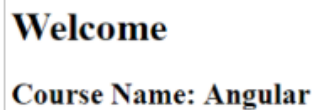
app.component.ts

```
1. import { Component } from '@angular/core';  
2. @Component({
```

```
3. selector: 'app-root',
4. template: `
5. <h1> Welcome </h1>
6. <h2> Course Name: {{ courseName }}</h2>
7. `,
8. styleUrls: ['./app.component.css']
9. })
10. export class AppComponent {
11.   courseName = "Angular";
12. }
```

Line 4-8: You can even write HTML code inside the component using the template property. Use backtick character (`) for multi-line strings.

Output:



Output screenshot showing the rendered HTML: **Welcome** and **Course Name: Angular**.

External Template

- By default, Angular CLI uses the external template.
- It binds the external template with a component using **templateUrl** option.

Example

app.component.html

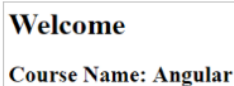
```
1. <h1> Welcome </h1>
2. <h2> Course Name: {{ courseName }}</h2>
```

app.component.ts

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

Line 5: templateUrl property is used to bind an external template file with the component

Output:



Output screenshot showing the rendered HTML: **Welcome** and **Course Name: Angular**.

Best Practices - Coding Style Rules



Always create a separate template file when the template code is more than 3 lines.

- ✓ The path mentioned in the templateUrl property should be always relative.

Elements of Template

The basic elements of template syntax:

- HTML
- Interpolation
- Template Expressions
- Template Statements

HTML

Angular uses HTML as a template language. In the below example, the template contains pure HTML code.

Interpolation

Interpolation is one of the forms of data binding where component's data can be accessed in a template. For interpolation, double curly braces {{ }} is used.

Template Expressions

- The text inside {{ }} is called as template expression.

1. {{ expression }}

- Angular first evaluates the expression and returns the result as a string. The scope of a template expression is a component instance.
- That means, if you write {{ courseName }}, courseName should be the property of the component to which this template is bound.

Template Statement

- Template Statements are the statements that respond to a user event.

1. (event) = statement

- For example (click) = "changeName()"
- This is called event binding. In Angular, all events should be placed in ().

Example:

app.component.ts

```
1. ...
2. export class AppComponent {
3.   courseName = "Angular";
4.
5.   changeName() {
6.     this.courseName = "TypeScript";
7.   }
8. }
```

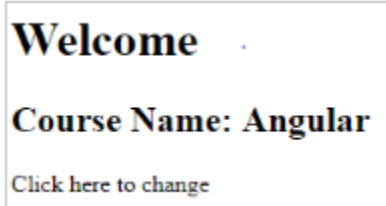
Line 5-7: changeName is a method of AppComponent class where you are changing courseName property value to "TypeScript"

app.component.html

```
1. <h1> Welcome </h1>
2. <h2> Course Name: {{ courseName }}</h2>
```

```
3. <p (click)="changeName()">Click here to change</p>
```

Line 3: changeName() method is bound to click event which will be invoked on click of a paragraph at run time. This is called event binding.

Output:

When a user clicks on the paragraph, the course name will be changed to 'Typescript'.

**Change Detection****How does Angular detect the changes and update the application at the respective places?**

Angular uses its change detection mechanism to detect the changes and update the application at the respective places. Angular applications **run faster** than Angular 1.x applications due to the improved **change detection mechanism**.

What is the change detection mechanism, and how it helps to run Angular applications so fast?

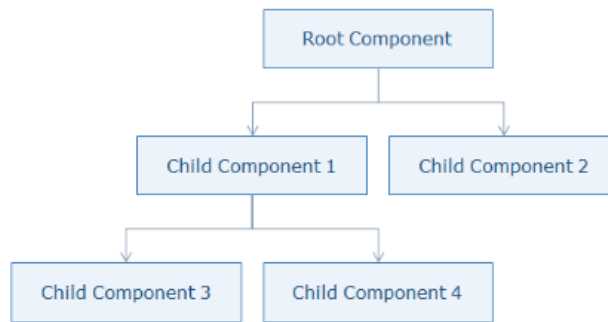
- Change Detection is a process in Angular that keeps views in sync with the models.
- In Angular, the flow is unidirectional from top to bottom in a component tree. A change in a web application can be caused by events, Ajax calls, and timers which are all asynchronous.

Who informs Angular about the changes?

- **Zones** inform Angular about the changes in the application. It automatically detects all asynchronous actions at run time in the application.

What happens when a change is detected? What does Angular do when a change is detected?

- Angular runs a change detector algorithm on each component from top to bottom in the component tree. This change detector algorithm is automatically generated at run time which will check and update the changes at appropriate places in the component tree.
- Angular is very fast though it goes through all components from top to bottom for every single event as it generates VM-friendly code. Due to this, Angular can perform hundreds of thousands of checks in a few milliseconds.



Structural Directives:

Now that you are familiar with the concept of templates, let us now understand directives in Angular.

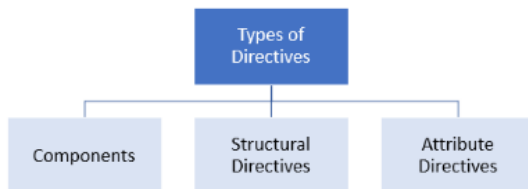
- Directives are used to change the behavior of components or elements. It can be used **in the form of HTML attributes**.
- You can create directives using classes attached with @Directive decorator which adds metadata to the class.

Why Directives?

- It modify the DOM elements
- It creates reusable and independent code
- It is used to create custom elements to implement the required functionality

Types of Directives

There are three types of directives available in Angular



Components

- Components are directives with a template or view.
- @Component decorator is actually @Directive with templates

Structural Directives

- A Structural directive changes the DOM layout by adding and removing DOM elements.

1. *directive-name = expression

- Angular has few built-in structural directives such as:
 - ngIf
 - ngFor
 - ngSwitch

ngIf

ngIf directive renders components or elements conditionally based on whether or not an expression is true or false.

Syntax:

1. *ngIf = "expression"

ngIf directive **removes the element from the DOM** tree.

Example:

app.component.ts

```
1. ...
2. export class AppComponent {
3.   isAuthenticated!: boolean;
4.   submitted = false;
5.   userName!: string;
6.
7.   onSubmit(name: string, password: string) {
8.     this.submitted = true;
9.     this.userName = name;
10.    if (name === "admin" && password === "admin") {
11.      this.isAuthenticated = true;
12.    } else {
13.      this.isAuthenticated = false;
14.    }
15.  }
16. }
```

Line 5: Notice the use of ! with the property username. This is called adding *definite assertion for variables during declaration*.

Definite assertion for variables

Angular CLI creates all new workspaces and projects with strict mode enabled. This leads to an improved compile time check for properties. If any property is not initialized either by providing a default value or by initializing within the constructor or if a parameter is not mapped to a property, the below error gets generated:

property <<property>> has no initializer and is not definitely assigned in the constructor.

This is known as a definite assignment error because there is no definite assignment for that property. The correct fix to this is to assign the property with a value in the constructor. An alternate is to permit the property with no definite assignment by using the definite assignment assertion. You can do that by adding ! to the property name.

userName in Line 5 of the above code doesn't have a value initialized and hence the use of !.

Line 7 -14: onSubmit method is invoked when the user clicks on the submit button in the template. This method checks the username and password values entered are correct or not and accordingly assigns a boolean value to isAuthenticated variable.

app.component.html

```
1. <div *ngIf="!submitted">
2. ...
3. <button (click)="onSubmit(username.value, password.value)">Login</button>
4. </div>
5.
6. <div *ngIf="submitted">
7. <div *ngIf="isAuthenticated; else failureMsg">
```



```

8. <h4>Welcome {{ userName }}</h4>
9. </div>
10. <ng-template #failureMsg>
11. <h4>Invalid Login !!! Please try again...</h4>
12. </ng-template>
13. <button type="button" (click)="submitted = false">Back</button>
14. </div>

```

Line 7: Username will be displayed if 'isAuthenticated' property value is true otherwise it will be an error message.

Output:

After entering the correct credentials and clicking on the Login button.



After entering incorrect credentials and clicking on the Login button.



ngFor

ngFor directive is used to iterate over-collection of data i.e., arrays

Syntax:

```
1. *ngFor = "expression"
```

Example:

app.component.ts

```

1. ...
2. export class AppComponent {
3.   courses: any[] = [
4.     { id: 1, name: "TypeScript" },
5.     { id: 2, name: "Angular" },
6.     { id: 3, name: "Node JS" },
7.     { id: 1, name: "TypeScript" }
8.   ];
9. }

```

Line 3-8: Creating an array of objects

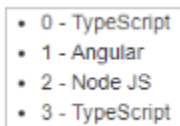
app.component.html

```
1. <ul>
2. <li *ngFor="let course of courses; let i = index">
3.   {{i}} - {{ course.name }}
4. </li>
5. </ul>
```

Line 2: ngFor iterates over courses array and displays the value of name property of each course. It also stores the index of each item in a variable called i

Line 3: {{ i }} displays the index of each course and course.name displays the name property value of each course

Output:



ngSwitch

- ngSwitch adds or removes DOM trees when their expressions match the switch expression. Its syntax is comprised of two directives, an attribute directive, and a structural directive.
- It is very similar to a switch statement in JavaScript and other programming languages.

Example:

app.component.ts

```
1. ...
2. export class AppComponent {
3.   choice = 0;
4.   nextChoice() {
5.     this.choice++;
6.   }
7. }
```

Line 3: Creates a choice property and initializes it to zero

Line 5-7: nextChoice() increments the choice when invoked

app.component.html

```
1. ...
2. <div [ngSwitch]="choice">
3.   <p *ngSwitchCase="1">First Choice</p>
4.   <p *ngSwitchCase="2">Second Choice</p>
5.   <p *ngSwitchCase="3">Third Choice</p>
6.   <p *ngSwitchCase="2">Second Choice Again</p>
7.   <p *ngSwitchDefault>Default Choice</p>
8. </div>
9. ...
```

Line 2: ngSwitch takes the value and based upon the value inside the choice property, it executes *ngSwitchCase. Paragraph elements will be added/removed from the DOM based on the value passed to the switch case.

Output:**Custom Structural Directive**

You can create custom structural directives when there is no built-in directive available for the required functionality. For example, when you want to manipulate the DOM in a particular way which is different from how the built-in structural directives manipulate.

Example:

Create a custom structural directive called 'repeat' which should repeat the element given a number of times. As there is no built-in directive available to implement this, a custom directive can be created.

To create a custom structural directive, create a class annotated with @Directive

```

1. @Directive({
2. })
3. class MyDirective{ }
  
```

Generate a directive called 'repeat' using the following command

```
1. D:\MyApp>ng generate directive repeat
```

This will create two files under the src/app folder with names repeat.directive.ts and repeat.directive.spec.ts (this is for testing). Now the app folder structure will look as shown below:

```

* app
  app.component.css
  app.component.html
  app.component.spec.ts
  app.component.ts
  app.module.ts
  repeat.directive.spec.ts
  repeat.directive.ts
  
```

It also adds repeat directive to the root module i.e., app.module.ts to make it available to the entire module as shown below in Line 7

app.module.ts

```

1. ...
2. import { RepeatDirective } from './repeat.directive';
3. @NgModule({
4.   declarations: [
5.     AppComponent,
6.     RepeatDirective
7.   ],
8.   ...
9. })
10. export class AppModule { }
  
```

Open **repeat.directive.ts** file and add the following code

```
1. import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
2. @Directive({
3.   selector: '[appRepeat]'
4. })
5. export class RepeatDirective {
6.   constructor(private templateRef: TemplateRef<any>, private viewContainer: ViewContainerRef) { }
7.   @Input() set appRepeat(count: number) {
8.     for (let i = 0; i < count; i++) {
9.       this.viewContainer.createEmbeddedView(this.templateRef);
10.    }
11.  }
12. }
```

Line 3: Annotate the class with @Directive which represents the class as a directive and specify the selector name inside brackets

Line 8: Create a constructor and inject two classes called TemplateRef which acquires <ng-template> content and another class called ViewcontainerRef which access the HTML container to add or remove elements from it

Line 10: Create a setter method for an appRepeat directive by attaching @Input() decorator which specifies that this directive will receive value from the component. This method takes the number passed to the appRepeat directive as an argument.

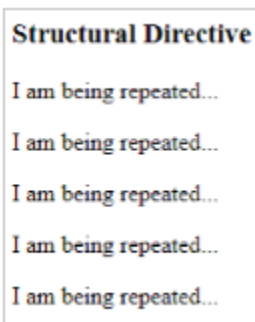
Line 12: As we need to render the elements based on the number passed to the appRepeat directive, run a for loop in which pass the template reference to a createEmbeddedView method which renders the elements into the DOM. This structural directive creates an embedded view from the Angular generated <ng-template> and inserts that view in a view container.

app.component.html

```
1. <h3>Structural Directive</h3>
2. <p *appRepeat="5">I am being repeated...</p>
```

Line 2: appRepeat directive is applied to the paragraph element. It will render five paragraph elements into the DOM.

Output:



Structural Directive

I am being repeated...

I am being repeated...

I am being repeated...

I am being repeated...

I am being repeated...

Attribute Directives

Attribute directives change the appearance/behavior of a component/element.

Following are built-in attribute directives:

- ngStyle
- ngClass

ngStyle

This directive is used to modify a component/element's style. You can use the following syntax to set a single CSS style to the element which is also known as style binding.

```
1. [style.<cssproperty>] = "value"
```

Example:**app.component.ts**

```
1. ...
2. export class AppComponent {
3.   colorName = 'yellow';
4.   color = 'red';
5. }
```

Line 3-4: colorName and color properties are initialized to default values

app.component.html

```
1. <div [style.background-color]="colorName" [style.color]="color">
2.   Uses fixed yellow background
3. </div>
```

Line 1: style.background-color will set the background-color of the text to yellow and style.color directive will set the color of the text to red.

Output:

Uses fixed yellow background

If there are more than one CSS styles to apply, you can use **ngStyle** attribute.

Example:**app.component.ts**

```
1. ...
2. export class AppComponent {
3.   colorName = 'red';
4.   fontWeight = 'bold';
5.   borderStyle = '1px solid black';
6. }
```

Line3-5: Create three properties called colorName, fontWeight and borderStyle and initialize them with some default values

app.component.html

```
1. <p [ngStyle]="{
2.   color:colorName,
3.   'font-weight':fontWeight,
4.   borderBottom: borderStyle
5. }">
6. Demo for attribute directive ngStyle
```

```
7. </p>
```

Line 1-5: ngStyle directive is used here to set multiple CSS styles for the given text

Output:



ngClass

It allows you to dynamically set and change the CSS classes for a given DOM element. Use the following syntax to set a single CSS class to the element which is also known as class binding.

```
1. [class.<css_class_name>] = "property/value"
```

Example:

app.component.ts

```
1. ...
2. export class AppComponent {
3.   isBordered = true;
4. }
```

Line 3: Create a Boolean property called isBordered and initialize it to true

app.component.html

```
1. <div [class.bordered]="isBordered">
2.   Border {{ isBordered ? "ON" : "OFF" }}
3. </div>
```

Line 1: Bind the isBordered property with the CSS class bordered. Bordered CSS class will be applied only if isBordered property evaluates to true.

In **app.component.css**, add the following CSS class

```
1. .bordered {
2.   border: 1px dashed black;
3.   background-color: #eee;
4. }
```

Output:



If you have **more than one CSS classes to apply**, then you can go for ngClass syntax.

Syntax:

```
1. [ngClass] = "{css_class_name1 : Boolean expression, css_class_name2: Boolean expression, .....}"
```

Example:

app.component.ts

```
1. ...
2. export class AppComponent {
```

```
3. isBordered = true;
4. isColor = true;
5. }
```

Line 3-4: Two Boolean properties called isBordered and isColor are initialized to true

app.component.html

```
1. <div [ngClass]="{bordered: isBordered, color: isColor}">
2.   Border {{ isBordered ? "ON" : "OFF" }}
3. </div>
```

Line 1: Two CSS classes called bordered and color are applied to a div tag. Both the classes are binded with isBordered and isColor properties where the CSS classes will be applied to div tag only if the properties return true

Add the following CSS classes in **app.component.css**

```
1. .bordered {
2.   border: 1px dashed black;
3.   background-color: #eee;
4. }
5. .color {
6.   color: blue;
7. }
```

Output:



Custom Attribute Directive

You can create a custom attribute directive when there is no built-in directive available for the required functionality. For Example, consider the following problem statement:

Problem Statement: Create an attribute directive called 'showMessage' which should display the given message in a paragraph when a user clicks on it and should change the text color to red. As there is no built-in directive available to implement this functionality, you need to go for a custom directive.

To create a custom attribute directive, we need to create a class annotated with @Directive

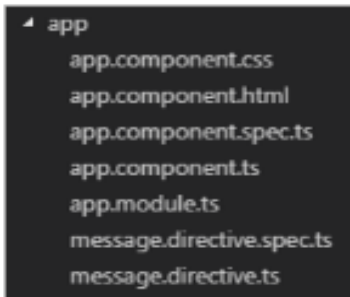
```
1. @Directive({
2.
3. })
4. class MyDirective { }
```

Example:

Generate a directive called 'message' using the following command

```
1. D:\MyApp> ng generate directive message
```

This will create two files under the src\app folder with the names message.directive.ts and message.directive.spec.ts (this is for testing). Now the app folder structure will look as shown below:



It also adds message directive to the root module i.e., **app.module.ts** to make it available to the entire module as shown below

```
1. ...
2. import { MessageDirective } from './message.directive';
3. @NgModule({
4.   declarations: [
5.     AppComponent,
6.     MessageDirective
7.   ],
8.   ...
9. })
10. export class AppModule { }
```

Open the **message.directive.ts** file and add the following code:

```
1. import { Directive, ElementRef, Renderer2, HostListener, Input } from '@angular/core';
2. @Directive({
3.   selector: '[appMessage]',
4. })
5. export class MessageDirective {
6.   @Input('appMessage') message!: string;
7.   constructor(private el: ElementRef, private renderer: Renderer2) {
8.     renderer.setStyle(el.nativeElement, 'cursor', 'pointer');
9.   }
10.  @HostListener('click') onClick() {
11.    this.el.nativeElement.innerHTML = this.message;
12.    this.renderer.setStyle(this.el.nativeElement, 'color', 'red');
13.  }
14. }
```

Line 3: Create a directive class with the selector name as appMessage

Line 8: @Input('appMessage') will inject the value passed to 'appMessage' directive into the 'message' property

Line10: Use constructor injection to inject ElementRef which holds the HTML element reference in which directive is used and Renderer2 which is used to set the CSS styles.

Line 11: Using Renderer2 reference, the cursor is being changed to pointer symbol

Line 14-16: onClick method is invoked when the click event is triggered on the directive which will display the message received and changes the element color to red.

app.component.ts

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

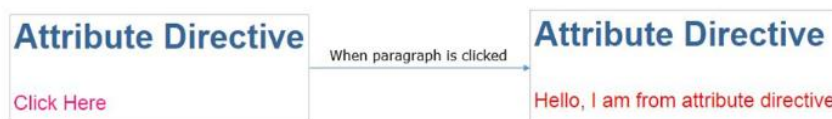
app.component.html

```
1. <h3>Attribute Directive</h3>
2. <p [appMessage]="myMessage">Click Here</p>
```

Line 2: Apply appMessage directive by assigning 'myMessage' property which will be sent to the directive on click of the paragraph

Add the following CSS styles to the **app.component.css** file

```
1. h3 {
2.   color: #369;
3.   font-family: Arial, Helvetica, sans-serif;
4.   font-size: 250%;
5. }
6. p {
7.   color: #ff0080;
8.   font-family: Arial, Helvetica, sans-serif;
9.   font-size: 150%;
10. }
```

Output:**Data Binding**

- Data Binding is a mechanism where data in view and model are in sync. Users should be able to see the same data in a view which the model contains.
- As a developer, you need to bind the model data in a template such that the actual data reflects in the view.

There are two types of data bindings based on the direction in which data flows.

- One-way Data Binding
- Two-way Data Binding

Data Direction	Syntax	Binding Type
One-way (Class -> Template)	<pre>{{ expression }}</pre> <pre>[target] = 'expression'</pre>	<ul style="list-style-type: none"> • Interpolation • Property • Attribute • Class • Style
One-way (Template -> Class)	<pre>(target) = 'statement'</pre>	<ul style="list-style-type: none"> • Event
Two-way	<pre>[(target)] = 'expression'</pre>	<ul style="list-style-type: none"> • Two way

target in the above table refers to a property/event/attribute-name(rarely used).

One-way Data Binding

The following are one-way data binding types:

1. Property Binding
2. Attribute Binding
3. Class Binding
4. Style Binding
5. Event Binding

Understanding Property Binding

Property binding is used when its required to set the property of a class with the property of an element

Syntax:

1. `` or
2. ``

Here the component's imageUrl property is bound to the value to the image element's property src.

Interpolation can be used as an alternative to property binding. Property binding is mostly used when it is required to set a non-string value.

Example:

First, create a folder called 'imgs' under the assets folder and copy any image into that folder.

app.component.ts

1. ...
2. export class AppComponent {
3. imageUrl: string = 'assets/imgs/logo.jpg';
4. }

Line 3: Create a property called imageUrl and initialize it to the image path

app.component.html

1. ``

Line 1: Bind imageUrl property with src property. This is called property binding

Note: this can also be written as:

```
1. <img bind-src='imgUrl' width=200 height=100>
```

Attribute Binding

- Property binding will not work for a few elements/pure attributes like ARIA, SVG, and COLSPAN. In such cases, you need to go for attribute binding.
- Attribute binding can be used to **bind a component property to the attribute directly**
- For example,

```
1. <td colspan = "{{ 2+3 }}">Hello</td>
```

- The above example gives an error as colspan is not a property. Even if you use property binding/interpolation, it will not work as it is a pure attribute. For such cases, use attribute binding.
- Attribute binding syntax starts with prefix attr. followed by a dot sign and the name of an attribute. And then set the attribute value to an expression.

```
1. <td [attr.colspan] = "2+3">Hello</td>
```

Example:

app.component.ts

```
1. ...
2. export class AppComponent {
3.   colspanValue: string = "2";
4. }
```

Line 3: Create a property called value and initialize to 2

app.component.html

```
1. <table border=1>
2. <tr>
3.   <td [attr.colspan]="colspanValue"> First </td>
4.   <td>Second</td>
5. </tr>
6. <tr>
7.   ...
8. </tr>
9. </table>
```

Line 3: attr.colspan will inform Angular that colspan is an attribute so that the given expression is evaluated and assigned to it. This is called attribute binding.

Output:

First	Second	
Third	Fourth	Fifth

Style Binding Style binding is used to set inline styles. Syntax starts with prefix style, followed by a dot and the name of a CSS style property.

Syntax:

```
1. [style.styleproperty]
```

Example:

```
1. <button [style.color] = "isValid ? 'blue' : 'red' ">Hello</button>
```

Here button text color will be set to blue if the expression is true, otherwise red.

Some style bindings will have a unit extension.

Example:

```
1. <button [style.font-size.px] = "isValid ? 3 : 6">Hello</button>
```

- Here text font size will be set to 3 px if the expression isValid is true, otherwise, it will be set to 6px.
- The ngStyle directive is preferred when it is required to set multiple inline styles at the same time.

Event Binding

User actions such as entering text in input boxes, picking items from lists, button clicks may result in a flow of data in the opposite direction: from an element to the component.

Event binding syntax consists of a target event with () on the left of an equal sign and a template statement on the right.

Example:

```
1. <button (click) ="onSubmit(username.value,password.value)">Login</button>
```

OR

```
1. <button on-click = "onSubmit(username.value,password.value)">Login</button>
```

Built-in Pipes:

Pipes provide a beautiful way of transforming the data inside templates, for the purpose of display.

Pipes in Angular

Pipes in Angular take an expression value as an input and transform it into the desired output before displaying it to the user. It provides a clean and structured code as you can reuse the pipes throughout the application, while declaring each pipe just once.

Syntax:

```
1. {{ expression | pipe }}
```

Example:

```
1. {{ "Angular" | uppercase }}
```

This will display ANGULAR

Various built-in pipes are provided by Angular for data transformations like transformation of numerical values, string values, dates, etc.. Angular also has built-in pipes for transformations for internationalization (i18n), where locale information is used for data formatting.

The following are commonly used built-in pipes for data formatting:

1. uppercase

2. lowercase
3. titlecase
4. currency
5. date
6. percent
7. slice
8. decimal

uppercase

This pipe converts the template expression into uppercase.

Syntax:

```
1. {{ expression | uppercase }}
```

Example:

```
1. {{ "Laptop" | uppercase }}
```

Output:

LAPTOP

lowercase

This pipe converts the template expression into lowercase.

Syntax:

```
1. {{ expression | lowercase }}
```

Example:

```
1. {{ "LAPTOP" | lowercase }}
```

Output:

laptop

titlecase

This pipe converts the first character in each word of the given expression into a capital letter.

Syntax:

```
1. {{ expression | titlecase }}
```

Example:

```
1. {{ "product details" | titlecase }}
```

Output:

Product Details

Passing Parameters to Pipes:

A pipe can also have optional parameters to change the output. To pass parameters, after a pipe name add a colon (:) followed by the parameter value.

Syntax:

```
1. pipename : parametervalue
```

A pipe can also have multiple parameters as shown below

```
1. pipename : parametervalue1:parametervalue2
```

Below are the built-in pipes present in Angular, which accept optional parameters using which the pipe's output can be fine-tuned.

currency

This pipe displays a currency symbol before the expression. By default, it displays the currency symbol \$

Syntax:

```
1. {{ expression | currency:currencyCode:symbol:digitInfo:locale }}
```

currencyCode is the code to display such as INR for the rupee, EUR for the euro, etc.

symbol is a Boolean value that represents whether to display currency symbol or code.

- **code**: displays code instead of a symbol such as USD, EUR, etc.
- **symbol** (default): displays symbol such as \$ etc.
- **symbol-narrow**: displays the narrow symbol of currency. Some countries have two symbols for their currency, regular and narrow. For example, the Canadian Dollar CAD has the symbol as CA\$ and symbol-narrow as \$.

digitInfo is a string in the following format

{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}

- minIntegerDigits is the minimum integer digits to display. The default value is 1
- minFractionDigits is the minimum number of digits to display after the fraction. The default value is 0
- maxFractionDigits is the maximum number of digits to display after the fraction. The default value is 3

locale is used to set the format followed by a country/language. To use a locale, the locale needs to be registered in the root module.

For Example, to set locale to French (fr), add the below statements in app.module.ts

```
1. import { registerLocaleData } from '@angular/common';
2. import localeFrench from '@angular/common/locales/fr';
3.
4. registerLocaleData(localeFrench);
```

Examples:

```
1. {{ 25000 | currency }} will display $25,000.00
2. {{ 25000 | currency:'CAD' }} will display CA$25,000.00
3. {{ 25000 | currency:'CAD':'code' }} will display CAD25,000.00
4. {{ 25000 | currency:'CAD':'symbol':'6.2-3' }} will display CA$025,000.00
```

5. `{{ 25000 | currency:'CAD': 'symbol-narrow':'1.3' }}` will display \$25,000.000
6. `{{ 250000 | currency:'CAD': 'symbol':'6.3' }}` will display CA\$250,000.000
7. `{{ 250000 | currency:'CAD': 'symbol':'6.3':'fr' }}` will display 250 000,000 CA\$

date

This pipe can be used to display the date in the required format

Syntax:

1. `{{ expression | date:format:timezone:locale }}`

An **expression** is a date or number in milliseconds

The **format** indicates in which form the date/time should be displayed. Following are the pre-defined options for it.

- 'medium': equivalent to 'MMM d, y, h:mm:ss a' (e.g. Jan 31, 2018, 11:05:04 AM)
- 'short': equivalent to 'M/d/yy, h:mm a' (e.g. 1/31/2018, 11:05 AM)
- 'long': equivalent to 'MMMM d, y, h:mm:ss a z' (e.g. January 31, 2018 at 11:05:04 AM GMT+5)
- 'full': equivalent to 'EEEE, MMMM d, y, h:mm:ss a zzzz' (e.g. Wednesday, January 31, 2018 at 11:05:04 AM GMT+05:30)
- 'fullDate': equivalent to 'EEEE, MMMM d, y' (e.g. Wednesday, January 31, 2018)
- 'longDate': equivalent to 'MMMM d, y' (e.g. January 31, 2018)
- 'mediumDate': equivalent to 'MMM d, y' (e.g. Jan 31, 2018)
- 'shortDate': equivalent to 'M/d/yy' (e.g. 1/31/18)
- 'mediumTime': equivalent to 'h:mm:ss a' (e.g. 11:05:04 AM)
- 'shortTime': equivalent to 'h:mm a' (e.g. 11:05 AM)
- 'longTime': equivalent to 'h:mm a' (e.g. 11:05:04 AM GMT+5)
- 'fullTime': equivalent to 'h:mm:ss a zzzz' (e.g. 11:05:04 AM GMT+05:30)

Timezone to be used for formatting. For example, '+0430' (4 hours, 30 minutes east of the Greenwich meridian) If not specified, the local system timezone of the end-user's browser will be used.

locale is used to set the format followed by a country/language. To use a locale, register the locale in the root module. For Example, to set locale to French (fr), add the below statements in app.module.ts

1. `import { registerLocaleData } from '@angular/common';`
2. `import localeFrench from '@angular/common/locales/fr';`
3. `registerLocaleData(localeFrench);`

Examples:

1. `{{ "6/2/2017" | date }}` will display Jun 2, 2017
2. `{{ "6/2/2017, 11:30:45 AM" | date:'medium' }}` will display Jun 2, 2017, 11:30:45 AM
3. `{{ "6/2/2017, 11:30:45 AM" | date:'mmss' }}` will display 3045
4. `{{ "1/31/2018, 11:05:04 AM" | date:'fullDate':'0':'fr' }}` will display mercredi 31 janvier 2018

5. `{{ 90000000 | date }}` will display Jan 2, 1970 – date pipe will start from Jan 1, 1970 and based on the given number of milliseconds, it displays the date

percent

This pipe can be used to display the number as a percentage

Syntax:

1. `{{ expression | percent:digitInfo:locale }}`

digitInfo is a string in the following format

{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}

- minIntegerDigits is the minimum integer digits to display. The default value is 1
- minFractionDigits is the minimum number of digits to display after the fraction. The default value is 0.
- maxFractionDigits is the maximum number of digits to display after the fraction. The default value is 3.

locale is used to set the format followed by a country/language. To use a locale, register the locale in the root module.

For Example, to set locale to French (fr), add the below statements in app.module.ts

1. `import { registerLocaleData } from '@angular/common';`
2. `import localeFrench from '@angular/common/locales/fr';`
3. `registerLocaleData(localeFrench);`

Examples:

1. `{{ 0.1 | percent }}` will display 10%
2. `{{ 0.1 | percent:'2.2-3' }}` will display 10.00%
3. `{{ 0.1 | percent:'2.2-3': 'fr' }}` will display 10.00 %

slice

This pipe can be used to extract a subset of elements or characters from an array or string respectively.

Syntax:

1. `{{ expression | slice:start:end }}`

The **expression** can be an array or string

start represents the starting position in an array or string to extract items. It can be a

- positive integer which will extract from the given position till the end
- negative integer which will extract the given number of items from the end

end represents the ending position in an array or string for extracting items. It can be

- positive number that returns all items before the end index
- negative number which returns all items before the end index from the end of the array or string

Examples:

1. `{{ ['a','b','c','d'] | slice:2 }}` will display c,d

2. `{{ ['a','b','c','d'] slice:1:3 }}` will display b,c
3. `{{ 'Laptop Charger' slice:3:6 }}` will display top
4. `{{ 'Laptop Charger' slice:-4 }}` will display rger
5. `{{ 'Laptop Charger' slice:-4:-2 }}` will display rg

number

This pipe can be used to format a number.

Syntax:

1. `{{ expression | number:digitInfo }}`

The **expression** should be numeric.

digitInfo is a string in the following format

{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}

- minIntegerDigits is the minimum integer digits to display. The default value is 1.
- minFractionDigits is the minimum number of digits to display after the fraction. The default value is 0.
- maxFractionDigits is the maximum number of digits to display after fraction. The default value is 3.

Example:

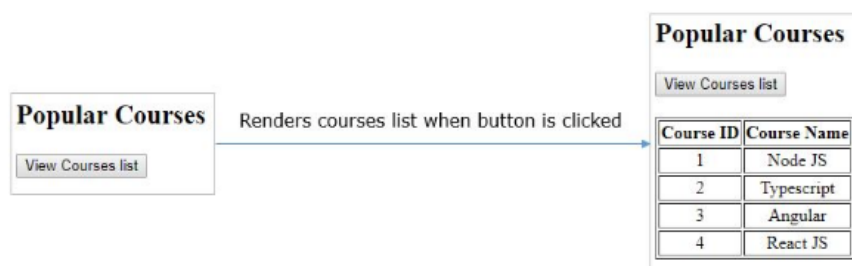
1. `{{ 25000 | number }}` will display 25,000
2. `{{ 25000 | number:'.3-5' }}` will display 25,000.000

Nested Components

- Nested component is a component that is loaded into another component
- The component where another component is loaded onto is called a container component/parent component.
- The root component is loaded in the index.html page using its selector name. Similarly, to load one component into a parent component, use the selector name of the component in the template i.e., the HTML page of the container component

Example for creating multiple components and load one into another::

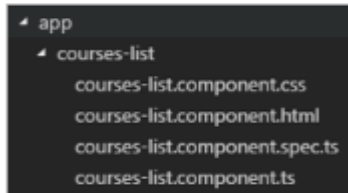
Problem Statement: Create an AppComponent which displays a button to view courses list on click of it and another component called CoursesListComponent which displays courses list. When the user clicks on the button in the AppComponent, it should load the CoursesList component within it to show the courses list



Create a component called the coursesList using the following CLI command

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

This command will create four files called `courses-list.component.ts`, `courses-list.component.html`, `courses-list.component.css`, `courses-list.component.spec.ts` and places them inside a folder called `courses-list` under the `app` folder as shown below



This command will also add the `CoursesListComponent` to the root module.

app.module.ts

```
1. ...
2. import { CoursesListComponent } from './courses-list/courses-list.component';
3. @NgModule({
4.   declarations: [
5.     AppComponent,
6.     CoursesListComponent
7.   ],
8.   ...
9. })
10. export class AppModule { }
```

Line 7: `CoursesListComponent` is added to the `declarations` property to make it available to all other components in the module.

courses-list.component.ts

```
1. ...
2. export class CoursesListComponent {
3.   courses = [
4.     { courseId: 1, courseName: 'Node JS' },
5.     { courseId: 2, courseName: 'Typescript' },
6.     { courseId: 3, courseName: 'Angular' },
7.     { courseId: 4, courseName: 'React JS' }
8.   ];
9. }
```

Line 3-8: `courses` is an array of objects where each object has properties called `courseId` and `courseName`

courses-list.component.html

```
1. <table border="1">
2. ...
3. <tbody>
4. <tr *ngFor="let course of courses">
5. <td>{{ course.courseId }}</td>
6. <td>{{ course.courseName }}</td>
7. </tr>
```

```
8. </tbody>
9. </table>
```

Line 4-7: ngFor iterates over courses array and renders courseId and courseName values

Add the following code in **courses-list.component.css**

```
1. tr{
2.   text-align:center;
3. }
```

Line 1-3: adds center alignment to a table row in an HTML page

app.component.html

```
1. ...
2. <button (click)="show=true">View Courses list</button><br/><br/>
3. <div *ngIf="show">
4.   <app-courses-list></app-courses-list>
5. </div>
```

Line 2: click event is bounded to button which will initialize show property to true when it is clicked

Line 4: It loads the CoursesList component only if the show property value is true.

Note: Here AppComponent is called Parent component or container component as we are loading another component in it.

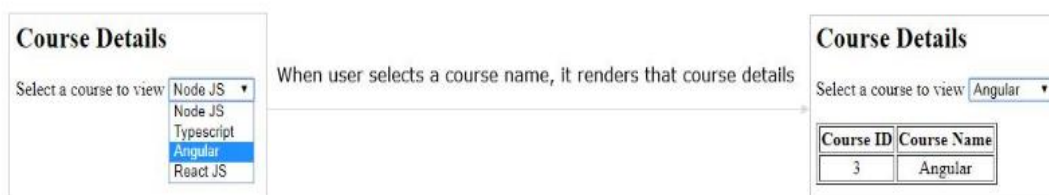
CoursesList component is called child component as it is being loaded in another component.

Passing data from Container Component to Child Component

- Component communication is needed if data needs to be shared between the components
- In order to pass data from container/parent component to child component, @Input decorator can be used.
- A child component can use @Input decorator on any property type like arrays, objects, etc. making it a data-bound input property.
- The parent component can pass value to the data-bound input property when rendering the child within it.

Example:

Problem Statement: Create an AppComponent that displays a dropdown with a list of courses as values in it. Create another component called CoursesList component and load it in AppComponent which should display the course details. When a user selects a course from the dropdown, corresponding course details should be loaded



Open the **courses-list.component.ts** file used in the previous example

Add input setter method for property cName in the component as shown below in Line 18

```

1. ...
2. export class CoursesListComponent {
3. ...
4. course!: any[];
5. @Input() set cName(name: string) {
6.   this.course = [];
7.   for (var i = 0; i < this.courses.length; i++) {
8.     if (this.courses[i].courseName === name) {
9.       this.course.push(this.courses[i]);
10.    }
11.  }
12. }
13. }
```

Line 7: @Input() specifies that cName property will receive value from its container component

Line 9-13: for loop will iterate over courses array and it checks for the courseName validity. If it matches, it adds that object to the course array

courses-list.component.html

```

1. <table border="1" *ngIf="course.length>0">
2. ...
3. <tbody>
4. <tr *ngFor="let c of course">
5. <td>{{ c.courseId }}</td>
6. <td>{{ c.courseName }}</td>
7. </tr>
8. </tbody>
9. </table>
```

Line 4-7: ngFor iterates on course array and displays courseId and courseName properties in a table

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

```

1. <h2>Course Details</h2>
2. Select a course to view
3. <select #course (change)="name = course.value">
4. <option value="Node JS">Node JS</option>
5. <option value="Typescript">Typescript</option>
6. <option value="Angular">Angular</option>
7. <option value="React JS">React JS</option></select>
8. <br /><br />
9. <app-courses-list [cName]="name"></app-courses-list>
```

Line 2-7: It displays a drop-down to select a course to display its details. When the user selects a value, it assigns the selected value to the name property.

Line 9: This will load CoursesListComponent and passes the name property value to the cName property of CoursesListComponent class.

Best Practices - Coding Style Rules



Place @Input decorator on the same line as the property to make it shorter and more readable.

Passing data from Child to Container Component

- If a child component wants to send data to its parent component, then it must create a property with @Output decorator.
- The only method for the child component to pass data to its parent component is through events. The property must be of type EventEmitter.

Example:

Problem Statement: Create an AppComponent that loads another component called CoursesList component. Create another component called CoursesListComponent which should display the courses list in a table along with a register button in each row. When a user clicks on the register button, it should send that courseName value back to AppComponent where it should display the registration successful message along with courseName.



courses-list.component.ts

```

1. ...
2. export class CoursesListComponent {
3.   @Output() onRegister = new EventEmitter<string>();
4.   ...
5.   register(courseName: string) {
6.     this.onRegister.emit(courseName);
7.   }
8. }

```

Line 3: Create a property called onRegister of type EventEmitter and attach @Output decorator which makes the property to send the data from child to parent

Line 7: this line emits the courseName value i.e, send the courseName value back to parent component.

courses-list.component.html

```

1. <table border="1">
2. ...
3. <tbody>
4. <tr *ngFor="let course of courses">
5. <td>{{ course.courseId }}</td>

```

```

6. <td>{{course.courseName}}</td>
7. <td><button (click)="register(course.courseName)">Register</button></td>
8. </tr>
9. </tbody>
10. </table>

```

Line 7: When the user clicks this button, it invokes the register method by passing courseName value.

app.component.html

```

1. <h2> Courses List </h2>
2. <app-courses-list (OnRegister)="courseReg($event)"></app-courses-list>
3. <br/><br/>
4. <div *ngIf="message">{{ message }}</div>

```

Line 3: Binds onRegister event with courseReg method of the parent component. When CoursesListComponent emits the value, onRegister event is triggered and it invokes courseReg method. \$event holds the value emitted by CoursesListComponent

Line 6: This renders the message property value which holds the value emitted

app.component.ts

```

1. ...
2. export class AppComponent {
3.   message!: string;
4.   courseReg(courseName: string) {
5.     this.message = `Your registration for ${courseName} is successful`;
6.   }
7. }

```

Line 6: courseReg method is invoked when onRegister event emits

Line 7: Assigns the string to message property which will be rendered in the template

Best Practices - Coding Style Rules

- ✓ Place @Output decorator on the same line as the property to make it shorter and more readable.
- ✓ Do not use the 'on' prefix for event names as it is the format used for built-in event names.

Component Life Cycle

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. export class AppComponent implements OnInit, DoCheck, AfterContentInit, AfterContentChecked,
5.   AfterViewInit, AfterViewChecked, OnDestroy {
6.   ngOnInit() { }
7.   ngDoCheck() { }
8.   ngAfterContentInit() { }
9.   ngAfterContentChecked() { }
10.  ngAfterViewInit() { }
11.  ngAfterViewChecked() { }
12.  ngOnDestroy() { }
13. }

```

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

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. }
```

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>
```

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:



Forms in Angular

Forms are crucial part of web applications. Forms enable users to provide data input into the application in contexts like performing user registration, user sign-in, updating profile, entering sensitive information like payment information and for performing other data-entry based tasks.

Forms in Angular:

Angular has two different approaches in dealing with forms: *reactive forms* and *template-driven forms*.

Both reactive forms and template-driven forms:

- can capture user-provided data,
- can capture user input events,
- can validate the user input, etc.
- have their own approaches of processing and managing the form data:
 - In template-driven forms, you will create the form completely in the template and need to rely on directives to create and manipulate the underlying form object model. Since the template-driven forms do not scale that well, they are more suitable only when you want to add a simple small form to the application. For example: a signup form.
 - In reactive forms, you can control the form completely from the component class and hence you will get direct, explicit access to the underlying forms object model. Hence, reactive forms are also known as '*model-driven forms*'. As reactive forms are more robust and scalable, they are more suitable for creating all kind of forms in an application, irrespective of the size of form.

State of forms and form controls in Angular:

Angular automatically tracks the changes happening to the form and form controls as and when user provides input and thereby controls the state and the validity of the form/form controls. Angular does this by associating respective keywords automatically for the forms/form controls depending on the context. The below table describes the details:

State detected for the form control	Context	Keyword associated by Angular
valid	Element value is valid	valid. The keyword becomes true if element is valid.
invalid	Element value is invalid	invalid. The keyword becomes true if element is invalid
dirty	Element value is changed	dirty. The keyword becomes true if element is dirty
pristine	Element value is unchanged	pristine. The keyword becomes true if element is pristine
touched	Element gets focus	touched. The keyword becomes true if element is touched
untouched	Element doesn't have a focus in it	untouched. The keyword becomes true if element is untouched

Angular also has the following built-in CSS classes which get auto-applied depending on the state. Code can be written inside these CSS classes which can change the appearance of the control suitably as per context.

CSS Class	Purpose
ng-valid	Applied if control's value is valid
ng-invalid	Applied if control's value is invalid
ng-dirty	Applied if control's value is changed
ng-pristine	Applied if control's value is not changed
ng-touched	Applied if control is touched/gets focus
ng-untouched	Applied if control is not touched/doesn't get focus

Advantages of Reactive Forms/Model Driven Forms

- Unit testing(using the Jasmine framework) on the validation logic can be performed, as it is written inside the component class.
- Form changes or events can be heard easily using reactive forms. Each FormGroup or FormControl has few events like valueChanges, statusChanges, etc., which can be subscribed to.
- Reactive forms are used in creating medium to large scale applications

Due to the advantages of Reactive forms, in most Angular applications, Reactive Forms Approach is chosen when creating forms.

Creating Reactive Forms in Angular

Step1:

To create a reactive form in Angular, **FormBuilder** class must be used. To make the FormBuilder class available, **ReactiveFormsModule** has to be imported in the root module.

Register the ReactiveFormsModule during bootstrapping, in the **app.module.ts**

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

Line 3: Import ReactiveFormsModule from @angular/forms module

Line 15: Add ReactiveFormsModule in the imports declaration to create reactive forms

Step 2:

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

```
1. ng generate component RegistrationForm
```

Step 3:

Add the below to **app.component.html**

```
1. <app-registration-form></app-registration-form>
```

Line 1: Loads RegistrationFormComponent in the root component

Step 4:

Bootstrap CSS framework (v3) is commonly used for adding basic structural layout and style for web applications. For structuring the RegistrationForm in this example, CSS classes from the Bootstrap CSS framework can be used.

To add Bootstrap CSS library to the application, install **bootstrap** as shown below:

```
1. npm install bootstrap@3.3.7 --save
```

Then, include bootstrap.min.css file in angular.json file as shown below:

```
1. ...
2. "styles": [
3.   "src/styles.css",
4.   "node_modules/bootstrap/dist/css/bootstrap.min.css"
5. ],
6. "scripts": [
7.   "node_modules/jquery/dist/jquery.min.js",
8.   "node_modules/bootstrap/dist/js/bootstrap.min.js"
```

9.]
10. ...

Note: When angular.json is modified, restart the server to see the changes reflected.

Step 5:

Include the below code to **registration-form.component.css**

```
1. .ng-valid[required] {  
2.   border-left: 5px solid #42A948; /* green */  
3. }  
4. .ng-invalid:not(form) {  
5.   border-left: 5px solid #a94442; /* red */  
6. }
```

Line 1-3: ng-valid CSS class changes left border of the textbox to green if form control has valid input

Line 5-7: ng-invalid CSS class changes left border of the textbox to red if form control has invalid data

Building Reactive Forms (Angular v13)

Add the following code in the **registration-form.component.ts** file

```
1. import { Component, OnInit } from '@angular/core';  
2. import { FormBuilder, FormGroup, Validators } from '@angular/forms';  
3. @Component({  
4.   selector: 'app-registration-form',  
5.   templateUrl: './registration-form.component.html',  
6.   styleUrls: ['./registration-form.component.css']  
7. })  
8. export class RegistrationFormComponent implements OnInit {  
9.   registerForm!: FormGroup;  
10.  submitted!:boolean;  
11.  constructor(private formBuilder: FormBuilder) { }  
12.  ngOnInit() {  
13.    this.registerForm = this.formBuilder.group({  
14.      firstName: ['', Validators.required],  
15.      lastName: ['', Validators.required],  
16.      address: this.formBuilder.group({  
17.        street: [],  
18.        zip: [],  
19.        city: []  
20.      })  
21.    });  
22.  }  
23. }
```

Line 2: Import FormBuilder class to create a reactive form. Also, import FormGroup class to create a group of form controls and Validators for validation

Line 11: Create a property registerForm of type FormGroup

Line 14: Inject a FormBuilder instance using constructor

Line 17: `formBuilder.group()` method creates a `FormGroup`. It takes an object whose keys are `FormControl` names and values are their definitions

Line 18-24: Create form controls such as `firstName`, `lastName`, and `address` as a subgroup with fields `street`, `zip`, and `city`. These fields are form controls.

For each form control:

- you can mention the default value as the first argument and
- the list of validators as the second argument:
- Validations can be added to the form controls using the built-in validators supplied by the `Validators` class.
- For example: Configure built-in required validator for each control using `[' ', Validators.required]` syntax.
- If multiple validators are to be applied, then use the syntax `[' ', [Validators.required, Validators.maxLength(10)]]`.

registration-form.component.html

```
1. <div class="container">
2. <h1>Registration Form</h1>
3. <form [formGroup]="registerForm">
4. <div class="form-group">
5. <label>First Name</label>
6. <input type="text" class="form-control" formControlName="firstName">
7. <div *ngIf="registerForm.controls['firstName'].errors" class="alert alert-danger">
8. Firstname field is invalid.
9. <p *ngIf="registerForm.controls['firstName'].errors?.['required']">
10. This field is required!
11. </p>
12. </div>
13. </div>
14. <div class="form-group">
15. <label>Last Name</label>
16. <input type="text" class="form-control" formControlName="lastName">
17. <div *ngIf="registerForm.controls['lastName'].errors" class="alert alert-danger">
18. Lastname field is invalid.
19. <p *ngIf="registerForm.controls['lastName'].errors?.['required']">
20. This field is required!
21. </p>
22. </div>
23. </div>
24. <div class="form-group">
25. <fieldset formGroupName="address">
26. <legend>Address:</legend>
27. <label>Street</label>
28. <input type="text" class="form-control" formControlName="street">
```

```

29. <label>Zip</label>
30. <input type="text" class="form-control" formControlName="zip">
31. <label>City</label>
32. <input type="text" class="form-control" formControlName="city">
33. </fieldset>
34. </div>
35. <button type="submit" class="btn btn-primary" (click)="submitted=true">Submit</button>
36. </form>
37. <br/>
38. <div [hidden]="!submitted">
39. <h3> Employee Details </h3>
40. <p>First Name: {{ registerForm.get('firstName')?.value }} </p>
41. <p> Last Name: {{ registerForm.get('lastName')?.value }} </p>
42. <p> Street: {{ registerForm.get('address.street')?.value }} </p>
43. <p> Zip: {{ registerForm.get('address.zip')?.value }} </p>
44. <p> City: {{ registerForm.get('address.city')?.value }} </p>
45. </div>
46. </div>

```

Line 3: formGroup is a directive that binds HTML form with the FormGroup property created inside a component class. A FormGroup has been created in component with the name registerForm. Here form tag is bound with FormGroup name called registerForm

Line 6, 16: Two text boxes for first name and last name are bound with the form controls created in the component using formControlName directive

Line 7-13: A validation error message is displayed when the firstName is modified and has validation errors.

Line 17-22: A validation error message is displayed when the lastName is modified and has validation errors.

Line 35: When the submit button is clicked, it initializes the submitted property value to true

Line 38: div tag will be hidden if the form is not submitted

Line 39-44: Using the get() method of FormGroup, each FormControl value is fetched and rendered.

Save all the files and observe the output:

The screenshot shows a web browser displaying a registration form. The form is titled 'Registration Form'. It contains several input fields: 'First Name', 'Last Name', 'Address' (with sub-fields for 'Street', 'Zip', and 'City'), and a 'Submit' button. The 'First Name' and 'Last Name' fields are highlighted in red, indicating they are required and currently empty. Below each of these fields, a red error message is displayed: 'Firstname field is invalid. This field is required!' and 'Lastname field is invalid. This field is required!' respectively. The 'Address' section is currently empty.

Registration Form

First Name
James

Last Name
Gosling

Address:

Street
ABC Street

Zip
457899

City
New York

Submit

Employee Details

First Name: James
Last Name: Gosling
Street: ABC Street
Zip: 457899
City: New York

Custom validation in Reactive Forms

Need for custom validation:

While creating forms, there can be situations for which built-in validators are not available. Few such examples include validating a phone number, validating if the password and confirm password fields matches or not, etc.. In such situations, custom validators can be created to implement the required validation functionality.

Custom validation in Reactive Forms of Angular:

Custom validation can be applied to form controls of a Reactive Form in Angular.

- Custom validators are implemented as separate functions inside the component.ts file.
- these functions can be added to the list of other validators configured for a form control.

Implementing custom validation in Reactive Forms of Angular:

Add one more field called email inside the example used for ReactiveForms previously. The below are the validations to be applied to the 'email' field:

- required,
- checks for standard email pattern, for example: abc@something.com, abc@something.co.in, etc.

For implementing the for custom validation, add a separate function which checks for standard email pattern inside **registration-form.component.ts** as shown below:

```
1. import { Component, OnInit } from '@angular/core';
2. import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';
3. @Component({
4.   selector: 'app-registration-form',
5.   templateUrl: './registration-form.component.html',
6.   styleUrls: ['./registration-form.component.css']
7. })
8. export class RegistrationFormComponent implements OnInit {
9.   registerForm!: FormGroup;
```



```

10. submitted!:boolean;
11. constructor(private formBuilder: FormBuilder) { }
12. ngOnInit() {
13.   this.registerForm = this.formBuilder.group({
14.     firstName: ['', Validators.required],
15.     lastName: ['', Validators.required],
16.     address: this.formBuilder.group({
17.       street: [],
18.       zip: [],
19.       city: []
20.     }),
21.     email: ['', validateEmail]
22.   });
23. }
24. }
25. function validateEmail(c: FormControl): any {
26.   let EMAIL_REGEXP = /^[a-zA-Z0-9_\-\.]+@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$/;
27.   return EMAIL_REGEXP.test(c.value) ? null : {
28.     emailInvalid: {
29.       message: "Invalid Format!"
30.     }
31.   };
32. }

```

Line 30-37: In this function, a regular expression pattern is taken for email and the input value of the form control is tested against the mentioned pattern. If the pattern matches, it means the entered input is valid and hence, the validation function returns null. Otherwise, the function returns an object with name 'emailInvalid' with one property called 'message' set to appropriate string message.

Line 21: Binds the required validator and the custom validator named validateEmail to the email field.

Tracking the Custom Validators in the Reactive Form's template (Angular v13)

For tracking the custom validators in the Reactive Form's template (Angular v13), add HTML controls for the email field in the registration-form.component.html file as shown below:

```

1. <div class="container">
2.   <h1>Registration Form</h1>
3.   <form [formGroup]="registerForm">
4.     <div class="form-group">
5.       <label>First Name</label>
6.       <input type="text" class="form-control" formControlName="firstName">
7.       <div *ngIf="registerForm.controls['firstName'].errors" class="alert alert-danger">
8.         Firstname field is invalid.
9.         <p *ngIf="registerForm.controls['firstName'].errors?.['required']">
10.          This field is required!
11.        </p>
12.      </div>

```

```

13. </div>
14. <div class="form-group">
15. <label>Last Name</label>
16. <input type="text" class="form-control" formControlName="lastName">
17. <div *ngIf="registerForm.controls['lastName'].errors" class="alert alert-danger">
18. Lastname field is invalid.
19. <p *ngIf="registerForm.controls['lastName'].errors?.['required']">
20. This field is required!
21. </p>
22. </div>
23. </div>
24. <div class="form-group">
25. <fieldset formGroupName="address">
26. <legend>Address:</legend>
27. <label>Street</label>
28. <input type="text" class="form-control" formControlName="street">
29. <label>Zip</label>
30. <input type="text" class="form-control" formControlName="zip">
31. <label>City</label>
32. <input type="text" class="form-control" formControlName="city">
33. </fieldset>
34. </div>
35. <div class="form-group">
36. <label>Email</label>
37. <input type="text" class="form-control" formControlName="email" />
38. <div *ngIf="registerForm.controls['email'].errors" class="alert alert-danger">
39. Email field is invalid.
40. <p *ngIf="registerForm.controls['email'].errors?.['required']">
41. This field is required!
42. </p>
43. <p *ngIf="registerForm.controls['email'].errors?.['emailInvalid']">
44. {{ registerForm.controls['email'].errors?.['emailInvalid'].message }}
45. </p>
46. </div>
47. </div>
48. <button type="submit" class="btn btn-primary" (click)="submitted=true">Submit</button>
49. </form>
50. <br/>
51. <div [hidden]="!submitted">
52. <h3>Employee Details </h3>
53. <p>First Name: {{ registerForm.get('firstName')?.value }} </p>
54. <p>Last Name: {{ registerForm.get('lastName')?.value }} </p>
55. <p>Street: {{ registerForm.get('address.street')?.value }} </p>
56. <p>Zip: {{ registerForm.get('address.zip')?.value }} </p>
57. <p>City: {{ registerForm.get('address.city')?.value }} </p>
58. <p>Email: {{ registerForm.get('email')?.value }} </p>

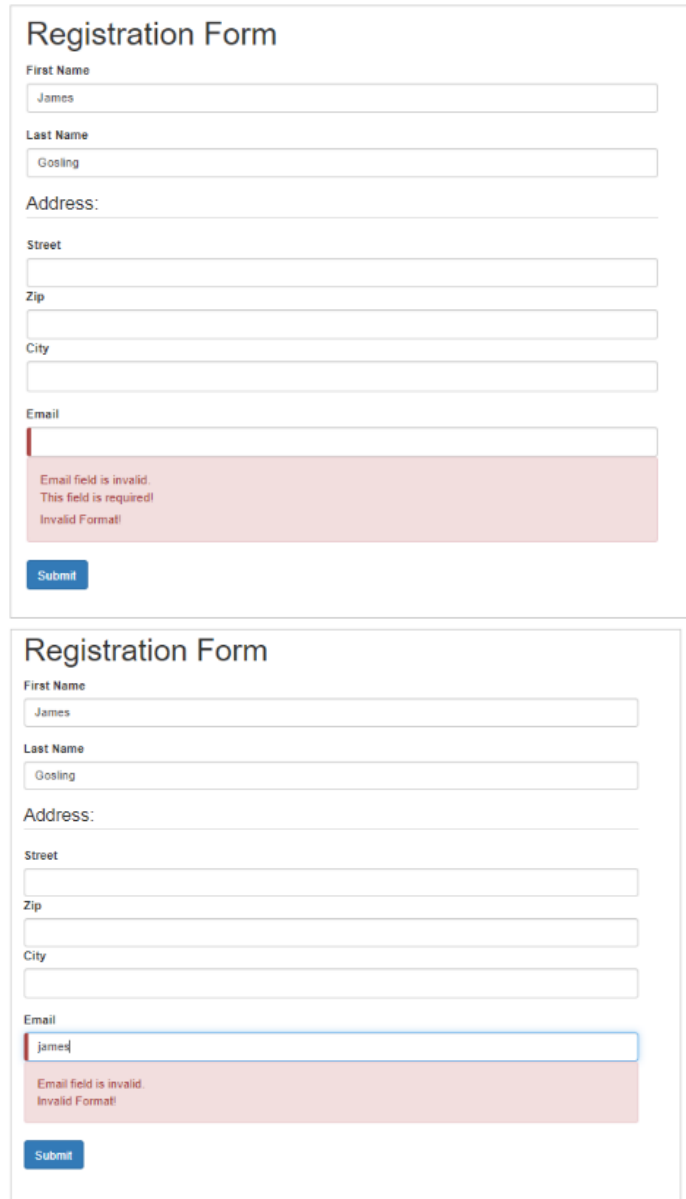
```

```
59. </div>
```

```
60. </div>
```

Line 44: Displays error message if email validation fails. errors object holds the error messages of all form controls

Output:



The image displays two screenshots of a web form titled "Registration Form". The form contains several input fields: "First Name" (with "James" entered), "Last Name" (with "Gosling" entered), "Address:" (with sub-fields for "Street", "Zip", and "City"), and "Email". In the top screenshot, the "Email" field is highlighted with a red border, and a red error message box is visible below it, stating "Email field is invalid. This field is required! Invalid Format!". A blue "Submit" button is at the bottom. The bottom screenshot shows the same form, but the "Email" field now has a blue border, and the error message box is no longer present, indicating a successful validation state.

Dependency Injection

Dependency Injection (DI) is a mechanism where the required resources will be injected into the code automatically.

Angular comes with an in-built dependency injection subsystem.

Why Dependency Injection?

It is because DI:

- allows developers to reuse the code across applications.
- makes the code loosely coupled.
- makes application development and testing much easier.

- allows the developer to ask for the dependencies from Angular. There is no need for the developer to explicitly create/instantiate them.

Services Basics:

- A service in Angular is a class that contains some functionality that can be reused across the application. A service is a singleton object. Angular services are a mechanism of abstracting shared code and functionality throughout the application.
- Angular Services come as objects which are wired together using dependency injection.
- Angular provides a few inbuilt services also can create custom services.

Why Services?

Services can be used to:

- share the code across components of an application.
- make HTTP requests.

Creating a Service

To create a service class, use the following command:

```
1. ng generate service book
```

The above command will create a service class as shown below:

```
1. @Injectable({
2.   providedIn:'root'
3. })
4. export class BookService
5. {
6. }
```

@Injectable() decorator makes the class injectable into application components.

Providing a Service

Following are the ways to provide services in an Angular application:

1. The first way to register service is to specify providedIn property using @Injectable decorator. This property is added by default when you generate a service using Angular CLI.

```
1. import { Injectable } from '@angular/core';
2.
3. @Injectable({
4.   providedIn: 'root'
5. })
6. export class BookService {}
```

Line 4: providedIn property registers BookService at the root level (app module).

When the BookService is provided at the root level, Angular creates a singleton instance of the service class and injects the same instance into any class that uses this service class. In addition, Angular also optimizes the application if registered through providedIn property by removing the service class if none of the components use it.

2. Services can also be provided across the application by registering it using the providers property in the @NgModule decorator of any module.

```
1. @NgModule({
2.   imports: [BrowserModule],
3.   declarations: [AppComponent, BookComponent],
4.   providers: [BookService],
5.   bootstrap: [AppComponent]
6. })
```

Line 4: When the service class is added in the providers property of the root module, all the directives and components will have access to the same instance of the service.

3. There is also a way to limit the scope of the service class by registering it in the providers' property inside the @Component decorator. Providers in component decorator and module decorator are independent. Providing a service class inside a component creates a separate instance for that component and its nested components.

```
1. import { BookService } from './book/book.service';
2. @Component({
3.   selector: 'app-root',
4.   styleUrls: ['./app.component.css'],
5.   templateUrl: './app.component.html',
6.   providers:[BookService]
7. })
```

Injecting a Service

The only way to inject a service into a component/directive or any other class is through a constructor. Add a constructor in a component class with service class as an argument as shown below:

```
1. constructor(private bookService: BookService){ }
```

BookService will then be injected into the component through constructor injection by the framework.

Best Practices - Coding Style Rules

- ✓ Use services for sharing data and functionality.
- ✓ Use a service with a single responsibility to make the testing easier.

Creating and Using Services

Problem Statement: Create a Book Component which fetches book details like id, name and displays them on the page in a list format. Store the book details in an array and fetch the data using a custom service.

Create **BookComponent** by using the following CLI command.

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

Create a file with the name **book.ts** under the book folder and add the following code.

```
1. export class Book {
2.   id!: number;
3.   name!: string;
4. }
```

Line 1-4: Create a Book class with two properties id and name to store book id and book name

Create a file with the name **books-data.ts** under the book folder and add the following code.

```
1. import { Book } from './book';
```

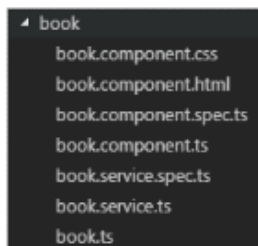
```
2. export var BOOKS: Book[] = [  
3.   { "id": 1, "name": "HTML 5" },  
4.   { "id": 2, "name": "CSS 3" },  
5.   { "id": 3, "name": "Java Script" },  
6.   { "id": 4, "name": "Ajax Programming" },  
7.   { "id": 5, "name": "jQuery" },  
8.   { "id": 6, "name": "Mastering Node.js" },  
9.   { "id": 7, "name": "Angular JS 1.x" },  
10.  { "id": 8, "name": "ng-book 2" },  
11.  { "id": 9, "name": "Backbone JS" },  
12.  { "id": 10, "name": "Yeoman" }  
13. ];
```

Line 3-14: Books is an array of type Book class which holds books objects where each object has id and name properties.

Create a service called **BookService** under the book folder using the following CLI command

```
1. D:\MyApp\src\app\book>ng generate service book
```

This will create two files called book.service.ts and book.service.spec.ts as shown below



Add the following code in **book.service.ts**

```
1. import { Injectable } from '@angular/core';  
2. import { Book } from './book';  
3. import { BOOKS } from './books-data';  
4. @Injectable({  
5.   providedIn:'root'  
6. })  
7. export class BookService {  
8.   getBooks() {  
9.     return BOOKS;  
10.  }  
11. }
```

Line 5-7: @Injectable() decorator makes the class as a service which can be injected into components of an application

Line 10-12: getBooks() method returns Books data.

Add the following code in the **book.component.ts** file

```
1. ...  
2. import { BookService } from './book.service';  
3. import { Book } from './book';
```

```
4.
5. ...
6. export class BookComponent implements OnInit {
7.   books!: Book[];
8.   constructor(private bookService: BookService) { }
9.   getBooks() {
10.    this.books = this.bookService.getBooks();
11.   }
12.   ngOnInit() {
13.    this.getBooks();
14.   }
15. }
```

Line 2,3: Imports BookService, Book class into a component

Line 10: Inject BookService class using a constructor

Line 12: Invokes getBooks() method from BookService class which in turn returns the books data.

book.component.html

```
1. <h2>My Books</h2>
2. <ul class="books">
3.   <li *ngFor="let book of books">
4.     <span class="badge">{{ book.id }}</span> {{ book.name }}
5.   </li>
6. </ul>
```

Line 3 - 5: ngFor iterates on books array and displays book id and name on the page

Add the following code in **book.component.css** which has styles for books

```
1. .books {
2.   margin: 0 0 2em 0;
3.   list-style-type: none;
4.   padding: 0;
5.   width: 15em;
6. }
7. .books li {
8.   cursor: pointer;
9.   position: relative;
10.  left: 0;
11.  background-color: #EEE;
12.  margin: .5em;
13.  padding: .3em 0;
14.  height: 1.6em;
15.  border-radius: 4px;
16. }
17. .books li:hover {
18.  color: #607D8B;
19.  background-color: #DDD;
20.  left: .1em;
```

```
21. }
22. .books .badge {
23. display: inline-block;
24. font-size: small;
25. color: white;
26. padding: 0.8em 0.7em 0 0.7em;
27. background-color: #607D8B;
28. line-height: 1em;
29. position: relative;
30. left: -1px;
31. top: -4px;
32. height: 1.8em;
33. margin-right: .8em;
34. border-radius: 4px 0 0 4px;
35. }
```

Update `app.component.html` to contain below code:

```
<app-book></app-book>
```

Output:



Services - Best Practices

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 Observables

- 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. @Component({
4.   selector: 'app-root',
5.   styleUrls: ['./app.component.css'],
6.   templateUrl: './app.component.html'
7. })
8. export class AppComponent {
9.   data!: Observable<number>;
10.  myArray: number[] = [];
11.  errors!: boolean;
12.  finished!: boolean;
13.  fetchData(): void {
14.    this.data = new Observable(observer => {
15.      setTimeout(() => { observer.next(11); }, 1000),
16.      setTimeout(() => { observer.next(22); }, 2000),
17.      setTimeout(() => { observer.complete(); }, 3000);
18.    });
19.    this.data.subscribe((value) => this.myArray.push(value),
20.      error => this.errors = true,
21.      () => this.finished = true);
22.  }
23. }
```

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

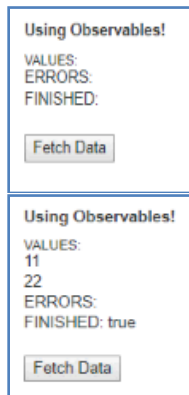
Line 4: ngFor loop is iterated on myArray which will display the values on the page

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:



Server Communication using HttpClient

- 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

```
1. 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

```
1. ...
2. import { HttpClientModule } from '@angular/common/http';
3. ...
4. @NgModule({
5.   imports: [BrowserModule, HttpClientModule],
6.   ...
7. })
8. export class AppModule { }
```

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

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

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.   books!: Book[];
4.   errorMessage!: string;
5.   constructor(private bookService: BookService) { }
6.   getBooks() {
7.     this.bookService.getBooks().subscribe({
8.       next: books => this.books = books,
9.       error:error => this.errorMessage = <any>error
10.    })
11.  }
12.  ngOnInit() {
13.    this.getBooks();
14.  }
15. }

```

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. <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. import { Book } from './book';
6. @Injectable({
7.   providedIn: 'root'
8. })
9. export class BookService {
10.   constructor(private http: HttpClient) { }
11. ...
12. addBook(book: Book): Observable<any> {
13.   const options = new HttpHeaders({ 'Content-Type': 'application/json' });
14.   return this.http.post('http://localhost:3020/addBook', book, { headers: options }).pipe(
15.     catchError(this.handleError));
16. }
17. ...
18. }
```

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

```

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

```

20. }

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

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 **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. import { Book } from './book';
6. @Injectable({
7.   providedIn: 'root'
8. })
9. export class BookService {
10.   constructor(private http: HttpClient) { }
11.   booksUrl = 'http://localhost:3020/bookList';
12. ...
13. deleteBook(bookId: number): Observable<any> {
14.   const url = `${this.booksUrl}/${bookId}`;
15.   return this.http.delete(url).pipe(
16.     catchError(this.handleError));
17. }
18. ...
19. }
```

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

```

24. this.bookService.deleteBook(id)
25. .subscribe({next:(book: any) => this.books = book});
26. }
27. ngOnInit(): void {
28. this.getBooks();
29. }
30. }

```

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

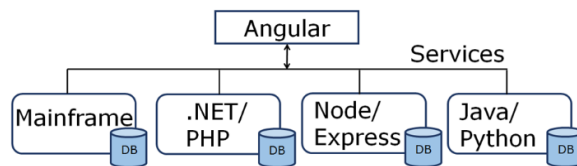
Line 25-29: Invokes the service class method deleteBook() which makes an HTTP call to server URL and the Observable containing the response is returned. You can subscribe to the observable and use the 'next' callback to handle the successfully returned value, as needed.

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.

Communicating with different backend services using Angular HttpClient

Angular can also be used to connect to different services written in different technologies/languages. For example, we can make a call from Angular to Node/Express or Java or Mainframe or .Net services, etc.



Routing Basics:

Configuring Routing in an Angular application

Routing means navigation between multiple views on a single page.

Routing allows to express some aspects of the application's state in the URL. The full application can be built without changing the URL.

Why Routing?

Routing allows to:

- Navigate between the views
- Create modular applications

Configuring Router

Angular uses Component Router to implement routing

A <base> tag must be added to the head tag in the HTML page to tell the router where to start with.

```
1. <base href="/">
```

- Angular component router belongs to @angular/router module. To make use of routing, Routes, RouterModule classes must be imported.

- Configuration should be done for the routes and the router will look for a corresponding route when a browser URL is changed.
- Routes is an array that contains all the route configurations. Then, this array should be passed to the RouterModule.forRoot() function in the application bootstrapping function

Example:

Consider the example used in the services concept.

Add the following in the **app-routing.module.ts** which is created under the app folder

```
1. import { NgModule } from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3. import { BookComponent } from './book/book.component';
4. import { DashboardComponent } from './dashboard/dashboard.component';
5. import { BookDetailComponent } from './book-detail/book-detail.component';
6. import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
7. const appRoutes: Routes = [
8.   { path: 'dashboard', component: DashboardComponent },
9.   { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
10.  { path: 'books', component: BookComponent },
11.  { path: 'detail/:id', component: BookDetailComponent },
12.  { path: '**', component: PageNotFoundComponent },
13. ];
14. @NgModule({
15.  imports: [
16.    RouterModule.forRoot(appRoutes)
17.  ],
18.  exports: [
19.    RouterModule
20.  ]
21. })
22. export class AppRoutingModule { }
```

Line 2: Imports Routes and RouterModule classes

Line 9-13: Configure the routes where each route should contain the path to navigate and the component class has to be invoked for a specific path.

Line 9: A route configuration must be provided for the default path i.e., path: "" and redirect it to the specific route using redirectTo option. pathMatch is required if redirectTo option is used which specifies how the given path should match. Here pathMatch: 'full' tells Router to match the given path completely. pathMatch has another value called 'prefix' where it checks if the path begins with the given prefix.

Line 12: The path 'detail/:id' has the route parameter id which will receive different values for the paramter 'id' based on the book selected, as part of the route itself.

Line 13: A route configuration must be provided for the wildcard route '*'. When users attempt to navigate to a route which may not be existing in your application, the application should be in a position to handle this gracefully. This can be done by configuring a wildcard route. The Angular router will automatically select this route when the requested URL doesn't match any of the other router paths.

Line 18: Pass the appRoutes array to forRoot method of RouterModule class to configure with the Router and add it to the imports property.

Now configure the Router module with NgModule that imports in **app.module.ts** to make it available to the entire application.

```
1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { HttpClientModule } from '@angular/common/http';
4. import { FormsModule } from '@angular/forms';
5. import { AppComponent } from './app.component';
6. import { BookComponent } from './book/book.component';
7. import { DashboardComponent } from './dashboard/dashboard.component';
8. import { BookDetailComponent } from './book-detail/book-detail.component';
9. import { AppRoutingModuleModule } from './app-routing.module';
10. import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
11. @NgModule({
12.   imports: [BrowserModule, HttpClientModule, FormsModule, AppRoutingModuleModule],
13.   declarations: [AppComponent, BookComponent, DashboardComponent, BookDetailComponent,
14.     PageNotFoundComponent],
15.   providers: [],
16.   bootstrap: [AppComponent]
17. })
18. export class AppModule { }
```

Line 9: Imports AppRoutingModuleModule from routing module file

Line 13: Add AppRoutingModuleModule class to the imports property

Router Links:

Navigate using Angular Router

After configuring the routes, the next step is to decide how to navigate. Navigation will happen based on user actions like clicking a hyperlink, clicking on a button, etc. Hence, there is hyperlink based navigation and programmatical navigation.

Hyperlink based navigation

RouterLink directive can be used with the anchor tag for using hyperlink based navigation in Angular. Have a look at code shown below:

app.component.ts

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

app.component.html

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

```
6. <router-outlet></router-outlet>
```

Line 3-4: Create hyperlinks and a routerLink directive and specify the paths to navigate. Here, if a user clicks on the Dashboard, it will navigate to /dashboard. routerLinkActive applies the given CSS class to the link when it is clicked to make it look like an active link(active is a CSS class defined in app.component.css which changes the link color to blue in this case).

Line 6: <router-outlet> is the place where the output of the component associated with the given path will be displayed. For example, if the user click on Books, it will navigate to /books which will execute BooksComponent class as mentioned in the configuration details and the output will be displayed in the router-outlet class.

Programmatical navigation

To navigate programmatically, use the navigate() method of the Router class. Inject the router class into the component and invoke the navigate method as shown below

```
1. this.router.navigate([url, parameters])
```

- URL is the route path to which we want to navigate
- Parameters are the route values passed along with the URL

Why Route Parameters?

Consider an e-commerce application having a *ProductList component* displaying list of products available. To view more details of a particular product, users usually click on the particular product, which opens a separate screen created as *ProductDetail component*. You want the ProductDetail component to display the specific information pertaining to the product that was clicked on the ProductList screen. To facilitate this, you would want to share the unique id of the product to the ProductDetail component. This is why route parameters were introduced.

What are Route Parameters?

Parameters passed along with URLs are called route parameters. Route parameters can be used to share the data from one component to next component (one screen to next).

Passing Route Parameters

Generate a component named dashboard using the following CLI command. This will display list of all the books available.

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

Add the following code in the **dashboard.component.ts** file

```
1. import { Component, OnInit } from '@angular/core';
2. import { Router } from '@angular/router';
3. import { Book } from '../book/book';
4. import { BookService } from '../book/book.service';
5. @Component({
6.   selector: 'app-dashboard',
7.   templateUrl: './dashboard.component.html',
8.   styleUrls: ['./dashboard.component.css']
9. })
10. export class DashboardComponent implements OnInit {
11.   books: Book[] = [];
12.   constructor(
13.     private router: Router,
14.     private bookService: BookService) { }
15.   ngOnInit(): void {
```

```
16. this.bookService.getBooks()
17. .subscribe({next:books => this.books = books.slice(1, 5)});
18. }
19. gotoDetail(book: Book): void {
20. this.router.navigate(['/detail', book.id]);
21. }
22. }
```

Line 2: Import Router class from @angular/router module

Line 16: Inject into the component class through a constructor

Line 20-21: Retrieve all the books by subscribing to the getBooks() method of the BookService and filter out the first 4 books to be displayed by the DashboardComponent

Line 24: The gotoDetail() method receives a particular book whose elaborate details need to be viewed in a separate screen. this.router.navigate() method is used to programmatically navigate to the specific URL '/detail' which will load a new component where book details can be viewed. The book id of the selected book is passed as route parameter, as shown here. So effectively, the path generated for the router will be detail/<book_id>.

Accessing Route Parameters

To access route parameters, use ActivatedRoute class

Generate BookDetail component using the following CLI command

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

Add the following code in the **book-detail.component.ts** file

```
1. import { Component, OnInit } from '@angular/core';
2. import { ActivatedRoute } from '@angular/router';
3. import { Book } from '../book/book';
4. import { BookService } from '../book/book.service';
5. @Component({
6. selector: 'app-book-detail',
7. templateUrl: './book-detail.component.html',
8. styleUrls: ['./book-detail.component.css'],
9. })
10. export class BookDetailComponent implements OnInit {
11. book!: Book;
12. error!: any;
13. constructor(
14. private bookService: BookService,
15. private route: ActivatedRoute
16. ) { }
17. ngOnInit() {
18. this.route.paramsMap.subscribe(params => {
19. this.bookService.getBook(params.get('id')).subscribe((book) => {
20. this.book = book ?? this.book;
21. });
22. });
23. }
```

```
24. goBack() {  
25.   window.history.back();  
26. }  
27. }
```

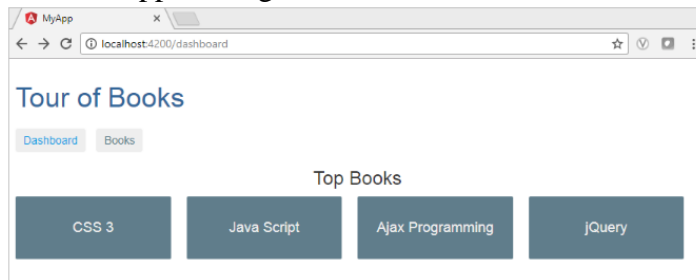
Line 2: Imports ActivatedRoute class to access route parameters

Line 15: Injects ActivatedRoute class into the component class through a constructor

Line 18-22: ActivatedRoute class has a paramsMap observable method that holds the route parameters. Subscribe to this observable to get the parameters using get() method. After getting the parameters, process the parameters as per the requirement.

Output:

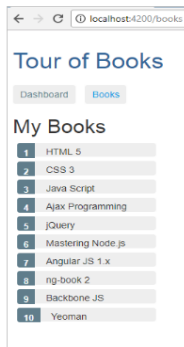
On load of the application, the Dashboard component is shown according to the route configuration mentioned in the app-routing.module.ts.



When a specific book is clicked, it renders the BookDetailComponent which displays details of that particular book, as is shown below:



When the Books link is clicked, it navigates to BooksComponent which displays the list of all the books, as is shown below:



Introduction to Route Guards

In the Angular application, users can navigate to any URL directly. That's not the right thing to do always.

Consider the following scenarios

- Users must login first to access a component
- The user is not authorized to access a component
- User should fetch data before displaying a component
- Pending changes should be saved before leaving a component

These scenarios must be handled through route guards.

A guard's return value controls the behavior of the router

- If it returns true, the navigation process continues
- If it returns false, the navigation process stops

Angular has canActivate interface which can be used to check if a user is logged in to access a component.

canActivate() method must be overridden in the guard class as shown below:

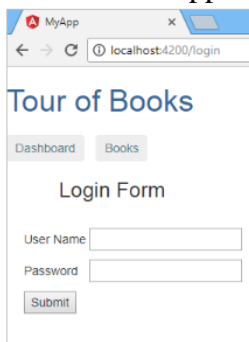
Using canActivate, access can be permitted to only authenticated users.

1. Class GuardService implements CanActivate{
2. canActivate(): boolean {
3. }
4. }

Adding Route Guards

Problem Statement: Consider the same example used for routing. Route guard must be used to BooksComponent. Only after logging in, the user should be able to access BooksComponent. If a user tries to give the URL of Bookscomponent in another tab or window, or if the user tries to reload the BooksComponent page, it should be redirected to LoginComponent.

When a user runs the application, by default it should load the login component as shown below:



After entering the correct credentials, it can load BooksComponent.

Create a **LoginComponent** using Angular CLI

1. ng g c Login

Add the following code to the **login.component.html** file

1. <h3 style="position: relative; left: 60px">Login Form</h3>
2. <div *ngIf="invalidCredentialMsg" style="color: red">
3. {{ invalidCredentialMsg }}
4. </div>
5.

6. <div style="position: relative; left: 20px">
7. <form [formGroup]="loginForm" (ngSubmit)="onFormSubmit()">
8. <p>User Name <input formControlName="username" /></p>
9. <p>
10. Password


```
11. <input
12. type="password"
13. formControlName="password"
14. style="position: relative; left: 10px"
15. />
16. </p>
17. <p><button type="submit">Submit</button></p>
18. </form>
19. </div>
```

Line 2: div tag will render error message for incorrect credentials

Line 7-18: A reactive form with two fields username and password is displayed

Add the following code to the **login.component.ts** file

```
1. import { Component } from '@angular/core';
2. import { FormBuilder, FormGroup } from '@angular/forms';
3. import { Router } from '@angular/router';
4. import { LoginService } from './login.service';
5. @Component({
6.   templateUrl: './login.component.html',
7.   styleUrls: ['./login.component.css'],
8. })
9. export class LoginComponent {
10.   invalidCredentialMsg!: string;
11.   loginForm!: FormGroup;
12.   constructor(
13.     private loginService: LoginService,
14.     private router: Router,
15.     private formbuilder: FormBuilder
16.   ) {
17.     this.loginForm = this.formbuilder.group({
18.       username: [],
19.       password: [],
20.     });
21.   }
22.   onFormSubmit(): void {
23.     const uname = this.loginForm.value.username;
24.     const pwd = this.loginForm.value.password;
25.     this.loginService
26.       .isUserAuthenticated(uname, pwd)
27.       .subscribe({next:(authenticated) => {
28.         if (authenticated) {
29.           this.router.navigate(['/books']);
30.         } else {
31.           this.invalidCredentialMsg = 'Invalid Credentials. Try again.';
32.         }

```

```
33. });  
34. }  
35. }
```

Line 25: onFormSubmit() method is invoked when the submit button is clicked in Login Form

Line 26-27: Fetching username and password values from the form

Line 28: Invoking isUserAuthenticated method of LoginService class which will check for the validity of username and password values and returns a Boolean value

Line 31-35: If the response is true, it will navigate to BooksComponent else assigns an error message to invalidCredentialMsg property

Add the following code to the **user.ts** file inside Login folder.

```
1. export class User {  
2.   constructor(public userId: number, public username: string, public password: string) { }  
3. }
```

Line 1-3: A User model class with three properties userId, username, and password is created

Add the following code to the **login.service.ts** file present inside login folder.

```
1. import { Injectable } from '@angular/core';  
2. import { Observable, of } from 'rxjs';  
3. import { map } from 'rxjs/operators';  
4. import { User } from './user';  
5. const USERS = [  
6.   new User(1, 'user1', 'user1'),  
7.   new User(2, 'user2', 'user2')  
8. ];  
9. const usersObservable = of(USERS);  
10. @Injectable({  
11.   providedIn: 'root'  
12. })  
13. export class LoginService {  
14.   private isLoggedIn = false;  
15.   getAllUsers(): Observable<User[]> {  
16.     return usersObservable;  
17.   }  
18.   isUserAuthenticated(username: string, password: string): Observable<boolean> {  
19.     return this.getAllUsers().pipe(  
20.       map(users => {  
21.         const Authenticateduser = users.find(user => (user.username === username) && (user.password ===  
           password));  
22.         if (Authenticateduser) {  
23.           this.isLoggedIn = true;  
24.         } else {  
25.           this.isLoggedIn = false;  
26.         }  
27.         return this.isLoggedIn;  
28.       })  
29.     );  
30.   }  
31. }
```

```
29. );  
30. }  
31. isUserLoggedIn(): boolean {  
32.   return this.isloggedIn;  
33. }  
34. }
```

Line 4: Imports User model class

Line 6-9: Creates an array called USERS of type User

Line 10: Converts USERS array as an observable type

Line 18-20: getAllUsers() method returns users array in Observable manner

Line 21: isUserAuthenticated method takes username and password values as inputs and returns a Boolean value of type Observable

Line 22-32: Invokes getAllUsers() methods which returns an observable array. After receiving it, it will find the entered credentials exist in the array or not. If the user exists, assigns true value to isloggedIn property otherwise false value to it

Line 34-36: isUserLoggedIn() method returns the value of isloggedIn which we will use in LoginGuardService class

Create another service class called **login-guard.service** inside login folder and add the following code:

```
1. import { Injectable } from '@angular/core';  
2. import { CanActivate, Router } from '@angular/router';  
3. import { LoginService } from './login.service';  
4. @Injectable({  
5.   providedIn: 'root'  
6. })  
7. export class LoginGuardService implements CanActivate {  
8.   constructor(private loginService: LoginService, private router: Router) { }  
9.   canActivate(): boolean {  
10.    if (this.loginService.isUserLoggedIn()) {  
11.     return true;  
12.    }  
13.    this.router.navigate(['/login']);  
14.    return false;  
15.   }  
16. }
```

Line 8: Implements CanActivate interface to LoginGuardService class

Line 10: Overrides canActivate() method

Line 11-15: Invokes isUserLoggedIn method from LoginService class which returns a Boolean value representing whether a user is logged in or not. If the user logs in, canActivate returns true otherwise navigate to the login component asking the user to login first to access BooksComponent

Add the following code in **app-routing.module.ts**

```
1. ...  
2. const appRoutes: Routes = [  
3.   { path: '', redirectTo: '/login', pathMatch: 'full' },
```

```

4. {path: 'login',component:LoginComponent},
5. { path: 'books', component: BookComponent, canActivate:[LoginGuardService] },
6. { path: 'dashboard', component: DashboardComponent},
7. { path: 'detail/:id', component: BookDetailComponent},
8. { path: '**', component: PageNotFoundComponent },
9. ];
10. ...

```

Line 5: Binds the LoginGuardService to the books path.

Update app.module.ts as below:

```

1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { HttpClientModule } from '@angular/common/http';
4. import { FormsModule, ReactiveFormsModule } from '@angular/forms';
5. import { AppComponent } from './app.component';
6. import { BookComponent } from './book/book.component';
7. import { DashboardComponent } from './dashboard/dashboard.component';
8. import { BookDetailComponent } from './book-detail/book-detail.component';
9. import { AppRoutingModuleModule } from './app-routing.module';
10. import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
11. import { LoginComponent } from './login/login.component';
12. @NgModule({
13. imports: [BrowserModule, HttpClientModule, ReactiveFormsModule,FormsModule,
    AppRoutingModuleModule],
14. declarations: [AppComponent, LoginComponent, BookComponent, DashboardComponent,
    BookDetailComponent, PageNotFoundComponent],
15. providers: [],
16. bootstrap: [AppComponent]
17. })
18. export class AppModule { }

```

Asynchronous Routing

- When an Angular application has a lot of components, it will increase the size of the application. In such cases, the application takes a lot of time to load.
- To overcome this problem, asynchronous routing is preferred, i.e, modules must be loaded lazily only when they are required instead of loading them at the beginning of the execution

Lazy Loading has the following benefits:

- Modules are loaded only when the user requests for it
- Load time can be speeded up for users who will be visiting only certain areas of the application

Lazy Loading Route Configuration: To apply lazy loading on modules, create a separate routing configuration file for that module and map an empty path to the component of that module.

Considering an example in the previous concept, consider BookComponent. To load it lazily, create the **book-routing.module.ts** file inside book folder and map an empty path to BookComponent(Line 8-10)

```

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. const bookRoutes: Routes = [
6. {
7. path: "",
8. component: BookComponent,
9. canActivate: [LoginGuardService]
10. }
11. ];
12.
13. @NgModule({
14. imports: [RouterModule.forChild(bookRoutes)],
15. exports: [RouterModule]
16. })
17. export class BookRoutingModule { }
```

The lazy loading and re-configuration will happen only once, i.e., when the route is first requested. Module and routes will be available immediately for subsequent requests.

Create a **book.module.ts** file inside book folder and add the following code:

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

Line 7: Adds BookRoutingModule class to the imports array

Line 8: Adds BookComponent to the declarations property

In the root routing configuration file **app-routing.module**, bind 'book' path to the BookModule using **loadChildren** property as shown below

```
1. import { NgModule } from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3. import { BookDetailComponent } from './book-detail/book-detail.component';
4. import { BookComponent } from './book/book.component';
5. import { DashboardComponent } from './dashboard/dashboard.component';
6. import { LoginGuardService } from './login/login-guard.service';
7. import { LoginComponent } from './login/login.component';
8. import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
9. const appRoutes: Routes = [
10. { path: "", redirectTo: '/login', pathMatch: 'full' },
```

```

11. { path: 'login', component: LoginComponent },
12. { path: 'books', loadChildren: () => import('./book/book.module').then(m => m.BookModule) },
13. { path: 'dashboard', component: DashboardComponent },
14. { path: 'detail/:id', component: BookDetailComponent } ,
15. { path: '**', component: PageNotFoundComponent }
16. ];
17. @NgModule({
18.   imports: [
19.     RouterModule.forRoot(appRoutes)
20.   ],
21.   exports: [
22.     RouterModule
23.   ]
24. })
25. export class AppRoutingModuleModule { }

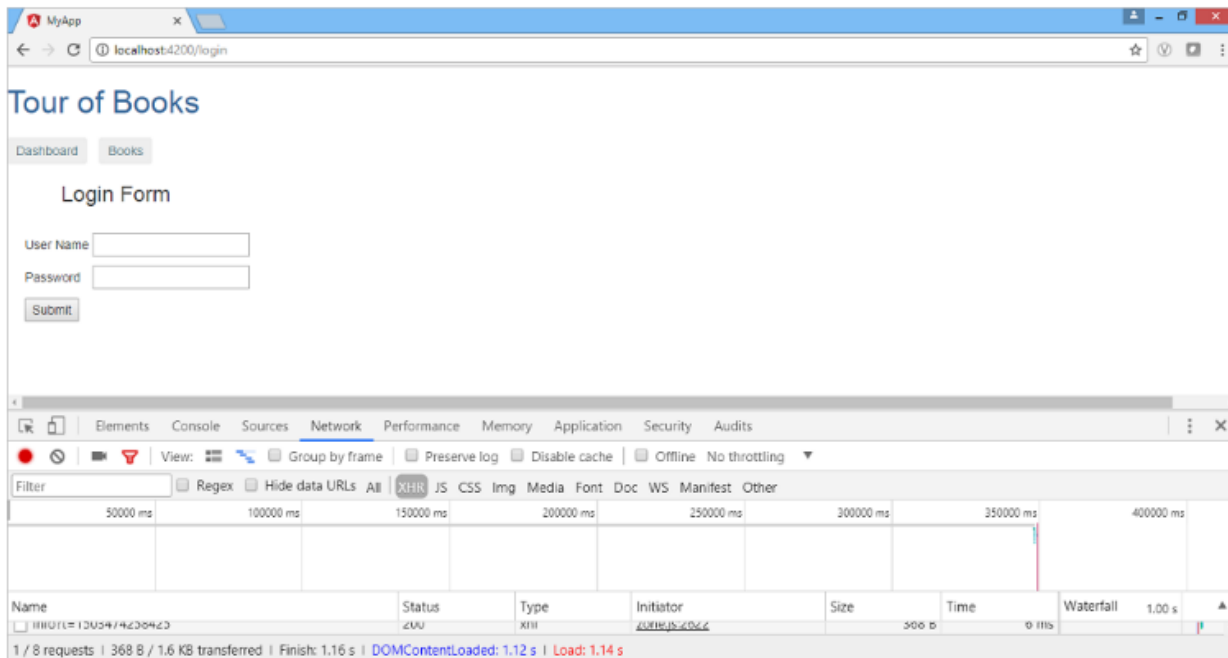
```

Line 12: Binds books path to BookModule using **loadChildren** property. From v8, Angular started making use of dynamic imports in lazy loading modules.

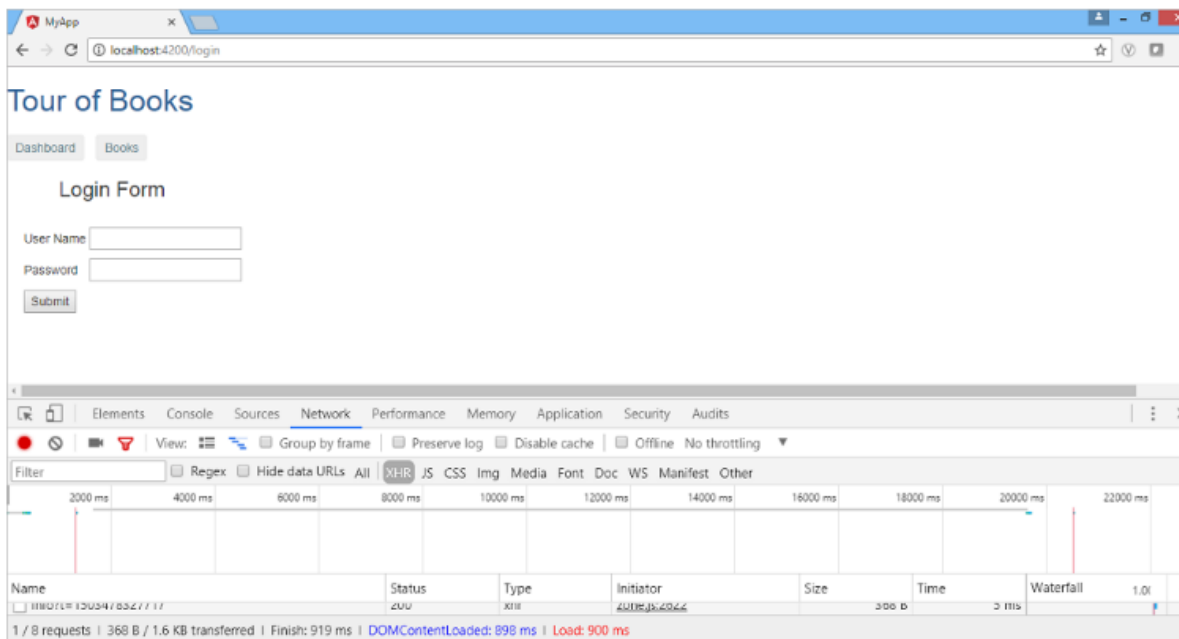
Note: Remove BookComponent class from app.module.ts file

Finally, it loads the requested route to the destination book component.

If lazy loading is not added to the demo, it has loaded in 1.14 s. Observe the load time at the bottom of the browser console. Press F12 in the browser and click Network tab and check the Load time



If lazy loading is added to the demo, it has loaded in 900 ms. As BookComponent will be loaded after login, the load time is reduced initially



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.

1. 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.

```

2. import { NgModule } from '@angular/core';
3. import { RouterModule, Routes } from '@angular/router';
4. import { BookComponent } from './book.component';
5. import { LoginGuardService } from '../login/login-guard.service';
6. import { DashboardComponent } from '../dashboard/dashboard.component';
7. import { BookDetailComponent } from '../book-detail/book-detail.component';
8. const bookRoutes: Routes = [
9. {
10. path: '',
11. component: BookComponent,
12. children: [
13. { path: 'dashboard', component: DashboardComponent },
14. { path: 'detail/:id', component: BookDetailComponent }
15. ],
16. canActivate: [LoginGuardService]
17. }];
18. @NgModule({
19. imports: [RouterModule.forChild(bookRoutes)],
20. exports: [RouterModule]

```

```

21. })
22. export class BookRoutingModule { }

```

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 { }

```

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>

```

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>
```

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

```
1. import { NgModule } from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3. import { LoginComponent } from './login/login.component';
4. import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
5. const appRoutes: Routes = [
6.   { path: '', redirectTo: '/login', pathMatch: 'full' },
7.   { path: 'login', component: LoginComponent },
8.   { path: 'books', loadChildren: () => import('./book/book.module').then(m => m.BookModule) },
9.   { path: '**', component: PageNotFoundComponent }
10. ];
11. @NgModule({
12.   imports: [
13.     RouterModule.forRoot(appRoutes)
14.   ],
15.   exports: [
16.     RouterModule
17.   ]
18. })
19. export class AppRoutingModuleModule { }
```

6. Update **app.module.ts** as below:

```
1. import { NgModule } from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { HttpClientModule } from '@angular/common/http';
4. import { ReactiveFormsModule } from '@angular/forms';
5. import { AppComponent } from './app.component';
6. import { AppRoutingModuleModule } from './app-routing.module';
7. import { LoginComponent } from './login/login.component';
8. @NgModule({
9.   imports: [BrowserModule, HttpClientModule, ReactiveFormsModule, AppRoutingModuleModule],
10.   declarations: [AppComponent, LoginComponent],
11.   providers: [],
12.   bootstrap: [AppComponent]
13. })
14. export class AppModule { }
```

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

```
1. ...
2. gotoDetail(book: Book): void {
3.   this.router.navigate(['/books/detail', book.id]);
4. }
5. ...
```

8. Update **dashboard.component.html**

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

9. Update dashboard.component.ts

```
1. import { Component, OnInit } from '@angular/core';
2. import { Router } from '@angular/router';
3. import { Book } from '../book/book';
4. import { BookService } from '../book/book.service';
5. @Component({
6.   selector: 'app-dashboard',
7.   templateUrl: './dashboard.component.html',
8.   styleUrls: ['./dashboard.component.css']
9. })
10. export class DashboardComponent implements OnInit {
11.   books: Book[] = [];
12.   constructor(
13.     private router: Router,
14.     private bookService: BookService) { }
15.   ngOnInit(): void {
16.     this.bookService.getBooks()
17.       .subscribe(books => this.books = books.slice(1, 5));
18.   }
19.   gotoDetail(book: Book): void {
20.     this.router.navigate(['/books/detail', book.id]);
21.   }
22. }
```