EXPERIMENT-1

1A - Angular Application Setup

AIM: Setting up the development environment

DESCRIPTION:

- Angular is a powerful and performance-efficient JavaScript framework for building single-page applications (SPAs) for both web and mobile. With its component-based architecture, Angular allows developers to create complex, customizable, modern, responsive, and user-friendly web applications. Widely used by leading tech companies, Angular's robust features and tooling make it a top choice for building scalable and maintainable web applications.

WHY WE CHOOSE ANGULAR?

- **CROSS-BROWSER COMPATIBILITY:**

Angular helps developers create web applications that are easily compatible across different browsers, addressing the challenges faced with earlier versions of the framework

- **TYPESCRIPT SUPPORT:**

Angular is written in TypeScript, a superset of JavaScript that provides better tooling, type safety,and improved productivity for developers

Improved Mobile Development:

Separate Concerns:

Angular addresses the distinct concerns of desktop and mobile applications, making it easier to develop for both platforms

Better Performance:

Angular's modern approach to handling issues like browser rendering, animation, and accessibility results in improved performance across all components

Seamless Integration:

Angular seamlessly integrates with various server-side technologies and databases, providing a complete client- side solution for web and mobile development

Setting up an AngularJS application involves a series of steps to prepare your development environment and create a basic project structure

1. *Install the latest version of Node.js and npm , which are required for running Angular applications*.

Code

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

2 .Install the Angular CLI, a powerful command-line tool that simplifies the process of creating, developing, and maintaining Angular applications.

Code

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

3 .Use Visual Studio Code, a popular and feature-rich code editor, to write, debug, and manage your Angular projects.

Code

```
ng new my-angular-project
```

Code

```
cd my-angular-project
```

Code

```
cd my-angular-project
```

1B. Components and Modules

AIM : Create a new component called hello and render Hello Angular on the page

DESCRIPTION:

Components are the basic building blocks of an Angular application. They emphasize the separation of concerns and allow for independent development of different parts of the application. Components are also reusable, which can save time and effort.

HOW TO CREATE COMPONENTS

The best way to create a component is with the Angular CLI.

Creating a component using the Angular CLI

To create a component using the Angular CLI:

1. From a terminal window, navigate to the directory containing your application.
2. Run the **ng generate component <component-name>** command, where **<component-name>** is the name of your new component.

By default, this command creates the following:

- A directory named after the component
- A component file, **<component-name>.component.ts**
- A template file, **<component-name>.component.html**
- A CSS file, **<component-name>.component.css**
- A testing specification file, **<component-name>.component.spec.ts**

Where **<component-name>** is the name of your component.

Component Structure :

app.component.ts

This file contains the TypeScript code for the component, including its properties, methods, and logic

app.component.html

This file contains the HTML template for the component, which defines the view that will be displayed

app.component.css

This file contains the CSS styles for the component, which define the appearance of the view.

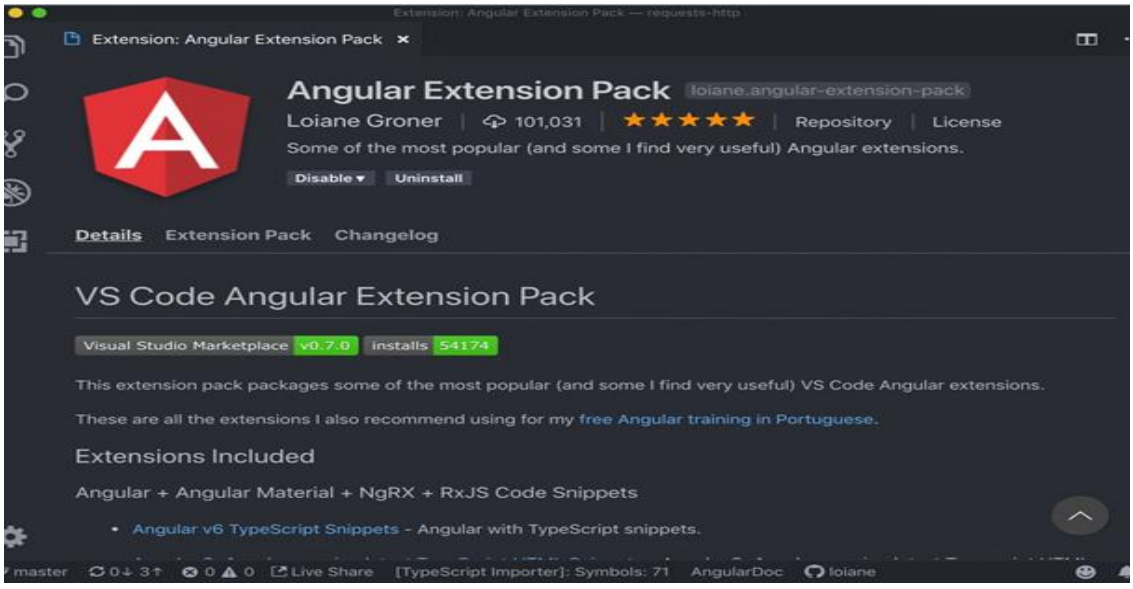
Modules

Modules are containers for components, directives, pipes, and services. They provide a wayto organize andstructure your application

DECLARATION OF A MODULE

angular.module(name.[requires].[configFn]);

AppModule:



declarations	Contains all user-defined components, directives, and pipes.
imports	Contains all module classes to be used across the application.
providers	Contains all service classes.
bootstrap	Contains the root component to load.

1C : Elements of Template

AIM : Add an event to the hello component template and when it is clicked, it should change the courseName

DESCRIPTION:

1.Templates separate the view layer from the rest of the framework.

2.You can change the view layer without breaking the application.

3. Templates in Angular represent a view and its role is to display data and change the data whenever an event occurs.

4. 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

Source code:

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


@Component({
  selector: 'app-hello',
```

```
  template: `
    <h1>Welcome</h1>
    <h2>Course Name: angular</h2>
  `,
  styleUrls: ['./hello.component.css']
})
export class HelloComponent {


}
```

External Template:

Example of External Template

- An **external template** in Angular is when the component's **HTML content is placed in a separate .html file**, and is linked to the component using the template Url property in the @Component decorator.
- · It binds the external template with a component using the template Url option.

Code Example

· app.component.html

Source code:

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

· app.component.ts

Source code:

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

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

}
```

Basic Elements of Template Syntax:

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 the components' data can be accessed in a template. For interpolation, double curly braces {{ }} are used.

Template Expression:

```
{{ expression }}
```

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

· Angular first evaluates the expression and returns the result as a string. The scope of a template expression is a component instance.

Template Statement:

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

· (event) = statement

· For example (click) = "changeName()"

· This is called event binding. In Angular, all events should be placed in ( ).

Example code

Hello.component.html

Souce code:

```
<h1>Welcome</h1>
<h2>Course Name: {{ courseName }}</h2>
<h2><button (click)="changeName()">Change Course</button></h2>
```

Hello.component.ts

Source code:

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

@Component({
  selector: 'app-hello',
```

```
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent {
  courseName='angular'
  changeName() {
    this.courseName='react'
  }

}
```

1D: Change Detection

AIM: progressively building the PoolCarz application

DESCRIPTION:

Change detection is a systema c process of identifying modifications made to so ware over me. In the development of the PoolCarz application on—a carpooling and ride-sharing platform—this concept plays a pivotal role in enabling agile development, improving maintainability, and ensuring consistent delivery of features. This case study explores how change detection on was applied progressively during the development phases of PoolCarz.

1. Introduction

Change Detection on in AngularJS is the process by which the framework synchronizes the data model and the view.

Whenever the data in the model changes, AngularJS automatically updates the corresponding parts of the view, and vice versa.

2. Concept Overview
   AngularJS uses a digest cycle to detect changes in data and update the DOM. This process ensures that the UI always reflects the latest state of the model.df
3. Key Mechanisms
   $scope Object: AngularJS stores all application data and methods inside the $scope object.
   $watch Func on: AngularJS internally keeps track of variables using $watch. It monitors expressions and triggers a digest when changes occur.

Digest Cycle: A digest cycle is a loop that checks all watched expressions in the $scope and updates the view if any changes are detected.

4. Digest Cycle Workflow User interaction or code modifies the model. $digest() cycle starts.
   All $watch expressions are checked for updates. If a change is found, the watcher callback is triggered. The view is updated accordingly.

Example:

```
<div ng-app="myApp" ng-controller="myCtrl">
<input type="text" ng-model="name">
<p>Hello, {{name}}</p>
</div>
```

```
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', func on($scope) {
$scope.name = "Sandeep";
});
</script>
```

Explanation:

When the user types in the input field, AngularJS detects the change and automatically updates the element due to its change detection on mechanism.

Performance Considerations : Too many $watch expressions can slow down the digest cycle. Use one-me bindings ({{ ::expression }}) for static data to improve performance.

Summary:

Change Detection is at the core of AngularJS's two-way data binding feature. It ensures synchronization on between the model and the view using $watch, $digest, and $apply cycles.

Exercise-2

2(a) Course Name: Angular JS

Module Name: Structural Directives – ngIf

Create a login form with username and password fields. If the user enters the correct credentials, it should render a "Welcome <<username>>" message otherwise it should render "Invalid !!! Please try again..." message.

Implementing a login form with ng-if in AngularJS:

This example demonstrates how to create a simple login form using AngularJS with the ng-if directive to conditionally display messages based on login credentials.

Definition and Usage:

The **ng-if directive** removes the HTML element if the expression evaluates to false.

If the if statement evaluates to true, a copy of the Element is added in the DOM.

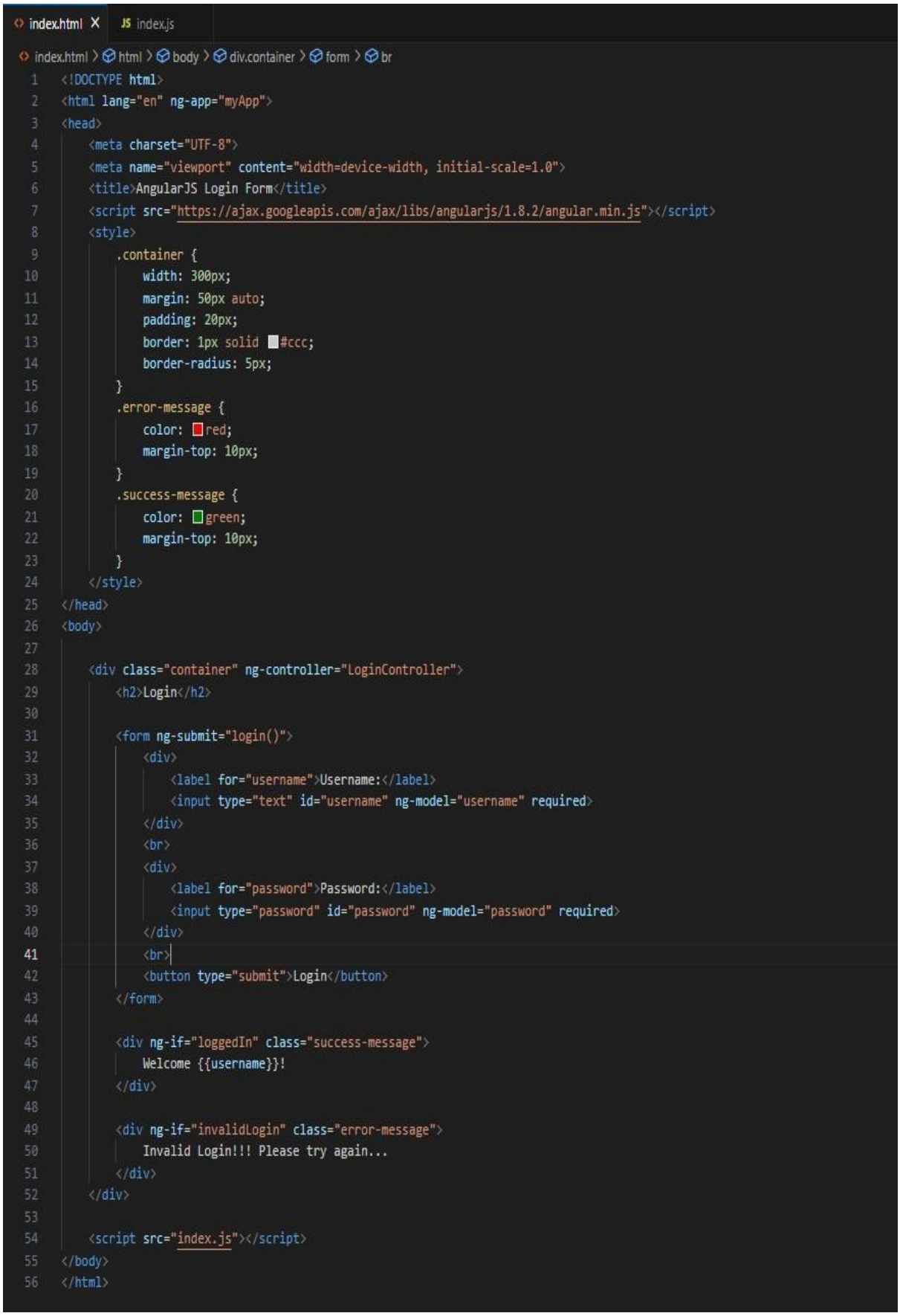
The ng-if directive is different from the ng-hide, which hides the display of the element, where the ng-if directive completely removes the element from the DOM.

Syntax:

<element ng-if="expression"></element>

Expression: An expression that will completely remove the element if it returns false. If it returns true, a copy of the element will be inserted instead.

HTML Code:(login.html)



```
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>AngularJS Login Form</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
    <style>
        .container {
            width: 300px;
            margin: 50px auto;
            padding: 20px;
            border: 1px solid #ccc;
            border-radius: 5px;
        }
        .error-message {
            color: red;
            margin-top: 10px;
        }
        .success-message {
            color: green;
            margin-top: 10px;
        }
    </style>
</head>
<body>

    <div class="container" ng-controller="LoginController">
        <h2>Login</h2>

        <form ng-submit="login()">
            <div>
                <label for="username">Username:</label>
                <input type="text" id="username" ng-model="username" required>
            </div>
            <br>
            <div>
                <label for="password">Password:</label>
                <input type="password" id="password" ng-model="password" required>
            </div>
            <br>
            <button type="submit">Login</button>
        </form>

        <div ng-if="loggedIn" class="success-message">
            Welcome {{username}}!
        </div>

        <div ng-if="invalidLogin" class="error-message">
            Invalid Login!!! Please try again...
        </div>
    </div>

    <script src="index.js"></script>
</body>
</html>
```

Angular js Code:(app.js)

index.html index.js

index.js > ...

```
angular.module('myApp', [])
    .controller('LoginController', function($scope) {
        $scope.username = '';
        $scope.password = '';
        $scope.loggedIn = false;
        $scope.invalidLogin = false;

        $scope.login = function() {
            // Simulate authentication (replace with actual backend API call)
            if ($scope.username === 'testuser' && $scope.password === 'password123') {
                $scope.loggedIn = true;
                $scope.invalidLogin = false;
            } else {
                $scope.loggedIn = false;
                $scope.invalidLogin = true;
            }
        };
    });
```

Output:

Output:

Explanation:

1. **ng-app="myApp":** This directive initializes the AngularJS application and connects it to the HTML document.
2. **ng-controller="LoginController":** This directive attaches the LoginController to the div element, making its scope available within that section.
3. **ng-model="username" and ng-model="password":** These directives create a two-way data binding between the input fields and the $scope.username and $scope.password properties in the controller, respectively.
4. **ng-submit="login()":** This directive calls the login() function in the controller when the form is submitted.
5. **ng-if="loggedIn" and ng-if="invalidLogin":** These directives conditionally render or remove the div elements from the DOM based on

the values of the loggedIn and invalidLogin boolean variables in the controller. If loggedIn is true, the "Welcome" message is displayed. If invalidLogin is true, the "Invalid Login" message is displayed. Otherwise, the respective message div is removed from the DOM.

6. LoginController: This controller handles the login logic.
    - $scope.username and $scope.password store the input values.
    - $scope.loggedIn and $scope.invalidLogin are boolean variables that control the visibility of the messages using ng-if.
    - $scope.login(): This function simulates user authentication. In a real application, this would involve sending the credentials to a backend server for validation. If the credentials are correct, loggedIn is set to true and invalidLogin to false. Otherwise, loggedIn is set to false and invalidLogin to true.

How it works:

- When the user enters credentials and clicks the login button, the login() function in LoginController is called due to ng-submit.
- The login() function evaluates the credentials.
- If the credentials are correct, $scope.loggedIn is set to true, and $scope.invalidLogin is set to false.
- Because $scope.loggedIn is now true, the div with the "Welcome" message is added to the DOM and displayed by ng-if.
- If the credentials are incorrect, $scope.loggedIn is set to false, and $scope.invalidLogin is set to true.
- The div with the "Invalid Login" message is then added to the DOM and displayed.

2(b) Course Name: Angular JS

Module Name: ngFor

Create a courses array and rendering it in the template using ngFor directive in a list format.

Definition:

*ngFor is used to loop through the dynamic lists in the DOM. Simply, it is used to build data presentation lists and tables in HTML DOM.

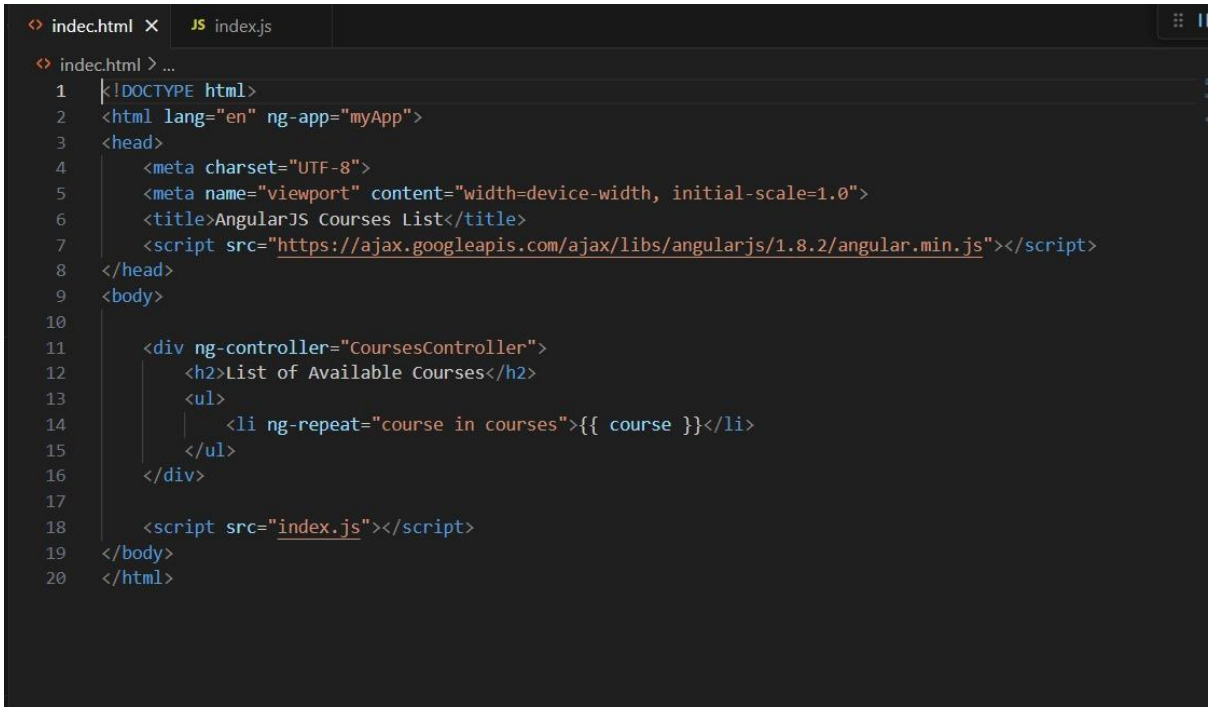
Syntax:

<element *ngFor="let item of items">...</element>

Example:

This example demonstrates how to create an array of courses in an AngularJS controller and then render that array in an unordered list (<ul>) using the ng-repeat directive.

Html code:(index.html)



Angular js code: (index.js)

indec.html | index.js

index.js > ...

```
angular.module('myApp', [])
    .controller('CoursesController', function($scope) {
        $scope.courses = [
            'HTML5',
            'CSS3',
            'JavaScript',
            'AngularJS',
            'Node.js',
            'MongoDB',
            'Express.js'
        ];
    });
```

Output:

List of Available Courses

- HTML5
- CSS3
- JavaScript
- AngularJS
- Node.js
- MongoDB
- Express.js

Explanation:

ng-app="myApp": This directive initializes the AngularJS application on the HTML document.

ng-controller="CoursesController": This attaches the CoursesController to the div element, making its scope and data available within that section.

CoursesController: This controller is defined in app.js and is responsible for managing the data to be displayed.

$scope.courses: An array of strings is created to store the names of the courses. In a real-world application, this data could be fetched from a backend API.

ng-repeat="course in courses": This is the core of the list rendering functionality.

ng-repeat is a repeater directive in AngularJS. It iterates over a collection (in this case, the courses array) and creates new HTML elements for each item in the collection**.**

course in courses: This syntax tells ng-repeat to iterate over the courses array and assign each item to a temporary variable named course for the current iteration.

The <li> element (and its content {{ course }}) is repeated for each item in the courses array.

{{ course }}: This is an AngularJS interpolation or data binding expression that displays the value of the course variable for the current iteration.

2(c) Course Name: Angular JS

Module Name: ngSwitch

Display the correct option based on the value passed to **ngSwitch directive**.

Defintion:

The **ng-switch Directive** in AngularJS is used to specify the condition to show/hide the child elements in HTML DOM. The HTML element will be displayed only if the expression inside the ng-switch directive returns true otherwise it will be hidden. It is supported by all HTML elements.

Syntax:

```
<element ng-switch="expression">
    <element ng-switch-when="value"> Contents... </element>
    <element ng-switch-when="value"> Contents... </element>
    <element ng-switch-default> Contents... </element>
</element>
```

Parameter:

- **expression:** It specifies the element has matched and will be displayed otherwise will be discarded.

Example:

This example demonstrates how to use the ng-switch directive in AngularJS to display different content based on the value of a selected option from a dropdown list.

Html code: (index.html)

```
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>AngularJS ngSwitch Example</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
</head>
<body>

    <div ng-controller="SwitchController">
        <h2>Select a Category</h2>

        <select ng-model="selectedCategory">
            <option value="">-- Select a Category --</option>
            <option value="electronics">Electronics</option>
            <option value="clothing">Clothing</option>
            <option value="books">Books</option>
            <option value="home">Home & Garden</option>
        </select>

        <hr>

        <div ng-switch="selectedCategory">
            <div ng-switch-when="electronics">
                <h3>Electronics Category</h3>
                <p>Explore our latest gadgets, smartphones, and accessories.</p>
            </div>
            <div ng-switch-when="clothing">
                <h3>Clothing Category</h3>
                <p>Browse our collection of stylish apparel for men and women.</p>
            </div>
            <div ng-switch-when="books">
                <h3>Books Category</h3>
                <p>Discover a wide range of books, from bestsellers to classics.</p>
            </div>
            <div ng-switch-when="home">
                <h3>Home & Garden Category</h3>
                <p>Find products to enhance your home and outdoor living space.</p>
            </div>
            <div ng-switch-default>
                <h3>Please select a category to see its details.</h3>
            </div>
        </div>
    </div>

    <script src="index.js"></script>
</body>
</html>
```

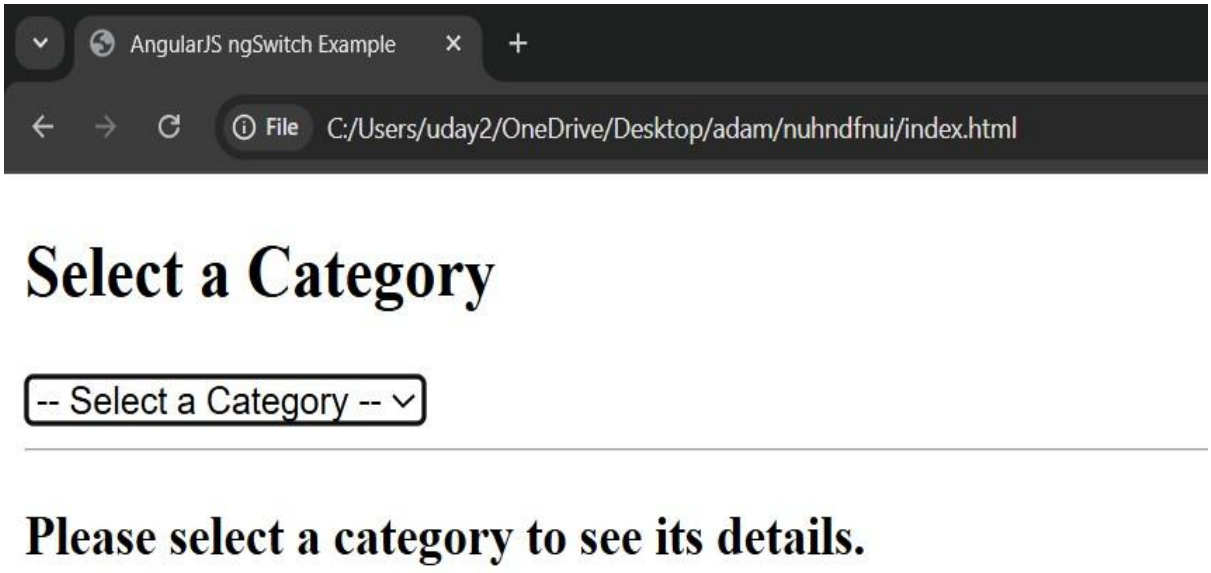
Angular js code: (index.js)

index.html JS index.js

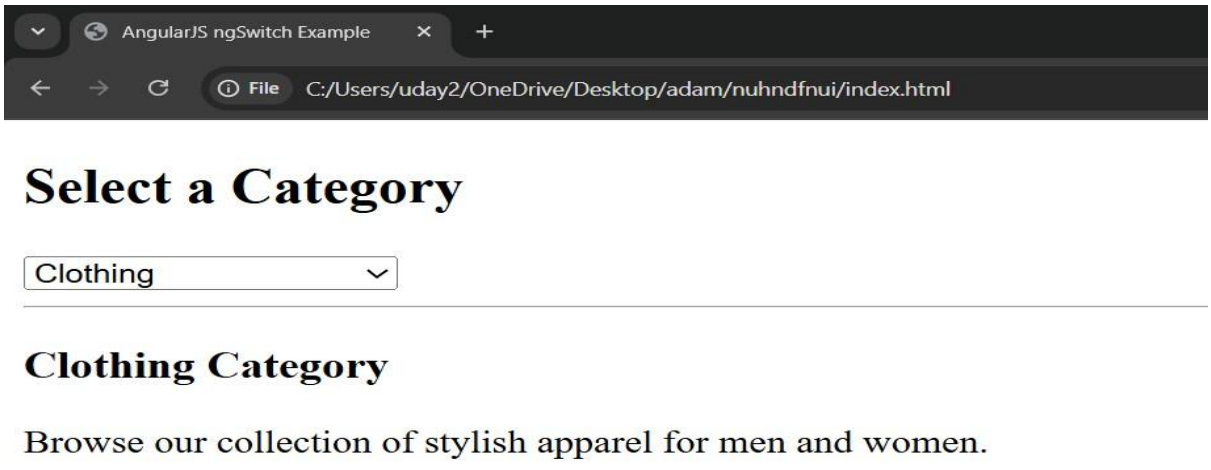
JS index.js > ...

```
1  angular.module('myApp', [])
2      .controller('SwitchController', function($scope) {
3          $scope.selectedCategory = ''; // Initialize with an empty value
4      });
```

Output:



Output:



Explanation:

- **ng-app="myApp":** Initializes the AngularJS application.
- **ng-controller="SwitchController":** Attaches the SwitchController to the div element, providing access to its scope.
- **<select ng-model="selectedCategory">:** Creates a dropdown (select box) where the selected value is bound to the $scope.selectedCategory variable. According to Tutlane the ng-model directive is used to bind the values of HTML controls.
- ** :** This is the main container for the ng-switch directive.
- It specifies the expression (selectedCategory) that AngularJS will evaluate to determine which content to display.
- ** , (etc.):** These directives represent individual cases within the ng-switch statement.

- If the value of selectedCategory matches the value specified in ng-switch-when, the content within that div will be displayed.
- According to Logicmojo ng-switch-when does not hide the element, it removes it from the DOM.
- **<div ng-switch-default>:** This directive defines the default content to be displayed if none of the ng-switch-when conditions match the selectedCategory value.
- It acts like the default case in a traditional switch statement.

How it works:

- When the page loads, selectedCategory is initially an empty string, so the ng-switch-default content is displayed.
- The user selects an option from the dropdown list.
- The ng-model="selectedCategory" directive updates the $scope.selectedCategory variable with the selected value.
- The ng-switch directive re-evaluates the selectedCategory expression.
- Based on the new value, AngularJS finds the corresponding ng-switch-when directive and renders its content. The previously displayed content (and its associated HTML elements) is removed from the DOM.
- If no ng-switch-when matches the selectedCategory, the ng-switch-default content will be displayed.

2(d) Course Name: Angular JS

Module Name: Custom Structural Directive

Create a custom structural directive called 'repeat' which should repeat the element given a number of times.

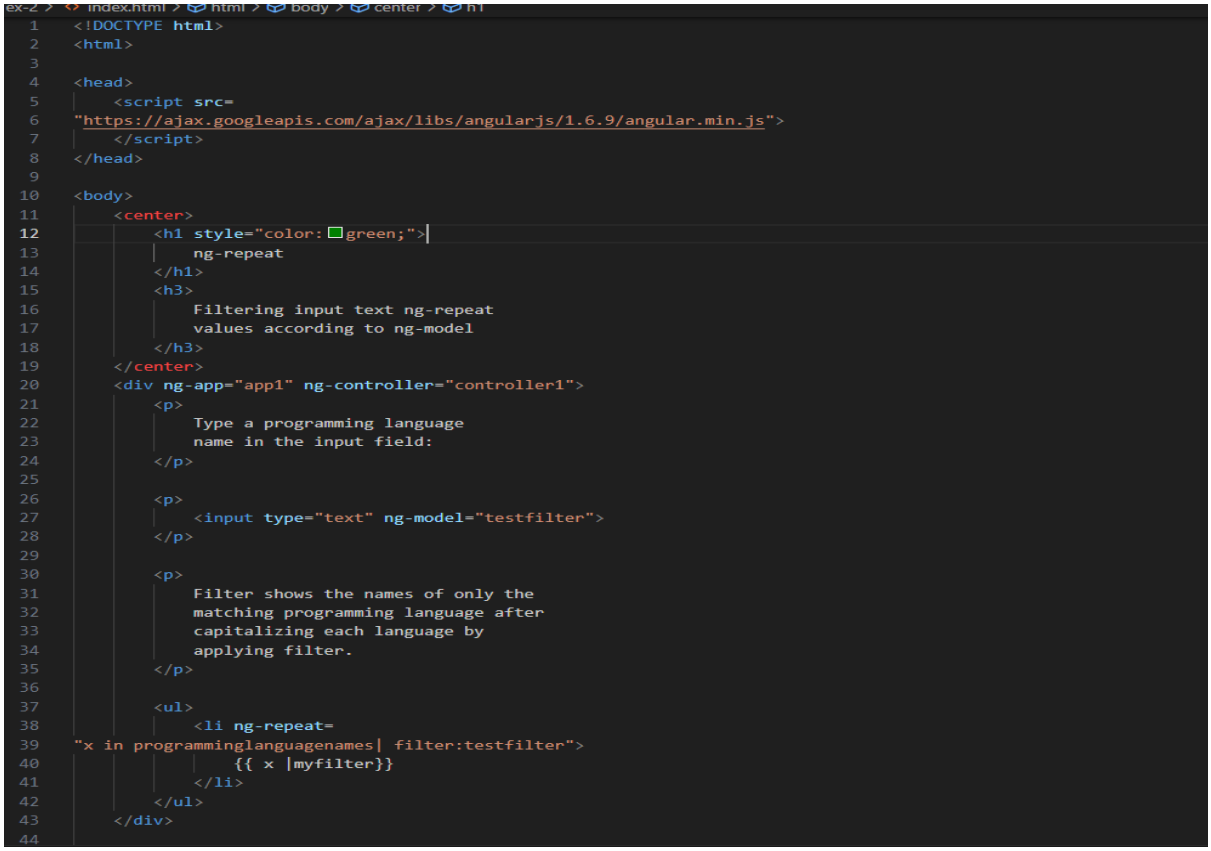
Definition:

Angular-JS ng-repeat directive is a handy tool to repeat a set of HTML code for a number of times or once per item in a collection of items. ng-repeat is mostly used on arrays and objects.

Syntax :

{{keyName}}

Html code: (index.html)



```
<!DOCTYPE html>
<html>

<head>
    <script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
    </script>
</head>

<body>
    <center>
        <h1 style="color:green;">
            ng-repeat
        </h1>
        <h3>
            Filtering input text ng-repeat
            values according to ng-model
        </h3>
    </center>
    <div ng-app="app1" ng-controller="controller1">
        <p>
            Type a programming language
            name in the input field:
        </p>

        <p>
            <input type="text" ng-model="testfilter">
        </p>

        <p>
            Filter shows the names of only the
            matching programming language after
            capitalizing each language by
            applying filter.
        </p>

        <ul>
            <li ng-repeat=
"x in programminglanguagenames| filter:testfilter">
                {{ x |myfilter}}
            </li>
        </ul>
    </div>

```

```
    <script>
        var app = angular.module('app1', []);
        app.filter('myfilter', function () {
            return function (x) {
                var i, c, txt = "";
                for (i = 0; i < x.length; i++) {
                    c = x[i];
                    c = c.toUpperCase();
                    txt += c;
                }
                return txt;
            };
        });

        app.controller('controller1', function ($scope) {
            $scope.programminglanguagenames = [
                'cobol',
                'pascal',
                'ruby',
                'php',
                'perl',
                'python',
                'c',
                'c++',
                'java',
                'html',
                'css',
                'javascript',
                'basic',
                'lisp',
                'smalltalk',
                'bootstrap'
            ];
        });
    </script>
</body>

</html>
```

Output 1:

ng-repeat

Filtering input text ng-repeat values according to ng-model

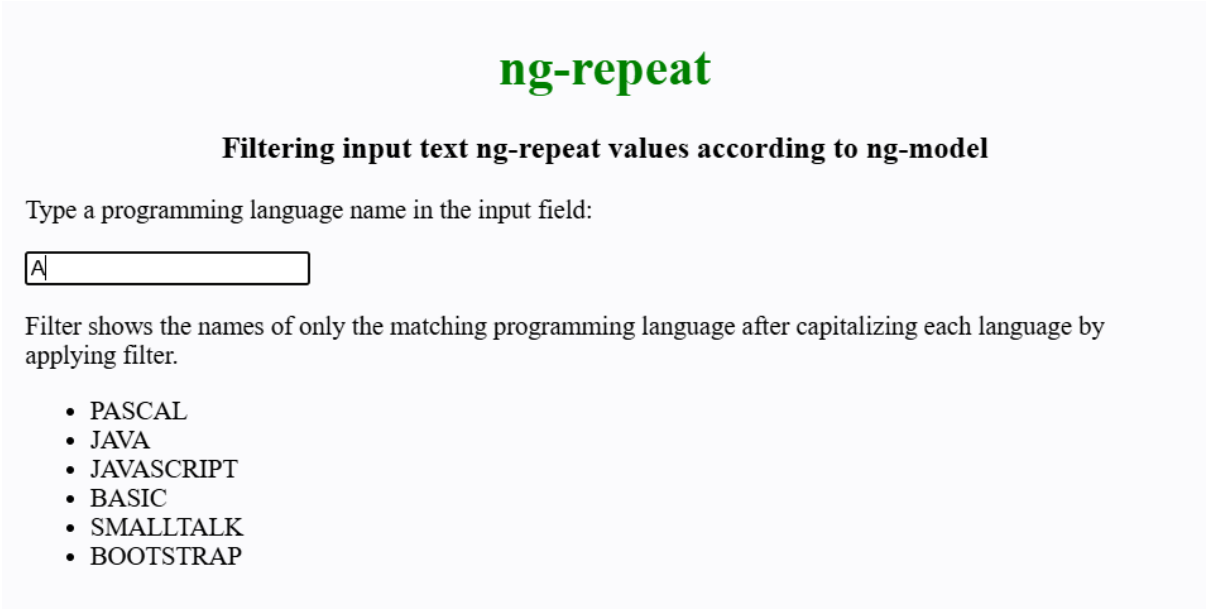
Type a programming language name in the input field:

c

Filter shows the names of only the matching programming language after capitalizing each language by applying filter.

- COBOL
- PASCAL
- C
- C++
- CSS
- JAVASCRIPT
- BASIC

Output-2:


ng-repeat
Filtering input text ng-repeat values according to ng-model
Type a programming language name in the input field:
A
Filter shows the names of only the matching programming language after capitalizing each language by applying filter.
PASCAL
JAVA
JAVASCRIPT
BASIC
SMALLTALK
BOOTSTRAP

Experiment-3

3A – Attribute Directives - ngStyle

Aim: Apply multiple CSS properties to a paragraph in a component using ngStyle.

Description:

There are three types of Angular Directives:

- Components
- Attribute Directive
- Structural Directives

Attribute Directives listen to and modify the behavior of other HTML elements, attributes, properties, and components.

Structural Directives render the content of an element conditionally or repeatedly.

ngStyle is an Angular JS directive that allows dynamic inline CSS binding. It changes styles dynamically based on component variables or expressions.

Syntax:

<p [ngStyle]="{'property': value}">Content</p>

Why do we need it?

- Changes styles based on conditions.
- Apply multiple CSS properties dynamically.
- Style changes without modifying CSS files.
- Great for Interactive UI components.

Source Code:

```
<p [ngStyle]="
    {'font-size.px':30,
    'color':'red',
    'font-weight':'bold',
    'font-style':'italic',
    'text-decoration':'underline'}">
    This paragragh will appear in bold, italic, in red color, underlined, and with font size 30px.
</p>

<router-outlet />
```

Output:

This paragragh will appear in bold, italic, in red color, underlined, and with font size 30px.

3B – ngClass

Aim: Apply multiple CSS classes to the text using ngClass directive.

Description: ngClass is used to add and remove CSS classes on an HTML element. The CSS classes are updated depending on the type of expression evaluation.

The types of expressions are:

i. String
the CSS classes listed in the string(space delimited) are added
e.g., [ngClass]="'class1 class2'"

ii. Array
the CSS classes declared as Array elements are added
e.g., [ngClass]="['first', 'second']"

iii. Object
keys are CSS classes that get added when the expression given in the value evaluates to a truthy value, otherwise they are removed.
e.g., [ngClass]="{'class1': true, 'class2': var1 }"

Source Code:

app.html

```
<p [ngClass]="{'Hover':true,'Color':true,'Big':true}">
    This paragraph will appear with CSS classes "Hover","Color","Big"
</p>

<router-outlet />
```

app.css

```
.Color{
    color: blue;
}
.Hover:hover{
    color: blueviolet;
    font-size: large;
}
.Big{
    font-size: xx-large;
```

```
}
```

Output:

Before hover:

After hover:

3C – Custom Attribute Directives

Aim: 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.

Description:

We need custom attribute directives because:

- Modify the elements without changing the DOM layout, unlike structural directives that add or remove elements.
- Change the appearance or behavior of DOM elements dynamically.

How to create custom attribute directives?

1. Create a directive using the following command

   ```
   ng generate directive <directive-name>
   ```

   This will create two files named “<directive-name>.ts” and “<directive-name>.spec.ts”.

2. Import ElementRef from @angular/core. ElementRef grants direct access to the host DOM element through its nativeElement property.

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

3. Add a variable to the constructor as follows:

```
constructor(private el: ElementRef) {}
```

Source Code:

app.html

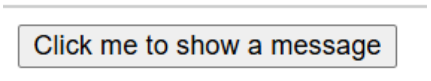
```
<button showMessage="Hello, this is your message!">
    Click me to show a message
</button>

<router-outlet />
```

show-message.ts

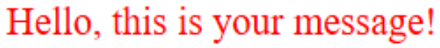
```
import { Directive, Input, HostListener, Renderer2, ElementRef } from '@angular/core';

@Directive({
  selector: '[showMessage]'
})
export class ShowMessageDirective {
  @Input('showMessage') message!: string;

  private messageElement: HTMLElement | null = null;

  constructor(private el: ElementRef, private renderer: Renderer2) {}

  @HostListener('click')
  onClick() {
    if (!this.messageElement) {
      this.messageElement = this.renderer.createElement('p');
      this.renderer.setStyle(this.messageElement, 'color', 'red');
      this.renderer.setStyle(this.el.nativeElement,'color','red');
      const text = this.renderer.createText(this.message);
      this.renderer.appendChild(this.messageElement, text);
      const parent = this.renderer.parentNode(this.el.nativeElement);
      if (parent) {
        this.renderer.insertBefore(parent, this.messageElement,
this.el.nativeElement.nextSibling);
      }
    }
  }
}
```

Output:

Before clicking:



After clicking:



Exercise-4

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)	{{ expression }} [target] = "expression"	Interpolation Property Attribute Class Style
One-way (Template - > Class)	(target) = "statement"	Event
Two-way	[(target)] = "expression"	Two way

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

Syntax:

```
<img [src]="imageUrl" />

<img bind-src="imageUrl" />
```

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

```
…
export class AppComponent
{
  imgUrl: string = 'assets/imgs/logo.jpg';
}
```

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

App.component.html

```
<img [src]="imgUrl" width="200" height="100" alt="Logo" />
```

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

Note: this can also be written as:

Problem Statement: Binding image with class property using property binding.

App.component.ts:

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  imgUrl = 'assets/imgs/logo.png';
}
```

App.component.html

```
<img [src] = ‘imgUrl’>
```

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,

```
<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.

Example:

app.component.ts

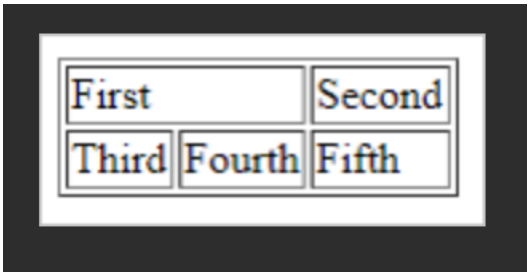
```
export class AppComponent {
  colspanValue: string = "2";
}
```

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

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  colspanValue: string = "2";
}
```

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:



Question : Binding colspan attribute of a table element to the class property.

App.component.ts:

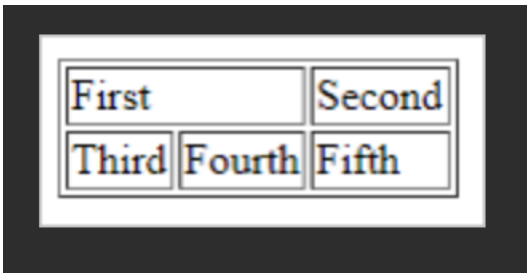
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  colspanValue: string = "2";
}
```

App.component.html:

```
<table border=”1”>
  <tr>
    <td [attr.colspan]=”colspanValue”>First</td>
    <td>Second</td>
  </tr>
  <tr>
```

```
   <td>Third</td>
   <td>Fourth</td>
   <td>Fifth</td>
  </tr>
</table>
```

Output:



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:

<img [src]='imgUrl'>

Example:
<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:

<button [style.color] = "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:

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

Problem Statement: Binding a textbox with a property using two-way data binding. The output is as shown below

App.component.ts

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

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

}

App.component.html:

```
<input type="text" [(ngModel)]="name" /> <br/>
<div>Your {{name}}</div>
```

App.module.ts:

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

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

Output:

Experiment 5

5A : Built-in Pipes

Aim : Display the product code in lowercase and product name in uppercase using built-in pipes.

Description :

Angular Pipes are powerful, lightweight functions designed to transform data directly within your templates, making your displayed information both readable and consistent.

Pipes are a special operator in Angular template expressions that allows you to transform data declaratively in your template. Pipes let you declare a transformation function once and then use that transformation across multiple templates. Angular pipes use the vertical bar character (|), inspired by the Unix pipe.

Angular provides a rich set of pre-built pipes to handle common data transformation needs, saving you time and effort. Here are a few commonly used pipes:

- **DatePipe:** Essential for displaying dates in various localized formats (e.g., {{ myDate | date:'fullDate' }}).
- **CurrencyPipe:** Formats numeric values into currency strings, respecting locale-specific symbols and decimals (e.g., {{ 123.45 | currency:'USD':'symbol':'1.2-2' }}).
- **UpperCasePipe & LowerCasePipe:** Simple yet effective for standardizing text casing (e.g., {{ myText | uppercase }}).
- **AsyncPipe:** Automatically subscribes to an Observable or Promise and unwraps its emitted values, managing subscriptions to prevent memory leaks (e.g., {{ myObservable | async }}).
- **JsonPipe:** Invaluable for debugging, it converts a JavaScript value into its JSON string representation (e.g., {{ myObject | json }}).

Built-in Pipes

Angular includes a set of built-in pipes in the @angular/common package:

- AsyncPipe
- CurrencyPipe
- DatePipe
- DecimalPipe
- I18nPluralPipe
- I18nSelectPipe
- JsonPipe
- KeyValuePipe
- LowerCasePipe
- PercentPipe
- SlicePipe
- TitleCasePipe
- UpperCasePipe

Using Pipes

Angular's pipe operator uses the vertical bar character (|) within a template expression. The pipe operator is a **binary operator**—the left-hand operand is the value passed to the transformation function, and the right-hand operand is the name of the pipe and any additional arguments (described below).

Total: {{ amount | currency }}

In this example, the value of amount is passed into the **CurrencyPipe**, where the pipe name is currency. It then renders the default currency for the user's locale.

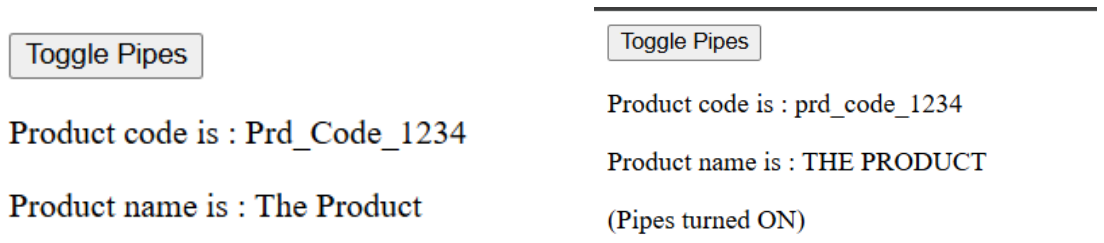
Combining Multiple Pipes in the Same Expression

You can apply multiple transformations to a value by using multiple pipe operators. Angular runs the pipes **from left to right**.

The following example demonstrates a combination of pipes to display a localized date in **all uppercase**:

The event will occur on {{ scheduledOn | date | uppercase }}.

Task : Display the product code in lowercase and product name in uppercase using built-in pipes.



Create New Angular Project using `ng new pipes`

Add toggle button and Product detail in app's html page:

expt5/pipes/src/app/app.html:

```
<button (click)="show()">Toggle Pipes</button>

@if(showVar){
    <p>Product code is : {{code | lowercase}}</p>
    <p>Product name is : {{name | uppercase}}</p>
    <p>(Pipes turned ON)</p>
}
@else{
    <p>Product code is : {{code }}</p>
    <p>Product name is : {{name }}</p>
}
<router-outlet />
```

expt5/pipes/src/app/app.ts :

```
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { LowerCasePipe, UpperCasePipe } from '@angular/common';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet,LowerCasePipe,UpperCasePipe],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('pipes');
  showVar=false;
  code='Prd_Code_1234';
  name='The Product';
  show() {
    if(this.showVar==false){
      this.showVar=true;
    }
    else{
      this.showVar=false;
    }
  }
}
```

5B: More Built-in Pipes

Aim:

Display product code in lowercase, product name in uppercase, price in currency format, and manufacturing date in full date format using pipes.

Description:

Here, in addition to `lowercase` and `uppercase`, we used:

- **CurrencyPipe** → formats price with ₹ symbol and decimals.
- **DatePipe** → converts raw date into a full readable date with time and timezone.

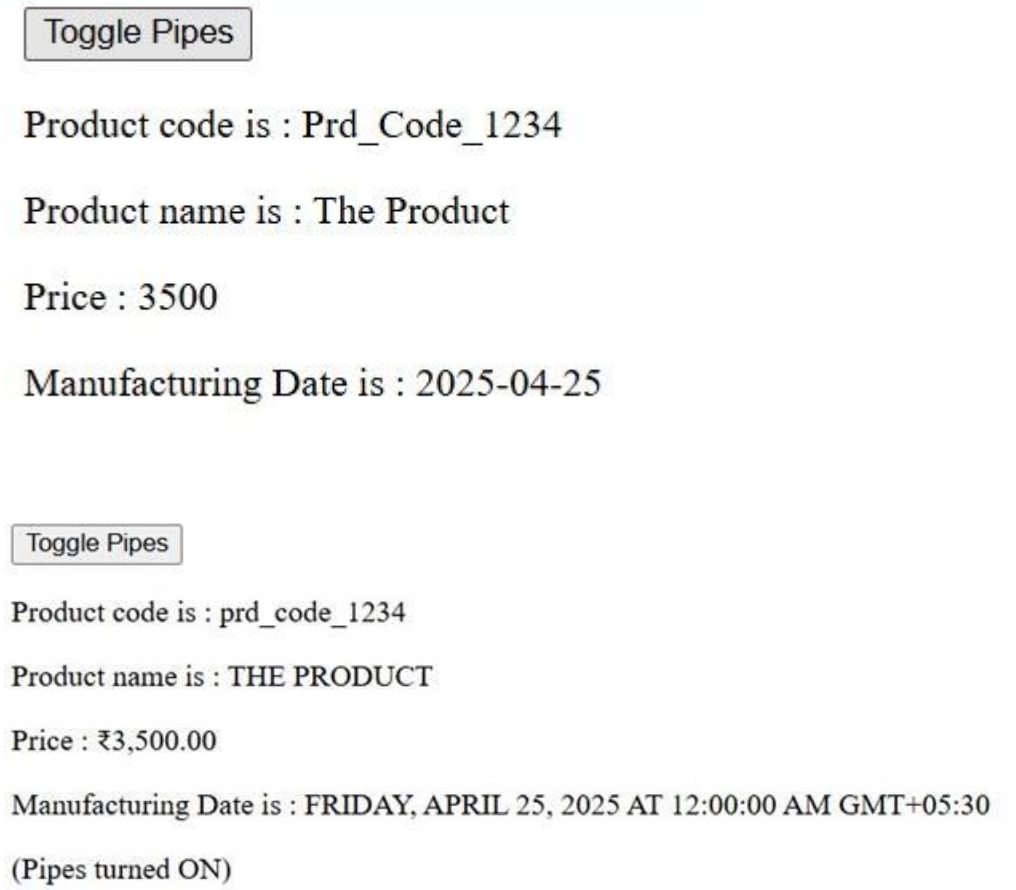
Code (app.ts):

```
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { LowerCasePipe, UpperCasePipe ,DatePipe, CurrencyPipe } from
'@angular/common';
@Component({
  selector: 'app-root',
  imports: [RouterOutlet,LowerCasePipe,UpperCasePipe, DatePipe,
CurrencyPipe],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('pipes');
  showVar=false;
  code='Prd_Code_1234';
  name='The Product';
  price = 3500;
  mfg_date = '2025-04-25';
  show() {
    this.showVar = !this.showVar;
  }
}
```

Code (app.html):

```
<button (click)="show()">Toggle Pipes</button>
 @if(showVar){
   <p>Product code is : {{code | lowercase}}</p>
   <p>Product name is : {{name | uppercase}}</p>
   <p>Price : {{price | currency:'INR'}}</p>
   <p>Manufacturing Date is : {{mfg_date | date:'full' | uppercase }}</p>
   <p>(Pipes turned ON)</p>
}
@else{
   <p>Product code is : {{code }}</p>
   <p>Product name is : {{name }}</p>
   <p>Price : {{price }}</p>
   <p>Manufacturing Date is : {{mfg_date }}</p>
}
<router-outlet />
```

Output:



Experiment 5C: Nested Component Basics

Aim:

Create a child component `course-list` and use it inside the app component to display courses.

Description:

Angular supports **nested components** where a parent component can include and interact with child components. Here, `course-list` is generated as a separate component and used in `app.component`.

Steps:

1. Generate course-list component inside `app`:

   ```
   ng g c course-list
   ```

   This creates:

   ```
   src/app/course-list/
      course-list.ts
      course-list.html
      course-list.css
   ```

2. **course-list.ts**

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

@Component({
  selector: 'app-course-list',
  templateUrl: './course-list.html',
  styleUrl: './course-list.css'
})
export class CourseList {
  courses: string[] = ['Angular', 'React', 'Vue', 'Node.js'];
}
```

3. **course-list.html**

```
<h3>Available Courses</h3>
<ul>
  <li *ngFor="let course of courses">{{ course }}</li>
</ul>
```

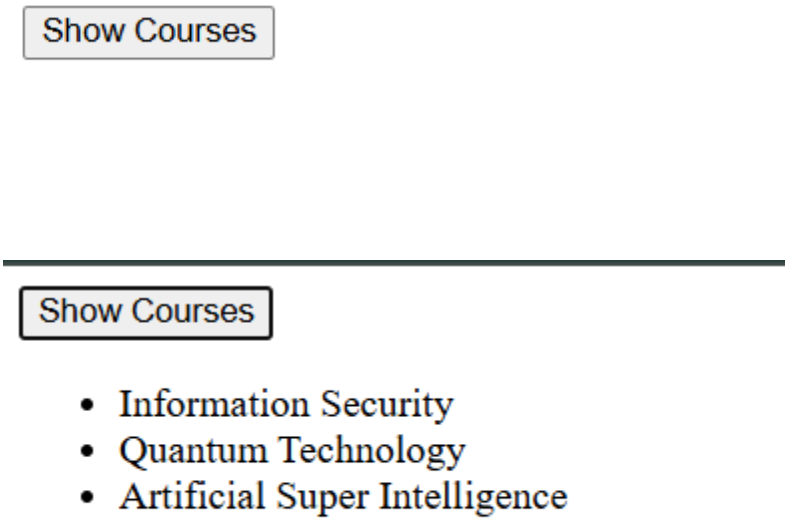
4. **app.html** (parent component)

```
<button (click)="show()">Toggle Pipes</button>
<!-- Existing pipe toggle content -->

<!-- Nested child component -->
<app-course-list></app-course-list>

<router-outlet />
```

Exp5\src\app\app.ts:

```
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { CourseList } from './course-list/course-list';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet , CourseList],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('Exp2');
  showVar = false;
    show() {
    if(this.showVar==false){
      this.showVar=true;
    }
    else{
      this.showVar=false;
    }
  }
}
```

✅ **Final Output**



6a

Module Name: Passing data from Container Component to Child Component Create an AppComponent that displays a dropdown with a list of courses as values in it. Create another component called the CoursesList component and load it in AppComponent which should display the course details. When the user selects a course from the

Angular - Passing Data from Container Component to Child Component

Step 1: Create Angular Project

Run the following command in your terminal:

ng new SOCLab

This will create a new angular project with name SOCLab

Step 2: Generate Components

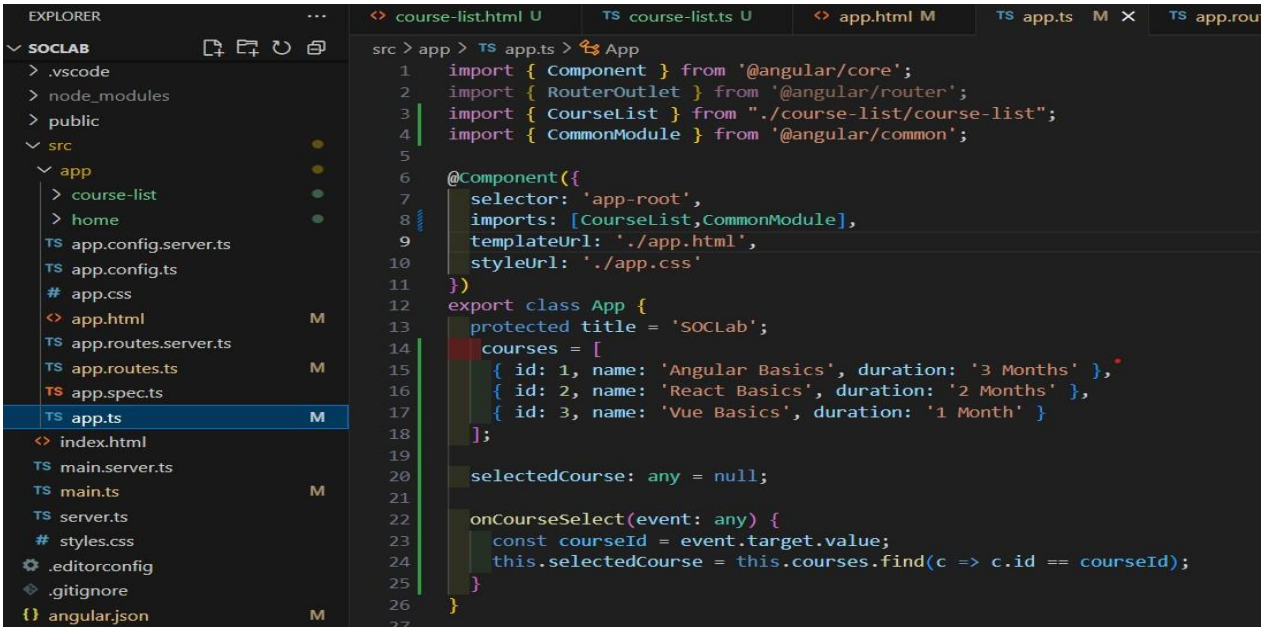
```
ng generate component app ng
generate component courses-
list
```

These are the two components need to achieve the requirements for the given question.

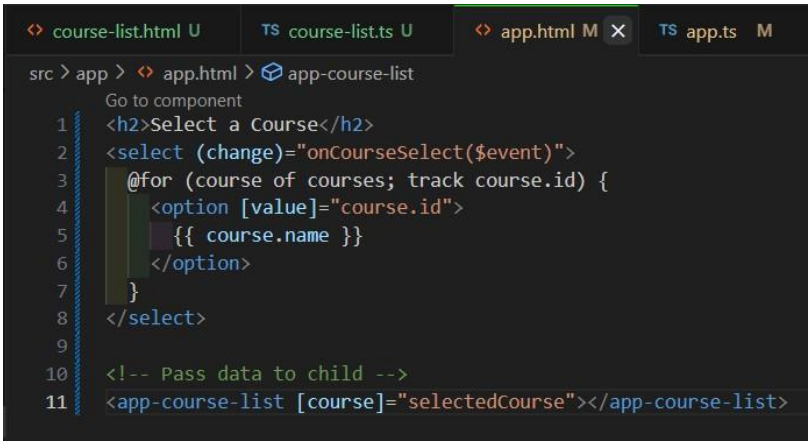
By these two commands two components are created with names **app** and **course-list**

Step 3: AppComponent (Parent Component)

Your Parent Component look like this(app.ts) file:

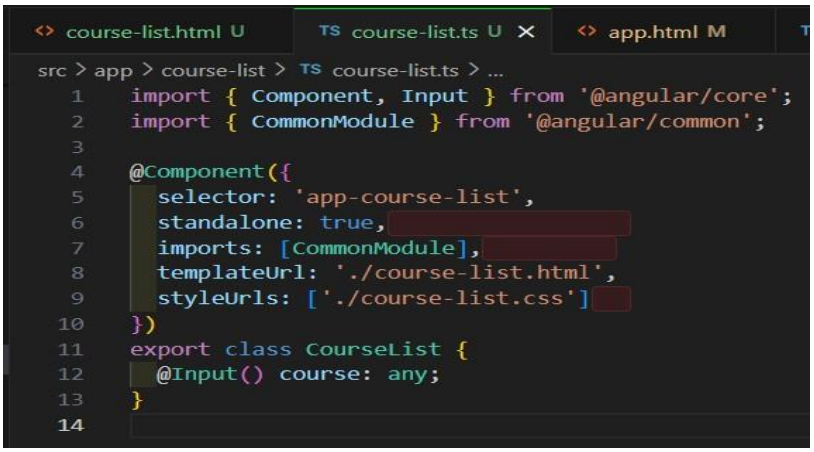


Your app.html file look like this:

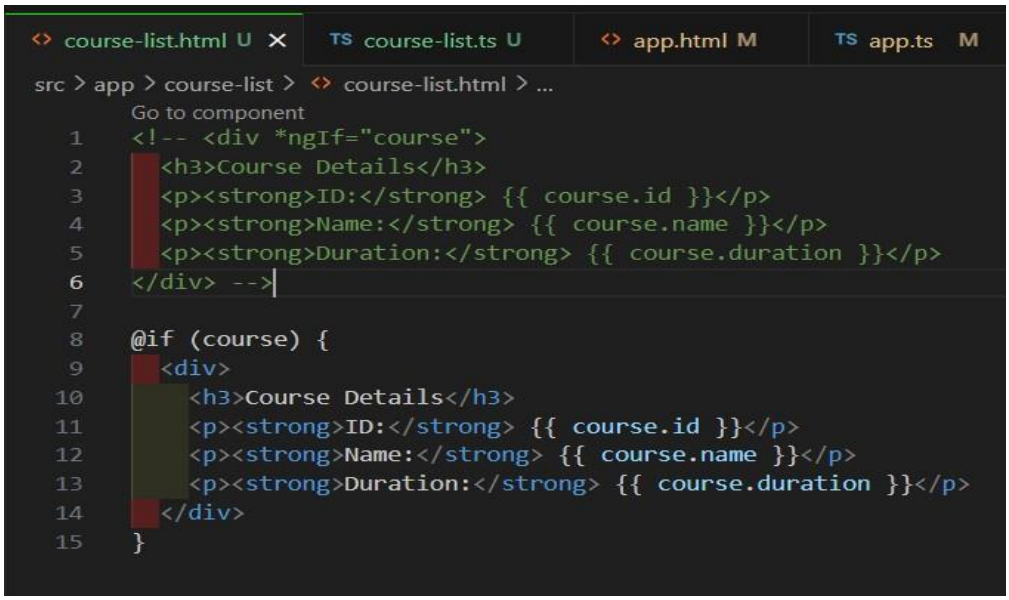


Step 4: CoursesListComponent (Child Component)

Your Child Component look like this(course-list.ts) file:



Your course-list.html file look like this:



Step 5: Make Changes in app.routes.ts file:

course-list.html U | course-list.ts U | app.html M | app.ts M | app.routes.ts M

src > app > app.routes.ts > ...

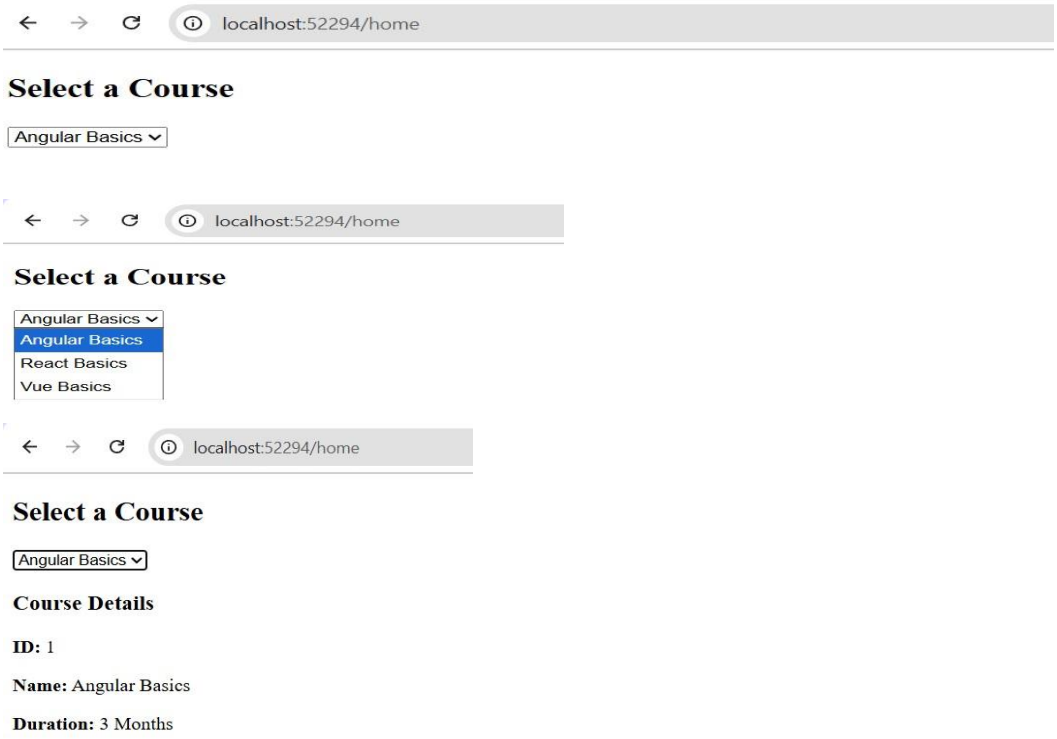
```ts
import { RouterModule, Routes } from '@angular/router';
import { App } from './app';
import { CourseList } from './course-list/course-list';
import { NgModule } from '@angular/core';
import { Home } from './home/home';

export const routes: Routes = [

{ path: '', redirectTo: '/home', pathMatch: 'full' }, // default route
  { path: 'home', component: App },
  { path: 'courses', component: CourseList },
  { path: '**', redirectTo: '/home' } // wildcard route

];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})

export class AppRoutingModule { }
```

Step 6: Run Application

The application will open in the browser. Select a course from the dropdown to see details displayed in the child component.

Output:

1. A dropdown list will appear with Angular, React, and Vue courses
2. When you select a course, its details (ID, Name, Duration) will be displayed below in the CoursesList component.



6-b

Module Name: Passing data from Child Component to ContainerComponent Create an AppComponent that loads another component called the CoursesListcomponent. Create another component called CoursesListComponent which should display the courses list in a table along with a register button in each row. When a userclicks on the register button, it should send that courseName value back toAppComponent where it should display the registration successful message along withcourseName.

To achieve data transfer from a child component (CoursesList Component) to a parent component (AppComponent) in Angular, the @Output () decorator along with EventEmitter is utilized.

1. CoursesList Component (Child)

Define courses data:

An array of course objects, each containing a name property.

Create courseRegistered event:

Declare an @Output () property named courseRegistered of type EventEmitter<string>. This emitter will send the selected course name to the parent.

Implement registerCourse method:

This method is triggered when the "Register" button is clicked. It takes the courseName as an argument and emits it using this.courseRegistered.emit (courseName).

Template (courses-list.component.html):

Iterate through the courses array to display each course in a table row. Include a "Register" button in each row, bound to the registerCourse method with the respective course.name.

COMPONENT

```ts
// courses-list.component.ts
import { Component, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-courses-list',
  templateUrl: './courses-list.component.html',
  styleUrls: ['./courses-list.component.css']
})
export class CoursesListComponent {
  courses = [
    { name: 'Angular Basics' },
    { name: 'React Fundamentals' },
    { name: 'Vue.js Essentials' },
    { name: 'Node.js Development' }
  ];

  @Output() courseRegistered = new EventEmitter<string>();

  registerCourse(courseName: string): void {
    this.courseRegistered.emit(courseName);
  }
}
```

CODE

```
<div>
  <table>
    <thead>
      <tr>
        <th>Action</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let course of courses">
        <td>{{ course.name }}</td>
        <td>
          <button (click)="registerCourse(course.name)">Register</button>
        </td>
      </tr>
    </tbody>
  </table>
</div>
```

2. **AppComponent (Parent)**

- Declare registeredCourseName: A property to store the name of the successfully registered course.
- Implement handleCourseRegistration method: This method is called when the courseRegistered event is emitted from the child. It receives the courseName and updates the registeredCourseName property.
- Template (app.component.html): Include the app-courses-list selector. Bind the courseRegistered output event to the handleCourseRegistration method using (courseRegistered)="handleCourseRegistration($event)". Display the registration successful message conditionally based on registeredCourseName.

COMPONENT

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

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

  handleCourseRegistration(courseName: string): void {
    this.registeredCourseName = courseName;
  }
}
```

CODE

```html
<!-- app.component.html -->
<div>
  <h1>Course Registration System</h1>
  <app-courses-list (courseRegistered)="handleCourseRegistration($event)"></app-courses-list>

  <div *ngIf="registeredCourseName">
    <p>Registration successful for: <strong>{{ registeredCourseName }}</strong>!</p>
  </div>
</div>
```

3. App Module (app. module.ts)

- Ensure both AppComponent and CoursesListComponent are declared in the declarations array of AppModule

```typescript
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { CoursesListComponent } from './courses-list/courses-list.component';

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

6-c

Module Name: Shadow DOM Apply ShadowDOM and None encapsulation modes to components

Applying Shadow DOM Encapsulation to Components

Shadow DOM encapsulation creates an isolated DOM subtree for a component, preventing styles and scripts from leaking in or out. This enhances component reusability and maintainability by ensuring self-contained units.

How to apply Shadow DOM:

- **Web Components (Native):** Use the attachShadow () method on an element to create a shadow root.

  ### TYPESCRIPT

```
const hostElement = document.getElementById('my-component');
const shadowRoot = hostElement.attachShadow({ mode: 'open' }); // or 'closed'
shadowRoot.innerHTML = `
    <style>
        /* Styles scoped to this shadow DOM */
        h2 { color: blue; }
    </style>
    <h2>Shadow DOM Content</h2>
`;
```

- **Frameworks (e.g., Angular):** Set the encapsulation property in the component decorator.

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

@Component({
    selector: 'app-shadow-component',
    template: `<h2>Shadow DOM Component</h2>`,
    styles: [`h2 { color: blue; }`],
    encapsulation: ViewEncapsulation.ShadowDom // Uses native Shadow DOM
})
export class ShadowComponent {}
```

Applying None Encapsulation to Components

None encapsulation (also known as "no encapsulation" or "global styles") means that a component's styles are not scoped and apply globally to the entire document. This can lead to style conflicts if not managed carefully.

How to apply None Encapsulation:

- **Web Components (Native):**

Styles are not placed within a shadow root and are instead added directly to the main document's <style> tags or linked stylesheets.

- **Frameworks (e.g., Angular):**

Set the encapsulation property in the component decorator to ViewEncapsulation.None.

TYPESCRIPT

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

@Component({
    selector: 'app-none-component',
    template: `<h2>None Encapsulation Component</h2>`,
    styles: [`h2 { color: red; }`],
    encapsulation: ViewEncapsulation.None // Styles apply globally
})
export class NoneComponent {}
```

Key Difference: Shadow DOM provides strong style and DOM isolation, while None encapsulation offers no isolation, making styles global. The choice depends on the desired level of encapsulation and potential for style conflicts within an application.

6-d

Module Name: Override component life-cycle hooks and logging the corresponding messages to understand the flow.

Component Lifecycle in Angular

In Angular, Components are the fundamental building blocks of an application. Understanding the lifecycle of these components is crucial for effective Angular Development. Angular provides several lifecycle hooks that allow developers to tap into key moments in a component's lifecycle and execute custom logic during those times.

Component Lifecycle Stages:

The component lifecycle in Angular consists of several stages:

Creation / Initialization:

- AngularJS compiles the template (template or template URL) and links it with the component's controller.
- $onInit () lifecycle hook is called once, after the bindings (@, =, <) are resolved. Used for component initialization (loading data, setting default values).

```
controller: function() {
  this.$onInit = function() {
    console.log("Component initialized");
  };
}
```

Change Detection / Updates:

- When parent scope values (inputs) change, AngularJS runs digest cycle.
- If the component has bindings with < or =, changes trigger $onChanges(changesObj).
- $doCheck () is called on every digest cycle (useful for custom change detection, but must be optimized).

```
this.$onChanges = function(changesObj) {
  console.log("Input changed:", changesObj);
};

this.$doCheck = function() {
  console.log("Digest cycle check");
};
```

Destruction / Cleanup:

- When the component is removed from the DOM, $onDestroy () is called.
- Use this for cleanup tasks like removing event listeners, cancelling intervals, etc.

```
this.$onDestroy = function() {
  console.log("Component destroyed");
};
```

Content Initialization:

- $postLink () is called after the component's template has been linked.
- This is where you can safely access/manipulate the DOM of your component.

```
this.$postLink = function() {
  console.log("DOM is ready for this component");
};
```

Lifecycle Hooks

Hook	When it runs
$onInit	After component is initialized, once bindings are available
$onChanges	When input bindings change
$doCheck	On every digest cycle (custom change detection)
$postLink	After DOM linking, safe to access/manipulate DOM
$onDestroy	Before component is destroyed, for cleanup

Override component life-cycle hooks and logging the corresponding messages to understand the flow

```
angular.module('myApp', [])
  .component('lifecycleDemo', {
    template: `
      <div>
        <h3>Lifecycle Demo Component</h3>
        <p>Check the console for lifecycle log messages.</p>
        <p>Input value: {{$ctrl.inputValue}}</p>
      </div>
    `,
    bindings: {
      inputValue: '<'
    },
    controller: function() {
      // Called when component is initialized
      this.$onInit = function() {
        console.log('$onInit: Component initialized');
      };

      // Called when input bindings change
      this.$onChanges = function(changesObj) {
        console.log('$onChanges: Input changed', changesObj);
      };
```

```
      // Called on every digest cycle (for custom change detection)
      this.$doCheck = function() {
        console.log('$doCheck: Digest cycle running');
      };

      // Called after the component's template has been linked (DOM ready)
      this.$postLink = function() {
        console.log('$postLink: Component DOM ready');
      };

      // Called before the component is destroyed
      this.$onDestroy = function() {
        console.log('$onDestroy: Component is about to be destroyed');
      };
    }
  });
```

How to Test

1. **Add this component to your HTML:**

```html
<div ng-app="myApp" ng-controller="MainCtrl as main">
  <lifecycle-demo input-value="main.value"></lifecycle-demo>
  <button ng-click="main.changeValue()">Change Value</button>
  <button ng-click="main.remove()">Remove Component</button>
</div>
```

2. **Define a controller to manage it:**

```javascript
angular.module('myApp')
  .controller('MainCtrl', function($scope) {
    var vm = this;
    vm.value = "Hello";

    vm.changeValue = function() {
      vm.value = "Updated at " + new Date().toLocaleTimeString();
    };

    vm.remove = function() {
      $scope.$destroy(); // destroys controller + component
    };
  });
```

What You'll See in the Console (Flow)

1. $onInit → when component initializes
2. $postLink → once DOM is ready
3. $doCheck → runs during each digest cycle
4. $onChanges → when input binding changes
5. $onDestroy → when the component is removed

Exercise-7a

Module Name: Template Driven Forms

To Create a course registration form as a template-driven form.

Description:

- The template-driven form is a type of Angular form that relies on the template for managing form state and validation. It is created directly in the HTML template using Angular directives.
- Template-driven forms use two-way data binding to update the data model in the component as changes are made in the template and vice versa. Template-driven forms allow you to use form-specific directives in your Angular template.
- These forms are simpler to set up and hence, mainly used for creating a simple and less complex form application.

Let's understand how to create and use template driven forms in an Angular application.

Prerequisites :

- Angular CLI
- Basics of HTML, CSS and JavaScript
- NodeJS and NPM

Approach:

To create a Template Driven Form, we need to create an Angular Application.

- In the newly created HTML file, create a form using an angular template-driven approach. Use Angular's form directives 'ngModel' to bind form fields.
- Add form controls such as 'input' fields. Use Validations Directives such as required, maxlength, minlength, etc.
- Open the ts file and define properties according to form fields.

- A form will be submitted only when all the required fields are filled.

Steps to Create Template-Driven Form in Angular

Step 1: Install Angular CLI

- If you haven't installed Angular CLI yet, install it using the following command:

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

Step 2: Create a New Angular Project

```
ng new my-angular-app --no-standalone
cd my-angular-app
```

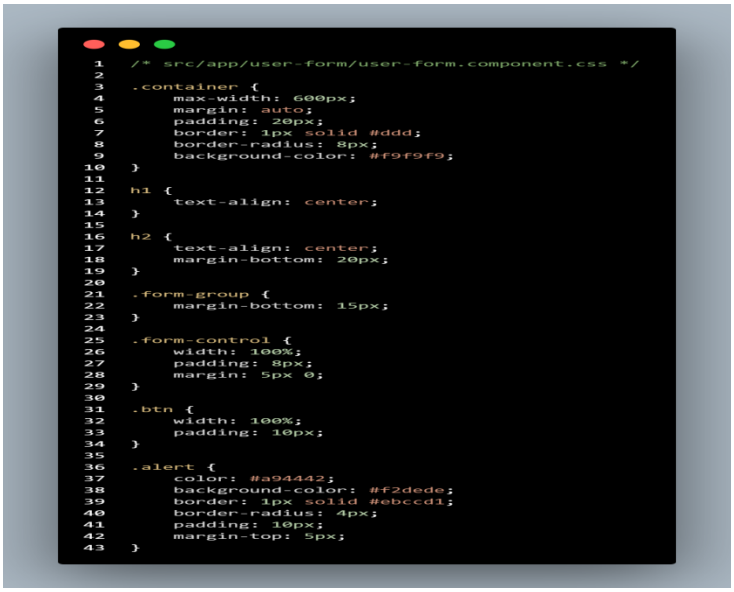
Step 3: Create a Component

```
ng generate component user-form
```

User-form Component

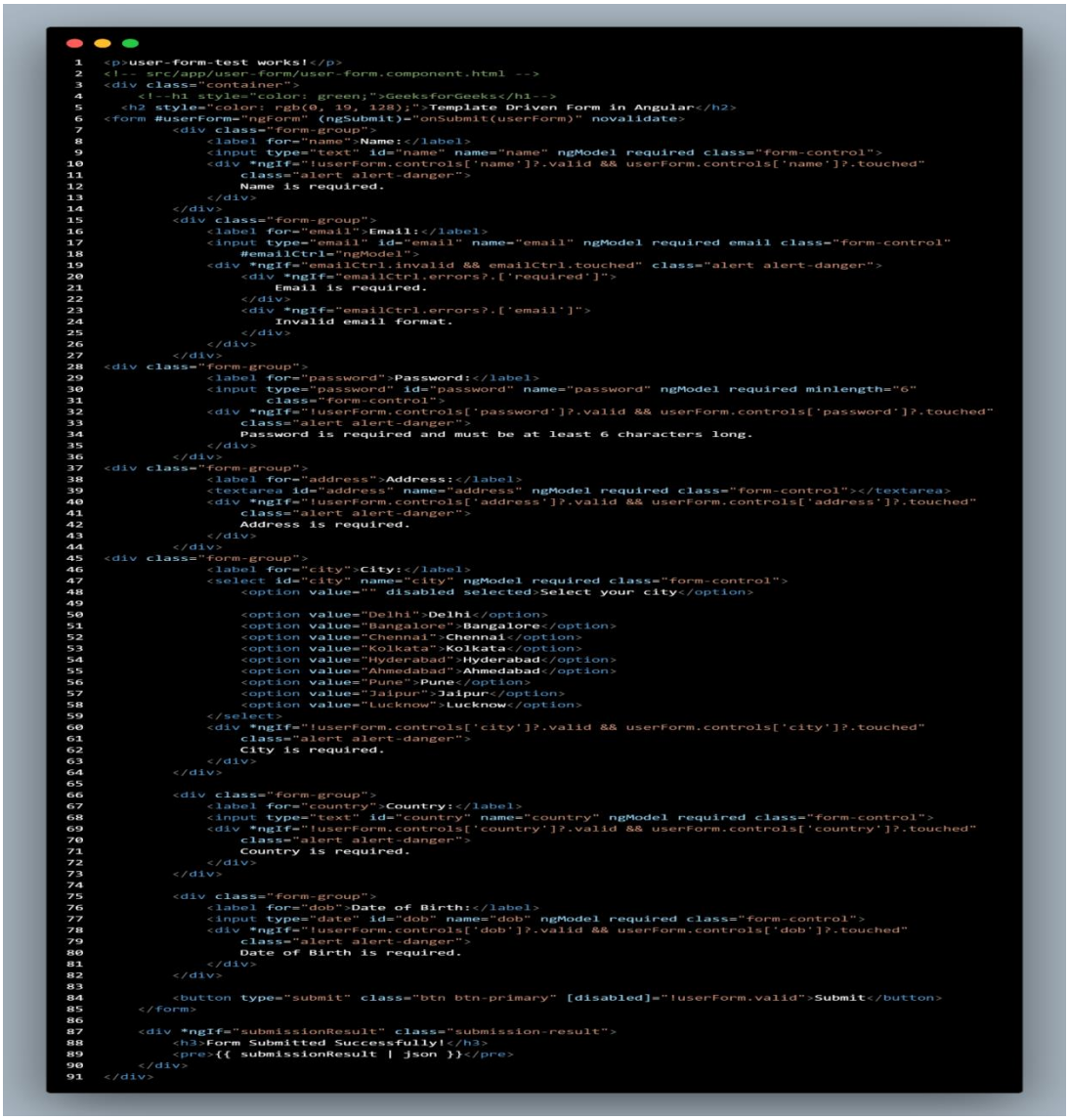
The below mentioned is a Component which is created to make a Template Driven Form.

/* src/app/user-form/user-form.component.css */



```
/* src/app/user-form/user-form.component.css */

.container {
    max-width: 600px;
    margin: auto;
    padding: 20px;
    border: 1px solid #ddd;
    border-radius: 8px;
    background-color: #f9f9f9;
}

h1 {
    text-align: center;
}

h2 {
    text-align: center;
    margin-bottom: 20px;
}

.form-group {
    margin-bottom: 15px;
}

.form-control {
    width: 100%;
    padding: 8px;
    margin: 5px 0;
}

.btn {
    width: 100%;
    padding: 10px;
}

.alert {
    color: #a94442;
    background-color: #f2dede;
    border: 1px solid #ebccd1;
    border-radius: 4px;
    padding: 10px;
    margin-top: 5px;
}
```

<!-- src/app/user-form/user-form.component.html -->



```html
<p>user-form-test works!</p>
<!-- src/app/user-form/user-form.component.html -->
<div class="container">
    <!--h1 style="color: green;">GeeksforGeeks</h1-->
  <h2 style="color: rgb(0, 19, 128);">Template Driven Form in Angular</h2>
<form #userForm="ngForm" (ngSubmit)="onSubmit(userForm)" novalidate>
        <div class="form-group">
            <label for="name">Name:</label>
            <input type="text" id="name" name="name" ngModel required class="form-control">
            <div *ngIf="!userForm.controls['name']?.valid && userForm.controls['name']?.touched"
                class="alert alert-danger">
                Name is required.
            </div>
        </div>
        <div class="form-group">
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" ngModel required email class="form-control"
                #emailCtrl="ngModel">
            <div *ngIf="emailCtrl.invalid && emailCtrl.touched" class="alert alert-danger">
                <div *ngIf="emailCtrl.errors?.['required']">
                    Email is required.
                </div>
                <div *ngIf="emailCtrl.errors?.['email']">
                    Invalid email format.
                </div>
            </div>
        </div>
<div class="form-group">
            <label for="password">Password:</label>
            <input type="password" id="password" name="password" ngModel required minlength="6"
                class="form-control">
            <div *ngIf="!userForm.controls['password']?.valid && userForm.controls['password']?.touched"
                class="alert alert-danger">
                Password is required and must be at least 6 characters long.
            </div>
        </div>
<div class="form-group">
            <label for="address">Address:</label>
            <textarea id="address" name="address" ngModel required class="form-control"></textarea>
            <div *ngIf="!userForm.controls['address']?.valid && userForm.controls['address']?.touched"
                class="alert alert-danger">
                Address is required.
            </div>
        </div>
<div class="form-group">
            <label for="city">City:</label>
            <select id="city" name="city" ngModel required class="form-control">
                <option value="" disabled selected>Select your city</option>

                <option value="Delhi">Delhi</option>
                <option value="Bangalore">Bangalore</option>
                <option value="Chennai">Chennai</option>
                <option value="Kolkata">Kolkata</option>
                <option value="Hyderabad">Hyderabad</option>
                <option value="Ahmedabad">Ahmedabad</option>
                <option value="Pune">Pune</option>
                <option value="Jaipur">Jaipur</option>
                <option value="Lucknow">Lucknow</option>
            </select>
            <div *ngIf="!userForm.controls['city']?.valid && userForm.controls['city']?.touched"
                class="alert alert-danger">
                City is required.
            </div>
        </div>

        <div class="form-group">
            <label for="country">Country:</label>
            <input type="text" id="country" name="country" ngModel required class="form-control">
            <div *ngIf="!userForm.controls['country']?.valid && userForm.controls['country']?.touched"
                class="alert alert-danger">
                Country is required.
            </div>
        </div>

        <div class="form-group">
            <label for="dob">Date of Birth:</label>
            <input type="date" id="dob" name="dob" ngModel required class="form-control">
            <div *ngIf="!userForm.controls['dob']?.valid && userForm.controls['dob']?.touched"
                class="alert alert-danger">
                Date of Birth is required.
            </div>
        </div>

        <button type="submit" class="btn btn-primary" [disabled]="!userForm.valid">Submit</button>
    </form>

    <div *ngIf="submissionResult" class="submission-result">
        <h3>Form Submitted Successfully!</h3>
        <pre>{{ submissionResult | json }}</pre>
    </div>
</div>
```

// src/app/user-form/user-form.component.ts

```typescript
import { Component } from '@angular/core';
import { CommonModule, JsonPipe } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-user-form',
  standalone: true,
  imports: [CommonModule, FormsModule, JsonPipe],
  templateUrl: './user-form-test.component.html',
  styleUrls: ['./user-form-test.component.css']
})
export class UserFormComponent {
  submissionResult: any;

  onSubmit(form: any) {
    if (form.valid) {
      this.submissionResult = form.value;
      console.log('Form submitted', this.submissionResult);
    }
  }
}
```

App Component:

The below mentioned is the App Component which is the first component which gets loaded when an Angular Application is started. It is having an HTML file and app.module.ts file which is used for handling all the imports in a non-standalone component.

<!-- src/app/app.component.html -->

<app-user-form-test></app-user-form-test>

// src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { UserFormComponent } from './user-form/user-form-test.component';

@NgModule({
  declarations: [], // no declarations needed for standalone components
  imports: [
    BrowserModule,
    AppComponent,      // standalone root component
    UserFormComponent  // standalone user form component
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Steps to Run the Application

Open the terminal, run this command from your root directory to start the application.

npx ng serve

Exercise-7(b)

Module Name: Model Driven Forms or Reactive Forms

To create an employee registration form using **Reactive Forms** in Angular.

1. Introduction

Reactive Forms (also called **Model-Driven Forms**) in Angular allow you to create forms where **form controls are defined in the component class** instead of the template. This approach is more powerful and scalable for complex forms.

Advantages of Reactive Forms:

- More predictable and robust.
- Easier to test and maintain.
- Provides synchronous access to form data.
- Enables easy validation and dynamic changes.

2. Steps to Create Employee Registration Form

1. Create Angular Project

```
ng new EmployeeApp
cd EmployeeApp
ng serve
```

2. Generate Employee Registration Component

```
ng generate component RegistrationForm
```

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

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ReactiveFormsModule, FormBuilder, Validators } from '@angular/forms';

@Component({
  selector: 'app-registration-form',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './registration-form.html',
  styleUrls: ['./registration-form.css']
})
```

```
export class RegistrationFormComponent {
  submitted = false;
  registerForm;

  constructor(private fb: FormBuilder) {
    this.registerForm = this.fb.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      address: this.fb.group({
        street: [''],
        zip: [''],
        city: ['']
      })
    });
  }

  onSubmit() {
    this.submitted = true;
    console.log(this.registerForm.value);
  }
}
```

4.Write the below-given code in **registration-form.html**

```
<div class="container">
  <h2 class="text-center mb-4">Registration Form</h2>

  <form [formGroup]="registerForm" (ngSubmit)="submitted = true">

    <!-- First Name -->
    <div class="form-group">
      <label>First Name</label>
      <input type="text" class="form-control" formControlName="firstName">
      <div *ngIf="registerForm.controls['firstName'].invalid &&
                  (registerForm.controls['firstName'].touched || submitted)"
           class="alert alert-danger">
        First name field is invalid.
        <p *ngIf="registerForm.controls['firstName'].errors?.['required']">
          This field is required!
        </p>
      </div>
    </div>
```

```
    <!-- Last Name -->                    Saved memory full ⓘ
    <div class="form-group">
      <label>Last Name</label>                                          Copy code
      <input type="text" class="form-control" formControlName="lastName">
      <div *ngIf="registerForm.controls['lastName'].invalid &&
                  (registerForm.controls['lastName'].touched || submitted)"
           class="alert alert-danger">
        Last name field is invalid.
        <p *ngIf="registerForm.controls['lastName'].errors?.['required']">
          This field is required!
        </p>
      </div>
    </div>

    <!-- Address -->
    <div class="form-group">
      <fieldset formGroupName="address">
        <legend>Address:</legend>

        <label>Street</label>
        <input type="text" class="form-control" formControlName="street">

        <label>Zip</label>
        <input type="text" class="form-control" formControlName="zip">

        <label>City</label>
        <input type="text" class="form-control" formControlName="city">
      </fieldset>
    </div>
```

```
    <!-- Submit Button -->                                    Copy code
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>

  <br />

  <!-- Display submitted details -->
  <div [hidden]="!submitted">
    <h3>Employee Details</h3>
    <p>First Name: {{ registerForm.get('firstName')?.value }}</p>
    <p>Last Name: {{ registerForm.get('lastName')?.value }}</p>
    <p>Street: {{ registerForm.get('address.street')?.value }}</p>
    <p>Zip: {{ registerForm.get('address.zip')?.value }}</p>
    <p>City: {{ registerForm.get('address.city')?.value }}</p>
  </div>
</div>
```

5. Write the below-given code in **registration-form.css**

Saved memory full

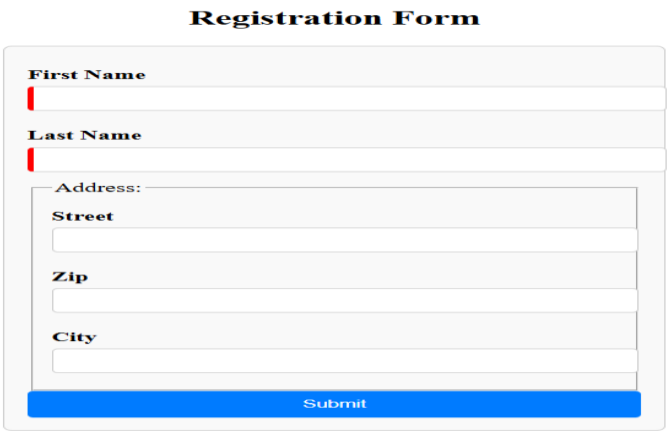
Copy code

```css
form {
  max-width: 400px;
  margin: 20px auto;
  padding: 15px;
  border: 1px solid #ccc;
  border-radius: 6px;
  background: #f9f9f9;
}

label {
  font-weight: bold;
  display: block;
  margin-top: 10px;
}

input {
  width: 100%;
  padding: 6px;
  margin: 5px 0 10px;
  border: 1px solid #ccc;
  border-radius: 4px;
}

.ng-valid[required] {
  border-left: 4px solid green;
}

.ng-invalid:not(form) {
  border-left: 4px solid red;
}

button {
  width: 100%;
  padding: 8px;
  background: #007bff;
  color: #fff;
  border: none;
  border-radius: 4px;
}
```

6. Write the below-given code in **app.html**

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

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



Exercise-7(c)

Module Name: Custom Validators in Reactive Forms

To create a custom validator for an email field in the employee registration form(reactive form)

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.

Registration-form.ts:

src > app > registration-form > TS registration-form.ts > RegistrationFormComponent

```ts
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  ReactiveFormsModule,
  FormBuilder,
  Validators,
  AbstractControl,
  ValidationErrors
} from '@angular/forms';

// Custom Email Validator
function customEmailValidator(control: AbstractControl): ValidationErrors | null {
  const value = control.value;
  if (!value) return null; // skip empty, required will catch it
  const emailPattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-z]{2,4}$/;
  return emailPattern.test(value) ? null : { invalidEmail: true };
}

@Component({
  selector: 'app-registration-form',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './registration-form.html',
  styleUrls: ['./registration-form.css']
})
export class RegistrationFormComponent {
  submitted = false;
  registerForm;
```

src > app > registration-form > TS registration-form.ts > RegistrationFormComponent

```ts
export class RegistrationFormComponent {
  registerForm;

  constructor(private fb: FormBuilder) {
    this.registerForm = this.fb.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      email: ['', [Validators.required, customEmailValidator]], // ✅ added email
      address: this.fb.group({
        street: [''],
        zip: [''],
        city: ['']
      })
    });
  }

  onSubmit() {
    this.submitted = true;
    console.log(this.registerForm.value);
  }
}
```

Registration-form.html:

src > app > registration-form > <> registration-form.html > div.container

```html
<div class="container">
  <h2 style="text-align:center; margin-bottom:20px;">Registration Form</h2>

  <form [formGroup]="registerForm" (ngSubmit)="onSubmit()">

    <!-- First Name -->
    <div class="form-group">
      <label>First Name</label>
      <input type="text" class="form-control" formControlName="firstName">
      <div *ngIf="registerForm.controls['firstName'].invalid &&
                  (registerForm.controls['firstName'].touched || submitted)"
           class="alert alert-danger">
        First name field is invalid.
        <p *ngIf="registerForm.controls['firstName'].errors?.['required']">
          This field is required!
        </p>
      </div>
    </div>

    <!-- Last Name -->
    <div class="form-group">
      <label>Last Name</label>
      <input type="text" class="form-control" formControlName="lastName">
      <div *ngIf="registerForm.controls['lastName'].invalid &&
                  (registerForm.controls['lastName'].touched || submitted)"
           class="alert alert-danger">
        Last name field is invalid.
        <p *ngIf="registerForm.controls['lastName'].errors?.['required']">
          This field is required!
```

```
      </p>
    </div>
  </div>

  <!-- Email -->
  <div class="form-group">
    <label>Email</label>
    <input type="email" class="form-control" formControlName="email">
    <div *ngIf="registerForm.controls['email'].invalid &&
        (registerForm.controls['email'].touched || submitted)"
      class="alert alert-danger">
      <p *ngIf="registerForm.controls['email'].errors?.['required']">
      Email is required!
      </p>
      <p *ngIf="registerForm.controls['email'].errors?.['invalidEmail']">
      Invalid email format!
      </p>
    </div>
  </div>

  <!-- Address -->
  <div class="form-group">
    <fieldset formGroupName="address">
      <legend>Address:</legend>
```

```
        <label>Street</label>
        <input type="text" class="form-control" formControlName="street">

        <label>Zip</label>
        <input type="text" class="form-control" formControlName="zip">

        <label>City</label>
        <input type="text" class="form-control" formControlName="city">
      </fieldset>
    </div>

    <!-- Submit Button -->
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>

  <br />

  <!-- Display submitted details -->
  <div [hidden]="!submitted">
    <h3>Employee Details</h3>
    <p>First Name: {{ registerForm.get('firstName')?.value }}</p>
    <p>Last Name: {{ registerForm.get('lastName')?.value }}</p>
    <p>Email: {{ registerForm.get('email')?.value }}</p>
    <p>Street: {{ registerForm.get('address.street')?.value }}</p>
```

```
  <br />

  <!-- Display submitted details -->
  <div [hidden]="!submitted">
    <h3>Employee Details</h3>
    <p>First Name: {{ registerForm.get('firstName')?.value }}</p>
    <p>Last Name: {{ registerForm.get('lastName')?.value }}</p>
    <p>Email: {{ registerForm.get('email')?.value }}</p>
    <p>Street: {{ registerForm.get('address.street')?.value }}</p>
    <p>Zip: {{ registerForm.get('address.zip')?.value }}</p>
    <p>City: {{ registerForm.get('address.city')?.value }}</p>
  </div>
</div>
```

Output:


Registration Form
First Name
Last Name
Email
Address:
Street
Zip
City
Submit

EXERCISE 8

8a. Custom Validators in Template Driven forms

Create a custom validator for the email field in the course registration form.

Custom validators:

A **custom validator** in Angular is a function or directive that you create to apply **your own validation rules** to form controls. It's used when the built-in validators like required, minLength, or email are not enough.

A validator checks the value of a form control and returns:

- null → **valid** (passes the validation)
- error → **invalid** (fails the validation)

Types of Custom Validators

1. For Reactive Forms → you define a **function**.
2. For Template-Driven Forms → you create a **directive**

Directive:

A directive is a class that adds behaviour to elements in your Angular applications. It's a way to **manipulate the DOM or attach logic** to HTML elements without modifying their actual markup.

Steps to Create a Custom Validator Directive:

1. **Generate the directive:**

   ng generate directive usernamevalidator

2. **Import required modules:**

3. Create the directive:

course-form.component.html | course-form.component.css | app.module.ts | import { Directive } from '@ar

```
import { Directive } from '@angular/core';
import { NG_VALIDATORS, Validator, AbstractControl, ValidationErrors } from '@angular/forms';
@Directive({
  selector: '[appForbiddenWord]', // This will be used in the template
  providers: [
    { provide: NG_VALIDATORS, useExisting: ForbiddenWordValidatorDirective, multi: true }
  ]
})
export class ForbiddenWordValidatorDirective implements Validator {
  validate(control: AbstractControl): ValidationErrors | null {
    const forbiddenWord = 'test';
    if (control.value && control.value.toLowerCase().includes(forbiddenWord)) {
      return { forbiddenWord: true }; // validation fails
    }
    return null; // validation passes
  }
}
```

4. Using the Validator in Template:

```
<form #myForm="ngForm">
  <input name="username" ngModel appForbiddenWord />
  <div *ngIf="myForm.controls.username?.errors?.forbiddenWord">
    The word "test" is not allowed.
  </div>
</form>
```

Problem Statement:

Create a custom validator for the email field in the course registration form.

Step 1: Create the file using the below command:

ng new <project-name>

Step 2: Navigate to the project directory:

cd <project name>

Step 3: Generate a component for the form:

ng generate component course-form

```
PS C:\Users\Niharika\OneDrive\Desktop\angular projects> cd course-registration
PS C:\Users\Niharika\OneDrive\Desktop\angular projects\course-registration> ng generate component course-form
>>
CREATE src/app/course-form/course-form.spec.ts (580 bytes)
CREATE src/app/course-form/course-form.ts (215 bytes)
CREATE src/app/course-form/course-form.css (0 bytes)
CREATE src/app/course-form/course-form.html (27 bytes)
PS C:\Users\Niharika\OneDrive\Desktop\angular projects\course-registration>
```

Step 4: Source code for created files:

course-form.component.ts:

TS course-form.ts | course-form.html | index.html

course-registration > src > app > course-form > TS course-form.ts > ...

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

@Component({
  selector: 'app-course-form',
  templateUrl: './course-form.component.html'
})
export class CourseFormComponent {

  onSubmit(form: NgForm) {
    if(form.valid) {
      console.log('Form Submitted!', form.value);
    }
  }

}
```

course-form.component.html(Template driven form):

course-form.ts | course-form.html | index.html

urse-registration > src > app > course-form > course-form.html > html

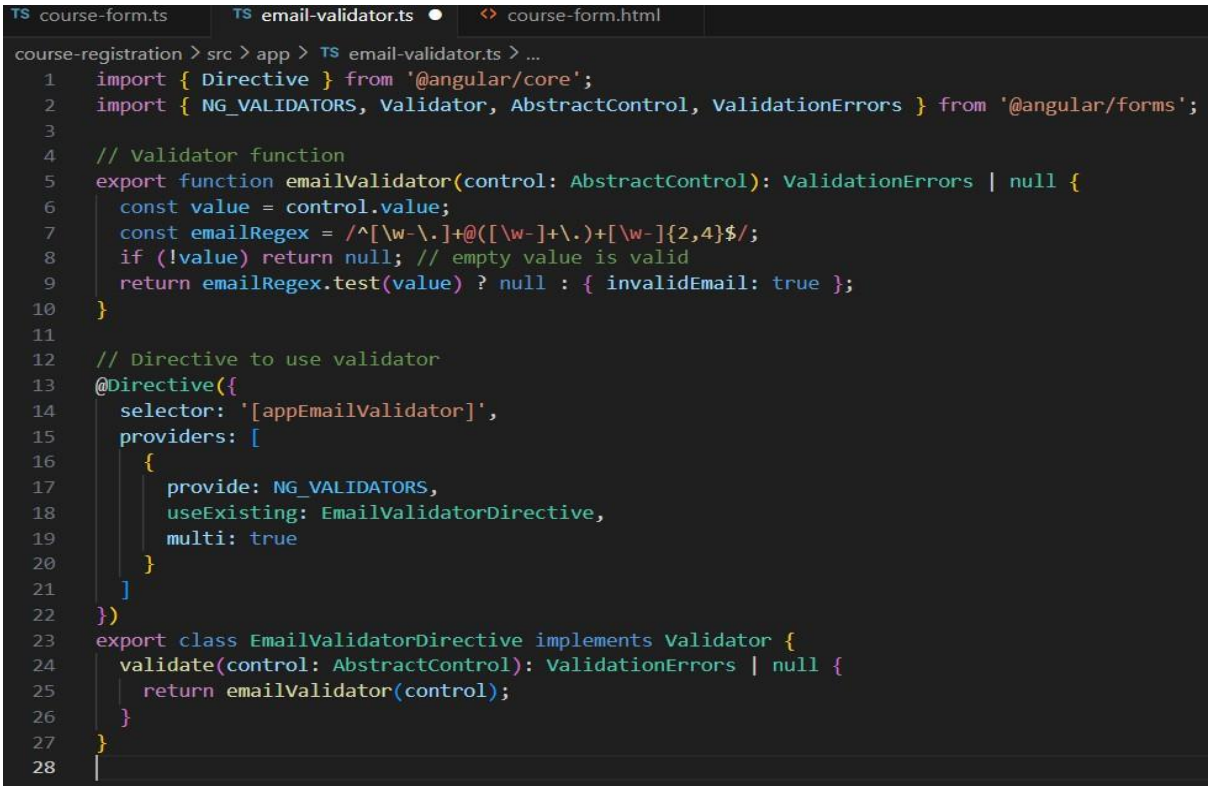
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CourseRegistration</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
<p>course-form works!</p>
<form #courseForm="ngForm" (ngSubmit)="onSubmit(courseForm)">
  <div>
    <label>Email:</label>
    <input type="email" name="email" ngModel appEmailValidator #emailCtrl="ngModel" required>
  </div>

  <div *ngIf="emailCtrl.errors?.invalidEmail && emailCtrl.touched">
    Invalid email address.
  </div>

  <button type="submit" [disabled]="courseForm.invalid">Submit</button>
</form>
</body>
</html>
```

Step 5: Create a Custom Email Validator Directive:

ng generate directive <name>

```
PS C:\Users\Niharika\OneDrive\Desktop\angular projects> cd course-registration
PS C:\Users\Niharika\OneDrive\Desktop\angular projects\course-registration> ng generate directive validators/e
>>
CREATE src/app/validators/email.spec.ts (191 bytes)
CREATE src/app/validators/email.ts (140 bytes)
PS C:\Users\Niharika\OneDrive\Desktop\angular projects\course-registration>
```

email.directive.ts:



Step 6: Add Directive to Module

app.module.ts:

Step 7: Add Component to App:

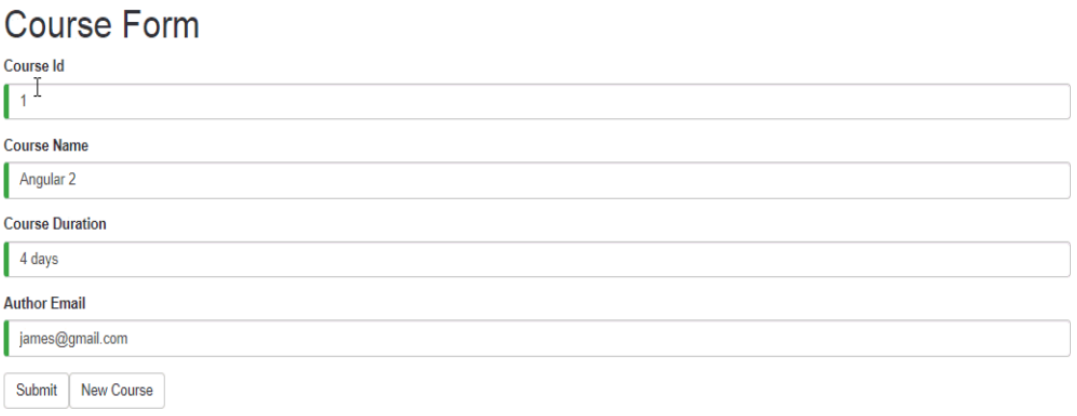
app.component.html:

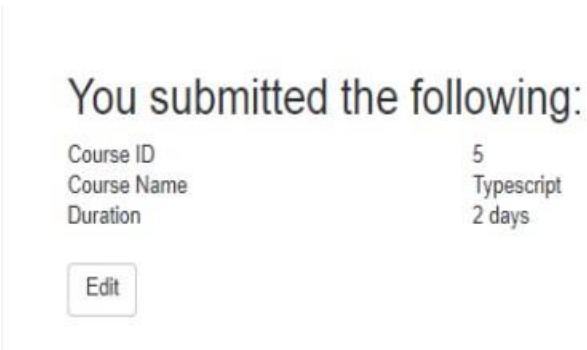
<app-course-form></app-course-form>

Step 8: Run the Application:

ng serve

OUTPUT:





8b.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.These are wired together using dependency injection. Angular provides a few inbuilt services and custom services that can be created.

Services can be used to:

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

Creating a Service

- Command to create a service: **ng generate service book**
- The above command will create a service class as shown below:

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

Providing a Service

- To register service is to specify providedIn property using @Injectable decorator.

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

@Injectable({

   providedIn: 'root'

})

export class BookService {}
```

Injecting a Service

- Inject a service into a component or any other class through constructor.

constructor(private bookService: BookService){ }

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

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.

To create a book component which fetches the book details like id,name and display them on the page in a list format by following below steps:

Step 1: Create a file using the following command:

ng new my-angular-app

Step 2: Change directory by using command :

cd my-angular-app

Step 3: Command to create a component named book:

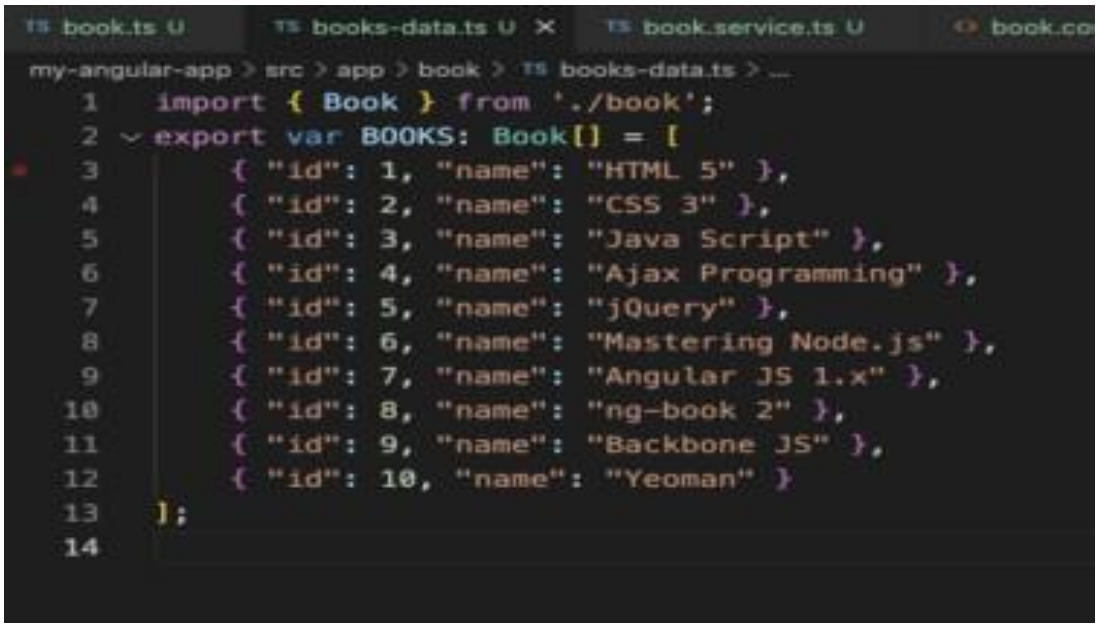
ng create component book

The above command creates the following files:

1. book.component.css
2. book.component.html
3. book.component.ts
4. book.component.spec.ts

Step 4: Create a file **book.ts** with the below code

Step 5: Create a file **books-data.ts** with the below code



Step 6: Command to create a service named book:

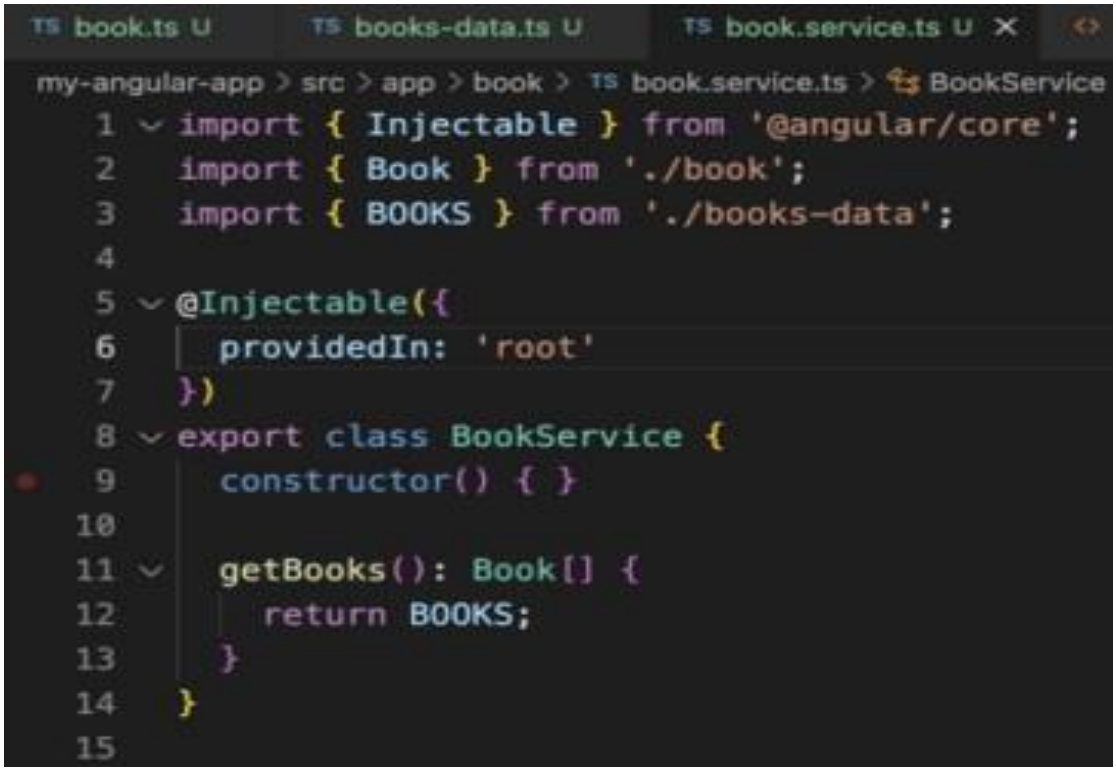
ng generate service book

This command will generate the following files

1.book.service.specs.ts

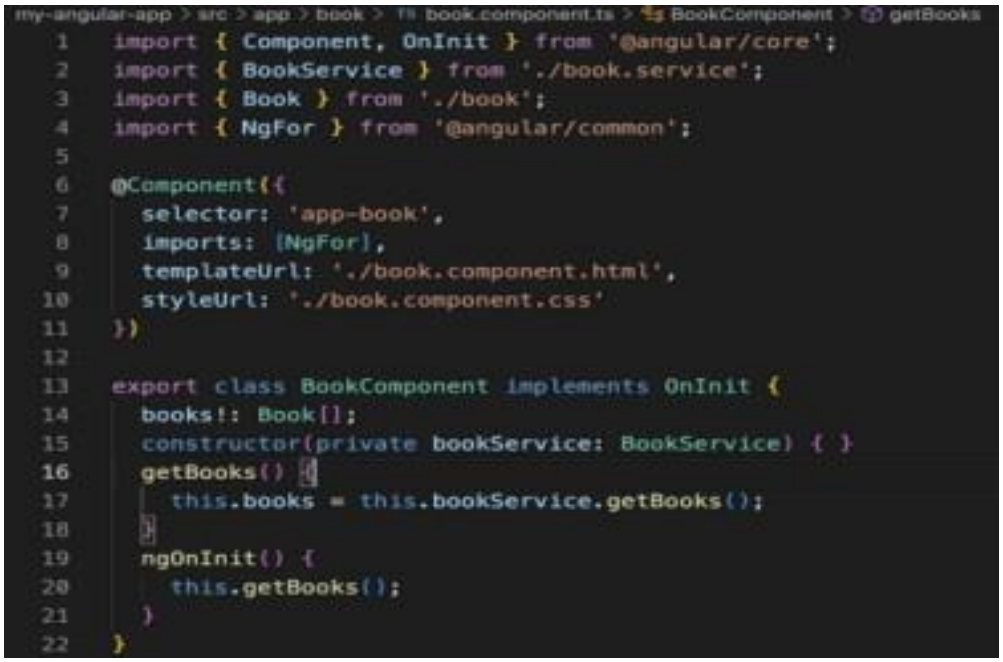
2.book.service.ts

Step 7: book.service.ts



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

Step 8: book.component.ts



- Imports BookService, Book class into a component
- Inject BookService class using a constructor
- Invokes getBooks() method from BookService class which in turn returns the books data.

Step 9: book.component.html

my-angular-app > src > app > book > book.component.html > ...

```
<p>book works!</p>
<h2>My Books</h2>
<ul class="books">
  <li *ngFor="let book of books">
    <span class="badge">{{book.id}}</span> {{book.name}}
  </li>
</ul>
```

- ngFor iterates on books array and displays book id and name on the page.

Step 10: book.component.css

app > src > app > book > # book.component.css > .books .badge

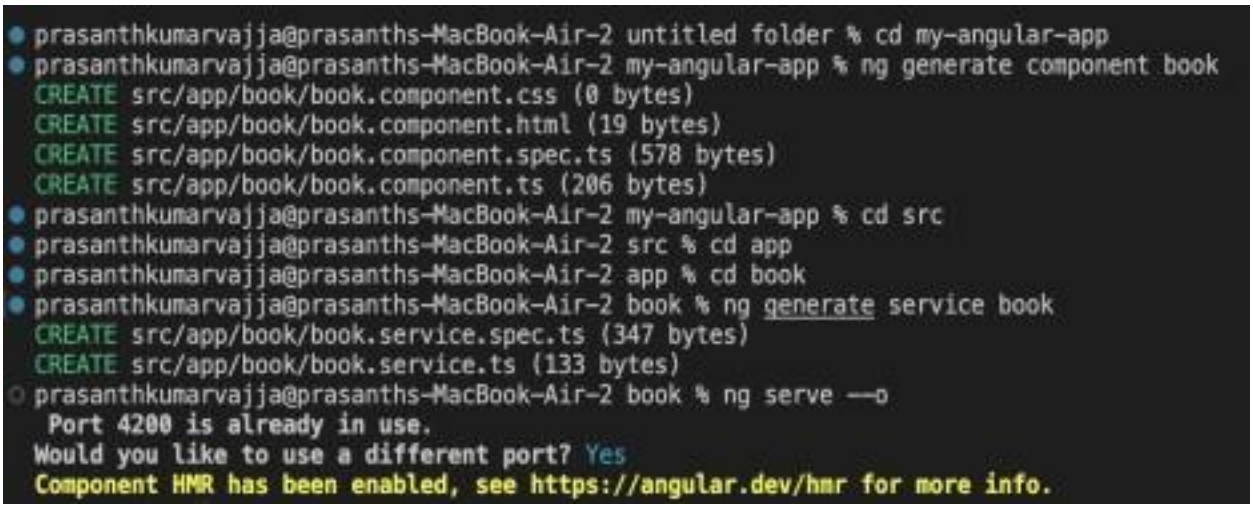
```
.books {
    margin: 0 0 2em 0;
    list-style-type: none;
    padding: 0;
    width: 15em;
}
.books li {
    cursor: pointer;
    position: relative;
    left: 0;
    background-color: #EEE;
    margin: .5em;
    padding: .3em 0;
    height: 1.6em;
    border-radius: 4px;
}
.books li:hover {
    color: #607D8B;
    background-color: #DDD;
    left: .1em;
}
.books .badge {
    display: inline-block;
    font-size: small;
    color: white;
    padding: 0.8em 0.7em 0 0.7em;
    background-color: #607D8B;
    line-height: 1em;
    position: relative;
    left: -1px;
    top: -4px;
    height: 1.8em;
    margin-right: .8em;
    border-radius: 4px 0 0 4px;
```

Step 11: app.component.ts

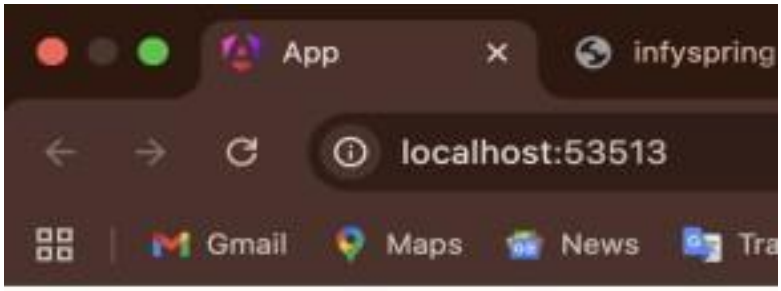
Step 12: app.component.html

<router-outlet></router-outlet>
<app-book></app-book>

Step 13: ng serve –o builds and serves the Angular app , opens it in default browser.



OUTPUT:



book works!

My Books

1. HTML 5
2. CSS 3
3. Java Script
4. Ajax Programming
5. jQuery
6. Mastering Node.js
7. Angular JS 1.x
8. ng-book 2
9. Backbone JS
10. Yeoman

8c.RxJS OBSERVABLES

Introduction to RxJS

Reactive Extensions for JavaScript (RxJS) is a third-party library widely used in Angular applications. It provides tools for working with asynchronous streams of data using the Observable pattern.

- **Observable**: Represents a data stream that can emit multiple values over time (unlike Promises, which resolve only once).
- **Observer:** Consumes the data stream emitted by an Observable through methods like next(),error() and complete().
- **Subscription:** A link between the Observable and the Observer that can be cancelled when data is no longer needed.

Why RxJS Observables?

Angular recommends using Observables for asynchronous operations due to several benefits:

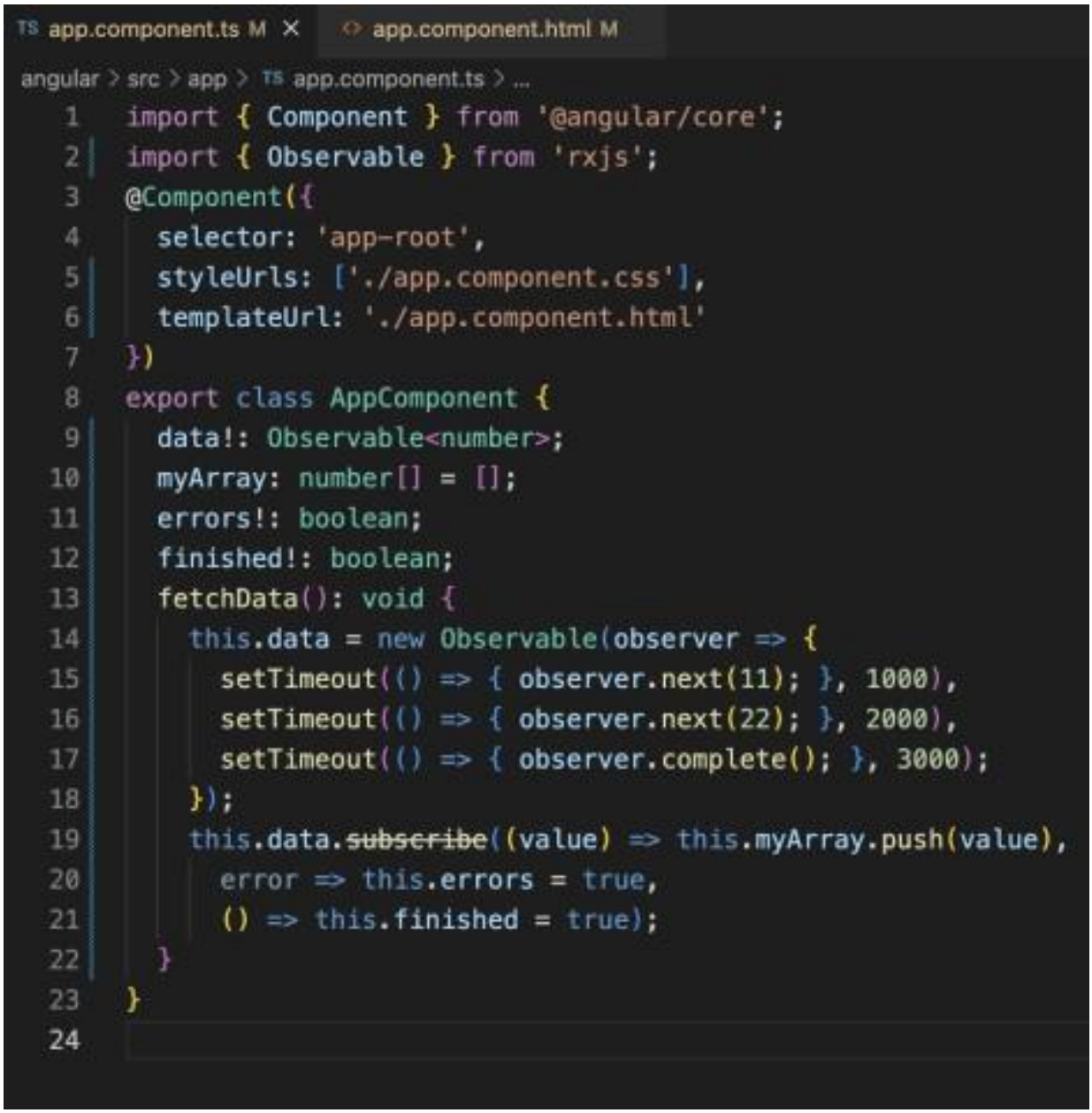
1. **Multiple Emissions**
    - Promises emit a **single value**, while Observables can emit **many values** over time.
2. **Cancellability**
    - Subscriptions to Observables can be cancelled, stopping data emission. Promises, once started, cannot be cancelled.
3. **Functional Operators**
    - Observables support operators like map,filter ,reduce and many more, making data handling flexible and powerful.

Problem statement :

Create and use an observable in Angular

Step 1: Follow first three steps of 8b

Step 2: app.component.ts



Explanation :

Line 2: Imports Observable from RxJS

Line 9: data is an Observable of numbers.

Line 13: fetchData() runs on button click.

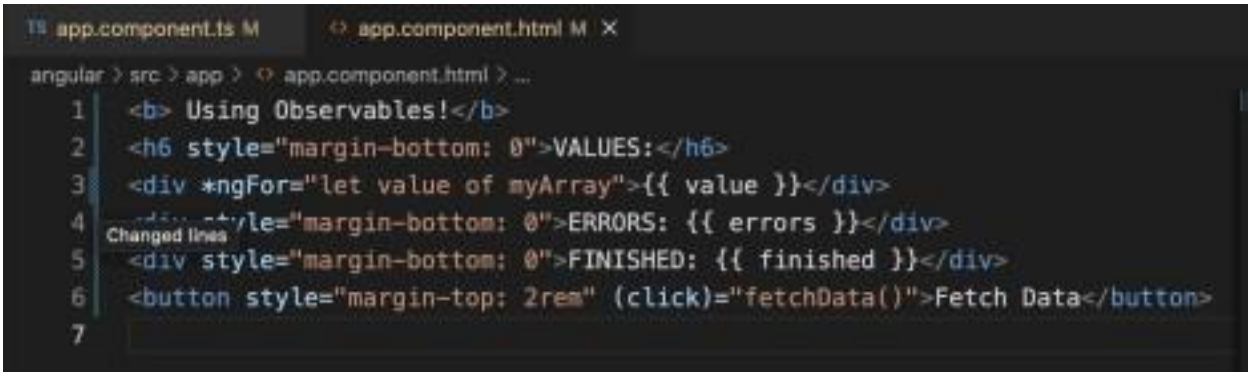
Line 14: Creates a new Observable in data.

Lines 15–17: next() emits values with delays; complete() ends stream.

Line 19: subscribe() listens with success, error, and complete callbacks.

On success, values are pushed to myArray; on error, error becomes true; and on complete, finished becomes true.

Step 3: app.component.html



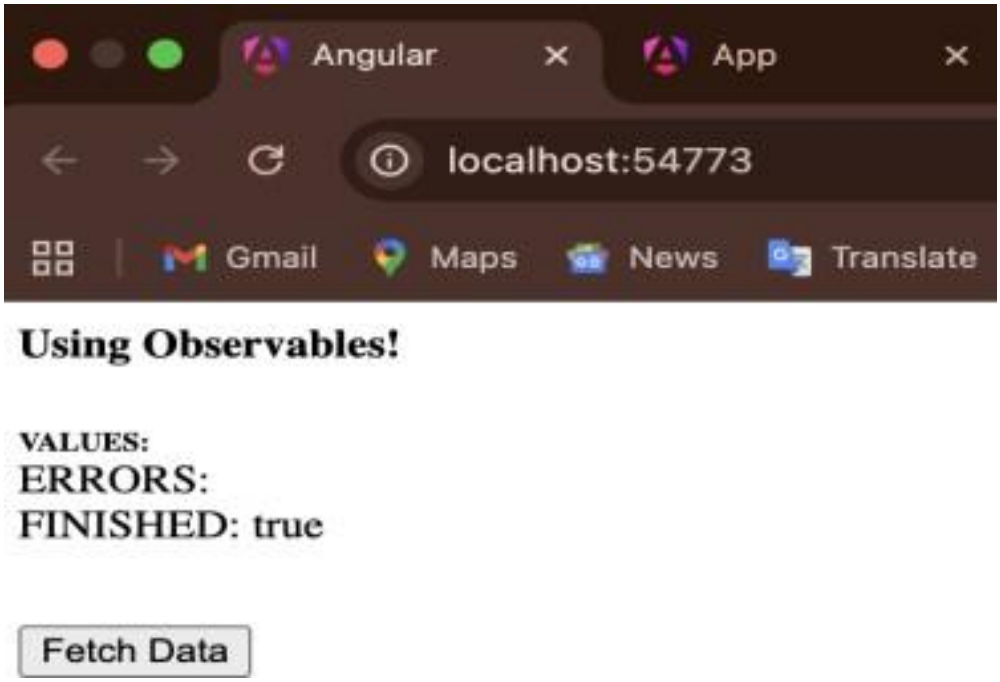
Explanation :

- ngFor loop is iterated on myArray and display the values on the page
- {{ errors }} will render the value of errors property if any
- Displays finished property value when complete() method of is executed
- Button click event is bound with fetchData() method which is invoked and creates an observable with a stream of numeric values.

OUTPUT:

Run the below CLI command to see the output

ng serve –o



Exercise -9

A. Server Communication using HttpClient:

- HttpClient is a service in Angular that allows your application to communicate with backend servers using HTTP requests.
- It is part of the @angular/common/http module.
- Supports GET, POST, PUT, DELETE, PATCH, etc.
- Works with Observables (RxJS), so you can handle asynchronous operations easily.

Advantages of HttpClient:

- **Simplified Requests:** Easy methods for GET, POST, PUT, DELETE.
- **Observable Support:** Handles asynchronous data using RxJS.
- **Type Safety:** Ensures correct data types for responses.
- **Automatic JSON Handling:** Parses responses and serializes requests.
- **Error Handling:** Easy management of network or server errors.
- **Interceptors:** Add tokens, logging, or handle errors globally.
- **Headers & Params:** Easily add headers or query parameters.
- **Consistent API:** Standardized HTTP communication across Angular apps.
- **Extensible & Maintainable:** Encourages modular, reusable, and testable code.

Observables and Subscribe:

- **Observable:** An Observable is a stream of data that arrives asynchronously over time. In Angular, HttpClient methods return Observables for HTTP requests.
  Usage of Observable:

```
getData(): Observable<any> {
  return this.http.get('https://api.example.com/data');
}
```

- **Subscribe:** To receive data from an Observable, use subscribe(). Triggers the HTTP request and handles success, error, and completion.
  Executes the request and processes the response.
  Without **`subscribe()`**, the HTTP request **does not run**.

```
this.dataService.getData().subscribe(
  (response) => console.log('Data:', response),  // Success
  (error) => console.error('Error:', error),    // Error
  () => console.log('Request complete')         // Completion (optional)
);
```

Problem Statement: Create an application for Server Communication using HttpClient

Goal: Create an Angular app that fetches data from a server (for example, a public API like JSONPlaceholder).

Key Feature: Use HttpClient to perform GET request.

Output: Display fetched data (e.g., list of posts) on the webpage

Step 1: Create a New Angular Project

1. Open Command Prompt / Terminal.
2. Run: **ng new server-communication**

Step 2: Open **app.component.ts** file and write:

src > app > TS app.component.ts > AppComponent > ngOnInit > next

```ts
import { Component, OnInit, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HttpClientModule, HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, HttpClientModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  posts: any[] = [];
  currentPage = 1;
  postsPerPage = 10;

  private http = inject(HttpClient);

  ngOnInit(): void {
    this.fetchPosts().subscribe({
      next: (data) => {
        this.posts = data;
      },
      error: (err) => console.error('Error fetching posts', err)
    });
  }

  fetchPosts(): Observable<any[]> {
    return this.http.get<any[]>('https://jsonplaceholder.typicode.com/posts');
  }

  // Pagination helpers
  get paginatedPosts() {
    const start = (this.currentPage - 1) * this.postsPerPage;
    return this.posts.slice(start, start + this.postsPerPage);
  }
```

```ts

  totalPages(): number {
    return Math.ceil(this.posts.length / this.postsPerPage);
  }

  changePage(page: number) {
    if (page >= 1 && page <= this.totalPages()) {
      this.currentPage = page;
    }
  }
}
```

This standalone Angular component fetches posts from a remote server using HttpClient, displays them in a list, and provides simple client-side pagination showing 10 posts per page. It handles errors during data fetching, calculates total pages, and allows navigation between

pages, demonstrating server communication, data binding, and pagination in an easy-to-understand way.

Step 3: Create a Service for Server Communication

Creating a service keeps the code clean. Run **ng generate service post.** This generates **post.service.ts** inside src/app. Open post.service.ts and write:

```
src > app > TS post.service.ts > ...
1   import { Injectable } from '@angular/core';
2   import { HttpClient } from '@angular/common/http';
3   import { Observable } from 'rxjs';
4
5   @Injectable({
6     providedIn: 'root'
7   })
8   export class PostService {
9     private apiUrl = 'https://jsonplaceholder.typicode.com/posts';
10
11    constructor(private http: HttpClient) {}
12
13    getPosts(): Observable<any[]> {
14      return this.http.get<any[]>(this.apiUrl);
15    }
16  }
```

The PostService is an Angular injectable service that provides a method to fetch posts from a remote server using HttpClient. It defines a getPosts() function that returns an Observable containing an array of posts from https://jsonplaceholder.typicode.com/posts, allowing components to subscribe and retrieve the data asynchronously.

Step 4: Open **app.component.html file** and write:

This HTML template displays a list of posts fetched from the server and supports pagination. It uses Angular's *ngFor directive to loop through paginatedPosts, showing the title and body of each post. Navigation buttons allow users to move between pages, with the current page and total pages displayed, while disabling buttons when at the first or last page.

```
src > app > <> app.component.html > ...
1   <div class="container">
2     <h1>Posts from Server</h1>
3
4     <ul>
5       <li *ngFor="let post of paginatedPosts" class="post-card">
6         <strong>{{ post.title }}</strong>
7         <p>{{ post.body }}</p>
8       </li>
9     </ul>
10
11    <div class="pagination">
12      <button (click)="changePage(currentPage - 1)" [disabled]="currentPage === 1">Previous</button>
13      <span>Page {{currentPage}} of {{ totalPages() }}</span>
14      <button (click)="changePage(currentPage + 1)" [disabled]="currentPage === totalPages()">Next</button>
15    </div>
16  </div>
```

Step 5: Open **app.component.css (Optional can be left as empty)** and write:

src > app > # app.component.css > ...

```css
.container {
  max-width: 800px;
  margin: 20px auto;
  font-family: Arial, sans-serif;
}

h1 {
  text-align: center;
  color: #333;
  margin-bottom: 20px;
}

.post-card {
  padding: 15px;
  margin-bottom: 15px;
  border: 1px solid #ddd;
  border-radius: 8px;
  background-color: #fafafa;
}

.post-card strong {
  display: block;
  font-size: 18px;
  margin-bottom: 5px;
}

.post-card p {
  margin: 0;
  font-size: 14px;
  color: #555;
}

.pagination {
  text-align: center;
  margin-top: 20px;
}
```

```css

.pagination button {
  margin: 0 10px;
  padding: 5px 15px;
  font-size: 14px;
  cursor: pointer;
}

.pagination button:disabled {
  cursor: not-allowed;
  opacity: 0.5;
}
```

This CSS styles the posts display and pagination in the application. The .container centers the content and sets a maximum width, while .post-card adds padding, borders, rounded corners, and background color for each post. Headings, text, and paragraphs are styled for readability, and the .pagination section centers the navigation buttons, with disabled buttons visually indicated by reduced opacity and a “not-allowed” cursor.

Step 6: Update the **Main.ts** and **Main.server.ts** files **:**

src > TS main.ts > ...

```ts
// Add Zone.js at the top to fix NG0908
import 'zone.js';
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent)
  .catch(err => console.error(err));
```

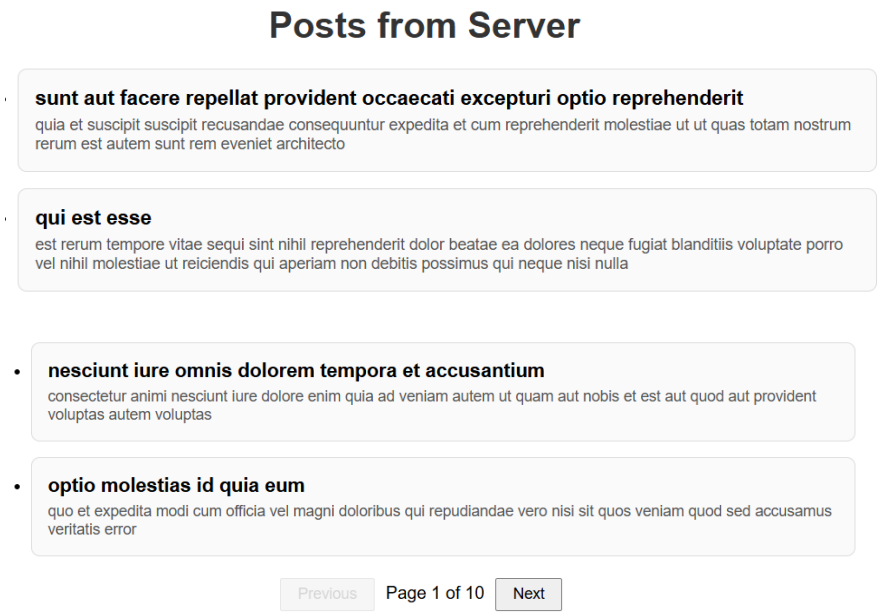
Note: Install Zone.js using the command **npm install zone.js**

src > TS main.server.ts > ...

```ts
import { BootstrapContext, bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { config } from './app/app.config.server';

const bootstrap = (context: BootstrapContext) =>
    bootstrapApplication(AppComponent,config,context)

export default bootstrap;
export { AppComponent};
```

These code snippets show how to bootstrap a standalone Angular application. The first snippet uses a custom bootstrap function with a configuration and context, while the second snippet demonstrates a basic bootstrap using bootstrapApplication with AppComponent. Additionally, zone.js is imported to enable Angular's change detection and prevent runtime errors like NG0908.

Step 7: Run the application using the Command **ng serve -- open**



B. Communicating with Different Backend Services Using Angular HttpClient

Angular HttpClient is a built-in service that allows an Angular application to communicate with various backend services and APIs over HTTP/HTTPS. It simplifies sending and receiving data from different servers, making it easier to integrate multiple backend services (such as REST APIs, cloud services, or microservices) into a single Angular application.

Key Features:

1.Multiple HTTP Methods – Supports GET, POST, PUT, PATCH, DELETE, and more for full CRUD operations.

2.Observable-Based – Returns RxJS Observable objects, enabling easy data streaming, cancellation, and transformation.

3.Type Safety – Allows defining response types (HttpClient.get<MyType>()) for safer and cleaner code.

4.Interceptors – Add authentication tokens, modify headers, handle errors, or log requests globally.

5.Request Options – Easily configure headers, query parameters, and response types.

6.Automatic JSON Handling – Automatically converts JSON responses to JavaScript/TypeScript objects.

7.Error Handling & Retry – Built-in mechanisms for catching errors and retrying failed requests.

8.Test-Friendly – Provides HttpClientTestingModule for mocking backend responses during testing.

Advantages:

Simplified Backend Communication – Handles complex HTTP requests with minimal code.

Reactive Programming – Works seamlessly with RxJS operators for real-time updates and data manipulation.

Centralized Logic – Services can manage API calls in one place, reducing code duplication.

Multiple Backend Integration – Easily connect to different APIs (e.g., user service, product service, payment gateway) within one application.

Scalable & Maintainable – Clear separation of frontend and backend logic makes the app easier to grow and maintain.

Why We Use HttpClient

We use HttpClient because it provides a secure, efficient, and structured way to interact with backend servers. It reduces manual work (like handling raw XMLHttpRequests), ensures better type checking, simplifies error handling, and integrates smoothly with Angular's dependency injection and change detection system.
This makes it ideal for applications that need to communicate with multiple backend services such as REST APIs, cloud endpoints, or third-party services.

Problem Statement: Create a custom service called ProductService in which Http class is used to fetch data stored in the JSON files.

Brief Overview: The ProductService is a custom Angular service that uses the HttpClient class to fetch product data from a JSON file stored in the project's assets. Instead of writing HTTP requests directly in a component, this service handles all the logic for making the GET request and returning the data as an Observable. Components can then simply inject the ProductService and subscribe to the data, keeping the code cleaner, more organized, and easier to maintain. This approach also makes it simple to update the data source or reuse the service in multiple components.

Step 1: Create a New Angular Project

1. Open Command Prompt / Terminal.
2. Run: **ng new product-service**

Step 2: Create an Product.Json file:

The products.json file, located in the public folder, stores a list of products in JSON format. It contains an array of objects where each object represents a product with two properties: name

(the product name) and price (the product cost). For example, it includes items like a Laptop priced at ₹75,000, a Mobile Phone at ₹25,000, and Headphones at ₹3,000. This file acts as the data source that the Angular application fetches through the ProductService to display the product list dynamically on the browser.

public > {} products.json > ...

```
[
  { "name": "Laptop", "price": 75000 },
  { "name": "Mobile Phone", "price": 25000 },
  { "name": "Headphones", "price": 3000 }
]
```

Step 3: Create a Service for Server Communication

Creating a service keeps the code clean. Run **ng generate service product.** This generates **product.service.ts** inside src/app. Otherwise You can directly create an file named product.service.ts under src/app. Open **product.service.ts** and write:

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

export interface Product {
  name: string;
  price: number;
}

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private jsonURL = 'assets/products.json';

  constructor(private http: HttpClient) {}

  getProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(this.jsonURL);
  }
}
```

The **ProductService** is an Angular service that uses the HttpClient to fetch product data from the products.json file located in the **assets** folder. It defines a Product interface to specify the structure of each product (name and price) and provides a getProducts() method that returns an **Observable** of a product array. This allows the application to retrieve the product list asynchronously and display it dynamically in the component.

Step 4:Open **app.component.ts** and write:

This Angular component (AppComponent) is a **standalone** component that displays a list of products by fetching data from the products.json file using Angular's HttpClient. It imports CommonModule to enable structural directives like *ngFor and HttpClientModule for making HTTP requests. When the component initializes, the loadProducts() method sends an HTTP GET request to retrieve the product data and assigns it to the products array. The template then dynamically displays each product's name and price in an unordered list.

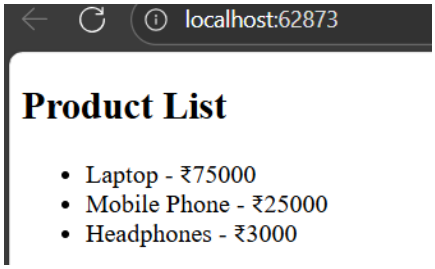
```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HttpClientModule, HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

interface Product {
  name: string;
  price: number;
}

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, HttpClientModule], // ✅ Important for *ngFor and HttpClient
  template: `
    <h2>Product List</h2>
    <ul>
      <li *ngFor="let product of products">
        {{ product.name }} - ₹{{ product.price }}
      </li>
    </ul>
  `,
})
export class AppComponent {
  products: Product[] = [];
  constructor(private http: HttpClient) {
    this.loadProducts();
  }

  loadProducts() {
  this.http.get<Product[]>('/products.json')
  .subscribe(data => this.products = data);

  }
}
```

Step 5: Open **main.ts** and update as follows:

```
src > TS main.ts > ...
import 'zone.js';                                  // <-- add this line
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent).catch(err => console.error(err));
```

This code is the entry point of the Angular application. It first imports zone.js, which Angular requires to track and update changes in the application. The bootstrapApplication function from @angular/platform-browser is then used to launch the app by bootstrapping the root component (AppComponent). If any error occurs during the startup process, it is caught and logged to the console.

Step 6: Run the application using the command **ng serve**



The output displays a **Product List** with three items—**Laptop (₹75,000)**, **Mobile Phone (₹25,000)**, and **Headphones (₹3,000)**—fetched dynamically from a JSON file.