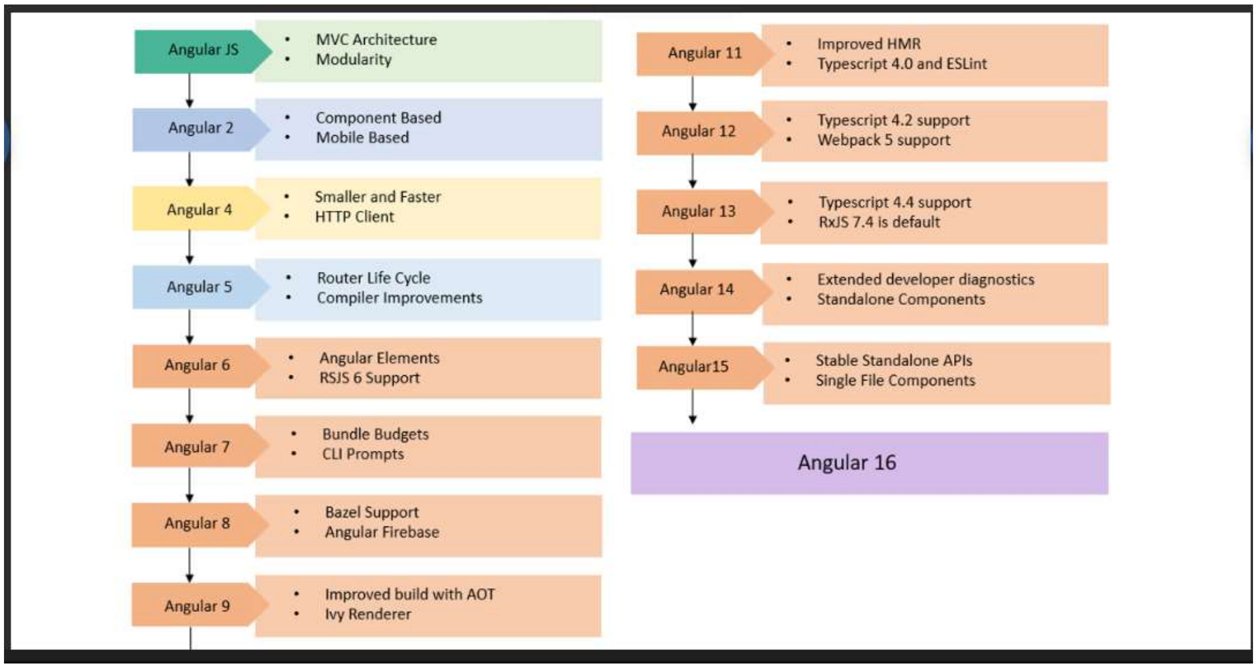
UNIT-V

Why Angular?
Angular 1 is a JavaScript framework from Google which was used for the development of web applications. Current Angular has come a long way from its inception.

What is Angular?

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

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



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

1. Component-Based Architecture

- Applications are built using components.
- Each component has its own logic, template, and style.
- Promotes reusability and modularity.

2. Two-Way Data Binding
   - Synchronizes the model and the view.
   - Changes in the UI update the data model automatically and vice versa.
3. Dependency Injection (DI)
   - Angular has a built-in DI system to inject services and objects.
   - Improves testability and scalability.
4. Powerful Routing
   - The Angular Router enables navigation between views and supports lazy loading.
   - Helps manage complex navigation logic and guards (like auth checks).
5. TypeScript-Based
   - Angular is written in TypeScript, a superset of JavaScript.
   - Brings static typing, interfaces, and decorators to improve development experience.
6. Directives
   - Custom HTML tags or attributes that enhance the behavior of elements.
   - Structural directives (like *ngIf, *ngFor) control layout.
   - Attribute directives modify the appearance or behavior of DOM elements.
7. RxJS and Reactive Programming
   - Uses RxJS for handling asynchronous events with Observables.
   - Encourages reactive design patterns for event handling and data streams.
8. Angular CLI (Command Line Interface)
   - Speeds up development with powerful scaffolding tools.
   - Handles building, testing, linting, and deploying apps.
9. Ahead-of-Time (AOT) Compilation
   - Compiles Angular HTML and TypeScript code into efficient JavaScript during build time.
   - Improves performance and reduces load times.
10. Testing Support
    - Designed with testing in mind (unit tests and end-to-end tests).
    - Integrated tools like Karma (unit tests) and Protractor (E2E tests).
11. Internationalization (i18n)
    - Built-in support for multiple languages and locale formatting.
    - Helps build apps for global users.
12. Performance Optimization Tools
    - Features like lazy loading, differential loading, and tree-shaking improve performance.

Angular Application Setup:

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

- Node.js (^18.15.0 || ^20.9.0) and npm (min version required 10.1.0)
- Angular CLI (v16)
- Visual Studio Code

Install Node.js and Visual Studio Code from their respective official websites.
To check whether the Node is installed or not in your machine, go to the Node command prompt and check the

Node version by typing the following command:

node -v

It will display the version of the Node installed.

Steps to install Angular CLI

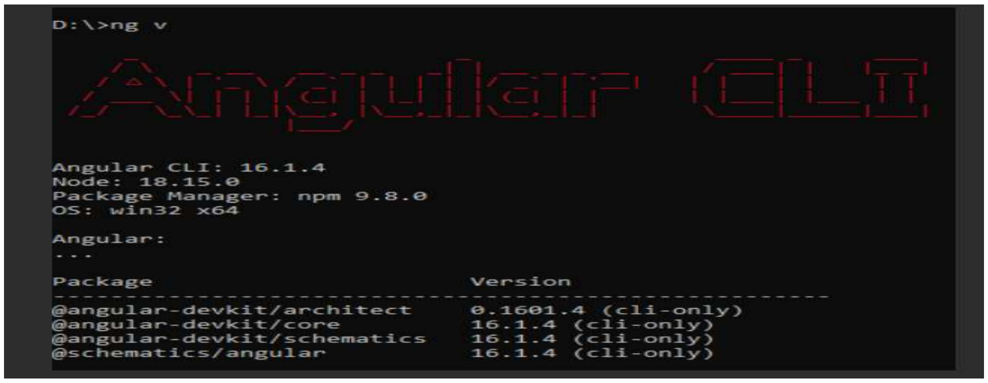
Angular CLI can be installed using Node package manager. The Angular version that will be discussed in this course is v16. Now run the following command to install the Angular CLI v16:

```
D:\> npm install -g @angular/cli@16
```

Test successful installation of Angular CLI using the following command.

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

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



```
D:\>ng v

Angular CLI: 16.1.4
Node: 18.15.0
Package Manager: npm 9.8.0
OS: win32 x64

Angular:
...

Package                         Version
----------------------------------------------------------
@angular-devkit/architect       0.1601.4 (cli-only)
@angular-devkit/core            16.1.4 (cli-only)
@angular-devkit/schematics      16.1.4 (cli-only)
@schematics/angular             16.1.4 (cli-only)
```

Angular CLI is a command-line interface tool to build Angular applications. It makes application development faster and easier to maintain.
Using CLI, you can create projects, add files to them, and perform development tasks such as testing, bundling, and deployment of applications.

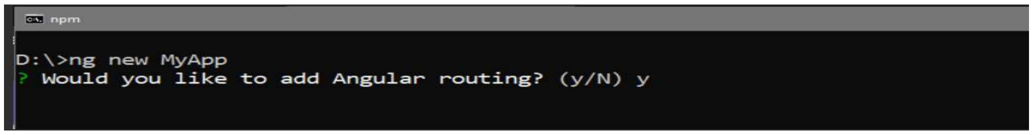
Angular CLI commands:

Command	Purpose
npm install -g @angular/cli@16	Installs Angular CLI v16 globally
ng new <project name>	Creates a new Angular application
ng serve –open	Builds and runs the application on lite-server and launches a browser
ng generate <name>	Creates class, component, directive, interface, module, pipe, and service
ng build	Builds the application
ng update @angular/cli @angular/core	Updates Angular to a newer version

Creating an Angular Application:
Creating an Angular application using Angular CLI
Exploring the Angular folder structure
Demo steps:
1. Create an application with the name 'MyApp' using the following CLI command

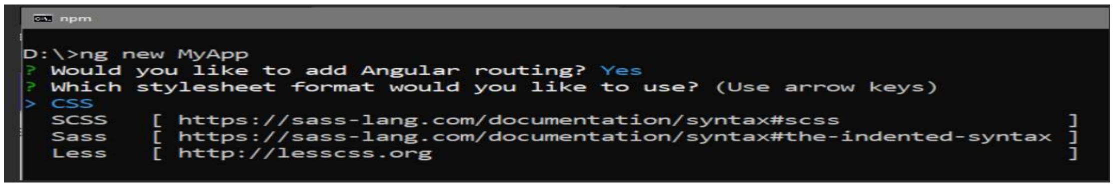
```
D:\>ng new MyApp
```

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



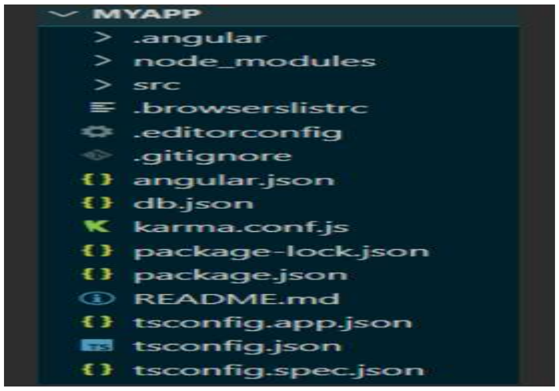
```
D:\>ng new MyApp
? Would you like to add Angular routing? (y/N) y
```

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



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

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



```
MYAPP
  > .angular
  > node_modules
  > src
  .browserslistrc
  .editorconfig
  .gitignore
  angular.json
  db.json
  karma.conf.js
  package-lock.json
  package.json
  README.md
  tsconfig.app.json
  tsconfig.json
  tsconfig.spec.json
```

Components and Modules

Components:

A **component** is essentially a **special type of directive** that is used to create reusable UI blocks with **their own template, logic, and scope**.

Open Visual Studio Code IDE. Go to the File menu and select the "Open Folder" option. Select the MyApp folder you have created earlier.

Observe for our AppComponent you have below files
app.component.ts
app.component.html
app.component.css
Let us explore each one of them
Go to src folder-> app -> open **app.component.ts** file

Observe the following code

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

Line 3: Adds component decorator to the class which makes the class a component
Line 4: Specifies the tag name to be used in the HTML page to load the component
Line 5: Specifies the template or HTML file to be rendered when the component is loaded in the HTML page. The template represents the view to be displayed
Line 6: Specifies the stylesheet file which contains CSS styles to be applied to the template.
Line 8: Every component is a class (AppComponent, here) and export is used to make it accessible in other components
Line 9: Creates a property with the name title and initializes it to value 'AngDemo'

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

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

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

Modules:

A **module** in AngularJS helps organize your code into reusable, manageable pieces.

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

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

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

Executing Angular Application:

Execute the application and check the output.

Open terminal in Visual Studio Code IDE by selecting View Menu -> Integrated Terminal.

Type the following command to run the application

D:\MyApp>ng serve –open

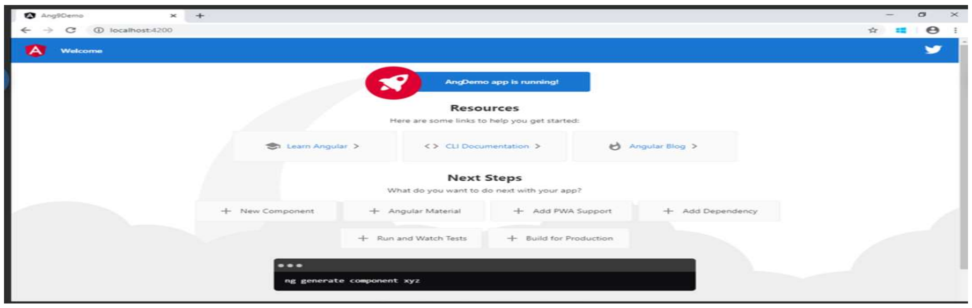
ng serve will build and run the application

--open option will show the output by opening a browser automatically with the default port.

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

D:\MyApp>ng serve --open --port 3000

Following is the output of the MyApp Application



```
ng generate component xyz
```

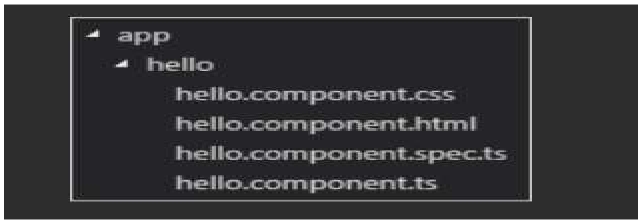
Creating a Component:

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

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

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

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



```
app
  hello
    hello.component.css
    hello.component.html
    hello.component.spec.ts
    hello.component.ts
```

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

```
1.  import { Component, OnInit } from '@angular/core';
2.
3.  @Component({
4.  selector: 'app-hello',
5.  templateUrl: './hello.component.html',
6.  styleUrls: ['./hello.component.css']
7.  })
8.  export class HelloComponent implements OnInit {
9.  courseName: string = "Angular";
10.
11. constructor() { }
12.
13. ngOnInit() {
14. }
15.
16. }
```

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

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

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

```
1.  p {
2.  color:blue;
3.  font-size:20px;
4.  }
```

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

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

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

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

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

8. Now run the application by giving the following command

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

Elements of Template

In Angular, templates are the HTML part of a component, defining what gets rendered on the screen. They use a combination of standard HTML, Angular-specific syntax, and directives to create dynamic and interactive user interfaces.

Templates Basics:

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

Example
app.component.html

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

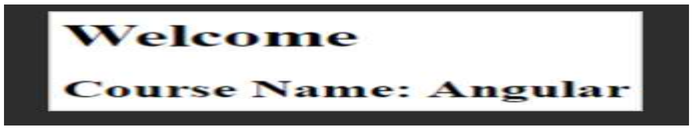
app.component.ts

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

@Component({
selector: 'app-root',
templateUrl:'./app.component.html',
6. styleUrls: ['./app.component.css']
7. })
8. export class AppComponent {
9. courseName = "Angular";
10. }
```

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

Output:

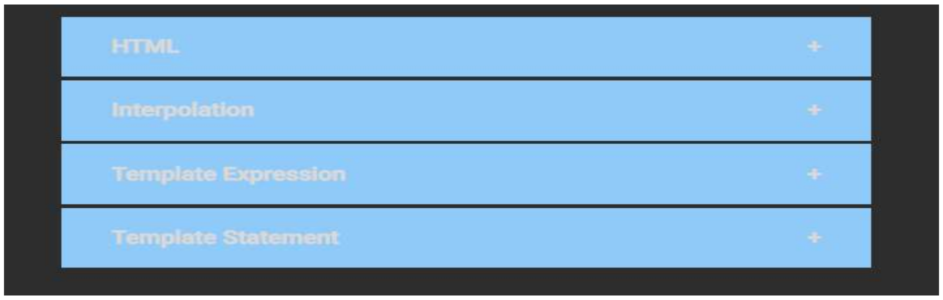


Welcome

Course Name: Angular

Elements of Template

The basic elements of template syntax are:



HTML +

Interpolation +

Template Expression +

Template Statement +

Example:

app.component.ts

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

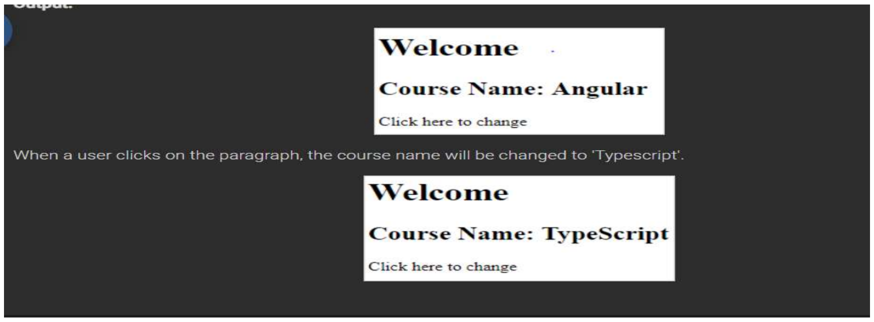
Line 5-7: changeName is a method of AppComponent class where you are changing courseName property value to "TypeScript".

app.component.html

```
1. <h1> Welcome </h1>
2. <h2> Course Name: {{ courseName }}</h2>
3. <p (click)="changeName()">Click here to change</p>
```

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

Output:


Welcome

Course Name: Angular

Click here to change

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

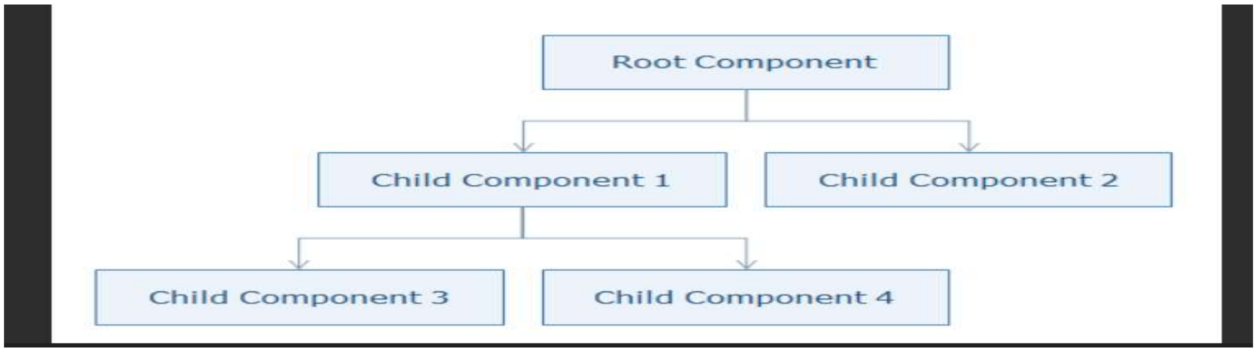
Welcome

Course Name: TypeScript

Click here to change

Change Detection:

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



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

*directive-name = expression

Angular has few built-in structural directives such as:

- ngIf
- ngFor
- ngSwitch

Attribute directives
An Attribute directives change the appearance/behavior of a component/element.
Following are built-in attribute directives:

- ngStyle
- ngClass

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

Syntax:
*ngIf = "expression"

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

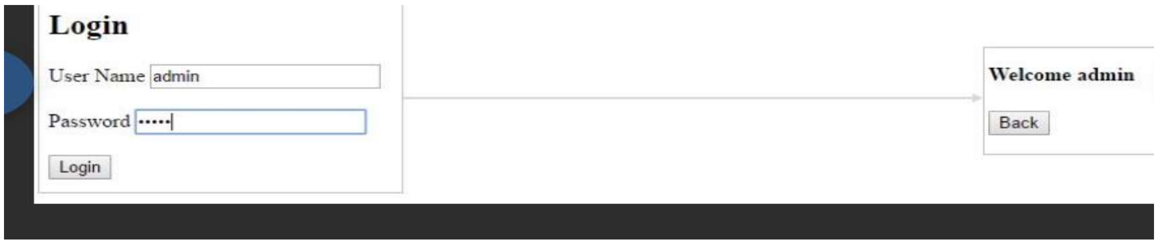
Example:
app.component.ts

```
...
export class AppComponent {
  isAuthenticated!: boolean;
  submitted = false;
  userName!: string;
  onSubmit(name: string, password: string) {
    this.submitted = true;
    this.userName = name;
    if (name === "admin" && password === "admin") {
      this.isAuthenticated = true;
    } else {
      this.isAuthenticated = false;
    }
  }
}
```

app.component.html

```
<div *ngIf="!submitted">
  ...
  <button (click)="onSubmit(username.value, password.value)">Login</button>
</div>
<div *ngIf="submitted">
  <div *ngIf="isAuthenticated; else failureMsg">
    <h4>Welcome {{ userName }}</h4>
  </div>
  <ng-template #failureMsg>
    <h4>Invalid Login !!! Please try again...</h4>
  </ng-template>
  <button type="button" (click)="submitted = false">Back</button>
</div>
```

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



ngIf:
Understanding ngIf directive
Invoking a directive on user's action
Demosteps:

Problem Statement: 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 Login!!! Please try again..." message as shown below:

1. Open **app.component.ts and** write the following code:

```
import { Component } from '@angular/core';
    @Component({
     selector: 'app-root',
     templateUrl: './app.component.html',
     styleUrls: ['./app.component.css'],
    })
    export class AppComponent {
     isAuthenticated!: boolean;
     submitted = false;
     userName!: string;
     onSubmit(name: string, password: string) {
       this.submitted = true;
       this.userName = name;
       if (name === 'admin' && password === 'admin') {
         this.isAuthenticated = true;
       } else {
         this.isAuthenticated = false;
       }
     }
    }
```

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

```
<div *ngIf="!submitted">
<form>
<label>User Name</label>
<input type="text" #username /><br /><br />
<label for="password">Password</label>
<input type="password" name="password" #password /><br />
</form>
<button (click)="onSubmit(username.value, password.value)">Login</button>
</div>
<div *ngIf="submitted">
<div *ngIf="isAuthenticated; else failureMsg">
<h4>Welcome {{ userName }}</h4>
</div>
<ng-template #failureMsg>
<h4>Invalid Login !!! Please try again...</h4>
</ng-template>
<button type="button" (click)="submitted = false">Back</button>
</div>
```

3. Add AppComponent to the bootstrap property in the root module file i.e., **app.module.ts**

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

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

4. Ensure the index.html displays app-root.

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

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

ngFor Directive:

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

Syntax:

*ngFor = "expression"

Example:

app.component.ts

```
...
export class AppComponent {
  courses: any[] = [
    { id: 1, name: "TypeScript" },
    { id: 2, name: "Angular" },
    { id: 3, name: "Node JS" },
    { id: 1, name: "TypeScript" }
  ];
}
```

app.component.html

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

```
</ul>
```
Output:

```
4
```

Demo : ngFor
Understanding ngFor directive
Iteration of an array using ngFor directive
Demosteps:
Problem Statement: Creating a courses array and rendering it in the template using ngFor directive in a list format as shown below

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

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  courses: any[] = [
    { id: 1, name: 'TypeScript' },
    { id: 2, name: 'Angular' },
    { id: 3, name: 'Node JS' },
    { id: 1, name: 'TypeScript' }
  ];
}
```

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

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

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

ngSwitch Directive:

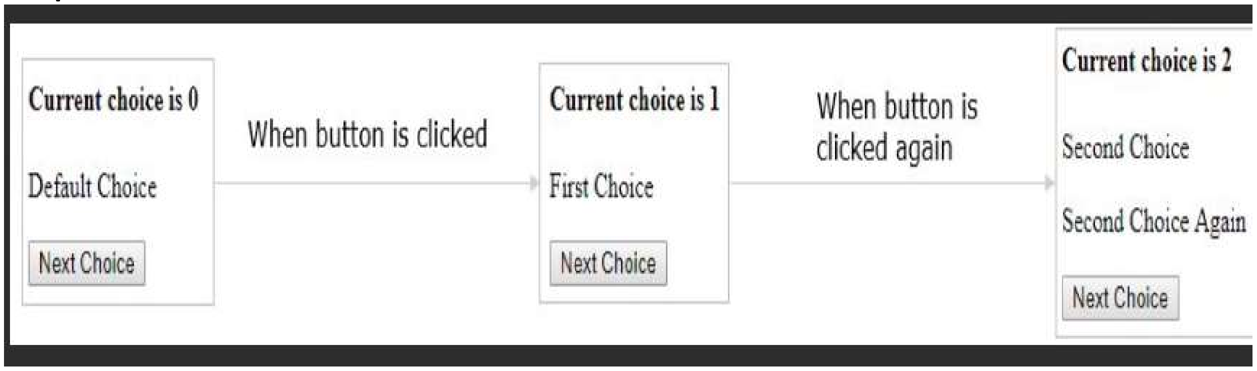
ngSwitch adds or removes DOM trees when their expressions match the switch expression. Its syntax is comprised of two directives, an attribute directive, and a structural directive.
It is very similar to a switch statement in JavaScript and other programming languages.
Example:
app.component.ts

```
...
export class AppComponent {
  choice = 0;
  nextChoice() {
    this.choice++;
  }
}
```

app.component.html

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

Output:



```
Current choice is 0

Default Choice

Next Choice
```

```
Current choice is 1

First Choice

Next Choice
```

```
Current choice is 2

Second Choice

Second Choice Again

Next Choice
```

Demo : ngSwitch

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

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  choice = 0;
  nextChoice() {
    this.choice++;
  }
}
```

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

```
<h4>
  Current choice is {{ choice }}
</h4>
<div [ngSwitch]="choice">
  <p *ngSwitchCase="1">First Choice</p>
  <p *ngSwitchCase="2">Second Choice</p>
  <p *ngSwitchCase="3">Third Choice</p>
  <p *ngSwitchCase="2">Second Choice Again</p>
  <p *ngSwitchDefault>Default Choice</p>
</div>
<div>
```

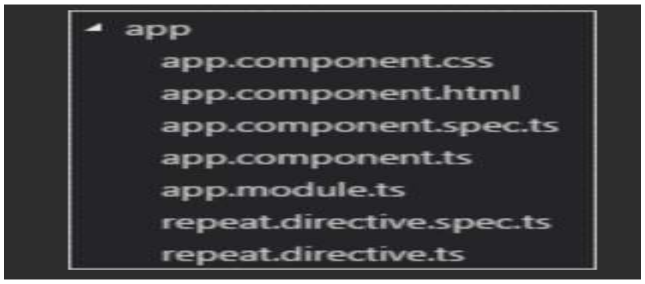
```
  <button (click)="nextChoice()">
              Next Choice
        </button>
</div>
```
3. Save the files and check the output in the browser

Custom Structural Directive:

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

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

To create a custom structural directive, create a class annotated with @Directive
```
@Directive({
})
class MyDirective{}
```
Generate a directive called 'repeat' using the following command
```
D:\MyApp>ng generate directive repeat
```
This will create two files under the src\app folder with names repeat.directive.ts and repeat.directive.spec.ts (this is for testing). Now the app folder structure will look as shown below:



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

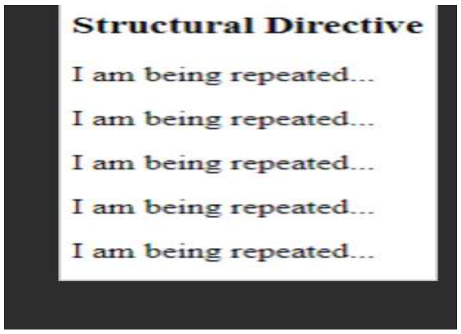
app.module.ts
```
...
import { RepeatDirective } from './repeat.directive';
@NgModule({
  declarations: [
    AppComponent,
    RepeatDirective
  ],
 ...
})
export class AppModule { }
```

Open **repeat.directive.ts** file and add the following code
```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
@Directive({
  selector: '[appRepeat]'
})
export class RepeatDirective {
  constructor(private templateRef: TemplateRef<any>, private viewContainer: ViewContainerRef) { }
  @Input() set appRepeat(count: number) {
    for (let i = 0; i < count; i++) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    }
  }
}
```

app.component.html

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

Output:



```
Structural Directive
I am being repeated...
I am being repeated...
I am being repeated...
I am being repeated...
I am being repeated...
```

Demo on Custom Structural Directive :

1. Generate a directive called 'repeat' using the following command:

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

2. Write the below-given code in **app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
        import { NgModule } from '@angular/core';
        import { AppComponent } from './app.component';
        import { RepeatDirective } from './repeat.directive';
        @NgModule({
          declarations: [
            AppComponent,
            RepeatDirective
          ],
          imports: [
            BrowserModule
          ],
          providers: [],
          bootstrap: [AppComponent]
        })
        export class AppModule { }
```

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

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
    @Directive({
     selector: '[appRepeat]'
    })
    export class RepeatDirective {
     constructor(private templateRef: TemplateRef<any>, private viewContainer:
ViewContainerRef) { }
     @Input() set appRepeat(count: number) {
      for (let i = 0; i < count; i++) {
       this.viewContainer.createEmbeddedView(this.templateRef);
      }
     }
    }
```

4. Write the below-given code in app.component.htm

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

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

Attribute Directives:

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

Following are built-in attribute directives:

- ngStyle
- ngClass

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

[style.<cssproperty>] = "value"

Example:

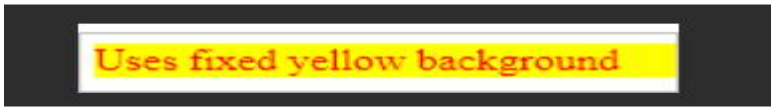
app.component.ts

```
...
export class AppComponent {
 colorName = 'yellow';
 color = 'red';
}
```

app.component.html

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

Output:

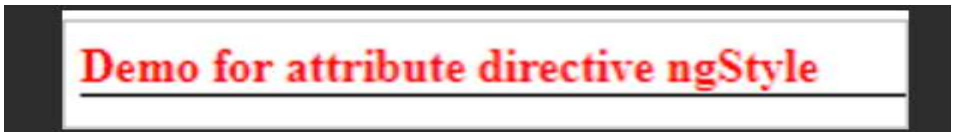


Uses fixed yellow background

Example:

app.component.ts

```
...
export class AppComponent {
  colorName = 'red';
  fontWeight = 'bold';
  borderStyle = '1px solid black';
}
```
app.component.html
```
<p [ngStyle]="{
        color:colorName,
        'font-weight':fontWeight,
        borderBottom: borderStyle
      }">
  Demo for attribute directive ngStyle
</p>
```
Output:


Demo for attribute directive ngStyle

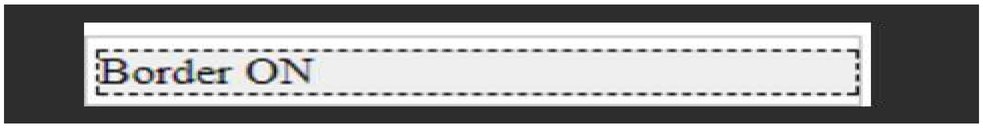
It allows you to dynamically set and change the CSS classes for a given DOM element. Use the following syntax to set a single CSS class to the element which is also known as class binding.

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

Example:

app.component.ts
```
...
export class AppComponent {
  isBordered = true;
}
```
app.component.html
```
<div [class.bordered]="isBordered">
  Border {{ isBordered ? "ON" : "OFF" }}
</div>
```
In **app.component.css**, add the following CSS class
```
.bordered {
      border: 1px dashed black;
      background-color: #eee;
}
```

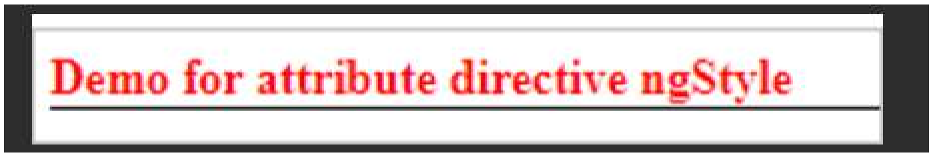
Output:


Border ON

Demo : ngStyle

Demosteps:

Problem Statement: Apply multiple CSS properties to a paragraph in a component using ngStyle. The output should be as shown below:


Demo for attribute directive ngStyle

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

```
import { Component } from '@angular/core';
            @Component({
              selector: 'app-root',
              templateUrl: './app.component.html',
              styleUrls: ['./app.component.css']
            })
            export class AppComponent {
              colorName = 'red';
              fontWeight = 'bold';
              borderStyle = '1px solid black';
            }
```

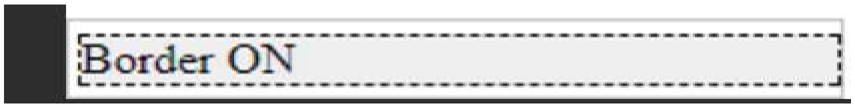
2. Write the below-given code in **app.component.html**

```
<p [ngStyle]="{
                    color:colorName,
                    'font-weight':fontWeight,
                    borderBottom: borderStyle
                  }">
              Demo for attribute directive ngStyle
            </p>
```

Demo : ngClass

Demosteps:

Problem Statement: Applying multiple CSS classes to the text using ngClass directive. The output should be as shown below


Border ON

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

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  isBordered = true;
}
```

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

```
<div [ngClass]="{bordered: isBordered}">
  Border {{ isBordered ? "ON" : "OFF" }}
</div>
```

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

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

Custom Attribute Directive:

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

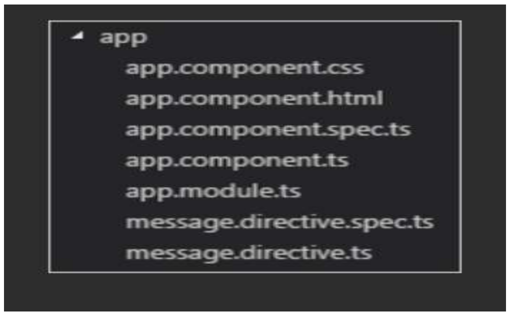
Problem Statement: Create an attribute directive called 'showMessage' which should display the given message in a paragraph when a user clicks on it and should change the text color to red. As there is no built-in directive available to implement this functionality, you need to go for a custom directive.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

```
@Directive({
})
class MyDirective { }
```

Example:
Generate a directive called 'message' using the following command

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

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



```
app
    app.component.css
    app.component.html
    app.component.spec.ts
    app.component.ts
    app.module.ts
    message.directive.spec.ts
    message.directive.ts
```

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

```
...
import { MessageDirective } from './message.directive';
@NgModule({
  declarations: [
    AppComponent,
    MessageDirective
  ],
  ...
})
export class AppModule { }
```

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

```
import { Directive, ElementRef, Renderer2, HostListener, Input } from '@angular/core';
@Directive({
  selector: '[appMessage]',
})
```

```
export class MessageDirective {
  @Input('appMessage') message!: string;
  constructor(private el: ElementRef, private renderer: Renderer2) {
    renderer.setStyle(el.nativeElement, 'cursor', 'pointer');
  }
  @HostListener('click') onClick() {
    this.el.nativeElement.innerHTML = this.message;
    this.renderer.setStyle(this.el.nativeElement, 'color', 'red');
  }
}
```

app.component.ts

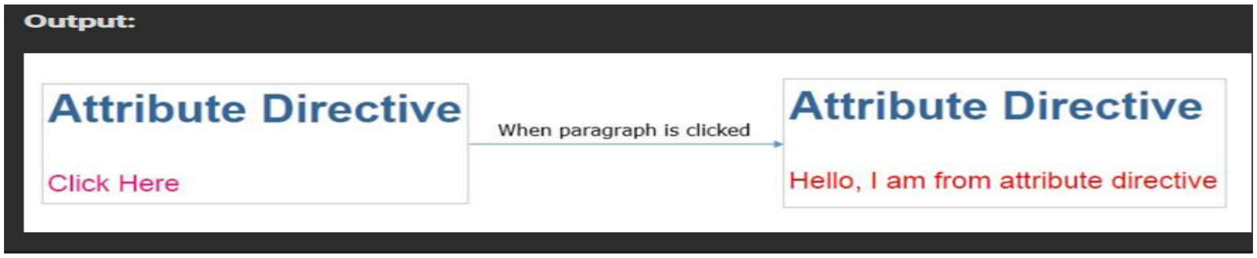
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myMessage="Hello, I am from attribute directive"
}
```

app.component.html

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

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

```
h3 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
p {
  color: #ff0080;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 150%;
}
```


Output:
Attribute Directive
Click Here
When paragraph is clicked
Attribute Directive
Hello, I am from attribute directive

Property 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

Types of Data Binding and it's syntax

• a 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

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

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' />
or
<img bind-src = 'imageUrl' />
```

ere 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

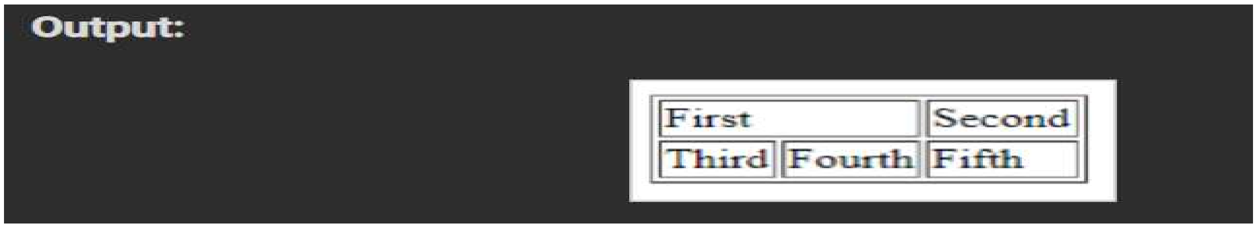
```
...
export class AppComponent {
  imgUrl: string = 'assets/imgs/logo.jpg';
}
```

app.component.html

```
<img [src]='imgUrl' width=200 height=100>
```

Attribute Binding:

Property binding will not work for a few elements/pure attributes like ARIA, SVG, and COLSPAN. In such cases, you need to go for attribute binding.

Attribute binding can be used to **bind a component property to the attribute directly**.
For example,
<td colspan = "{{ 2+3 }}">Hello</td>
he 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.
<td [attr.colspan] = "2+3">Hello</td>
Example:
app.component.ts
```
...
export class AppComponent {
  colspanValue: string ="2";
}
```
app.component.html
```
<table border=1>
   <tr>
      <td [attr.colspan]="colspanValue"> First </td>
      <td>Second</td>
   </tr>
   <tr>
      ...
   </tr>
</table>
```


First		Second
Third	Fourth	Fifth

Style and Event 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:
[style.styleproperty]

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.font-size.px] = "isValid ? 3 : 6">Hello</button>
Here text font size will be set to 3 px if the expression isValid is true, otherwise, it will be set to 6px.
The ngStyle directive is preferred when it is required to set multiple inline styles at the same time.
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>

Built-in Pipes:

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

Pipes in Angular:

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

Syntax:
{{ expression | pipe }}
Example:
{{ "Angular" | uppercase }}

his will display ANGULAR

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

- uppercase
- lowercase
- titlecase
- currency
- date
- percent
- slice
- decimal

Examples of commonly used Built-in Pipes:
Uppercase
This pipe converts the template expression into uppercase.
Syntax:

{{ expression | uppercase }}
Example:
{{ "Laptop" | uppercase }}

Passing Parameters to Angular Pipes:
A pipe can also have optional parameters to change the output. To pass parameters, after a pipe name add a colon(:) followed by the parameter value.
Syntax:
pipename : parametervalue
A pipe can also have multiple parameters as shown below
pipename : parametervalue1:parametervalue2

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

currency

This pipe displays a currency symbol before the expression. By default, it displays the currency symbol $
Syntax:
{{ expression | currency:currencyCode:symbol:digitInfo:locale }}
currencyCode is the code to display such as INR for the rupee, EUR for the euro, etc.
symbol is a Boolean value that represents whether to display currency symbol or code.

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

digitInfo is a string in the following format
{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}

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

locale is used to set the format followed by a country/language. To use a locale, the locale needs to be registered in the root module.
For Example,to set locale to French (fr), add the below statements in app.module.ts
import { registerLocaleData } from '@angular/common';
import localeFrench from '@angular/common/locales/fr';
registerLocaleData(localeFrench);
Examples:
{{ 25000 | currency }} will display $25,000.00
{{ 25000 | currency:'CAD' }} will display CA$25,000.00
{{ 25000 | currency:'CAD':'code' }} will display CAD25,000.00
{{ 25000 | currency:'CAD':'symbol':'6.2-3'}} will display CA$025,000.00
{{ 25000 | currency:'CAD': 'symbol-narrow':'1.3'}} will display $25,000.000
{{ 250000 | currency:'CAD':'symbol':'6.3'}} will display CA$250,000.000
{{ 250000 | currency:'CAD':'symbol':'6.3':'fr'}} will display 250 000,000 CA$

date

This pipe can be used to display the date in the required format
Syntax:
{{ expression | date:format:timezone:locale }}
An **expression** is a date or number in milliseconds
The **format** indicates in which form the date/time should be displayed. Following are the pre-defined options for it.

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

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

Timezone to be used for formatting. For example, '+0430' (4 hours, 30 minutes east of the Greenwich meridian) If not specified, the local system timezone of the end-user's browser will be used.
locale is used to set the format followed by a country/language. To use a locale, register the locale in the root module.
For Example, to set locale to French (fr), add the below statements in app.module.ts

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

Examples:
{{ "6/2/2017" | date }} will display Jun 2, 2017
{{ "6/2/2017, 11:30:45 AM" | date:'medium' }} will display Jun 2, 2017, 11:30:45 AM
{{ "6/2/2017, 11:30:45 AM" | date:'mmss' }} will display 3045
{{"1/31/2018, 11:05:04 AM" | date:'fullDate':'0':'fr'}} will display mercredi 31 janvier 2018
{{ 90000000 | date }} will display Jan 2, 1970 – date pipe will start from Jan 1, 1970 and based on the given number of milliseconds, it displays the date

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

Syntax:
{{ expression | percent:digitInfo:locale }}

digitInfo is a string in the following format
{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}

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

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

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

Examples:
{{ 0.1 | percent }} will display 10%
{{ 0.1 | percent:'2.2-3' }} will display 10.00%
{{ 0.1 | percent:'2.2-3': 'fr' }} will display 10.00 %

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

Syntax:
{{ expression | slice:start:end }}

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

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

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

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

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

number
This pipe can be used to format a number.

Syntax:
{{ expression | number:digitInfo }}

gitInfo is a string in the following format
{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}

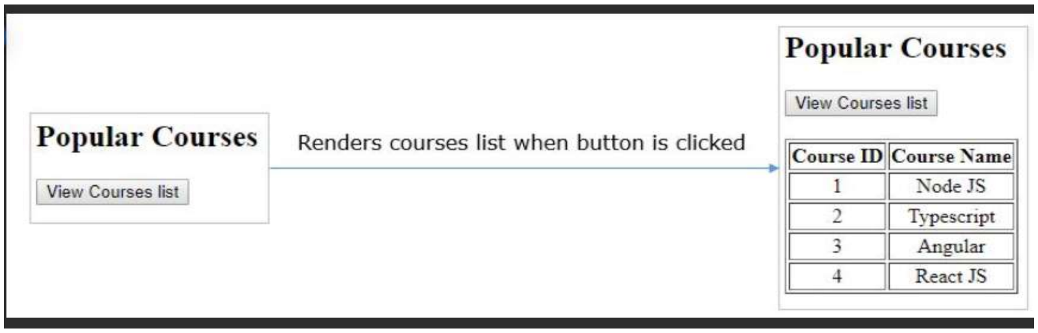
- minIntegerDigits is the minimum integer digits to display. The default value is 1.
- minFractionDigits is the minimum number of digits to display after the fraction. The default value is 0.
- maxFractionDigits is the maximum number of digits to display after fraction. The default value is 3.

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

Nested Components Basics:

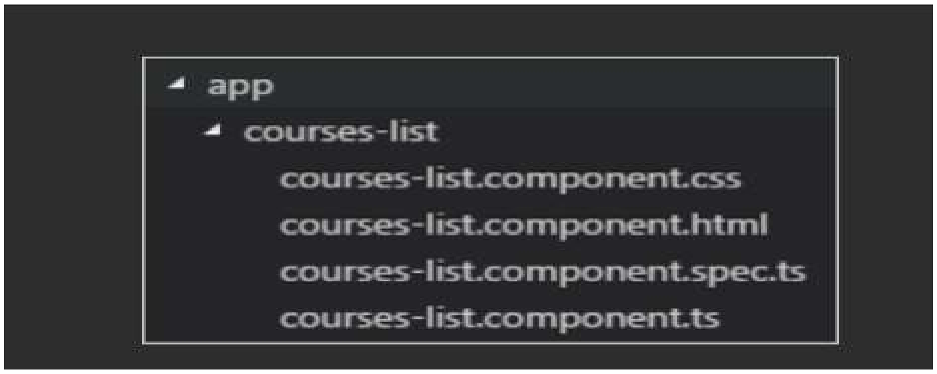
Nested component is a component that is loaded into another component
The component where another component is loaded onto is called a container component/parent component.
The root component is loaded in the index.html page using its selector name. Similarly, to load one component into a parent component,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

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



Course ID	Course Name
1	Node JS
2	Typescript
3	Angular
4	React JS

Create a component called the coursesList using the following CLI command
D:\MyApp>ng generate component coursesList
This command will create four files called courses-list.component.ts, courses-list.component.html,courses-list.component.css, courses-list.component.spec.ts and places them inside a folder called courses-list under the app folder as shown below



```
app
  courses-list
    courses-list.component.css
    courses-list.component.html
    courses-list.component.spec.ts
    courses-list.component.ts
```

This command will also add the CoursesList component to the root module.

app.module.ts

```
...
import { CoursesListComponent } from './courses-list/courses-list.component';
@NgModule({
  declarations: [
    AppComponent,
    CoursesListComponent
  ],
 ...
})
export class AppModule { }
```

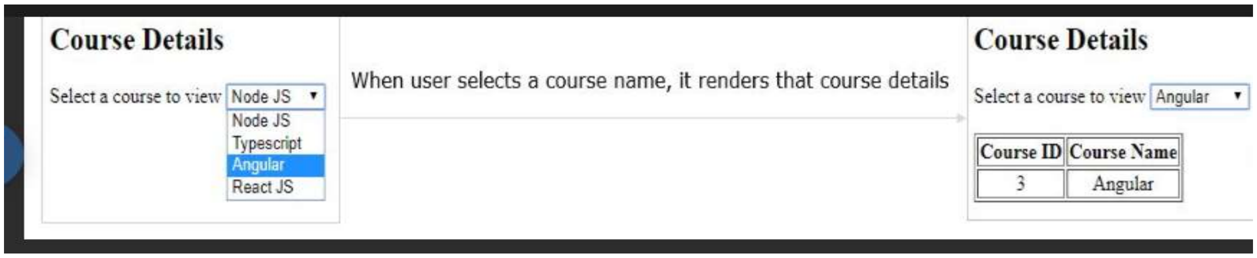
courses-list.component.ts

```
...
export class CoursesListComponent {
  courses = [
    { courseId: 1, courseName: 'Node JS' },
    { courseId: 2, courseName: 'Typescript' },
    { courseId: 3, courseName: 'Angular' },
    { courseId: 4, courseName: 'React JS' }
  ];
}
```
courses-list.component.html
```
<table border="1">
 ...
  <tbody>
    <tr *ngFor="let course of courses">
      <td>{{course.courseId}}</td>
      <td>{{course.courseName}}</td>
    </tr>
  </tbody>
</table>
```
Add the following code in **courses-list.component.css**
```
tr{
   text-align:center;
}
```
app.component.html
```
...
<button (click)="show=true">View Courses list</button><br/><br/>
<div *ngIf="show">
   <app-courses-list></app-courses-list>
</div>
```
Creating a nested component
Loading nested component in a container component

Demosteps:
Problem Statement: Loading CourseslistComponent in the root component when a user clicks on the View courses list button as shown below,

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

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


Course Details
Select a course to view Node JS
Node JS
Typescript
Angular
React JS
When user selects a course name, it renders that course details
Course Details
Select a course to view Angular

Course ID	Course Name
3	Angular

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

```
...
export class CoursesListComponent {
...
  course!: any[];
  @Input() set cName(name: string) {
    this.course = [];
    for (var i = 0; i < this.courses.length; i++) {
      if (this.courses[i].courseName === name) {
        this.course.push(this.courses[i]);
      }
    }
  }
}
```

courses-list.component.html

```
<table border="1" *ngIf="course.length>0">
 ...
  <tbody>
    <tr *ngFor="let c of course">
      <td>{{c.courseId}}</td>
      <td>{{c.courseName}}</td>
    </tr>
  </tbody>
</table>
```

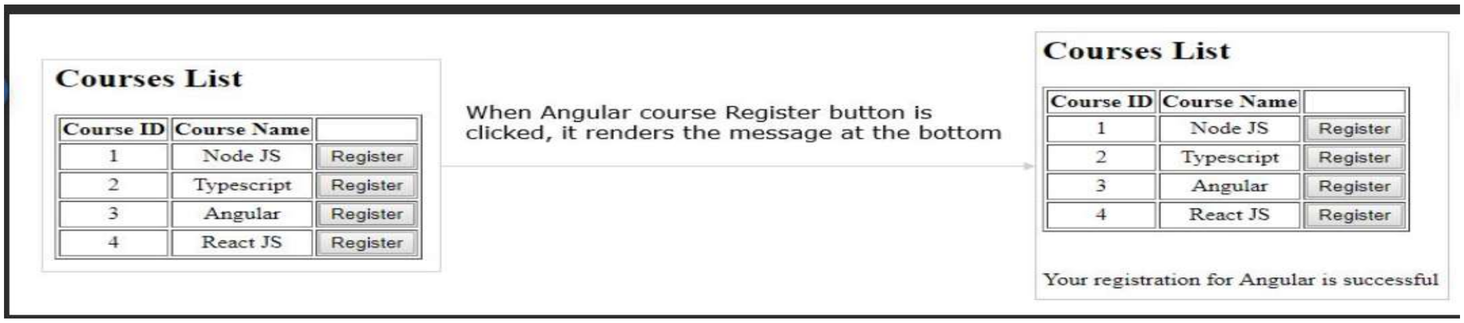
Add the following code in **app.component.html**

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

Passing data from Child Component to Container Component

If a child component wants to send data to its parent component, then it must create a property with @Output decorator.
The only method for the child component to pass data to its parent component is through events. The property must be of type EventEmitter
Example:
Problem Statement: Create an AppComponent that loads another component called CoursesList component. Create another component called CoursesListComponent which should display the courses list in a table along with a register button in each row. When a user clicks on

the register button, it should send that courseName value back to AppComponent where it should display the registration successful message along with courseName


Courses List
When Angular course Register button is clicked, it renders the message at the bottom
Courses List

Course ID	Course Name	
1	Node JS	Register
2	Typescript	Register
3	Angular	Register
4	React JS	Register

Course ID	Course Name	
1	Node JS	Register
2	Typescript	Register
3	Angular	Register
4	React JS	Register

Your registration for Angular is successful

courses-list.component.ts

```
...
export class CoursesListComponent {
  @Output() onRegister = new EventEmitter<string>();
  ...
  register(courseName: string) {
    this.onRegister.emit(courseName);
  }
}
```

courses-list.component.html

```
<table border="1">
...
  <tbody>
    <tr *ngFor="let course of courses">
      <td>{{course.courseId}}</td>
      <td>{{course.courseName}}</td>
      <td><button (click)="register(course.courseName)">Register</button></td>
    </tr>
  </tbody>
</table>
```

app.component.html

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

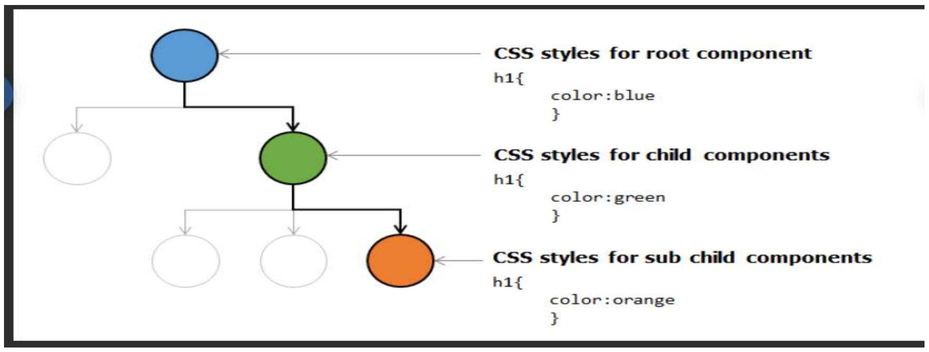
app.component.ts

```
...
export class AppComponent {
  message!: string;
  courseReg(courseName: string) {
    this.message = `Your registration for ${courseName} is successful`;
  }
}
```

Shadow DOM:

Shadow DOM is a web component standard by W3C. It **enables encapsulation for DOM tree and styles**. Shadow DOM hides DOM logic behind other elements and confines styles only for that component.
For example, in an Angular application, n number of components will be created and each component will have its own set of data and CSS styles. When these are integrated, there is a chance that the data and styles may be applied to the entire application. Shadow DOM encapsulates data and styles for each component to not flow through the entire application.

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


CSS styles for root component
h1{
color:blue
}
CSS styles for child components
h1{
color:green
}
CSS styles for sub child components
h1{
color:orange
}

Component Life Cycle

A component has a life cycle that is managed by Angular. It includes creating a component, rendering it, creating and rendering its child components, checks when its data-bound properties change, and destroy it before removing it from the DOM.
Angular has some methods/hooks which provide visibility into these key life moments of a component and the ability to act when they occur.
Following are the lifecycle hooks of a component. The methods are invoked in the same order as mentioned in the table below:

Interface	**Hook**	**Support**
OnChanges	ngOnChanges	Directive, Component
OnInit	ngOnInit	Directive, Component
DoCheck	ngDoCheck	Directive, Component

Interface	Hook	Support
AfterContentInit	ngAfterContentInit	Component
AfterContentChecked	ngAfterContentChecked	Component
AfterViewInit	ngAfterViewInit	Component
AfterViewChecked	ngAfterViewChecked	Component
OnDestroy	ngOnDestroy	Directive, Component

Syntax:

```
import { Component, OnInit,  DoCheck, AfterContentInit, AfterContentChecked,
      AfterViewInit, AfterViewChecked, OnDestroy } from '@angular/core';
    ...
    export class AppComponent implements OnInit,  DoCheck,  AfterContentInit, AfterContentChecked,
      AfterViewInit, AfterViewChecked,  OnDestroy {

      ngOnInit() {  }

      ngDoCheck() {  }

      ngAfterContentInit() { }

      ngAfterContentChecked() { }

      ngAfterViewInit() {  }

      ngAfterViewChecked() {   }

       ngOnDestroy() {  }
    }
```

Model Driven Forms or Reactive Forms:

ntroduction to Forms in Angular:

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

Forms in Angular:

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

- can capture user-provided data,
- can capture user input events,
- can validate the user input, etc.
- have their own approaches of processing and managing the form data:

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

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

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

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

State Detection CSS classes and its purpose

<u>Step1:</u>

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

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { RegistrationFormComponent } from './registration-form/registration-form.component';
@NgModule({
  declarations: [
    AppComponent,
    RegistrationFormComponent
  ],
  imports: [
    BrowserModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 2:
Create a component called **RegistrationForm** using the following CLI command

```
ng generate component RegistrationForm
```

Step 3:
Add the below to **app.component.html**
Step 4:
Bootstrap CSS framework (v3) is commonly used for adding basic structural layout and style for web applications. For structuring the RegistrationForm in this example, CSS classes from the Bootstrap CSS framework can be used.
To add Bootstrap CSS library to the application, install **bootstrap** as shown below:

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

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

```
...
"styles": [
         "src/styles.css",
         "node_modules/bootstrap/dist/css/bootstrap.min.css"
        ],
        "scripts": [
         "node_modules/jquery/dist/jquery.min.js",
         "node_modules/bootstrap/dist/js/bootstrap.min.js"
        ]
...
```

Step 5:
Include the below code to **registration-form.component.css**

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

Building Reactive Forms (Angular v13+)

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

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
@Component({
  selector: 'app-registration-form',
  templateUrl: './registration-form.component.html',
  styleUrls: ['./registration-form.component.css']
})
export class RegistrationFormComponent implements OnInit {
  registerForm!: FormGroup;
  submitted!:boolean;

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

For each form control:

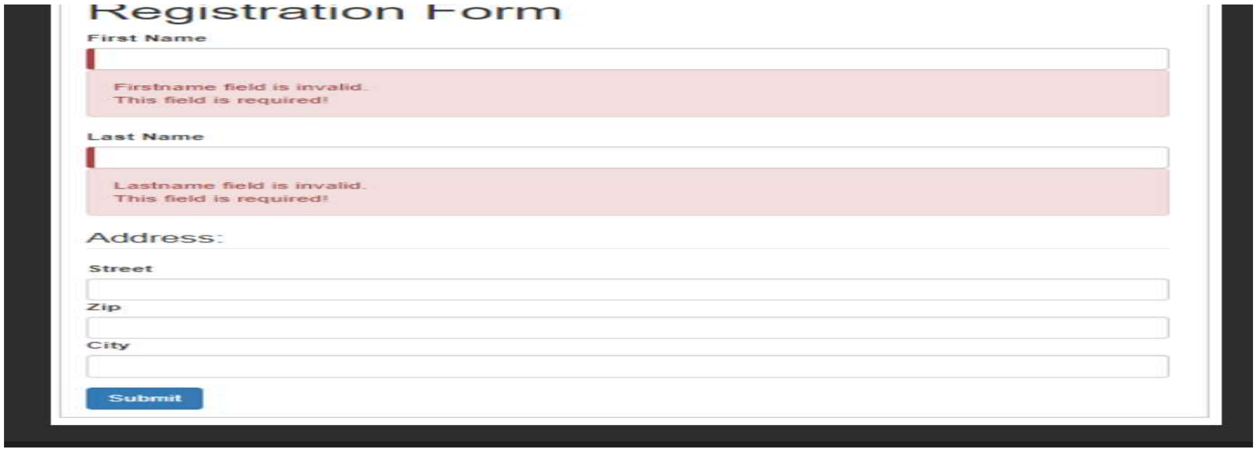
- you can mention the default value as the first argument and
- the list of validators as the second argument:
- Validations can be added to the form controls using the built-in validators supplied by the Validators class.
- For example: Configure built-in required validator for each control using ['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

registration-form.component.html

```
<div class="container">
   <h1>Registration Form</h1>
   <form [formGroup]="registerForm">
```

```
    <div class="form-group">
      <label>First Name</label>
      <input type="text" class="form-control" formControlName="firstName">
      <div *ngIf="registerForm.controls['firstName'].errors" class="alert alert-danger">
          Firstname field is invalid.
          <p *ngIf="registerForm.controls['firstName'].errors?.['required']">
              This field is required!
          </p>
      </div>
    </div>
    <div class="form-group">
      <label>Last Name</label>
      <input type="text" class="form-control" formControlName="lastName">
      <div *ngIf="registerForm.controls['lastName'].errors" class="alert alert-danger">
          Lastname field is invalid.
          <p *ngIf="registerForm.controls['lastName'].errors?.['required']">
              This field is required!
          </p>
      </div>
    </div>
    <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>
    <button type="submit" class="btn btn-primary" (click)="submitted=true">Submit</button>
  </form>
<br/>
  <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>
```

Save all the files and observe the output:


Registration Form

First Name

Firstname field is invalid.
This field is required!

Last Name

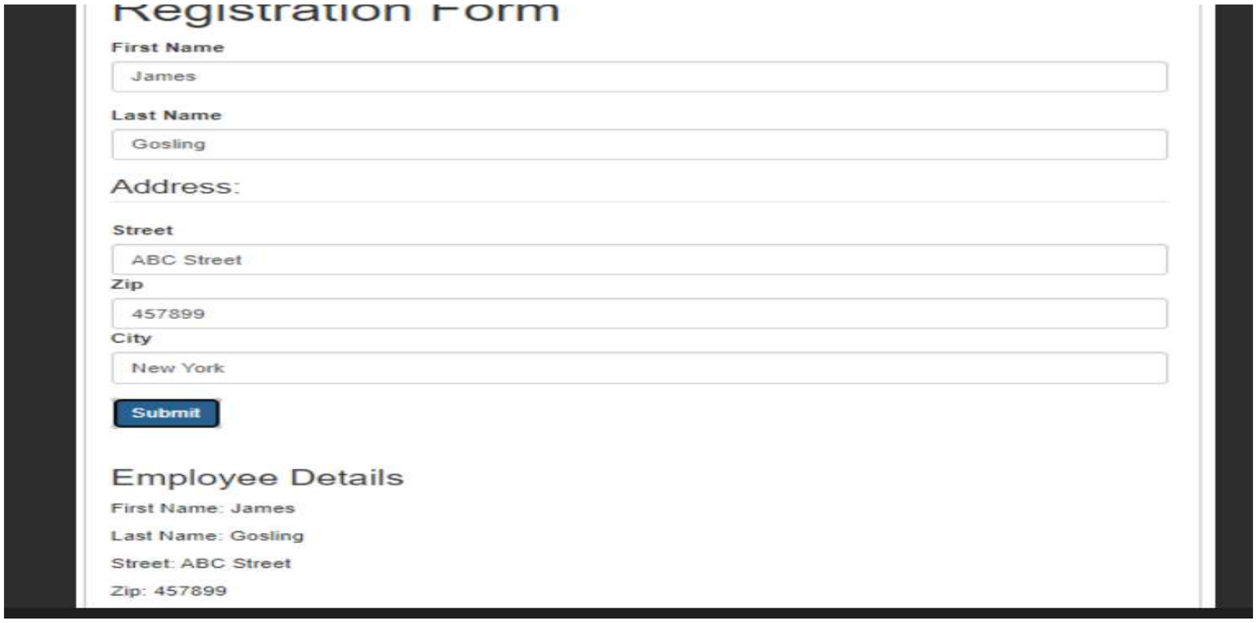
Lastname field is invalid.
This field is required!

Address:

Street

Zip

City


Registration Form

First Name: James

Last Name: Gosling

Address:

Street: ABC Street

Zip: 457899

City: New York

Employee Details

First Name: James

Last Name: Gosling

Street: ABC Street

Zip: 457899

Custom Validators in Reactive Forms:

Need for custom validation:

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

Custom validation in Reactive Forms of Angular:

Custom validation can be applied to form controls of a Reactive Form in Angular.
Custom validators are implemented as separate functions inside the component.ts file.
these functions can be added to the list of other validators configured for a form control.

Implementing custom validation in Reactive Forms of Angular:

Add one more field called email inside the example used for ReactiveForms previously. The below are the validations to be applied to the 'email' field:
required,
checks for standard email pattern, for example: abc@something.com, abc@something.co.in, etc.

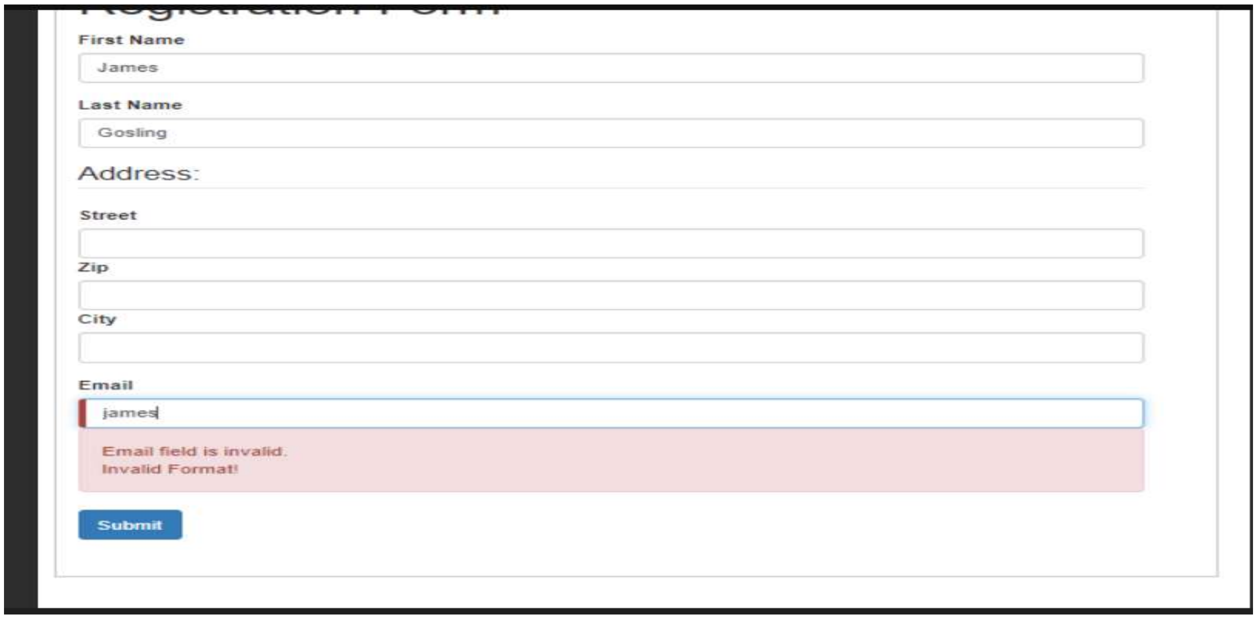
For implementing the for custom validation, add a separate function which checks for standard email pattern inside **registration-form.component.ts** as shown below:

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';
@Component({
  selector: 'app-registration-form',
  templateUrl: './registration-form.component.html',
  styleUrls: ['./registration-form.component.css']
})
export class RegistrationFormComponent implements OnInit {
  registerForm!: FormGroup;
  submitted!:boolean;

  constructor(private formBuilder: FormBuilder) { }
  ngOnInit() {
    this.registerForm = this.formBuilder.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      address: this.formBuilder.group({
        street: [],
        zip: [],
        city: []
      }),
      email: ['', validateEmail]
    });
  }
}
function validateEmail(c: FormControl): any {
  let EMAIL_REGEXP = /^([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$/;
  return EMAIL_REGEXP.test(c.value) ? null : {
    emailInvalid: {
      message: "Invalid Format!"
    }
  };
}
```

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

```
<div class="container">
    <h1>Registration Form</h1>
    <form [formGroup]="registerForm">
      <div class="form-group">
        <label>First Name</label>
        <input type="text" class="form-control" formControlName="firstName">
            <div *ngIf="registerForm.controls['firstName'].errors" class="alert alert-danger">
            Firstname field is invalid.
        <p *ngIf="registerForm.controls['firstName'].errors?.['required']">
          This field is required!
        </p>
      </div>
    </div>
    <div class="form-group">
      <label>Last Name</label>
      <input type="text" class="form-control" formControlName="lastName">
     <div *ngIf="registerForm.controls['lastName'].errors" class="alert alert-danger">
        Lastname field is invalid.
        <p *ngIf="registerForm.controls['lastName'].errors?.['required']">
          This field is required!
        </p>
      </div>
    </div>
    <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>
    <div class="form-group">
      <label>Email</label>
      <input type="text" class="form-control" formControlName="email" />
      <div *ngIf="registerForm.controls['email'].errors" class="alert alert-danger">
        Email field is invalid.
        <p *ngIf="registerForm.controls['email'].errors?.['required']">
          This field is required!
        </p>
        <p *ngIf="registerForm.controls['email'].errors?.['emailInvalid']">
          {{ registerForm.controls['email'].errors?.['emailInvalid'].message }}
        </p>
    </div>
    </div>
    <button type="submit" class="btn btn-primary" (click)="submitted=true">Submit</button>
  </form>
<br/>
  <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>
    <p>Email: {{ registerForm.get('email')?.value }}</p>
  </div>
 </div>
```
Output:



First Name

James

Last Name

Gosling

Address:

Street

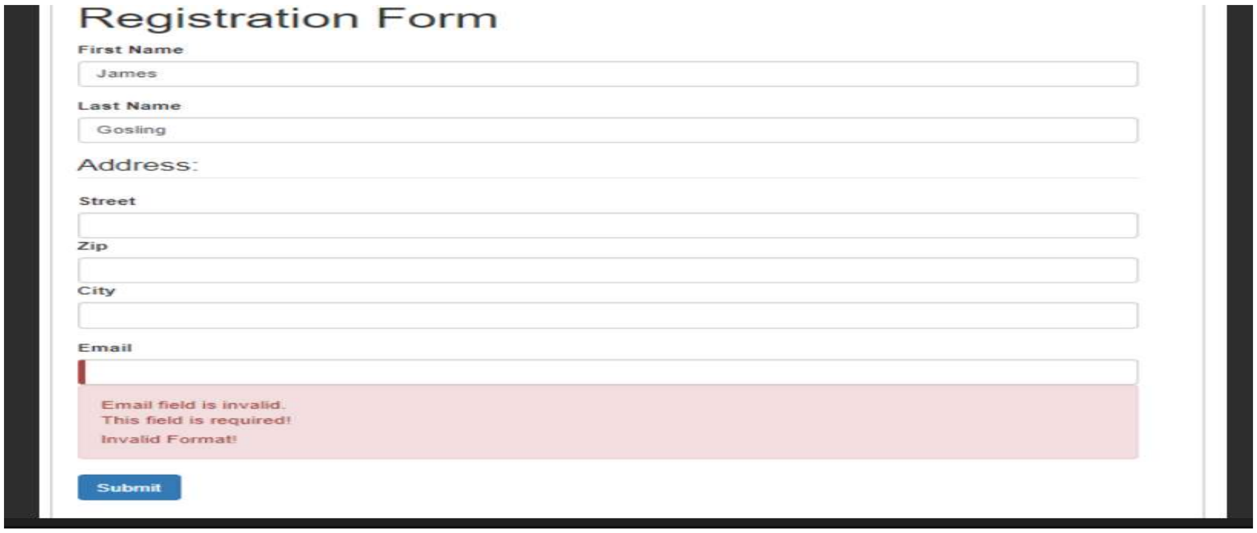
Zip

City

Email

james

Email field is invalid.
Invalid Format!



Registration Form

First Name

James

Last Name

Gosling

Address:

Street

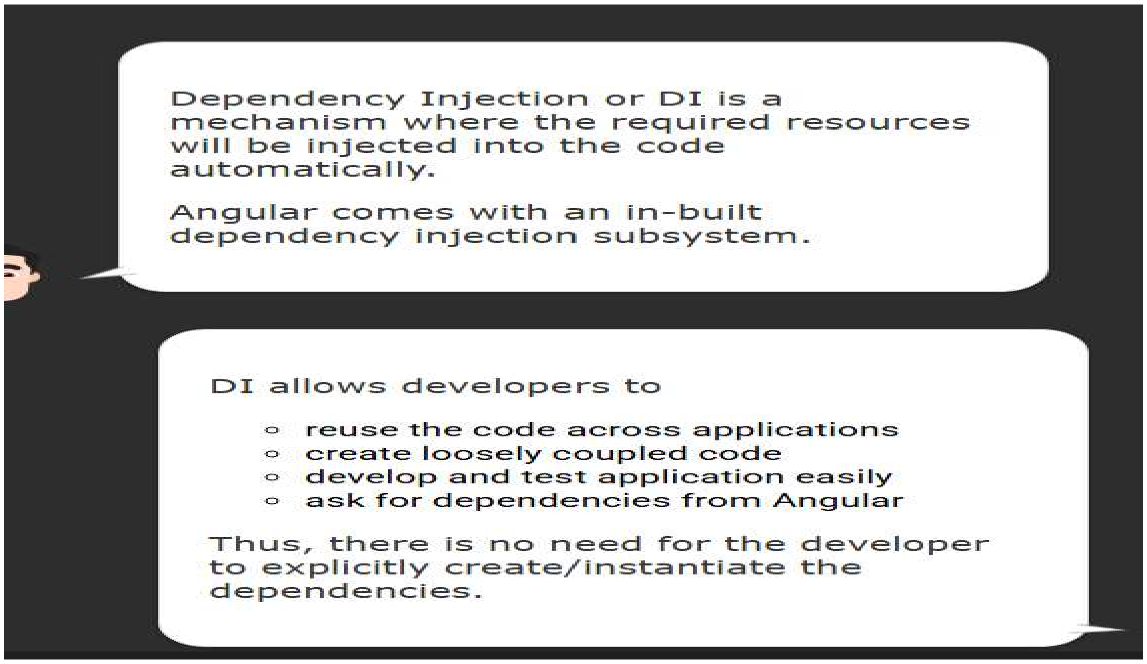
Zip

City

Email

Email field is invalid.
This field is required!
Invalid Format!

Dependency Injection:



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

Angular comes with an in-built dependency injection subsystem.

DI allows developers to

- reuse the code across applications
- create loosely coupled code
- develop and test application easily
- ask for dependencies from Angular

Thus, there is no need for the developer to explicitly create/instantiate the dependencies.

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

Why Services?
Services can be used to:
share the code across components of an application.
make HTTP requests.

Creating a Service
To create a service class, use the following command:

```
ng generate service book
```

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

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

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

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

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

```
import { Injectable } from '@angular/core';
@Injectable({
    providedIn: 'root'
})
export class BookService {}
```

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

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

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

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

Injecting a Service

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

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

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

Problem Statement:

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

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

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

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

```
export class Book {
   id!: number;
   name!: string;
}
```

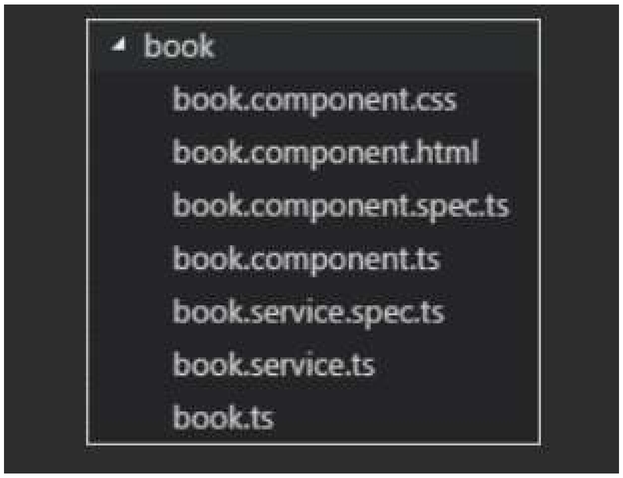
Create a file with the name **books-data.ts** under the book folder and add the following code.

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

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

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

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



```
book
    book.component.css
    book.component.html
    book.component.spec.ts
    book.component.ts
    book.service.spec.ts
    book.service.ts
    book.ts
```

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

```
import { Injectable } from '@angular/core';
import { Book } from './book';
import { BOOKS } from './books-data';
@Injectable({
   providedIn:'root'
})
export class BookService {
 getBooks() {
   return BOOKS;
 }
}
```

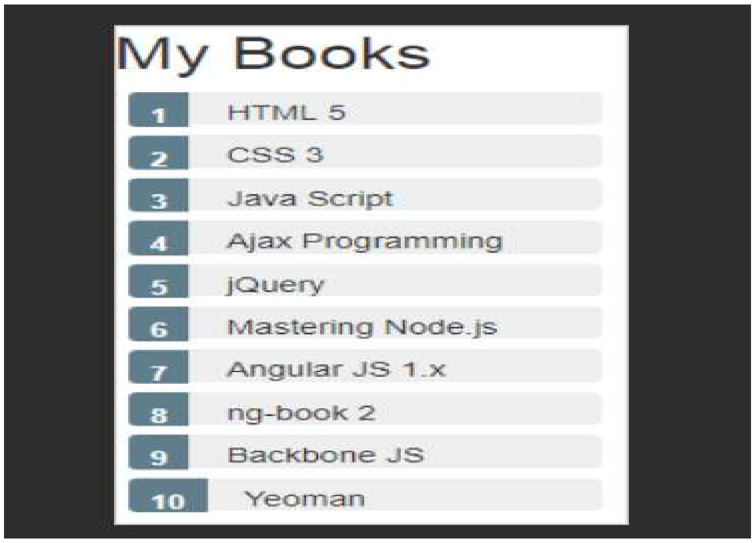
Add the following code in the **book.component.ts** file

```
...
import { BookService } from './book.service';
import { Book } from './book';
...
export class BookComponent implements OnInit {
 books!: Book[];
```

```
  constructor(private bookService: BookService) { }
  getBooks() {
    this.books = this.bookService.getBooks();
  }
  ngOnInit() {
    this.getBooks();
  }
}
```
book.component.html
```
<h2>My Books</h2>
<ul class="books">
  <li *ngFor="let book of books">
    <span class="badge">{{book.id}}</span> {{book.name}}
  </li>
</ul>
```
Add the following code in **book.component.css** which has styles for books
```
.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;
}
```

Update app.component.html to contain below code:

Output:



```
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
```

RxJS Observables:

RxJS

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

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

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

Why RxJS Observables?

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

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

<u>Create and use an observable in Angular</u>

Example:

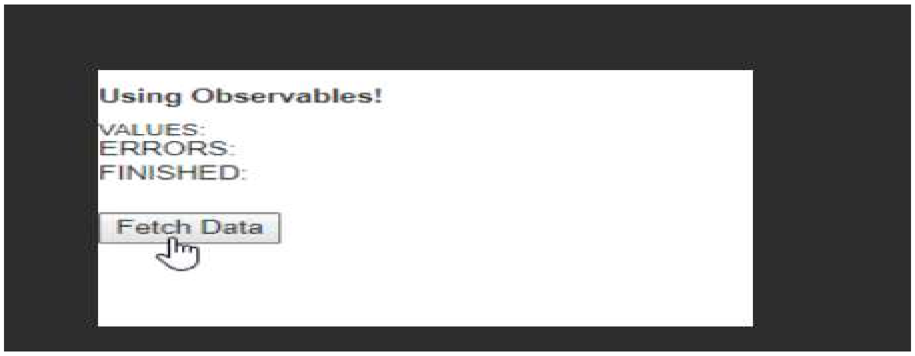
app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
@Component({
  selector: 'app-root',
  styleUrls: ['./app.component.css'],
  templateUrl: './app.component.html'
})
export class AppComponent {
  data!: Observable<number>;
  myArray: number[] = [];
  errors!: boolean;
  finished!: boolean;
  fetchData(): void {
    this.data = new Observable(observer => {
      setTimeout(() => { observer.next(11); }, 1000),
      setTimeout(() => { observer.next(22); }, 2000),
      setTimeout(() => { observer.complete(); }, 3000);
    });
    this.data.subscribe((value) => this.myArray.push(value),
      error => this.errors = true,
      () => this.finished = true);
  }
}
```

app.component.html

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

Output:



```
Using Observables!
VALUES:
ERRORS:
FINISHED:
```

Server Communication using HttpClient

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

Making a GET request:
The following statement is used to fetch data from a server
this.http.get(url)
tp.get by default returns an observable

<u>Using Server communication in the example used for custom services:</u>
Add HttpClientModule to the **app.module.ts** to make use of HttpClient class

```
...
import { HttpClientModule } from '@angular/common/http';
...
@NgModule({
  imports: [BrowserModule, HttpClientModule],
  ...
})
export class AppModule { }
```

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

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse, HttpHeaders } from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError, tap } from 'rxjs/operators';
import { Book } from './book';
@Injectable({
   providedIn:'root'
})
export class BookService {
  constructor(private http: HttpClient) { }
  getBooks(): Observable<Book[]> {
    return this.http.get<Book[]>('http://localhost:3020/bookList').pipe(
      tap((data: any) => console.log('Data Fetched:' + JSON.stringify(data))),
      catchError(this.handleError));
  }
...
}
```

<u>Error handling</u>

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

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

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse, HttpHeaders } from '@angular/common/http';
import { catchError, tap } from 'rxjs/operators';
import { Observable, throwError } from 'rxjs';
import { HttpErrorResponse } from '@angular/common/http';
import { Book } from './book';
@Injectable({
  providedIn: 'root'
})
export class BookService {
 ...
 private handleError(err: HttpErrorResponse): Observable<any> {
   let errMsg = '';
   if (err.error instanceof Error) {
     // A client-side or network error occurred. Handle it accordingly.
     console.log('An error occurred:', err.error.message);
     errMsg = err.error.message;
   } else {
     // The backend returned an unsuccessful response code.
     // The response body may contain clues as to what went wrong,
     console.log(`Backend returned code ${err.status}`);
     errMsg = err.error.status;
   }
   return throwError(()=>errMsg);
 }
}
```

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

```
...
export class BookComponent implements OnInit {
 books!: Book[];
 errorMessage!: string;
 constructor(private bookService: BookService) { }
 getBooks() {
  this.bookService.getBooks().subscribe({
     next:  books => this.books = books,
     error:error => this.errorMessage = <any>error
   })
 }
 ngOnInit() {
   this.getBooks();
 }
}
```

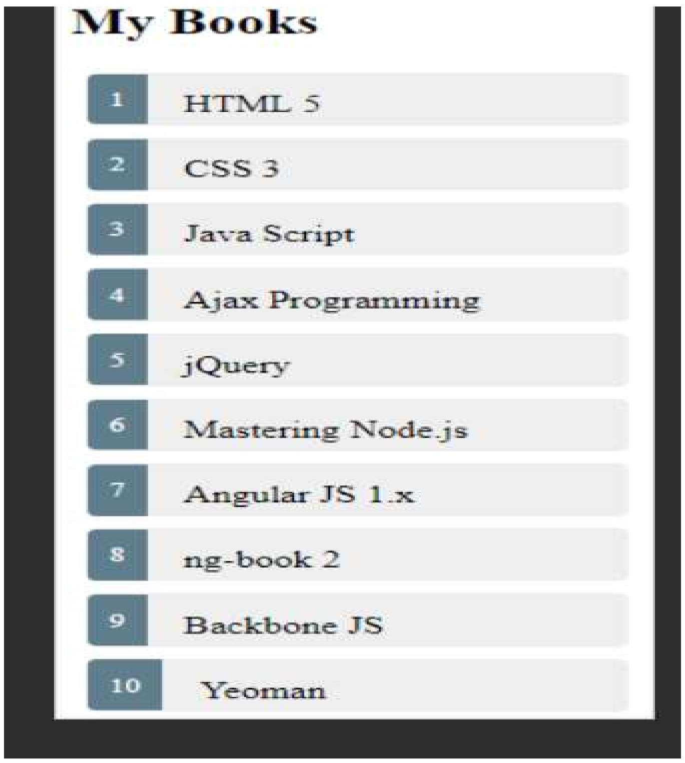
An observable in Angular begins to publish values only when someone has subscribed to it. To retrieve the value contained in the Observable returned by getBooks() of service, subscribe to the observable by calling the subscribe() method and pass an observer object which can listen to the three types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

Notification Type	Details
next	Required. Handler for each delivered value. Gets called zero or more times once execution starts.
error	Optional. Handler for handling an error notification. If an error occurs, it stops the execution of the observable instance.
complete	Optional. Handler for handling the execution-completion notification. If any values have been delayed, those can be still delivered to the next handler even after execution is complete.

book.component.html

```
...
<ul class="books">
  <li *ngFor="let book of books">
    <span class="badge">{{book.id}}</span> {{book.name}}
  </li>
</ul>
<div class="error" *ngIf="errorMessage">{{errorMessage}}</div>
```

Output:



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

Making a POST request:

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

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

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

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError, tap } from 'rxjs/operators';
import { Book } from './book';
@Injectable({
   providedIn: 'root'
})
export class BookService {
  constructor(private http: HttpClient) { }
  ...
  addBook(book: Book): Observable<any> {
    const options = new HttpHeaders({ 'Content-Type': 'application/json' });
    return this.http.post('http://localhost:3020/addBook', book, { headers: options }).pipe(
      catchError(this.handleError));
  }
...
}
```

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

```
...
export class BookComponent implements OnInit {
  books!: Book[];
  errorMessage!: string;
  constructor(private bookService: BookService) { }
  getBooks() {
    this.bookService.getBooks().subscribe({
      next:  books => this.books = books,
      error:error => this.errorMessage = <any>error
    })
  }
  addBook(bookId: string, name: string): void {
    let id=parseInt(bookId)
    this.bookService.addBook({id, name })
      .subscribe({next:(book: any) => this.books.push(book)});
  }
  ngOnInit(): void {
    this.getBooks();
  }
}
```

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

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

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError, tap } from 'rxjs/operators';
import { Book } from './book';
@Injectable({
   providedIn: 'root'
})
export class BookService {
  constructor(private http: HttpClient) { }
  ...
  updateBook(book: Book): Observable<any> {
    const options = new HttpHeaders({ 'Content-Type': 'application/json' });
    return this.http.put<any>('http://localhost:3020/update', book, { headers: options }).pipe(
      tap((_: any) => console.log(`updated hero id=${book.id}`)),
      catchError(this.handleError)
    );
  }
...
}
```

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

```
...
export class BookComponent implements OnInit {
  books!: Book[];
  errorMessage!: string;
  constructor(private bookService: BookService) { }
  getBooks() {
    this.bookService.getBooks().subscribe({
      next:  books => this.books = books,
      error:error => this.errorMessage = <any>error
    })
  }
  addBook(bookId: string, name: string): void {
    let id=parseInt(bookId)
    this.bookService.addBook({id, name })
      .subscribe({next:(book: any) => this.books.push(book)});
  }
  updateBook(bookId: string, name: string): void {
    let id=parseInt(bookId)
    this.bookService.updateBook({ id, name })
      .subscribe({next:(book: any) => this.books = book});
  }
  ngOnInit(): void {
    this.getBooks();
  }
}
```

Making a DELETE request :

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

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

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError, tap } from 'rxjs/operators';
import { Book } from './book';
@Injectable({
    providedIn: 'root'
})
export class BookService {
  constructor(private http: HttpClient) { }
  booksUrl = 'http://localhost:3020/bookList';
  ...
  deleteBook(bookId: number): Observable<any> {
    const url = `${this.booksUrl}/${bookId}`;
    return this.http.delete(url).pipe(
      catchError(this.handleError));
  }
...
}
```

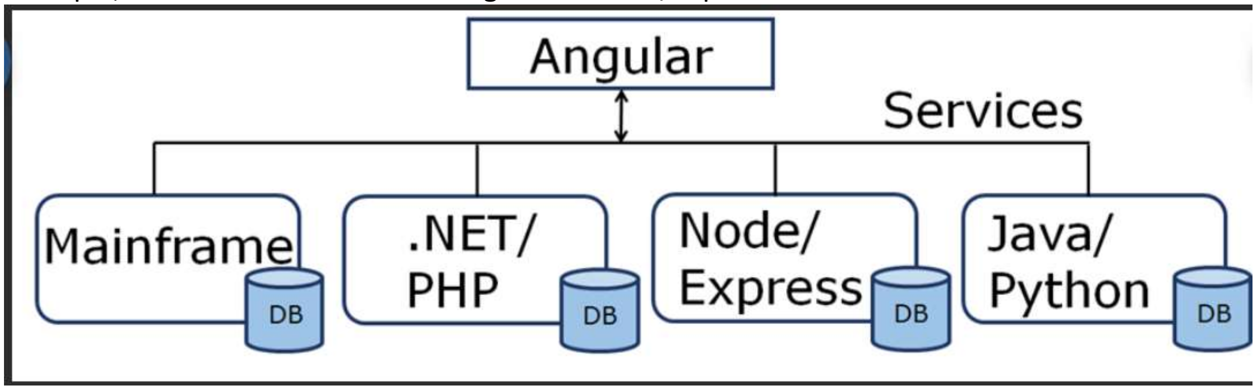
Modify the code in the **book.component.ts** file as shown below

```
...
export class BookComponent implements OnInit {
  books!: Book[];
  errorMessage!: string;
  constructor(private bookService: BookService) { }
  getBooks() {
    this.bookService.getBooks().subscribe({
      next:  books => this.books = books,
      error:error => this.errorMessage = <any>error
    })
  }
  addBook(bookId: string, name: string): void {
    let id=parseInt(bookId)
    this.bookService.addBook({id, name })
      .subscribe({next:(book: any) => this.books.push(book)});
  }
  updateBook(bookId: string, name: string): void {
    let id=parseInt(bookId)
    this.bookService.updateBook({ id, name })
      .subscribe({next:(book: any) => this.books = book});
  }
  deleteBook(bookId: string): void {
    let id=parseInt(bookId)
    this.bookService.deleteBook(id)
      .subscribe({next:(book: any) => this.books = book});
  }
  ngOnInit(): void {
    this.getBooks();
  }
}
```

Communicating with different backend services using Angular HttpClient – Internal

Communicating with different backend services using Angular HttpClient

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



Routing Basics

Routing means navigation between multiple views on a single page.
Routing allows to express some aspects of the application's state in the URL. The full application can be built without changing the URL.

Why Routing?

Routing allows to:

- Navigate between the views
- Create modular applications

Configuring Router

Angular uses Component Router to implement routing
A <base> tag must be added to the head tag in the HTML page to tell the router where to start with.
<base href="/">
Angular component router belongs to @angular/router module. To make use of routing, Routes, RouterModule classes must be imported.
Configuration should be done for the routes and the router will look for a corresponding route when a browser URL is changed.
Routes is an array that contains all the route configurations. Then, this array should be passed to the RouterModule.forRoot() function in the application bootstrapping function

Example:

Consider the example used in the services concept.
Add the following in the **app-routing.module.ts** which is created under the app folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { BookComponent } from './book/book.component';
import { DashboardComponent } from './dashboard/dashboard.component';
import { BookDetailComponent } from './book-detail/book-detail.component';
import { AppRoutingModule } from './app-routing.module';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
@NgModule({
  imports: [BrowserModule, HttpClientModule, FormsModule, AppRoutingModule],
  declarations: [AppComponent, BookComponent, DashboardComponent, BookDetailComponent,
PageNotFoundComponent],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

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

<u>Hyperlink based navigation</u>

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

app.component.ts

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

<u>Programmatical navigation</u>

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

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

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

Why Route Parameters?

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

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

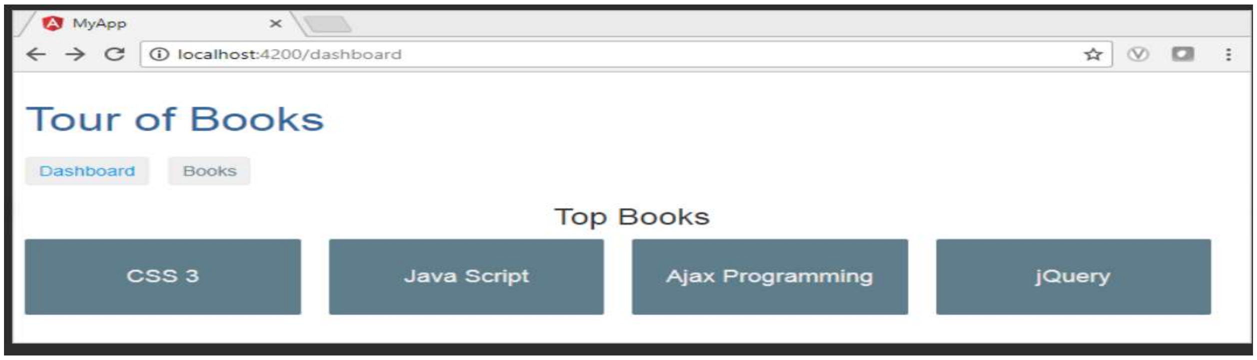
Passing Route Parameters
Generate a component named dashboard using the following CLI command. This will display list of all the books available.

```
D:\MyApp>ng generate component dashboard
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { Book } from '../book/book';
import { BookService } from '../book/book.service';
@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.css']
})
export class DashboardComponent implements OnInit {
  books: Book[] = [];
  constructor(
    private router: Router,
    private bookService: BookService) { }
  ngOnInit(): void {
    this.bookService.getBooks()
      .subscribe({next:books => this.books = books.slice(1, 5)});
  }
  gotoDetail(book: Book): void {
    this.router.navigate(['/detail', book.id]);
  }
}
```

Accessing Route Parameters
To access route parameters, use ActivatedRoute class
Generate BookDetail component using the following CLI command

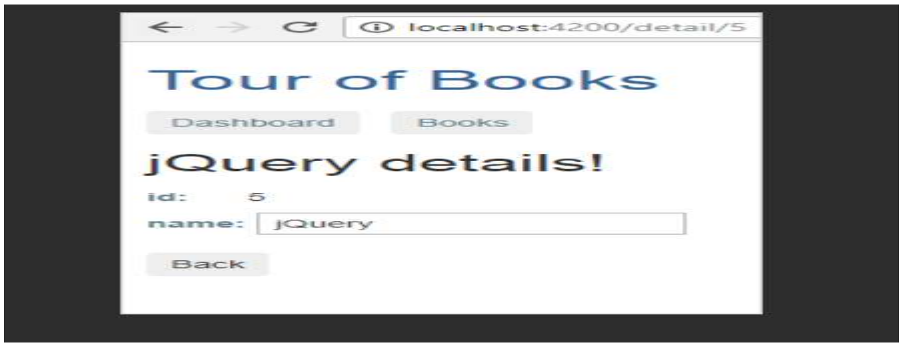
```
D:\MyApp>ng generate component BookDetail
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Book } from '../book/book';
import { BookService } from '../book/book.service';
@Component({
  selector: 'app-book-detail',
  templateUrl: './book-detail.component.html',
  styleUrls: ['./book-detail.component.css'],
})
```

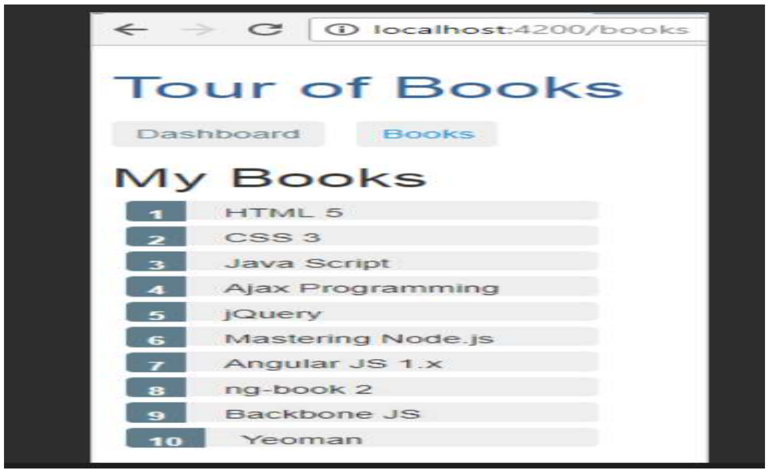
```
export class BookDetailComponent implements OnInit {
 book!: Book;
 error!: any;
 constructor(
  private bookService: BookService,
  private route: ActivatedRoute
 ) { }
 ngOnInit() {
  this.route.paramsMap.subscribe(params => {
   this.bookService.getBook(params.get('id')).subscribe((book) => {
    this.book = book ?? this.book;
   });
  });
 }
 goBack() {
  window.history.back();
 }
}
```
Output:
On load of the application, the Dashboard component is shown according to the route configuration mentioned in the app-routing.module.ts.



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

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





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

Route Guards

In the Angular application, users can navigate to any URL directly. That's not the right thing to do always.
Consider the following scenarios
Users must login first to access a component
The user is not authorized to access a component
User should fetch data before displaying a component
Pending changes should be saved before leaving a component

These scenarios must be handled through route guards.
A guard's return value controls the behavior of the router
If it returns true, the navigation process continues
If it returns false, the navigation process stops

There are 2 ways using which route guards can be implemented.
1. Class based approach
2. Functional approach

1.Class based approach
Angular has canActivate interface which can be used to check if a user is logged in to access a component
canActivate() method must be overridden in the guard class as shown below:

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

```
class GuardService implements CanActivate{
  canActivate( ): boolean {
  }
}
```

2.Functional approach
Functional route guards are more effective, and to convert a class-based guard into functional guard, use the "inject" function, as illustrated in the example below. This method offers dependency injection within a functional context and has been available since Angular v14.
Using functional route guards, we can build a reusable and modular function that can be applied directly to the canActivate and canActivateChild functions of Angular Router.

```
import { inject} from '@angular/core';
import { Router } from '@angular/router';
import { LoginService } from './login.service';
export const authGuard=()=>{
const loginService=inject(LoginService)
const router=inject(Router)
if (loginService.isUserLoggedIn()) {
            return true;
        }
   router.navigate(['/login']);
   return false;
}
const appRoutes: Routes = [
   { path: 'dashboard', component: DashboardComponent },
   { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
   { path: 'books', component: BookComponent , canActivate:[authGuard] }, //functional route guard
   { path: 'detail/:id', component: BookDetailComponent },
   {path: 'login',component:LoginComponent},
   { path: '**', component: PageNotFoundComponent },
];
```

Asynchronous Routing:

When an Angular application has a lot of components, it will increase the size of the application. In such cases, the application takes a lot of time to load.
To overcome this problem, asynchronous routing is preferred, i.e, modules must be loaded lazily only when they are required instead of loading them at the beginning of the execution
Lazy Loading has the following benefits:
Modules are loaded only when the user requests for it
Load time can be speeded up for users who will be visiting only certain areas of the application

Lazy Loading Route Configuration:

To apply lazy loading on modules, create a separate routing configuration file for that module and map an empty path to the component of that module.
Considering an example in the previous concept, consider BookComponent. To load it lazily, create the **book-routing.module.ts** file inside book folder and map an empty path to BookComponent(Line 8-10)

```
1.  import { NgModule } from '@angular/core';
2.  import { RouterModule, Routes } from '@angular/router';
3.  import { BookComponent } from './book.component';
4.  import { LoginGuardService } from '../login/login-guard.service';
5.
6.  const bookRoutes: Routes = [
7.  {
8.  path: '',
9.  component: BookComponent,
10. canActivate: [LoginGuardService]
11. }
12. ];
```

```

@NgModule({
imports: [RouterModule.forChild(bookRoutes)],
exports: [RouterModule]
})
export class BookRoutingModule { }

```

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

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

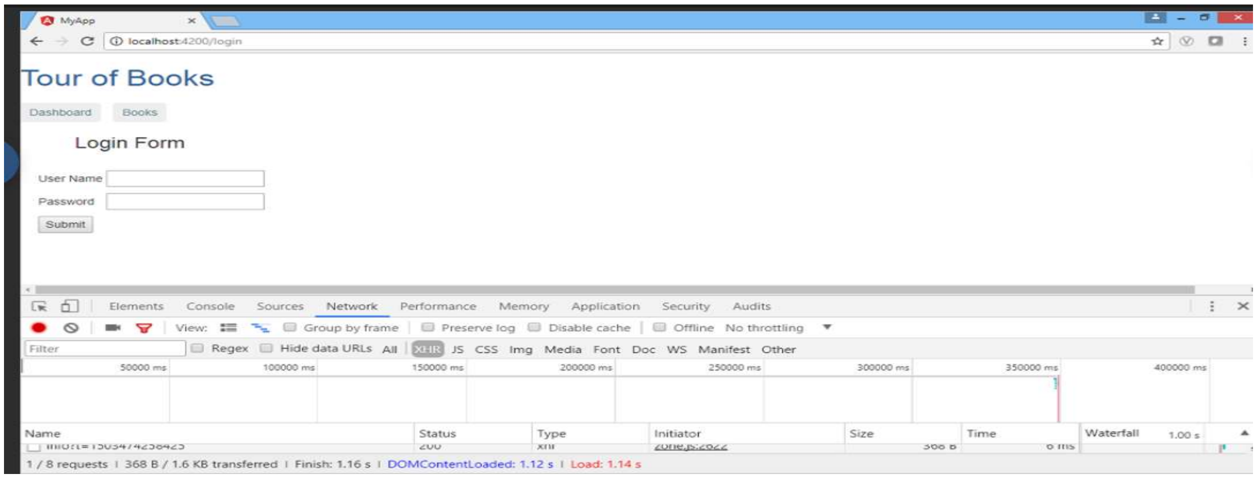
```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { BookComponent } from './book.component';
import { BookRoutingModule } from './book-routing.module';
@NgModule({
  imports: [CommonModule, BookRoutingModule],
  declarations: [BookComponent]
})
export class BookModule { }
```

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

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

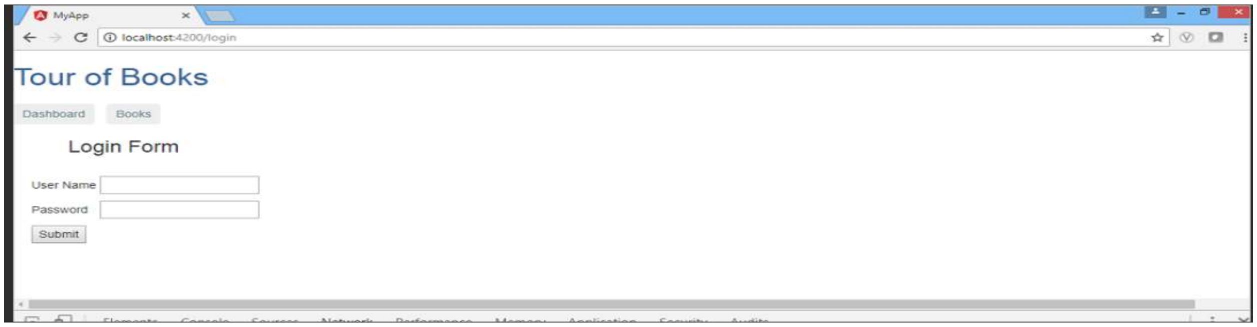
Finally, it loads the requested route to the destination book component.

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



1 / 8 requests | 368 B / 1.6 KB transferred | Finish: 1.16 s | DOMContentLoaded: 1.12 s | Load: 1.14 s

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



Nested Routes

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

Steps to create child routes using Angular:

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

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { BookComponent } from './book.component';
import { authGuard} from '../login/login-guard.service';
import { DashboardComponent } from '../dashboard/dashboard.component';
import { BookDetailComponent } from '../book-detail/book-detail.component';
const bookRoutes: Routes = [
  {
    path: '',
    component: BookComponent,
    children: [
      { path: 'dashboard', component: DashboardComponent },
      { path: 'detail/:id', component: BookDetailComponent }
    ],
    canActivate: [authGuard]
  }];
@NgModule({
  imports: [RouterModule.forChild(bookRoutes)],
  exports: [RouterModule]
})
export class BookRoutingModule { }
```

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

```
import { NgModule } from '@angular/core';
import { BookComponent } from './book.component';
import { BookRoutingModule } from './book-routing.module';
import { FormsModule } from '@angular/forms';
import { BookDetailComponent } from '../book-detail/book-detail.component';
import { DashboardComponent } from '../dashboard/dashboard.component';
import { CommonModule } from '@angular/common';
@NgModule({
  imports: [ CommonModule, BookRoutingModule, FormsModule],
  declarations: [BookComponent, BookDetailComponent, DashboardComponent]
})
export class BookModule { }
```

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

```
<br/>
        <h2>MyBooks</h2>
        <ul class="books">
          <li *ngFor="let book of books " (click)="gotoDetail(book)">
                <span class="badge">{{book.id}}</span> {{book.name}}
          </li>
        </ul>
        <div>
          <router-outlet></router-outlet>
        </div>
        <div class="error" *ngIf="errorMessage">{{errorMessage}}</div>
```

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

```
<h1>{{title}}</h1>
<nav>
    <a [routerLink]='["/books"]' routerLinkActive="active">Books</a>
</nav>
<router-outlet></router-outlet>
```

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

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

6. Update app.module.ts as below:

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

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

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

8. Update dashboard.component.html

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

```
</div>
```

9. Update dashboard.component.ts

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