Deloitte.

Angular

Deloitte Technology Academy (DTA)

Agenda

Topics	Descriptions	Duration
Introduction	What Is Angular?, Central Features of the Angular Framework, Scope and Goal of Angular, Angular vs. AngularJS, Installing and Using Angular, Adding Angular and Dependencies to Your App, Building Blocks, Angular Application, and a Basic Angular Application	18 hours
Components	What Is a Component?, An Example Component, Component Starter, Developing a Simple Login Component, Component Decorator Properties, Component Lifecycle Hooks, and Using Lifecycle Hook	
Data and Event Binding	Binding Syntax, One-Way Output Binding, Binding Displayed Output Values, Two-Way Binding of Input Fields, Input Binding Examples, Binding Events, and Setting Element Properties	
Directives	What Are Directives?	
Structural Directives and Attribute Directives	Structural Directives, Apply Styles by Changing Classes, Changing Classes—Example, Applying Styles Directly, Directives and Property Binding, Controlling Element Visibility, and Setting Image Source Dynamically Adding and Removing Elements Dynamically, Creating Tables/Lists with ngFor, ngSwitch, and ngIf/else, Attribute Directives	

Agenda (Cont.)

Topics	Descriptions	Duration
Service and Dependency Injection	What Is a Service?, Creating a Service, Dependency Injection, Injecting Services, Using a Service in a Component: Dedicated Instance, Using onInit to Initialize Component Data, and Using a Shared Service Instance	18 hours
Pipes and Data Formatting	What Are Pipes?, Formatting Changes in Angular, Using a Built-in Pipe, Using Pipes in Hyper Text Markup Language (HTML), Chaining Pipes, Decimal Pipe, Currency Pipe, and Custom Pipes	
Forms	Template Driven Forms, A Basic Angular Form, Binding Input Fields, Accessing the Form Object, Binding the Form Submit Event, Angular Validations, Displaying Form Validation State, Displaying Field Validation State, Displaying Validation State Using Classes, Disabling Submit when Form is Invalid, Submitting the Form, Binding to Object Variables, and Reactive Forms	

Agenda

</>

Topic	Descriptions	Duration
HTTP Client	The Angular HTTP Client, Using the HTTP Client—Overview, Setting up the Root Component, Service Using HTTP Client and Interceptors, Importing Individual HTTP Providers Into Services, Service Imports, The Observable Object Type, Making an HTTP GET Call, Using the Service in a Component, Importing Observable Methods, Enhancing the Service with .map() and .catch(), GET Request	20 hours
Single Page Application and Angular Routing	Routing and Navigation, The Component Router, Traditional Browser Navigation, and Component Router Terminology, Setting up the Component Router, Configuring Routes, Bootstrapping Routing, and Programmatic Navigation, Creating Routes With Route Parameters, Navigating With Route Parameters, and Using Route Parameter Values, Retrieving the Route Parameter, Query Parameters, and Guards at Gate/Authentication	
Modules	Lazy Loading of Module, Exporting Modules and Exporting Services	

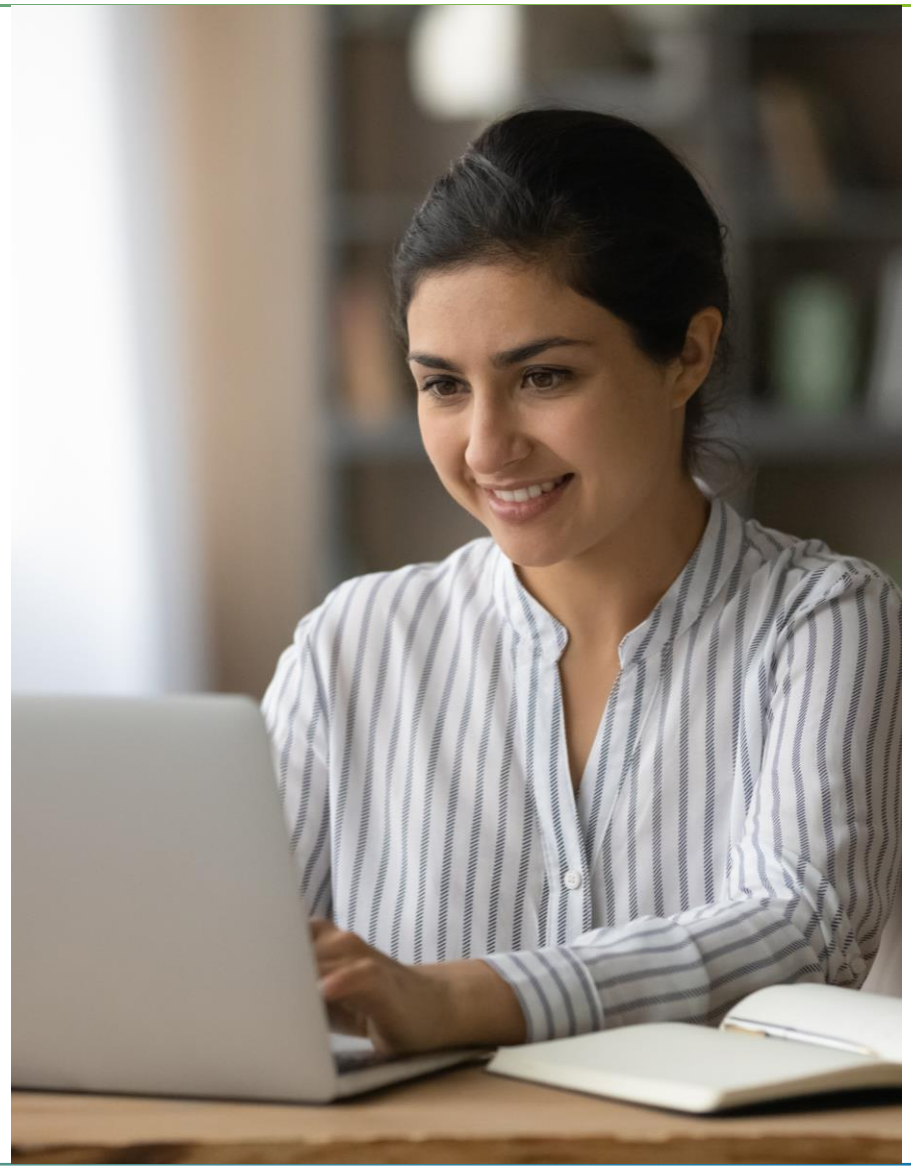
Learning Objectives

< / >

By the end of this session, you will be able to:

- Explain the basic Angular concepts
- Use components, event binding, and forms to build an application in Angular
- Use structural directives, attribute directives, and property bindings
- Create services in Angular
- Explain dependency injections in Angular
- Use HTTP client using root component and interceptors
- Explain observable object type and GET Request
- Explain routing and navigation in Angular
- Set up a component router, bootstrapping routing and programmatic navigation

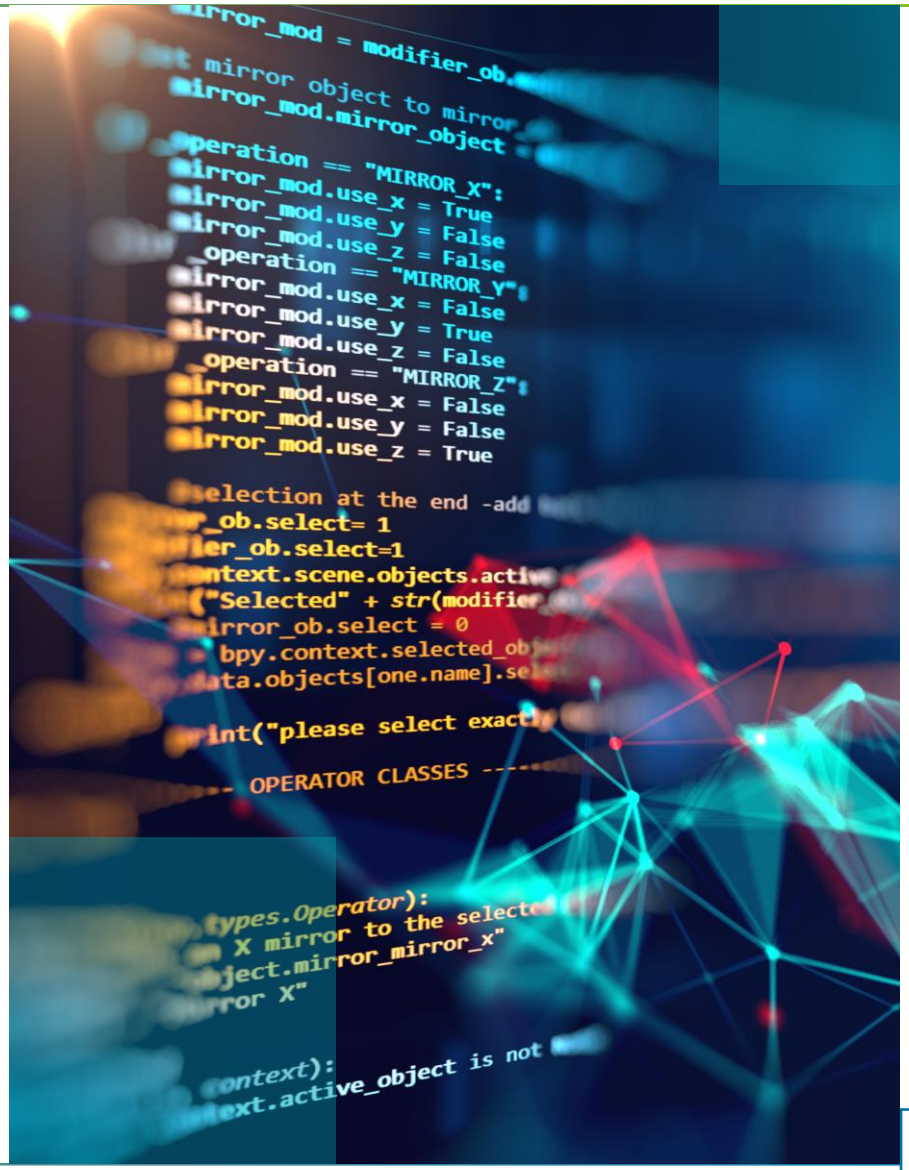
Introduction



Introduction to Angular

What is Angular?

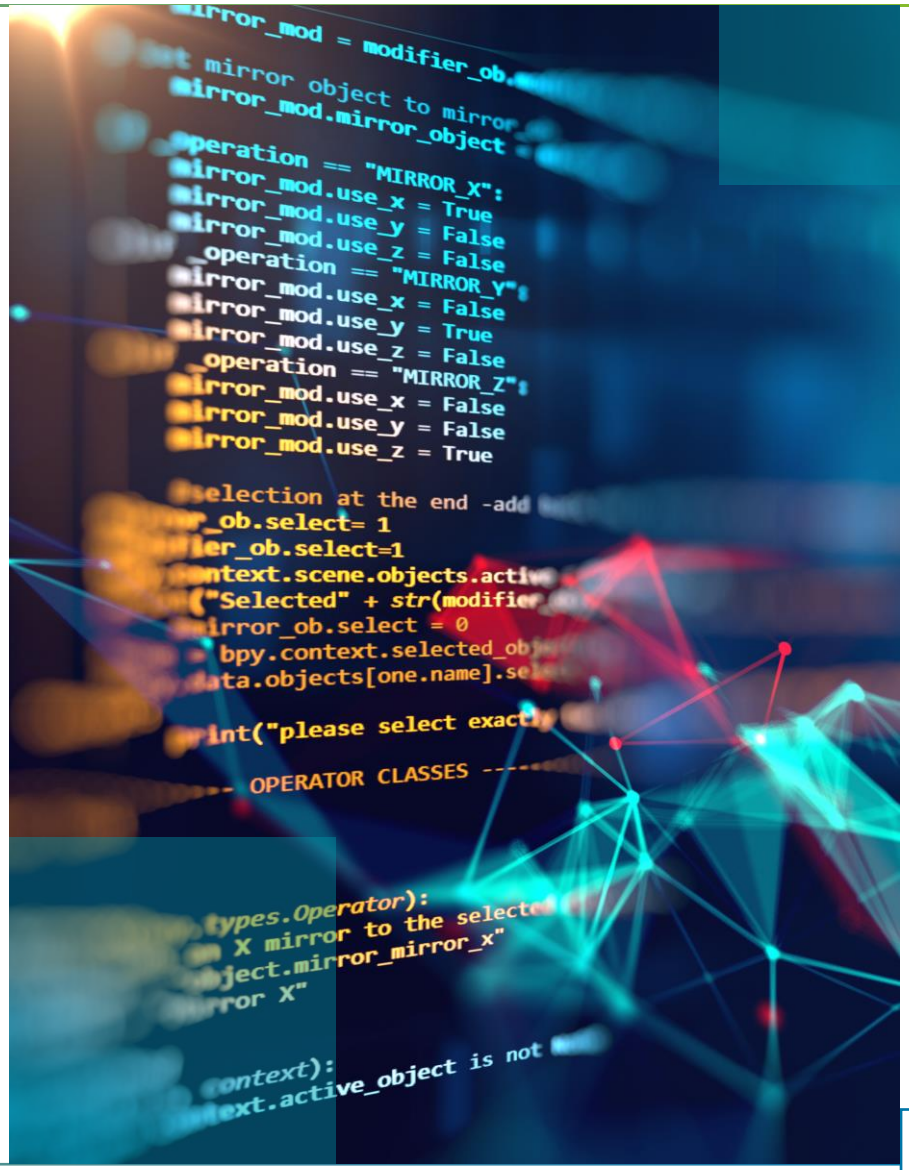
- Angular is a JavaScript (TypeScript) framework that makes you able to create reactive Single Page Applications (SPAs).
- Angular is completely based on components. A tree structure is formed with child and parent components from several components.
- 2+ version and beyond of Angular are generally known as Angular. Angular 1.0 was the very first version and was referred to as AngularJS
- The team that created AngularJS completely rewrote it to create Angular.
- Angular 14 is the latest version released into the market as on Sep 2022



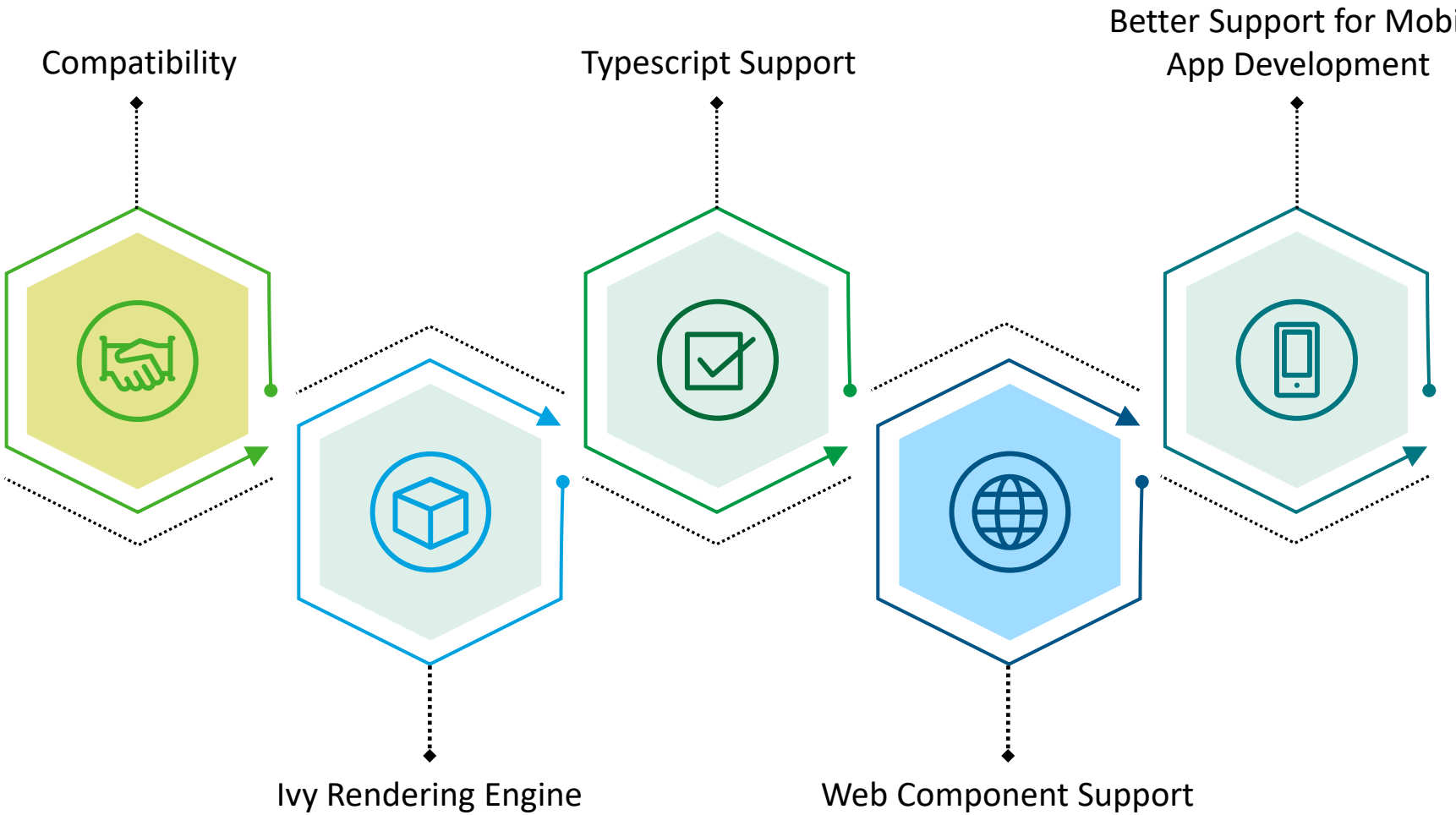
Introduction to Angular (Cont.)

What is SPA?

- A single-page application is a web application or a website that provides users with a very fluid, reactive, and fast experience.
- A menu, buttons, and blocks are contained on a single page.
- When a user selects any of them the current page is dynamically rewritten instead of entire new pages loading from a server.
- This is how it has the ability to have such reactive quick speed.



Features of Angular



Features of Angular (Cont.)

Compatibility	Angular ensures you to easily migrate your old Angular apps to a newer Angular version without any trouble. You can easily make seamless transitions from old Angular versions to modern ones.
Ivy Rendering Engine	Ivy is the Angular engine, which ensures better debugging, quicker compilations, smaller bundles, and dynamic loading of modules.
Typescript Support	The use of typescript improves productivity significantly.
Web Component Support	The use of components helps in creating loosely coupled units of application that can be developed and maintained easily.
Better Support for Mobile App Development	Angular addresses the concern of both mobile and desktop applications.

Why Angular?

Automatic synchronization with two-way data binding	The data between model and view is seamlessly synchronized.
Code consistency and reusability	Angular command-line interface (CLI) tool allows us to keep the entire team on the same page while enabling us to create initial projects, perform tests and integrate diverse features in the same project.
Supported by Google	Angular is supported by google which is one of the biggest firms in technology and provides long-term support for Angular to scale up application development.
Declarative UI	Angular is easy to learn and understand, offers swift compilation and has a wide range of tutorials, documentation, and support for newcomers. The Angular framework uses HTML that when compared to JavaScript is a less complex language.

Scope and Goal of Angular

< / >

Scope	Goal
In the future we are expecting that there will be significant growth in the field of web applications and web development. Currently, Angular is used predominantly by developers.	The primary purpose of Angular is to develop single-page applications.

Setting up Angular Environment

</>

Install Visual Studio Code	https://code.visualstudio.com
Install Node JS	https://nodejs.org/en/download/

Commands:	
To Install Angular CLI	`npm install -g @angular/cli`
To Install Typescript	`npm install –g typescript`
To Create New Angular Project	`ng new <project-Name>`
To Start the Angular Application	`ng serve` `ng serve - - port 4400`

package.json

< / >

To run an application, the packages listed under the dependencies section of package.json are important.

package.json: This is a npm configuration file that includes all the dependencies. The package.json file tells which libraries will be installed into node_modules when you run npm install.

The dependencies section of package.json contains:

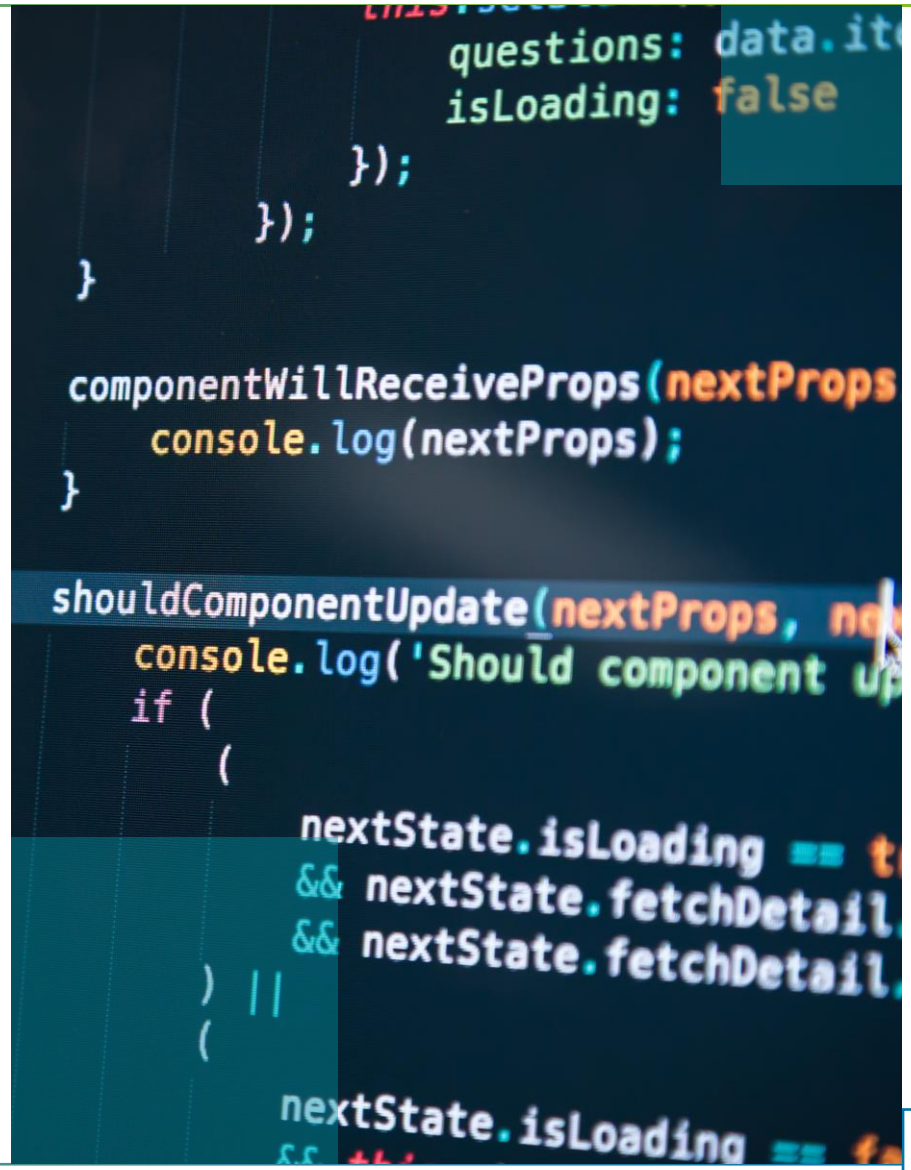
Angular Packages	Support Packages	Polyfill Packages
This file identifies which Angular module is to be loaded when the application starts	Third party libraries that must be present for Angular applications to run	Gaps in a browser's JavaScript implementation are plugged by Polyfills.

The ng add command can be used to add a new dependency.

Building Blocks

</ >

- Angular framework's basic building blocks are NgModules, which are organized Angular components. Related code are collected into functional sets by NgModules; a set of NgModules define an Angular application.
- An application always has at least a root module that enables bootstrapping and typically has many more feature modules.
- An application typically has many features modules, but it always has at least a root module that enables bootstrapping.
- A module in Angular is a class with @NgModule decorator added to it. @NgModule metadata will contain the declarations of components, pipes, directives, and services that are to be used across the application.



A Basic Angular Application

</>

- Create a folder for your application in the desired location on your system and open it on VSCode. Open Cmd and enter the command: `ng create hello-world`
- The Ng command can be used to run the application after you change the directory to the folder created.
  ```
  cd hello-world
  ng serve
  ```
- At run time the ng serve command dynamically injects all the application code, inline templates, and styles into the index.html file.

main.ts	This file identifies which Angular module is to be loaded when the application starts.
app.module.ts	Source code specific to your application starts here. This file can be thought of as the core configuration of the application, from all the necessary and relevant dependencies being loaded, to which components will be used within your application being declared, and marking which is the main entry point component of your application.
app.component.ts	The functionality of the application is driven by the actual Angular code is app.component.ts. An Angular component is nothing but a TypeScript class, decorated with some metadata and attributes. All the functionality and data of the component is encapsulated in the class, while the @Component decorator specifies how the data translates into the HTML.

A Basic Angular Application (Cont.)

</>

index.html

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

A Basic Angular Application (Cont.)

</>

main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {
  enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule)
.catch(err => console.error(err));
```

A Basic Angular Application (Cont.)

</>

app.module.ts

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

A Basic Angular Application (Cont.)

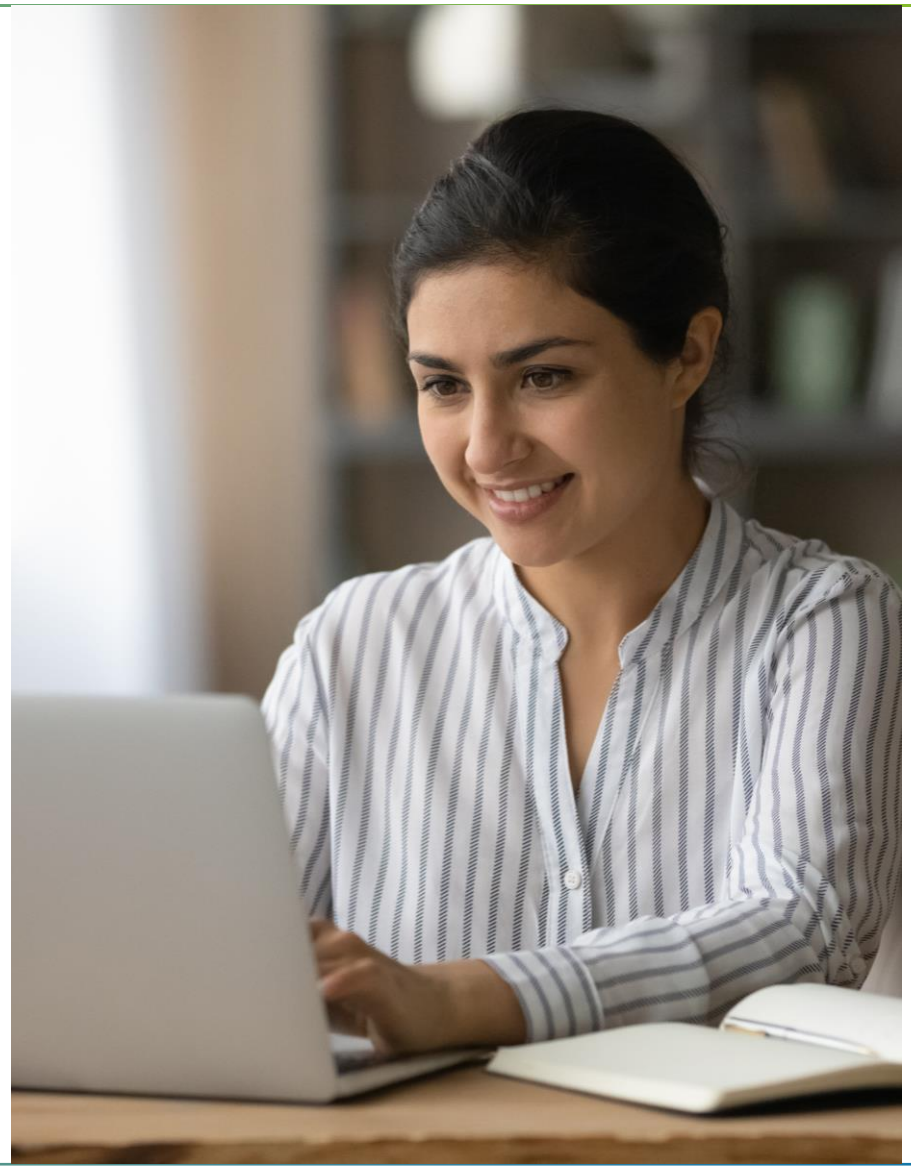
</>

app.component.ts

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

</ / >

Components



Why Components in Angular?

</ >

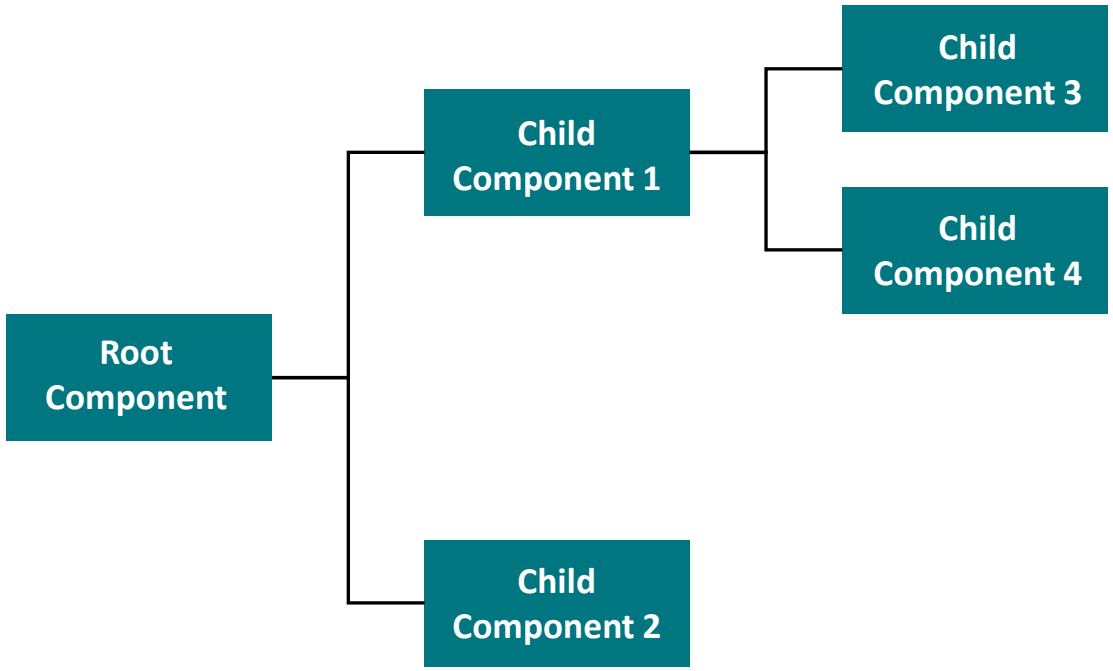
What Is a Component?

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

Why Components?

- Components are reusable.
- You can generate a component using the below command.

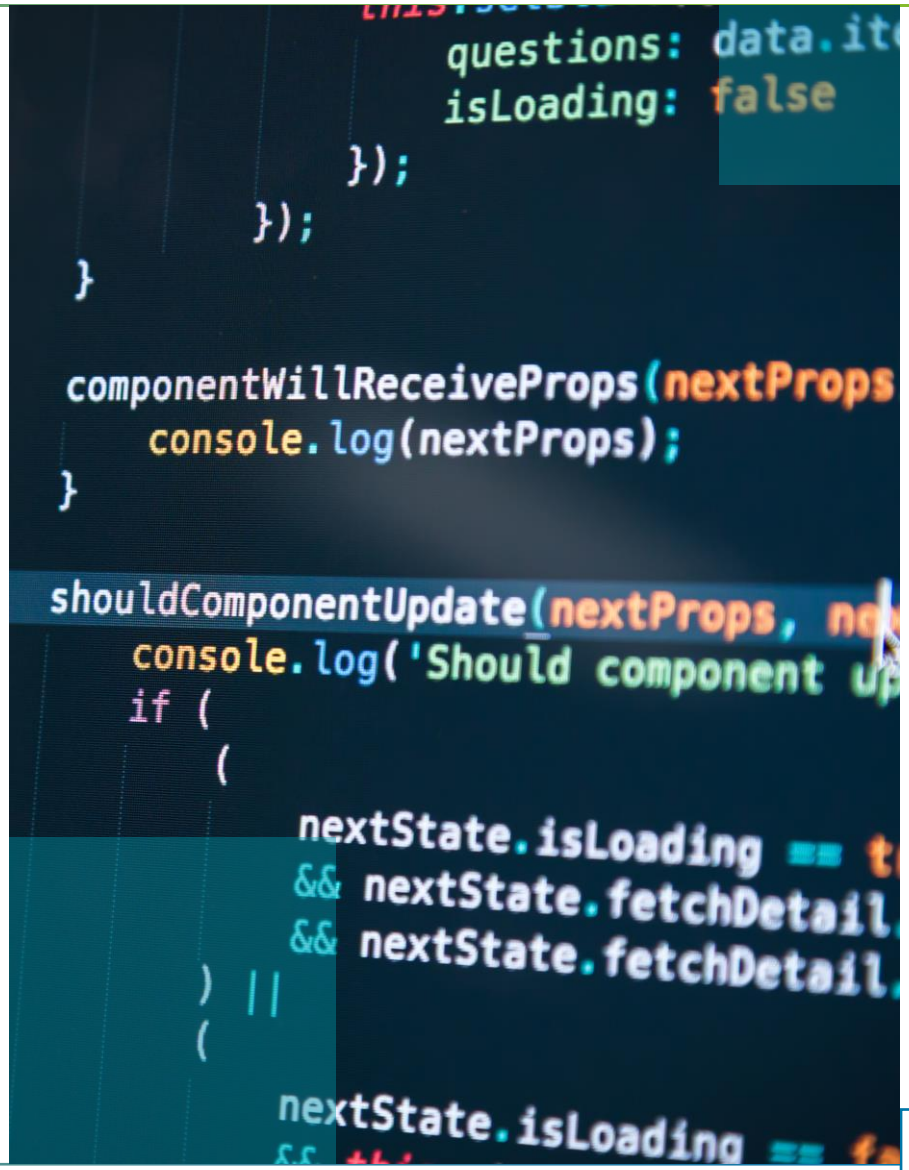
```
D:\MyApp>ng generate component coursesList
```



Component Decorator

</>

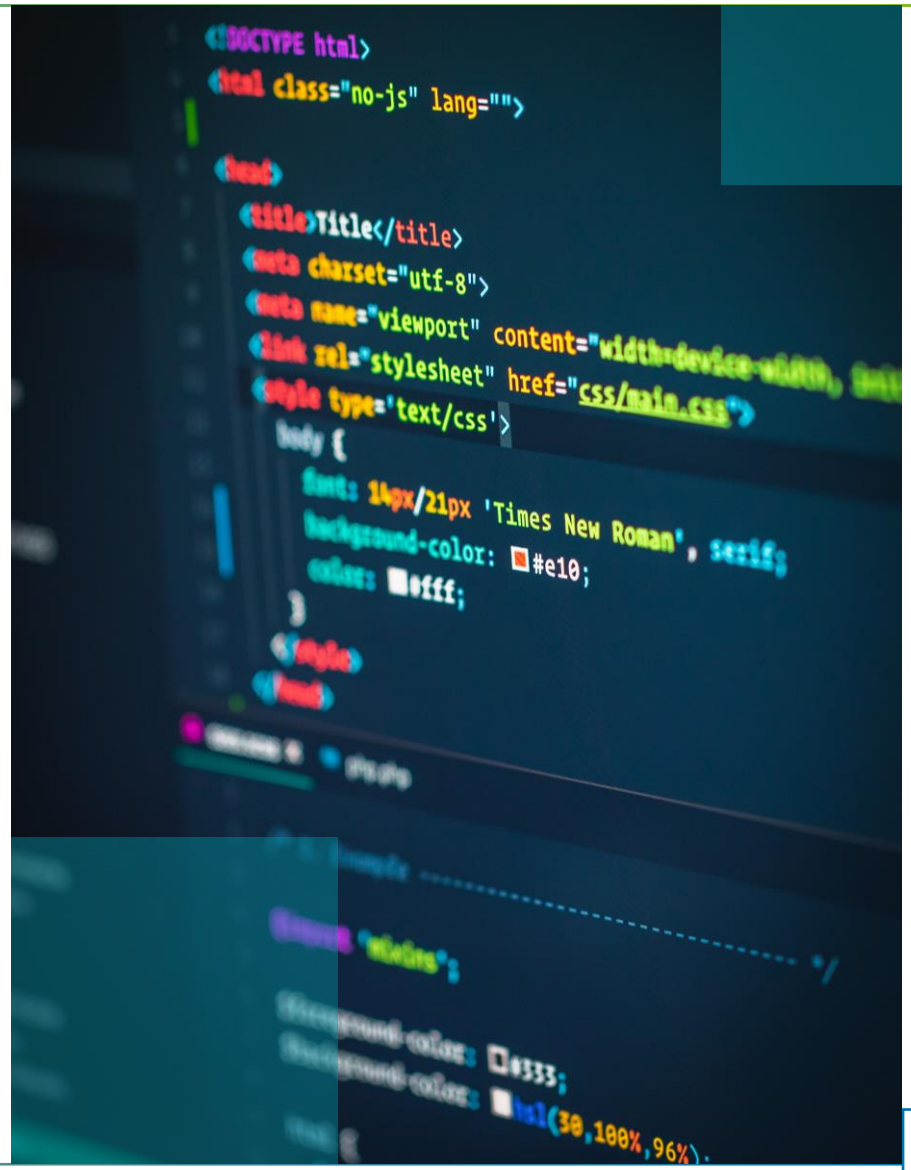
You can mark a class as an Angular component by using a component decorator. This allows you to provide additional metadata that determines how at runtime the component should be processed, instantiated, and used.



Component Starter

</>

- Angular CLI is used to create an Angular Component. Type the following command in the terminal,
  `ng g c component-name`
- A component-name folder will be created with the following files:
  - `component-name.component.ts`
  - `component-name.component.html`
  - `component-name.component.css`
- Open component-name.component.ts. The @Component() decorator indicates that the following class is a component. Metadata about the component is also provided by @Component(), including its selector, templates, and styles.



Component Metadata

</>

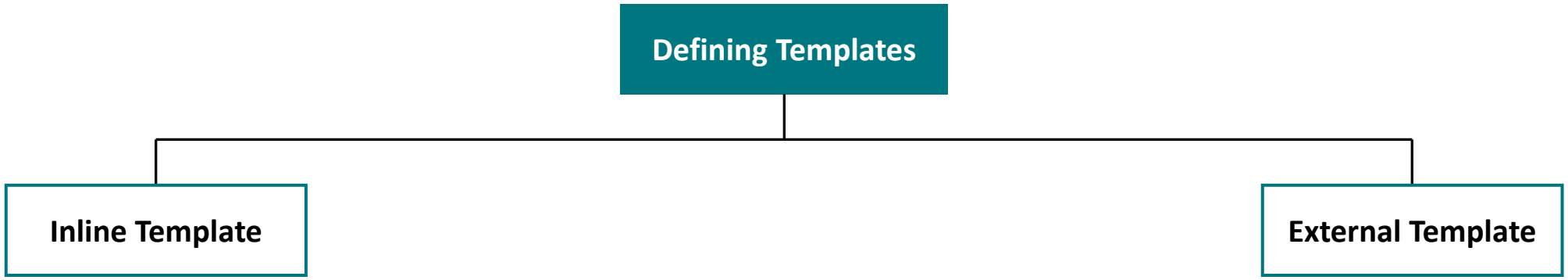
- The metadata for a component tells Angular where to get the major building blocks that it needs for the component to be to created and present and its view. In particular, it associates a template with the component, either directly with inline code, or by reference. Together, the component and its template describe a view.
- The @Component decorator identifies the class immediately below it as a component class and specifies its metadata.
- selector: this tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML.
- templateUrl: it is the module-relative address of this component's HTML template.
- styleUrls: it links the Cascading Style Sheets (CSS) classes with the component.

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

Templates

- Templates in Angular represent a view whose role is to display data and change the data whenever an event occurs. Its default language for templates is HTML.
- Template's separate view layer from the rest of the framework so we can change the view layer without breaking the application.

There are two ways of defining templates as shown below:



Inline Template

</>

We can create a template inline into the component class itself using the template property of @Component decorator.

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

@Component({
  selector: 'app-test',
  template: `
  <h1> Welcome </h1>
  <h2> User: {{ User }}</h2>
`,
  styleUrls: ['./test.component.css']
})
export class TestComponent implements OnInit
{
  User : string = "Deloitte";
  constructor() { }

  ngOnInit(): void {
  }
}
```

External Template

</>

By default, Angular CLI uses the external template.

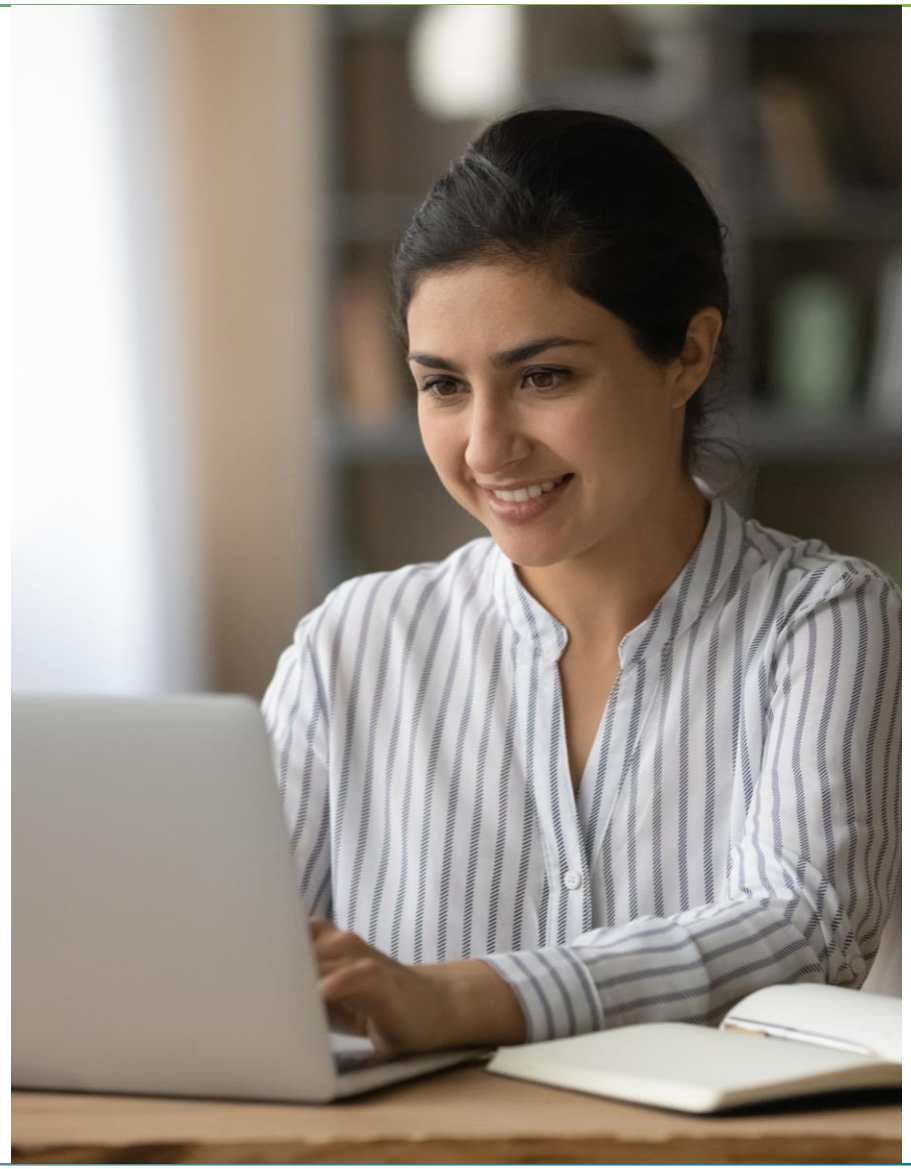
It binds the template with a component using the templateUrl option.

TemplateUrl is used in external format whereas in case of inline template, we use a template instead of templateUrl.

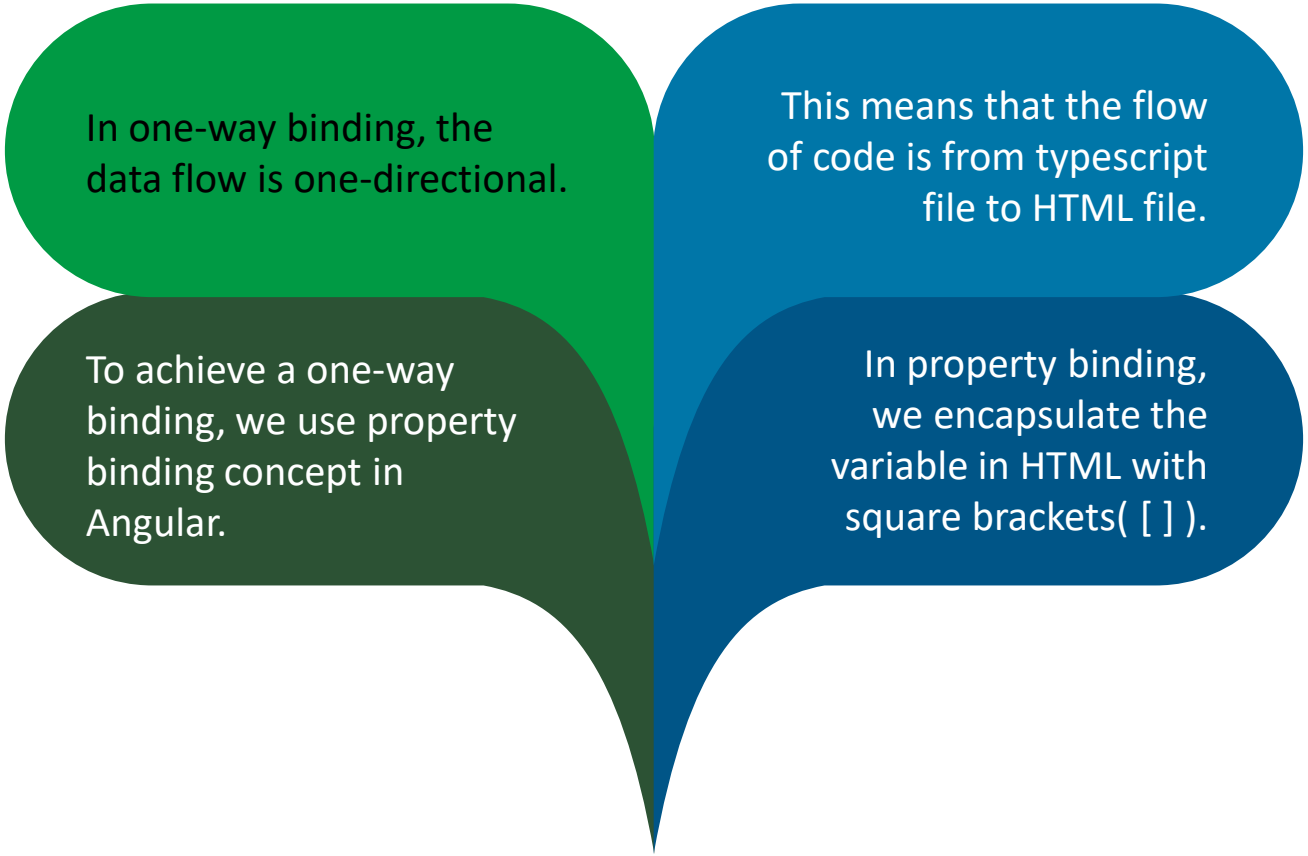
```
test.component.ts
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-test',
  templateUrl: `./test.component.html`,
  styleUrls: ['./test.component.css']
})
export class TestComponent implements OnInit {
  Text : string = "Deloitte";
  changeText() {
    this.Text = "Learning";
}
  constructor() { }
  ngOnInit(): void {
  }
}

test.component.html
<h1>Welcome</h1>
<h2>Text : {{Text}}</h2>
<p (click)="changeText()">Click here to change</p>
```

</ />

Data and Event Binding



One way binding



Binding Syntax

</>

Based on your application's state, your page stays up-to-date automatically using data binding. You use data binding to specify things such as the source of an image, the state of a button, or data for a particular user.

Angular provides three categories of data binding according to the direction of data flow:

- From source to view
- From view to source
- In a two-way sequence of view to the source to view

Types	Syntaxes	Categories
Interpolation	`{{expression}}`	One-way from data source to view target
Property Binding	`[target]="expression"`	One-way from data source to view target
Event	`(target)="statement"`	One-way from view target to data source
Two-way	`[(target)]="expression"`	Two-way

One-way Binding Example

</>

app.component.ts

```
import { Component } from "@angular/core";
@Component({
selector: "my-app",
templateUrl: "./app.component.html",
styleUrls: ["./app.component.css"],
})
export class AppComponent {
title = "Displaying the content with one way binding";
}
```

app.component.html

```
<h3>without one way binding</h3>
<hr />
<h3 [textContent]="title"></h3>
```

One-way Binding Example (Cont.)

</>

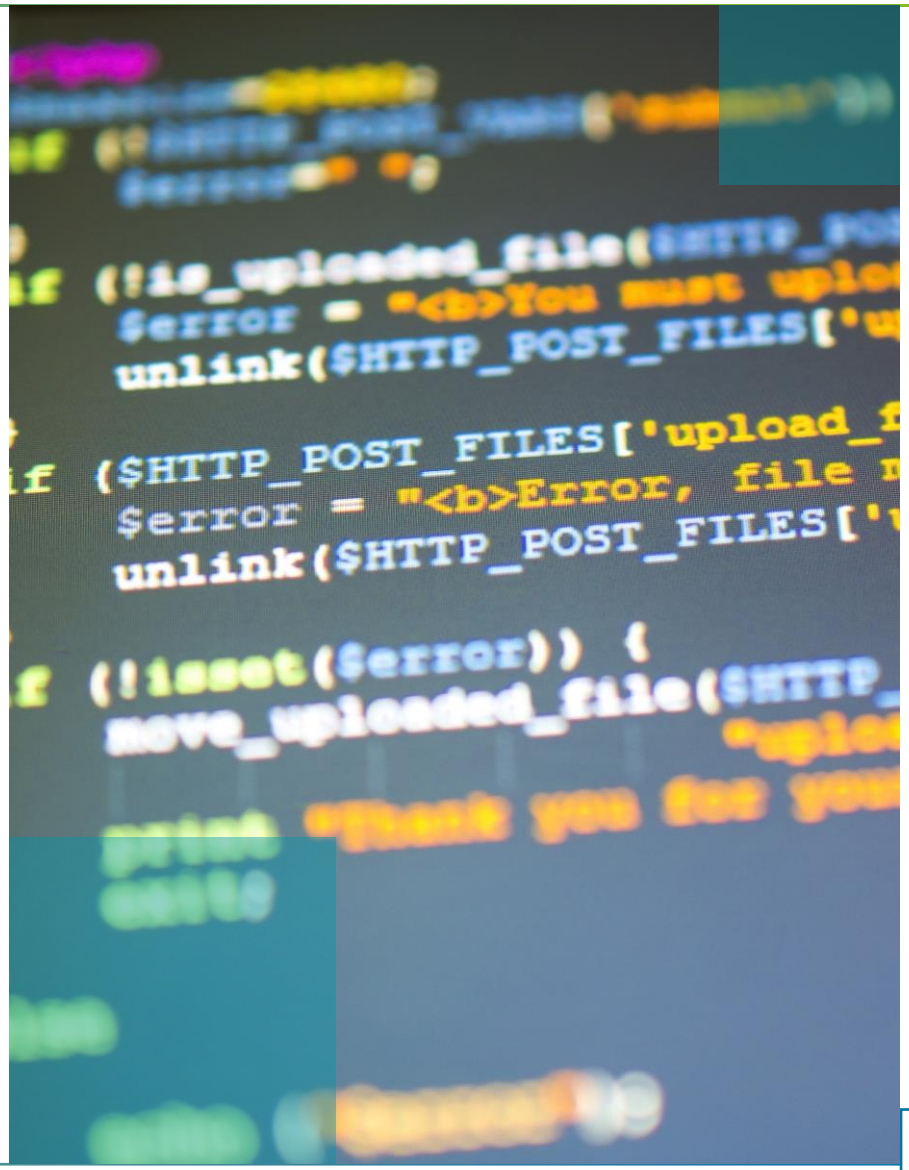
app.module.ts

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

import { AppComponent } from "./app.component";

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

Two-way Binding of Input Fields

</ />

- The data flow is bi-directional in a two-way binding.
- This means that the flow of code can go from either ts file to HTML file or HTML file to ts file.
- A ngModel in a box syntax will be used to achieve a two-way bindin
- You must import ‘FormsModule’ from ‘@angular/forms to ensure the app will not break.
- Any view changes are also propagated to the component class. Also, reflected in the view are any changes to the properties in the component class.
- Once the ngModel directive is declared and set equal to the name of the property, then two properties are bound in order to two-way binding works.



Two-way data binding Example

</>

app.component.ts

```
import { Component } from "@angular/core";
@Component({
selector: "my-app",
templateUrl: "./app.component.html",
})
export class AppComponent {
data = "data entered";
}
```

app.component.html

```
<input [(ngModel)]="data" type="text">
<hr>
<h3> Entered data is {{data}}</h3>
```

Two-way data binding Example (Cont.)

</>

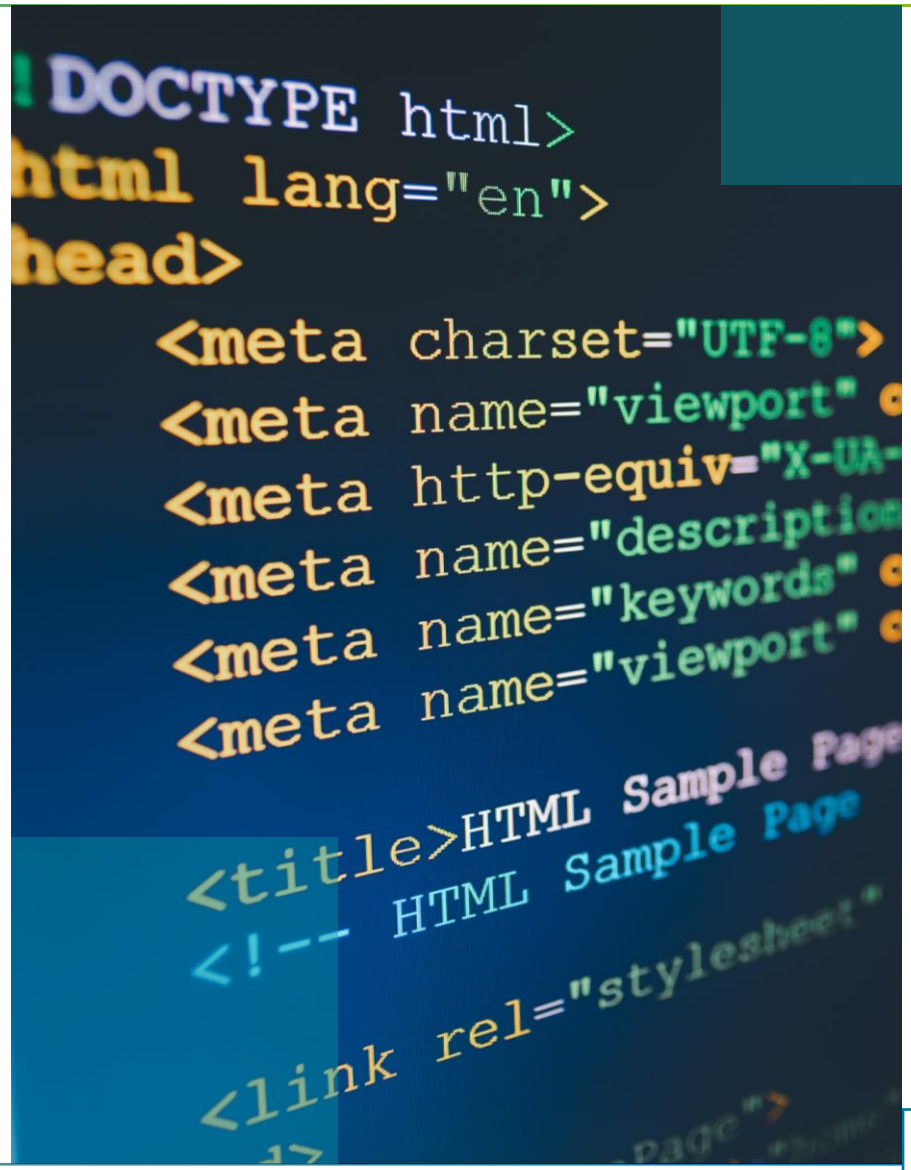
app.module.ts

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

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

Event Binding

</>

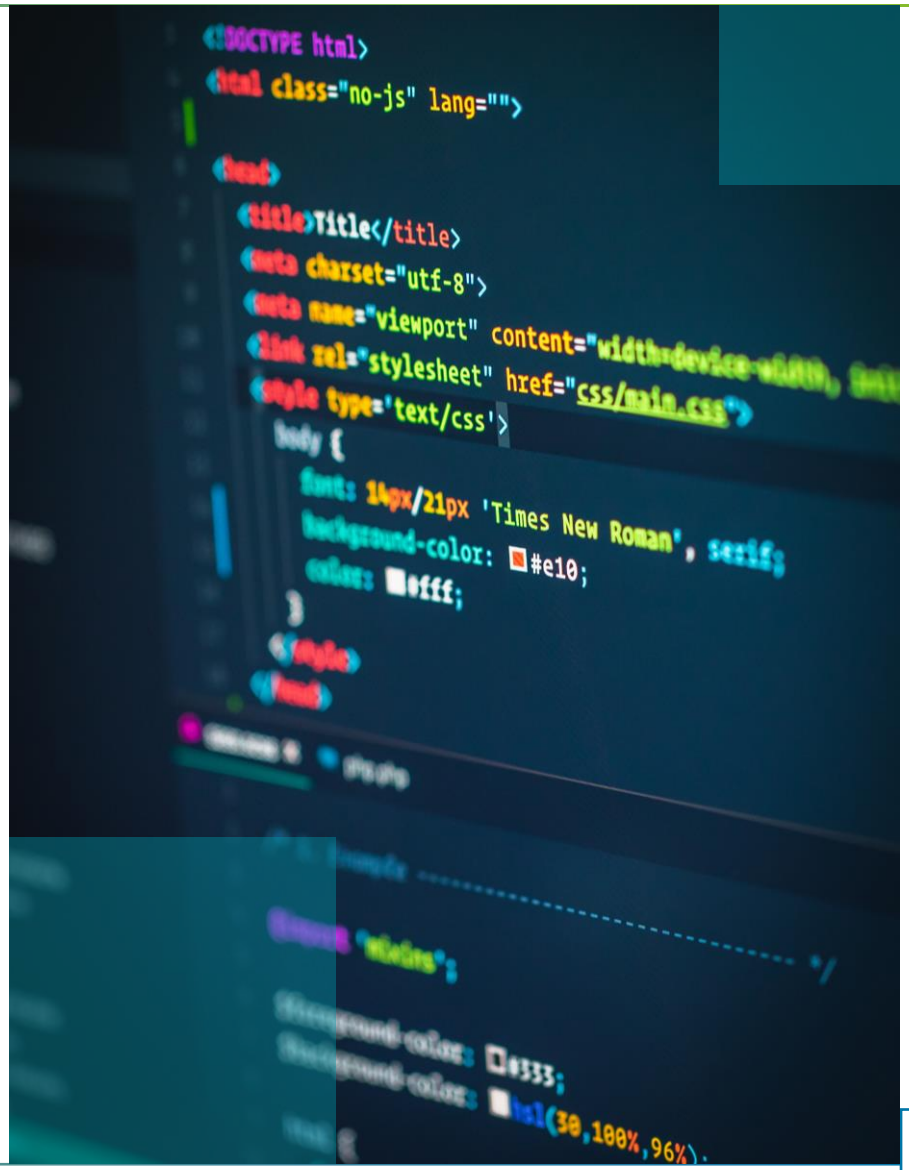
- Event binding lets you listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.
- Use the Angular event binding syntax to bind an event. This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right.
- Event Binding syntax:
  `<button (click)="onSave()">Save</button>`



Property binding

How to set an element property to a component property, valuelink:

- In order for an src property of an <img> element to bind to a component's property, the target, src , needs to be placed in square brackets followed by an equal sign and then the property.
- In the class declare the `itemImageUrl` property, in this case, AppComponent.



Binding Data

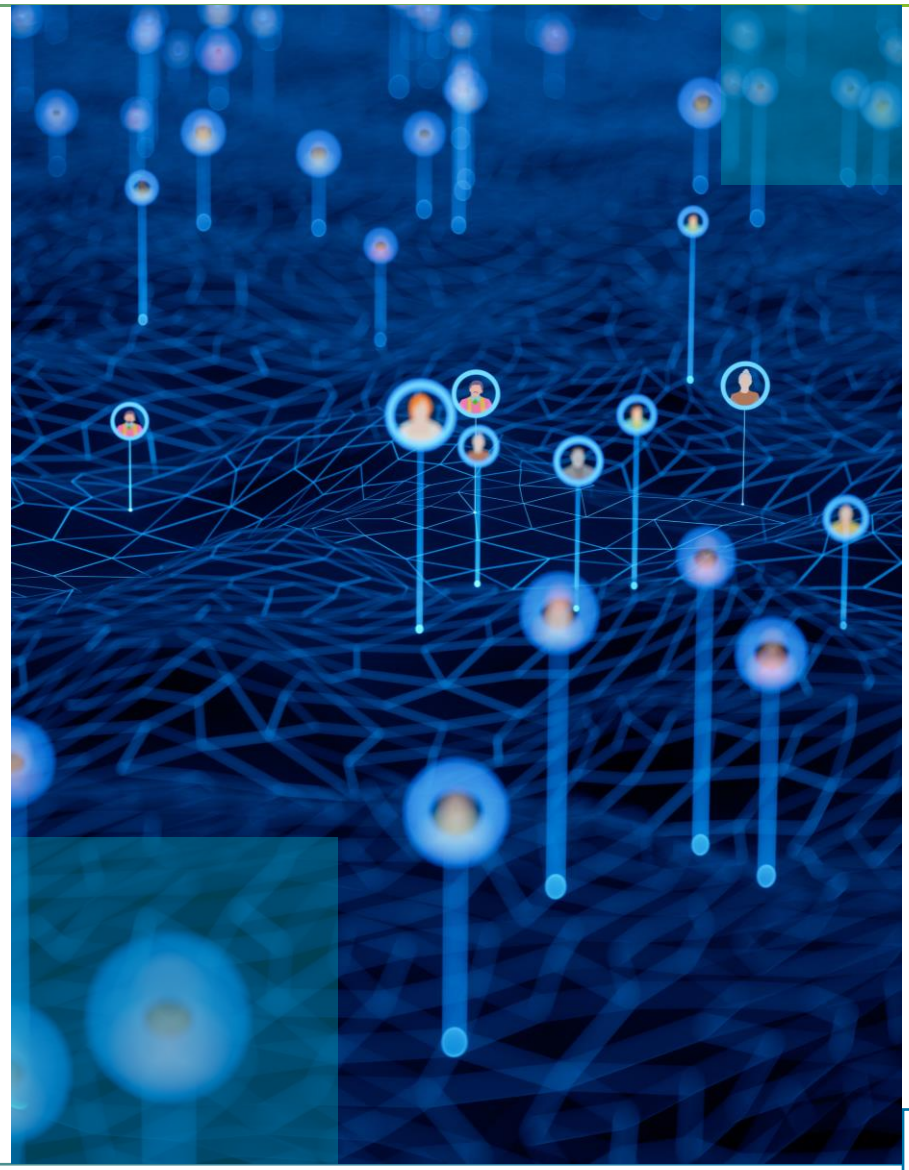
Data binding in AngularJS is the synchronization between the model and the view.

Data binding automatically keeps your page up-to-date based on your application's state.

Data binding is used to specify things such as the source of an image, the state of a button, or data for a particular user.

The target of a data binding can be a property, an event, or an attribute name.

Every public member of a source directive is automatically available for binding in a template expression or statement.



Binding Data (Cont.)

</>

The following table summarizes the targets for the different binding types:

TYPES	TARGETS	EXAMPLES

Property	Element property
	Component property
	Directive property
		`alt, src, test, and ngClass` in the following:
		`<img [alt]="test.name" [src]="testImageUrl">`
		`<app-test-detail [test]="currentTest"></app-test-detail>`
		`<div [ngClass]="{'special': isSpecial}"></div>`

Event	Elementevent
	Component event
	Directive event
		click, `deleteRequest`, and `myClick` in the following:
		`<button type="button" (click)="onSave()">Save</button>`
		`<app-test-detail (deleteRequest)="deleteTest()"></app-test-detail>`
		`<div (myClick)="clicked=$event" clickable>click here</div>`

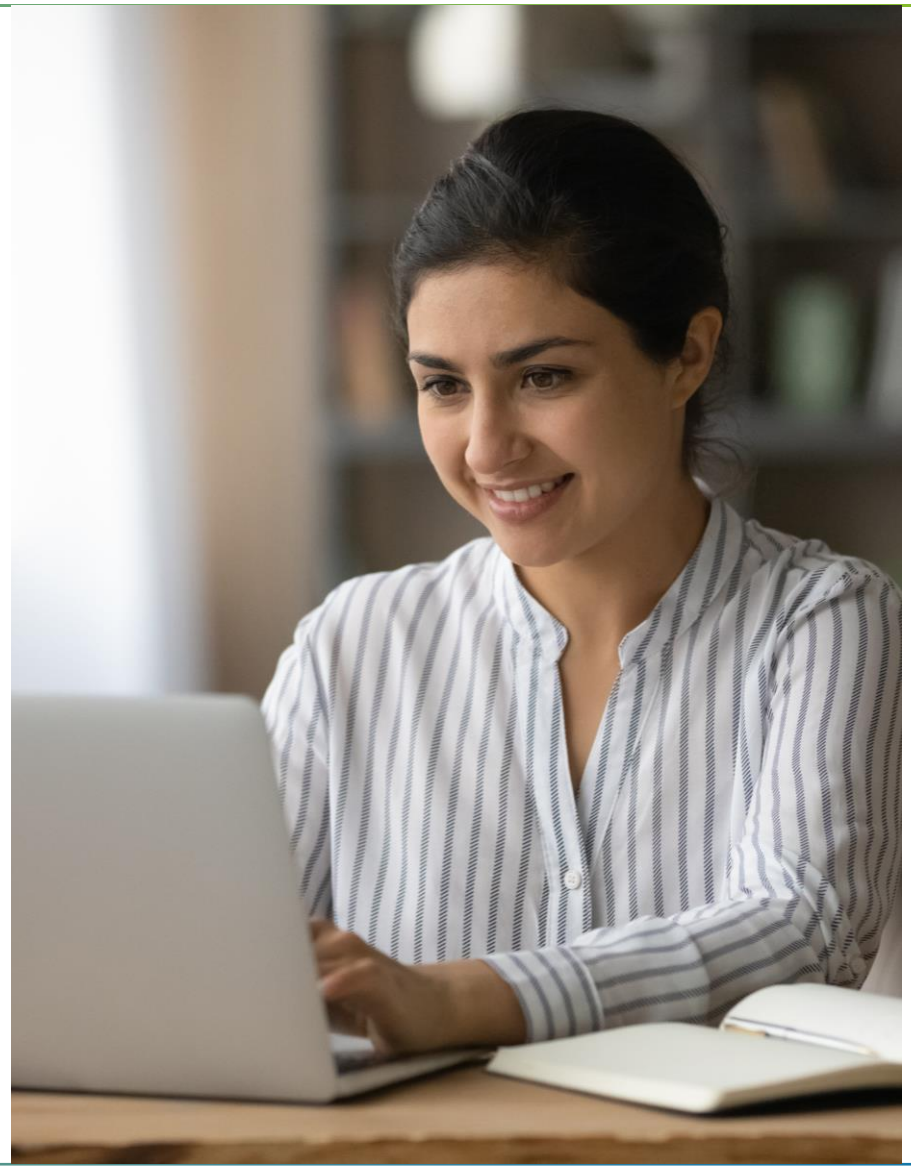
Two-way	Event and property	`<input [(ngModel)]="name">`
Attribute	Attribute (the exception)	`<button type="button" [attr.aria-label]="test">test</button>`

Binding Data (Cont.)

</>

TYPES	TARGETS	EXAMPLES
Class	class property	`<div [class.test]="isTest">Test</div>`
Style	style property	`<button type="button" [style.color]="isColor ? 'red' :'green'">`

Directives

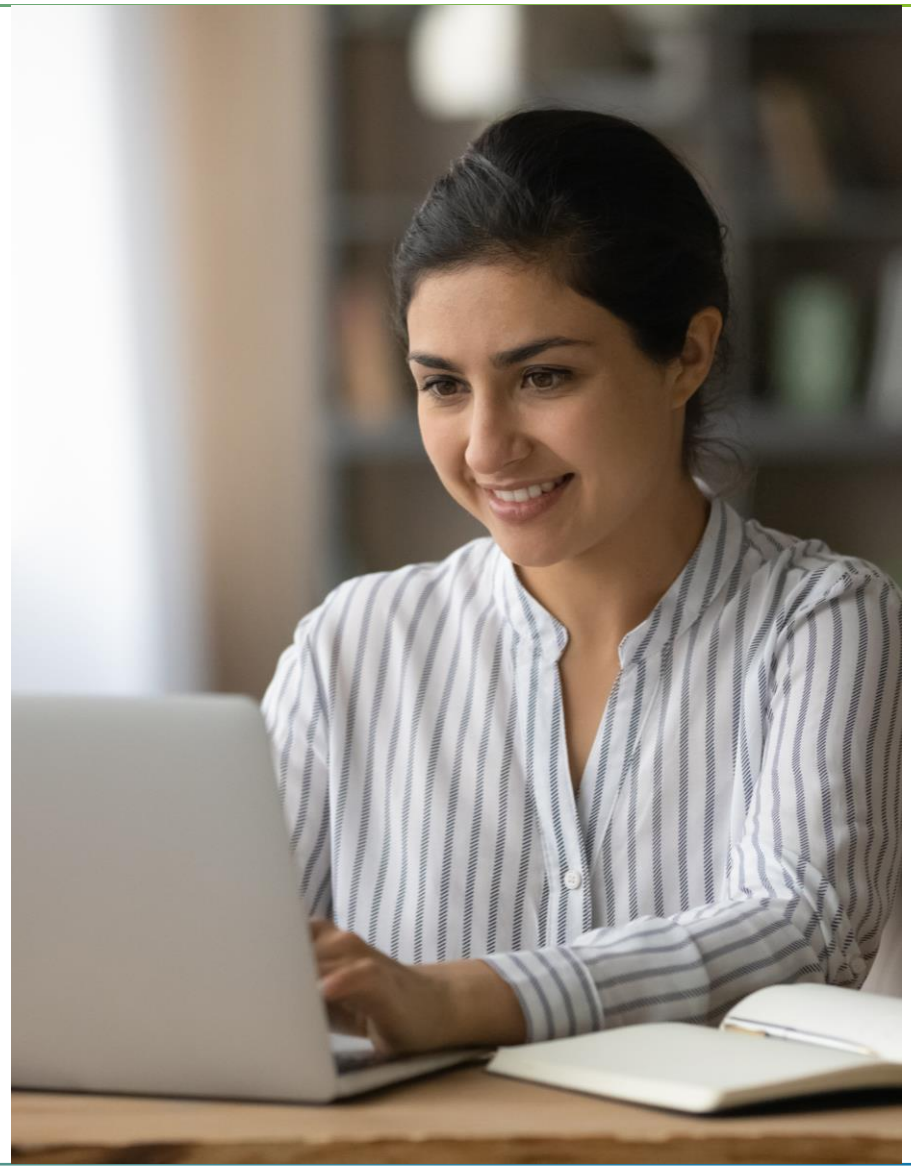


What Are Directives?

Directives are classes that add additional behavior to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see.

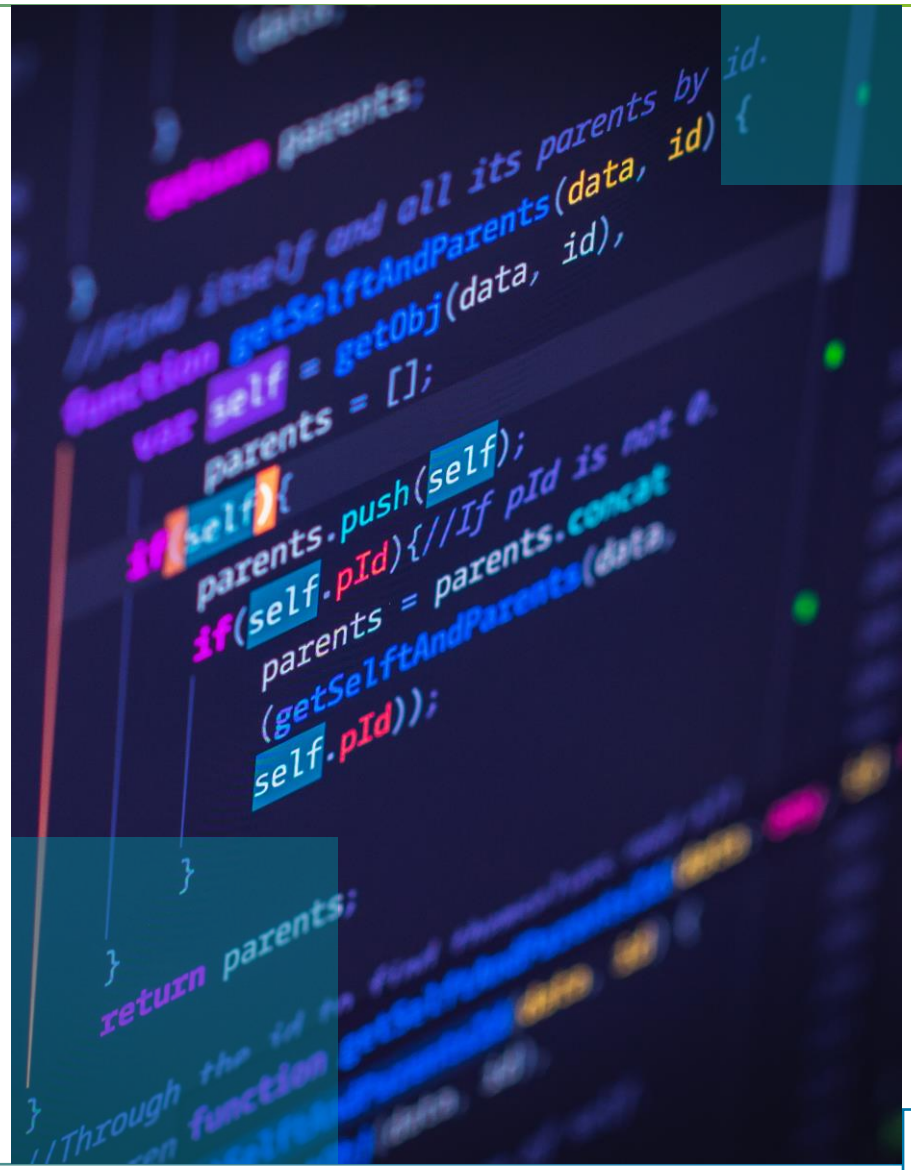
Directive Types	Details
Components	Used with a template and this type of directive is the most common directive type.
Attribute directives	Change the appearance or behavior of an element, component, or another directive.
Structural directives	Change the DOM layout by adding and removing DOM elements.

Structural Directives



Structural Directives

- Structural directives are directives that change the DOM layout by adding and removing DOM elements.
- Angular provides a set of built-in structural directives (such as ngIf, ngForOf, ngSwitch, and others) which are commonly used in all Angular projects.



Adding and Removing Elements Dynamically

</>

1. The First step for adding and removing elements dynamically would be to **Import FormsModule**.

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

Adding and Removing Elements Dynamically (Cont.)

</>

2. In the second step, Update the TS file. For this, we will use the angular forms library to import FormArray, FormControl, FormBuilder, and FormGroup.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, FormArray, FormBuilder } from '@angular/forms'

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

Adding and Removing Elements Dynamically (Cont.)

</>

2. In the second step, Update the TS file. For this, we will use the angular forms library to import FormArray, FormControl, FormBuilder, and FormGroup. (Cont.)

```
…
 productForm: FormGroup;

  constructor(private fb:FormBuilder) {

    this.productForm = this.fb.group({
      name: '',
      quantities: this.fb.array([]) ,
    });
  }
…
```

Adding and Removing Elements Dynamically (Cont.)

</>

2. In the second step, Update the TS file. For this, we will use the angular forms library to import FormArray, FormControl, FormBuilder, and FormGroup. (Cont.)

```
…
quantities() : FormArray {
    return this.productForm.get("quantities") as FormArray
  }

  newQuantity(): FormGroup {
    return this.fb.group({
      qty: '',
      price: '',
    })
  }
 …
```

Adding and Removing Elements Dynamically (Cont.)

</>

2. In the second step, Update the TS file. For this, we will use the angular forms library to import FormArray, FormControl, FormBuilder, and FormGroup. (Cont.)

```
…
addQuantity() {
    this.quantities().push(this.newQuantity());
  }

  removeQuantity(i:number) {
    this.quantities().removeAt(i);
  }

  onSubmit() {
    console.log(this.productForm.value);
  }
```

Adding and Removing Elements Dynamically (Cont.)

</>

3. Now, create Template code. For this, we will use ngModel so that we can write code in HTML form. We will use the file app.component.html to add the below code. In the following form, we can also use the Bootstrap class.

```
<div class="container">

  <h1> Dynamically adding new fields</h1>

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

    <p>
      <label for="name">Product Name:</label>
      <input type="text" id="name" name="name" formControlName="name" class="form-control">
    </p>
…
```

Adding and Removing Elements Dynamically (Cont.)

</>

3. Now, create Template code. For this, we will use ngModel so that we can write code in HTML form. We will use the file app.component.html to add the below code. In the following form, we can also use the Bootstrap class. (Cont.)

```
…
     <table class="table table-bordered" formArrayName="quantities">
      <tr>
        <th colspan="2">Add Multiple Quantity:</th>
        <th width="150px"><button type="button" (click)="addQuantity()" class="btn btn-primary">Add
More</button></th>
      </tr>
      <tr *ngFor="let quantity of quantities().controls; let i=index" [formGroupName]="i">
        <td>
            Quantity :
            <input type="text" formControlName="qty" class="form-control">
        </td>
…
```

Adding and Removing Elements Dynamically (Cont.)

</>

3. Now, create Template code. For this, we will use ngModel so that we can write code in HTML form. We will use the file app.component.html to add the below code. In the following form, we can also use the Bootstrap class. (Cont.)

```
…
        <td>
            Price:
            <input type="text" formControlName="price" class="form-control">
        </td>
        <td>
            <button (click)="removeQuantity(i)" class="btn btn-danger">Remove</button>
        </td>
      </tr>
    </table>
…
```

Adding and Removing Elements Dynamically (Cont.)

</>

3. Now, create Template code. For this, we will use ngModel so that we can write code in HTML form. We will use the file app.component.html to add the below code. In the following form, we can also use the Bootstrap class. (Cont.)

```
…
<button type="submit" class="btn btn-success">Submit</button>
       </form>
  <br/>
  {{this.productForm.value | json}}
</div>
…
```

Creating Tables or Lists with ngFor

</>

NgFor is a built-in template directive that makes it easy to iterate over something like an array or an object and create a template for each item.

Here's a basic example of its use:

```
<ul>
  <li *ngFor="let user of users">{{ user.name }}</li>
</ul>
```

The * character before ngFor creates a parent template. It's a shortcut to the following syntax: template="ngFor let item of items".

ngSwitch and ngIf

< / >

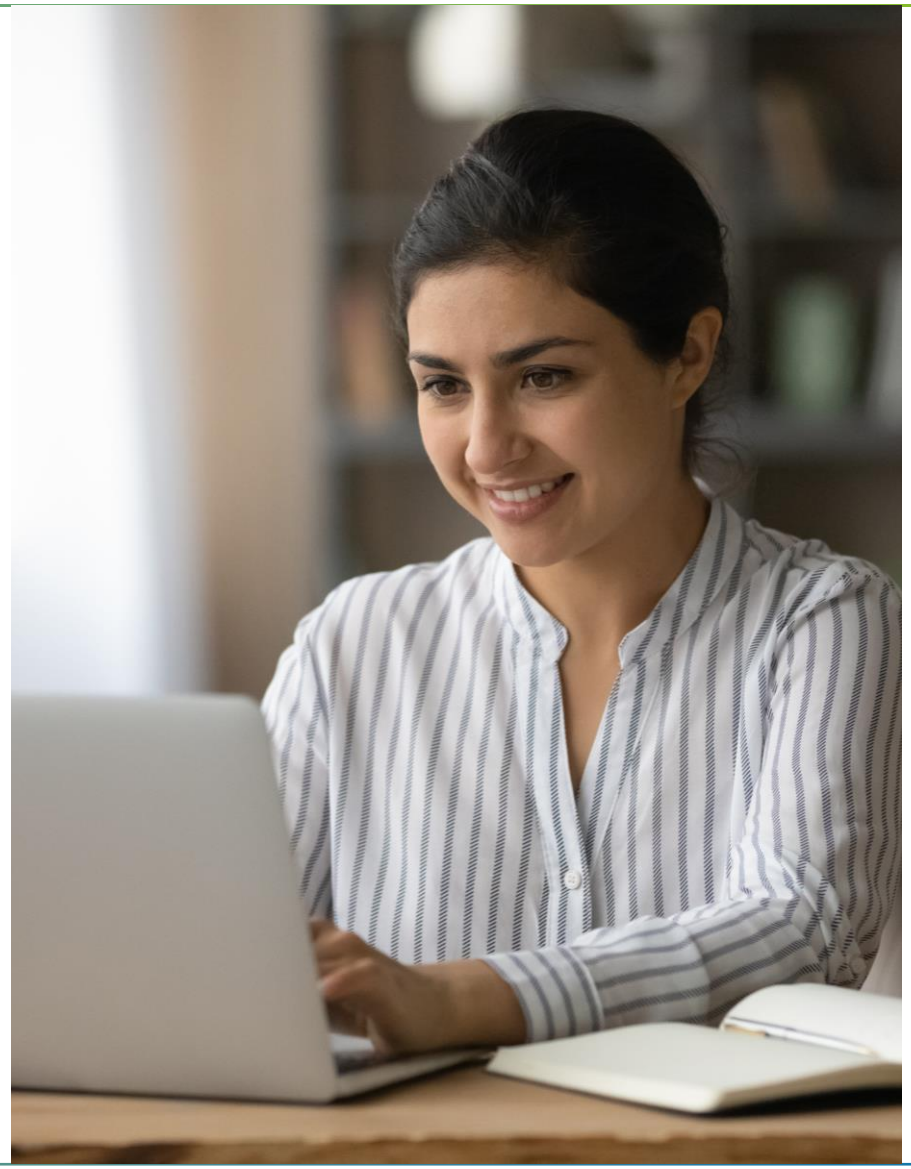
ngSwitch

- ngSwitch adds or removes the DOM trees when their expressions match the switch expression.
- Its syntax comprised of two directives attribute and a structural directive.

ngIf

ngIf directive renders components or elements conditionally based on whether the expression is true or false.

Attribute Directives



Apply Styles by Changing Classes

- Use class and style bindings to add and remove CSS class names from an element's class attribute and to set styles dynamically.
- To bind to multiple classes, type the following: `[class]="classExpression"`
- A single HTML element can have its CSS class list and style values bound to multiple sources.
- The expression can be one of:
  - A space-delimited string of class names.
  - An object with class names as the keys and truthy or falsy expressions as the values.
  - An array of class names.

Changing Classes—Example

</>

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    singers: Singer[] = [
       {
      'artist’: ‘abc xyz’,
      'country': 'USA'
    },
…
```

```
…
    {
      'artist': ‘pqr',
      'country': 'Canada'
    },
    {
      'artist': ‘def',
      'country’: ‘India'
    },
   ];
}
```

Changing Classes—Example (Cont.)

</>

app.component.html

```
<div class="container">
<h4>NgClass</h4>
<div *ngFor="let celeb of singers">
  <p [ngClass]="{
    'text-success':celeb.country === 'USA',
    'text-secondary':celeb.country === 'Canada',
    'text-info':celeb.country === 'India'
  }">{{ celeb.artist }} ({{ celeb.country }})
</p>
</div>
</div>
…
```

Applying Styles Directly

- Everything you know about CSS stylesheets, selectors, rules, and media queries can be applied directly to Angular applications.
- Additionally, Angular can bundle component styles with components, enabling a more modular design than regular stylesheets.
- There are several ways to add styles to a component:
  - By setting styles or styleUrls metadata
  - Inline in the template HTML
  - With CSS imports
- The styles in the style file apply only to this component. They are not inherited by any components nested within the template nor by any content projected into the component.

Controlling Element Visibility

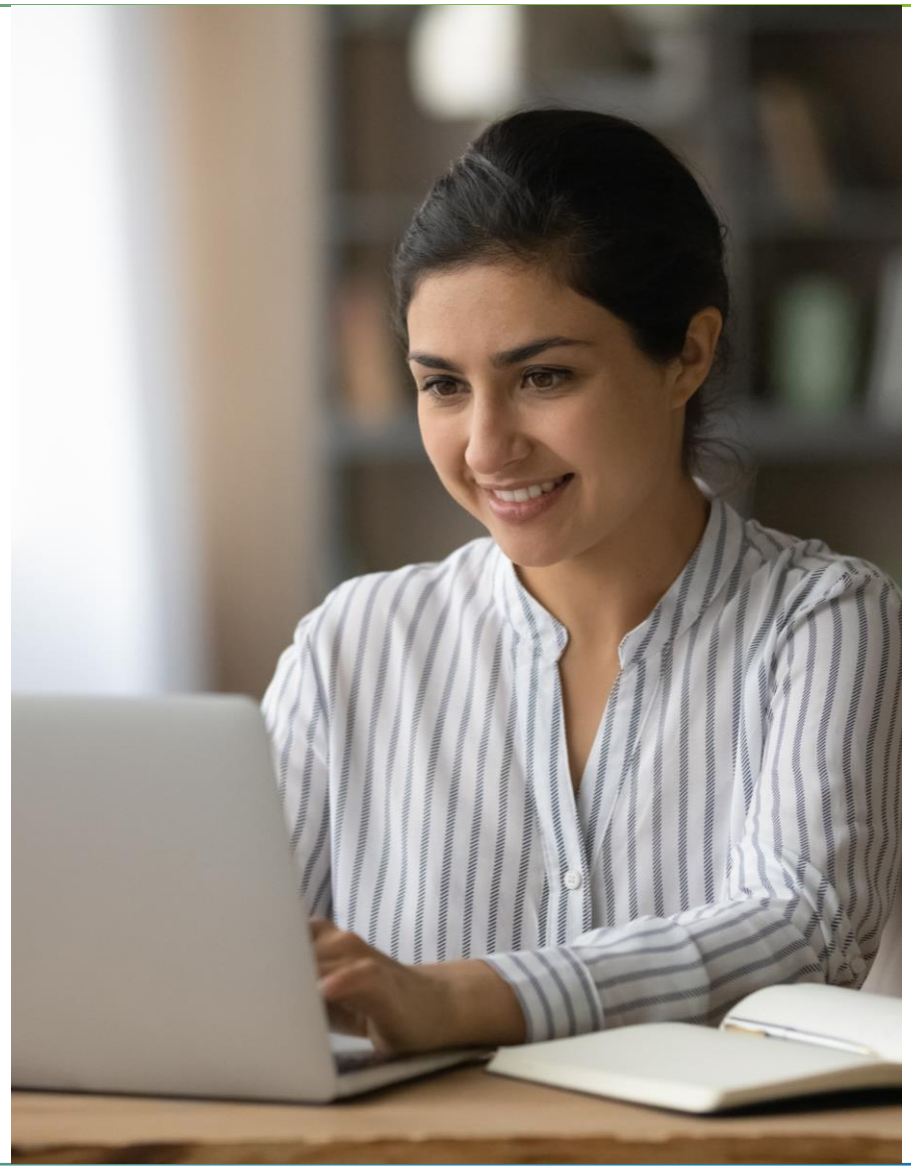
</>

*ngIf directive in Angular can be used to toggle or show and hide elements. The Angular ngIf will add or remove an element from the DOM, based on a condition such as true or false.

Below is an example of that :

```
<button (click)="toggle()" id="bt">
    {{buttonName}}
</button>
<ng-container *ngIf="show">
    <div style="margin: 0 auto;text-align: left;">
        <div>
            <label>Name:</label>
            <div><input id="tbname" name="yourname" /></div>
        </div>
        <div>
            <label>Email Address:</label>
            <div><input name="email" id="email" /></div></div>
        <div>
            <label>Additional Information (optional):</label>
            <div><textarea rows="5" cols="46"></textarea></div>
        </div>
    </div>
</ng-container>
```

Service and Dependency Injection



What Is a Service?

</ >

Services

- Services in Angular is a class that contains some functionality which can be reused across the application
- A service is a singleton object
- Service must be decorated with `@Injectable` decorator
- Angular has an in-built dependency injection subsystem
- An angular service is an object that can be wired into a component through dependency injection

Why Services?

- Can be used to share the code across the components of an application
- Can be used to make HTTP requests

Creating a Service

</>

Creating a Service

We can create a service using the below command:

```
ng generate service book
```

The above command will create a class as shown below:

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

Creating a Service (Cont.)

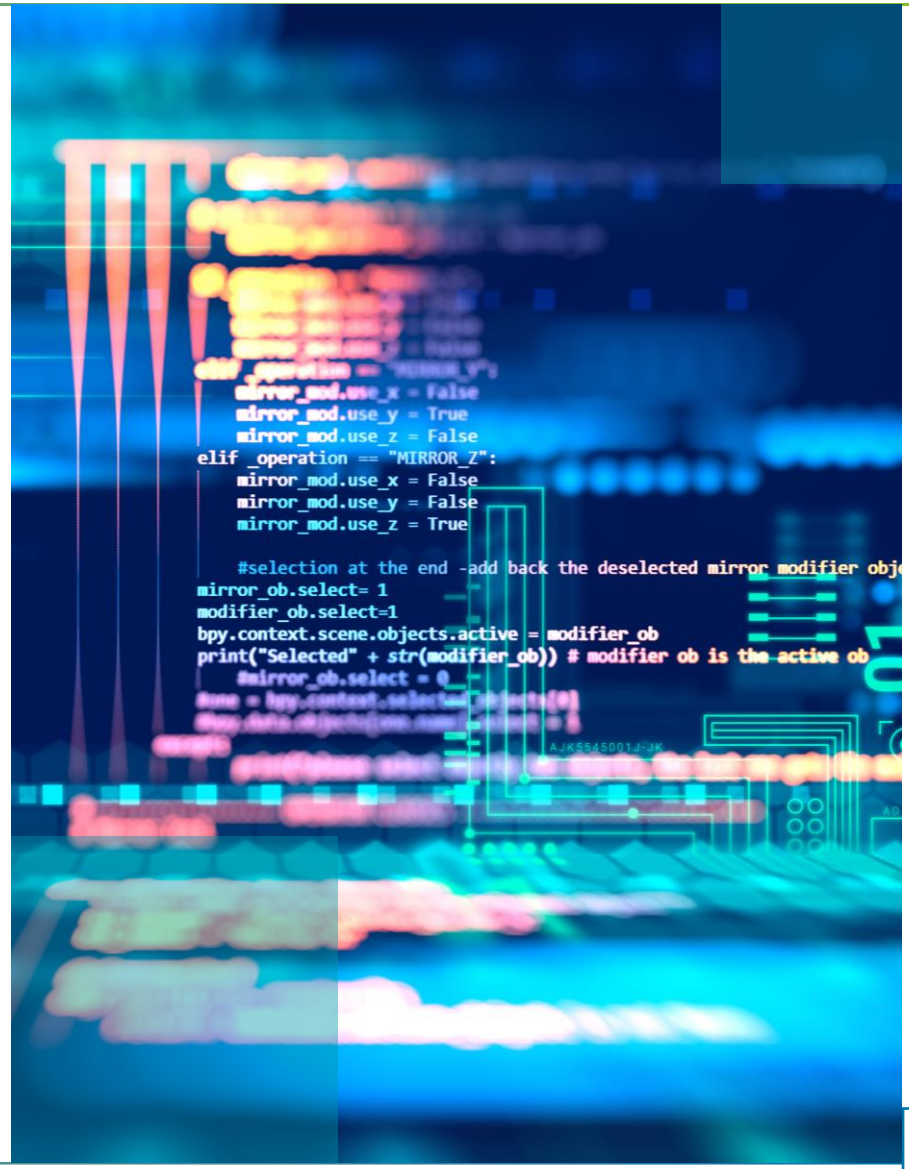
</>

- Angular identifies the class as a service if decorated with `@Injectable` decorator.
- Services can be provided across the application by registering it using the provider's property in the `@Ngmodule` decorator of any module.
- There is a way to limit the scope of a service class by registering it in the provider's property in `@component` to that service.
- The service will be visible only to the component and its child component.

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

Using a Shared Service Instance

- Angular components often need to share data between them. To do this, the following techniques can be used:
  - Using shared services
  - Using Subjects or Observables to share data between components
- Marking a class with `@Injectable` ensures that the compiler will generate the necessary metadata to create the class's dependencies when the class is injected



What Is Dependency Injection?

< / >

Dependency Injection

- Is a mechanism where the required class will be injected into the component
- Angular has an in-built dependency injection subsystem

Why Dependency Injection?

- Allows developers to reuse the code across the application
- Makes the code loosely coupled

Note: There is no need for developers to explicitly create/instantiate the resources.

Injecting Services

</>

Injecting Service Into Component

The only way to inject a service into a component or class is through the constructor. In the below example, book service will be injected into the component through constructor injection by the framework.

```
import ( BookService ) from './book.service;
export class BookComponent implements OnInit {
books: Book[];
constructor(private bookService: BookService) {
)
getBooks() {
this books= this.bookService.geBooks();
}
ngOnInit() {
this.getBooks();
```

Component Life Cycle

< / >

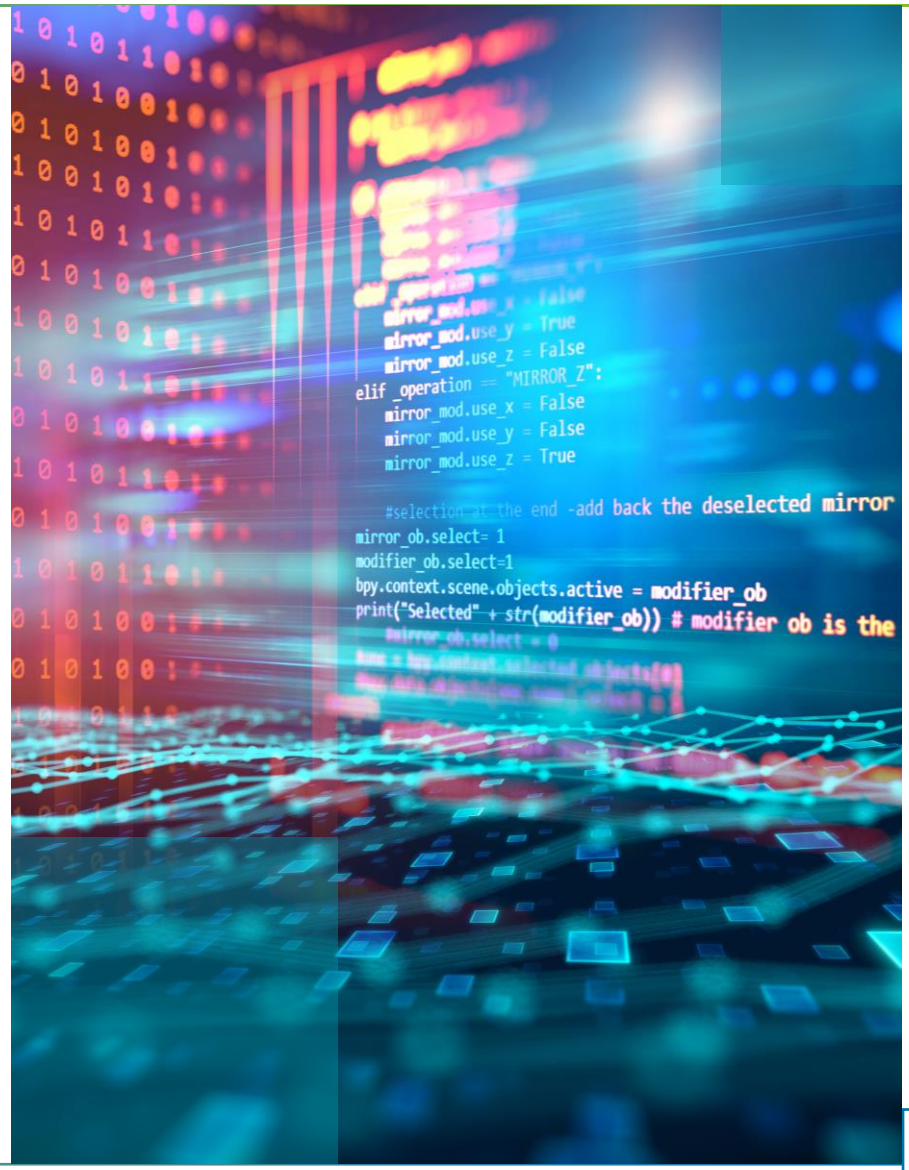
- When Angular instantiates, the component class the lifecycle of a component instance starts. The component and child views are then rendered.
- Next, change detection occurs in the lifecycle, Angular checks if and when data-bound properties change and then updates the view and component instance as needed.
- The lifecycle is complete when Angular destroys the component instance and removes its rendered template from the DOM.
- Directives have a similar lifecycle as Angular creates, updates, and destroys instances in the course of execution.

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

Using OnInit to Initialize Component Data

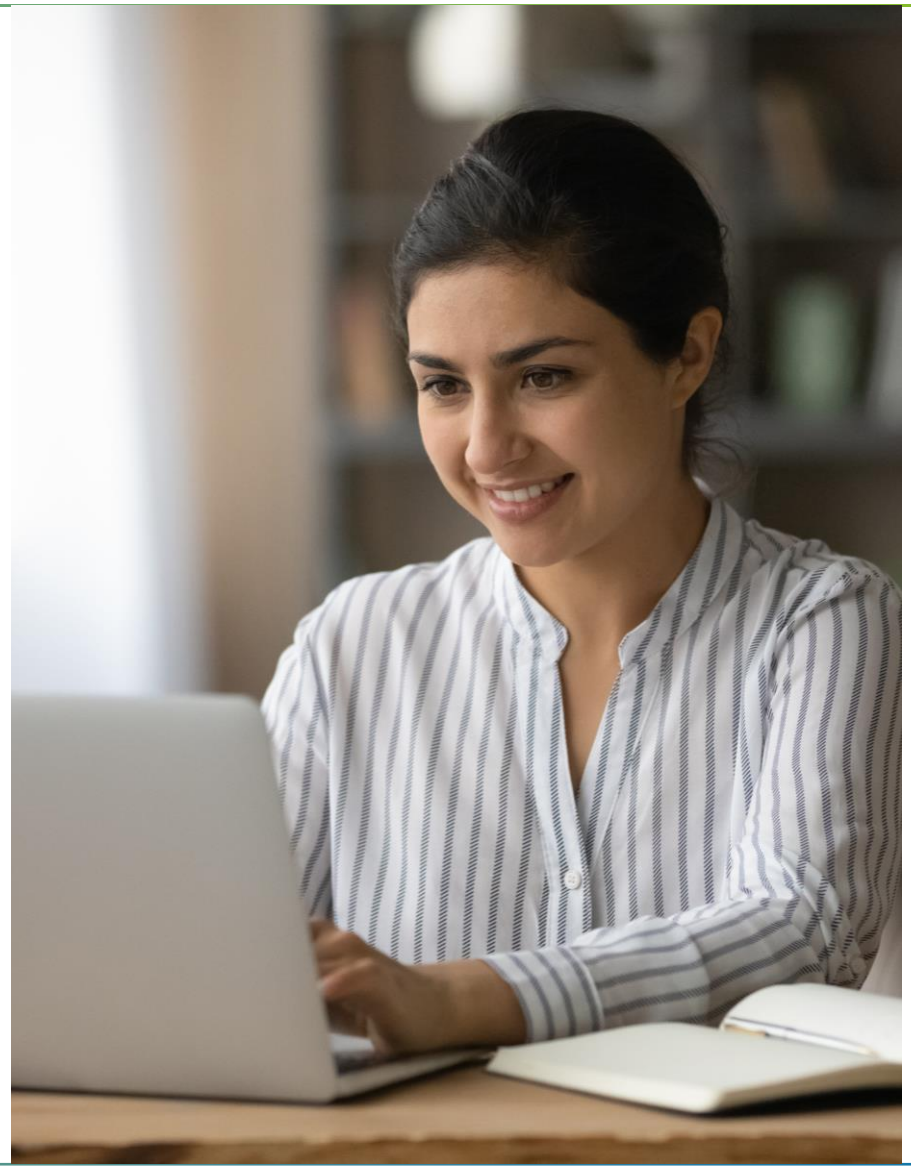
- The `OnInit` is a lifecycle hook that is called after angular has initialized all data-bound properties of a directive.
- We will define a `ngOnInit()` method to handle any additional initialization tasks.

```
interface OnInit {
      ngOnInit(): void
```



</ / >

Pipes and Data Formatting



What Are Pipes?

</>

Pipes in Angular

- Are used to formatting the data before being displayed to the user
- Formats the data inside the template
- A Pipe takes expression value as input and transforms it into desired output

List of built-in pipes in Angular

- uppercase
- lowercase
- titlecase
- currency
- date
- percent
- slice
- decimal
- json
- i18nplural
- i18nselect

Syntax

```
{{ expression 1 pipe }}
```

Pipes

uppercase: This pipe converts template expression into uppercase.

```
{{ "Angular" | uppercase }}
```

lowercase: It converts an expression to lower case.

titlecase: It converts expression to title case.

Passing parameters

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

Pipes (Cont.)

Currency

- Currency code is to display INR for the rupees, EUR for euro, etc.
- Symbol displays symbol such as $.

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

digitInfo is a string in the format shown here

`{minIntegerDigits}.{minFractionDigits}{maxFractionDigits}`

To use locale, we need to register locale in the root module

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

Formatting Changes in Angular

</>

The format indicates which form, date/time to be displayed:

- medium equivalent to ‘MMM d,y, hh:mm:ss’
- short equivalent to ‘M/d/yy, hh:mm’
- long, full, fulldate, longdate, shortdate, shorttime, longtime are other format values

Date

```
{{"Angular" | uppercase })
```

Percent

```
{{ expression | percent:digitInfo:locale})
```

```
digitInfo is the string in below format
{minIntegerDigits}.{minFractionDigits}{maxFractionDigits}
```

Slice

```
{{ expression | slice.start.end})
```

json

Note: This pipe can be used to display given expression in json format.

Using a Built-in Pipe

</>

```
<div class="card card-block">
  <h4 class="card-title">Date</h4>
  <div class="card-text">
    <p ngNonBindable>{{ dateVal | date: 'shortTime' }}</p> (1)
    <p>{{ dateVal | date: 'shortTime' }}</p>

    <p ngNonBindable>{{ dateVal | date:'fullDate' }}</p>
    <p>{{ dateVal | date: 'fullDate' }}</p>

    <p ngNonBindable>{{ dateVal | date: 'd/M/y' }}</p>
    <p>{{ dateVal | date: 'd/M/y' }}</p>
  </div>
</div>
```

Using Pipes in HTML

</>

```
 <div style = "width:100%;">
   <div style = "width:40%;float:left;border:solid 1px
black;">
      <h1>Uppercase Pipe</h1>
      <b>{{title | uppercase}}</b><br/>

      <h1>Lowercase Pipe</h1>
      <b>{{title | lowercase}}</b>

      <h1>Currency Pipe</h1>
      <b>{{6500.23 | currency:"USD"}}</b><br/>
      <b>{{6500.23 | currency:"USD":true}}</b>
      <h1>Date pipe</h1>
  <b>{{todaydate | date:'d/M/y'}}</b><br/>
      <b>{{todaydate | date:'shortTime'}}</b>

      <h1>Decimal Pipe</h1>
      <b>{{ 454.76878989 | number: '3.4-4' }}</b> </div>
   …
```

Using Pipes in HTML (Cont.)

```
<div class="card card-block">
  <h4 class="card-title">Date</h4>
  <div class="card-text">
    <p ngNonBindable>{{ dateVal | date: 'shortTime' }}</p> (1)
    <p>{{ dateVal | date: 'shortTime' }}</p>

    <p ngNonBindable>{{ dateVal | date:'fullDate' }}</p>
    <p>{{ dateVal | date: 'fullDate' }}</p>

    <p ngNonBindable>{{ dateVal | date: 'd/M/y' }}</p>
    <p>{{ dateVal | date: 'd/M/y' }}</p>
  </div>
</div>
```

Chaining Pipes

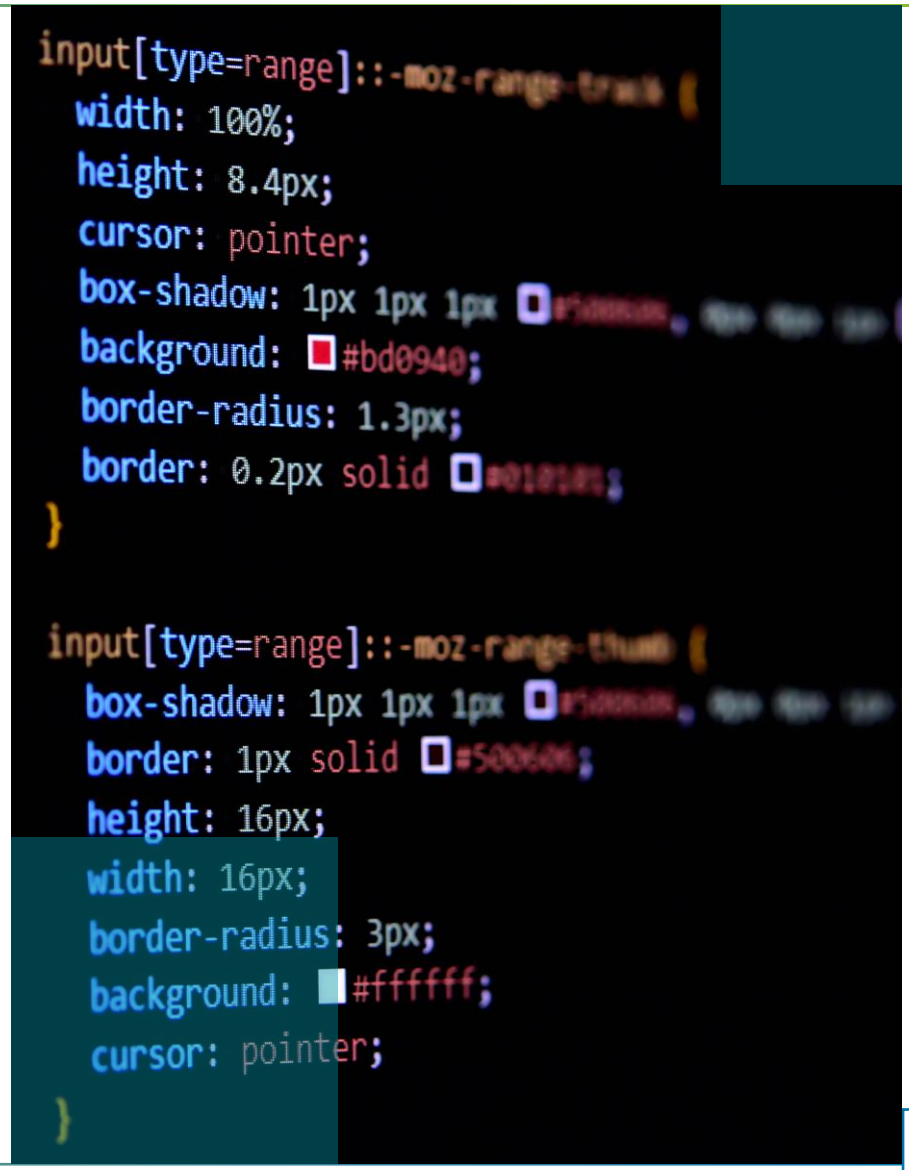
</>

The chaining pipe is used to perform multiple operations within a single expression.

In the following example, chained pipes first apply a format to a date value, then convert the formatted date to uppercase characters.

Example: The chained hero's birthday is:

```
{{ birthday | date | uppercase}}
```



Decimal Pipe

</>

Decimal Pipe	• Transforms a number into a string with a decimal point, formatted according to locale rules • At the very basic, the decimal pipe takes the following format `{{ value_expression | number [ : digitsInfo [ : locale ] ] }}`
Value	It is simply the decimal number you want to format
Number	Is the name of the pipe
Decimal-representation	This is where you specify the actual formatting style for the decimal number

Currency Pipe

</>

Currency Pipe	Transforms a number to a currency string, formatted according to locale rules.

```
{{ value_expression | currency [ : currencyCode [ : display [ :
digitsInfo [ : locale ] ] ] ] }}
```

Custom Pipes

Angular also has the facility to create custom pipes.

The general way to define a custom pipe is given below:

- To implement functionalities such as sorting, filtering, etc., custom pipes should be created.
- We can create our own custom pipe by inheriting the PipeTransform interface.
- We need to override the transform method where we need to write custom pipe functionality.

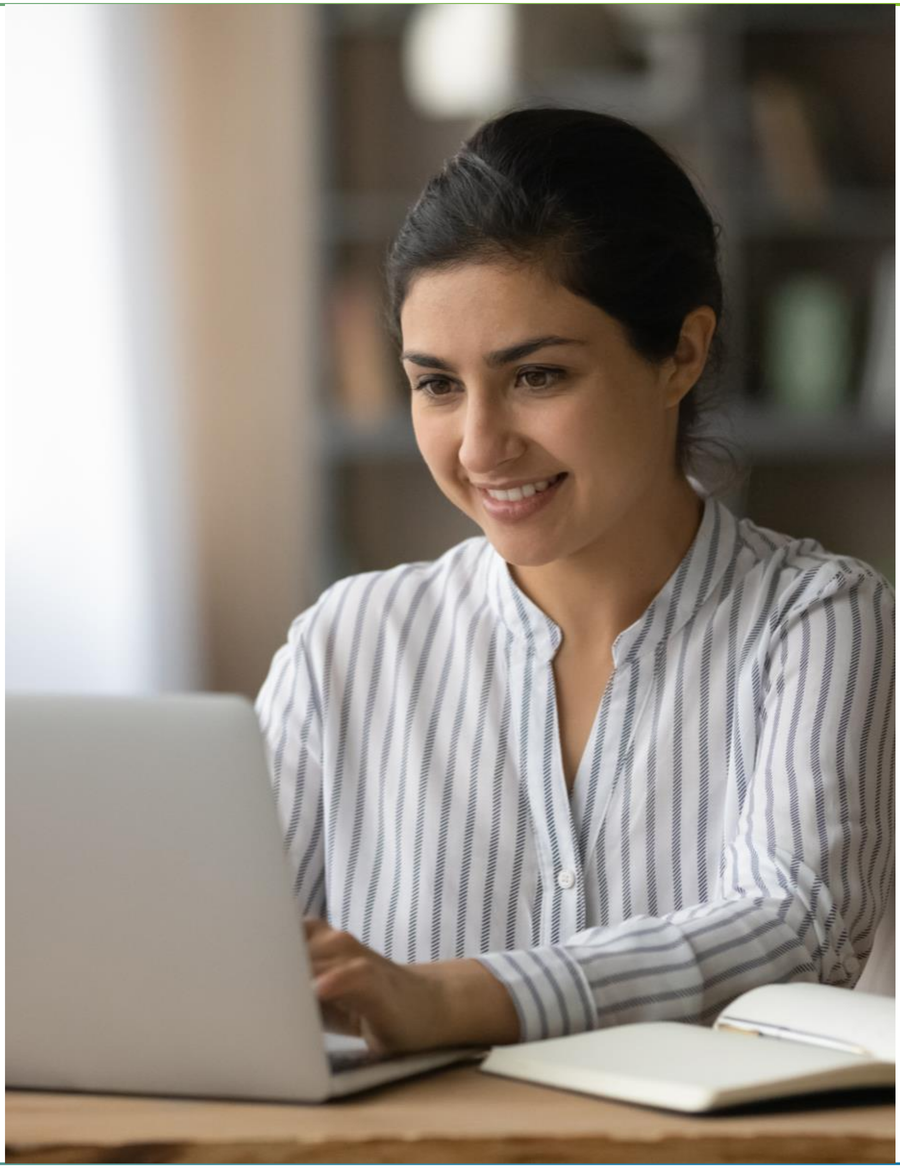
```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({name: 'Pipename'})
export class Pipeclass implements PipeTransform {
   transform(parameters): returntype { }
}
```

Where:

'Pipename'	This is the name of the pipe
Pipeclass	This is the name of the class assigned to the custom pipe
Transform	This is the function to work with the pipe
Parameters	These are the parameters, which are passed to the pipe
Returntype	This is the return type of the pipe

</>

Forms



Template Driven Forms

</>

- Forms are a crucial part of the web application through which we get the majority of our data inputs from users
- We can create two types of forms in Angular:
  - Template driven forms—used to create small to medium-sized forms
  - Model or Reactive forms—used to create large-size forms
  - We need the register forms module during bootstrapping

Template Driven Forms

- Template-driven forms are created using Angular Template syntax
- Data binding, validation, etc., will be written in template itself

```
•••
import { ForasModule } from '@angular/forms’;

@NgModule({
 •••
 imports: [
  BrowserModule,
  Formsmodule
 ],
 •••
 })
 export class Appmodule
```

Template Driven Forms (Cont.)

< / >

ngForm

- ngForm is a built-in directive that will have an instance of each form element created in the application
- name attribute is mandatory when [(ngModel)] is used on form elements
- Each input element will be an instance of FormControl class which will get registered with the name attribute
- Angular runs the control validation process whenever the form control value changes
  i.e., for input, angular runs validation on every keystroke
- A form with heavy validation requirements can be too expensive
- ngModel is used to track the state and validity of an element using the following keywords

Keyword	Purpose
valid	True if element value is valid
invalid	True if element value is invalid
dirty	True if element value is changed
pristine	True if element value is unchanged
touched	True if the elements gets focus
untouched	True if the element doesn’t have a focus in it

Template Driven Forms (Cont.)

</ >

Advantages	Disadvantages
As we put the entire code for data binding and validation in a form template, it is simple to code and very useful for small to medium-sized forms.	As the tags or complex validations increase in the template, the readability of the form decreases.

A Basic Angular Form with Input Fields

</>

Configure FormsModule in AppComponent as shown below:

```
import { FormsModule } from '@angular/forms';
@NgModule({
   declarations: [
      AppComponent,
      TestComponent
   ],
   imports: [
      BrowserModule,
      FormsModule
   ],
   providers: [],
   bootstrap: [AppComponent]
})
export class AppModule { }
```

A Basic Angular Form with Input Fields (Cont.)

</>

Add the below code in `test.component.html` file as follows:

```
<form #userName="ngForm" (ngSubmit)="onClickSubmit(userName.value)">
   <input type="text" name="username" placeholder="username" ngModel>
   <br/>
   <br/>
   <input type="submit" value="submit">
</form>
```

A Basic Angular Form with Input Fields (Cont.)

</>

Create onClickSubmit() method inside test.component.ts file as shown below:

```
import { Component, OnInit } from '@angular/core';
@Component({
   selector: 'app-test',
   templateUrl: './test.component.html',
   styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnInit {

   ngOnInit() {
   }
   onClickSubmit(result) {
      console.log("Entered name: " + result.username);
   }
}
```

Open app.component.html and change the content as specified below:

```
<app-test></app-test>
```

Displaying Form Validation State

</>

To add validation to a template-driven form, you will add the same validation attributes as you would with native HTML form validation. Angular uses directives to match these attributes with validator functions in the framework.

```
<input type="text" id="name" name="name"
class="form-control"
      required minlength="4"
appForbiddenName=“abc"
      [(ngModel)]="test.name" #name="ngModel">
<div *ngIf="name.invalid"
    class="alert">
  <div *ngIf="name.errors?.['required']">
    Name is required.
  </div>
  <div *ngIf="name.errors?.['minlength']">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors?.['forbiddenName']">
    Invalid name.
  </div>
</div>
```

Reactive Forms

</>

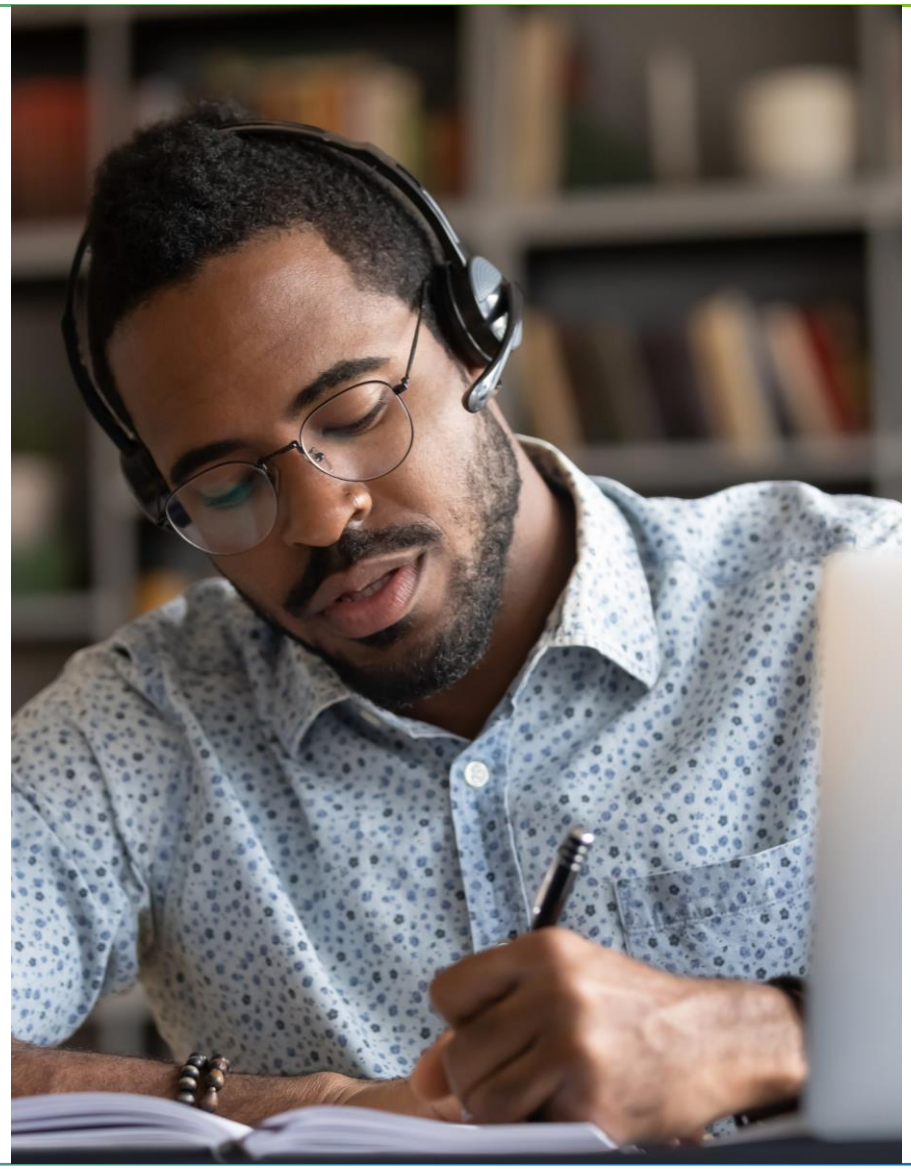
- In Reactive forms, we need to create form control objects in a component class and should bind them with the HTML form element in the template.
- We need to import ReactiveFormsModule to create Reactive forms.
- FormBuilder class can be used to create Reactive forms.
- A component can observe the form control state changes and react to them.

Advantages of Reactive Forms

- Unit testing can be done on validation as it is written inside the component class.
- Complex validations can be performed.

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

Hands-On Labs



Hands-On Activity 1

</ >

Activity Details	
Problem Statement	Write a program to demonstrate navigating between different routes in an Angular application.

Hands-On Activity 2

</ >

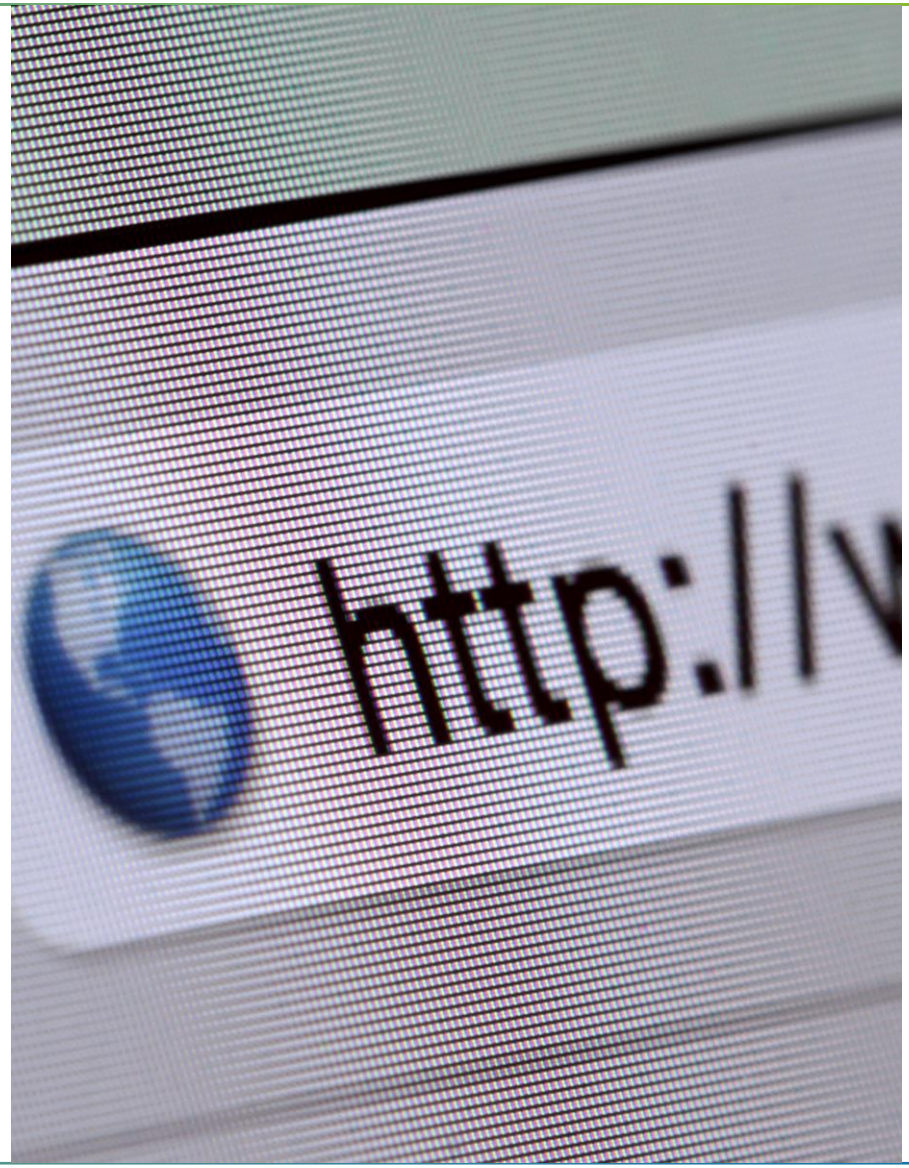
Activity Details	
Problem Statement	Write a program to implement Validations in Angular.

Hands-On Activity 3

</ >

Activity Details	
Problem Statement	Write a program to demonstrate property decorators in Angular.

HTTP Client



The Angular HTTP Client

</>

- Communication with a server from most front-end applications happens over the HTTP protocol, to download or upload data and access other back-end services.
- Angular provides a client HTTP API for Angular applications, the HttpClient service class in `@angular/common/http`.
- The HTTP client service offers the following major features:

The ability to request typed response objects

Testability features

Streamlined error handling

Request and response interception

Using the HTTP Client—Overview

</>

Why Do We Need HttpClient?

- To send or get the data over HTTP protocol the front-end of applications communicates with back-end services using either fetch API or XMLHttpRequest interface.
- In Angular with the help of HttpClient this communication is completed.

What Is HttpClient?

- HttpClient is a built-in service class available in the `@angular/common/http package`.
- It has multiple signatures and returns types for each request. It uses the RxJS observable-based APIs, which means it returns the observable and what we need to subscribe to it.
- This API was developed based on the XMLHttpRequest interface exposed by browsers.

Using the HTTP Client—Overview

Features of HttpClient
Provides typed request and response objects
Contains testability features
Intercepts request and response
Supports RxJS observable-based APIs
Supports streamlined error handling
Performs the GET, POST, PUT, and DELETE operations

Setting Up the Root Component and Service Using HTTP Client

</>

- You need to import the Angular HttpClientModule before you can use HttpClient. In the root AppModule most apps do so.
- Shown in the ConfigService example, the HttpClient service can then inject as a dependency of an application class.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}

App/config/config/config.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

Interceptors

</>

- With interception, HTTP requests are inspected and transformed by interceptors you declared from your application to a server. The same interceptors can also inspect and transform a server's responses on their way back to the application. A forward-and-backward chain of request/response handlers are formed by multiple interceptors.
- A variety of implicit tasks can be performed by interceptors, from authentication to logging, in a routine, standard way, for every HTTP request/response.
- **Write an interceptor**
- To implement an interceptor, declare a class that implements the intercept() method of the HttpInterceptor interface.
- Here is a do-nothing noop interceptor that passes the request through without touching it.

```
import { Injectable } from '@angular/core';
import {
  HttpEvent, HttpInterceptor, HttpHandler, HttpRequest
} from '@angular/common/http';

import { Observable } from 'rxjs';

/** Pass untouched request through to the next request
handler. */
@Injectable()
export class NoopInterceptor implements HttpInterceptor
{

  intercept(req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

Service Imports and Using the Service in a Component

</>

To use the service, you have set up in your project, all you need to do is to import it where needed and bring functions from it through the component's constructor.

```
import { Component, OnInit } from '@angular/core';
import { DataService } from '../data.service';
@Component({
  selector: 'app-test',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.css']
})
export class testComponent implements OnInit {
public test = []
constructor(private list: DataService) { }
ngOnInit(): void {
    this.test = this.list.getList();
  }
}
```

Making a HTTP GET Call

</>

- Use the `HttpClient.get()` method to fetch data from a server.
- The asynchronous method sends HTTP request and returns an Observable that emits the requested data when the response is received.
- The return type varies based on the observation and responseType values that you pass to the call.
- The get() method takes two arguments; the endpoint URL from which to fetch, and to configure the request the options object that is used.

Importing Observable Methods

</>

Observables for all transactions are made use of by the HttpClient service. The RxJS observable and operator symbols that appear in the example snippets must be imported. These ConfigService imports are typical.

```
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from
'rxjs/operators';
```

Enhancing the Service With .map()

</>

.map(): An employee detail is represented by an array containing multiple objects that you received. Each employee needs to have their id retrieved in an array object.

```
 // What you have
var employees = [
 { id: 1, name: 'abc' },
 { id: 2, name: 'def' },
 { id: 3, name: 'ghi' },
 { id: 4, name: 'jkl' }
];
// What you need
[1, 2, 3, 4]
```

You can achieve this using `.map()` like below.

```
var employeeIds = employees.map(employee => employee.id);
```

The new value in the resulting array is returned after the callback runs for each value in an array. The original array and resulting array length will be the same.

Enhancing the Service With .catch()

</>

To use the catch() of the observable before delegating the error response to a method you need to use Observable.throw() method. The catch needs to return an observable.

```
.catch(e => { console.log(e); return Observable.of(e); })
```

If you'd like to stop the pipeline after a caught error, then do this:

```
.catch(e => Observable.empty())
```

This catch transforms the error into a null val and then the filter doesn't lets false values through. This will, however, stop the pipeline for ANY false value, so if you think those might come through and you want them to, you'll need to be more explicit/creative.

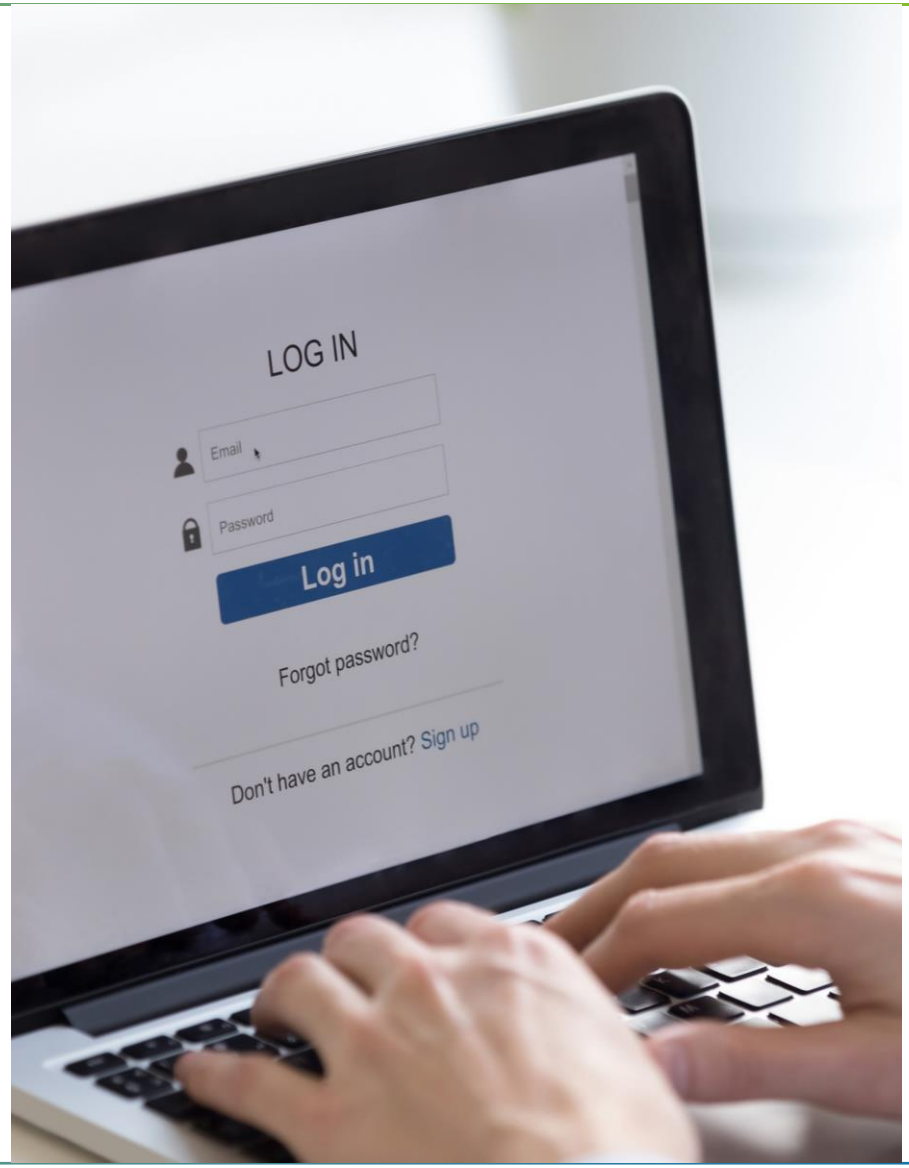
GET Request

</>

- `HttpClient.get()` method is an asynchronous method that performs an HTTP get a request in Angular applications and returns an Observable.
- And that Observable emits the requested data when the response is received from the server.
- The get()method takes two arguments—The endpoint URL from which to fetch, and the options object that is used to configure the request.

```
//Http Client get method
public getUsers(): Observable<any> {
    const url = 'https://path/api/users?page=1';
    return this.http.get<any>(url);
}
```

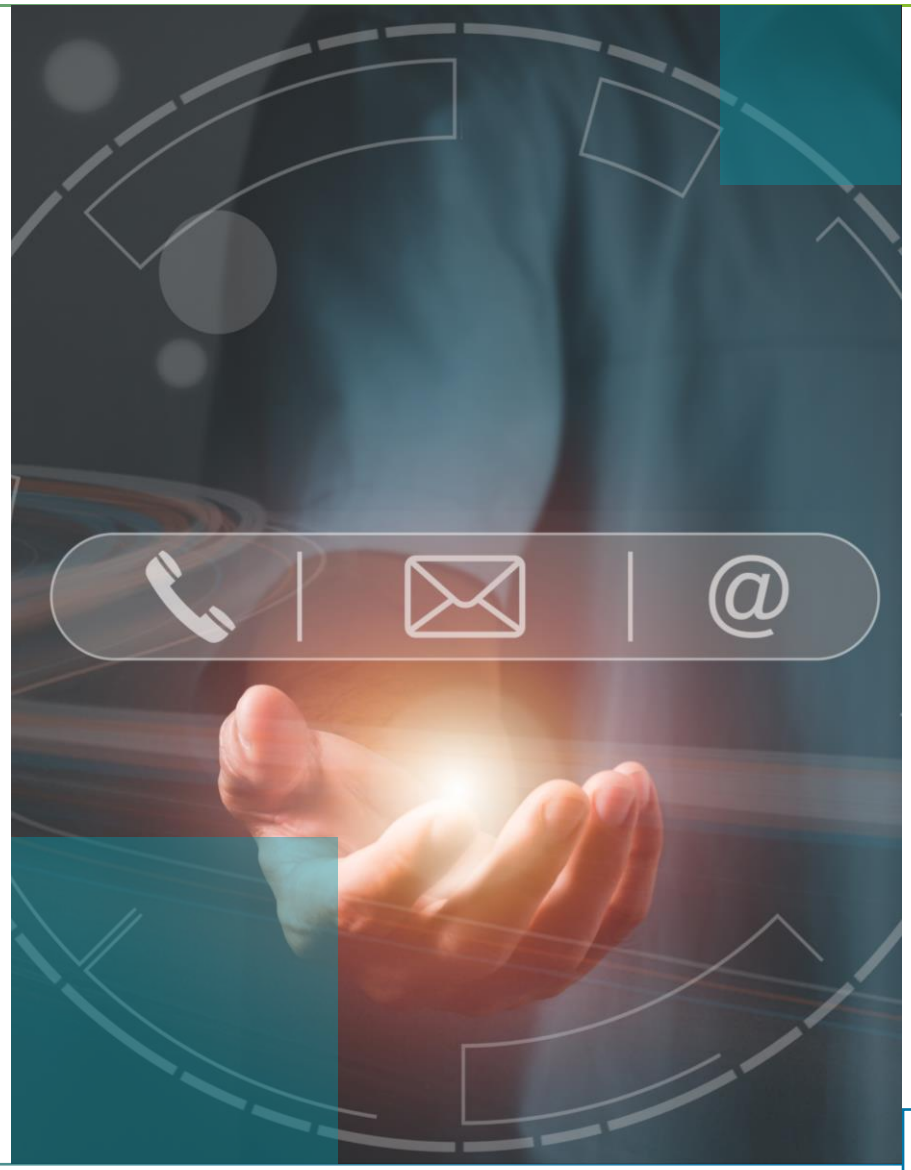
Single Page Application and Angular Routing



Single Page Application

< / >

- All of your application's functions exist on a single HTML page In a SPA (Single Page Application). As your application's features are accessed by the user, instead of loading a new page the browser needs to provide only the parts that matter to the user.
- Your application’s user experience can significantly improve with this pattern.
- Routes are used to define how users navigate through your application.
- Add routes to define how users navigate from one part of your application to another. You can also configure routes to guard against unexpected or unauthorized behavior.



Routing and Navigation

</>

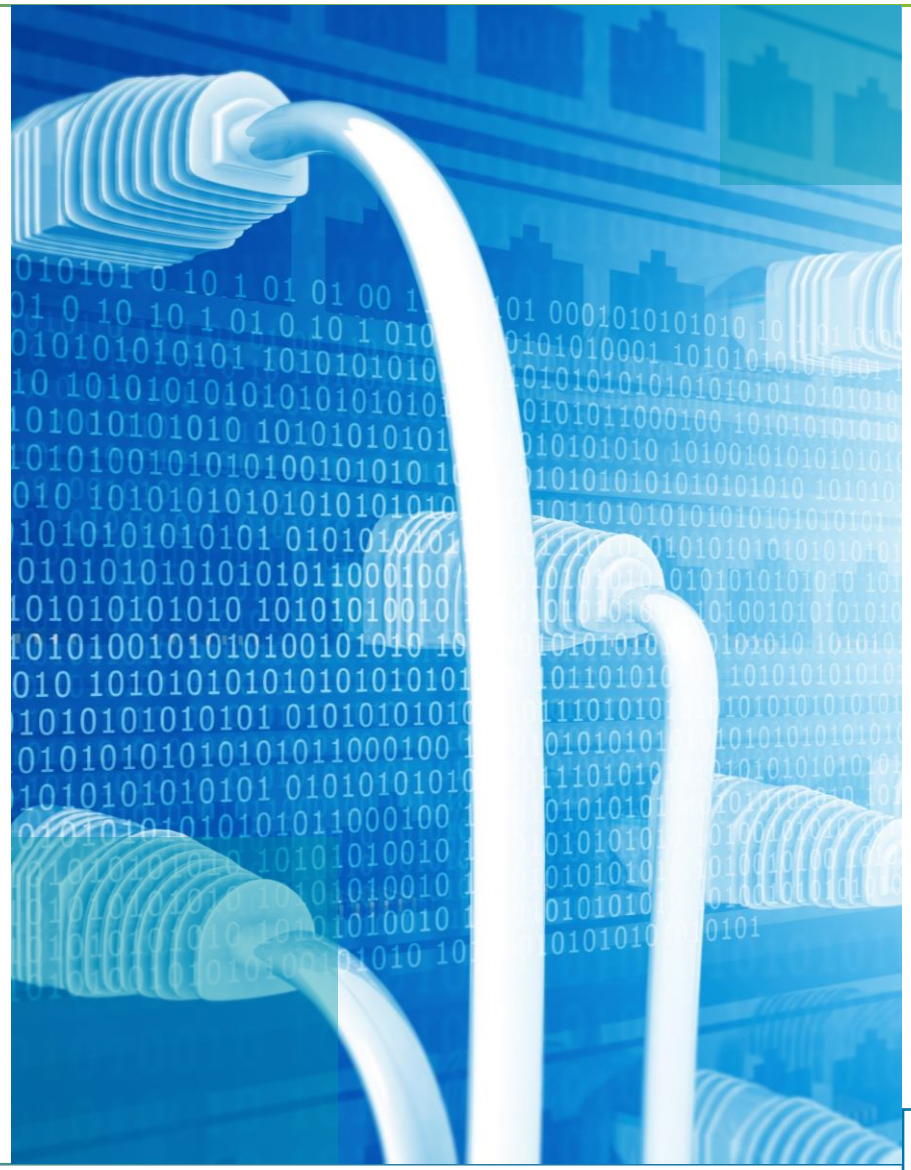
- The router is dedicated to routing and imported by the root AppModule.
- By convention, the module class name is AppRoutingModule and it belongs in the app-routing.module.ts in the src/app directory.
- Run ng generate to create the application routing module.
- The `forRoot()` method supplies the service providers and directives needed for routing and performs the initial navigation based on the current browser URL.
- A typical Angular Route has two properties:

Properties	Details
path	The browser address bar URL matches this string.
component	The component that the router should create when navigating to the route.

The Component Router

</ />

- The Angular router is an official Angular routing library, written and maintained by the Angular core team.
- The duties of a JavaScript router are taken care of by the Angular Router:
  - It activates all required Angular components to compose a page when a user navigates to a certain URL.
  - It lets users navigate from one page to another without page reload.
  - It updates the browser's history so when navigating back and forth between pages the user can use the forward and back buttons.
  - In addition, Angular Router allows us to resolve data before a page is displayed, redirect a URL to another URL, run scripts when a page is activated or deactivated, and lazy load parts of our application.



The Component Router (Cont.)

</>

- To create a route there are three fundamental building blocks.
- In AppModule, import the AppRoutingModule and add it to the imports array.
- Default CLI AppModule with routing:

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule // CLI adds
AppRoutingModule to the AppModule's imports
array
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The Component Router (Cont.)

</>

1. Import RouterModule and Routes into your routing module. CLI application routing module

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from
'@angular/router'; // CLI imports router

const routes: Routes = []; // sets up routes
constant where you define your routes

// configures NgModule imports and exports
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

2. Define your routes in your Routes array. AppRoutingModule (excerpt)

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
];
```

The Component Router (Cont.)

</>

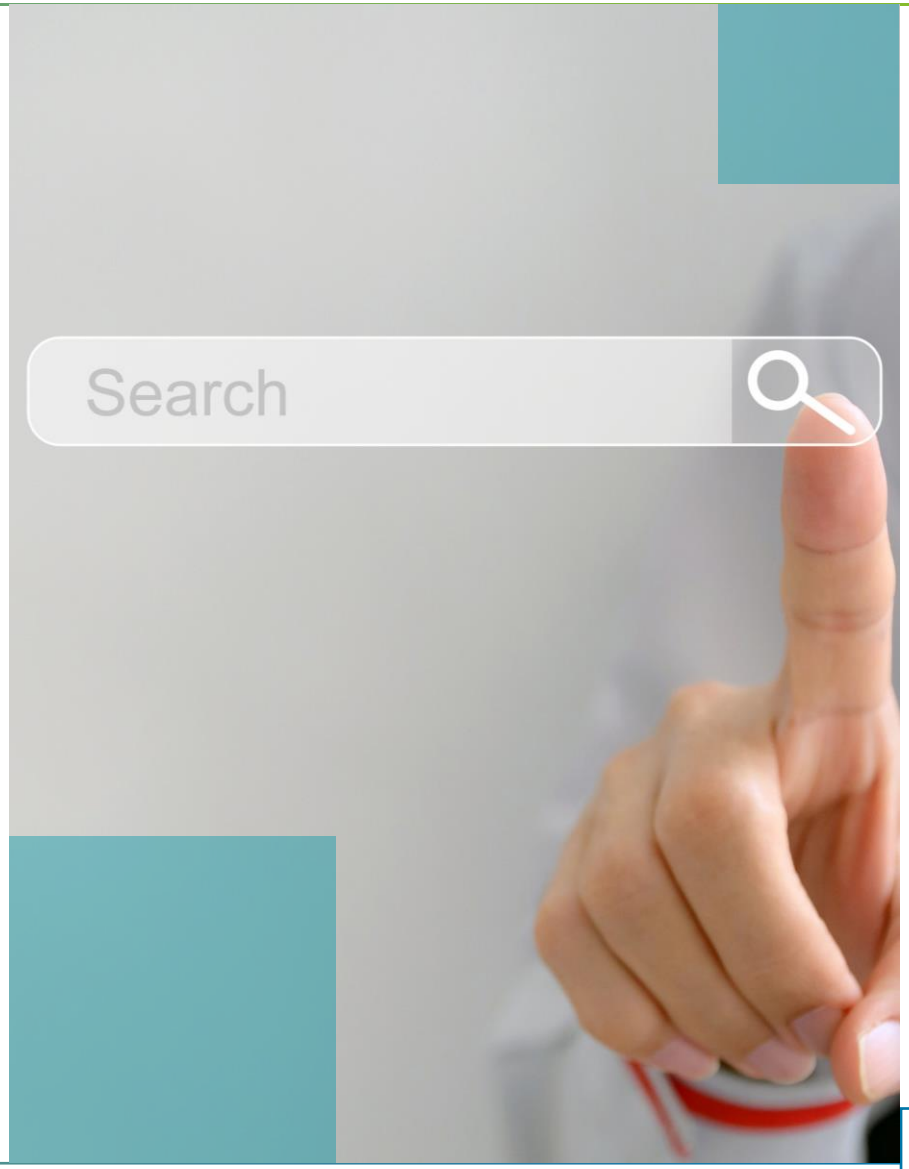
3. Add your routes to your application. Template with router link and router-outlet

```
<h1>Angular Router App</h1>
<!-- This nav gives you links to click, which tells the router which route to use
(defined in the routes constant in  AppRoutingModule) -->
<nav>
  <ul>
    <li><a routerLink="/first-component" routerLinkActive="active"
ariaCurrentWhenActive="page">First Component</a></li>
    <li><a routerLink="/second-component" routerLinkActive="active"
ariaCurrentWhenActive="page">Second Component</a></li>
  </ul>
</nav>
<!-- The routed views render in the <router-outlet>-->
<router-outlet></router-outlet>
```

Traditional Browser Navigation

The Angular router performs the following steps in order when a user navigates to a page:

- The browser URL the user wants to travel to is read.
- A URL redirect is applied (if defined).
- Which router state that corresponds to the URL is identified.
- In the router state the guards that are defined are run.
- The required data for the router state is resolved.
- The Angular components to display the page are activated.
- It manages navigation and When a new page is requested the steps above are repeated.



Component Router Terminology

</>

router service	In our application the global Angular router service
router configuration	Definition of all possible router states our application can be in
router state	At a point in time the state of the router, expressed as a tree of activated route snapshots
activated route snapshot	Provides access to the URL, data, and parameters for a router state node
guard	Script that runs when a route is loaded, activated or deactivated
resolver	Script that fetches data before the requested page is activated
router outlet	Location in the DOM where Angular Router can place activated components

Setting Up the Component Router

< / >

To set up routing in our Angular application, we need to do three things:

1. Create a routing configuration that defines the possible states for our application
2. Import the routing configuration into our application
3. Add a router outlet to tell Angular Router where to place the activated components in the DOM

Configuring Routes

</>

- A list of the URLs we want our application to support is needed to create our routing configuration.
- It is recommended to store the routing configuration for an Angular module with a filename ending in -routing.module.ts in a file by the official Angular style guide. a separate Angular module that has a name ending in RoutingModule is exported with this.
- Our current module is called AppModule, so we create a file `src/app/app-routing.module.ts` and export our routing configuration as an Angular module called AppRoutingModule.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
const routes: Routes = [
  {
    path: '',
    redirectTo: 'tests',
    pathMatch: 'full'
  },
  {
    path: 'tests',
    component: AppComponent
  }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  providers: []
})
export class AppRoutingModule {
}
```

Bootstrapping Routing

</ />

- A way for you to quickly get started with a project is provided with Angular and many other frameworks, bootstrapping is the process of generating framework-specific project startup files.
- Upon startup, the initial bit of code that is executed is bootstrapping. The very first piece of code to run after startup is what the entire operating system depends on to work properly.
- AppModule as a NgModule class are identified by the @NgModule decorator. Angular is told how to launch and compile the application by a metadata object taken by @NgModule.

declarations	This application's lone component
imports	Import BrowserModule to have browser specific services such as DOM rendering, sanitization, and location
providers	Service providers
bootstrap	The root component that Angular creates and inserts into the index.html host web page

Bootstrapping Routing (Cont.)

< / >

The Angular CLI default application created has only one component, AppComponent, it is in both the declarations and the bootstrap arrays.

declarations array	imports array	providers array	bootstrap array
Angular is told which components belong to that module by the module's declarations array.	Appearing exclusively in the @NgModule metadata object is the module's imports array. Here the Angular is told about other NgModules that this particular module needs to function properly.	You list the services the app needs in the providers array. Service are available app-wide when listed here.	By bootstrapping the root AppModule the application launches, which is also known as an entryComponent. Among other things, the bootstrapping process creates the component(s) listed in the bootstrap array and inserts each one into the browser DOM.

Programmatic Navigation

</>

- In Angular, we can also programmatically navigate through a Router service we inject into our component, like so:
- In the example:
  - We added click handlers to each anchor tag to call functions on our HeaderComponent.
  - We inject and store a reference to the Router into our HeaderComponent.
  - We then call the navigate function on the router to navigate between the different URLs.

```
import {Router} from "@angular/router";
@Component({
  selector: 'app-header',
  template: `
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" (click)="goHome()">Search App</a>
  <ul class="nav navbar-nav">
    <li class="nav-item">
      <a class="nav-link" (click)="goHome()">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" (click)="goSearch()">Search</a>
    </li>
  </ul>
</nav>   })
class HeaderComponent {
  constructor(private router: Router) {}

  goHome() {
    this.router.navigate(['']);  }
  goSearch() {
    this.router.navigate(['search']);
  }
}
```

Creating Routes With Route Parameters

</>

- Essential in determining the route and a dynamic part of the Route are the Route parameters.
- For example, consider the following route:

```
{ path: 'product', component: ProductComponent }
```

- The route above only match if the URL is /product
- Our URL should look something like this to retrieve the product details:

```
/product/1
/product/2
```

- The id of the product is where the second URL segment (1 & 2). As per the selected Product the id is dynamic and changes. To handle such a scenario, we can send any dynamic value for a URL segment through an route parameters in the angular router.
- Parameter can be defined by adding a forward slash followed colon and a placeholder (id) as shown below:

```
{ path: 'product/:id', component: ProductDetailComponent }
```

Navigating With Route Parameters

</>

- We need to provide both the route parameter routerLink directive and path.
- This is completed by adding the productID as the second element to the routerLink parameters array, shown below:

```
<a [routerLink]="['/Product', ‘2’]">{{product.name}} </a>
```

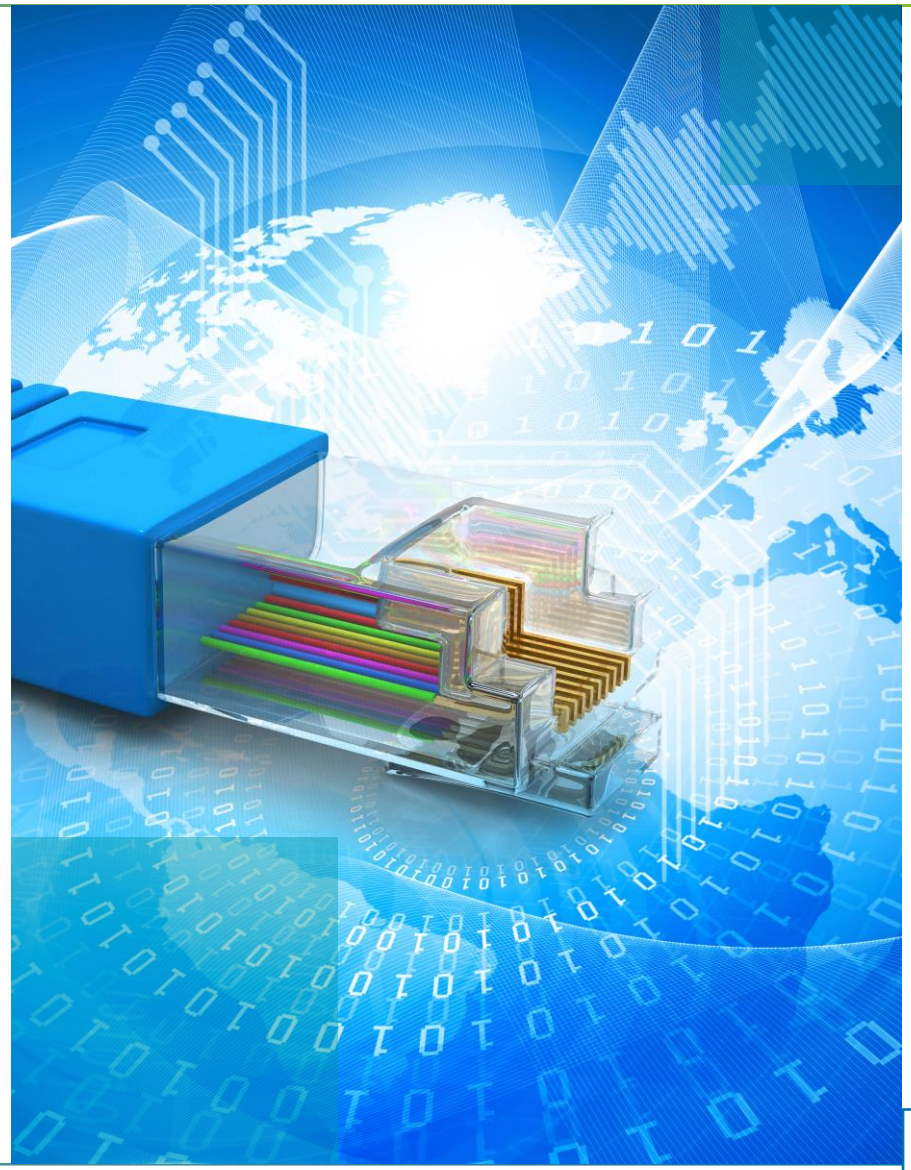
- Which translates to the URL /product/2
- This dynamically takes the value of id from the product object

```
<a [routerLink]="['/Product',
product.productID]">{{product.name}} </a>
```

Retrieving the Route Parameter

</>

- The route parameter needs to extracted from the URL by the component.
- This is completed by the ActivatedRoute service from the angular/router module to get the parameter value.
- **ActivatedRoute**
  - The ActivatedRoute is a service, which keeps track of the currently activated route associated with the loaded Component.
  - To use ActivatedRoute, we need to import it into our component `Import { ActivatedRoute } from '@angular/router'.`
  - Then inject it into the component using dependency injection `Constructor(private _Activatedroute:ActivatedRoute).`
- In the ParamMap object the Angular adds the map of all the route parameters, from the ActivatedRoute service this can be accessed.
- Parameters are easier to work with in the ParamMap, you can use getAll or get methods to get the value of the parameters in the component and use the has method to check if a certain parameter exists.



Retrieving the Route Parameter (Cont.)

</>

ActivatedRoute can be used in two ways to get the parameter value from the ParamMap object.

Using Snapshot

- `this.id=this._Activatedroute.snapshot.paramMap.get("id");`
- The initial value of the route is returned by the snapshot property. To access the value of the id, you can now access the paramsMap array, as shown above

Using Observable

- `this._Activatedroute.paramMap.subscribe(params => { this.id = params.get('id'); });`
- By subscribing to the paramMap observable property of the activateRoute, the value of the id can be retrieved, as shown above

- To retrieve the value of the parameter we usually use the ngOninit life cycle hook, when the component is initialized.
- When the component is again navigated to by the user, Angular reuses the existing instance and does not create a new component. In such instance, the ngOnInit method of the component is not recalled.
- So, you need a way to access the value of the parameter.

Query Parameters

</>

- Optional parameters are allowed to pass across any route in the application by query parameters in Angular.
- Query parameters differ from regular route parameters, which are only available on one route and are not optional.

```
testQuery() {
  this.router.navigate(
    ['/products'],
    { queryParams: { name: 'abc' } }
  );
}

You can also provide multiple query parameters

testQuery() {
  this.router.navigate(
    ['/products'],
    { queryParams: { name: 'abc', 'grade': 'A' }
}
  );
}
```

Guards at Gate/Authentication

- Authentication is the process of matching the pre-defined set of user identities in the system with the visitor of a web application. In another word, it is the process of recognizing the user's identity. Authentication is a very important process in the system concerning security.
- **Guards in Routing:**
  - In a web application, a resource is referred to by URL. Every user in the system will be permitted access to a set of URLs. For instance, an administrator may be assigned all the URLs coming under the administration section.
- Angular provides a concept called Router Guards which can be used to prevent unauthorized access to a certain part of the application through routing. Angular provides multiple guards and they are as follows:

CanActivate	Used to stop the access to a route
CanActivateChild	Used to stop the access to a child route
CanDeactivate	Used to stop ongoing process getting feedback from user. For example, delete process can be stopped if the user replies in negative
Resolve	Used to pre-fetch the data before navigating to the route
CanLoad	Used to load assets

Guards at Gate/Authentication (Cont.)

</>

Create a new service that implements the route guard. It is generally sufficient to call it something like auth-guard.service, but you may call it whatever you want.

```
// src/app/auth/auth-guard.service.ts
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthService } from './auth.service';
@Injectable()
export class AuthGuardService implements CanActivate {
  constructor(public auth: AuthService, public router: Router) {}
  canActivate(): boolean {
    if (!this.auth.isAuthenticated()) {
      this.router.navigate(['login']);
      return false;
    }
    return true;
  }
}
```

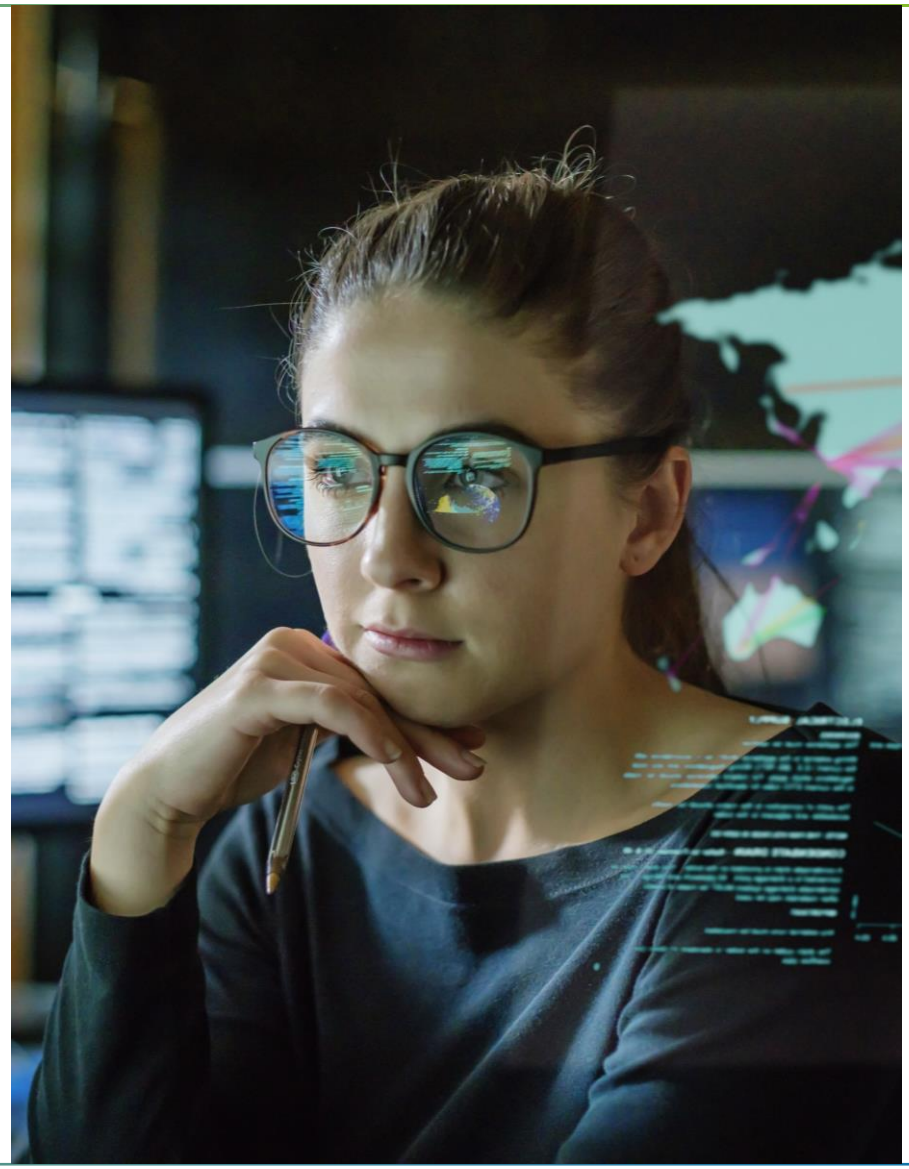
Guards at Gate/Authentication (Cont.)

</>

- The service has a single method canActivate and injects AuthService and Router. To properly implement the CanActivate interface this method is necessary.
- A boolean is returned using the canActivate method, indicating the allowance of navigation to a route. They are re-routed to some other place if the user is not authenticated, in this case, a route called /login.
- Now, you can apply the guard to any routes you wish to protect.

```
// src/app/app.routes.ts
import { Routes, CanActivate } from '@angular/router';
import { ProfileComponent } from
'./profile/profile.component';
import {
  AuthGuardService as AuthGuard
} from './auth/auth-guard.service';
export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'profile',
    component: ProfileComponent,
    canActivate: [AuthGuard]
  },
  { path: '**', redirectTo: '' }
];
```

Modules



Lazy Loading of Module

</>

- By default, NgModules are eagerly loaded, which means that as soon as the application loads, so do all the NgModules, whether or not they are immediately necessary
- You should consider lazy loading for large applications with lots of routes, by design pattern it loads NgModules as needed and Lazy loading helps keep initial bundle sizes smaller, which in turn help decrease load times
- To set up a lazy-loaded feature module, follow below two steps:
  - Create the feature module with the CLI, using the --route flag
  - Configure the routes

  To lazy load Angular modules, use loadChildren (instead of component) in your AppRoutingModule routes configuration as follows.

```
const routes: Routes = [
  {
    path: 'items',
    loadChildren: () =>
import('./items/items.module').then(m =>
m.ItemsModule)
  }
];

In the lazy-loaded module's routing module, add
a route for the component.

const routes: Routes = [
  {
    path: '',
    component: ItemsComponent
  }
];
```

Exporting Modules

</>

- You can organize and streamline your code by creating shared modules. You can put commonly used pipes, components, and directives, into one module and then just that module can be imported wherever you need it in other parts of your application.
- By re-exporting FormsModule and CommonModule, any other module that imports this SharedModule, gains access to directives like NgFor and NgIf from CommonModule and can use [(ngModel)] to bind to component properties, a FormsModule directive.
- Even though the components SharedModule declares may not bind with [(ngModel)] and there may be no need for FormsModule to be imported by SharedModule, FormsModule can still be exported by SharedModule without listing it among its imports. This way, other modules can be given access to FormsModule without having to import it directly into the @NgModule decorator.

Consider following module :

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { CustomerComponent } from
'./customer.component';
import { NewItemDirective } from './new-
item.directive';
import { OrdersPipe } from './orders.pipe';

@NgModule({
 imports:      [ CommonModule ],
 declarations: [ CustomerComponent,
NewItemDirective, OrdersPipe ],
 exports:      [ CustomerComponent,
NewItemDirective, OrdersPipe,
                 CommonModule, FormsModule ]
})
export class SharedModule { }
```

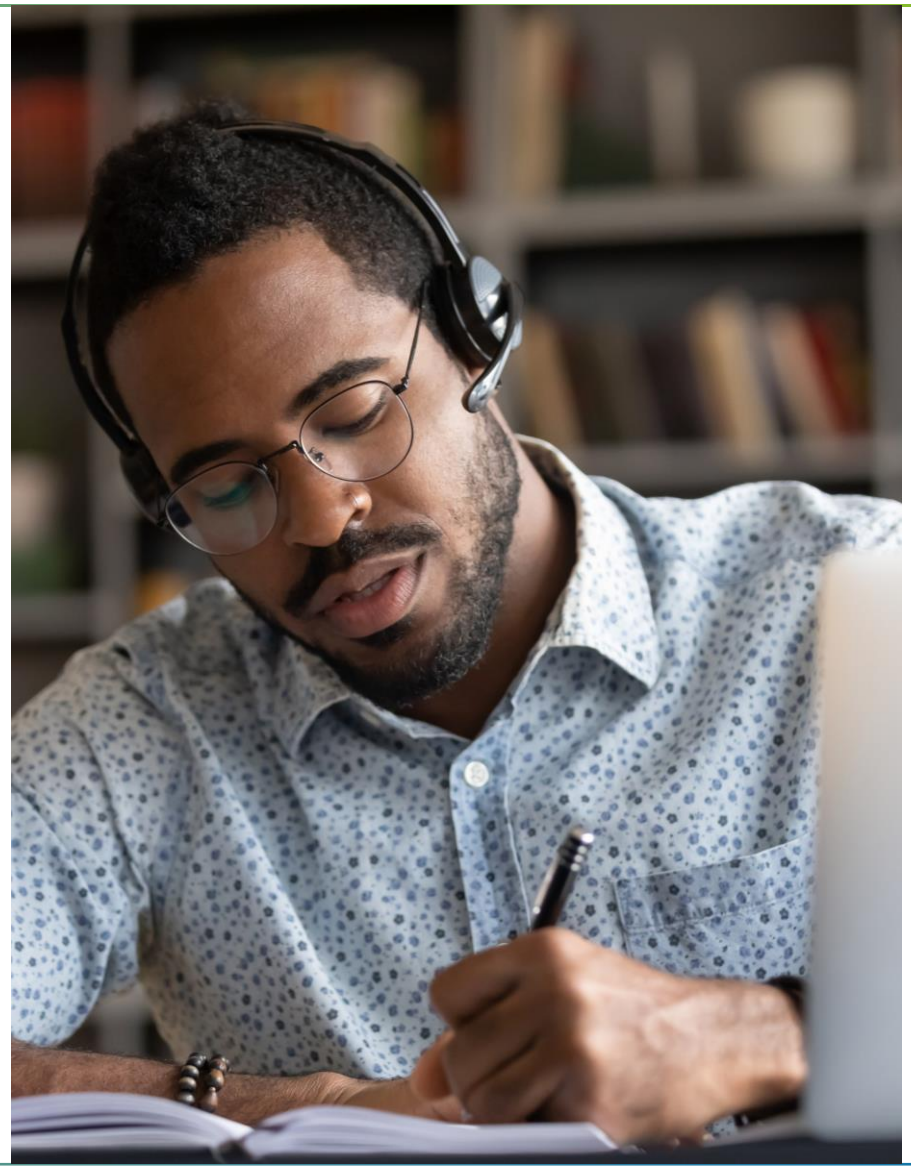
Exporting Services

</>

To export the services, you need to modify the definition of the SharedModule and instead of defining our service in the providers property, a static method called forRoot needs to be created to export the service along with the module itself.

```
import { NgModule, ModuleWithProviders } from
'@angular/core';
import { CounterService } from './counter.service';
@NgModule({})
export class SharedModule {
static forRoot(): ModuleWithProviders {
return {
ngModule: SharedModule,
providers: [CounterService]
};
}
}
```

Hands-On Labs



Hands-On Activity 1

</ >

Activity Details	
Problem Statement	White a program to make an HTTP Post Request to a backend server using an HTTP client.

Summary

Here is the summary of the key learning points.

- Angular is a JavaScript (TypeScript) framework that helps create reactive SPAs.
- A component is a basic building block of an Angular application.
- Data binding automatically keeps your page up-to-date based on your application's state. Data binding is used to specify things such as the source of an image, the state of a button, or data for a particular user.
- Directives are classes that add additional behavior to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see.
- Structural directives are directives that change the DOM layout by adding and removing DOM elements.
- Services in Angular is a class that contains some functionality that can be reused across the application.
- Forms are a crucial part of the web applications through which we get most of our data inputs from users.
- Pipes are used to format the data before being displayed to the user.
- Templates in Angular represent a view whose role is to display data and change the data whenever an event occurs. Its default language for templates is HTML.
- Angular provides a client HTTP API for Angular applications, the HttpClient service class in `@angular/common/http`.
- All of your application's functions exist on a single HTML page In a SPA (Single Page Application). As your application's features are accessed by the user, instead of loading a new page the browser needs to provide only the parts that matter to the user.

Thank You

Deloitte.