Angular

Angular has evolved from AngularJS (v1) to modern component-based versions (v2+), with major updates like v4 skipping v3 for version alignment, v9 introducing the default Ivy engine for better performance, and recent versions (like v19+) focusing on standalone components, Signals, hydration, and developer experience, following a fast release cycle with 6 months of active support and 12 months of LTS (Long Term Support) per major release for stability.

- AngularJS (v1): The original framework (2010), JavaScript-based, using concepts like scopes, directives, and controllers.
- Angular 2+ (2016+): A complete rewrite, introducing TypeScript, component architecture, mobile focus, and the Angular CLI, making it a distinct framework from AngularJS.
- Angular 4 (2017): Skipped v3 to sync with npm packages; focused on performance, smaller compiler, and faster rendering.
- Ivy Engine (v8 Preview, v9 Default): A significant milestone in v9, making Angular apps faster, smaller, and easier to debug by default.

- Angular 12 (2021): Default strict mode, inline Sass, Ivy everywhere, deprecation of IE11 support, faster builds.
- Angular 13 (2021): Standalone Components (now stable), improved SSR, faster server-side rendering.
- Angular 14 (2022): Simplified imports, Angular Material updates, RxJS 7 support.
- Angular 15 (2022): Official Standalone Components, enhanced hydration for SSR.
- Angular 16 (2023): Introduction of Signals, optimized build system, fine-grained change detection.
- Angular 17 (2023): Improved Server-Side Rendering (SSR), automatic tree-shaking, CLI updates.
- Angular 18 (2024): Dynamic component loading, support for modern frameworks, new Material components.

- Release Cycle: New major versions released roughly every 6 months.
- Support Model: 18 months total: 6 months of Active Support (new features, patches) and 12 months of Long-Term Support (LTS, critical fixes only).

Install Angular CLI and setup a new project

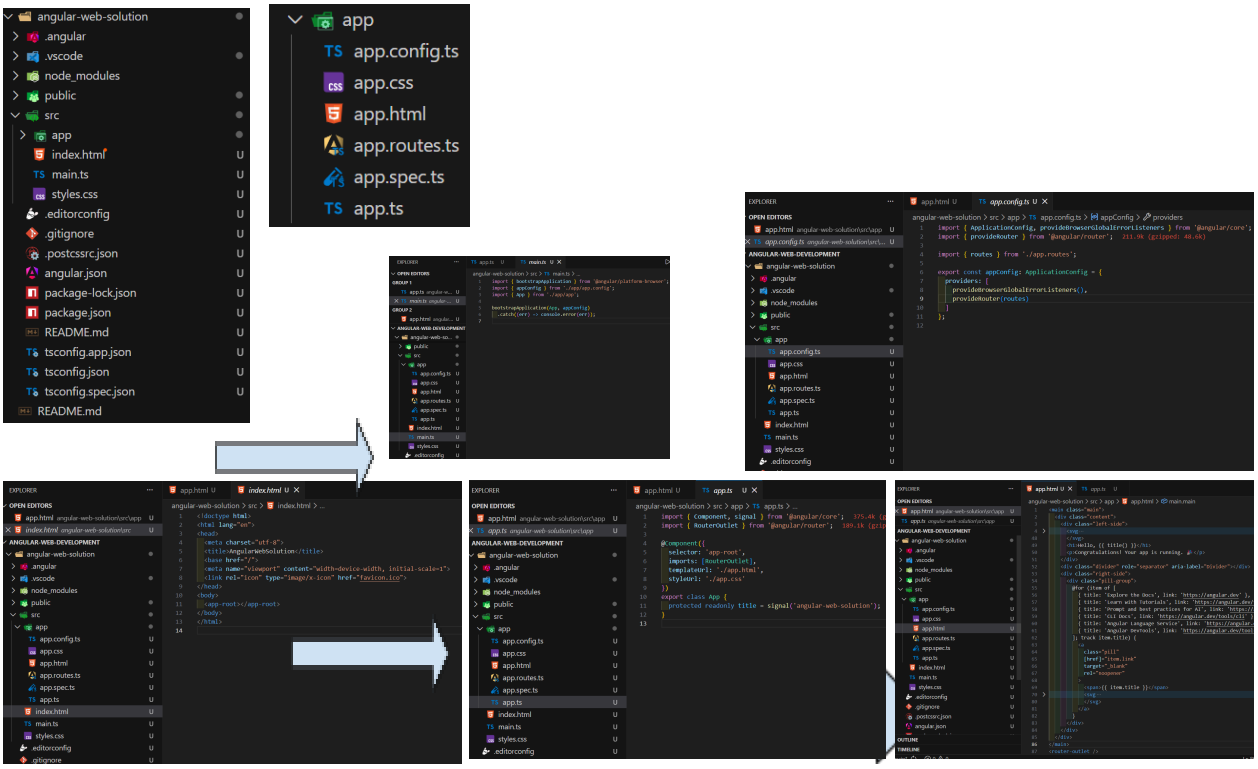
npm install -g @angular/cli
ng v
ng version

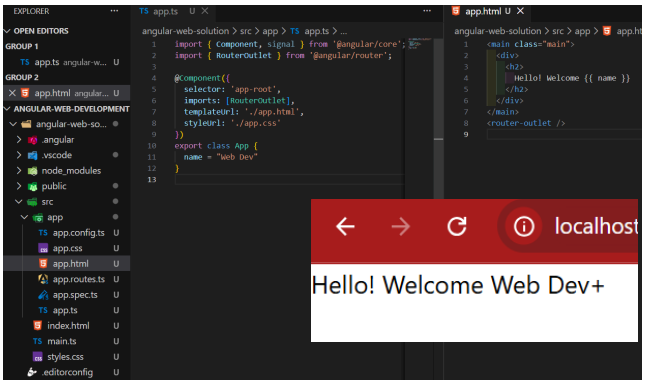
Create new angular project -> ng new <project name>

We can select what css method we need, enable SSR and SSG, AI agents to setup like wise

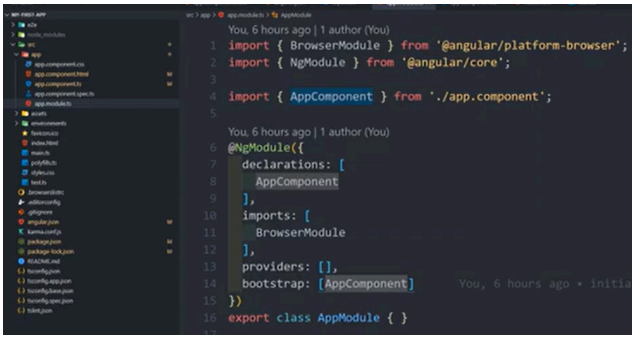
Start the new project -> ng serve

Structure of the project





Angular traditional modules



An Angular module, or NgModule, is a class decorated with @NgModule() metadata that helps organize an application into cohesive, functional blocks. It serves as a compilation context for components, directives, and pipes, and configures the dependency injection system, making the app more scalable and maintainable.

Key Purpose

The primary purposes of NgModule include:

- Organizing code by grouping related components, directives, pipes, and services into logical units based on feature, domain, or workflow..
- Configuring the compiler and injector, telling Angular how to process templates and manage dependencies.
- Facilitating modularity and reusability, allowing these organized blocks of code to be imported and used in other parts of the application or even in different applications.
- Enabling performance optimizations like lazy loading, where modules are loaded only when needed, reducing initial load time.
-

Core Metadata Properties

The @NgModule decorator takes a metadata object that describes the module's contents and dependencies. The most important properties are:

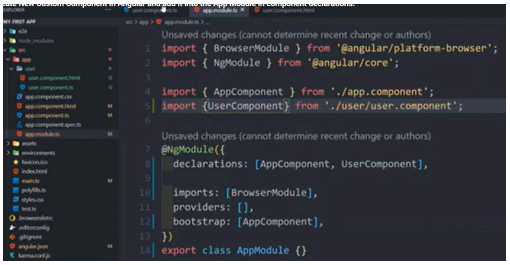
- declarations: Lists the components, directives, and pipes that belong to this module. A declarable class can only belong to one module.
- imports: Imports functionality from other modules whose exported classes are needed by the components within the current module. Examples include built-in Angular modules like BrowserModule, FormsModule, or RouterModule.
- providers: Registers services that the module contributes to the application's global collection of services, making them available for dependency injection throughout the app.
- exports: Specifies a subset of the declared components, directives, and pipes that should be visible and usable by other modules that import the current one.
- bootstrap: Used only in the root module (AppModule) to specify the main application view (root component) that Angular should insert into the index.html host web page to launch the application.
-

Note on Standalone Components

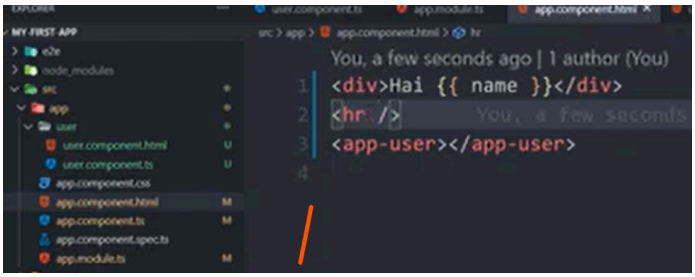
The Angular team now recommends using standalone components for all new code, which removes the need for NgModule in many cases. However, NgModule remains essential for understanding existing projects, utilizing third-party libraries that still rely on the modular structure, and enabling features like lazy loading through the router configuration

Custom component declaration with ngModule

Defining new custom component for the project.

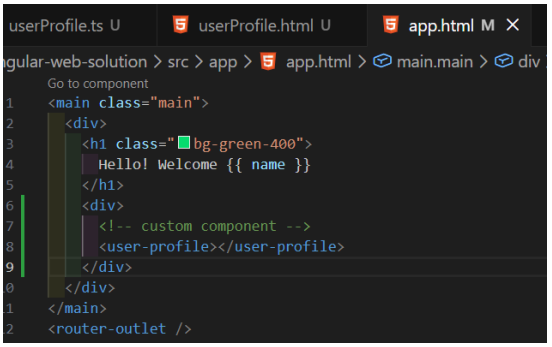


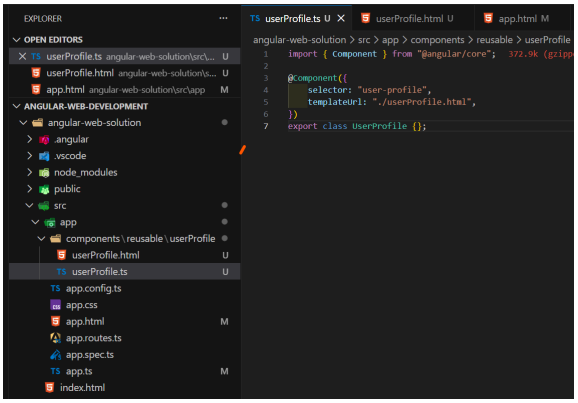
Register inside the app.module. Only one app.module

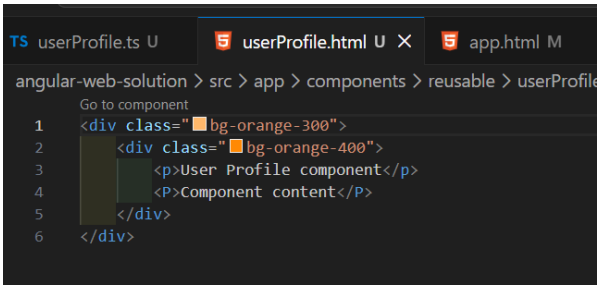


Using inside the index file

Modern







Creating new components by Angular CLI —> ng g c <component name>
Ng g c <path> >>> components/reusable/user-list
G-generate / c - components

We can write template instead of templetUrl. Template contains html content. It is good to maintain template instead of templateUrl only if your html content has only one or two lines.

styleUrls: []
Styles: [`<css styling>`]

Selectors

Selector value must be unique

Selector: [“app-custom-selector”] ===>>> this will be an attribute
<div app-custom-selector >

Selector: “.app-custom-selector”   ===>>> this will be a class name

These selector names can be taken to css stylings also. All tag selector name, class or attribute

Data binding and string interpolation

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

@Component({
    selector: "app-user-profile",
    templateUrl: "./user-profile.html",
})
export class UserProfile {
    id: number = 32;
    name: String = "Jhon Doe";

    getUserName () {
        return this.name;
    }
};
```

```
<div class="bg-orange-300">
    <div class="bg-orange-400">
        <p>User Profile component</p>
        <P>User id = {{ id }} >>>>>> user name = {{ getUserName()}}</P>
    </div>
</div>
```

Property binding

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

@Component({
    selector: 'app-user-profile',
    templateUrl: './user-profile.html',
})
export class UserProfile {
    id: number = 32;
    name: string = 'John Doe';
    isDisabled = false;

    constructor(private cdr: ChangeDetectorRef) {
        setTimeout(() => {
            this.isDisabled = true;
            this.cdr.markForCheck();   // force re-render
        }, 1000);
    }

    getUserName() {
        return this.name;
    }

    showAlert() {
        alert('Hello angular!');
    }
}
```

```html
<div class="bg-orange-300">
    <div class="bg-orange-400">
        <p>User Profile component</p>
        <p>User id = {{ id }} >>>>>> user name = {{ getUserName() }}</p>

        <button
            type="button"
            class="px-8 py-2 rounded-lg bg-purple-400 cursor-pointer"
            [disabled]="isDisabled"
            (click)="showAlert()"
        >
            {{ isDisabled }}
        </button>
        <div>{{ isDisabled }}</div>
    </div>
</div>
```

Classic Angular behavior (Angular ≤ 15)

Default behavior

- Angular uses Zone.js
- Any async task (setTimeout, Promise, XHR, events)
- Automatically triggers change detection

So this always re-rendered by default:

Modern Angular behavior (Angular 16 / 17 / 18)

Angular intentionally changed the architecture for performance.

Key change

Angular now supports zone-less change detection

What this means:

- setTimeout() does NOT trigger re-render
- Angular will not scan the component tree automatically
- You must be explicit about updates

In large enterprise apps:

- Zone.js causes unnecessary re-renders
- Every async task triggers full tree checks
- Performance suffers badly

Modern Angular favors:
Predictability
Explicit updates
High performance

Angular's official industrial approach

Angular does NOT expect you to use ChangeDetectorRef everywhere
Instead, Angular promotes **reactive state. Use RxJS / Signals instead of mutable state or Observables**

Angular does NOT re-render automatically anymore. Angular re-renders when state changes through Angular mechanisms

Angular mechanisms:

Signals
Observables (async pipe)
Events (click, input)
Input reference changes

Event binding

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

@Component({
    selector: 'app-user-profile',
    templateUrl: './user-profile.html',
})
export class UserProfile {
    id: number = 32;
    name: string = 'John Doe';
    cdr: ChangeDetectorRef;
    value = 0;

    constructor(private detectionRef: ChangeDetectorRef) {
        this.cdr = detectionRef
    }

    getUserName() {
        return this.name;
    }

    incrementVal() {
        this.value += 1;
        this.cdr.markForCheck(); // force re-render
    }
}
```

```
<div class="bg-orange-300">
    <div class="bg-orange-400">
        <p>User Profile component</p>
        <p>User id = {{ id }} >>>>>> user name = {{ getUserName() }}</p>

        <button
            type="button"
            class="px-8 py-2 rounded-lg bg-purple-400 cursor-pointer"
            (click)="incrementVal()"
        >
            increment +
        </button>
        <div class="mt-5">VALUE: {{ value }}</div>
    </div>
</div>
```

Binding on click event to the button, actually in here we dont need force re render since earlier angular versions re render in events.

Input control

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

@Component({
    selector: 'app-user-profile',
    templateUrl: './user-profile.html',
})
export class UserProfile {
    id: number = 32;
    name: string = 'John Doe';
    value = 0;

    getUserName() {
        return this.name;
    }

    onUpdateValue(event: any) {
        this.value = event.target.value
    }

}
```

```
<div class="bg-orange-300">
    <div class="bg-orange-400">
        <p>User Profile component</p>
        <p>User id = {{ id }} >>>>>> user name = {{ getUserName() }}</p>

        <input
            type="text"
            class="px-8 py-2 rounded-lg bg-purple-400"
            (input)="onUpdateValue($event)"
        >
        <div class="mt-5">VALUE: {{ value }}</div>
    </div>
</div>
```

$event object pass the eventdetails to the functions

One way data binding and two way data binding

In Angular, the key difference is the direction of data flow: one-way binding is unidirectional (either component-to-view or view-to-component), while two-way binding is bidirectional, automatically synchronizing changes between the component's data model and the view (UI).

One-Way Data Binding

Data flows in a single direction, which makes the flow predictable and easier to debug.

Type	Direction	Syntax	Description
Interpolation	Component to View	{{ value }}	Displays component property values in the template.
Property Binding	Component to View	[property]="value"	Sets an element's property to the component's value.
Event Binding	View to Component	(event)="handler()"	Listens for a DOM event (e.g., a user click) and executes component code.

Pros & Cons:

- Pros: Easier to trace and debug data changes, prevents accidental data overwrites, and generally offers better performance for read-only data.
- Cons: Requires more boilerplate code for interactions where the view needs to update the component data explicitly.

Two-Way Data Binding

Data flows in both directions simultaneously. Changes in the component automatically update the view, and changes in the view (from user input) automatically update the component's model.

- Syntax: [(ngModel)]="value" (known as the "banana in a box" syntax).
- Usage: Primarily used in form inputs where real-time synchronization of user input is needed. It requires importing the FormsModule in the application module to use ngModel.

Pros & Cons:

- Pros: Less boilerplate code, perfect for interactive forms, ensures the UI and data model are always in sync.
- Cons: Can lead to complex code and make data flow harder to track in large applications, potentially causing performance issues due to constant synchronization.

Summary Comparison

Feature	One-Way Binding	Two-Way Binding
Data Flow	Unidirectional (one way)	Bidirectional (two way)
Debugging	Easier to debug and track data flow	More complex to track data origin
Syntax Examples	{{}}, [], ()	[(ngModel)]
Use Cases	Displaying read-only data, dashboards, stateless interactions	Interactive forms, real-time user input synchronization
Performance	Generally better performance	Potential performance overhead in complex UIs

Best Practice: Prefer one-way binding whenever possible and only use two-way binding when necessary, such as for form inputs, to maintain a clear and manageable data flow in your Angular application

Angular two-way binding is a mechanism that synchronizes data between the component class (model) and the template (view), so that changes in one are immediately reflected in the other. It is primarily achieved using the built-in [(ngModel)] directive for form elements, a syntax informally known as the "banana-in-a-box"

Two-way binding in modern Angular is essentially syntactic sugar for a combination of two one-way bindings:

- Property Binding ([]): Data flows from the component to the view, setting an element's property (e.g., [value]="userName").
- Event Binding (()): Data (user input) flows from the view to the component, updating the component's property in response to an event (e.g., (input)="userName = $event.target.value").

The [(ngModel)] syntax combines these two, automatically handling the flow in both directions for form controls.

Implementation with [(ngModel)]

To use two-way binding with standard HTML form elements like <input>, <select>, or <textarea>, you need to follow these steps:

1. Import FormsModule: In your main application module (e.g., app.module.ts), you must import the FormsModule from @angular/forms.

   ```
   ——————————————————————————————————————————
   import { FormsModule } from '@angular/forms';

   @NgModule({
     imports: [
       // ... other imports
       FormsModule
     ],
     // ...
   })
   export class AppModule { }
   ——————————————————————————————————————————
   ```

2. Use the [(ngModel)] syntax: In your component's template, bind a component property using the [(ngModel)] syntax.In the component's TypeScript file (e.g., app.component.ts):

   ```
   ——————————————————————————————————————————
   export class AppComponent {
     userName: string = 'Initial Name';
   }
   ——————————————————————————————————————————
   ```

3. In the component's HTML file (e.g., app.component.html):

   ```
   ——————————————————————————————————————————
   <input type="text" [(ngModel)]="userName">
   <p>Current User Name: {{ userName }}</p>
   ——————————————————————————————————————————
   ```

As you type into the input field, the userName property in the component updates, and the interpolated value in the <p> tag updates instantly.

Custom Two-Way Binding (Parent/Child Components)

You can also implement custom two-way binding between parent and child components using @Input() and @Output() decorators.

- The @Input() property holds the data passed from the parent.
- The @Output() property must be named with the input property's name followed by the suffix Change (e.g., if input is size, output is sizeChange). It uses an EventEmitter to emit updates to the parent.

This setup allows the parent component to use the same [(propertyName)] "banana-in-a-box" syntax for the custom component.

You must import the FormsModule from @angular/forms into the relevant module file (app.module.ts or a feature module) or directly into the component if it's a standalone component.
In a non-standalone component (using NgModules):
In your app.module.ts (or relevant feature module), add the import and include it in the @NgModule's imports array

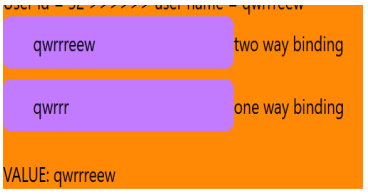
user-profile.ts

```
import { Component, ChangeDetectorRef } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
    imports: [FormsModule],
    selector: 'app-user-profile',
    templateUrl: './user-profile.html',

})
export class UserProfile {
    id: number = 32;
    name: string = 'John Doe';
    value = 0;

    getUserName() {
        return this.name;
    }

    onUpdateValue(event: any) {
        this.name = event.target.value
    }

}
```

User-profile.html

```
<div class="bg-orange-300">
    <div class="bg-orange-400">
        <p>User Profile component</p>
        <p>User id = {{ id }} >>>>>> user name = {{ getUserName() }}</p>

        <input
            type="text"
            class="px-8 py-2 rounded-lg bg-purple-400"
            [(ngModel)]="name"
        ><span>two way binding</span>
        <br/>
        <input
            type="text"
            class="mt-2 px-8 py-2 rounded-lg bg-purple-400"
            (input)="onUpdateValue($event)"
        ><span>one way binding</span>
        <div class="mt-5">VALUE: {{ name }}</div>
    </div>
</div>
```

Look at this different between two inputs about synchronizing the current value of the variable, only the two way binding synchronous both directions, During the change name of the first input, second input's value not get update, Because of it is one way data binding. It is responds only to data modifications and not to read data



ngIf Structural Directive

```
<div *ngIf="isUserCreated">
  User is created and the username is {{ userName }}
</div>
```

This directive was deprecated by now

```
<div *ngIf="isUserCreated; else noUser">
  User is created and the username is {{ userName }}
</div>

<ng-template #noUser>
  <div>no User is Created</div>
</ng-template>
```

ngStyle Directive

```
<div class="my-2" [ngStyle]="{ 'background-color': getColor() }">
  {{ "User" }} with ID {{ userId }} is {{ getUserStatus() }}
</div>
```

```
getColor() {
  if (this.userStatus === 'online') {
    return 'green';
  }
  return 'red';
}
```

ngClass attribute Directive

```
@Component({
  selector: 'app-user',
  templateUrl: './user.component.html',
  styles: [
    `
      .offline {
        color: white;
      }
    `,
  ],
})
```

```
<div
  class="my-2"        You, a few seconds ago • Uncommitted chang
  [ngStyle]="{ backgroundColor: getColor() }"
  [ngClass]="{ offline: userStatus === 'offline' }"
>
  {{ "User" }} with ID {{ userId }} is {{ getUserStatus() }}
</div>
```

ngFor Structural Directive

```
export class UsersComponent implements OnInit {
  allowNewUser = false;
  userName = 'Test User';
  isUserCreated = false;
  users = ['user1', 'user2'];

  constructor() {
    setTimeout(() => {
      this.allowNewUser = true;
    }, 3000);
  }

  changeUserCreatedStatus() {
    this.isUserCreated = true;
    this.users.push(this.userName);          You, 2
  }
```

```
<app-user *ngFor="let user of users"></app-user>
```

Send Data from Parent to Child Component using Custom Properties with @Input()

```
@Component({
  selector: 'app-user',
  templateUrl: './user.component.html',
  styleUrls: ['./user.component.css'],
})
export class UserComponent implements OnInit {
  @Input() userName: string;

  constructor() {}

  ngOnInit(): void {}
}
```

```
export class UserComponent implements OnInit {
  @Input('user') userName: string;

  constructor() {}

  ngOnInit(): void {}
}
```

```
<div class="mt-3">
  <h3>Users List</h3>
  <app-user *ngFor="let user of usersList" [userName]="user"></app-user>
</div>
```

```
<div class="mt-3">
  <h3>Users List</h3>
  <app-user *ngFor="let user of usersList" [user]="user"></app-user>
</div>
```

Send Data from Child to Parent Component. Binding to Custom Events using @Output

```
@Component({
  selector: 'app-users',
  templateUrl: './users.component.html',
  styleUrls: ['./users.component.css'],
})
export class UsersComponent implements OnInit {
  usersList = [];
  constructor() {}

  ngOnInit(): void {}

  onUserAdded(event: string) {
    this.usersList.push(event);
  }
}
```

Parent component

```
@Component({
  selector: 'app-add-user',
  templateUrl: './add-user.component.html',
  styleUrls: ['./add-user.component.css'],
})
export class AddUserComponent implements OnInit {
  userName: string;
  @Output() userAdded = new EventEmitter<string>();
  constructor() {}

  ngOnInit(): void {}

  onUserAdded() {
    this.userAdded.emit(this.userName);
  }
}
```

Adduser component - Child

Creating event, emit this event whenever specified action happened. Whenever this event emit parent component get notified

```
<app-add-user (userAdded)="onUserAdded($event)"></app-add-user>

<div class="mt-3">
  <h3>Users List</h3>
  <app-user *ngFor="let user of usersList" [user]="user"></app-user>
</div>
```

Catch the event. Then invoke the parent method. Pass the event to the parent method.

View Encapsulation in Angular. Difference between Emulated, None, and Shadow Dom

```
@Component({
  selector: 'app-users',
  templateUrl: './users.component.html',
  styleUrls: ['./users.component.css'],
  encapsulation: ViewEncapsulation.Emulated,
})
export class UsersComponent implements OnInit {
  usersList = [];
  constructor() {}

  ngOnInit(): void {}

  onUserAdded(event: string) {
    this.usersList.push(event);
  }
}
```

Local References in Angular. Access the HTML Element in the Typescript file

```
<div class="form-group">
  <h3>Add User</h3>
  <label>User Name: </label>
  <input type="text" class="form-control" #userInput />
  <div class="mt-2">
    <button class="btn btn-primary" (click)="onUserAdded(userInput)">
      Add User
    </button>
  </div>
</div>
```

```
@Component({
  selector: 'app-add-user',
  templateUrl: './add-user.component.html',
  styleUrls: ['./add-user.component.css'],
})
export class AddUserComponent implements OnInit {
  @Output() userAdded = new EventEmitter<string>();
  constructor() {}

  ngOnInit(): void {}

  onUserAdded(input: HTMLInputElement) {
    this.userAdded.emit(input.value);
  }
}
```

Access HTML Elements in The DOM & Template with @ViewChild and the type ElementRef

```
export class AddUserComponent implements OnInit {
  @Output() userAdded = new EventEmitter<string>();

  @ViewChild('userInput') userInput: ElementRef;
  constructor() {}

  ngOnInit(): void {}

  onUserAdded() {
    console.log(this.use this: this
    this.userAdded.emit(this.userInput.nativeElement.value);
  }
}
```

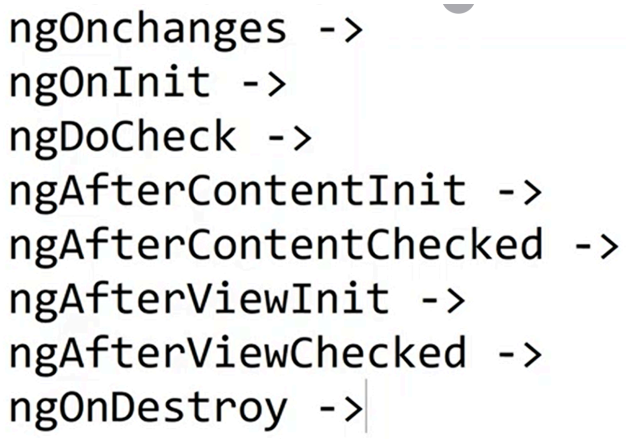
Projecting the HTML Content written between the component using ng-content

```
<div class="mt-3">
  <h3>Users List</h3>
  <app-user *ngFor="let user of usersList" [user]="user">
    <div>Inserting that is written between the component</div>
  </app-user>
</div>
```

```
<app-user *ngFor="let user of usersList" [user]="user">
  <p>Username: {{ user }}</p>
</app-user>
</div>
```

```
<div>
  <ng-content></ng-content>
</div>
```

Component Life cycle methods



Angular calls the following lifecycle hook methods in this order:

- constructor(): The standard TypeScript class constructor runs first when Angular instantiates the component. It should be used only for dependency injection and initial local variable assignment, not complex logic or data fetching.
- ngOnChanges(): Called before ngOnInit (if the component has data-bound inputs) and whenever one or more data-bound input properties change. It receives a SimpleChanges object detailing the current and previous values.
- ngOnInit(): Called once after the first ngOnChanges(). This is the ideal place for component initialization logic, such as fetching initial data from an API or setting up subscriptions.
- ngDoCheck(): Called immediately after ngOnChanges() and ngOnInit(), and then during every subsequent change detection run. It allows for custom change detection logic but should be used sparingly due to potential performance impacts.
- ngAfterContentInit(): Called once after Angular projects external content into the component's view using <ng-content>. This hook is useful for performing actions after the projected content has been initialized.
- ngAfterContentChecked(): Called after ngAfterContentInit() and every subsequent ngDoCheck() to respond after the projected content has been checked for changes.
- ngAfterViewInit(): Called once after Angular initializes the component's view and child views. This hook is used to access and manipulate view-related elements, such as those queried with @ViewChild or @ViewChildren.
- ngAfterViewChecked(): Called after ngAfterViewInit() and every subsequent ngAfterContentChecked(). It is used to respond to changes after the component's view and child views have been checked.
- ngOnDestroy(): Called just before Angular destroys the component instance. This is the final hook, used for crucial cleanup tasks like unsubscribing from Observables, detaching event handlers, and stopping timers to prevent memory leaks

Subsequent changes (new @Input property added name, when specific button click name value got changed -> @Input() name: string -> this name value coming from parent component

Ngdocheck calls for the most first time and whenever every action. Event get triggered, maybe leads to performance issues when using

Getting access to the ng-content HTML template using @ContentChild

Angular Directives

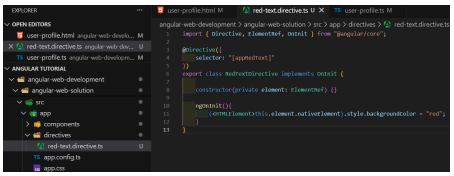
Angular directives are classes that add additional behavior to elements in your Angular applications, allowing you to manipulate the Document Object Model (DOM) and create reusable functionalities. allowing developers to extend HTML's capabilities and dynamically manipulate the Document Object Model (DOM).

There are three main types of directives in Angular

- **Component Directives:** These are the most common and fundamental building blocks of an Angular application. Components are essentially directives with their own templates and encapsulated logic, defined using the @Component() decorator.
- **Structural Directives:** These directives change the structure or layout of the DOM by adding, removing, or manipulating elements. They are easily recognizable in templates by a leading asterisk (*).
    - *ngIf: Conditionally adds or removes an element from the DOM based on whether an expression is true or false.
    - *ngFor: Repeats a portion of the template once for each item in a list or collection.
    - *ngSwitch: A set of cooperating directives (*ngSwitchCase, *ngSwitchDefault) that display one element from a set of choices based on a match condition.
- **Attribute Directives:** These directives modify the appearance or behavior of an existing element, component, or another directive. They are applied as attributes within HTML tags and are wrapped in square brackets [] when used in templates.
    - [ngClass]: Adds or removes a set of CSS classes dynamically.
    - [ngStyle]: Adds or removes a set of inline HTML styles dynamically.
    - [ngModel]: Provides two-way data binding for form elements, syncing data between the model and the view.

Developers can also create custom directives (both attribute and structural) to encapsulate specific, reusable behaviors across their applications

Create Basic Custom attribute Directive in Angular



// creating directive ng g d <directive name/path>

```
  <p appRedText>User Profile component</p> // inside component html template
```

```
import { RedTextDirective } from "../../../directives/red-text.directive";

@Component({

    imports: [FormsModule, RedTextDirective], // inside component ts file
```

Using rendered2

```
Directive({
 selector: '[appRendererHighlight]',
)
xport class RendererHighlightDirective implements OnInit {
 constructor(private element: ElementRef, private renderer: Renderer2) {}

 ngOnInit() {
   this.renderer.setStyle(
     this.element.nativeElement,
     'background-color',
     'green'
   );
 }
```

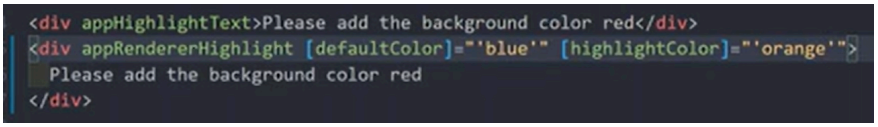
Accessing the events for the directive element using HostListener

```
@Directive({
  selector: '[appRendererHighlight]',
})
export class RendererHighlightDirective implements OnInit {
  constructor(private element: ElementRef, private renderer: Renderer2) {}

  ngOnInit() {}

  @HostListener('mouseenter') onmouseover(event: Event) {
    this.renderer.setStyle(
      this.element.nativeElement,
      'background-color',
      'red'
    );
  }

  @HostListener('mouseleave') onmouseleave(event: Event) {
    this.renderer.setStyle(
      this.element.nativeElement,
      'background-color',
      'red'
    );
  }
}
```

Using HostBinding Decorator to bind the Properties for the Directive element

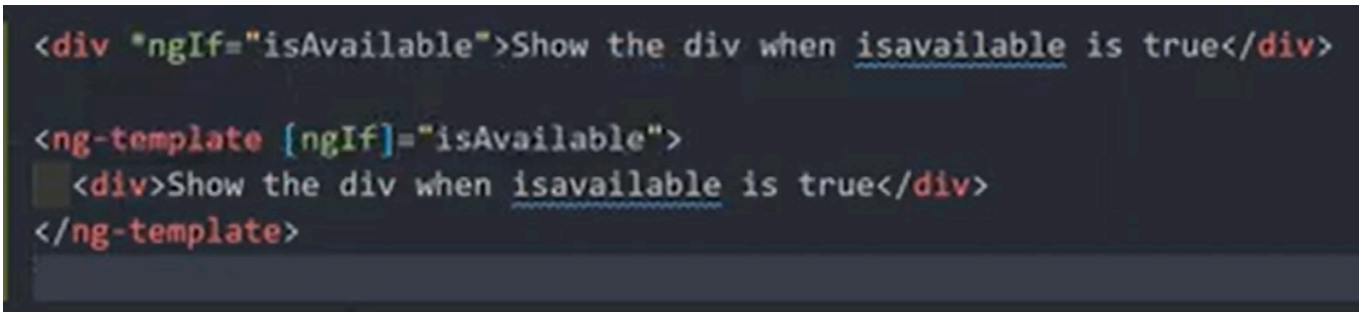
```
You, a few seconds ago | 1 author (You)
@Directive({
  selector: '[appRendererHighlight]',
})
export class RendererHighlightDirective implements OnInit {
  @HostBinding('style.backgroundColor') color: string;
  constructor(private element: ElementRef, private renderer: Renderer2) {}

  ngOnInit() {}                (method) RendererHighlightDirective.onmouseov
                               : Event): void
  @HostListener('mouseenter') onmouseover(event: Event) {
    this.color = 'red';
  }

  @HostListener('mouseleave') onmouseleave(event: Event) {
    this.color = 'yellow';
  }

  @HostListener('click') onclick(event: Event) {
    this.color = 'pink';       You, a few seconds ago • Uncommitted change
  }
}
```

Sending input data to the Directives as Input properties

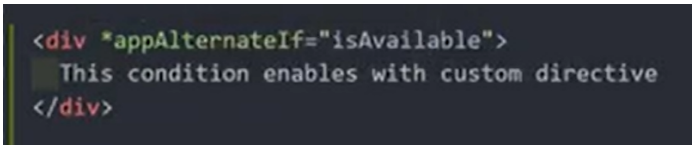


Structural directives

When using directives with star notation, angular internally converted into ng-template element like this



Creating our own structural directive



Import the custom directive from .ts file. ⇒ using onChange is prefer instead of onInit

How to use ngSwitch, ngSwitchCase, ngSwitchDefault Directive

Services Introduction. Create a Simple Service and use the service in Components

An Angular service is a TypeScript class decorated with @Injectable() that provides reusable logic or data-handling functionality to components and other parts of an application. Services are a fundamental part of Angular's modular design, allowing components to focus solely on the user experience and view logic

Key Concepts

- Modularity and Reusability: Services help separate concerns. Instead of putting data fetching, validation, or logging logic in a component, you put it in a service, which can then be used by any number of components, directives, or other services.
- Dependency Injection (DI): This is the mechanism Angular uses to provide components with access to services. When a component declares a dependency on a service (usually in its constructor), Angular's injector system supplies an instance of that service.
- @Injectable() Decorator: This decorator tells Angular that a class can be injected as a dependency. By default, using @Injectable({ providedIn: 'root' }) registers the service at the application level, creating a single, shared (singleton) instance used throughout the entire application.
- Scope: You can control the scope of a service.
    - Root-level: A single, application-wide instance (singleton).
    - Component-level: A new instance of the service is created for each new instance of that component. This is useful for component-specific state or isolated functionality.

Common Use Cases

Services are ideal for tasks that don't involve the UI:

- Fetching and sharing data from a server (e.g., using HttpClient).
- Managing application state across multiple components.
- Handling user authentication and authorization.
- Providing common utility functions like data formatting or validation.
- Centralized logging and error handling

Create custom service, Dependency injecting and invoking the service

Services/report.service.ts

```
export class ReportService {
    instantReport(status: string): void {
        alert(status);
    }
}
```

Dependency injection and Invoking service

```
import { Component } from '@angular/core';
import { RedTextDirective } from "../../../directives/red-text.directive";
import { ReportService } from '../../../services/report.service';

@Component({
  selector: 'app-service-portal',
  imports: [RedTextDirective],
  templateUrl: './service-portal.html',
  styleUrl: './service-portal.css',
  providers: [ReportService]
})
export class ServicePortal {

  constructor(private reportService: ReportService) {}

  invokeService() {
    this.reportService.instantReport("Service invoked successfully")
  }
}
```

Pass data from services to the components

```
<div class="bg-green-200 rounded-xl border border-gray-400">
    <div class="bg-green-200 rounded-xl">
        <p class="rounded-t-xl text-center py-3" appRedText>Order list component</p>
        <div class="p-5">
            <!-- <p>1</p> -->
            @for (item of orders; let i = $index; track item) {
                <app-order></app-order>
            } @empty {
                <li>No items found.</li>
            }
        </div>
    </div>
</div>
// traverse list and render list according to modern angular (21)
```

Order service

```
export class OrderService {
    orders: { name: string; price: number }[] = [
        {
            name: 'Order Name 1',
            price: 23,
        },
        {
            name: 'Order Name 2',
            price: 43,
        },
    ];
}
```

Order list component ts

```
import { Component, OnInit } from '@angular/core';
import { RedTextDirective } from '../../../directives/red-text.directive';
import { Order } from "../order/order";
import { OrderService } from '../../../services/order.service';

@Component({
  selector: 'app-order-list',
  imports: [RedTextDirective, Order],
  templateUrl: './order-list.html',
  styleUrl: './order-list.css',
  providers: [OrderService]
})
export class OrderList implements OnInit {

  orders: { name: string, price: number}[] = [];

  constructor(private orderService: OrderService) {}

  ngOnInit() {
    this.orders = this.orderService.orders;
  }
}
```

Order list component template

```
<div class="bg-green-200 rounded-xl border border-gray-400">
    <div class="bg-green-200 rounded-xl">
        <p class="rounded-t-xl text-center py-3" appRedText>Order list component</p>
        <div class="p-5">
            <!-- <p>1</p> -->
            <div class="flex flex-col gap-2">
                @for (item of orders; let i = $index; track item) {
                    <app-order [order]="item"></app-order>
                } @empty {
                    <li>No items found.</li>
                }
            </div>
        </div>
    </div>
</div>
```

Order component ts

```
import { Component, Input } from '@angular/core';
import { RedTextDirective } from '../../../directives/red-text.directive';

@Component({
  selector: 'app-order',
  imports: [RedTextDirective],
  templateUrl: './order.html',
  styleUrl: './order.css',
})
export class Order {

  @Input() order: { name: string, price: number } = { name: "", price : 0};

}
```

Order component template

```
<div class="bg-green-200 rounded-xl border border-gray-400">
    <div class="bg-green-200 rounded-xl">
        <p class="rounded-t-xl text-center py-3" appRedText>Order component</p>
        <div class="p-5">
            <p>{{ order.name }}</p>
            <p>{{ order.price }}</p>
        </div>
    </div>
</div>
```

Importing services from the different components will create separate multiple components local for each importing component. Those are different instances internally.

```
@Component({
  providers: [OrderService]
})
export class OrderList implements OnInit {

  constructor(private orderService: OrderService) {}
```

Let's see how to share the same instance among child components

```
app module -> highest level -> same instance
will be shared to all components and also
services
app component -> next highest level ->
same instance will be shared to all child
 components but services have different
instance
child components -> each one will be having
different instances
```

Remove providers statement from @component decorator of the child components, will share the same instance from the parent component.

Importing service from user list component (parent)
Importing service from user component (child)
User list render the set of users (child components)

If we are modifying one order price from the order component we have to import service from the order component, but it will create another copy of instance and modify the change. It will not be reflected to the source instance

So we need to remove the provider statement from the order component and it will inherit its parent component service instance (inherit from the order list component). This will reflect changes back to the parent service instance. Modifications visible

Example for above discussion

```
export class OrderService {
    orders: { name: string; price: number }[] = [
        {
            name: 'Order Name 1',
            price: 23,
        },
        {
            name: 'Order Name 2',
            price: 43,
        },
    ];

    addOrders(name: string, price: number) {
        this.orders.push({ name, price });
    }

    increasePrice(name: string) {
        console.log(this.orders);

        this.orders = this.orders.map((item, index): {name: string, price: number} => {
            if(item.name === name){
                return { name: item.name, price: item.price + 1};
            }else{
                return item;
            }
        });
    }
}
```

```
import { Component, DoCheck, OnInit } from '@angular/core';
import { RedTextDirective } from '../../../directives/red-text.directive';
import { Order } from "../order/order";
import { OrderService } from '../../../services/order.service';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-order-list',
  imports: [FormsModule, RedTextDirective, Order],
  templateUrl: './order-list.html',
  styleUrl: './order-list.css',
  providers: [OrderService]
})
export class OrderList implements OnInit, DoCheck {

  newOrder: string = '';
  orders: { name: string, price: number}[] = [];
```

```
  constructor(private orderService: OrderService) {}

  ngOnInit() {
    this.orders = this.orderService.orders;
  }

  ngDoCheck(): void {
    this.orders = this.orderService.orders;
    console.log(this.orders);

  }

  addOrder() {
    this.orderService.addOrders(this.newOrder.split(" ")[0], parseInt(this.newOrder.split("
")[1]));
  }
}
```

```
import { Component, Input } from '@angular/core';
import { RedTextDirective } from '../../../directives/red-text.directive';
import { OrderService } from '../../../services/order.service';

@Component({
  selector: 'app-order',
  imports: [RedTextDirective],
  templateUrl: './order.html',
  styleUrl: './order.css',
  // providers: [OrderService]
})
export class Order {

  @Input() order: { name: string, price: number } = { name: "", price : 0};

  constructor(private orderService: OrderService) {}

  increasePrice(name: string) {
    this.orderService.increasePrice(name);
    console.log(this.orderService.orders);
  }
}

<div class="bg-green-200 rounded-xl border border-gray-400">
    <div class="bg-green-200 rounded-xl">
        <p class="rounded-t-xl text-center py-3" appRedText>Order component</p>
        <div class="p-5">
            <p>{{ order.name }}</p>
            <p>{{ order.price }}</p>
            <button class="cursor-pointer" (click)="increasePrice(order.name)" >Increase
price</button>
        </div>
    </div>
</div>
```

Using services are advanced solution instead for @input and @outpu usage

Injecting Services into another Services. Usage of @Injectable decorator

Both services. If you want to inject a service into another service. Both services should be in the highest possible level of the parent.

```
You, a few seconds ago | 1 author (You)
import { LogService } from './log.service';
import { Injectable } from '@angular/core';

You, a few seconds ago | 1 author (You)
@Injectable()
export class UserService {
  constructor(private logService: LogService) {}
  users = [
    { name: 'Leela', status: 'Active' },
    { name: 'Leela2', status: 'Active' },
    { name: 'Leela3', status: 'Active' },
  ];

  addUser(name: string, status: string) {
    this.users.push({ name, status });
  }

  updateStatus(id: number, status: string) {
    this.users[id].status = status;
  }
}
```

```
Unsaved changes (cannot determine recent change or authors)
@Injectable()
export class UserService {
  constructor(private logService: LogService) {}
  users = [
    { name: 'Leela', status: 'Active' },
    { name: 'Leela2', status: 'Active' },
    { name: 'Leela3', status: 'Active' },
  ];

  addUser(name: string, status: string) {
    this.users.push({ name, status });
    this.logService.logStatus(status);
  }

  updateStatus(id: number, status: string) {
    this.users[id].status = status;

    this.logService.logStatus(status);
  }
}
```

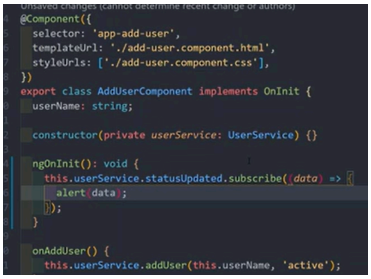
But we don't need @injectablr decorator when we injecting to the components due to @Component decorator

Making the cross component communication using the services by event emitter

In the service
Wherever the event emit this will return observable.

```
  statusUpdated = new EventEmitter<string>();

  addUser(name: string, status: string) {
    this.users.push({ name, status });
    this.logService.logStatus(status);
  }

  updateStatus(id: number, status: string) {
    this.users[id].status = status;
    this.statusUpdated.emit(status);
    this.logService.logStatus(status);
  }
}
```

Subscribe to observable in another component receives data.



```
@Component({
  selector: 'app-add-user',
  templateUrl: './add-user.component.html',
  styleUrls: ['./add-user.component.css'],
})
export class AddUserComponent implements OnInit {
  userName: string;

  constructor(private userService: UserService) {}

  ngOnInit(): void {
    this.userService.statusUpdated.subscribe((data) => {
      alert(data);
    });
  }

  onAddUser() {
    this.userService.addUser(this.userName, 'active');
```

```
/src
├── app
│   ├── core
│   │   ├── interceptors
│   │   │   └── auth.interceptor.ts
│   │   ├── guards
│   │   │   └── auth.guard.ts
│   │   ├── auth.service.ts
│   │   └── user.service.ts
│   ├── shared
│   │   ├── components
│   │   │   └── navbar/
│   │   │   └── sidebar/
│   │   ├── directives
│   │   │   └── debounce.directive.ts
│   │   ├── pipes
│   │   │   └── currency-format.pipe.ts
│   │   └── shared.module.ts
│   ├── features
│   │   ├── admin
│   │   │   ├── components
│   │   │   │   └── admin-dashboard.component.ts
│   │   │   ├── services
│   │   │   │   └── admin.service.ts
│   │   │   ├── admin.module.ts
│   │   │   └── admin-routing.module.ts
│   │   ├── user
│   │   │   ├── components
│   │   │   │   └── user-profile.component.ts
│   │   │   │   └── user-settings.component.ts
│   │   │   ├── services
│   │   │   │   └── user.service.ts
│   │   │   ├── user.module.ts
│   │   │   └── user-routing.module.ts
│   │   ├── products
│   │   │   ├── components
│   │   │   │   └── product-list.component.ts
│   │   │   │   └── product-details.component.ts
│   │   │   ├── services
│   │   │   │   └── product.service.ts
│   │   │   ├── products.module.ts
│   │   │   └── products-routing.module.ts
│   │   └── state
│   │       ├── reducers
│   │       │   └── auth.reducer.ts
│   │       │   └── user.reducer.ts
│   │       └── actions
│   │           └── auth.actions.ts
│   │           └── user.actions.ts
│   ├── app.component.ts
│   ├── app.module.ts
│   └── app-routing.module.ts
├── assets
├── environments
├── styles
├── main.ts
└── index.html
```

The best industrial folder structure for an Angular web application is a scalable, feature-based organization that leverages modules to promote modularity, maintainability, and lazy loading. This structure uses a "core/shared/feature" approach recommended by the official Angular Style Guide.
Recommended Industrial Folder Structure

The main application logic resides in the src/app folder, which is typically broken down into the following sub-folders:

- core/: This module contains singleton services, guards, interceptors, and components used only once in the application's lifetime (e.g., header, footer, navigation). These are essential, core functionalities that should be imported once into the root AppModule.
- shared/: This module contains components, directives, and pipes used across multiple feature modules. It centralizes common UI elements (e.g., buttons, input fields, modals) to avoid duplication and ensure consistency. It should contain only declarables and should not provide services.
- features/ (or pages/, modules/, views/): This is where the bulk of your application's logic lives, organized by specific business features (e.g., user-profile, products, orders). Each feature should ideally be its own module and can be lazy-loaded to improve performance.
    - Within each feature folder, you group all related components, services, and routing for that specific feature.
- assets/: Stores static assets such as images, fonts, and configuration files.
- environments/: Contains environment-specific configuration files (e.g., development, staging, production).

Angular routing

app.routes.ts

```
import { Routes } from '@angular/router';
import { HomePage } from './features/landing/pages/home.page/home.page';
import { OrderListPage } from './features/orders/pages/order-list.page/order-list.page';
import { OrderCategoriesPage } from
'./features/orders/pages/order-categories.page/order-categories.page';

export const routes: Routes = [
    {
        path: '',
        component: HomePage
    },
    {
        path: 'orders',
        component: OrderListPage
    },
    {
        path: 'categories',
        component: OrderCategoriesPage
    }
];
```

app.config.ts

```
import { ApplicationConfig, provideBrowserGlobalErrorListeners } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideRouter(routes)
  ]
};
```

app.ts

```
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { Navbar } from "./shared/components/navbar/navbar";
import { Footer } from "./shared/components/footer/footer";
import { OrderList } from './features/orders/components/order-list/order-list';
import { OrderListPage } from './features/orders/pages/order-list.page/order-list.page';
import { HomePage } from "./features/landing/pages/home.page/home.page";
import { OrderCategoriesPage } from
"./features/orders/pages/order-categories.page/order-categories.page";

@Component({
  selector: 'app-root',
  imports: [RouterOutlet, Navbar, Footer, OrderListPage, HomePage, OrderCategoriesPage,
OrderCategoriesPage],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {}
```

App.html

```
<nav>
  <app-navbar></app-navbar>
</nav>
<main class="main px-5">
  <router-outlet />
</main>
<footer>
  <app-footer></app-footer>
</footer>
```

Navbar.html

```
<main class="main">
    <div class="flex flex-col gap-3 p-5">
        <div
            class="flex justify-between items-center w-full px-5 bg-linear-65 from-blue-950 to-violet-800 py-5 ro"
        >
            <h1 class="text-center text-white text-2xl font-bold">Hello! Welcome {{ name }}</h1>
            <div class="flex flex-row gap-5 text-white font-semibold text-2xl">
                <p class="cursor-pointer hover:text-blue-200"><a routerLink="/">Home</a></p>
                <p class="cursor-pointer hover:text-blue-200"><a routerLink="/orders">Orders</a></p>
                <p class="cursor-pointer hover:text-blue-200"><a routerLink="/categories">Categories</a></p>
            </div>
        </div>
    </div>
</main>
```

navbar.ts

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

@Component({
  selector: 'app-navbar',
  imports: [RouterLink],
  templateUrl: './navbar.html',
  styleUrl: './navbar.css',
})
export class Navbar {
  name = "Web Dev+"
}
```

Dynamic path names

```
<li class="nav-item">
  <a [routerLink]="['/categories']">Categories</a>
</li>
```

Navigate between pages using router programmatically

```
import { Component } from '@angular/core';
import { Route, Router } from '@angular/router';

@Component({
  selector: 'app-order-categories-page',
  imports: [],
  templateUrl: './order-categories.page.html',
  styleUrl: './order-categories.page.css',
})
export class OrderCategoriesPage {

  constructor(private router: Router) {}

  gotoOrders() {
    this.router.navigateByUrl("/orders");
    // NOTE: combine and merge - make single dynamic url
    this.router.navigate(["/oredrs", "order-id"])
  }

}
```

gotoOrders can be placed into onclick event.

Passing and Fetching Parameters to Routes using ActivatedRoute snapshot

Fetch Route Parameters Reactively using Params Subscribe with ActivatedRoute

Fetch param data without re loading the component (data changing without component re loading)

```
<a [routerLink]="['/users', 1, 'leela']">Get Details Of Leela</a>
```

```
export class UserComponent implements OnInit {
  user: { id: string; name: string };
  constructor(private route: ActivatedRoute) {}

  ngOnInit(): void {
    this.user = {
      id: this.route.snapshot.params['id'],
      name: this.route.snapshot.params['name'],
    };
    this.route.params.subscribe((data: Params) => {
      this.user = {
        id: data['id'],
        name: data['name'],
      };
    });
  }
}
```

Listening to data changes

Passing Query Parameters and Fragments to the Url Route with the Template and Program

Query params

```
<div>
  <a
    [routerLink]="['/users', 1, 'leela']"
    [queryParams]="{ page: 1, search: 'leela' }"
    >Get Details Of Leela</a
  >
</div>
```

fragments

```
<div>
  <a
    [routerLink]="['/users', 1, 'leela']"
    [queryParams]="{ page: 1, search: 'leela' }"
    [fragment]="'load'"
    >Get Details Of Leela</a
  >
</div>
```

```
constructor(private route: ActivatedRoute, private router: Router) {}
```

```
  getRamaDetails() {
    this.router.navigate(['/users', 2, 'Rama'], {
      queryParams: { page: 1, search: 'Leela' },
      fragment: 'loading',
    });
  }
}
```

Retrieving Query Parameters and Fragments from the URL through Typescript

```
this.route.snapshot.queryParams
this.route.snapshot.fragment);
```

```
this.route.queryParams.subscribe((data) => {
  console.log(data);
});

this.route.fragment.subscribe((data) => {
  console.log(data);
});
```

Setting up the child or Nested Routes using the children key in routing module

```
const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'users',
    component: UsersComponent,
    children: [{ path: ':id/:name', component: UserComponent }],
  },

  { path: 'categories', component: CategoriesComponent },
];
```

```
<ul class="nav flex-column">
  <li class="nav-item">
    <a [routerLink]="['/users', 1, 'Rama']" class="nav-link"
      >Get Rama Details</a
    >
  </li>
  <li class="nav-item">
    <a [routerLink]="['/users', 1, 'Krishna']" class="nav-link"
      >Get Krishna Details</a
    >
  </li>
  <li class="nav-item">
    <a [routerLink]="['/users', 1, 'Leela']" class="nav-link"
      >Get Leela Details</a
    >
  </li>
</ul>

<div>
  <router-outlet></router-outlet>
</div>
```

Preserve or merge the query parameters by forwarding with queryparamsHandling

```
onUserEdit() {
  this.router.navigate(['/
    queryParamsHandling: ''
  });
}
```

```
onUserEdit() {
  this.router.navigate(['/users', this.user.id, this.user.name, 'edit'],
    queryParamsHandling: 'preserve',
  });
}
```

Implement Custom 404 Page adding wildcard Route, redirectTo option - Wildcard

```
{ path: 'not-found', component: PageNotFoundComponent },
{ path: '**', redirectTo: 'not-found' },
```

Separate all the Routing configuration code into another file app-routing.module

Introduction to Routing Guards. Implementation of canActivate Route Guard

canActivate
canActivateChild
canDeactivate

canActive

auth.service.ts

```
export class AuthService {

    isLoggedIn : boolean = true;

    signIn (): void {
        this.isLoggedIn = true;
    }

    signOut (): void {
        this.isLoggedIn = false;
    }

    isAthuenticated (): boolean {
        return this.isLoggedIn;
    }
}
```

auth.gaurd.ts

```
import { ActivatedRouteSnapshot, CanActivate, GuardResult, MaybeAsync, Router, RouterStateSnapshot }
from "@angular/router";
import { AuthService } from "../services/auth.service";
import { Injectable } from "@angular/core";

@Injectable()
export class AuthGaurd implements CanActivate{

    constructor (private authService: AuthService, private router: Router) {}

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): MaybeAsync<GuardResult> {
        if(this.authService.isAthuenticated()){
            return true;
        }else{
            return this.router.navigate([""]);
        }
    }
}
```

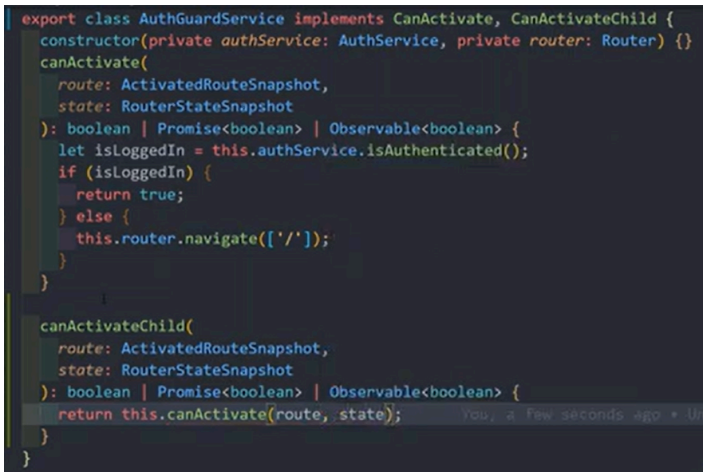
app.config.ts

```
import { ApplicationConfig, provideBrowserGlobalErrorListeners } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { AuthService } from './core/services/auth.service';
import { AuthGaurd } from './core/gaurds/auth.guard';

export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideRouter(routes),
    AuthService,
    AuthGaurd
  ]
};
```

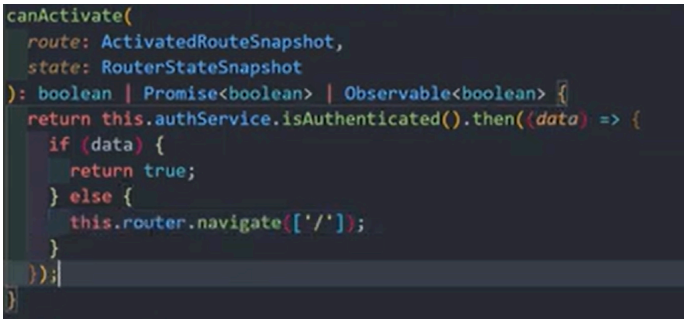
app.routes.ts

```
import { Routes } from '@angular/router';
import { HomePage } from './features/landing/pages/home.page/home.page';
import { OrderListPage } from './features/orders/pages/order-list.page/order-list.page';
import { OrderCategoriesPage } from
'./features/orders/pages/order-categories.page/order-categories.page';
import { AuthGaurd } from './core/gaurds/auth.guard';

export const routes: Routes = [
    {
        path: '',
        component: HomePage
    },
    {
        path: 'orders',
        component: OrderListPage,
        canActivate: [AuthGaurd]
    },
    {
        path: 'categories',
        component: OrderCategoriesPage
    }
];
```

Implement canActivateChild Route Guard for the Nested Child Routes



Async/promise operations



Controlling Navigation with CanDeactivate Route Guard

CanActivate

The CanActivate guard is used to determine if a user can access a specific route in the first place.

- Purpose: Primarily used for authentication and authorization checks (e.g., ensuring a user is logged in or has the necessary role/permissions to view a page).
- When it runs: Before the target component is even created or activated.
- Outcome:
    - If it returns true, navigation continues, and the route/component is activated.
    - If it returns false, navigation is canceled, and the user stays on the current page (often redirected to a login or access-denied page).
- Example use case: Protecting an /admin route so only administrators can access it.

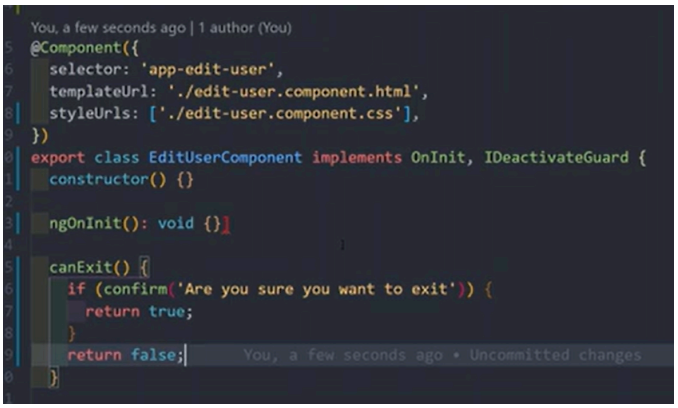
CanDeactivate

The CanDeactivate guard is used to determine if a user can navigate away from the current route.

- Purpose: Primarily used for preventing accidental data loss. It can prompt the user for confirmation (e.g., a "save changes?" dialog) before allowing them to leave a page with unsaved form data.
- When it runs: When a user attempts to navigate to a new route, but before the current component is destroyed or the new route is activated.
- Outcome:
    - If it returns true, the navigation proceeds to the new route.
    - If it returns false, navigation is canceled, and the user remains on the current component.
- Example use case: Displaying a confirmation dialog when a user tries to leave a form with unsaved changes.

Summary of Differences

Feature	CanActivate	CanDeactivate
Timing	Runs before the route is activated.	Runs before leaving the current route.
Primary Use	Access control (authentication/authorization).	Preventing data loss (unsaved changes).
Access to Component	Does not have access to the target component instance (it's not created yet).	Has access to the current component instance, which allows checking its state (e.g., form.dirty).
Decision	Decides if a user can enter a page.	Decides if a user can exit a page.



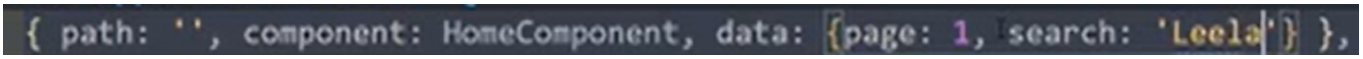
Implementing CanDeactivate Route Guard in the real-time scenario for the component

```
export class EditUserComponent implements OnInit, IDeactivateGuard {
  user: { id: string; name: string };
  editUserDetails: { id: string; name: string };
  constructor(private route: ActivatedRoute) {}

  ngOnInit(): void {
    this.route.params.subscribe((data: Params) => {
      this.user = {
        id: data['id'],
        name: data['name'],
      };

      this.editUserDetails = { ...this.user };
    });
  }
```

```
<div>
  <div class="form-group">
    <label>Id:</label>
    <input type="text" class="form-control" [(ngModel)]="editUserDetails.id" />
  </div>
  <div>
    <div class="form-group">
      <label>Name:</label>
      <input
        type="text"
        class="form-control"
        [(ngModel)]="editUserDetails.name"
      />
    </div>
  </div>
</div>
```

```
canExit() {
  console.log(this.user);
  console.log(this.editUserDetails);

  if (
    this.editUserDetails.id !== this.user.id ||
    this.editUserDetails.name !== this.user.name
  ) {
    if (confirm('Are you sure you want to exit')) {
      return true;
    } else {
      return false;
    }
  }

  return false;
}
```

Passing Static Data to the Route and also Access the static data



```
{ path: '', component: HomeComponent, data: {page: 1, search: 'Leela'} },
```

```
export class HomeComponent implements OnInit {
  constructor(private route: ActivatedRoute) {}

  ngOnInit(): void {
    this.route.data.subscribe((data: Data) => {
      console.log(data);
    });
  }
```

Get Dynamic Data before entering into the component using the Resolve Guard

```
interface User {
    id: string;
    name: string;
}

@Injectable()
export class UserResolveService implements Resolve<User> {

    constructor(private userService: UserService) {

    }

    resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): User | Observabl
        let id = route.params['id'];

        let details = this.userService.getUser(id);
        return details;

    }
}
```

```
{
  path: ':id/:name/edit',
  component: EditUserComponent,
  canDeactivate: [DeactivateGuardService],
  resolve: { user: UserResolveService }
},
```

```
ngOnInit(): void {

  this.route.data.subscribe(data => {
    this.user = {
      id: data['id'],
      name: data['name'],
    };
    this.editUserDetails = { ...this.user };
  })
    You, a few seconds ago • Uncommitted char
}
```

How to use Hash Urls as Fragments in the url for the internal pages

hashRouter

Understand the core of the Observables in rxjs. Need of subscribe & unsubscribe

In Angular, an Observable is a powerful technique from the RxJS library used for asynchronous programming and handling multiple values emitted over time, such as HTTP responses, user input events, and timers. They implement the publisher/subscriber pattern.

Key Concepts

- Publisher/Producer: The Observable is the producer of a stream of data or events.
- Subscriber/Observer: The consumer of the data provided by the observable. An observer is an object with next(), error(), and complete() callback functions to handle notifications from the observable.
- Subscription: To receive values, an observer must subscribe() to an observable. The subscription represents the ongoing execution and can be used to cancel the process using unsubscribe().
- Operators: RxJS provides a rich set of operators (like map(), filter(), debounceTime(), etc.) that allow you to transform, filter, and combine data streams in powerful ways using the pipe() method.
- Laziness: Observables are "lazy"; the data-producing function does not execute until an observer subscribes to it. This allows you to define the data flow recipe without running it prematurely.

Common Use Cases in Angular

Observables are used extensively throughout the Angular framework:

- HTTP Requests: Angular's HttpClient methods (e.g., get(), post()) return observables. Subscribing to them initiates the request and handles the response or error when it arrives.
- Event Handling: They can be used to manage user interactions like keystrokes or button clicks, often using the fromEvent creation function.
- Forms: Reactive forms expose observables like valueChanges and statusChanges to listen for and react to user input over time.
- Routing: The Angular Router uses observables to manage navigation events and route parameters.

Subscribing and Unsubscribing

You can subscribe in the component's TypeScript file or use the async pipe in the template

```typescript
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';

// ...

private dataSubscription: Subscription;

ngOnInit(): void {
  this.dataSubscription = this.dataService.getData().subscribe({
    next: (value) => console.log('Received: ' + value),
    error: (err) => console.error('Error: ' + err),
    complete: () => console.log('Stream complete')
  });
}

ngOnDestroy(): void {
  // Important to prevent memory leaks
  if (this.dataSubscription) {
    this.dataSubscription.unsubscribe();
  }
}
```

Using interval observable from RXJS

```
export class HomeComponent implements OnInit, OnDestroy {
  intervalSubscription: Subscription;
  routeSubscription: Subscription;
  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.routeSubscription = this.route.data.subscribe((data: Data) => {
      console.log(data);
    });

    this.intervalSubscription = interval(1000).subscribe(count => {
      console.log(count);
    })
  }

  ngOnDestroy() {
    this.intervalSubscription.unsubscribe();
    this.routeSubscription.unsubscribe();
  }
}
```

Angular observables unsubscribed automatically when ever leaving from the component but RxJS observables must unsubscribe manually during component destruction.

Create our own custom Observable in Rxjs. How to use observer.next option

```
let customObservable = Observable.create(observer => {
  let count = 0;
  setInterval(() => {
    observer.next(count);
    count++;
  }, 1000);
});

customObservable.subscribe(data => {
  console.log(data);
})
```

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Route, Router } from '@angular/router';
import { Observable, Subscription } from 'rxjs';

@Component({
  selector: 'app-order-categories-page',
  imports: [],
  templateUrl: './order-categories.page.html',
  styleUrl: './order-categories.page.css',
})
export class OrderCategoriesPage implements OnInit, OnDestroy{

  mySubscription: Subscription | undefined;

  constructor(private router: Router) {}

  ngOnInit(): void {
    let customObservable = new Observable<number>(observer => {
      let count = 0;
      setInterval(() => {
        count++;
        observer.next(count)
      }, 1000);
    })

    this.mySubscription = customObservable.subscribe(data => {
      console.log(data);
    })
  }

  ngOnDestroy(): void {
    this.mySubscription?.unsubscribe();
  }

  gotoOrders() {
    this.router.navigateByUrl("/orders");
  }

}
```