**Angular**

**Angular is a TypeScript-based, open-source, web application development framework developed by Google. It is used to build single-page applications (SPAs) with a component-based architecture**.

**AngularJS (1.x) vs Angular (2+):**

* AngularJS uses javascript, it is based on MVC model and does direct DOM manipulations.
* Angular uses typescript (a superset of javascript), it is based on component model and uses a virtual DOM.

**Virtual DOM (VDOM):**

The Virtual DOM is a lightweight, in-memory representation of the real DOM. It’s a JavaScript object that mirrors the structure of the actual DOM elements.

**How VDOM works:**

* When your app’s state changes (e.g., user input, API response), a new virtual DOM is created.
* The framework compares the new virtual DOM with the previous version (this process is called diffing).
* It then calculates the minimum number of changes needed.
* Finally, it updates the real DOM in a batch and efficient way, avoiding unnecessary re-renders.

**Single Paged Applications (SAPs):**

An SAP loads a single HTML page and dynamically updates the content without reloading the entire page as the user interacts with it. On the initial load, the browser downloads the core HTML, CSS, and JavaScript and dynamically fetches and updates (only the necessary content) , instead of requesting a full new page from the server.

**Advantages of implementing SAPs:**

* Separating concerns: UI rendering on the client, data on the server.
* Leveraging client-side rendering for smoother interactions.
* Reducing redundant page loads, hence making apps feel faster and more responsive.

**Angular Component’s Lifecycle**

**Prerequisites for understanding Component lifecycle:**

**What Is a ‘Change Detection Cycle’ in Angular?**

A change detection cycle is a process where Angular checks your application for changes like whether data has changed, and whether the DOM needs to be updated.

Angular does this automatically and frequently every time something "could" have changed. Angular triggers change detection in many situations, such as:

* User input = Typing in an input field (keyup, click, etc.)
* Timers = setTimeout(), setInterval()
* HTTP calls = Receiving an API response
* Observable emits RxJS data changes
* Component lifecycle changes = Creating or destroying components
* Zone changes = Basically any async event Angular is aware of

**Important Notes:**

* Angular does not compare objects deeply it checks references.
* If you mutate an object or array (e.g., user.name = 'John'), Angular doesn't detect the change automatically unless a new reference is assigned. Hence ngOnChanges() is not called.
* That’s where ngDoCheck() helps it lets you manually check for deep changes.

**What is <ng-content>?**

<ng-content> is an Angular directive that acts as a placeholder for content passed from the parent component into a child component. It is used build reusable UI containers that let parent components control content.

Example:

//child component

@Component({

selector: 'app-card',

template:

`<div class="card">

<ng-content></ng-content>

</div>`

})

export class CardComponent { }

//parent component

<app-card>

<h2>Angular Card</h2>

<p>This is a reusable card with custom content.</p>

</app-card>

**What is ChangeDetectorRef?**

In Angular, ChangeDetectorRef is a service that gives you low-level control over Angular's change detection mechanism. It can be used to manually trigger change detection or to detach and reattach components from the change detection tree.

**Example:**

import {Component, ViewChild, ElementRef, AfterViewInit, ChangeDetectorRef} from '@angular/core';

@Component({

selector: 'app-demo',

templateUrl: './demo.component.html'

})

export class DemoComponent implements AfterViewInit {

@ViewChild('para') paraRef!: ElementRef;

length: number = 0;

constructor(private cdr: ChangeDetectorRef) {}

ngAfterViewInit(): void {

this.length = this.paraRef.nativeElement.textContent.length;

// Force Angular to detect changes again

this.cdr.detectChanges();

}

}

**Why Would You Need detectChanges()?**

Sometimes, you want to update a value after Angular has finished its initial rendering (like inside ngAfterViewInit()), but Angular has already done change detection.

If you update a bound property at that point, Angular throws: ExpressionChangedAfterItHasBeenCheckedError. This is Angular’s strict way of enforcing predictable, one-way data flow during its rendering.

**Why can’t Angular just re-run change detection automatically?**

Change detection usually happens due to events like user input, async data, etc. Imagine if Angular kept running change detection every time you changed something it could lead to infinite loops or degraded performance.

**Component lifecycle:**

**1. constructor:** Called when the component instance is created. Used for basic initialization and to inject dependencies (but avoid logic dependent on bindings or Angular features).

**2. ngOnChanges():** It is called whenever any data-bound @Input() property changes. It receives a SimpleChanges object that contains the previous and current values of the changed input properties.

This happens frequently, so any operation you perform here impacts performance significantly.

If your component has no inputs or you use it without providing any inputs, the framework will not call ngOnChanges().

Example scenario: A parent component that passes a value to a child component using @Input().The child component implements ngOnChanges() to react when the input value changes.

// parent.component.ts

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

@Component({

selector: 'app-parent',

template: `

<input [(ngModel)]="parentValue" placeholder="Type something..." />

<app-child [data]="parentValue"></app-child>`

})

export class ParentComponent {

parentValue = 'Initial Value';

}

// child.component.ts

import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';

@Component({

selector: 'app-child',

template: `<p>Child received: {{ data }}</p>`

})

export class ChildComponent implements OnChanges {

@Input() data!: string;

ngOnChanges(changes: SimpleChanges) {

if (changes['data']) {

const prev = changes['data'].previousValue;

const curr = changes['data'].currentValue;

console.log(`data changed from "${prev}" to "${curr}"`);

}

}

}

**3. ngOnInit():** Called once after the component's inputs (@Input()) are initialized but before the component is rendered. Ideal for initialization logic, such as fetching data from a service, setting up default values, starting timers or listeners.

**4. ngDoCheck():**Called immediately after ngOnChanges() on every change detection run, and immediately after ngOnInit() on the first run.

It is a lifecycle hook called during every change detection cycle, even if no data has changed. We can use it when there is a need to implement custom change detection logic.

Example: If you modify an object’s property (instead of replacing the object), Angular won’t know unless you tell it. ngDoCheck() lets you do that.

import { Component, Input, DoCheck } from '@angular/core';

@Component({

selector: 'app-child',

template: `<p>Message: {{ user.message }}</p>`

})

export class ChildComponent implements DoCheck {

@Input() user!: { message: string };

private oldMessage = '';

ngDoCheck(): void {

if (this.user.message !== this.oldMessage) {

console.log('Message changed from', this.oldMessage, 'to', this.user.message);

this.oldMessage = this.user.message;

}

}

}

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

@Component({

selector: 'app-parent',

template: `

<button (click)="updateMessage()">Change Message</button>

<app-child [user]="userObj"></app-child> `

})

export class ParentComponent {

userObj = { message: 'Hello' };

updateMessage() {

this.userObj.message = 'Hello again!';

}

}

**5. ngAfterContentInit():** It is a lifecycle hook in Angular that is called once after Angular projects external content into the component's view using <ng-content>. It lets you inspect or manipulate the projected content after it's inserted.

**6. ngAfterContentChecked():** It is a lifecycle hook that is called after ngAfterContentInit() and every subsequent time Angular performs change detection on the projected content.

**7. ngAfterViewInit():** Called once, after the component's view has been fully initialized. Meaning, all DOM elements in the component template are available and child components used in the template are also initialized.

This lifecycle hook is generally used when we want to interact with child components after they are rendered.

Avoid updating bindings in ngAfterViewInit().It can cause ExpressionChangedAfterItHasBeenCheckedError if you change something Angular just rendered.

Use ChangeDetectorRef.detectChanges(), If you must update a bound value, trigger change detection manually.

**8. ngAfterViewChecked():** It is called after ngAfterViewInit() and after the view and child views have been checked by the Angular change detection system.

**9. ngOnDestroy():** Called just before Angular destroys the component. Ideal for cleanup (e.g., unsubscribing from Observables).

**Angular Bindings**

**Angular supports four main types of bindings.**

**1. String Interpolation (One-way Data Binding)**

Syntax: {{ expression }}

Purpose: Display component properties in the template. Interpolation evaluates the expression inside {{ }} and converts it to a string.

Example:

// app.component.ts

export class AppComponent {

title = 'Angular Data Binding';

}

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

<h1>{{ title }}</h1>

**2. Property Binding (One-way Data Binding)**

Syntax: [property]="expression"

Purpose: Set DOM element’s JavaScript property from the component.

Example:

// app.component.ts

export class AppComponent {

imageUrl = 'https://example.com/image.jpg';

}

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

<button [disabled]="!isAgreed">Submit</button>

**2.1. Attribute Binding**

Syntax: [attr.attributeName] ="expression"

Purpose: Sets the HTML attribute value.

Example:

<td [attr.colspan]="2"></td>

**2.2. Class Binding**

Syntax: [attr.attributeName] ="expression"

Example:

<div [class.active]="isActive"></div>

**2.3. Style Binding**

Syntax: [attr.attributeName] ="expression"

Example:

<div [style.background-color]="bgColor"></div>

**3. Event Binding**

Syntax: (event)="handlerFunction($event)"

Purpose: Capture DOM events and call component methods.

Example:

// app.component.ts

export class AppComponent {

showMessage() {

alert('Button clicked!');

}

}

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

<button (click)="showMessage()">Click Me</button>

**4. Two-way Data Binding**

Syntax: [(ngModel)]="property" (Requires FormsModule to be imported)

Purpose: Sync data between component and input field.

Example:

// app.component.ts

export class AppComponent {

username = '';

}

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

<input [(ngModel)]="username">

<p>Hello, {{ username }}</p>

Note: While [(ngModel)] is the most common and convenient way to achieve two-way binding in Angular, it's not the only way.

**Angular Decorators**

**A Decorator is always prefixed with @. We must place the Decorator immediately before the class definition. We can also build our own decorators.**

**1. @Component()**

@Component() is a decorator used to define a component in Angular. Components control a patch of the screen called a view. @Component() associates metadata with the component such as its HTML template, CSS styles, and selector.

Syntax:

@Component({

selector: 'app-my-component',

template / templateUrl: './my-component.component.html',

styles / styleUrls: ['./my-component.component.css']

})

export class HelloComponent {

name = 'Angular';

}

Explanation:

* selector: Defines the custom HTML tag that represents this component in templates.
* template / templateUrl: Defines the HTML view of the component.
* styles / styleUrls: Define CSS for the component.
* standalone: (Introduced in Angular 14) Declares that the component is self-contained and does not need to be part of an NgModule.

**2. @NgModule()**

An NgModule is a class marked by the @NgModule decorator. This decorator accepts metadata that tells Angular how to compile component templates and configure dependency injection.

Syntax:

@NgModule({

declarations: [ComponentA, ComponentB],

imports: [BrowserModule, FormsModule],

providers: [SomeService],

bootstrap: [AppComponent]

})

export class AppModule { }

declarations: The components, directives, and pipes that belong to the module.

imports: Other modules whose exported classes are needed in this module.

providers: Services available to the injector of this module.

bootstrap: The root component that Angular creates and inserts into the index.html.

**3. @Input() and @Output()**

The **@Input()** decorator allows a parent component to bind to a property in a child component and pass data to it.

Example:

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

@Component({

selector: 'app-parent',

template: `<app-child [message]="parentMessage"></app-child>`

})

export class ParentComponent {

parentMessage = 'Hello from Parent!';

}

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

@Component({

selector: 'app-child',

template: `<p>Received message: {{ message }}</p>`

})

export class ChildComponent {

@Input() message: string = '';

}

The **@Output()** decorator is used with EventEmitter to allow the child component to emit events/data to the parent.

Example:

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

@Component({

selector: 'app-parent',

template: `

<app-child (messageEvent)="receiveMessage($event)"></app-child>

<p>{{ receivedMessage }}</p>

`

})

export class ParentComponent {

receivedMessage: string = '';

receiveMessage(message: string) {

this.receivedMessage = message;

}

}

import { Component, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-child',

template: `<button (click)="sendMessage()">Send Message</button>`

})

export class ChildComponent {

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

sendMessage() {

this.messageEvent.emit('Hello from Child!');

}

}

**4. @Injectable**

@Injectable() is a decorator in Angular used to mark a class as available for dependency injection (DI). It tells Angular that the class can be instantiated and injected into other components or services.

Injectable allows control over the class’s scope using the providedIn property:

* 'root': The service is a singleton and available application-wide.
* 'any': A new instance is created in each lazy-loaded module.
* Specific Module: The service is only available within that module's injector scope any other module that explicitly imports the specified module.

Syntax:

@Injectable({

providedIn: 'root' | 'any' | ModuleClass

})

export class SomeClass {}

Also, instead of using providedIn in the @Injectable() decorator, you can opt to manually register the service in the providers array of a specific module.

Example:

@Injectable()

export class LoggingService { }

@NgModule({

providers: [LoggingService] // manual control here

})

export class FeatureModule { }

Why would you do this?

* To limit the scope of a service to a specific module or feature.
* To provide different instances of a service in different modules.
* For lazy-loaded modules, if you want the service to be created only when the module is loaded.

**5. @ViewChild**

@ViewChild() is a decorator in Angular used to get a reference to a child component, directive, or DOM element from a parent component class.

Using ViewChild we can access properties and methods of a child component or directive.

Syntax:

@ViewChild(selector: string | Type<any>, options?: { static?: boolean })

selector: A template reference variable, component, or directive type.

static: Determines when the query is resolved. true for access in ngOnInit, false for access in ngAfterViewInit.

Example: Accessing a Component

@Component({

selector: 'app-child',

template: '<p>Child Component</p>'

})

export class ChildComponent {

sayHello() {

console.log('Hello from Child');

}

}

@Component({

selector: 'app-parent',

template: '<app-child></app-child>'

})

export class ParentComponent {

@ViewChild(ChildComponent) child!: ChildComponent;

ngAfterViewInit() {

this.child.sayHello();

}

}

Example: Accessing a DOM Element

<input #myInput type="text">

@ViewChild('myInput') inputRef!: ElementRef;

ngAfterViewInit() {

this.inputRef.nativeElement.focus();

}

**Angular Directives**

**A directive is a class annotated with @Directive() decorator. It tells Angular to attach certain behavior to an element or even transform it.**

Angular provides two main types of directives:

1. Attribute Directives
2. Structural Directives

**1. Attribute Directives**

Attribute directive is used to change the appearance or behavior of an existing element, such as adding styles or handling events. ngClass, ngStyle and ngModel are some of the built in attribute directives.

Example:

<div [ngClass]="{'text-success': isSuccess,'text-danger': !isSuccess}">

<div [ngStyle]="{ 'background-color': isSuccess ? 'yellow' : 'green'}”>

**2. Structural Directives**

Structural directive changes the structure of the DOM, adding or removing elements based on conditions. They are denoted by an asterisk (\*) preceding the directive name. ngIf, ngFor and ngSwitch are some of the built in structural directives.

Example:

<element \*ngIf="condition"> <!—Content> </element>

<element \*ngFor="let item of items "> <!—Content> </element>

<element [ngSwitch]="expression">

<element \*ngSwitchCase="value1"> <!—Content> </element>

</element>

**Creating a Custom Directives:**

Custom Directives can also be created to add specific behaviors to elements, enhancing the functionality of an Angular application.

npm command: ng generate directive my-highlight

Example:

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

@Directive({

selector: '[appMyHighlight]'

})

export class MyHighlightDirective {

constructor(private el: ElementRef) { }

@HostListener('mouseenter') onMouseEnter() {

this.highlight('yellow');

}

@HostListener('mouseleave') onMouseLeave() {

this.highlight(null);

}

private highlight(color: string | null) {

this.el.nativeElement.style.backgroundColor = color;

}

}

@Component({

selector: 'app-root',

template:

`<p appMyHighlight>

Hover over this text to see it highlighted.

</p>`

})

export class AppComponent { }

**Angular Signals**

**A signal is a wrapper around a value, introduced in Angular (from v16 onwards), which is capable of notifying interested consumers when that value changes. Signals can contain any value, from simple primitives to complex data structures.**

**Why do we need Signals?**

Before Signals:

* Angular relied on Zone.js + Change Detection to check if something in the app changed, this caused the entire component tree to be checked on events (clicks, HTTP responses, etc).
* Even smallchanges could trigger large, unnecessary re-renders.

With Signals:

* We get fine-grained reactivity only the parts of the UI (or computations) that depend on changed data are updated.
* No need to check unrelated parts of the UI.

**Types of Signals**

1. Writable Signals

These are signals you can directly update. You can use the set() method to change a writable signal’s value, or the update() method to modify it based on a specific function.

Example:

import { signal } from '@angular/core';

export class CounterComponent {

counter = signal(0); // signal holding number

increment() {

this.counter.update(value => value + 1);

}

reset() {

this.counter.set(0);

}

}

In template:

<button (click)="increment()">Increment</button>

<button (click)="reset()">Reset</button>

<p>Counter: {{ counter() }}</p>

Whenever we call increment() or reset(), the displayed value of counter in the template updates immediately

2. Computed Signals

These signals get their value from other signals. You set up a computed signal using the computed() function and telling it how to derive its value. When any of the signals it depends on changes, the computed signal updates accordingly.

Example:

import { computed } from '@angular/core';

export class CounterComponent {

price = signal(10);

quantity = signal(5);

totalCost = computed(() => price() \* quantity());

}

In template:

<p>Double: {{ totalCost() }}</p>

Here, totalCost() recalculates whenever price or quantity changes.

3. Effects

An effect is a process that is triggered whenever there is a change in one or more signal values. This functionality is implemented using the effect() function.

Example:

import { effect } from '@angular/core';

export class CounterComponent {

isAuthenticated = signal(false);

constructor() {

effect(() => {

if (this.isAuthenticated()) {

console.log('User is authenticated. Redirecting to dashboard…');

} else {

console.log('User is not authenticated. Redirecting to login page…');

}

});

}

}

An effect is triggered by changes in isAuthenticated signal, leading to specific actions.

**Key methods on Signal**

* signal.set(newValue) = Set new value
* signal.update(fn) = Update value based on old value
* signal.mutate(fn) = Mutate value in place (for objects, arrays)
* signal() = Get value (read)
* computed(() => … ) = Create a derived signal
* effect(() => … ) = Run code when signals change

**HttpInterceptor**

**HttpInterceptor is an interface provided by the @angular/common/http package. It allows you to intercept and manipulate HTTP requests and responses made using HttpClient.**

**Why Use HttpInterceptor?**

It enables centralized handling of cross-cutting concerns such as:

* Authentication: Attach JWT tokens
* Error Handling: Show alerts on failure
* Retry/Timeout Policies: Retry failed requests, limit timeouts
* Logging: Debug outgoing requests
* Caching: Avoid redundant network calls
* Headers Modification: Set custom or standard headers
* Encryption/Decryption: Encryption and decryption of http payload

**How It Works?**

1. Create an interceptor class that implements HttpInterceptor.

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

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

import { Observable } from 'rxjs';

@Injectable()

export class MyInterceptor implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

// modify or log the request here

return next.handle(req); // pass it to the next interceptor or the backend

}

}

2. Register it in the providers array.

import { HTTP\_INTERCEPTORS } from '@angular/common/http';

@NgModule({

providers: [

{

provide: HTTP\_INTERCEPTORS,

useClass: MyInterceptor,

multi: true

}

]

})

export class AppModule {}

**Router Events Interceptor**

**Angular’s Router provides an event-based API to let you observe what’s happening during navigation. By subscribing to router.events, you can intercept URL changes, perform logic before or after navigation, and handle special cases like redirects, navigation errors, or canceled navigations.**

**Why Use Router Event Interception?**

* Show loader: During navigation start
* Track user routes: For analytics/logging
* Scroll position reset: After navigation
* Block unwanted navigation: Confirm unsaved changes
* Handle 404s: Log or redirect

**Common Router Events:** NavigationStart, NavigationEnd, NavigationCancel, NavigationError, GuardsCheckStart/End, Scroll.

Example:

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

import { Router, NavigationStart, NavigationEnd, NavigationError, NavigationCancel, Event } from '@angular/router';

@Component({

selector: 'app-root',

template: `

<div \*ngIf="loading" class="loader">Loading...</div>

<router-outlet></router-outlet>

`

})

export class AppComponent implements OnInit {

loading = false;

constructor(private router: Router) {}

ngOnInit(): void {

this.router.events.subscribe((event: Event) => {

if (event instanceof NavigationStart) {

this.loading = true;

console.log('Navigation started to:', event.url);

} else if (event instanceof NavigationEnd || event instanceof NavigationCancel || event instanceof NavigationError) {

this.loading = false;

console.log('Navigation ended/cancelled/errored:', event);

}

});

}

}

**Route Guards**

**Route Guards are interfaces provided by Angular that allow you to control navigation to and from routes based on conditions such as user authentication, unsaved changes, user roles, etc.**

**While Router Events Interceptor can only observe navigation, Route Guards can control navigation flow by blocking, allowing, or redirecting navigations.**

**Built-in Route Guards in Angular:**

* CanActivate: Decide if a route can be activated/accessed
* CanActivateChild: Decide if child routes can be activated/accessed
* CanDeactivate: Decide if a route can be left. A common scenario is preventing navigation away from unsaved forms.
* CanLoad: Prevent lazy-loaded module from loading
* CanMatch: Advanced matching logic for route selection

Example:

//auth.guard.ts

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

import { CanActivate, Router, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree } from '@angular/router';

import { Observable } from 'rxjs';

import { AuthService } from './auth.service';

@Injectable({

providedIn: 'root'

})

export class AuthGuard implements CanActivate {

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

canActivate(

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot

): boolean | UrlTree | Observable<boolean | UrlTree> | Promise<boolean | UrlTree> {

if (this.authService.isLoggedIn()) {

return true;

} else {

return this.router.createUrlTree(['/login']);

}

}

}

//app-routing.module.ts

const routes: Routes = [

{ path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] },

{ path: 'login', component: LoginComponent }

];