Angular Part - 3

Standalone components

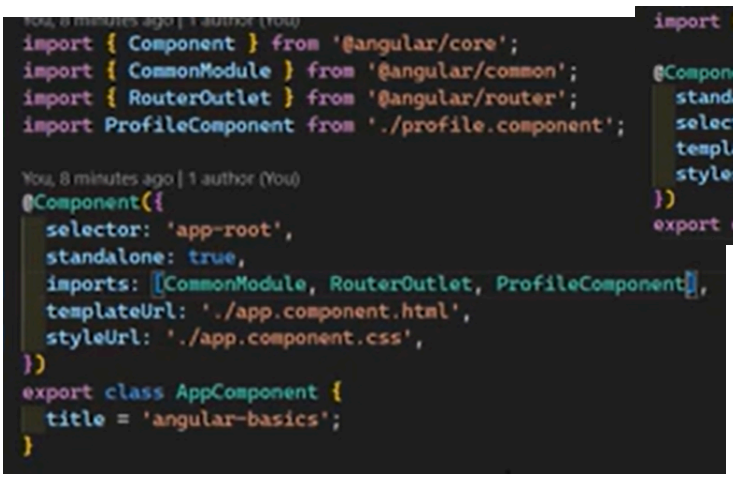
An Angular standalone component is a self-contained component that does not need to be declared in an @NgModule. This feature, introduced in Angular 14 and stable in Angular 15+, simplifies the application structure, reduces boilerplate code, and streamlines the development experience. As of Angular 19, standalone is the default for new projects.

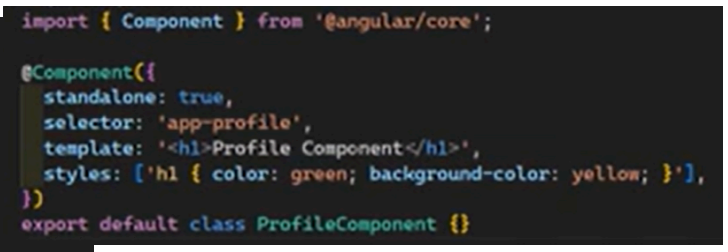
Key Concepts

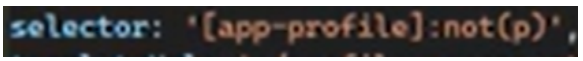
- Self-Contained: A standalone component manages its own dependencies by importing the features it uses (like other components, directives, pipes, or services) directly within its @Component decorator's imports array.
- Module-less: It removes the need for the traditional declarations and imports arrays within a separate @NgModule file, making the component definition clearer and more modular.
- Improved Performance: This architecture enables better "tree-shaking" (removing unused code) during the build process, which can lead to smaller bundle sizes and faster application startup times.
- Easier Lazy Loading: Standalone components simplify lazy loading, as entire modules are no longer required for this purpose. You can simply use the loadComponent function in your routing configuration.

ng generate component my-standalone --standalone

Alternatively, you can manually set the standalone: true property in the component's metadata:







support of pseudo classes

:host, :host-context, ng-deep pseudo classes for stylings. And also encapsulation levels (none, emulated and shadowDom)

Leveraging Input Properties for Seamless Component Data Exchange

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

@Component({
  standalone: true,
  selector: 'app-counter',
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.css'],
})
export default class CounterComponent {
  @Input({ required: true, transform: changeValue }) count:
  number = 0;
}

function changeValue(value: number) {
  return value * 10;
}
```

Example:

```
export class OrderCategoriesPage implements OnInit, OnDestroy{

  numval: number = 0;
  changeDetectorRef!: ChangeDetectorRef;

  constructor(private router: Router, private cdr: ChangeDetectorRef) {
    this.changeDetectorRef = cdr
  }

  ngOnInit(): void {

    setInterval(() => {
      this.numval = this.numval + 2;
      this.changeDetectorRef.markForCheck();    // force re-render
    }, 1000)
  }
}
```

```
<div class="mt-5">
    <div>
        <app-counter [count]="numval" ></app-counter>
    </div>
</div>
```

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

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

  @Input() count: number = 0;

}
```

Inbuilt transforms (boolean attribute and numberattribute)

```
@Input({ transform: booleanAttribute }) showcounter:
boolean = false;
```

```
@Input({ required: true, transform: changeValue, alias:
'dummynamecounter' })
count: string = '';
```

Getters and setters

```
export class GetterInputComponent {
  private _title = '';
  @Input()
  get title() {
    return this._title;
  }

  set title(value: string) {
    this._title = value.trim().toUpperCase();
  }
}
```

```
<h3>Getter input component</h3>
<div>{{ title }}</div>
```

```
export class AppComponent {
  title = 'angular-basics';
  counter: number = 20;
}
```

```
<div>
  <app-getter-input [title]="title"></app-getter-input>
</div>
```

Inheritance

```
@Component({
  standalone: true,
  imports: [CommonModule],
  selector: 'app-counter',
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.css'],
  inputs: ['count', 'message', 'showcounter', 'title'],
})
export default class CounterComponent extends
GetterInputComponent {
  @Input({ required: true, transform: changeValue, alias:
  'dummynamecounter' })
  count: string = '';

  @Input({ transform: trimValue }) message: string = '';

  @Input({ transform: booleanAttribute }) showcounter:
  boolean = false;
}
```

Send Custom Events Using Output Decorator

There is App component and Panel component, panel component as a child component resides in the app component. We are trying to send data from the child to the parent (panel -> app)

```
export class AppComponent {
  title = 'angular-basics';
  counter: number = 20;
  receivedDataFromChild = '';

  dataReceived(data: string) {
    this.receivedDataFromChild = data;
  }
}
```

App component

```
<div>
  <app-panel (dataEvent)="dataReceived($event)"></
  app-panel>      You, 1 second ago • Uncommitted ch
</div>

<div *ngIf="receivedDataFromChild">{{
receivedDataFromChild }}</div>
```

Panel component

```
export class PanelComponent {
  @Output() dataEvent = new EventEmitter<string>();
  sendData() {
    this.dataEvent.emit('data from child');
  }
}
```

Panel component

```
<div>
  <button (click)="sendData()">Send data to parent</
  button>
</div>
```

Ng-content - rendering child content

```
<div>
  <app-card>
    <p>this is the first card component</p>
  </app-card>
</div>

<div>
  <app-card>
    <p>This is the second card component</p>
  </app-card>      You, 8 seconds ago • Uncommi
</div>
```

```
<div class="card">
  <ng-content></ng-content>
</div>
```

```
<div>
  <app-card>
    <p title>this is the first card component</p>
    <p body>This is teh bod</p>
  </app-card>
</div>

<div>
  <app-card>
    <p title>This is the se
    <p body>This is the bod
  </app-card>
</div>
```

```
<div class="card">
  <ng-content select="[title]"></ng-content>
  <ng-content select="[body]"></ng-content>

  <ng
</divDa-ng-container
```

In Angular, the host element is the DOM element that matches a component's or directive's selector. It acts as the container for the component's rendered template and view.

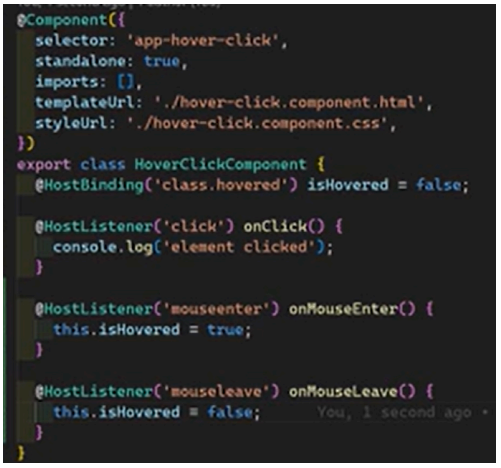
For example, if you define a component with the selector app-user-profile, the corresponding HTML element <app-user-profile> in your application's main HTML file (or another component's template) is its host element.

Interacting with the Host Element

Angular provides several ways to interact with the host element from within the component or directive's class:

- Styling: You can target and style the host element using the special :host CSS pseudo-class selector within the component's stylesheet. This allows you to apply styles to the container itself, while maintaining view encapsulation.

- Binding: You can bind properties, attributes, and events to the host element. The modern approach is to use the host property within the @Component or @Directive decorator




```
host: {
  '[class.active]': 'isActive', // Bind a class
  '(click)': 'onClick($event)',  // Listen to an event
  'role': 'button'               // Set a static attribute
 }
```

Alternate to ngOnDestroy lifecycle method

```
constructor(private destroyRef: DestroyRef) {
  console.log('constructor called');

  destroyRef.onDestroy(() => {
    console.log('destroyref fired');
  });
  //console.log(this.title);
}
```

ngAfterViewInit

```
@ViewChild('leela') leelaElement!: ElementRef;
```

```
ngAfterViewInit() {
  console.log(this.leelaElement.nativeElement);
  console.log('after view init initialized');
}
```

Access child content components

```
<app-life-cycle-hooks [title]="title" [counter]="counter"
*ngIf="isActive">
  <app-profile></app-profile>
</app-life-cycle-hooks>
```

Instead of @ViewChild

```
@ContentChild(ProfileComponent) profileContent!:
ProfileComponent;
```

```
ngAfterContentInit() {
  console.log(this.profileContent);
  console.log('after contentinit fired');
}
```

ngAfterViewChecked

afterRender & afterNextRender Lifecycle Hooks

In Angular, afterRender() is a function that registers a callback to run after every change detection cycle completes and the DOM has been updated in the browser. It is primarily used for manual DOM manipulation and synchronizing with third-party libraries when the standard data binding isn't sufficient.

Key Features

- Execution Timing: The callback runs after the browser has finished painting the UI, ensuring the DOM is stable and ready for interaction.
- Browser-Only: These functions only run in the browser (client-side) and are ignored during server-side rendering (SSR) or pre-rendering, preventing errors when using browser-specific APIs.
- Application-wide Hook: Unlike traditional component lifecycle hooks (like ngAfterViewInit), afterRender() is an application-wide function that can be called from anywhere in your application (components, services, etc.), provided an injection context is available (e.g., in a constructor).
- Performance Optimization (Phases): You can specify an optional phase (e.g., EarlyRead, Write, Read) to control the order of DOM operations, which helps minimize performance issues like layout thrashing.

When to Use afterRender()

- Measuring DOM Elements: Getting the size or location of an element after an update.
- Syncing with Third-Party Libraries: Initializing or updating a library (e.g., a charting library) that requires direct DOM access.
- Manual DOM Updates: Performing complex animations or scrolling to specific elements that need an up-to-date DOM state.

Related Hooks

- afterNextRender(): Similar to afterRender(), but the callback runs only once after the next rendering cycle. This is ideal for one-time initialization tasks.
- afterRenderEffect(): Introduced in Angular 19, this combines the timing of afterRender() with the reactivity of signals. The callback executes after rendering, but also reacts to changes in its signal dependencies. This is more specialized and often used for advanced cases involving signal-based state and DOM updates

Specifying phases

afterrenderphase.read // write

Static : true allows access before initializing

```
@ViewChild('myRef', { static: true, read: ElementRef }) myRef!: ElementR
@ViewChild('hoverComponent', {read: HoverClickComponent}) hoverComponent
@ViewChild('containerRef', {read: ViewContainerRef}) containerRef!: View
@ViewChild(HighlightDirective, {read: HighlightDirective}) highlightDirr
```

```
<p #myRef appHighlight>view-child-example works!</p>

<div>
  <app-hover-click #hoverComponent></app-hover-click>
</div>

<div #containerRef></div>
```

Accessing multiple appeared elements

```
tasks = ['task 1', 'task 2', 'task 3', 'task 4'];
title = { name: 'angular-basics' };
counter: number = 20;
receivedDataFromChild = '';
isActive = true;

@ViewChildren(TaskComponent) taskComponents!: QueryList<TaskComponent>;

dataReceived(data: string) {
  this.receivedDataFromChild = data;
}

ngAfterViewInit() {
  console.log(this.taskComponents);
}
```

```
<div>
  <app-task *ngFor="let task of tasks" [taskName]="task"></app-task>
</div>
```

Reading as elementREf will give the all html element specification

```
@ViewChildren(TaskComponent, {read: ElementRef})
```

```
ngAfterViewInit() {
  this.taskComponents.forEach(taskComponent => {
    console.log(taskComponent);
  });
  this.taskComponents.changes.subscribe(() => {
    console.log("child components has changed");
  });

  setTimeout(() => {
    this.tasks.push("hi leela");
  }, 3000)
}
```

contentChild vs contentChildren

```ts
import { Component, ContentChild } from '@angular/core';
import { ChildComponent } from '../child/child.component';

@Component({
  selector: 'app-parent',
  standalone: true,
  imports: [],
  templateUrl: './parent.component.html',
  styleUrl: './parent.component.css'
})
export class ParentComponent {
  @ContentChild(ChildComponent) childComponent!: ChildComponent;

  ngAfterContentInit() {
    console.log(this.childComponent.showMessage());
  }
```

```
This is the root child
Hello from child component
```

```html
<app-parent>
  <app-child></app-child>
</app-parent>
```

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

```ts
tentChildren(ChildComponent, ) childComponent!: QueryList<ChildComponent>;
```

Programmatically rendering components using ngComponentOutlet

In Angular, you can dynamically render components in two primary ways: using the NgComponentOutlet directive in the template for simple scenarios, or using the ViewContainerRef class in TypeScript code for more programmatic control and complex use cases like lazy-loading or binding inputs/outputs

Using NgComponentOutlet (Template-based)

NgComponentOutlet is a structural directive that provides a simple, declarative way to render a component. It's useful when the component type to be rendered might change based on simple conditions.

```
<p>Profile for {{user.name}}</p>
<ng-container
  *ngComponentOutlet="getBioComponent();
  inputs: greetingInputs;
  outputs: greetingOutputs"
/>
```

```
import { Component, Input, signal } from '@angular/core';
import { AdminBioComponent } from './admin-bio.component';
import { StandardBioComponent } from './standard-bio.component';
@Component({
  selector: 'app-custom-dialog',
  templateUrl: './custom-dialog.component.html',
  standalone: true,
  imports: [NgComponentOutlet]
})
export class CustomDialogComponent {
  @Input() user: any; // Assume a user object with an isAdmin property
  // Define inputs and outputs for the dynamic component
  greetingInputs = signal({ name: this.user.name });
  greetingOutputs = {
    // Wire up an output event
    onRefresh: () => this.handleRefresh()
  };
  getBioComponent() {
    return this.user.isAdmin ? AdminBioComponent : StandardBioComponent;
  }
  handleRefresh() {
    console.log('Refresh event received from dynamic component');
    // Handle the event
  }
}
```

Using ViewContainerRef (TypeScript-based)

This method provides more granular control over the component lifecycle and its location in the DOM. It's ideal for scenarios requiring precise manipulation of the view or creating components from a service

```
<button (click)="loadContent()">Load content</button>
<!-- Anchor for dynamic components -->
<ng-container #container></ng-container>
```

```
import { Component, ViewChild, ViewContainerRef, inject } from '@angular/core';
import { LeafContentComponent } from './leaf-content.component';

@Component({
  selector: 'app-inner-item',
  templateUrl: './inner-item.component.html',
  standalone: true,
})
export class InnerItemComponent {
  @ViewChild('container', { read: ViewContainerRef, static: true })
  viewContainer!: ViewContainerRef;

  // For modern Angular, ComponentFactoryResolver is no longer needed
  // We can directly use viewContainer.createComponent()

  loadContent() {
    // Clear any previous components from the container
    this.viewContainer.clear();

    // Create the component and attach it to the view container
    const componentRef = this.viewContainer.createComponent(LeafContentComponent);

    // Access the component instance to set inputs or subscribe to outputs
    componentRef.instance.someInput = 'some value';
    componentRef.instance.someOutput.subscribe(event => {
      console.log('Output event:', event);
    });
  }
}
```

```
export class HostComponent {
  @ViewChild('containerRef', { read: ViewContainerRef, static: true })
  container!: ViewContainerRef;
  componentRef!: ComponentRef<DynamicComponent>;
  constructor() {}

  ngOnInit() {}

  loadComponent() {
    this.container.clear();
    this.componentRef = this.container.createComponent
    (DynamicComponent);
    this.componentRef.instance.message = 'Hi Leela';

    this.componentRef.instance.action.subscribe((messsage) => {
      console.log(message);
    });
  }
}
```

Lazyload component

```
<div>
  <ng-container *ngComponentOutlet="profileComponent"></ng-container>
</div>
```

```
export class AppComponent {
  tasks = ['task 1', 'task 2', 'task 3', 'task 4'];
  title = { name: 'angular-basics' };
  counter: number = 20;
  receivedDataFromChild = '';
  isActive = true;
  isAdmin = false;

  profileComponent: {
    new (): AdminProfileComponent | UserProfileComponent;
  } | null = null;

  @ViewChildren(TaskComponent, { read: TaskComponent })
  taskComponents!: QueryList<TaskComponent>;

  dataReceived(data: string) {
    this.receivedDataFromChild = data;
  }

  ngOnInit() {
```

```
ngOnInit() {
  this.getProfileComponent();
}
```

```
async getProfileComponent() {
  if (this.isAdmin) {
    const { AdminProfileComponent } = await import(
      './admin-profile/admin-profile.component'
    );
    this.profileComponent = AdminProfileComponent;
  } else {
    const { UserProfileComponent } = await import(
      './user-profile/user-profile.component'
    );
    this.profileComponent = AdminProfileComponent;
  }
}
```

Angular17's Change Detection Strategy: Default vs. onPush

Angular provides two main change detection strategies: Default (or CheckAlways) and OnPush (or CheckOnce). The strategy determines how often Angular checks a component and its children for updates, with OnPush offering significant performance benefits.

Default Strategy

The Default strategy (used if no strategy is specified) is comprehensive but less performant for large applications.

- How it Works: Angular checks every component in the component tree from top to bottom whenever any change might have occurred in the application.
- Triggering Events: Change detection is automatically triggered by asynchronous events that are monkey-patched by Zone.js, such as:
    - DOM events (clicks, keyup, etc.)
    - HTTP requests
    - JavaScript timers (setTimeout(), setInterval())
- Behavior: It performs "dirty checking," comparing current template expression values with previous values for all components to ensure the UI is in sync with the model, even if only a small part of the application state has changed.

OnPush Strategy

The OnPush strategy is an optimization technique that gives developers more control over when change detection runs for a component and its subtree.

- How it Works: Angular skips change detection for a component and its children unless one of the following conditions is met:
    - An @Input() property changes (Angular performs a shallow reference check: previousValue === currentValue is false).
    - An event is triggered from within the component or its children (e.g., a button click handled by a component method).
    - An observable linked to the component's template via the async pipe emits a new value (the async pipe calls markForCheck() internally).
    - Change detection is explicitly triggered manually using ChangeDetectorRef methods like markForCheck() or detectChanges().
- Benefits: This strategy significantly boosts performance by avoiding unnecessary checks in large component trees. It encourages the use of immutable data structures or observables to ensure changes are detected correctly.
- Implementation: To use it, set the changeDetection property in the component's decorator:

```
@Component({
  selector: 'app-on-push',
  templateUrl: './on-push.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush, // Set the strategy here
})
```

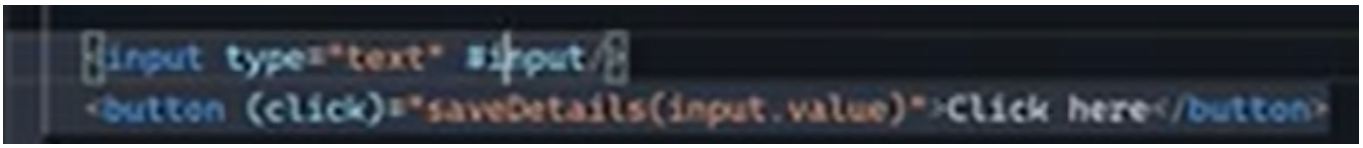
Zoneless Change Detection (Modern Angular)
With the introduction of signals, modern Angular (v17.1+) is moving towards a zoneless architecture. This approach moves away from relying on Zone.js to detect all asynchronous operations and instead uses explicit notifications when a signal used in a template is updated. This allows for highly optimized, "semi-local" change detection where only the specific components affected by a change are re-rendered, further enhancing performance.

```ts
export class ChildPushComponent {
  count$!: Observable<number>;

  constructor() {
    this.count$ = new Observable((subscriber) => {
      let value = 0;
      setInterval(() => {
        subscriber.next(value++);
      }, 1000);
    });
  }

  updateCount() {
    //this.count++;
  }
}
```

```html
<div>The count value in onpush strategy {{ count$ | async }}</div>
```

Template Expressions & Context in Angular 18

```
Text INterpolation
{{ }}


Template Statements

methods or properties

<button (click)="saveDetails();">Click Here</button>
```




```
BInding

connection between the angular

text interpolations
property binding
event binding
two-way binding
```

```
{{ }}

Assignments (=, +=, -=, ...)
Operators such as new, typeof, or instanceof
Chaining expressions with ; or ,
The increment and decrement operators ++ and --
Some of the ES2015+ operators
No support for the bitwise operators such as | and &


Interpolated expression - component instanceof
template input variable

  <ul>
    @for (customer of customers; track customer) {
      <li>{{ customer.name }}</li>
    }
  </ul>
```

```
Template referebnce variable
  <label for="customer-input">Type something:
  <input id="customer-input" #customerInput>{{ customerInput.value }}
</label>
```

```
<tr>
  <td [colSpan]="1 + 1">Leela Web Dev</td>
</tr>
```

```
<div data-value="1"></div>

[attr.data-value]="property"

One of the primary use cases for attribute binding is to set ARIA attributes.

  <button type="button" [attr.aria-label]="actionName">{{ actionName }} with Aria</button>

  colspan


  [colSpan]

  [attr.colspan]
```

```
CLass Binding



Binding a single class:
Binding to multiple classes:
Binding to multiple classes with an array:


SIgle class
class.className
```

```
<div [class.active]="isActive">This is an active div</
```

isActive = true in ts file

```
<div [ngClass]="{ active: isActive, highlight:
isHighlighted }">        You, 1 second ago • Unc
  This is an active div
</div>
```

```
classes = ['active', 'highlight'];
```

```
You, 1 second ago | 1 author (You) | Go to component
<div [ngClass]="classes">This is an active div</div>
```

```
Binding to  asingle style

[style.width] = 'width'

[style.background-color]="expression"
[style.backgroundColor]="expression"

[style]="styleexpression"
"width: 100px; height: 100px; background-color: cornflowerblue;".

{width: '100px', height: '100px', backgroundColor: 'cornflowerblue'}.

Single style binding
[style.width]="width"


Single style binding with units
[style.width.px]="width"
```

```
You, 1 hour ago | 1 author (You) | Go to component
<div [style.background-color]="isHighlighted ? 'yellow' :
'white'">
  Highlighted Content
</div>      You, 1 hour ago • Uncommitted changes
```

```
<p [style.fontSize.px]="fontSize">This text size changes
dynamically.</p>
```

```
export class AppComponent {
  myStyles = 'width: 100px; height: 100px;
  background-color: cornflowerblue;';
}
```

```
<div [style]="myStyles">This element has multiple styles
applied.</div>      You, 3 minutes ago • Uncommitted chan
```

Event binding

```
<button>click here</button>

(event-name)="template sta,ement "
```

Two way data binding

```
[()]

<app-sizer [(sizer)]="variable"></app-sizer>
```

Property bind + event binding

```
export class SizerComponent {
  @Input() size = 0;
  @Output() onSizeChange = new EventEmitter<number>();

  Inc() {
    this.resize(1);
  }

  Dec() {
    this.resize(-1);
  }

  resize(value: number) {
    const sizeValue = this.size + value;
    this.onSizeChange.emit(sizeValue);
  }
}
```

```
<app-sizer [size]="fontSizePx" (onSizeChange)="fontSizePx =
$event"></app-sizer>
```

Two way binding solution for the above example:

```
export class SizerComponent {
  @Input() size = 0;
  @Output() sizeChange = new EventEmitter<number>();

  Inc() {
    this.resize(1);
  }

  Dec() {
    this.resize(-1);
  }

  resize(value: number) {
    const sizeValue = this.size + value;
    this.onSizeChange.emit(sizeValue);
  }
}
```

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
```

this event emitter must be follow this convention(<input>Change)

Angular 18 Control Flow: Using If and For Loop Effectively

Older style

```
<div *ngIf="a > b">{{ a }} is greater than {{ b }}</div>
<div *ngIf="a < b">{{ a }} is less than {{ b }}</d
<div *ngIf="a === b">{{ a }} is equal to {{ b }}</
```

```
<div>
  @switch(userRole) { @case('admin') {
  <div>You are an administrator</div>
  } @case('user') {
  <div>You are an regular user</div>
  } @default {
  <div>your role is not recognized</div>
  } }
</div>
```

Modern style

```
<div>
  @if (a > b) {
  <div>{{ a }} is greater than {{ b }}</div>
  } @else if (a < b) {
  <div>{{ a }} is less than {{ b }}</div>
  } @else {
    <div>{{a}} is equal to {{}}</div>
  }
</div>
```

```
<div>
  @if (users$ | async; as users) {
  <div>{{ users.length }}</div>
  }
</div>
```

```
import { of } from 'rxjs';
export class AppComponent {
  a = 20;
  b = 16;
  users$ = of([1,2,3,4])
}
```

This track value should be unique value. Used as a unique key to dom elements

```
<div>
  @for(item of items; track item) {
  <div>{{ item }}</div>
  }
</div>
```

```
<div>
  @for(item of items; track item) {
  <div>value: {{ item }}</div>
  } @empty {
  <div>No items in the array</div>
  }
</div>
```

```
<div>
  @for(item of items; track item; let count=$count,
  index=$index, first=$first,
  last=$last, even=$even, odd=$odd) {
  <div>Value: {{ item }}</div>
  <div>Count: {{ count }}</div>
  <div>Index: {{ $index }}</div>
  <div>First: {{ $first }}</div>
  <div>last: {{ $last }}</div>
  <div>even: {{ $even }}</div>
  <div>odd: {{ $odd }}</div>
  <hr />
  }
</div>
```

```
The @if block replaces *ngIf for expressing conditional parts of the UI.


The @switch block replaces ngSwitch with major benefits:

it does not require a container element to hold the condition expression or each conditional template;
it supports template type-checking, including type narrowing within each branch.


The @for block replaces *ngFor for iteration, and has several differences compared to its structural directive NgFor predecessor:

tracking expression (calculating keys corresponding to object identities) is mandatory but has better ergonomics (it is enough to write an expression instead of creating the trackBy method);

uses a new optimized algorithm for calculating a minimal number of DOM operations to be performed in response to changes in a collection, instead of Angular's customizable diffing implementation (IterableDiffer);
has support for @empty blocks.
```

The track setting replaces NgFor's concept of a trackBy function. Because @for is built-in, we can provide a better experience than passing a trackBy function, and directly use an expression representing the key instead. Migrating from trackBy to track is possible by invoking the trackBy function:

Local template variables let and its usage in angular Template developer preview

```
@let userName = user.name; @let greeting = 'Hello' + ' '
+userName; @let details
= details$ | async;

<div>{{ userName }}</div>
<div>{{ greeting }}</div>

@if (details) {
<div>{{ details.length }}</div>
} @else {
<div>No Details found</div>
} @let value = 1;

<button (click)="value = value + 1">Click here</button>
```

Pipes

```
Pipes are simple functions to use in templates to accept an input value and return a transformed value.

Pipes are useful because you can use them throughout your application, while only declaring each pipe once.

For example, you would use a pipe to show a date as April 15, 1988 rather than the raw string format.

You can create your own custom pipes to expose reusable transformations in templates.

To apply a pipe, use the pipe operator (|) within a template expression as shown in the following code example.
```

```
<div>{{ birthday | date : "d MMMM, " }}</div>

<div>{{ "LeelaWebDev" | lowercase }}</div>

<div>{{ 1234.5 | currency : "USD" : "code" : "1.3-3" }}</div>
<div>{{ 1234.5 | number : "1.2-2" }}</div>

<div>{{ 0.1234 | percent : "1.2-2" }}</div>

<p>{{ [1, 2, 3, 4, 5] | slice : 1 : 3 }}</p>
<p>{{ "ANGULAR" | slice : 0 : 3 }}</p>

<p>{{ { name: "leela" } | json }}</p>

<p>{{ details$ | async }}</p>
```

20 July,
leelawebdev
USD1,234.500
1,234.50
12.34%

2,3

ANG

{ "name": "leela" }

1,2,3,4

```
import { Pipe, PipeTransform } from
'@angular/core';

@Pipe({
  standalone: true,
  name: 'greet',
})
export class GreetPipe implements
PipeTransform {
  transform(value: string): string {
    return `Hello ${value}`;
  }
}
```

```
<p>{{ "Leela" | greet }}</p>
```

```
import { Pipe, PipeTransform } from
'@angular/core';

@Pipe({
  standalone: true,
  name: 'exponentialStrength',
})
export class ExponentialStrengthPipe
implements PipeTransform {
  transform(value: number, exponent: number)
  : number {
    return Math.pow(value, exponent);
  }
}
```

```
<p>{{ 2 | exponentialStrength : 10 }}</p>
```

Directives

```
You, 47 seconds ago | 1 author (You) | Go to component
Directives are classes that add additional behavior to elements
in your Angular
applications.


Use Angular's built-in directives to manage forms, lists, styles,
and what users see.

Components
Attribute Directives
Structiral Directives
```

```
Builtin Directives
ngClass
ngSTyle
ngModel
```

Custom directives

```
import { Directive, Input, TemplateRef, ViewContainerRef } from
'@angular/core';

@Directive({
  selector: '[appHasPermission]',
  standalone: true,
})
export class HasPermissionDirective {
  @Input() appHasPermission!: string;
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) {}

  ngOnInit() {
    const hasPermission = this.appHasPermission === 'admin';
    if (hasPermission) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
}
```

```
import { Directive, Input, TemplateRef, ViewContainerRef } from
'@angular/core';

@Directive({
  selector: '[appDynamicList]',
  standalone: true,
})
export class DynamicListDirective {
  @Input() appDynamicList!: any[];
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) {}

  ngOnInit() {
    this.viewContainer.clear();
    for (const item of this.appDynamicList) {
      const context = { $implicit: item };
      this.viewContainer.createEmbeddedView(this.templateRef,
      context);
    }
  }
}
```

```
1 <div *appHasPermission="'admiddn'">This is the admin</div>
2
3 <div *appDynamicList="items; let item = $item">{{ item.name }}</div>
```

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

@Directive({
  selector: '[appColor]',
  standalone: true,
})
export class ColorDirective {
  renderer = inject(Renderer2);
  hostEl = inject(ElementRef).nativeElement;

  constructor() {}

  ngOnInit() {
    this.renderer.setStyle(this.hostEl, 'color', 'red');
    this.renderer.setStyle(this.hostEl, 'border', '1px solid black');
    this.renderer.setStyle(this.hostEl, 'padding', '1px solid black');
  }
}
```

Leveraging two directives

```
<div>                          (directive) BackgroundcolorDirect
  <app-widget appColor appBackgroundcolor> </app-widget>
</div>
```

We can use host directives (directive composition)

```
@Component({
  selector: 'app-widget',
  standalone: true,
  imports: [],
  templateUrl: './widget.component.html',
  styleUrl: './widget.component.css',
  hostDirectives: [ColorDirective, BackgroundcolorDirective],
})
export class WidgetComponent {}
```

Making reusable directives

```
<div>
  <app-widget color="yellow"> </app-widget>
</div>
```

```
@Component({
  selector: 'app-widget',
  standalone: true,
  templateUrl: './widget.component.html',
  styleUrl: './widget.component.css',
  hostDirectives: [
    {
      directive: ColorDirective,
      inputs: ['color'],
    },
    BackgroundcolorDirective,
  ],
})
export class WidgetComponent {}
```

```
export class ColorDirective {
  renderer = inject(Renderer2);
  hostEl = inject(ElementRef).nativeElement;

  @Input() color = 'red';

  constructor() {}

  ngOnInit() {
    this.renderer.setStyle(this.hostEl, 'color', this.color);
    this.renderer.setStyle(this.hostEl, 'border', '1px solid black');
    this.renderer.setStyle(this.hostEl, 'padding', '8px');
  }
}
```

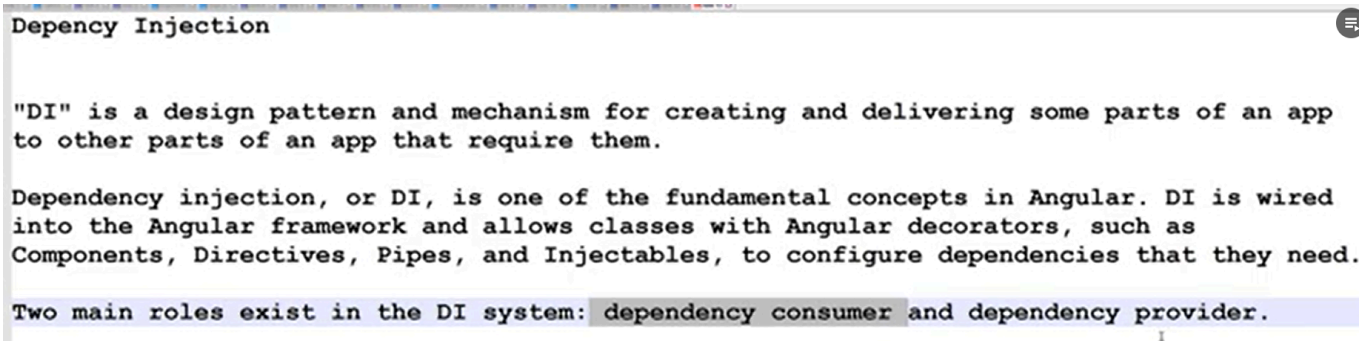
Events

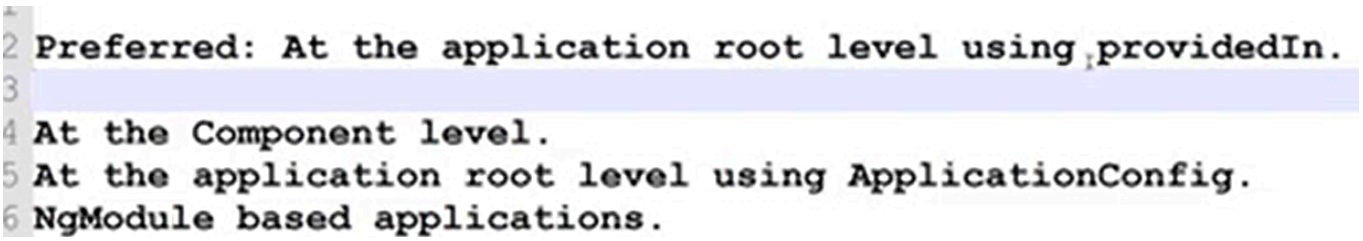
```
@Directive({
  selector: '[appColor]',
  standalone: true,
})
export class ColorDirective {
  renderer = inject(Renderer2);
  hostEl = inject(ElementRef).nativeElement;

  @Input() color = 'red';
  @Output() colorChanged = new EventEmitter();

  constructor() {}

  ngOnInit() {
    this.renderer.setStyle(this.hostEl, 'color', this.color);
    this.renderer.setStyle(this.hostEl, 'border', '1px solid black');
    this.renderer.setStyle(this.hostEl, 'padding', '8px');

    setInterval(() => {
      this.colorChanged.emit();
    }, 1000);
  }
}
```

```
@Component({
  selector: 'app-widget',
  standalone: true,
  templateUrl: './widget.component.html',
  styleUrl: './widget.component.css',
  hostDirectives: [
    {
      directive: ColorDirective,
      inputs: ['color'],
      outputs: ['colorChanged'],
    },
    BackgroundcolorDirective,
  ],
})
export class WidgetComponent {}
```

```
<div>
  <app-widget color="yellow" (colorChanged)="doSomething"> </app-widget>
</div>
```

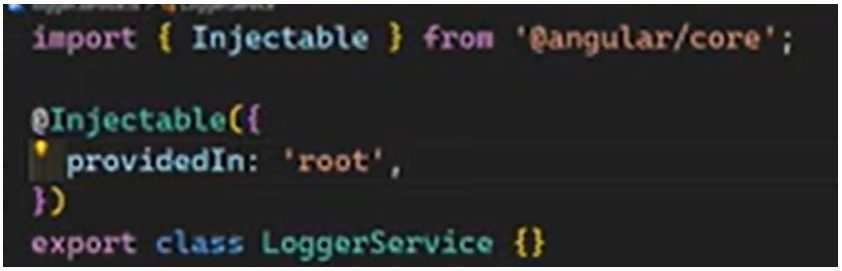
Angular 18 Dependency Injection





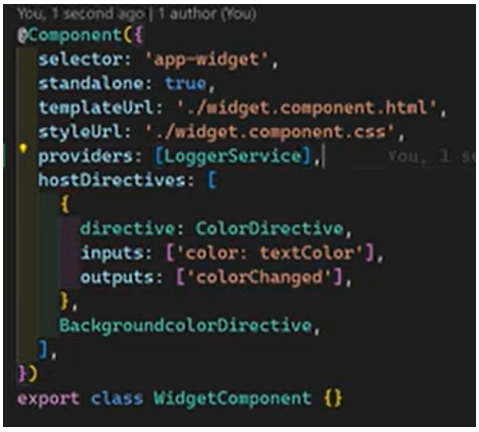
Providing dependecy

At the application root level using providedin



single shared instance

At the component level

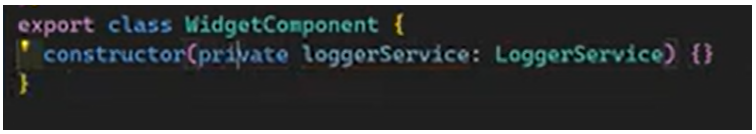


instance for this component and its child components

rootlevel using application config

Consuming dependencies

constructor method

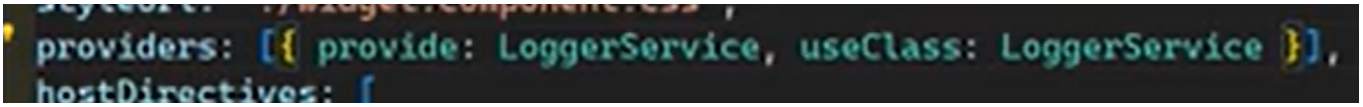


Inject method

Using useClass in standalone components —----------

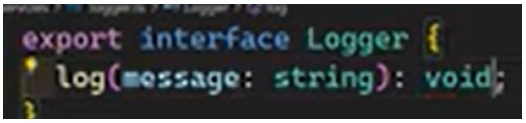
Providers: [LoggerService]

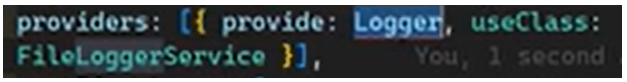
Expanded version >>>>>



useClass, useExisting, useFactory, useValue

Best practices for using interfaces





useFactory for Dynamic Dependency Injection —----------

conditionally injecting services

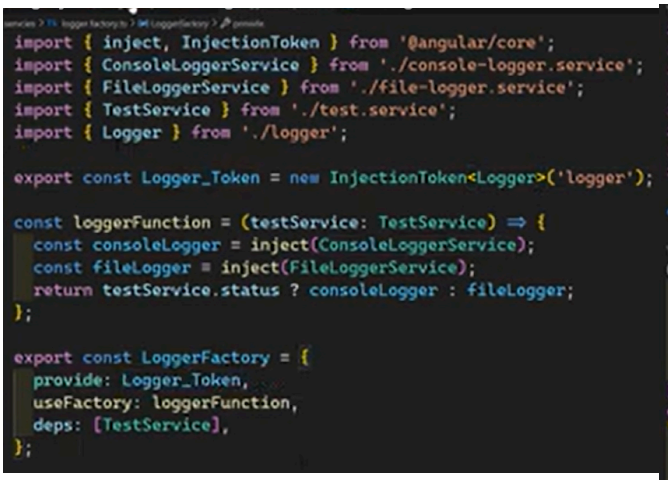
useValue and useExisting —----------

Injection Context in Angular 18: Understanding Different Runtime Contexts

The injection process happens at runtime context. Most of the time up to now we have injected services in constructor of any specific class, component or service
As well as we have seen the service factory

```
import { ConsoleLoggerService } from './console-logger.service';
import { FileLoggerService } from './file-logger.service';
import { TestService } from './test.service';

const loggerFunction = (testService: TestService) => {
  return testService.status
    ? new ConsoleLoggerService()
    : new FileLoggerService();
};

export const LoggerFactory = {
  provide: ConsoleLoggerService,
  useFactory: loggerFunction,
  deps: [TestService],
};
```

Using tokens



```
import { inject, InjectionToken } from '@angular/core';
import { ConsoleLoggerService } from './console-logger.service';
import { FileLoggerService } from './file-logger.service';
import { TestService } from './test.service';
import { Logger } from './logger';

export const Logger_Token = new InjectionToken<Logger>('logger');

const loggerFunction = (testService: TestService) => {
  const consoleLogger = inject(ConsoleLoggerService);
  const fileLogger = inject(FileLoggerService);
  return testService.status ? consoleLogger : fileLogger;
};

export const LoggerFactory = {
  provide: Logger_Token,
  useFactory: loggerFunction,
  deps: [TestService],
};
```

```
import { Component, inject, Inject, InjectionToken } from '@angular/core';
import { ColorDirective } from '../color.directive';
import { BackgroundcolorDirective } from '../backgroundcolor.directive';
import { FileLoggerService } from '../servcies/file-logger.service';
import { ConsoleLoggerService } from '../servcies/console-logger.service';
import { Logger } from '../servcies/logger';
import { TestService } from '../servcies/test.service';
import { LoggerFactory } from '../servcies/logger.factory';

const Logger_Token = new InjectionToken<Logger>('logger');

@Component({
  selector: 'app-widget',
  standalone: true,
  templateUrl: './widget.component.html',
  styleUrl: './widget.component.css',
  providers: [LoggerFactory],
})
export class WidgetComponent {
  constructor(@Inject(Logger_Token) private logger: Logger) {
    this.logger.log('hi leela');
  }
}
```

Routes

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs'; 144k (gzipped: 28.6k)

@Injectable({
  providedIn: 'root',
})
export class PermissionService {
  isTeamPresent(userToken: any, teamId: string):
  Observable<boolean> {
    const hasPermission =
      userToken && userToken.permissions.includes('team:$
      {teamId}');
    return of(hasPermission);
  }
}

export const UserToken = {
  permissions: ['team:1', 'team:2'],
};
```

```
import { Routes } from '@angular/router';
import { canActivateTeam } from './servcies/
canActivateTeam';
import { TeamComponent } from './team/team.component';

export const routes: Routes = [
  {
    path: 'team/:id',
    component: TeamComponent,
    canActivate: [canActivateTeam],
  },
];
```

```
import { inject } from '@angular/core';
import {
  ActivatedRouteSnapshot,
  CanActivateFn,
  RouterStateSnapshot,
} from '@angular/router';
import { PermissionService, UserToken } from './permission.
service';

export const canActivateTeam: CanActivateFn = (
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
) => {
  const permissionService = inject(PermissionService);
  return permissionService.isTeamPresent(UserToken, route.
  params['id']);
};
```

Continue upto part 82….

Signals

Angular Signals are a system that granularly tracks how and where your state is used throughout an application, allowing the framework to optimize rendering and improve runtime performance. Introduced in Angular 16 and expanded in subsequent versions (including v19 in late 2024 and v20 in 2025), they provide a simpler, synchronous reactive model compared to RxJS

Core Concepts and Primitives

- Writable Signals: Create a signal with an initial value. You can change its value directly using .set(newValue) or .update(current => newValue).

```
const count = signal(0);
count.set(1); // Explicitly set to 1
count.update(v => v + 1); // Increment to 2
```

  Computed Signals: Read-only signals derived from other signals. They are lazily evaluated and memoized, meaning they only re-calculate when their dependencies change and are accessed.

- Effects: Functions that run whenever the signals they depend on change. These are typically used for side effects like logging, manual DOM manipulation, or syncing with local storage.
- Signal Inputs & Queries: Modern alternatives to @Input, @ViewChild, and @ContentChild that treat incoming data and DOM queries as signals for better reactivity.
- Resource API: Introduced in newer versions (v19/v20) to handle asynchronous operations, like fetching data from a server, directly within the signal system.

Why Use Signals?

1. Fine-Grained Reactivity: Unlike standard change detection that may check the entire component tree, signals allow Angular to update only the specific parts of the UI that depend on changed data.
2. Performance: They reduce the reliance on zone.js, leading to faster runtime execution and smaller bundle sizes in "zoneless" applications.
3. Predictability: Signals are strictly synchronous and always have a value, avoiding the complexities of asynchronous data streams found in RxJS.
4. Integration with OnPush: Using signals in templates automatically marks OnPush components for check, simplifying manual change detection management.

Best Practices

- Avoid Mutating Objects Directly: When a signal holds an object or array, replace the entire value instead of mutating properties to ensure Angular detects the change.
- Prefer Computed over Effects: Use computed() for deriving data and save effect() for external side effects.
- Encapsulation in Services: Keep writeable signals private within services and expose them as read-only signals using .asReadonly() to control how state is modified.

Signals in Angular are a way to manage state and reactive data flows, similar to how observables or BehaviorSubject are used in RxJS.

However, signals are more lightweight and built directly into Angular's reactive system. Here's a detailed breakdown with explanations and examples.

What Are Signals?
A signal is essentially a wrapper around a value that notifies consumers (functions, components, or services) whenever the value changes.

This allows for a reactive data flow, where changes in the signal's value can trigger updates elsewhere in the application.

Characteristics of Signals:
They can hold any type of value: primitives like numbers or strings, or more complex structures like objects or arrays.
Signals expose a getter function for reading their current value.
Signals notify Angular when their value is used, allowing Angular to track dependencies automatically. This is useful for creating reactive applications.

Types of Signals
There are two primary types of signals in Angular:

Writable signals: These can be updated by the developer, meaning you can both read and modify their values.

Read-only signals: These only allow reading the value but cannot be modified directly.

```
export class Footer implements OnInit {

  // countVal: number = 3;
  countVal: WritableSignal<number> = signal(3);

  ngOnInit(): void {
    setInterval(() => {
      this.countVal.update(val => val + 1)
    }, 500)
  }

}
```

```
<app-counter [count]="countVal"></app-counter>
```

```
export class Counter {

  @Input() count: any = 0;
}
```

```
<div>
  current count value: {{ count() }}
</div>
```

Computed Signals

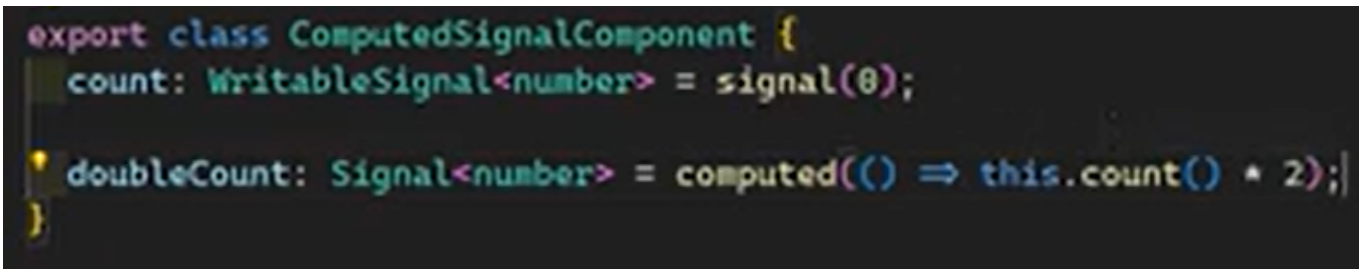
Computed signals in Angular are a powerful way to derive values from other signals and track dependencies between them. Let's break down the concept of computed signals, including their lazy evaluation and memoization, with detailed examples.

Computed signals are read-only signals that derive their values based on other signals. Unlike writable signals, computed signals cannot be directly modified. They are "computed" by a function that uses other signals as input

To create a computed signal, you use the computed() function and pass a derivation function. Here's a simple example:

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

Whenever the count signal changes, Angular automatically knows that the doubleCount signal needs to update because it depends on count. Angular will trigger re-evaluation of doubleCount the next time you access its value.



Lazy Evaluation

Computed signals are lazily evaluated. This means that Angular does not compute their values until you actually read them. If you never use the doubleCount() signal in your code, its derivation function is never executed.

Memoization

Computed signals are also memoized, which means they cache their calculated value. Once the value is computed, Angular will store it. If you read the value again without any changes to the dependencies (e.g., count in our case), Angular returns the cached value instead of recomputing it.

When to Use Computed Signals

Data Transformation: Use computed signals when you need to transform or compute data based on other signals.

Expensive Operations: If the derivation function is expensive (e.g., sorting or filtering large datasets), computed signals are ideal because they ensure that the computation only happens when necessary, and the result is cached.

Read-Only Data: Since computed signals are read-only, they are perfect for data that should not be directly modified by components or services but rather derived from other sources.

Conclusion

Computed signals in Angular provide a reactive, efficient way to derive data based on other signals. Their lazy evaluation and memoization ensure that computations are performed only when needed and avoid unnecessary recalculations. This makes them ideal for scenarios where performance is critical, such as filtering or transforming large datasets, or performing expensive computations.

Dynamic Dependencies: Computed Signals

```
export class ComputedSignalComponent {
  showCount = signal(false);
  count: WritableSignal<number> = signal(0);

  doubleCount: Signal<number> = computed(() => this.count() * 2);

  conditionalCount = computed(() => {
    console.log('executing the computed');
    if (!this.showCount()) {
      return 'nothing to show here';
    } else {
      return `the count value is ${this.count()}`;
    }
  });

  ngOnInit() {}      You, 1 minute ago • Uncommitted changes

  getCondition() {
    console.log(this.conditionalCount());
  }

  increment() {
    this.count.update((value) => value + 1);
  }

  updateshowcount() {
    this.showCount.update((value) => !value);
```

At the beginning conditional count executed once and evaluated, In there this.showcount is false and goes inside if block hence else block is not evaluated. Therefore at that time showcount is only the dependency for the computed signal. Afterwards if increment count happened any round the conditionalcount computed signal not evaluated since its dependency is only the showcount at this time

Once the update show count is executed show count is updated (false -> true), then evaluated computed signal, then goes to the else block. Now the dynamic dependency is the count value. Now this signal computed function executed for always when the increment count happens

Effects

Effects allow you to trigger side effects when one or more signal values change, and they serve as an important tool for integrating signals into different parts of an application. Here's a detailed breakdown of each section with examples:

What is an Effect?

An effect is a function that runs whenever a signal changes. Signals in Angular are reactive variables that notify consumers when their values are updated. When you create an effect using the effect() function, Angular tracks the signals used within that effect and re-runs the function when any of those signals change.

```
const count = signal(0);

effect(() => {
  console.log(`The current count is: ${count()}`);
});
```

How Effects Work:

Initial Execution:

Effects always run at least once when they are created. This means the above effect() will immediately log the count, even if count has not yet changed.

Effects always run at least once when they are created. This means the above effect() will immediately log the count, even if count has not yet changed.

Tracking Dependencies:

When the effect runs, it tracks the signals it interacts with. In this case, it tracks count(). If the value of count changes, Angular knows to run the effect again.

Asynchronous Execution:

Effects execute asynchronously during Angular's change detection process. This means the effect will be re-executed after Angular finishes processing changes in the application state.

```
export class ComputedSignalComponent {
  count = signal(0);

  increment() {
    this.count.update((value) => value + 1);
  }

  constructor() {      You, 29 seconds ago • Uncommitted ch
    effect(() => {
      console.log('The current count is: ${this.count()}');
    });
  }
}
```

```
effect(() => {
  a.set(b() + 1); // BAD PRACTICE: This creates a circular update loop
});

In this case, if b changes, a will also change. But when a is updated, the effect will run
again, potentially creating a loop or an ExpressionChanged error.
```

Use Cases for Effects:

Although effects are rarely needed in most application code, they are valuable for side-effect-heavy operations where reactive signals interact with non-Angular systems, such as logging, browser APIs, or custom rendering.

When Not to Use Effects:

Avoid using effects for state propagation—i.e., don't use effects to drive changes in one signal based on another. This can lead to unwanted consequences like:

ExpressionChangedAfterItHasBeenChecked errors.
Infinite loops of re-triggered updates.
Unnecessary change detection cycles, which hurt performance.

When to Use Computed Signals Instead:
If you want to model state that depends on other state, use computed signals rather than effects. Computed signals automatically derive values based on other signals and can propagate changes without risk of side effects.

Summary:
Effects are a powerful way to trigger operations based on signal changes, but they should be used cautiously.
Common use cases include logging, DOM manipulation, syncing with browser APIs like localStorage, and custom rendering with third-party libraries.
Do not use effects for state propagation, as this can cause errors or performance issues.
Instead, rely on computed signals for handling reactive state changes.

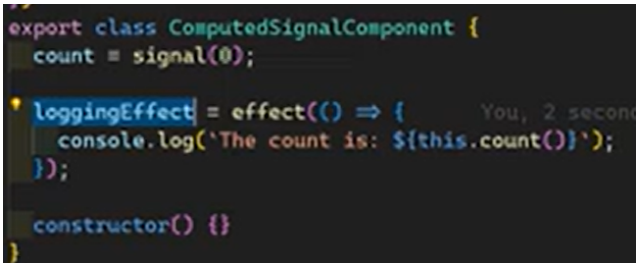
How to Create and Destroy Effects

Understanding Injection Context and Effects in Angular Signals

In Angular, signals allow for reactive data flow. With effects, you can respond to changes in signal values. However, to create an effect, it needs to be done within an injection context where Angular's inject() function can be accessed. This is typically within a component, directive, or service.

Creating Effects in a Component Constructor

When creating an effect inside a component, it monitors the changes to a signal and performs some action when the signal updates. The easiest way to register an effect is within a component's constructor, because the component provides an injection context by default.



Creating effect outside from the inject context (this time constructor injection context) required to manually inject into the injection context

Equality functions —----------

In the context of Angular signals, equality functions allow you to customize how changes to signal values are detected.

By default, Angular signals use referential equality (===) to determine if a new value is different from the previous one.

However, this approach only checks if two variables reference the same object in memory.

For complex data types like arrays or objects, even if the contents are the same, creating a new instance of the array or object would not be considered equal using referential equality.

Equality functions can be provided to both writable and computed signals to alter how the equality check is performed.

This is useful when dealing with more complex data types where you want to compare the content (deep equality) rather than just the references.

Assigning same value ([“test”]) leads to execute effect once again since it creates new reference although it’s content is same as previous

Deep equality check with Loadsh isEqual function

Default behavior: Signals use referential equality (===), meaning any new instance of an object or array is considered different, even if the contents are the same.
Custom equality functions: You can override the default behavior with a custom function like Lodash's _.isEqual to perform deep equality checks, ensuring that updates only happen when the content truly changes.
Writable and computed signals: Both writable and computed signals can take advantage of custom equality functions to fine-tune when updates occur.

Read Without Tracking Dependencies —----------

In Angular's reactive system, signals are used to manage and track state changes efficiently.

When working with signals, especially within reactive functions like computed or effect, it's crucial to control which signal reads are tracked as dependencies.

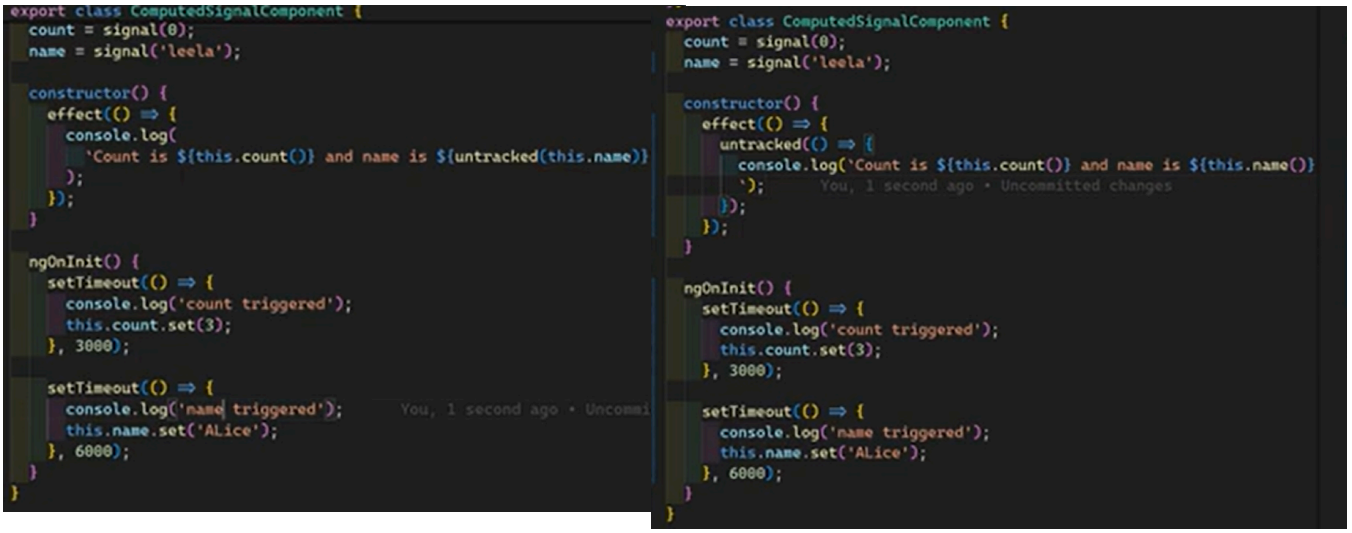
By default, reading a signal's value within such functions establishes a dependency, causing the function to re-evaluate whenever the signal changes.

However, there are scenarios where you might want to read a signal without creating such a dependency. This is where the untracked function becomes useful.

Understanding untracked

The untracked function in Angular allows you to read a signal's value without marking it as a dependency of the current reactive context.

This means that changes to the untracked signal won't trigger re-evaluations of the computed or effect function. This is particularly beneficial when you want to perform operations that shouldn't react to certain signal changes.



```
export class ComputedSignalComponent {
  count = signal(0);
  name = signal('leela');

  constructor() {
    effect(() => {
      console.log(
        `Count is ${this.count()} and name is ${untracked(this.name)}`
      );
    });
  }

  ngOnInit() {
    setTimeout(() => {
      console.log('count triggered');
      this.count.set(3);
    }, 3000);

    setTimeout(() => {
      console.log('name triggered');
      this.name.set('ALice');
    }, 6000);
  }
}
```

```
export class ComputedSignalComponent {
  count = signal(0);
  name = signal('leela');

  constructor() {
    effect(() => {
      untracked(() => {
        console.log(`Count is ${this.count()} and name is ${this.name()}
        `);
      });
    });
  }

  ngOnInit() {
    setTimeout(() => {
      console.log('count triggered');
      this.count.set(3);
    }, 3000);

    setTimeout(() => {
      console.log('name triggered');
      this.name.set('ALice');
    }, 6000);
  }
}
```

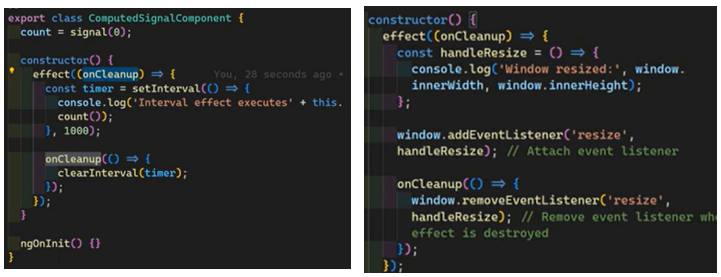
Cleanups —----------

What Are Effects in Angular Signals?

In Angular signals, effects allow you to respond to changes in signals without returning any values (i.e., performing side effects like logging, fetching data, or interacting with the DOM). Sometimes these effects need to be cleaned up when a signal changes or when the effect is destroyed.

Cleanup Function in Effects

When you define an effect, you can optionally pass an onCleanup function. This function allows you to register cleanup logic to be executed before the effect runs again or when the effect is destroyed (e.g., navigating away from a component that uses the effect).



```
export class ComputedSignalComponent {
  count = signal(0);

  constructor() {
    effect((onCleanup) => {
      const timer = setInterval(() => {
        console.log('Interval effect executes' + this.
        count());
      }, 1000);

      onCleanup(() => {
        clearInterval(timer);
      });
    });
  }

  ngOnInit() {}
}
```

```
constructor() {
  effect((onCleanup) => {
    const handleResize = () => {
      console.log('Window resized:', window.
      innerWidth, window.innerHeight);
    };

    window.addEventListener('resize',
    handleResize); // Attach event listener

    onCleanup(() => {
      window.removeEventListener('resize',
      handleResize); // Remove event listener wh
      effect is destroyed
    });
  });
```

Signals & RxJS Interop —----------

rxjs-interop package

toSignal:

The toSignal function converts an RxJS Observable into an Angular signal. Signals are reactive variables in Angular that store state and notify the app whenever their value changes. toSignal automatically subscribes to the Observable and updates the signal's value as the Observable emits new values

```
import { toSignal } from '@angular/core/
rxjs-interop';

You, 1 minute ago | 1 author (You)
@Component({
  selector: 'app-computed-signal',
  standalone: true,
  imports: [],
  templateUrl: './computed-signal.component.html',
  styleUrl: './computed-signal.component.css',
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ComputedSignalComponent {
  counterObservable = interval(1000);

  counter = toSignal(this.counterObservable, {
  initialValue: 0 });       You, 1 minute ago • Unco
}
```

```
<div>Counter: {{ counter() }}</div>
<div>Counter: {{ counterObservable | async }}</div>
```

manualCleanup

What it does:
By default, when a component that uses toSignal is destroyed, Angular automatically unsubscribes from the Observable. However, you can override this behavior using the manualCleanup option, which allows you to control when to clean up (unsubscribe) from the Observable

```
  counterObservable = interval(1000);
  private subscription!: Subscription;

  counter = toSignal(this.counterObservable, {
    initialValue: 0,
    manualCleanup: true,
  });

  ngOnInit() {
    this.subscription = this.counterObservable.
    subscribe();
  }

  ngOnDestroy() {
    if (this.subscription) {
      this.subscription.unsubscribe(); // Manually
      clean up the subscription       You, 2 second
    }
  }
```

```
export class ComputedSignalComponent {
  errorMessage: string = '';
  // Observable that immediately throws an error
  observable = throwError(() => new Error('An
  error occurred!'));       You, 27 seconds ago • U
  value = toSignal(this.observable);

  ngOnInit() {
    try {
      this.value(); // Trying to read the signal
      will throw the error
    } catch (error: any) {
      this.errorMessage = error.message; // Catch
      the error and display it
    }
  }
}
```

toObservable —----------

toObservable is a utility in Angular that converts a reactive signal into an Observable.

This allows us to take the value of a signal, which is used in Angular's reactive system, and use it with RxJS operators like switchMap, map, etc. Essentially, it lets you integrate Angular's signals with RxJS.

Monitoring Signal Changes: When a signal's value changes, the toObservable utility tracks the changes using an effect and emits the updated value through the Observable. This is helpful for integrating signals with asynchronous streams like HTTP requests.

Injection Context: toObservable relies on an injection context (e.g., a component or service). If there's no automatic injection context, you can provide an Injector manually.

On subscription, toObservable might emit the last known value of the signal immediately (if available) via a ReplaySubject.

Signals don't emit values synchronously like Observables do. Even if you update a signal multiple times in quick succession, only the last stable value will be emitted.

```ts
import { interval, of, Subscription, switchMap, throwError } from 'rxjs';
import { toObservable, toSignal } from '@angular/core/rxjs-interop';
import { CommonModule } from '@angular/common';

You, 1 second ago | 1 author (You)
@Injectable({
  providedIn: 'root',      You, now • Uncommitted changes
})
class QueryService {
  query = signal('');
}

You, 1 second ago | 1 author (You)
@Component({
  selector: 'app-computed-signal',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './computed-signal.component.html',
  styleUrl: './computed-signal.component.css',
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ComputedSignalComponent {
  query = inject(QueryService).query;

  query$ = toObservable(this.query);

  results$ = this.query$.pipe(switchMap((query) => of(query)));

  onQueryChange(event: Event) {
    this.query.set((event.target as HTMLInputElement).value);
  }
}
```

```html
<div><input type="text" (input)="onQueryChange($event)" /></div>

<ul>
  @for (result of results$ | async; track result) {
  <li>{{ result }}</li>
  }
</ul>
```

outputFromObservable / outputToObseravable —----------

```
export class ComputedSignalComponent {
@Output() nameChange = new EventEmitter<string>();
 nameChange$ = new Observable((observer) => {
   let counter = 1;
   setInterval(() => {
     this.nameChange.emit('name changed - ${counter++}');
     observer.next('name changed - ${counter++}');
   }, 1000);
 });

 ngOnInit() {
   this.nameChange$.subscribe();
 }
}
```

```
export class AppComponent {

 onNameChange(value: string) {
   console.log(value)
 }
}
```

```
<app-computed-signal (nameChange)="onNameChange($event)"></app-computed-signal>
```

outputfromobserablbel

```
@Component({
 selector: 'app-computed-signal',
 standalone: true,
 imports: [],
 templateUrl: './computed-signal.component.html',
 styleUrl: './computed-signal.component.css',
 changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ComputedSignalComponent {
 nameChange$ = new Observable((observer) => {
   let counter = 1;
   setInterval(() => {
     observer.next('name changed - ${counter++}');
   }, 1000);
 });

 nameChange = outputFromObservable(this.nameChange$);
}
```

```
export class AppComponent {
 nameChange$!: Observable<string>;
 @ViewChild('computedSignal') computedSignal: ComputedSignalComponent | null =
   null;

 ngAfterViewInit() {
   if (this.computedSignal) {
     this.nameChange$ = outputToObservable(this.computedSignal.nameChange);
   }
 }
}
```

Signal Input —----------

Overview of Signal Inputs
Signal Inputs are a feature in Angular that allow component values to be bound from parent components.

These values are exposed using a Signal, which can change throughout the lifecycle of your component. This feature is currently in developer preview.

Optional Inputs: By default, inputs are optional unless specified otherwise. You can provide an initial value, or it defaults to undefined.
Required Inputs: These inputs must have a value and are declared using input.required.

```
export class ComputedSignalComponent {
 firstName = input<string>();
 age = input(0);

 lastName = input.required<string>();
}
```

```
<app-computed-signal
 [firstName]="firstName"
 [lastName]="lastName"
 [age]="age"
></app-computed-signal>
```

```
age = input(0, { alias: 'studentAge' });
```

```
ageMultiplied = computed(() => this.age() * 2);
```

```
@Component({
  selector: 'app-computed-signal',
  standalone: true,
  imports: [],
  templateUrl: './computed-signal.component.html',
  styleUrl: './computed-signal.component.css',
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ComputedSignalComponent {
  firstName = input<string>();
  age = input(0, { alias: 'studentAge' });

  lastName = input.required<string>();

  ageMultiplied = computed(() => this.age() * 2);

  constructor() {
    effect(() => {
      console.log(this.firstName());
    });
  }
}
```

```
export class ComputedSignalComponent {
  disabled = input(false, {
    transform: (value: boolean | string) => {
      return typeof value === 'string' ? value === '' : value;
    },
  });
}
```

Type Safety:

Signal inputs provide better type safety, as they don't require you to give initial values or use workarounds to convince TypeScript that an input will always have a value.

Automatic Transform Checking:

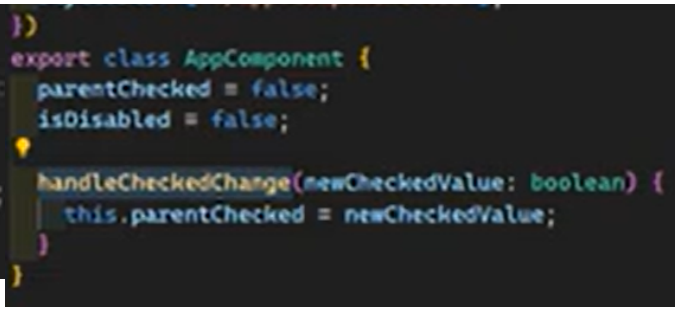
Transforms are checked automatically to ensure that they match the expected input types.

Automatic Change Detection:

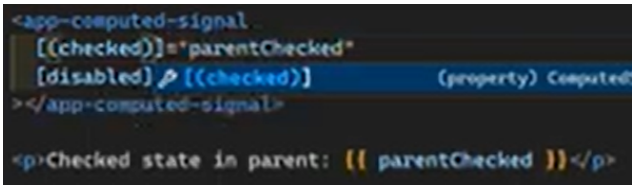
When used in templates, signal inputs automatically mark OnPush components as dirty.

Model Inputs —----------

```
Overview of Model Inputs in Angular


Model inputs are a new Angular feature (currently in developer preview) that
allow two-way data binding directly within a component.

This means that a component can both read and write values to a property bound
from a parent component, enabling the propagation of new values back to the
parent.

Key Differences Between Model Inputs and Standard Inputs:
Model Inputs: These allow both reading and writing of the bound values. The
component can modify the input, and the parent component will be notified of the
changes.
Standard Inputs: These are read-only for the child component. The parent passes
a value, but the child cannot directly modify it.
```



Same behaviour without checkChange method



Signal Queries —----------

In Angular, signal-based queries offer a way to retrieve and interact with elements, components, or directives in a component's template (view) or projected content (content).

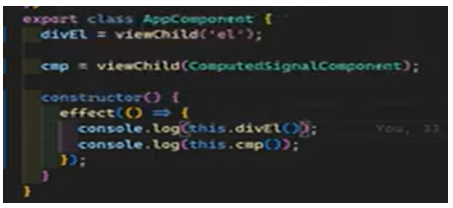
The main difference between signal-based and traditional decorator-based queries is that signal-based queries return reactive Signal objects, enabling you to integrate them with Angular's reactivity system.

Angular provides two main types of queries:

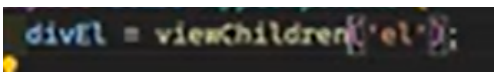
View Queries: These retrieve results from the component's own template.
Content Queries: These retrieve results from content projected into the component.

View queries let you retrieve elements or components that are part of the component's template.

viewChild: Querying a Single Element or Component
The viewChild function retrieves a single result, either by a string reference (template variable) or by the component/directive type.



For retrieve multiple elements



Reading as element

computd signals

string reference (template variable) or by the component/directive type.

Traditional Angular queries use decorators like @ViewChild and @ViewChildren. The signal-based approach replaces decorators with functional API calls like viewChild and viewChildren, returning reactive Signal objects.

Key Differences:

Reactive: Signal-based queries automatically update when the view changes, while decorator-based queries require manual handling to react to changes.
Functional API: Signal-based queries use functions instead of decorators, which provides a more functional programming style.

Content Quries —----------

Content queries in Angular allow a component to access child elements that are projected into its content from a parent component, rather than those defined directly in its own template. These queries are useful when you want to interact with or manipulate projected content in a dynamic way.

```
export class ComputedSignalComponent {
  headerEl = contentChild<ElementRef>('h1', { descendants: true });
  widget = contentChild(WidgetComponent, { read: ElementRef });

  constructor() {
    effect(() => {
      console.log(this.headerEl());
      console.log(this.widget());
    });
  }
}
```

Child queries in Angular allow you to access elements or child components from the template of your component. These queries can be for:

Single elements (e.g., viewChild or contentChild).

Multiple elements (e.g., viewChildren or contentChildren).

Before Angular 18, we used decorators like @ViewChild, @ContentChild, etc., to query child elements. With Angular 18, the new signal-based queries approach allows you to access these elements via signals, making it more reactive.

How Signal-Based Queries Work
A signal-based query returns the results as signals, allowing you to track and respond to changes more naturally. You can use these queries with:

viewChild: For querying a single element from the view.
viewChildren: For querying multiple elements from the view.
contentChild: For querying a single element projected into a component's <ng-content>.
contentChildren: For querying multiple elements projected into <ng-content>.

Required Queries
Sometimes, you want to enforce that a query must always find at least one result (e.g., a viewChild query should never be undefined). You can use .required() to make sure a query is mandatory. If no result is found, Angular will throw a runtime error.

Query Availability Timing
When Angular builds a component, there is a period where the signal query has been created but the template has not been fully processed. During this time, queries will return:

undefined for a single result query (like viewChild).
An empty array for multiple results queries (like viewChildren).

Query results are resolved lazily, meaning Angular will only process and update the query when you explicitly read the signal.

Query Declarations and Usage
You can only declare signal-based queries in class properties. These functions should not be called in other parts of the component (e.g., in a constructor).

```html
<div #requiredEl>Iam required</div>
<div *ngIf="showElement">I am optional</div>
<button (click)="toggleElment()">Toggle optional element</button>
```

```ts
@Component({
  selector: 'app-computed-signal',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './computed-signal.component.html',
  styleUrl: './computed-signal.component.css',
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ComputedSignalComponent {
  requiredEl = viewChild.required('requiredEl');
  optionalEl = viewChild('optionalEl');
  showElement = signal(false);

  ngAfterViewInit() {
    console.log(this.requiredEl());
    console.log(this.optionalEl());
  }

  toggleElement() {
    this.showElement.update((value) => !value);
  }
}
```

Reactivity: Query results are exposed as signals, which can be used in computed or effect signals.
Lazy Resolution: Results are resolved on-demand when you access the signal, making the process more efficient.
Stricter Type Safety: By marking queries as required, you eliminate undefined from the type signature when a result is mandatory.

Routing

Route Params & Component Binding

```ts
export const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: 'groceries', component: GroceryListComponent },
  { path: 'groceries/edit/:id', component: EditGroceryItemComponent },
];
```

```ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes, withComponentInputBinding()),
  ],
};
```

```ts
export class EditGroceryItemComponent {
  grocery$!: Observable<any>;
  constructor(private groceryService: GroceryService) {}
  @Input() set id(groceryId: string) {
    //groceryDetails of that id
    this.grocery$ = this.groceryService.getGroceryById(+groceryId);
  }
}
```

```ts
@Injectable({
  providedIn: 'root',
})
export class GroceryService {
  private groceries = [
    { id: 1, name: 'Apples' },
    { id: 2, name: 'bananas' },
    { id: 3, name: 'Carrots' },
  ];

  getGroceryById(id: number) {
    const grocery = this.groceries.find((item) => item.id === id);
    return of(grocery);
  }
}
```

Parent Route Data Inheritance —----------

```ts
export const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  {
    path: 'groceries/:categoryId',
    component: GroceryListComponent,
    children: [
      { path: 'details/:groceryId', component: EditGroceryItemComponent },
    ],
  },
];
```

```ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(
      routes,
      withComponentInputBinding(),
      withRouterConfig({ paramsInheritanceStrategy: 'always' })
    ),
  ],
};
```

```ts
export class EditGroceryItemComponent {
  categoryId = input('categoryId');
  groceryId = input('groceryId');

  grocerySignal = toSignal(
    toObservable(this.groceryId).pipe(
      switchMap((id) => {
        return this.groceryService.getGroceryById(+id);
      })
    )
  );

  groceryService = inject(GroceryService);

  ngOnInit() {
    console.log('child component');
    console.log(this.categoryId());
    console.log(this.groceryId());
  }
}
```

Wildcard routing and path redirects —----------

```
export const routes: Routes = [
  {
    path: 'old-user-page',
    redirectTo: ({ queryParams }) => {
      const errorHandler = inject(ErrorHandler);
      const userIdParam = queryParams['userId'];
      if (userIdParam) {
        return `/user/${userIdParam}`;
      } else {
        errorHandler.handleError(
          new Error('Attempted navigation to user page without user ID.')
        );
        return '/not-found';
      }      You, 5 seconds ago • Uncommitted changes
    },
  },
  {
    path: 'groceries/:categoryId',
```

```
export const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '', redirectTo: '/first-component', pathMatch: 'full' },
  { path: 'user/:userId', component: UserComponent },

  {
    path: 'groceries/:categoryId',
    component: GroceryListComponent,
    children: [
      { path: 'details/:groceryId', component: EditGroceryItemComponent },
    ],
  },
  { path: '**', component: PageNotFoundComponent },
];
```

Nest Routes with Standalone Components —----------

```
Go to component | You, 1 second ago | 1 author (You)
<h2>First Component</h2>

<nav>
  <ul>
    <li>
      <a routerLink="child-a">Child A</a>
    </li>
    <li>
      <a routerLink="child-b">Child B</a>
    </li>
  </ul>
</nav>
<router-outlet></router-outlet>       You, 1
```

```
export const routes: Routes = [
  {
    path: 'first-component',       You, 2 seconds ago • U
    component: FirstComponent,
    children: [
      { path: 'child-a', component: ChildAComponent },
      { path: 'child-b', component: ChildBComponent },
    ],
  },
```

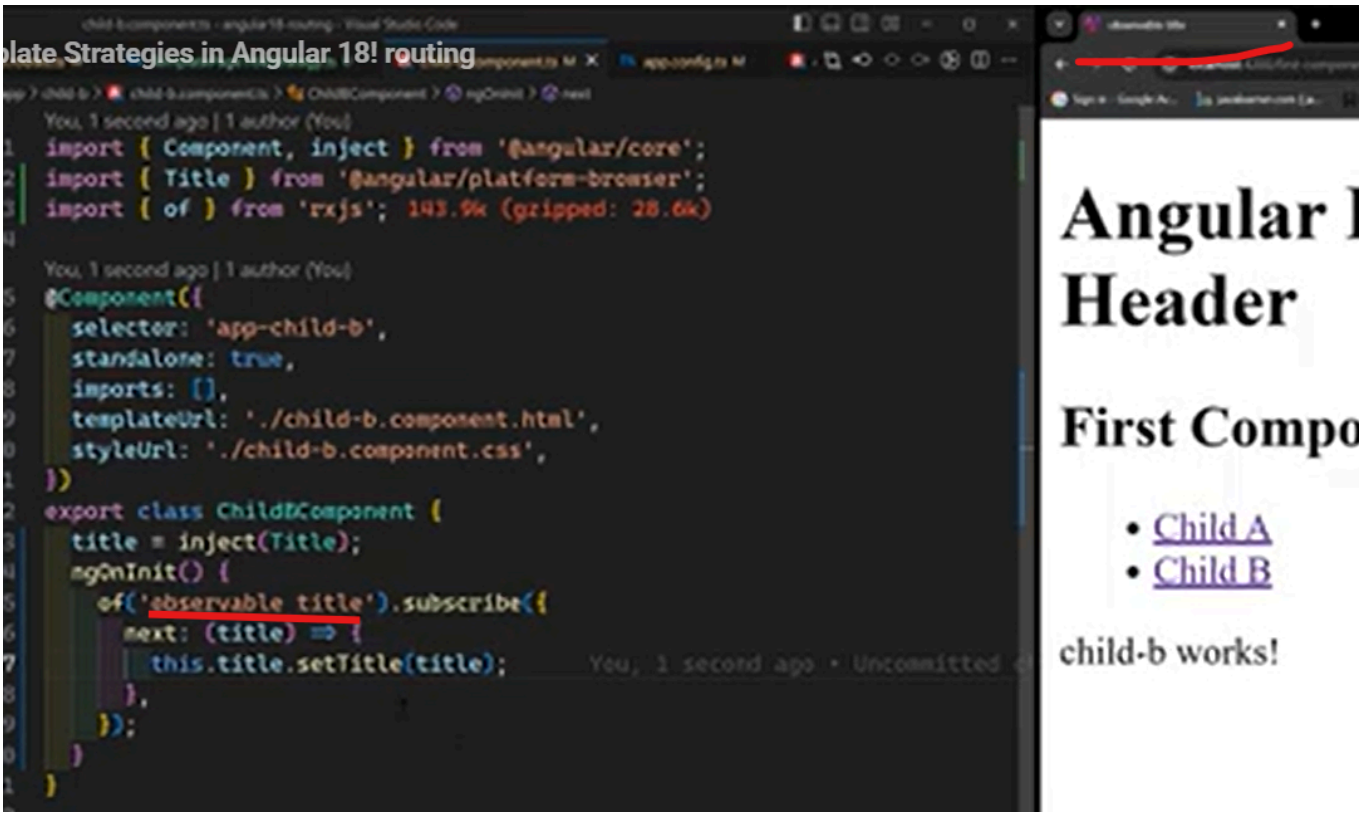
Dynamic Page Titles with Template Strategies —----------

```ts
const resolvesChildATitle = () => {
  return Promise.resolve('Child A - Dynamic Title');
};

export const routes: Routes = [
  {
    path: 'first-component',
    component: FirstComponent,
    title: 'First Component',
    children: [
      {
        path: 'child-a',
        component: ChildAComponent,
        title: resolvesChildATitle,
      },
      { path: 'child-b', component: ChildBComponent },
    ],
  },
```

We can defined and use custom strategy for titles of the pages

```ts
import { Injectable } from '@angular/core';
import { Title } from '@angular/platform-browser';
import { RouterStateSnapshot, TitleStrategy } from '@angular/router';

@Injectable({
  providedIn: 'root',
})
export class TemplatePageTitleStrategy extends TitleStrategy {
  constructor(private title: Title) {
    super();
  }

  override updateTitle(routerState: RouterStateSnapshot): void {
    const title = this.buildTitle(routerState);
    if (title !== undefined) {
      this.title.setTitle(`My Application | ${title}`);
    }
  }
}
```

```ts
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/
core';
import { provideRouter, TitleStrategy } from '@angular/router';

import { routes } from './app.routes';
import { TemplatePageTitleStrategy } from './services/
TemplatePageTitleStrategy';
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    { provide: TitleStrategy, useClass: TemplatePageTitleStrategy },
  ],
};
```



Relative Routes and Navigating —----------

```
<a [routerLink]="['items']">Go to Items</a>
```

Routerlink always try to navigate as relative router. If the current location is ‘home’, this will navigate into “home/items”

Absolute routing

```
constructor(private router: Router) {}
goToItem() {
  this.router.navigate(['/home/items']);
}
```

```
this.router.navigate(['hero', hero.id]);
```

Relative routing

```
constructor(private router: Router, private route: ActivatedRoute) {}
goToItem() {
  this.router.navigate(['items'], { relativeTo: this.route });
}
```

Sending and Capturing the Dynamic Route Params Programmatically —----------

```
export class HeroDetailComponent {
  heroService = inject(HeroService);

  route = inject(ActivatedRoute);
  ngOnInit() {
    console.log(this.route.snapshot);
  }
}
```

all informations

```
export class HeroDetailComponent {
  heroService = inject(HeroService);
  hero$!: Observable<Hero | undefined>;

  route = inject(ActivatedRoute);
  ngOnInit() {
    const heroId = this.route.snapshot.params['id'];
    this.hero$ = this.heroService.getHero(heroId);
  }
}
```

but this will work once since ngOnInit only executes at the beginning, so we need to observe the changes of params

```
@Component({
  selector: 'app-hero-detail',
  standalone: true,
  imports: [CommonModule, RouterLink],
  templateUrl: './hero-detail.component.html',
  styleUrl: './hero-detail.component.css',
})
export class HeroDetailComponent {
  heroService = inject(HeroService);
  hero$!: Observable<Hero | undefined>;

  route = inject(ActivatedRoute);
  ngOnInit() {
    this.route.params.subscribe((data) => {
      this.hero$ = this.heroService.getHero(data['id']);
    });
  }
}
```

```
  heroService = inject(HeroService);
  hero$!: Observable<Hero | undefined>;

  route = inject(ActivatedRoute);
  ngOnInit() {
    this.hero$ = this.route.params.pipe(
      switchMap((data) => {
        return this.heroService.getHero(data['id']);
      })
    );
  }
}
```

```
export class HeroDetailComponent {
  heroService = inject(HeroService);
  route = inject(ActivatedRoute);
  hero$ = this.route.params.pipe(
    switchMap((data) => {
      return this.heroService.getHero(data['id']);
    })
  );
}
```

Query Parameters & Fragments —----------

```
goToHero(hero: Hero) {
  this.router.navigate(['herodetail'], { queryParams: { id:
  hero.id } });
}
```

```
heroService = inject(HeroService);
route = inject(ActivatedRoute);
hero$ = this.route.queryParams.pipe(
  switchMap((data) => {
    return this.heroService.getHero(data['id']);
  })
);
}
```

```
<div *ngIf="hero$ | async as hero">
  <div>
    <h2>{{ hero.name }}</h2>
  </div>

  <a [routerLink]="['/herodetail']" [queryParams]="{ id: 2 }">Go
  to Batman</a>
</div>
```

Fragments

```
@Component({
  selector: 'app-hero-list',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './hero-list.component.html',
  styleUrl: './hero-list.component.css',
})
export class HeroListComponent {
  heroService = inject(HeroService);
  router = inject(Router);
  heroes$ = this.heroService.getHeroes();

  goToHero(hero: Hero) {
    this.router.navigate(['herodetail'], {
      queryParams: { id: hero.id },
      fragment: 'leelawebdev',
    });
  }
}
```

```
<div *ngIf="hero$ | async as hero">
  <div>
    <h2>{{ hero.name }}</h2>
  </div>

  <a
    [routerLink]="['/herodetail']"
    [queryParams]="{ id: 2 }"
    fragment="leelawebdevjkdhsa"
    >Go to Batman</a
  >
</div>
```

Lazy loading of standalone components —----------

Lazy loading is a technique that delays the loading of a module or component until it is required. This improves initial load time by not loading all features of an application at once.

Standalone components in Angular 18 can be loaded lazily just like modules were loaded in the past. With the introduction of standalone components, Angular now supports lazy loading individual components, which simplifies the architecture of applications.

Route Guards

Route guards allow you to control access to different routes in your application based on certain conditions, such as authentication. They are used to prevent users from accessing specific parts of the application if they do not meet the criteria (e.g., not logged in or lacking permissions).

canActivate: Prevents navigation to a route unless the condition is met.
canActivateChild: Guards child routes.
canDeactivate: Prevents a route from being navigated away from.
canMatch: Controls whether a route can be matched based on conditions.
resolve: Preloads data before the route is activated.
canLoad: Prevents the entire lazy-loaded module from loading.

CanActivate —----------

Authentication: Prevent users from accessing routes (e.g., dashboard, admin panel) without being logged in.
Authorization: Restrict access based on roles (e.g., only admins can access certain pages).
Data Validity: Ensure that specific data is loaded or valid before accessing a route (e.g., a settings page that requires fetching user data first).

CanActivateChild —----------

In Angular 18, the CanActivateChild guard is used to determine if a route's child routes can be activated. It's similar to CanActivate, but it specifically guards child routes of a parent route. You would typically use this guard when you want to apply a security check or permissions validation at a parent route level to control access to all its child routes.

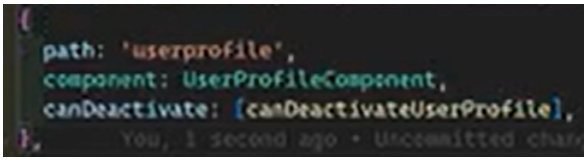
all for child routes

specify once for all child routes

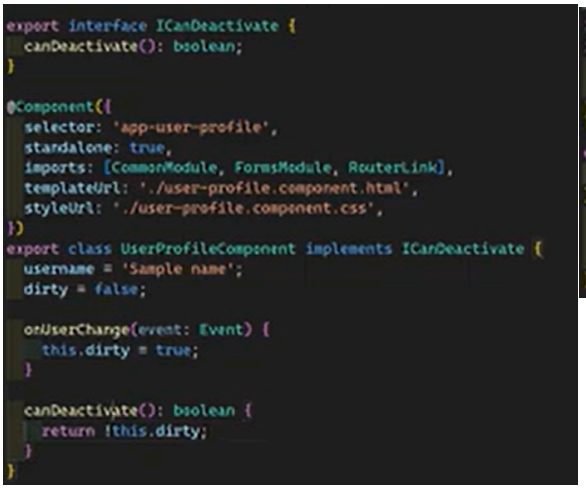
CanDeactivate Guard Save Users from Unsaved Data Loss —----------

The CanDeactivate route guard in Angular helps prevent users from accidentally leaving a component with unsaved changes.

When navigating away from a route, this guard can prompt users to confirm their decision if certain conditions are met, such as unsaved form data.



Reusable generic solution for all components



```ts
export interface ICanDeactivate {
  canDeactivate(): boolean;
}

@Component({
  selector: 'app-user-profile',
  standalone: true,
  imports: [CommonModule, FormsModule, RouterLink],
  templateUrl: './user-profile.component.html',
  styleUrl: './user-profile.component.css',
})
export class UserProfileComponent implements ICanDeactivate {
  username = 'Sample name';
  dirty = false;

  onUserChange(event: Event) {
    this.dirty = true;
  }

  canDeactivate(): boolean {
    return !this.dirty;
  }
}
```

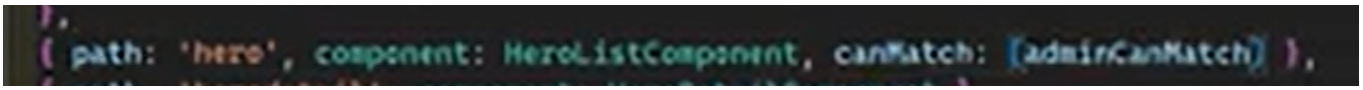
```ts
import { CanDeactivateFn } from '@angular/router';
import {
  ICanDeactivate,
  UserProfileComponent,
} from '../user-profile/user-profile.component';

export const canDeactivateUserProfile: CanDeactivateFn<ICanDeactivate> = (
  component
) => {
  return !component.dirty
    ? true
    : confirm('Changes will be lost. ARe you sure');
};
```

CanMatch —----------

The CanMatch route guard in Angular 18 is used to control whether a route should be loaded based on certain conditions.

Unlike CanActivate and CanActivateChild, which control access to routes, CanMatch specifically determines if a route should be matched (made available) in the router. This guard is particularly useful for scenarios where you want to dynamically include or exclude routes based on criteria such as user roles, feature flags, or app state.

```ts
import { inject } from '@angular/core';
import { CanMatchFn } from '@angular/router';
import { AuthService } from './auth.service';

export const adminCanMatch: CanMatchFn = () => {
  const authService = inject(AuthService);

  const userRole = authService.getUserRole();
  return userRole === 'admin';
};
```



```ts
{ path: 'hero', component: HeroListComponent, canMatch: [adminCanMatch] },
```

Determines if a route should match in the router, essentially controlling route availability

Controls if a route can be activated (i.e., whether the component can be loaded and viewed)

Prevents routes from matching in the router altogether if conditions aren't met

Route still exists in the route configuration, but users are redirected if they can't activate

Ideal for dynamic routing needs, like feature flags or role-based display of routes

Typically used for authentication, authorization, or other access checks

Only runs when Angular evaluates available routes (i.e., the first match attempt)

Runs when the route is being activated, such as after clicking a link or typing the URL

Routes with canMatch failing conditions won't show up as matching links

canActivate-guarded routes still show as accessible links but may redirect if blocked

Conditional visibility of routes (e.g., platform-based restrictions, role-based menus)

Access control to routes after initial route matching (e.g., restricting access to logged-in users)

canMatch: Use this guard when you need to conditionally make a route available in the router based on dynamic factors like user roles, feature flags, or device types. It's helpful for scenarios where routes should not even appear as accessible if they do not meet certain criteria.

canActivate: Use this guard to control whether a route can be entered after it has matched. It's ideal for handling access control based on authentication, authorization, or any checks that need to happen before a user enters the route's component.

canActivate: Use this guard to control whether a route can be entered after it has matched. It's ideal for handling access control based on authentication, authorization, or any checks that need to happen before a user enters the route's component.

canMatch:

Only display certain routes if a feature flag is enabled.
Show admin routes only if the user role matches "Admin".
Display mobile-only routes for mobile devices.
canActivate:

Block access to certain routes if a user is not logged in.
Restrict certain routes to users with specific permissions.
Show error pages or redirect users if they don't meet criteria to view the route.

Resolve —----------

The Resolve route guard in Angular is designed to fetch data before a route is activated.

This guard is helpful for preloading data and ensuring that a route only renders once required data is available.

In Angular 18, we can set up a Resolve guard in standalone components to streamline data loading, especially when dealing with asynchronous data sources like HTTP APIs.

Let's go through a detailed example to set up a Resolve guard in Angular 18 with standalone components.

or

Pre-fetching Data: Ensure that essential data (like posts or user profiles) is loaded before navigating to a page.

Reducing Loading Spinners: When data is available instantly, users don't experience blank pages or loading indicators.
Improving SEO and Accessibility: By loading content before page render, web crawlers and screen readers can access content immediately.

Path vs Hash —----------

Angular offers two main strategies for handling browser URLs:

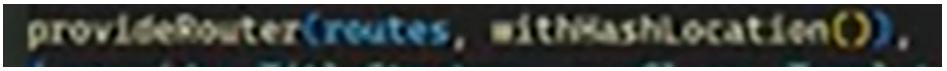
PathLocationStrategy (HTML5 pushState): This is the default strategy and uses clean URLs (e.g., localhost:4200/crisis-center). It leverages the history.pushState method to update the browser URL without reloading the page.

HashLocationStrategy: This strategy appends a # in URLs (e.g., localhost:4200/#/crisis-center). It's useful for older browsers where changing the path without a full reload is unsupported.

2. Why Choose PathLocationStrategy (HTML5 pushState) by Default?
SEO-Friendly: Clean URLs (without #) are better recognized by search engines.

Server-Side Rendering: Path-based URLs allow easier server-side rendering, which can improve initial page load performance.

User Experience: URLs look cleaner and are easier to share.



Use PathLocationStrategy when possible, as it supports SEO and server-side rendering.

Choose HashLocationStrategy if hosting constraints prevent server configuration for handling Angular routes.